Contents

Part I

Foundations

Need:
consistent notation for moves / trees / sides / ...
build my own game-board typesetting macros?
pseudo-code / program-listing style...
do a couple full game rules
(fairy) chess boards
go boards
hex boards
USE TikZ package for graphs/trees?

Chapter 1

Introduction

1.1 Overview

The goal of this book is to be an encyclopedic tutorial on the programming techniques for traditional games. That is, how to program the "artificial intelligence" to allow a computer to play games such as chess, shogi, go (weiqi). These are games that have no hidden information and no randomness.

We will discuss games such as backgammon and poker that differ in critical ways. We will give a brief introduction to techniques and research on these games.

We may discuss "wargames" (so called hex and chit games), though there is very little literature written on programming for them.

We will *not* be discussing "video games".

We aim to be tutorial in that we will explain the ideas behind the various techniques as well as giving explicit implementations for specific games. (Sometimes it can be less than obvious how to proceed from an academic paper written in a theoretical style to an actual implementation in code.)

We will be encyclopedic in that we will cover a wide range of techniques.

Note that though we will be giving implementations for specific games, the goal is not to dive too deeply into any specific game. (For example, we will provide a limited discussion of position evaluation for chess, rather giving a few elements as representative of what might be used.) No doubt readers of this book will be able to improve on the provided systems, which will be reasonable but by no means "world champion" level game systems. Our goal is to provide tools and examples for readers to be able to develop their own games, hopefully giving them a speedier route to "cutting edge" of games development and research.

Some people may be wondering what interest there might be in developing players for games outside of the major and well-known ones such as chess, shogi, or go. And even for those, there may not seem to be much of a demand that isn't already satisfied. However, there are many games for which there are no programs developed. Additionally, one finds that even in typical video games there can be embedded "mini games" which sometimes take the form of board games. Instead of just using a typical chess player, one could use the techniques from this book to develop a player for a unique game, thus potentially giving better flavor — why should an alien creature from a far off star-system play an identical game to chess (even if the pieces are renamed)? Much more interesting if they play an unknown game with the same depth and complexity as chess, but unique for that game! Perhaps they could play Edgar Rice Bourrough's Jetan from his Mars ... or perhaps they could play a three- or four-dimensional game. The techniques in this book should give you the tools to implement programs to play all of this at a solid level.

(Indeed, the targeted level of play for all implemented systems is that they should, at a minimum, be able to beat the author...)

1.2 Games

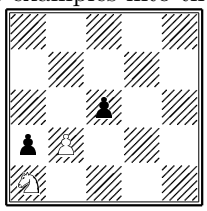
vocabulary for move / board / game / state / evaluation / etc.

vocabulary for pieces / colors / etc.

1.3 Programming language

We will use the Rust programming language for our examples. Although it is not a very wide-spread language, it is ideally suited for programming game systems. It is low-level with extremely high-performance possible (comparable to C/C++), but it allows for generic and high-level programming through its traits systems. Additionally, its "safety" properties allows for spending less time debugging memory issues than in C++, for example.

For those who are interested in using other programming languages should find it relatively easy to translate the examples into their language of choice.



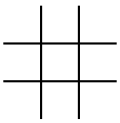
Chapter 2

Warm Up — Tic-Tac-Toe

The simple game of TIC-TAC-TOE (also known as NOUGHTS-AND-CROSSES or ...) is a simple child's game which is well-known.

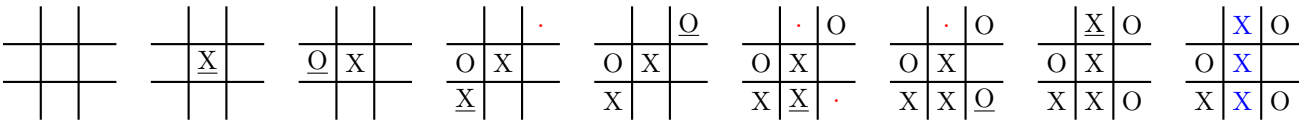
2.1 Rules

For completeness, we'll review the rules here: the game is played on a 3×3 grid of cells



by two players "X" and "O". Players alternate (with "X" moving first) selecting an empty cell and claiming it by marking it with **X** or **O**, respectively. When a player has claimed three cells in a straight line (horizontally, vertically, or diagonally), then that player wins; i.e., 3-in-a-row wins. If the board fills with no 3-in-a-row, then the game is declared drawn (sometimes called a "cat's game" or "Nick's game" or ...).

