CS3192 Section 4
Large Games

Andrea Schalk

Department of Computer Science, University of Manchester

Large games

Section 4 covers how computer programs for games such as Chess, Go, Othello, Checkers and similar games work.

Large games

Section 4 covers how computer programs for games such as Chess, Go, Othello, Checkers and similar games work.

These games have very large game trees—typically far too large to be held in memory entirely, and certainly too large to try to find all strategies.

Large games

Section 4 covers how computer programs for games such as Chess, Go, Othello, Checkers and similar games work.

These games have very large game trees—typically far too large to be held in memory entirely, and certainly too large to try to find all strategies.

In fact, even the methods in the previous section will not work on such games—the game trees are so large that carrying out alpha-beta search would take far too long to return a value and thus a good move.

Large games

Section 4 covers how computer programs for games such as Chess, Go, Othello, Checkers and similar games work.

These games have very large game trees—typically far too large to be held in memory entirely, and certainly too large to try to find all strategies.

In fact, even the methods in the previous section will not work on such games—the game trees are so large that carrying out alpha-beta search would take far too long to return a value and thus a good move.

There are three problems which have to be solved to write such a program which we will discuss in some detail. Finally we will have a look at how Chess-playing programs developed, since Chess is the game for which the most effort has been made when it comes to writing programs.

The three problems

In order to write a game-playing program, the following problems have to be solved.

The three problems

In order to write a game-playing program, the following problems have to be solved.

Board representation and move generation. Clearly we have to think about how the board (and the pieces) are represented internally, and how the moves are to be generated. Typically, once this has been solved it can be left alone.

The three problems

In order to write a game-playing program, the following problems have to be solved.

Board representation and move generation. Clearly we have to think about how the board (and the pieces) are represented internally, and how the moves are to be generated. Typically, once this has been solved it can be left alone.

Alpha-beta search. Despite the fact that we cannot hope to employ the minimax algorithm with alpha-beta pruning, this technique still plays a vital role in game-playing programs. There are some variants that might be implemented, and typically some effort is spent on cataloguing search results.

The three problems

In order to write a game-playing program, the following problems have to be solved.

Board representation and move generation. Clearly we have to think about how the board (and the pieces) are represented internally, and how the moves are to be generated. Typically, once this has been solved it can be left alone.

Alpha-beta search. Despite the fact that we cannot hope to employ the minimax algorithm with alpha-beta pruning, this technique still plays a vital role in game-playing programs. There are some variants that might be implemented, and typically some effort is spent on cataloguing search results.

Evaluation function. Since alpha-beta search cannot be carried out until a leaf is reached, the search stops instead at a pre-defined depth. To obtain a value for a position at this depth, a function has to be created which assigns one based entirely on the state of the board at the time. This is known as the 'evaluation function'.

The three problems

In order to write a game-playing program, the following problems have to be solved.

Board representation and move generation. Clearly we have to think about how the board (and the pieces) are represented internally, and how the moves are to be generated. Typically, once this has been solved it can be left alone.

Alpha-beta search. Despite the fact that we cannot hope to employ the minimax algorithm with alpha-beta pruning, this technique still plays a vital role in game-playing programs. There are some variants that might be implemented, and typically some effort is spent on cataloguing search results.

Evaluation function. Since alpha-beta search cannot be carried out until a leaf is reached, the search stops instead at a pre-defined depth. To obtain a value for a position at this depth, a function has to be created which assigns one based entirely on the state of the board at the time. This is known as the 'evaluation function'.

The faster the program, the higher the depth to which it can carry out alpha-beta search (before it has to 'guess' a value for a position), and the better it will play. Hence speed is of the essence when writing such programs, and is a concern for all the components mentioned above.

Task 1

Representing the board and related issues

Representing the board–array

In order to illustrate our thoughts, we often use Chess as an example. However, there's no need to be familiar with the game beyond the rudiments.

Representing the board–array

Obvious representation of a Chess board: 8×8 array.

Each field holds information about the piece that occupies the corresponding field on the board (if any).

Representing the board–array

Obvious representation of a Chess board: 8×8 array.
Each field holds information about the piece that occupies the corresponding field on the board (if any).

To generate moves: Pick piece, generate possible target fields, then:

- check target field not occupied by own piece;
- if piece is a rook, bishop, pawn or queen, check whether the way to target is empty;
- if piece is a king check that target position cannot be reached by an enemy piece in one step.

Representing the board–array

Obvious representation of a Chess board: 8×8 array.
Each field holds information about the piece that occupies the corresponding field on the board (if any).

To generate moves: Pick piece, generate possible target fields, then:

- check target field not occupied by own piece;
- if piece is a rook, bishop, pawn or queen, check whether the way to target is empty;
- if piece is a king check that target position cannot be reached by an enemy piece in one step.

Need:

- loop over all fields (to pick piece);
- loop over all possible target positions;
- loop to check for obstructions along the way.

Complicated, not fast.

Board representation–$0x88$

Faster: Assign a number to each square on the board given by one byte, four high bits: row; four low bits: column.

Board representation–$0x88$

Faster: Assign a number to each square on the board given by one byte, four high bits: row; four low bits: column.

		a	b	c	d	e	f	g	h	
		0000	0001	0010	0011	0100	0101	0110	0111	low bits
8	0111	112	113	114	115	116	117	118	119	
7	0110	96	97	98	99	100	101	102	103	
6	0101	80	81	82	83	84	85	86	87	
5	0100	64	65	66	67	68	69	70	71	
4	0011	48	49	50	51	52	53	54	55	
3	0010	32	33	34	35	36	37	38	39	
2	0001	16	17	18	19	20	21	22	23	
1	0000	0	1	2	3	4	5	6	7	
	high bits									

Board representation–$0x88$

		a	b	c	d	e	f	g	h	
		0000	0001	0010	0011	0100	0101	0110	0111	low bits
8	0111	112	113	114	115	116	117	118	119	
7	0110	96	97	98	99	100	101	102	103	
6	0101	80	81	82	83	84	85	86	87	
5	0100	64	65	66	67	68	69	70	71	
4	0011	48	49	50	51	52	53	54	55	
3	0010	32	33	34	35	36	37	38	39	
2	0001	16	17	18	19	20	21	22	23	
1	0000	0	1	2	3	4	5	6	7	
	high bits									

To move one field to the left or right, just subtract or add one.

Board representation–$0x88$

		a	b	c	d	e	f	g	h	
		0000	0001	0010	0011	0100	0101	0110	0111	low bits
8	0111	112	113	114	115	116	117	118	119	
7	0110	96	97	98	99	100	101	102	103	
6	0101	80	81	82	83	84	85	86	87	
5	0100	64	65	66	67	68	69	70	71	
4	0011	48	49	50	51	52	53	54	55	
3	0010	32	33	34	35	36	37	38	39	
2	0001	16	17	18	19	20	21	22	23	
1	0000	0	1	2	3	4	5	6	7	
	high bits									

To move up a row, add 16, to move down a row, subtract 16.

Board representation–$0x88$

		a	b	c	d	e	f	g	h	
		0000	0001	0010	0011	0100	0101	0110	0111	low bits
8	0111	112	113	114	115	116	117	118	119	
7	0110	96	97	98	99	100	101	102	103	
6	0101	80	81	82	83	84	85	86	87	
5	0100	64	65	66	67	68	69	70	71	
4	0011	48	49	50	51	52	53	54	55	
3	0010	32	33	34	35	36	37	38	39	
2	0001	16	17	18	19	20	21	22	23	
1	0000	0	1	2	3	4	5	6	7	
	high bits									

Board: represented as an array with 128 entries, only 64 of which correspond to actual fields.

Board representation–$0x88$

		a	b	c	d	e	f	g	h	
		0000	0001	0010	0011	0100	0101	0110	0111	low bits
8	0111	112	113	114	115	116	117	118	119	
7	0110	96	97	98	99	100	101	102	103	
6	0101	80	81	82	83	84	85	86	87	
5	0100	64	65	66	67	68	69	70	71	
4	0011	48	49	50	51	52	53	54	55	
3	0010	32	33	34	35	36	37	38	39	
2	0001	16	17	18	19	20	21	22	23	
1	0000	0	1	2	3	4	5	6	7	
	high bits									

Board: represented as an array with 128 entries, only 64 of which correspond to actual fields.

This is much faster than the first version. To check whether a number i is a valid position on the board, check whether it satisfies $i \& 0x88 == 0$ (&: bitwise).

Board representation–bitboards

Idea: for each colour and piece, represent where such a piece can be found using a ‘bitboard’.

Board representation–bitboards

Idea: for each colour and piece, represent where such a piece can be found using a 'bitboard'.

Board representation–bitboards

Idea: for each colour and piece, represent where such a piece can be found using a 'bitboard'.

The white pawns:

Board representation–bitboards

Idea: for each colour and piece, represent where such a piece can be found using a 'bitboard'.

The white pawns:

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

Board representation–bitboards

Idea: for each colour and piece, represent where such a piece can be found using a ‘bitboard’.

The white pawns:

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

Need: one 64-bit word for each piece. Operations: bit-wise—this is really fast!

Board representation–bitboards

Idea: for each colour and piece, represent where such a piece can be found using a ‘bitboard’.

The white pawns:

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

Need: one 64-bit word for each piece. Operations: bit-wise—this is really fast!

Example: bitboard for all black pieces: bit-wise ‘or’ of all bitboards for black pieces.

Board representation–bitboards

Idea: for each colour and piece, represent where such a piece can be found using a ‘bitboard’.

The white pawns:

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

Need: one 64-bit word for each piece. Operations: bit-wise—this is really fast!

Move of a piece by a row: shift the bitboard by 8.

Board representation–bitboards

Idea: for each colour and piece, represent where such a piece can be found using a 'bitboard'.

The white pawns:

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

Need: one 64-bit word for each piece. Operations: bit-wise—this is really fast!

Empty fields: bitboard for all pieces negated.

Board representation–bitboards

Idea: for each colour and piece, represent where such a piece can be found using a 'bitboard'.

The white pawns:

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

Need: one 64-bit word for each piece. Operations: bit-wise—this is really fast!

All legal moves of pawns by one field can be stored in a bitboard (similarly for all legal moves of pawns by two fields). Constant bitboards can be prepared at compile time to be available in a library.

Board representation–bitboards

Idea: for each colour and piece, represent where such a piece can be found using a ‘bitboard’.

The white pawns:

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

Need: one 64-bit word for each piece. Operations: bit-wise—this is really fast!

Pawn captures: shifting the bitboard by 7 or 9 and bit-wise ‘and’ it with the bitboard for pieces of the opposite colour.

Board representation–bitboards

Idea: for each colour and piece, represent where such a piece can be found using a ‘bitboard’.

The white pawns:

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

Need: one 64-bit word for each piece. Operations: bit-wise—this is really fast!

Only disadvantage: the code becomes more complicated; turning a bitboard of possible moves into a list of possible moves, for example.

Board representation–bitboards

Idea: for each colour and piece, represent where such a piece can be found using a ‘bitboard’.

The white pawns:

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

Need: one 64-bit word for each piece. Operations: bit-wise—this is really fast!

Only disadvantage: the code becomes more complicated; turning a bitboard of possible moves into a list of possible moves, for example.

Advantages: fast; bitboards required more than once only have to be computed once; several moves can be generated at the same time.

Beyond board representation

Typically it is not sufficient merely to represent the pieces on the board.

Beyond board representation

Typically it is not sufficient merely to represent the pieces on the board.

In Chess, for example, we have to know whose turn it is, whether a player can still castle, and whether a capture *en passant* is possible.

Beyond board representation

Typically it is not sufficient merely to represent the pieces on the board.

In Chess, for example, we have to know whose turn it is, whether a player can still castle, and whether a capture *en passant* is possible. Worse, there are rules about repeating previous positions in Chess (which will lead to a draw), so the program has to have a way of remembering those!

Beyond board representation

Typically it is not sufficient merely to represent the pieces on the board.

In Chess, for example, we have to know whose turn it is, whether a player can still castle, and whether a capture *en passant* is possible. Worse, there are rules about repeating previous positions in Chess (which will lead to a draw), so the program has to have a way of remembering those!

Chess programs typically use a large hash table to keep track of positions that have occurred in play.

Hash tables

Hash tables are not merely useful when it comes to determining whether a position is being repeated.

Hash tables

Hash tables are not merely useful when it comes to determining whether a position is being repeated.

When we carry out an alpha-beta search from a given position, we will search to a given depth. When it is our turn again, we will repeat that from the now current position—but we have searched this position before, only to a depth of two less than we now require!

Hash tables

Hash tables are not merely useful when it comes to determining whether a position is being repeated.

When we carry out an alpha-beta search from a given position, we will search to a given depth. When it is our turn again, we will repeat that from the now current position—but we have searched this position before, only to a depth of two less than we now require!

Hash tables are used in Chess programs to keep track of which positions have been searched before, what value has been found for them, and to which depth they were searched at the time.

Hash tables

Hash tables are not merely useful when it comes to determining whether a position is being repeated.

When we carry out an alpha-beta search from a given position, we will search to a given depth. When it is our turn again, we will repeat that from the now current position—but we have searched this position before, only to a depth of two less than we now require!

Hash tables are used in Chess programs to keep track of which positions have been searched before, what value has been found for them, and to which depth they were searched at the time.

A hash function frequently used consists of assigning to each pair, consisting of a piece and a field on the board, a large random number. The idea is that this number encodes the fact that the corresponding piece occupies the corresponding field. Then one sums up the appropriate numbers for the given position to obtain the hash key. A checksum process can be applied to make sure later that 'the right' position is looked up.

Doing and undoing moves

In the course of a game, our program will not only have to make moves, but it will also have to be able to undo them.

Doing and undoing moves

In the course of a game, our program will not only have to make moves, but it will also have to be able to undo them.

This is not in order to allow the opponent to reconsider his position, but because in order to conduct an alpha-beta search the program has to make moves to find out which positions are reachable.

Doing and undoing moves

In the course of a game, our program will not only have to make moves, but it will also have to be able to undo them.

This is not in order to allow the opponent to reconsider his position, but because in order to conduct an alpha-beta search the program has to make moves to find out which positions are reachable.

It then has to undo those moves to try others, and ultimately to get back to the position where it started the search.

Doing and undoing moves

In the course of a game, our program will not only have to make moves, but it will also have to be able to undo them.

This is not in order to allow the opponent to reconsider his position, but because in order to conduct an alpha-beta search the program has to make moves to find out which positions are reachable.

It then has to undo those moves to try others, and ultimately to get back to the position where it started the search.

This is best done by keeping a stack of moves with sufficient information to undo them. This is typically much cheaper than keeping a list of positions through which one has gone.

Task 2

Evaluation function

The task

In order to implement a board representation and a move-generator, as well as alpha-beta search the programmer does not have to know much about the game in question—it suffices if he is familiar with the rules.

The task

In order to implement a board representation and a move-generator, as well as alpha-beta search the programmer does not have to know much about the game in question—it suffices if he is familiar with the rules.

This is entirely different for the evaluation function. In order to turn a game position into a meaningful number, the programmer must have considerable knowledge about the game.

The task

In order to implement a board representation and a move-generator, as well as alpha-beta search the programmer does not have to know much about the game in question—it suffices if he is familiar with the rules.

This is entirely different for the evaluation function. In order to turn a game position into a meaningful number, the programmer must have considerable knowledge about the game.

If the values provided do not judge the given position accurately then the program can't possibly play well—until very close to the end, they are all the program has to judge which move it should make.

The task

In order to implement a board representation and a move-generator, as well as alpha-beta search the programmer does not have to know much about the game in question—it suffices if he is familiar with the rules.

This is entirely different for the evaluation function. In order to turn a game position into a meaningful number, the programmer must have considerable knowledge about the game.

If the values provided do not judge the given position accurately then the program can't possibly play well—until very close to the end, they are all the program has to judge which move it should make.

There is no such thing as 'the right' evaluation function. A big part of writing a game-playing program is to watch it play and fine-tune the evaluation function to improve it.

The task

In order to implement a board representation and a move-generator, as well as alpha-beta search the programmer does not have to know much about the game in question—it suffices if he is familiar with the rules.

This is entirely different for the evaluation function. In order to turn a game position into a meaningful number, the programmer must have considerable knowledge about the game.

If the values provided do not judge the given position accurately then the program can't possibly play well—until very close to the end, they are all the program has to judge which move it should make.

There is no such thing as 'the right' evaluation function. A big part of writing a game-playing program is to watch it play and fine-tune the evaluation function to improve it.

There are no hard and fast rules for what makes a good evaluation function; they are mostly based on heuristics.

Speed

When writing a game-playing program, speed is always an issue. Hence it pays to calculate the desired evaluation function in such a way to make the process as fast as possible.

Speed

When calculating the evaluation function for two successive positions, the value often does not change very much, and, in fact, the actual calculations are very similar.

Speed

When calculating the evaluation function for two successive positions, the value often does not change very much, and, in fact, the actual calculations are very similar.

We can make this work in our favour if we can express the evaluation function in terms of the contributions made by the different pieces on their various fields.

Speed

When calculating the evaluation function for two successive positions, the value often does not change very much, and, in fact, the actual calculations are very similar.

We can make this work in our favour if we can express the evaluation function in terms of the contributions made by the different pieces on their various fields.

Let p be the current position, and e the evaluation function. Then if

$$e(p) = e_{s_1}(s_1\text{'s place in } p) + \cdots + e_{s_n}(s_n\text{'s place in } p),$$

where $s_1, \ldots, s_n$ are the pieces involved,

Speed

When calculating the evaluation function for two successive positions, the value often does not change very much, and, in fact, the actual calculations are very similar.

We can make this work in our favour if we can express the evaluation function in terms of the contributions made by the different pieces on their various fields.

Let p be the current position, and e the evaluation function. Then if

$$e(p) = e_{s_1}(s_1\text{'s place in } p) + \cdots + e_{s_n}(s_n\text{'s place in } p),$$

where $s_1, \ldots, s_n$ are the pieces involved, the value of a new position resulting from one piece s being moved is

$$score(move) = e_s(s\text{'s new field}) - e_s(s\text{'s old field}).$$

Problems: For many games this kind of evaluation function is not good enough since it does not take the relative position of pieces into account.

Techniques

It may be difficult to design an evaluation function in such a way that it can take immediate future moves into account (captures in Chess, for example).

Techniques

It may be difficult to design an evaluation function in such a way that it can take immediate future moves into account (captures in Chess, for example).

In such a situation it makes sense to use the move-generating facility and look one or two further moves ahead in evaluating a position. That will tell us about immediately possible captures and similar important changes we might expect in the near future.

Techniques

It may be difficult to design an evaluation function in such a way that it can take immediate future moves into account (captures in Chess, for example).

In such a situation it makes sense to use the move-generating facility and look one or two further moves ahead in evaluating a position. That will tell us about immediately possible captures and similar important changes we might expect in the near future.

Typically the evaluation function is split into a number of functions which score one particular aspect of the current position.

Techniques

It may be difficult to design an evaluation function in such a way that it can take immediate future moves into account (captures in Chess, for example).

In such a situation it makes sense to use the move-generating facility and look one or two further moves ahead in evaluating a position. That will tell us about immediately possible captures and similar important changes we might expect in the near future.

Typically the evaluation function is split into a number of functions which score one particular aspect of the current position.

These are then put together using weights, and often the actual fine-tuning consists of playing with the weights.

Techniques

It may be difficult to design an evaluation function in such a way that it can take immediate future moves into account (captures in Chess, for example).

In such a situation it makes sense to use the move-generating facility and look one or two further moves ahead in evaluating a position. That will tell us about immediately possible captures and similar important changes we might expect in the near future.

Typically the evaluation function is split into a number of functions which score one particular aspect of the current position.

These are then put together using weights, and often the actual fine-tuning consists of playing with the weights.

It is important that an evaluation function judge any position from both players' point of view. Having many pieces on the board does not give White any advantage if Black is about to checkmate him!

Relevant constituent parts

Material. The number and kind of pieces on the board. Chess: Each piece has a value; Go: count number of pieces on board, Othello: same.

Relevant constituent parts

Material. The number and kind of pieces on the board. Chess: Each piece has a value; Go: count number of pieces on board, Othello: same.

Not equally useful for all games: Othello: not number of pieces is important, but their locations (corners). Player with fewer pieces might have better position. There are other games where the number of pieces may be irrelevant.

Relevant constituent parts

Material. The number and kind of pieces on the board.

Space. Influence. Often: can divide board into areas of influence; player controls that area, for example in Go.

Relevant constituent parts

Material. The number and kind of pieces on the board.

Space. Influence. Often: can divide board into areas of influence; player controls that area, for example in Go.

Chess: count number of fields threatened by one player; Othello: count number of pieces which cannot be taken by the opponent. Calculate size, possible with weights for very important fields.

Relevant constituent parts

Material. The number and kind of pieces on the board.

Space. Influence.

Mobility. Ability to move. Having many different available moves: advantageous, *e.g.* in Othello. Chess: not clear this is useful.

Relevant constituent parts

Material. The number and kind of pieces on the board.

Space. Influence.

Mobility. Ability to move.

Tempo. Initiative. Go: one player has the initiative, that is, he acts, other player reacts to his moves.

Relevant constituent parts

Material. The number and kind of pieces on the board.

Space. Influence.