For example, here is a sample play, where the O player slips up on the first move and allows X to set up a winning combination (it was necessary for O to respond to X's central play by playing to a corner.) (We mark immediate threats with a dot.)



And X has won with a line down the center.

Although the game is quite simple and has little interest for actual play by experienced players (as all games inevitably end in draws), we will go ahead and develop a player for it. The simplicity of this game will allow us to build a full system and look at a variety of aspects of game programming without getting bogged down in complicated and game-specific details of rules or game representations. And, as it is such a simple game, we can do a deeper analysis of the game. There are fewer than $3^9 = 19\,683$ different possible board states the game can be in — each of the 9 squares can be empty, marked "X", or marked "O" (three possibilities). The player to move can be determined by the number of X's and O's on the board (if they are the same, it is the first player's move, if different then the second player's.) Note that this count is an overestimate, since some of the counted possibilities are impossible in a game (for example, a board filled completely with X's.) In fact, the exact number of legal board positions that can occur in a game is ???TODO??? — we will compute this number in our work below).

Later in this work, we will investigate many more complex games of real interest.

2.2 Board Representation

The first step we'll need is to find a way to represent the state of the game.

First, we define the "side" and a "cell" which may be empty or contain a side.

```
pub enum Side {
    X=1,
    O=2,
}
type Cell = Option<Side>;
```

We'll define the possible results for a game

```
pub enum Result {
    XWin=1,
    OWin=2,
    Draw=3,
}
```

Then we'll create a board struct to hold a representation of the board using the most obvious approach where we simply make a three-by-three array to hold the state of each cell in the TIC-TAC-TOE grid. The coördinates corresponding to cells in our array are as follows:

(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)

We define a Square type as an alias for the ordered pair and define a couple of accessor functions.

```
#[derive(Clone,Copy,Debug,Move)]
pub struct Board {
    cells : [[Cell; 3]; 3],
    moves_played : usize,
    to_move : Option<Side>,
    result : Option<Result>,
}
type Square = (usize,usize);
impl Board {
    fn new() -> Self {
        Board {
            cells        : [[None; 3]; 3],
            moves_played : 0,
            to_move      : Some(Side::X),
            result       : None,
        }
    }
    fn reset(&mut self) {
        *self = Board::new();
    }
    fn cell(&self, square: Square) -> Cell {
        self.cells[square.0][square.1]
    }
    fn set_cell(&mut self, square: Square, cell: Cell) {
        self.cells[square.0][square.1] = cell;
    }

}
```

We'll probably want a little nicer representation than the default Rust version:

```
impl fmt::Display for Board {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        //write!(f, "{}", self.to_char())
        fn ch(c: Cell) -> char {
            match c {
                Some(Side::X) => 'X',
                Some(Side::O) => 'O',
                None          => ' ',
            }
        }
        write!(f, "{}|{}|{}\n-+-+-\n{}|{}|{}\n-+-+-\n{}|{}|{}\n",
            ch(self.cells[0][0]), ch(self.cells[0][1]), ch(self.cells[0][2]),
            ch(self.cells[1][0]), ch(self.cells[1][1]), ch(self.cells[1][2]),
            ch(self.cells[2][0]), ch(self.cells[2][1]), ch(self.cells[2][2])
            )?;
        write!(f, "moves played: {}\n", self.moves_played)?;
        write!(f, "to move: {}\n", self.to_move)?;
        write!(f, "result: {}\n", self.result)
    }
}
}
```

The next thing we need is to analyze a board position to determine if somebody has won the position or if it is a draw or if the game is still on-going. For this we will simply check every possible winning triplet to see that each cell contains a piece and that all the cells are identical. For this, we'll simply exhaustively list all possibilities and go through them:

```
impl Board {
    fn compute_result(&self) -> Option<Result> {
        static let triplets = [
            // horizontal
            [(0,0), (0,1), (0,2)], [(1,0), (1,1), (1,2)], [(2,0), (2,1), (2,2)],
            // vertical
            [(0,0), (1,0), (2,0)], [(0,1), (1,1), (2,1)], [(0,2), (1,2), (2,2)],
            // diagonal
            [(0,0), (1,1), (2,2)], [(0,2), (1,1), (2,0)]
        ];
        for trip in triplets {
            if self.cell(trip[0]).is_some
                && self.cell(trip[0])==self.cell(trip[1])
                && self.cell(trip[0])==self.cell(trip[2])
            {
                return (
                    match self.cell(trip[0]) {
                        Side::X => Some(Result::XWin),
                        Side::O => Some(Result::OWin),
                        //None => Some(Result::Draw),//impossible
                    }
                );
            }
        }
        if self.moves_played==9 {
            return Some(Draw);
        } else {
            return None;
        }
    }
}
```