Mobility. Ability to move.

Tempo. Initiative. Go: one player has the initiative, that is, he acts, other player reacts to his moves.

Other games: try ‘parity argument’: often find positions where player who moves next wins/loses, can be simple to evaluate (see Nim, Connect-4).

Relevant constituent parts

Material. The number and kind of pieces on the board.

Space. Influence.

Mobility. Ability to move.

Tempo. Initiative.

Threats. Can one of the players capture (or threaten to capture) a piece? Connect-4, Go-Moku: can a player win in the next move? Othello: is a player threatening to take a corner?

Relevant constituent parts

Material. The number and kind of pieces on the board.

Space. Influence.

Mobility. Ability to move.

Tempo. Initiative.

Threats.

Shape. How pieces on the board relate to each other. Chess: line of pawns much stronger than other grouping. Go: shape is ‘territory to be’—stones outline territory which the player can defend when threatened.

Relevant constituent parts

Material. The number and kind of pieces on the board.

Space. Influence.

Mobility. Ability to move.

Tempo. Initiative.

Threats.

Shape. How pieces on the board relate to each other. Chess: line of pawns much stronger than other grouping. Go: shape is ‘territory to be’—stones outline territory which the player can defend when threatened.

Judging shape: often very difficult. Change of shape value: incremental over time, long-term target. Evaluation function partially based on shape: can’t just simply add piece-based functions.

Relevant constituent parts

Material. The number and kind of pieces on the board.

Space. Influence.

Mobility. Ability to move.

Tempo. Initiative.

Threats.

Shape.

Known Patterns. Go: libraries of sequences of moves in small areas (joseki)—preserves balance between players.

Relevant constituent parts

Material. The number and kind of pieces on the board.

Space. Influence.

Mobility. Ability to move.

Tempo. Initiative.

Threats.

Shape.

Known Patterns. Go: libraries of sequences of moves in small areas (joseki)—preserves balance between players.

Chess: bishop capturing a pawn on border is often trapped; Othello: sacrifice one corners in exchange for another. Deciding when a pattern applies is hard!

Fine-tuning

Deducing constraints. Chess: every piece gets a material value. Know: rook more important than pawn, *e.g.*, so value should be according. Can deduce values from experience.

Fine-tuning

Deducing constraints. Chess: every piece gets a material value. Know: rook more important than pawn, *e.g.*, so value should be according. Can deduce values from experience.

Know, *e.g.*: one rook less than two pawns and bishop, or two pawns and knight, but not less than one pawn and bishop/knight.

So: weight of a rook should be below weight of pawns and bishop, but above one pawn and bishop. Get fewer possibilities to try.

Fine-tuning

Deducing constraints.

Hand tweaking. Happens typically in practice. Programmers watch implementation play, judge which parameters to change and how. Perform the change and watch again. Reasonably fast but requires game-specific knowledge.

Fine-tuning

Deducing constraints.

Hand tweaking.

Optimization techniques. Employ general optimization techniques.

Fine-tuning

Deducing constraints.

Hand tweaking.

Optimization techniques. Employ general optimization techniques. Example: 'hill climbing': Make small changes to parameters, keep them if they improve the performance. Need measure to judge performance, for example the percentage of won games against some opponent.

Fine-tuning

Deducing constraints.

Hand tweaking.

Optimization techniques. Employ general optimization techniques. Example: ‘hill climbing’: Make small changes to parameters, keep them if they improve the performance. Need measure to judge performance, for example the percentage of won games against some opponent.

Often slow; risks being stuck when each small change makes performance worse, but big change might bring huge gains (‘local optima’).

Fine-tuning

Deducing constraints.

Hand tweaking.

Optimization techniques. Employ general optimization techniques. Example: ‘hill climbing’: Make small changes to parameters, keep them if they improve the performance. Need measure to judge performance, for example the percentage of won games against some opponent.

Often slow; risks being stuck when each small change makes performance worse, but big change might bring huge gains (‘local optima’).

Can be modified by randomly sticking with some changes which do not improve performance. ‘Randomness’ controlled by some probabilities (start out fairly high, become smaller as a good value is approached). Adjusted method is slower than original, but can get good values.

Fine-tuning

Deducing constraints.

Hand tweaking.

Optimization techniques.

Learning. Early: Thought good Chess programs would mimic human reasoning with machine-based learning most important aspect. That's not what happened! All world-class game-playing programs use other principles foremost.

Fine-tuning

Deducing constraints.

Hand tweaking.

Optimization techniques.

Learning. Early: Thought good Chess programs would mimic human reasoning with machine-based learning most important aspect. That's not what happened! All world-class game-playing programs use other principles foremost.

Examples for learning: genetic algorithms, neural networks. Both: rather slow; main advantage: do not require game-specific knowledge. Reason for slowness: number of test games required is typically very high (commercial game programmers tried about 3000 matches to allow the program to learn—the result was worse than hand tweaking). Further problem: If opponent is too good program loses all the time and never starts learning.

Measuring performance

In order to fine-tune our program we need to be able to measure its performance.

Measuring performance

In order to fine-tune our program we need to be able to measure its performance.

Option: run program on large suit of test positions taken from high-quality human games, see whether program can follow the winner's actions. (Method can be used to check any program, *e.g.* try on 'checkmate in two' kind of positions.)

Measuring performance

In order to fine-tune our program we need to be able to measure its performance.

Option: run program on large suit of test positions taken from high-quality human games, see whether program can follow the winner's actions. (Method can be used to check any program, *e.g.* try on 'checkmate in two' kind of positions.)

Typically combined with playing program against known opponent for many matches, *e.g.* against another program, or version of itself which has different weights, so that the two can be compared to each other.

Measuring performance

In order to fine-tune our program we need to be able to measure its performance.

Option: run program on large suit of test positions taken from high-quality human games, see whether program can follow the winner's actions. (Method can be used to check any program, *e.g.* try on 'checkmate in two' kind of positions.)

Typically combined with playing program against known opponent for many matches, *e.g.* against another program, or version of itself which has different weights, so that the two can be compared to each other.

Problem with playing program against versions of itself: same lines are explored over and over. To avoid this: start the program(s) from positions a few moves into a game.

Task 3

Alpha-beta search

Alpha-beta search

All game-playing programs contain variants of the minimax algorithm with alpha-beta pruning.

Alpha-beta search

All game-playing programs contain variants of the minimax algorithm with alpha-beta pruning.

In this context, the algorithm is usually referred to as alpha-beta search, because it is a search for the best available move.

Alpha-beta search

All game-playing programs contain variants of the minimax algorithm with alpha-beta pruning.

In this context, the algorithm is usually referred to as alpha-beta search, because it is a search for the best available move.

There are some ways of fiddling with this to adjust it to the game in question. The thought is always to make it faster so that it can search deeper.

Iterative Deepening

Most programs give themselves a time limit to come up with a move. This is to avoid running out of time by spending it all early into the game.

Iterative Deepening

Most programs give themselves a time limit to come up with a move. This is to avoid running out of time by spending it all early into the game.

Problem: It might take longer to search to a pre-defined depth than time allows.

Iterative Deepening

Most programs give themselves a time limit to come up with a move. This is to avoid running out of time by spending it all early into the game.

Problem: It might take longer to search to a pre-defined depth than time allows.

Solution: Don’t do a strictly depth-first search, but search to a higher and higher depth.

Iterative Deepening

Most programs give themselves a time limit to come up with a move. This is to avoid running out of time by spending it all early into the game.

Problem: It might take longer to search to a pre-defined depth than time allows.

Solution: Don't do a strictly depth-first search, but search to a higher and higher depth.

This is not as expensive as it sounds: Searching to shallow depths includes few moves and is cheap.

Iterative Deepening

Most programs give themselves a time limit to come up with a move. This is to avoid running out of time by spending it all early into the game.

Problem: It might take longer to search to a pre-defined depth than time allows.

Solution: Don't do a strictly depth-first search, but search to a higher and higher depth.

This is not as expensive as it sounds: Searching to shallow depths includes few moves and is cheap. Results from the previous level can be used to order the moves on the next level to get the optimal amount of pruning.

Iterative Deepening

Most programs give themselves a time limit to come up with a move. This is to avoid running out of time by spending it all early into the game.

Problem: It might take longer to search to a pre-defined depth than time allows.

Solution: Don't do a strictly depth-first search, but search to a higher and higher depth.

This is not as expensive as it sounds: Searching to shallow depths includes few moves and is cheap. Results from the previous level can be used to order the moves on the next level to get the optimal amount of pruning. Also can keep such results in hash table and re-use them.

Iterative Deepening

Most programs give themselves a time limit to come up with a move. This is to avoid running out of time by spending it all early into the game.

Problem: It might take longer to search to a pre-defined depth than time allows.

Solution: Don't do a strictly depth-first search, but search to a higher and higher depth.

This is not as expensive as it sounds: Searching to shallow depths includes few moves and is cheap. Results from the previous level can be used to order the moves on the next level to get the optimal amount of pruning. Also can keep such results in hash table and re-use them.

Obvious advantage: When time runs out we give the best move found so far, and that will at least be sensible. This is known as iterative deepening.

Modified alpha-beta search

In the description of alpha-beta search there is no pre-conceived idea as to the value of the current position.

Modified alpha-beta search

In the description of alpha-beta search there is no pre-conceived idea as to the value of the current position.

But if we use a hash table to keep track of results so far we can estimate a value.

Reminder: alpha-beta search

As the search reports back a value for the child of the current position we get

- successively increasing lower bounds for a max node (α);
- successively decreasing upper bounds for a min node (β).

Reminder: alpha-beta search

As the search reports back a value for the child of the current position we get

- successively increasing lower bounds for a max node (α);
- successively decreasing upper bounds for a min node (β).

As we descend into the tree we keep track of the current values of α and β by passing them down and updating them as appropriate.

Reminder: alpha-beta search

As the search reports back a value for the child of the current position we get

- successively increasing lower bounds for a max node (α);
- successively decreasing upper bounds for a min node (β).

As we descend into the tree we keep track of the current values of α and β by passing them down and updating them as appropriate.

- For a max node: only consider moves with value at least α. If we find such a move we adjust α accordingly.

Reminder: alpha-beta search

As the search reports back a value for the child of the current position we get

- successively increasing lower bounds for a max node (α);
- successively decreasing upper bounds for a min node (β).

As we descend into the tree we keep track of the current values of α and β by passing them down and updating them as appropriate.

- For a max node: only consider moves with value at least α. If we find such a move we adjust α accordingly.

  Find a value above β: that part of the tree that is irrelevant; return to parent without adjusting α or β.

Reminder: alpha-beta search

As the search reports back a value for the child of the current position we get

- successively increasing lower bounds for a max node (α);
- successively decreasing upper bounds for a min node (β).

As we descend into the tree we keep track of the current values of α and β by passing them down and updating them as appropriate.

- For a max node: only consider moves with value at least α. If we find such a move we adjust α accordingly.

  Find a value above β: that part of the tree that is irrelevant; return to parent without adjusting α or β.
- For a min node: only consider moves with value at most β. If we find such a move we adjust β accordingly.

Reminder: alpha-beta search

As the search reports back a value for the child of the current position we get

- successively increasing lower bounds for a max node (α);
- successively decreasing upper bounds for a min node (β).

As we descend into the tree we keep track of the current values of α and β by passing them down and updating them as appropriate.

- For a max node: only consider moves with value at least α. If we find such a move we adjust α accordingly.

  Find a value above β: that part of the tree that is irrelevant; return to parent without adjusting α or β.

- For a min node: only consider moves with value at most β. If we find such a move we adjust β accordingly.

  If we find a value below α: that part of the tree is irrelevant; return to the parent without adjusting α or β.

Modified alpha-beta search

Get provisional value v from earlier searches. Decide that real value will be between $\alpha \leq v$ and $\beta \geq v$. Use this as start for our search (instead of $-\infty$ and ∞).

Modified alpha-beta search

Get provisional value v from earlier searches. Decide that real value will be between $\alpha \leq v$ and $\beta \geq v$. Use this as start for our search (instead of $-\infty$ and ∞).
The carry out alpha-beta search, but on a max node:

Modified alpha-beta search

Get provisional value v from earlier searches. Decide that real value will be between $\alpha \leq v$ and $\beta \geq v$. Use this as start for our search (instead of $-\infty$ and ∞).
The carry out alpha-beta search, but on a max node:

- only consider moves which lead to a value at least α (this allows more pruning); (as before, only then α was guaranteed minimum;)

Modified alpha-beta search

Get provisional value v from earlier searches. Decide that real value will be between $\alpha \leq v$ and $\beta \geq v$. Use this as start for our search (instead of $-\infty$ and ∞).
The carry out alpha-beta search, but on a max node:

- only consider moves which lead to a value at least α (this allows more pruning); (as before, only then α was guaranteed minimum;)
- find value w above β: stop the search and report w back.

Modified alpha-beta search

Get provisional value v from earlier searches. Decide that real value will be between $\alpha \leq v$ and $\beta \geq v$. Use this as start for our search (instead of $-\infty$ and ∞).
The carry out alpha-beta search, but on a max node:

- only consider moves which lead to a value at least α (this allows more pruning); (as before, only then α was guaranteed minimum;)
- find value w above β: stop the search and report w back.

On a min node

- only consider moves which lead to a value at most β (again, this allows more pruning); (again as before, only then β was guaranteed maximum);

Modified alpha-beta search

Get provisional value v from earlier searches. Decide that real value will be between $\alpha \leq v$ and $\beta \geq v$. Use this as start for our search (instead of $-\infty$ and ∞).
The carry out alpha-beta search, but on a max node:

- only consider moves which lead to a value at least α (this allows more pruning); (as before, only then α was guaranteed minimum;)
- find value w above β: stop the search and report w back.

On a min node

- only consider moves which lead to a value at most β (again, this allows more pruning); (again as before, only then β was guaranteed maximum);
- if you find value w below α: stop the search and report w back.

Modified alpha-beta search

Search will report new value w and the following cases may arise:

Modified alpha-beta search

Search will report new value w and the following cases may arise:

- Value w is between α and β. This will be the correct value.

Modified alpha-beta search

Search will report new value w and the following cases may arise:

- Value w is between α and β. This will be the correct value.
- Value w is larger than β. Means our original upper bound β was too low and v too pessimistic.

Modified alpha-beta search

Search will report new value w and the following cases may arise:

- Value w is between α and β. This will be the correct value.
- Value w is larger than β. Means our original upper bound β was too low and v too pessimistic. Have to adjust our preliminary value v to w, and might consider allowing a larger range.

Modified alpha-beta search

Search will report new value w and the following cases may arise:

- Value w is between α and β. This will be the correct value.
- Value w is larger than β. Means our original upper bound β was too low and v too pessimistic. Have to adjust our preliminary value v to w, and might consider allowing a larger range.

  This is known as 'failing high'.

Modified alpha-beta search

Search will report new value w and the following cases may arise:

- Value w is between α and β. This will be the correct value.
- Value w is larger than β. Means our original upper bound β was too low and v too pessimistic. Have to adjust our preliminary value v to w, and might consider allowing a larger range.

  This is known as 'failing high'.
- Value w below α. Means our original lower bound α was too high and preliminary value v was overly optimistic.

Modified alpha-beta search

Search will report new value w and the following cases may arise:

- Value w is between α and β. This will be the correct value.
- Value w is larger than β. Means our original upper bound β was too low and v too pessimistic. Have to adjust our preliminary value v to w, and might consider allowing a larger range.

  This is known as 'failing high'.
- Value w below α. Means our original lower bound α was too high and preliminary value v was overly optimistic. Have to adjust preliminary value v to w, maybe allow a larger range.

Modified alpha-beta search

Search will report new value w and the following cases may arise:

- Value w is between α and β. This will be the correct value.
- Value w is larger than β. Means our original upper bound β was too low and v too pessimistic. Have to adjust our preliminary value v to w, and might consider allowing a larger range.

  This is known as 'failing high'.
- Value w below α. Means our original lower bound α was too high and preliminary value v was overly optimistic. Have to adjust preliminary value v to w, maybe allow a larger range.

  This is known as 'failing low'.

Modified alpha-beta search

Search will report new value w and the following cases may arise:

- Value w is between α and β. This will be the correct value.
- Value w is larger than β. Means our original upper bound β was too low and v too pessimistic. Have to adjust our preliminary value v to w, and might consider allowing a larger range.

  This is known as 'failing high'.
- Value w below α. Means our original lower bound α was too high and preliminary value v was overly optimistic. Have to adjust preliminary value v to w, maybe allow a larger range.

  This is known as 'failing low'.

This technique is known as aspiration search.

Benefits of aspiration search

In the best case: best move explored first, considered range contains correct value. Then:

Benefits of aspiration search

In the best case: best move explored first, considered range contains correct value. Then:

Total size of the tree searched reduced to

$$(\sqrt{b})^d,$$

where b: branching factor of the tree and d: depth of search.

Benefits of aspiration search

In the best case: best move explored first, considered range contains correct value. Then:

Total size of the tree searched reduced to

$$(\sqrt{b})^d,$$

where b: branching factor of the tree and d: depth of search.

So might be able to search twice as deeply in the same time—in the best case.

Benefits of aspiration search

In the best case: best move explored first, considered range contains correct value. Then:

Total size of the tree searched reduced to

$$(\sqrt{b})^d,$$

where b: branching factor of the tree and d: depth of search.

So might be able to search twice as deeply in the same time—in the best case.

This algorithm is implemented in most game-playing programs.

Benefits of aspiration search

In the best case: best move explored first, considered range contains correct value. Then:

Total size of the tree searched reduced to

$$(\sqrt{b})^d,$$

where b: branching factor of the tree and d: depth of search.

So might be able to search twice as deeply in the same time—in the best case.

This algorithm is implemented in most game-playing programs.

Good idea to include the current values of α and β in the hash table of previously searched positions.

Move ordering

In order for alpha-beta search to perform at its best (prune as often as possible) good moves have to be explored first.

Move ordering

In order for alpha-beta search to perform at its best (prune as often as possible) good moves have to be explored first.

For this we can use earlier search results, for example (where we have searched the current position to a lower depth).

Move ordering

In order for alpha-beta search to perform at its best (prune as often as possible) good moves have to be explored first.

For this we can use earlier search results, for example (where we have searched the current position to a lower depth).

Alternatively, we can build in heuristics to order the available moves. (Chess: Capturing moves, moves leading to check, moves turning pawn into other piece.)

Move ordering

In order for alpha-beta search to perform at its best (prune as often as possible) good moves have to be explored first.

For this we can use earlier search results, for example (where we have searched the current position to a lower depth).

Alternatively, we can build in heuristics to order the available moves. (Chess: Capturing moves, moves leading to check, moves turning pawn into other piece.)

Or we may have found a good move from a sibling in the game tree—this move might still be available.

Move ordering

In order for alpha-beta search to perform at its best (prune as often as possible) good moves have to be explored first.

For this we can use earlier search results, for example (where we have searched the current position to a lower depth).

Alternatively, we can build in heuristics to order the available moves. (Chess: Capturing moves, moves leading to check, moves turning pawn into other piece.)

Or we may have found a good move from a sibling in the game tree—this move might still be available.

Often it is sufficient to make sure the first few moves are the best candidates, because the others may be pruned in any case. Algorithms like HeapSort or SelectionSort deliver sorted items one by one.

Move ordering

In order for alpha-beta search to perform at its best (prune as often as possible) good moves have to be explored first.

For this we can use earlier search results, for example (where we have searched the current position to a lower depth).

Alternatively, we can build in heuristics to order the available moves. (Chess: Capturing moves, moves leading to check, moves turning pawn into other piece.)

Or we may have found a good move from a sibling in the game tree—this move might still be available.

Often it is sufficient to make sure the first few moves are the best candidates, because the others may be pruned in any case. Algorithms like HeapSort or SelectionSort deliver sorted items one by one.

Can search the first move(s) with big window for potential value (see aspiration search), and later moves with smaller ones. This is known as principal variation search.

Not winning from winning positions

A potential problem with alpha-beta search is a situation where the program knows it can win, but the code does not force progress, so that the actual win is never achieved.

Not winning from winning positions

A potential problem with alpha-beta search is a situation where the program knows it can win, but the code does not force progress, so that the actual win is never achieved.

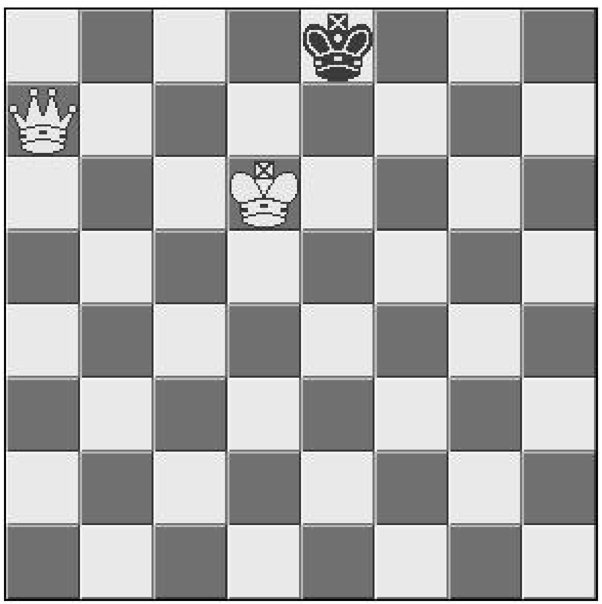
Assume every position from which the program can win is assigned a value of 1000. When looking for the next move, the program will ensure that the value remains 1000.