Finally, we need to be able to get all possible moves and make or unamke a move on the board. (When we get to the search routines, we'll see why it is necessary to be able to *un*make a move.) (These won't necessarily be the most efficient implementations, but they will be sufficient for our purposes here. Later we'll look at more performant approaches.)

```
impl Board {
    fn moves(&self) -> Vec<Square> {
        let mut ml = Vec::new();
        for r in (0..3) {
            for c in (0..3) {
                if self.cells[r][c].is_none() {
                    ml.push((r,c));
                }
            }
        }
        return ml;
    }
    // returns true if move was valid for making
    fn make_move(&mut self, m: Square) -> boolean {
        if self.cell(m).is_some() || self.to_move.is_none() {
            false
        } else {
            self.set_cell(m, self.to_move);
            self.moves_played += 1;
            self.to_move =
                match self.to_move {
                    Some(Side::X) => Some(Side::O),
                    Some(Side::O) => Some(Side::X),
                    // None => None, // impossible
                };
            self.result = self.compute_result();
            if self.result.is_some() { self.to_move = None; }
            true
        }
    }
    // returns true if move was valid for unmaking
    fn unmake_move(&mut self, m: Square) -> boolean {
        if self.cell(m).is_none() || self.moves_played==0 {
            false
        } else {
            self.set_cell(m, None);
            self.moves_played -= 1;
```

```
            self.to_move = Some(if self.moves_played%2==0 {Side::X} else {Side::O});
            self.result = None;
            true
        }
    }
}
```

To test out our representation, we'll implement a simple REPL. This will allow us to interactively make moves, reset the board, etc. We will expand it later as we implement further functionality for searching, etc.

```
pub fn main() {
    let mut board = Board::new();
    let mut quit = false;
    while (!quit) {
        println!("Board:\n{}", board);
        println!("[reset] reset the board\n(1) [moves] show possible moves\n(2) [move n] make move\n(3) [unmove n] unmake move\n[quit] end program
        // get input
        println!("> ");
        match choice {
            "reset" => { board.reset(); }
            "moves" => { println!("Moves: {}", board.moves()); }
            "move n" => { get input & make move }
            "unmove n" => { get input & unmake move }
            "quit" => { quit = true; }
            _ => {}
        }
    }
}
```

2.3 Search

The first programming we'll do is write a solver for the game. This will tell us, for any position, the *theoretical value* of the position — that is, if we assume two players who play the best possible move at each position, whether the game is a win for the first player or a win for the second player or a draw. It is a theoretical fact, first proven by von Neumann, that one of these three possibilities must hold. Our algorithm will in fact be a kind of constructive "proof" of this.

The basic idea is that we systematically examine each possible move: we temporarily make the move on the board, then repeat the process on the resulting position, making each possible move by the opponent, etc. until we reach positions which are won for one of the players or draw positions. (Positions where the game has ended are called "terminal" positions.) We then proceed back up the tree, assuming that each player will make the beset possible move. That is, if there is a move which will lead to a forced win for that player, he will select it, otherwise if there is a move leading to a drawn game, he will select that, otherwise (and only then) will he choose a move which leads to a forced win by the opponent.

Some examples should make this clear:

Now here is the straightforward code for searching:

```
pub fn fullsearch_v1(b: &mut Board) -> (Option<Side>,usize) {
    // check if game is just over
    let res = b.is_game_move();
    match res {
        Some(s) => { return (s,1); }
        None => { }
    }
    if b.to_move==Side.X {
        let mut nodes = 1;
        let ml = b.moves();
        // initialize to loss
        let mut best_so_far = Some(Side.O);
        for m in ml {
            b.make_move(m);
            let (result,n) = fullsearch_v1(b);
            b.unmake_move(m);
            nodes += n;
            best_so_far =
                match (result, best_so_far) {
                    (_,Some(Side.X)) | (Some(Side.X),_) => Some(Side.X),
                    (_,None) | (None,_) => None,
                    _ => Some(Side.O),
                };
        }
        return (best_so_far, nodes);
```

```
    } else if b.to_move==Side.O {
        let ml = b.moves();
        let mut best_so_far = Some(Side.X);
        for m in ml {
            b.make_move(m);
            let (result,n) = fullsearch_v1(b);
            b.unmake_move(m);
            nodes += n;
            best_so_far =
                match (result, best_so_far) {
                    (_,Some(Side.O)) | (Some(Side.O),_) => Some(Side.O),
                    (_,None) | (None,_) => None,
                    _ => Some(Side.X),
                };
        }
        return (best_so_far, nodes);
    } else {
        panic!("Unexpected: nobody to move in position {}", b);
    }
}
```

But looking at this code, we readily see that the two branches depending on who is to move are nearly identical, differing in only the preference for side, In particular, notice that the behaviour of the two sides to play are symmetric — each wants to so let's implement a more unified version. We'll add a helper function to the Side enum to give us the opposing side of a given one.

```
impl Side {
    pub fn other(self) -> Self {
        match self {
            Side.O => Side.X,
            Side.X => Side.O,
        }
    }
}
```

Then we can implement the more compact version

```
pub fn fullsearch_v2(b: &mut Board) -> (Option<Side>,usize) {
    // check if game is just over
    let res = b.is_game_move();
    match res {
        Some(s) => { return (s,1); }
        None => { }
    }
    let tomove = b.to_move;
    let other = tomove.other();
    let mut nodes = 1;
    let ml = b.moves();
    // initialize to loss
    let mut best_so_far = Some(other);
    for m in ml {
        b.make_move(m);
        let (result,n) = fullsearch_v2(b);
        b.unmake_move(m);
        nodes += n;
        best_so_far =
            match (result, best_so_far) {
                (_,Some(tomove)) | (Some(tomove),_) => Some(tomove),
                (_,None) | (None,_) => None,
                _ => Some(other),
            };
    }
    return (best_so_far, nodes);
}
```

The next modification is a speed-up: you may have noticed that once we find a winning move for the side to move, then the result will always be a win for that side. It would be a irrational to play a drawing or losing move if you've already found a winning move! Thus, in this case, we can return immediately from the function once we have such a move.:

```
pub fn fullsearch_v3(b: &mut Board) -> (Option<Side>,usize) {
    // check if game is just over
    let res = b.is_game_move();
    match res {
        Some(s) => { return (s,1); }
        None => { }
    }
    let tomove = b.to_move;
    let other = tomove.other();
    let mut nodes = 1;
```

```
    let ml = b.moves();
    // initialize to loss
    let mut best_so_far = Some(other);
    for m in ml {
        b.make_move(m);
        let (result,n) = fullsearch_v3(b);
        b.unmake_move(m);
        nodes += n;
        best_so_far =
            match (result, best_so_far) {
                (_,Some(tomove)) | (Some(tomove),_) => { return (Some(tomove),nodes) },
                (_,None) | (None,_) => None,
                _ => Some(other),
            };
    }
    return (best_so_far, nodes);
}
```

This is a tiny change to a single line, but we'll see it can give a significant improvement to the number of nodes searched.

Searcher	nodes searched	time (ms)
fullsearch_v1		
fullsearch_v2		
fullsearch_v3		

Note that a lot of programming can be effectively accomplished like this: don't be afraid to implement the simplest, most straightforward code that will achieve your goal, and then go back and refine it. Often you will have a better chance of getting it correct this way, rather than attempting to immediately write the more complicated or "elegant" version. Then you can improve your correct solution for performance, elegance, etc. The simpler version is more likely to be correct and gives you a reference to verify the more complex implementation.

See `chapter-1-tictactoe.rs`.

2.4 Transposition tables

In our search above, it turns out that there is a lot of repeated and redundant computation, due to "transpositions" — differing sequences of moves that end up with the same position. Then all search from that position on will be identical, though the path leading to it was different. So if we store the result from the first time we encounter this position, then we can simply reuse it without having to re-search.