Not winning from winning positions

A potential problem with alpha-beta search is a situation where the program knows it can win, but the code does not force progress, so that the actual win is never achieved.

Assume every position from which the program can win is assigned a value of 1000. When looking for the next move, the program will ensure that the value remains 1000.

But:

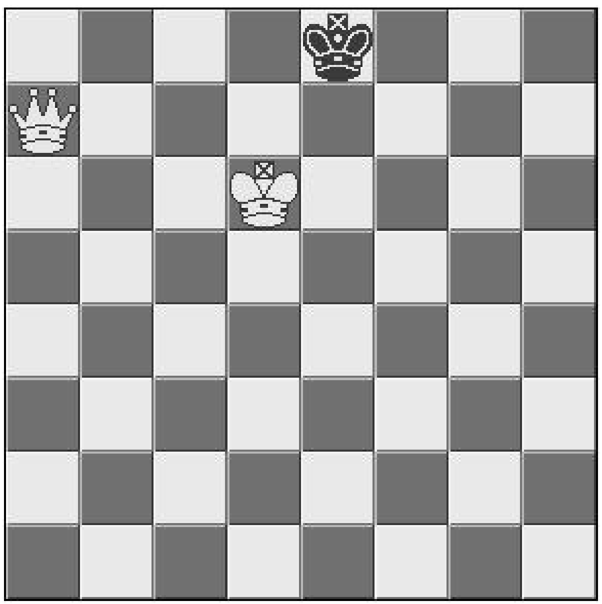


Not winning from winning positions

A potential problem with alpha-beta search is a situation where the program knows it can win, but the code does not force progress, so that the actual win is never achieved.

Assume every position from which the program can win is assigned a value of 1000. When looking for the next move, the program will ensure that the value remains 1000.

But:



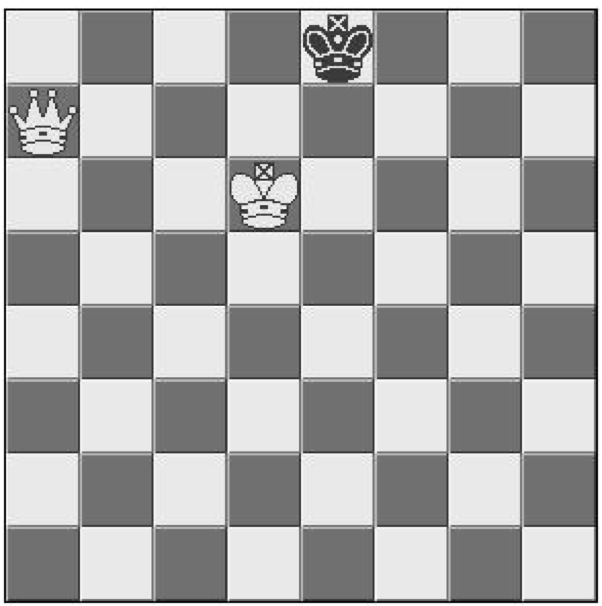
If White moves the king to $e6$ (one field to the right) then he is still in a winning position, with Black's only valid moves being to $d8$ and $f8$.

Not winning from winning positions

A potential problem with alpha-beta search is a situation where the program knows it can win, but the code does not force progress, so that the actual win is never achieved.

Assume every position from which the program can win is assigned a value of 1000. When looking for the next move, the program will ensure that the value remains 1000.

But:



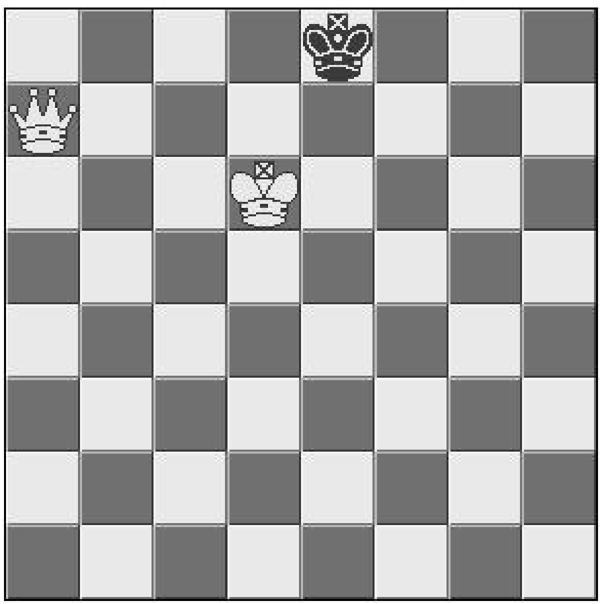
If White moves the king to $e6$ (one field to the right) then he is still in a winning position, with Black's only valid moves being to $d8$ and $f8$.
Let's assume Black moves to $d8$. Then moving the king back to $d6$ again gives White a winning position.

Not winning from winning positions

A potential problem with alpha-beta search is a situation where the program knows it can win, but the code does not force progress, so that the actual win is never achieved.

Assume every position from which the program can win is assigned a value of 1000. When looking for the next move, the program will ensure that the value remains 1000.

But:



If White moves the king to $e6$ (one field to the right) then he is still in a winning position, with Black's only valid moves being to $d8$ and $f8$.
Let's assume Black moves to $d8$. Then moving the king back to $d6$ again gives White a winning position.
But if Black now moves back to $e8$, we are back where we started and our program might go into a loop. This will lead to a draw since there are rules about repeating the same position.

Not winning from winning positions

A potential problem with alpha-beta search is a situation where the program knows it can win, but the code does not force progress, so that the actual win is never achieved.

Assume every position from which the program can win is assigned a value of 1000. When looking for the next move, the program will ensure that the value remains 1000.

Can avoid this by assigning slightly lower values to winning positions, for example

$$1000 - \text{ number of moves req'd to get win.}$$

Then alpha-beta search will work properly.

The horizon effect

General problem: The computer cannot see anything which is beyond its horizon, that is, that happens below its search depth.

The horizon effect

General problem: The computer cannot see anything which is beyond its horizon, that is, that happens below its search depth.

This can lead to the program trying to fend off bad events (capture of its piece, for example) by keeping them below the horizon.

The horizon effect

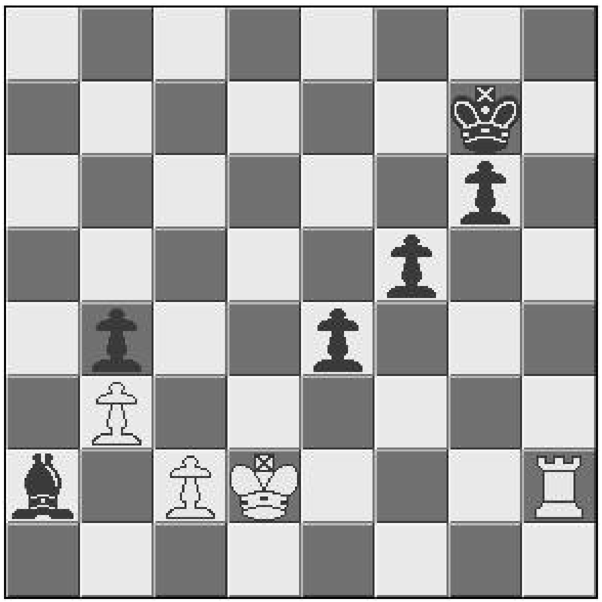
General problem: The computer cannot see anything which is beyond its horizon, that is, that happens below its search depth.

This can lead to the program trying to fend off bad events (capture of its piece, for example) by keeping them below the horizon.

In order to avoid, say, the capture of one of its pieces the program may try pointless moves which merely postpone the inevitable—typically these moves do not progress the program's play.

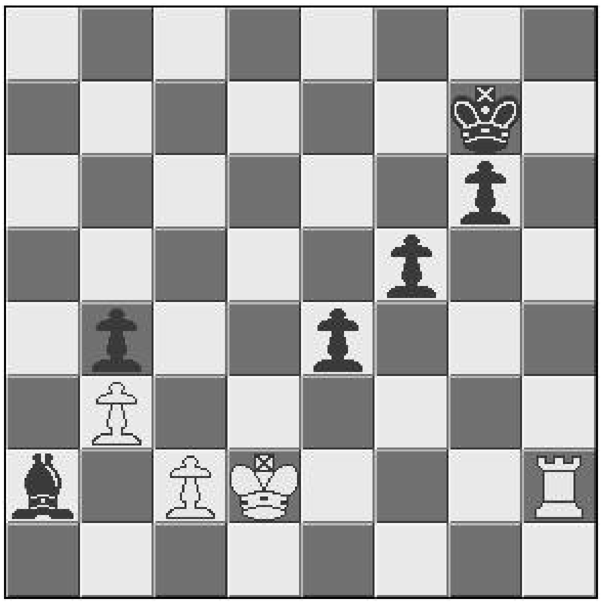
The horizon effect

General problem: The computer cannot see anything which is beyond its horizon, that is, that happens below its search depth.



The horizon effect

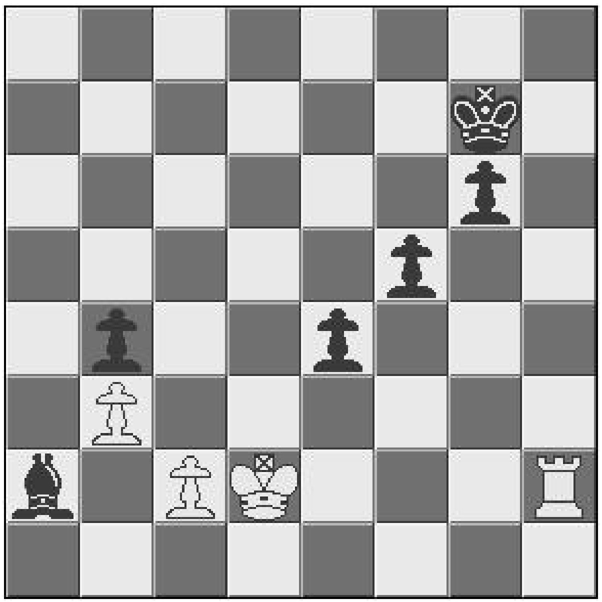
General problem: The computer cannot see anything which is beyond its horizon, that is, that happens below its search depth.



The black bishop is trapped by the white pawns. It will be captured (*e.g*: White rook: h2, h1, a1, a2).

The horizon effect

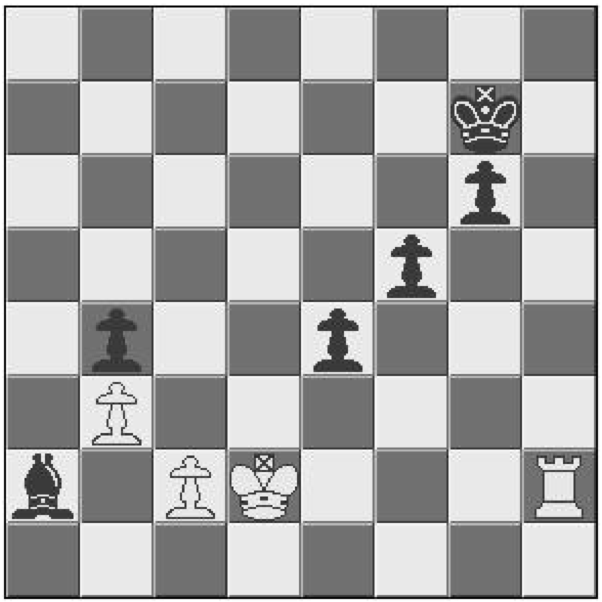
General problem: The computer cannot see anything which is beyond its horizon, that is, that happens below its search depth.



The black bishop is trapped by the white pawns. It will be captured (*e.g*: White rook: h2, h1, a1, a2).
If the program playing Black searches 6 moves ahead might move black pawn e4 to e3, checking the king. White has to react to this by moving the king or capturing this pawn.

The horizon effect

General problem: The computer cannot see anything which is beyond its horizon, that is, that happens below its search depth.



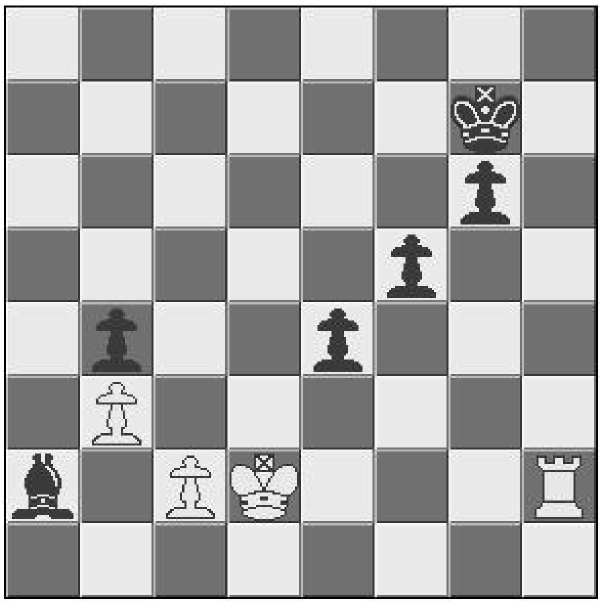
The black bishop is trapped by the white pawns. It will be captured (*e.g*: White rook: h2, h1, a1, a2).

If the program playing Black searches 6 moves ahead might move black pawn e4 to e3, checking the king. White has to react to this by moving the king or capturing this pawn.

That delays capture of the bishop so that the program thinks it is safe.

The horizon effect

General problem: The computer cannot see anything which is beyond its horizon, that is, that happens below its search depth.



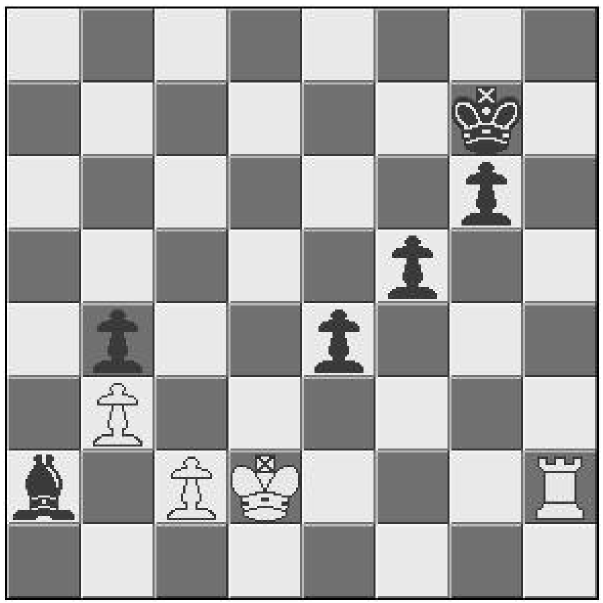
The black bishop is trapped by the white pawns. It will be captured (*e.g*: White rook: h2, h1, a1, a2).

If the program playing Black searches 6 moves ahead might move black pawn e4 to e3, checking the king. White has to react to this by moving the king or capturing this pawn.

That delays capture of the bishop so that the program thinks it is safe. The program might so sacrifice all its pawns, setting itself up for a loss.

The horizon effect

General problem: The computer cannot see anything which is beyond its horizon, that is, that happens below its search depth.



The black bishop is trapped by the white pawns. It will be captured (*e.g*: White rook: h2, h1, a1, a2).

If the program playing Black searches 6 moves ahead might move black pawn e4 to e3, checking the king. White has to react to this by moving the king or capturing this pawn.

That delays capture of the bishop so that the program thinks it is safe. The program might so sacrifice all its pawns, setting itself up for a loss.

Solutions: Add knowledge so that program can detect when piece is trapped. Increase overall depth of search in such situations so that horizon is windened. Whenever piece is threatened, search to deeper level selectively.

Selective extension

Many games do not search to fixed depth everywhere. Instead the select an appropriate depth, which is greater whenever

Selective extension

Many games do not search to fixed depth everywhere. Instead the select an appropriate depth, which is greater whenever

- there is reason to believe that the current value for a position is inaccurate or

Selective extension

Many games do not search to fixed depth everywhere. Instead the select an appropriate depth, which is greater whenever

- there is reason to believe that the current value for a position is inaccurate or
- when the current line of play is particularly important.

Selective extension

Many games do not search to fixed depth everywhere. Instead the select an appropriate depth, which is greater whenever

- there is reason to believe that the current value for a position is inaccurate or
- when the current line of play is particularly important.

For example, when currently set depth is reached search deeper for all moves which are likely to lead to change of evaluation considerably (Chess: capturing moves, check moves). This is known as quiescent search.

Selective extension

Many games do not search to fixed depth everywhere. Instead the select an appropriate depth, which is greater whenever

- there is reason to believe that the current value for a position is inaccurate or
- when the current line of play is particularly important.

For example, when currently set depth is reached search deeper for all moves which are likely to lead to change of evaluation considerably (Chess: capturing moves, check moves). This is known as quiescent search.

Alternatively one might increase search depth whenever the currently explored line contains a capturing move. Can only do this in a limited way, or the program will keep looking deeper and deeper!

Selective extension

Many games do not search to fixed depth everywhere. Instead the select an appropriate depth, which is greater whenever

- there is reason to believe that the current value for a position is inaccurate or
- when the current line of play is particularly important.

For example, when currently set depth is reached search deeper for all moves which are likely to lead to change of evaluation considerably (Chess: capturing moves, check moves). This is known as quiescent search.

Alternatively one might increase search depth whenever the currently explored line contains a capturing move. Can only do this in a limited way, or the program will keep looking deeper and deeper!

Many programs search deeper on what they think is the best move (see principal variation search).

Artificial phenomena

Sometimes carrying out an exhaustive search is better than the best move a human being will find!

Artificial phenomena

Sometimes carrying out an exhaustive search is better than the best move a human being will find!

One program, playing a Grandmaster, suddenly seemed to offer a rook for capture for no reason that the assembled experts could discern. After the game was over they made the machine go back to that position and asked it what would happen if it had made the move judged 'obviously better' by the audience.

Artificial phenomena

Sometimes carrying out an exhaustive search is better than the best move a human being will find!

One program, playing a Grandmaster, suddenly seemed to offer a rook for capture for no reason that the assembled experts could discern. After the game was over they made the machine go back to that position and asked it what would happen if it had made the move judged ‘obviously better’ by the audience. The machine pointed out an intricate mate which it was trying to avoid.

Artificial phenomena

Sometimes carrying out an exhaustive search is better than the best move a human being will find!

One program, playing a Grandmaster, suddenly seemed to offer a rook for capture for no reason that the assembled experts could discern. After the game was over they made the machine go back to that position and asked it what would happen if it had made the move judged ‘obviously better’ by the audience. The machine pointed out an intricate mate which it was trying to avoid. Arguably, it would have been better off leaving the rook alone and just hoping that the opponent wouldn’t see the mate!

Artificial phenomena

Sometimes carrying out an exhaustive search is better than the best move a human being will find!

One program, playing a Grandmaster, suddenly seemed to offer a rook for capture for no reason that the assembled experts could discern. After the game was over they made the machine go back to that position and asked it what would happen if it had made the move judged ‘obviously better’ by the audience. The machine pointed out an intricate mate which it was trying to avoid. Arguably, it would have been better off leaving the rook alone and just hoping that the opponent wouldn’t see the mate!

Mistakes or weaknesses in a program can be explored over and over (until the creator finds a chance to fix this, since these programs don’t learn).

Artificial phenomena

Sometimes carrying out an exhaustive search is better than the best move a human being will find!

One program, playing a Grandmaster, suddenly seemed to offer a rook for capture for no reason that the assembled experts could discern. After the game was over they made the machine go back to that position and asked it what would happen if it had made the move judged ‘obviously better’ by the audience. The machine pointed out an intricate mate which it was trying to avoid. Arguably, it would have been better off leaving the rook alone and just hoping that the opponent wouldn’t see the mate!

Mistakes or weaknesses in a program can be explored over and over (until the creator finds a chance to fix this, since these programs don’t learn). Many tournaments between various programs seemed to be more about who could discover whose built-in faults, rather than whose program genuinely played best!

Chess-playing programs

It begins

1950: Claude Shannon outlines a Chess-playing algorithm.

It begins

1950: Claude Shannon outlines a Chess-playing algorithm.

Evaluation function: Number of pieces, each with a weight, mobility, pawn formations.

It begins

1950: Claude Shannon outlines a Chess-playing algorithm.

Evaluation function: Number of pieces, each with a weight, mobility, pawn formations.

Search: depth-first minimax with two options.

It begins

1950: Claude Shannon outlines a Chess-playing algorithm.

Evaluation function: Number of pieces, each with a weight, mobility, pawn formations.

Search: depth-first minimax with two options.

- Search to a given depth, the same everywhere; and he called that the ‘fixed depth’ method.

It begins

1950: Claude Shannon outlines a Chess-playing algorithm.

Evaluation function: Number of pieces, each with a weight, mobility, pawn formations.

Search: depth-first minimax with two options.

- Search to a given depth, the same everywhere; and he called that the 'fixed depth' method.
- Search to a variable depth (depending on the 'type' of a position). Current move 'obviously bad': don't search it. Notion of 'stability' to decide where to stop. He called this the 'variable depth' method.