In order to store, we will encode all possible board positions into an integer and use it to index into an array. We use a straightforward approach: for each cell, we code empty as 0, X as 1, and O as 2, then compute a code for a position as:

$$\sum_{i=0}^{8} a_i 3^i = a_0 \cdot 3^0 + a_1 \cdot 3^1 + a_2 \cdot 3^2 + a_3 \cdot 3^3 + a_4 \cdot 3^4 + a_5 \cdot 3^5 + a_6 \cdot 3^6 + a_7 \cdot 3^7 + a_8 \cdot 3^8$$

and noting that the maximum possible value achievable is $3^9 - 1 = 19682$, so we can store the results in a relatively small array of 19683 elements (a little larger than 19k entries).

Our modified search is then as follows:

```
impl Board {
    pub fn code(&self) -> usize {
          *   1 + *   3 + *   9
        + *  27 + *  81 + * 243
        + * 729 + *2187 + *6561
    }
}
pub fn search(..., hash: Vector<...>) {
}
```

2.5 Alternative representations

Next, to allow us to try a few different representations, we'll define a generic trait (or "interface") that the boards will all satisfy. Then, as long as our search (and other functionality) uses only the trait, then we can use any

of the representations with the same search code. (Note that we will not incur any performance penalties for genericity of our search code in Rust, due to its policy of "monomorphization" of generic (templated) code.

```
pub trait Board {
    type Square : Debug,Display,Clone,Copy,Move;
    fn new() -> Self;
    fn reset(&mut self);
    fn cell(&self, square: Square) -> Cell;
    fn set_cell(&mut self, square: Square, cell: Cell);
}
```

A slight variation, which is sometimes convenient, is to collapse the two-dimensional array to a one-dimensional array. Although it is less convenient to access a cell by row and column (though it is easy enough to write helper functions, if one really wants), it often turns out to be more convenient to have a single integer index denoting each

0	1	2
3	4	5
6	7	8

```
pub struct Board_array : Board {
    cells : [Cell; 9],
}
impl Board for Board_array {
    type Square = usize;
    fn new() -> Self {
        Board_array { cells: [None; 9] }
    }
    fn reset(&mut self) {
        self.cells = [None; 9];
    }
    fn cell(&self, square: Square) -> Cell {
        self.cells[square]
    }
    fn set_cell(&mut self, square: Square, cell: Cell) {
        self.cells[square] = cell;
    }
    // helper
    fn rowcol_to_square(row: usize, col: usize) -> Square {
        row * 3 + col
    }
}
```

A less obvious approach, but taking much less space and allowing us to do some cute tricks in implemetation is a "bit-board" approach. That is, we use the individual bits in an integer to encode a single fact about each cell. For TIC-TAC-TOE, we will use one bit-board to indicate cells selected by X and another to indicate cells selected by O. This representation is surprisingly flexible and allows for fast computation of many features of the board (though the power of this representation won't be clear until we look at more complicated games later.)

Note that we need to use an integer type which has at least 9 bits — one for each cell of the board. Note that this takes 4 bytes total typical machine, whereas the cell approach takes at least 9 bytes.

```
type BitBoard = u16;
pub struct Board_bitboard {
    x : BitBoard,
    O : BitBoard,
}
impl Board for Board_array {
    type Square = BitBoard;
    fn new() -> Self {
        Board_array { X:0, O:0 }
    }
    fn reset(&mut self) {
        self.X = 0;
        self.O = 0;
    }
    fn cell(&self, square: Square) -> Cell {
        if self.X&square!=0 { Some(Side::X) }
        else if self.O&square!=0 { Some(Side::O) }
        else None
    }
    fn set_cell(&mut self, square: Square, cell: Cell) {
        match cell {
            Some(Side::X) => { self.X |= square; self.O &= !square; }
            Some(Side::O) => { self.X &= !square; self.O |= square; }
            None => { self.X &= !square; self.O &= !square; }
        }
    }
}
```

2.6 Move representation

For tic-tac-toe, a move can be represented in a variety of ways, though the most natural representation will depend on the board representation. For the three board representations mentioned above, we have the following move representations corresponding

1. a pair (r, c) giving the row and column of the cell
2. an index i in the range 0..9
3. a BitBoard with a single bit set

In the case here of TIC-TAC-TOE, we will just use the Square representation as a move. (For other games, we will need a more general Move trait, but we'll postpone discussion of this until later.

We also need to include these additional elements:

- player to-move
- number of moves played
- routine to make or "unmake" a move on the board, (we'll see the need for this in the next section).