It begins

1950: Claude Shannon outlines a Chess-playing algorithm.

Evaluation function: Number of pieces, each with a weight, mobility, pawn formations.

Search: depth-first minimax with two options.

- Search to a given depth, the same everywhere; and he called that the 'fixed depth' method.
- Search to a variable depth (depending on the 'type' of a position). Current move 'obviously bad': don't search it. Notion of 'stability' to decide where to stop. He called this the 'variable depth' method.

Shannon thought this would be a useful application for computers, and would give insights into how one makes intelligent decisions.

The first Chess programs

1951: Alan Turing creates first Chess playing algorithm, and first Chess-playing program plays game against human opponent in Manchester. (Program: carried out by human, very weak.)

The first Chess programs

1951: Alan Turing creates first Chess playing algorithm, and first Chess-playing program plays game against human opponent in Manchester. (Program: carried out by human, very weak.)

First real Chess programs appear. In 1966: First match between Soviet and US American program ends 3:1. (Rules: 1 hour time for every 20 moves, so 3 minutes on average per move.) Searches cut off after time runs out; moves explored at random.

The first Chess programs

1951: Alan Turing creates first Chess playing algorithm, and first Chess-playing program plays game against human opponent in Manchester. (Program: carried out by human, very weak.)

First real Chess programs appear. In 1966: First match between Soviet and US American program ends 3:1. (Rules: 1 hour time for every 20 moves, so 3 minutes on average per move.) Searches cut off after time runs out; moves explored at random.

1974: First world computer Chess championships. Repeated every three years.

Improvements by mid-eighties

Hash tables to keep track of positions searched to which depth, and values discovered. (Often no update of value!)

Improvements by mid-eighties

Hash tables to keep track of positions searched to which depth, and values discovered. (Often no update of value!)

Opening libraries used so that programs follow ‘approved opening lines’.

Improvements by mid-eighties

Hash tables to keep track of positions searched to which depth, and values discovered. (Often no update of value!)

Opening libraries used so that programs follow ‘approved opening lines’.

Hash tables to store ‘tricky positions’ for future games—a limited amount of learning takes place.

Improvements by mid-eighties

Hash tables to keep track of positions searched to which depth, and values discovered. (Often no update of value!)

Opening libraries used so that programs follow 'approved opening lines'.

Hash tables to store 'tricky positions' for future games—a limited amount of learning takes place.

Iteratively deepening search implemented so that moves can be pre-ordered. No pruning based on search to low depths, or program will avoid sacrifices. Many early programs made that mistake!

Improvements by mid-eighties

Hash tables to keep track of positions searched to which depth, and values discovered. (Often no update of value!)

Opening libraries used so that programs follow ‘approved opening lines’.

Hash tables to store ‘tricky positions’ for future games—a limited amount of learning takes place.

Iteratively deepening search implemented so that moves can be pre-ordered. No pruning based on search to low depths, or program will avoid sacrifices. Many early programs made that mistake!

Search to variable depth, depending on whether the current position is judged to be ‘tricky’ or relatively straight-forward.

Artificial Intelligence?

Many techniques employed by the first Chess-playing programs turned out to be useful for other problems (iteratively deepening search for theorem provers, for example).

Artificial Intelligence?

Many techniques employed by the first Chess-playing programs turned out to be useful for other problems (iteratively deepening search for theorem provers, for example).

But: early ideas about ‘artificial intelligence’ were proven less than useful by these programs.

Artificial Intelligence?

Many techniques employed by the first Chess-playing programs turned out to be useful for other problems (iteratively deepening search for theorem provers, for example).

But: early ideas about 'artificial intelligence' were proven less than useful by these programs.

The algorithms did not try to mimic human decision making at all.

Artificial Intelligence?

Many techniques employed by the first Chess-playing programs turned out to be useful for other problems (iteratively deepening search for theorem provers, for example).

But: early ideas about 'artificial intelligence' were proven less than useful by these programs.

The algorithms did not try to mimic human decision making at all.

Nor did any true learning take place.

Artificial Intelligence?

Many techniques employed by the first Chess-playing programs turned out to be useful for other problems (iteratively deepening search for theorem provers, for example).

But: early ideas about ‘artificial intelligence’ were proven less than useful by these programs.

The algorithms did not try to mimic human decision making at all.

Nor did any true learning take place.

The early, strong claims regarding the possibilities of AI turned to out to be vastly exaggerated. Today, Artificial Intelligence often is about search techniques and the machine learning is very different from human learning!

Further improvements

By the mid- to late eighties, the following had been achieved.

Further improvements

By the mid- to late eighties, the following had been achieved.

Inclusion of large opening databases covering most approaches.

Further improvements

By the mid- to late eighties, the following had been achieved.

Inclusion of large opening databases covering most approaches.

Development of endgame databases; all five piece endgames were solved. Some of these solutions were genuinely new, in that people didn't realize that one player could force a win in some of these. As a result, the official Chess rules were changed.

Further improvements

By the mid- to late eighties, the following had been achieved.

Inclusion of large opening databases covering most approaches.

Development of endgame databases; all five piece endgames were solved. Some of these solutions were genuinely new, in that people didn't realize that one player could force a win in some of these. As a result, the official Chess rules were changed.

First Chess-specific circuitry employed by 'Deep Thought'; very fast move generation.

Further improvements

By the mid- to late eighties, the following had been achieved.

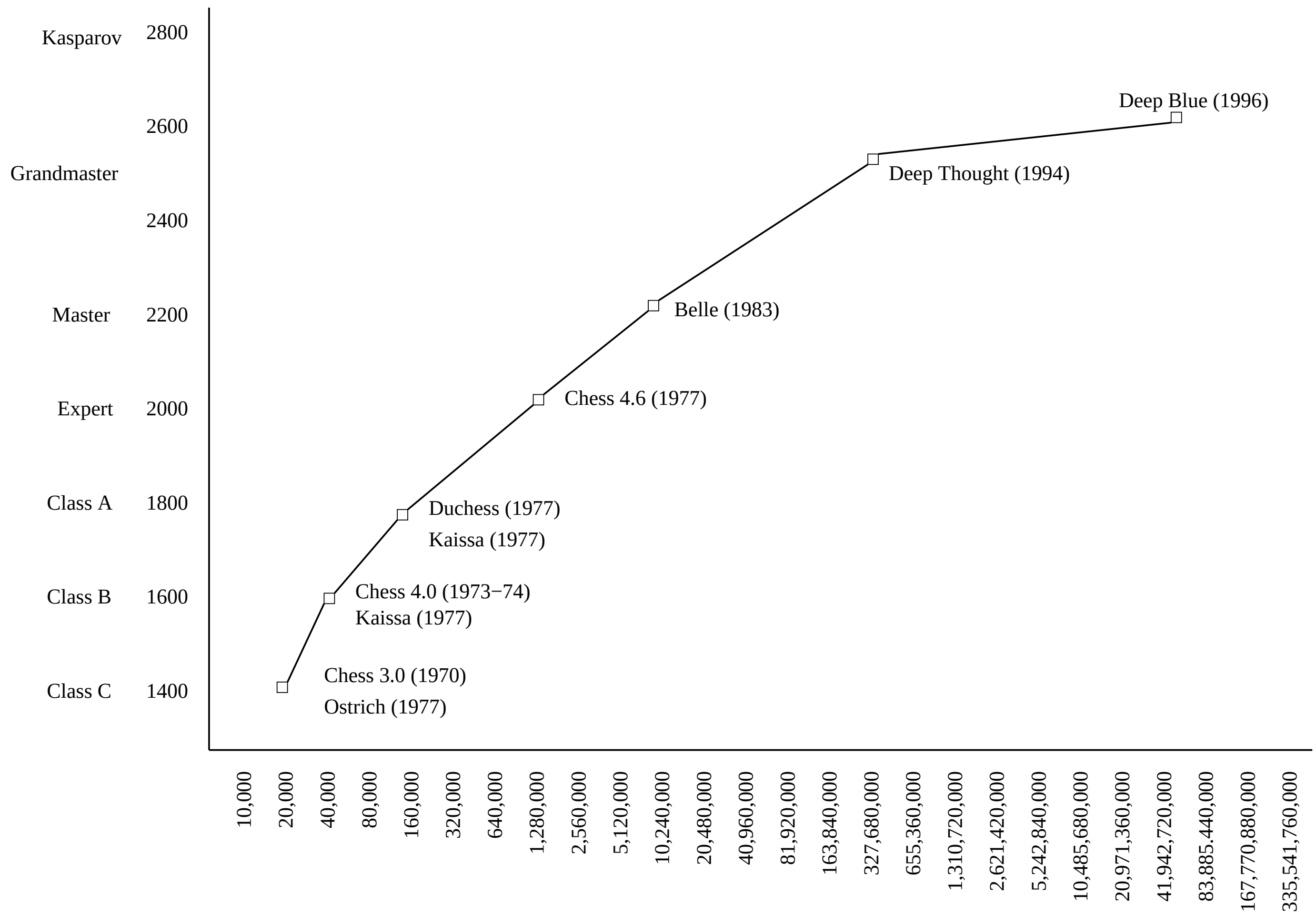
Inclusion of large opening databases covering most approaches.

Development of endgame databases; all five piece endgames were solved. Some of these solutions were genuinely new, in that people didn't realize that one player could force a win in some of these. As a result, the official Chess rules were changed.

First Chess-specific circuitry employed by 'Deep Thought'; very fast move generation.

Since late eighties: Main development has gone into specialized hardware.

Speed increases strength



Number of positions examined in three minutes, official ranking. (Note logarithmic scale along horizontal axis!) Where is perfect play?

Depth of search

To give some idea of how much strenght a program gains by searching to a greater depth, here are the results of a program (called ‘Belle’) playing against copies of itself which searched to a different depth (late seventies).

Depth of search

To give some idea of how much strenght a program gains by searching to a greater depth, here are the results of a program (called ‘Belle’) playing against copies of itself which searched to a different depth (late seventies).

	3	4	5	6	7	8	rating
3		4					1091
4	16		5.5				1332
5		14.5		4.5			1500
6			15.5		2.5		1714
7				17.5		3.5	2052
8					16.5		2320

Depth of search

To give some idea of how much strenght a program gains by searching to a greater depth, here are the results of a program (called 'Belle') playing against copies of itself which searched to a different depth (late seventies).

	4	5	6	7	8	9	rating
4		5	.5	0	0	0	1235
5	15		3.5	3	.5	0	1570
6	19.5	16.5		4	1.5	1.5	1826
7	20	17	16		5	4	2031
8	20	19.5	18.5	15		5.5	2208
9	20	20	18.5	16	14.5		2328

Depth of search

To give some idea of how much strenght a program gains by searching to a greater depth, here are the results of a program (called 'Belle') playing against copies of itself which searched to a different depth (late seventies).