Thus we'll extend the Board trait as follows:

```
pub trait Board {
    ...
    fn moves(&self) -> Vec<Self::Move>
    fn to_move(&self) -> Option<Side>;
    fn moves_played(&self) -> usize;
    fn make_move(&mut self, m: Self::Move);
    fn unmake_move(&mut self, m: Self::Move):
}
```

Summing up this section, we have the following code. (We additionally fill in various necessary boiler-plate code for initialization, etc.)

```
// two sides playing
#[derive(Clone,Copy,Debug,Display,Eq,PartialEq)]
pub enum Side {
    X=1,
    O=2
}

// possibly empty board cell
pub type Cell = Option<Side>;

pub trait Board {
    type Square : Debug,Display,Clone,Copy,Move;
    fn new() -> Self;
    fn reset(&mut self);
    fn cell(&self, square: Square) -> Cell;
    fn set_cell(&mut self, square: Square, cell: Cell);
    fn moves(&self) -> Vec<Self::Move>
    fn to_move(&self) -> Option<Side>;
    fn moves_played(&self) -> usize;
    fn is_game_over(&self) -> Option<Option<Side>>
    fn make_move(&mut self, m: Self::Move) -> boolean;
    fn unmake_move(&mut self, m: Self::Move) -> boolean:
}

// board representations
pub struct Board_rowccol : Board {
    cells : [[Cell; 3]; 3],
    moves_played : usize,
}

struct Board_array : Board {
    cells : [Cell; 9],
    moves_played : usize,
}

type BitBoard = u16;
pub struct Board_bitboard {
    type Square = BitBoard;
```

```
    x : BitBoard,
    O : BitBoard,
    moves_played : usize,
}

impl Board for Board_bitboard {
    type Move = BitBoard;

    // initialization
    fn new() -> Self {
        Board { cells: [None; 9], to_move: Some(Side.X), moves_played: 0 }
    }
    fn reset(&mut self) {
        *self = new();
    }
    fn to_move(&self) -> Option<Side> {
        if (is_game_over().is_some()) {None}
        else if moves_played&1==0 {Some(Side::X)}
        else {Some(Side::O)}
    }
    // an inefficient but straightforward implementation:
    // iterate through integers 0..8, map to bitboard and those with empty cells
    fn moves(&self) -> Vector<Move> {
        (0..9).map(|i|1<<i).filter(|m|(self.x|self.o)&m==0).collect()
    }
    [inline]
    // use some bit operations to check for run of 3,
    // could alternatively just check for each possibility directly,
    // which could be just as good in this case, but
    fn is_win(bb: BitBoard) -> boolean {
           (bb & (bb>>1) & (bb>>2) & 0b001_001_001 != 0) // horizontal
        || (bb & (bb>>3) & (bb>>6) & 0x000_000_111 != 0) // vertical
        || (bb & 0b100_010_001) || (bb & 0b001_010_100)  // diagonal
    }
    fn is_game_over(&self) -> Option<Option<Side>> {
        if is_win(self.x) { Some(Some(Side::X)) }
        else if is_win(self.o) { Some(Some(Side::O)) }
        else if moves_played==9 { Some(None) }
        else { None }
    }
    fn make_move(&mut self, m: Move) -> boolean {
        if (moves_played&1)==0 {
            self.x |= m;
        } else {
            self.o |= m;
        }
        self.moves_played += 1;
        true
    }
    fn unmake_move(&mut self, m: Move) -> boolean {
        self.moves_played -= 1;
        if (moves_played&1)==0 {
            self.x &= !m;
        } else {
            self.o &= !m;
        }
        true
    }
}

impl Board for Board_Array {
    // initialization
    pub fn new() -> Self {
        Board { cells: [None; 9], to_move: Some(Side.X), moves_played: 0 }
    }
    pub fn reset(&mut self) {
        *self = new();
    }

    // an inefficient but straightforward implementation
    // we return None to indicate the game is still being played,
    // otherwise return Some(Some(side)) where side won
    // and Some(None) for a drawn game
    pub fn is_game_over(&self) -> Option<Option<Side>> {
        for &(s,d) in [
            (0,1),(3,1),(6,1), // horizontal
            (0,3),(1,3),(2,3), // vertical
            (0,4),(3,2),       // diagonal
            ].iter()
        {
            if self.cells[s]==self.cells[s+d]
```

```
                && self.cells[s]==self.cells[s+d+d]
                && self.cells[s].is_some()
            {
                return Some(GameResult::Win(self.cells[s].unwrap()));
            }

        }
        if self.moves_played==9 {
            // draw?
            return Some(GameResult::Draw);
        } else {
            // not over
            return None;
        }
    }

    // a highly inefficient but straightforward implementation:
    // iterate through integers 0..8 and keep those with empty cell
    pub fn moves(&self) -> Vector<Move> {
        (0..9).filter(|i| self.cells[i].is_none()).collect()
    }

    pub fn make_move(&mut self, m: Move) -> Result<Side,()> {
        // check for ended game
        if self.cells[m]!=None || self.to_move==None {
            Err(())
        } else {
            self.moves_played += 1;
            self.cells[m] = self.to_move;
            self.to_move = if self.to_move==Side.O {Side.X} else {Side.O};
            if self.is_game_over() { self.to_move = None; }
            Ok(self.to_move)
        }
    }
    pub fn unmake_move(&mut self, m: Move) -> Result<Side,()> {
    }
}
```