	4	5	6	7	8	9	rating
4		5	.5	0	0	0	1235
5	15		3.5	3	.5	0	1570
6	19.5	16.5		4	1.5	1.5	1826
7	20	17	16		5	4	2031
8	20	19.5	18.5	15		5.5	2208
9	20	20	18.5	16	14.5		2328

Three or four levels more of search means outclassing one's opponent!

Improvement for increasing depth

Another way of measuring whether increasing the depth of search improves the program is to check whether search to a higher depth leads to a different move being chosen.

Improvement for increasing depth

Another way of measuring whether increasing the depth of search improves the program is to check whether search to a higher depth leads to a different move being chosen.

level	percentage of moves picked different from predecessor	approximate rating
4	33.1	1300
5	33.1	1570
6	27.7	1796
7	29.5	2037
8	26.0	2249
9	22.6	2433
10	17.7	2577
11	18.1	2725

Improvement for increasing depth

Another way of measuring whether increasing the depth of search improves the program is to check whether search to a higher depth leads to a different move being chosen.

level	percentage of moves picked different from predecessor	approximate rating
4	33.1	1300
5	33.1	1570
6	27.7	1796
7	29.5	2037
8	26.0	2249
9	22.6	2433
10	17.7	2577
11	18.1	2725

Often different moves are picked when searching to a higher depth. This explains why programs which search more levels play better.

Improvement for increasing depth

Another way of measuring whether increasing the depth of search improves the program is to check whether search to a higher depth leads to a different move being chosen.

level	percentage of moves picked different from predecessor	approximate rating
4	33.1	1300
5	33.1	1570
6	27.7	1796
7	29.5	2037
8	26.0	2249
9	22.6	2433
10	17.7	2577
11	18.1	2725

Often different moves are picked when searching to a higher depth. This explains why programs which search more levels play better.

This table shows that the benefit is diminished as overall depth increases.

Hardware for Chess

The following table gives an overview over Chess-playing programs and the hardware they were running on.

Hardware for Chess

Name	Year	Description
Ostrich	1981	5-processor Data General system
Ostrich	1982	8-processor Data General system
Cray Blitz	1983	2-processor Cray XMP
Cray Blitz	1984	4-processor Cray XMP
Sun Phoenix	1986	Network of 20 VAXs and Suns
Chess Challenger	1986	20 8086 microprocessors
Waycool	1986	64-processor N/Cube system
Waycool	1988	256-processor N/Cube system
Deep Thought	1989	3 2-processor VLSI chess circuits
Star Tech	1993	512-processor Connection Machine
Star Socrates	1995	1,824-processor Intel Paragon
Zugzwang	1995	96-processor GC-Powerplus distributed system (based on the PowerPC)
Deep Blue	1996	32-processor IBM RS/6000 SP with 6 VLSI chess circuits per processor

Man *versus* machine

1978: First ‘serious’ match of man against computer. Man won.

Man *versus* machine

1978: First ‘serious’ match of man against computer. Man won.

Late eighties: First Grandmaster beaten by program in tournament (rather than in display match).

Man *versus* machine

1978: First 'serious' match of man against computer. Man won.

Late eighties: First Grandmaster beaten by program in tournament (rather than in display match).

First program ranked Grandmaster (Deep Thought); beaten by World Champion in only 41 moves.

Man *versus* machine

1978: First 'serious' match of man against computer. Man won.

Late eighties: First Grandmaster beaten by program in tournament (rather than in display match).

First program ranked Grandmaster (Deep Thought); beaten by World Champion in only 41 moves.

1993: Program beats top 20 player. Computers participate increasingly in Chess tournaments, usually on dedicated hardware.

Man *versus* machine

1978: First ‘serious’ match of man against computer. Man won.

Late eighties: First Grandmaster beaten by program in tournament (rather than in display match).

First program ranked Grandmaster (Deep Thought); beaten by World Champion in only 41 moves.

1993: Program beats top 20 player. Computers participate increasingly in Chess tournaments, usually on dedicated hardware.

Commercial interest: IBM takes on the team that came up with Deep Thought, and they develop Deep Blue. (Previously: academic effort, with much weaker commercially available programs.)

Man *versus* machine

1978: First ‘serious’ match of man against computer. Man won.

Late eighties: First Grandmaster beaten by program in tournament (rather than in display match).

First program ranked Grandmaster (Deep Thought); beaten by World Champion in only 41 moves.

1993: Program beats top 20 player. Computers participate increasingly in Chess tournaments, usually on dedicated hardware.

Commercial interest: IBM takes on the team that came up with Deep Thought, and they develop Deep Blue. (Previously: academic effort, with much weaker commercially available programs.)

1996: Deep Blue takes on the world champion, match ends 2 to 4.

Man *versus* machine

1978: First ‘serious’ match of man against computer. Man won.

Late eighties: First Grandmaster beaten by program in tournament (rather than in display match).

First program ranked Grandmaster (Deep Thought); beaten by World Champion in only 41 moves.

1993: Program beats top 20 player. Computers participate increasingly in Chess tournaments, usually on dedicated hardware.

Commercial interest: IBM takes on the team that came up with Deep Thought, and they develop Deep Blue. (Previously: academic effort, with much weaker commercially available programs.)

1996: Deep Blue takes on the world champion, match ends 2 to 4.

1997: Rematch, ending 2.5 to 3.5, Kasparov makes mistake in final and deciding match.

Man *versus* machine–II

Comparing the way man and machine play, we find that programs are based on

- raw speed to do deep searching;

Man *versus* machine–II

Comparing the way man and machine play, we find that programs are based on

- raw speed to do deep searching;
- libraries for openings and endgames.

Man *versus* machine–II

Comparing the way man and machine play, we find that programs are based on

- raw speed to do deep searching;
- libraries for openings and endgames.

Seeing a Chess board, good human players will only ever consider a handful of moves. We have no idea how they make the decision which moves to look at in more detail. This has something to do with pattern recognition, at which computers are very bad.

Man *versus* machine–II

Comparing the way man and machine play, we find that programs are based on

- raw speed to do deep searching;
- libraries for openings and endgames.

Seeing a Chess board, good human players will only ever consider a handful of moves. We have no idea how they make the decision which moves to look at in more detail. This has something to do with pattern recognition, at which computers are very bad.

Chess-playing programs have done very little to improve our understanding of how humans think and make decisions.

Other games

In Chess the very best programs (with many man-hours invested in them) play as good as the best human players.

Other games

In Chess the very best programs (with many man-hours invested in them) play as good as the best human players.

In 1982 a program called IAGO was assessed as playing Othello (Reversi) at world championship level, but it didn't take part in tournaments.

Other games

In Chess the very best programs (with many man-hours invested in them) play as good as the best human players.

In 1982 a program called IAGO was assessed as playing Othello (Reversi) at world championship level, but it didn't take part in tournaments.

In 1994 a program called Chinook became world Checkers champion, but the reigning world champion had to forfeit the match due to illness.

Other games

In Chess the very best programs (with many man-hours invested in them) play as good as the best human players.

In 1982 a program called IAGO was assessed as playing Othello (Reversi) at world championship level, but it didn't take part in tournaments.

In 1994 a program called Chinook became world Checkers champion, but the reigning world champion had to forfeit the match due to illness.

Go-playing programs currently are way below even good amateurs, let alone professionals.

Summary of Section 4

- Three tasks have to be solved when writing a game-playing program: Designing an internal board representation and generating valid moves, designing an evaluation function and implementing (some variant of) alpha-beta search. All considerations are overshadowed by the need for speed.

Summary of Section 4

- Three tasks have to be solved when writing a game-playing program: Designing an internal board representation and generating valid moves, designing an evaluation function and implementing (some variant of) alpha-beta search. All considerations are overshadowed by the need for speed.
- Board representations should make the generation of moves, doing and undoing them fast.

Summary of Section 4

- Three tasks have to be solved when writing a game-playing program: Designing an internal board representation and generating valid moves, designing an evaluation function and implementing (some variant of) alpha-beta search. All considerations are overshadowed by the need for speed.
- Board representations should make the generation of moves, doing and undoing them fast.
- Evaluation functions require knowledge about the game in question. They assign a value to a board position statically, from just what is on the board.

Summary of Section 4

- Three tasks have to be solved when writing a game-playing program: Designing an internal board representation and generating valid moves, designing an evaluation function and implementing (some variant of) alpha-beta search. All considerations are overshadowed by the need for speed.
- Board representations should make the generation of moves, doing and undoing them fast.
- Evaluation functions require knowledge about the game in question. They assign a value to a board position statically, from just what is on the board.
- Alpha-beta search is concerned with assigning a value to a position by searching the game tree below it and eventually applying the evaluation function. Searching to greater depth will result in a better program, so any gain in speed goes into searching to a greater depth. There are many tricks one might try to employ in order to concentrate on searching the relevant parts of the game tree; in particular ordering moves to search the most promising ones first.

Summary of Section 4

- Three tasks have to be solved when writing a game-playing program: Designing an internal board representation and generating valid moves, designing an evaluation function and implementing (some variant of) alpha-beta search. All considerations are overshadowed by the need for speed.
- Board representations should make the generation of moves, doing and undoing them fast.
- Evaluation functions require knowledge about the game in question. They assign a value to a board position statically, from just what is on the board.
- Alpha-beta search is concerned with assigning a value to a position by searching the game tree below it and eventually applying the evaluation function. Searching to greater depth will result in a better program, so any gain in speed goes into searching to a greater depth. There are many tricks one might try to employ in order to concentrate on searching the relevant parts of the game tree; in particular ordering moves to search the most promising ones first.
- Most effort so far has gone into creating Chess-playing programs. They have profited from faster hardware, and many improvements have been made which are very Chess-specific: better heuristics, opening and endgame libraries, and the like.

Summary of Section 4

- Three tasks have to be solved when writing a game-playing program: Designing an internal board representation and generating valid moves, designing an evaluation function and implementing (some variant of) alpha-beta search. All considerations are overshadowed by the need for speed.
- Board representations should make the generation of moves, doing and undoing them fast.
- Evaluation functions require knowledge about the game in question. They assign a value to a board position statically, from just what is on the board.
- Alpha-beta search is concerned with assigning a value to a position by searching the game tree below it and eventually applying the evaluation function. Searching to greater depth will result in a better program, so any gain in speed goes into searching to a greater depth. There are many tricks one might try to employ in order to concentrate on searching the relevant parts of the game tree; in particular ordering moves to search the most promising ones first.
- Most effort so far has gone into creating Chess-playing programs. They have profited from faster hardware, and many improvements have been made which are very Chess-specific: better heuristics, opening and endgame libraries, and the like.