Chapter 3

Simple Player

In this chapter we build simple players for the games of MANCALA, MINI-CHESS, and MINI-SHOGI. This will show us the issues that can arise, but our implementation will be simplistic and naïve. Later chapters will extend and enhance the techniques used here, but these algorithms form the core of the "classical model" of game playing.

Our core algorithms will be general and will be reused for each of the games, with only game-specific modules needing to be changed for move-generation and position evaluation. We'll find that a surprisingly large number of games can be handled with general code, though tuning for optimal performance will still need to be done on a per-game basis.

We will define a general "interface" ...

```
trait Side : Clone,Copy,Debug,Display,Eq,Move,PartialEq {
    fn first_mover() -> Self;
    fn other(&self) -> Self;
    fn isnull(&self) -> boolean;
    fn nullmove() -> Self;
}
trait Move : Clone,Copy,Debug,Display,Move,Eq,PartialEq {
    fn nullmove() -> Self;
    fn is_nullmove(&self) -> boolean;
}
trait MoveResult : Clone,Copy,Debug,Display,Move,Eq,PartialEq {
    fn is_sameside(&self) -> boolean;
    fn is_other(&self) -> boolean;
    fn is_error(&self) -> boolean;
}
trait GameResult<Side:Side> {
    fn win(s:&Side) -> Self;
    fn loss(s:&Side) -> Self;
    fn draw() -> Self;
    fn is_win(&self, s:&Side) -> boolean;
    fn is_loss(&self, s:&Side) -> boolean;
    fn is_draw(&self) -> boolean;
}
trait Game {
    type Move : Move;
    type Side : Side;
    type MoveResult : MoveResult;
    type GameResult : GameResult;

    fn new() -> Self;
    fn reset(&mut self);

    fn result(&self) -> GameResult;

    fn make_move(&mut self, m: Self::Move) -> MoveResult;
    fn unmake_move(&mut self, m: Self::Move) -> MoveResult;

    fn moves(&self) -> Vec<Move>;
}
trait EvalScale : Copy,Clone,Move,Eq,PartialEq,Ord,PartialOrd {
    fn is_mate_in_n(self) -> Option<isize>;
    fn mate_in_n(n:isize) -> Self;
    fn increment_mate(self) -> Self;
    fn maximum() -> Self;
    fn minimum() -> Self;
}
```

```
trait Evaluator<G:Game> {
    type Scale : EvalScale;
    // relative to mover
    fn eval(g: &Game) -> Scale;
    fn eval_abs(g: &Game) -> Scale;
    fn eval_rel(g: &Game, s: &Game::Side) -> Scale;
}
```

Once we have the general ...

- Game, Board, Move traits
- Searchers
- Transposition Table

We'll also touch on other aspects the details of which are necessarily highly game-dependent, but common needs across games.

Hashing, textual display, debugging, FEN, perft, time-controls,

3.1 Min-max search

The basis of most game-playing programs is the "min-max" search which is based on the idea of searching exhaustively through every possible sequence of moves to the end of the game and then "backing up" the results, based on the idea that each player will play the best possible move, knowing that the other player is doing the same.

We again reiterate that a "game" for this section will mean a two-player game of perfect information and no randomness.

The fundamental observation is the following: Suppose in a given position π, we know the result (win/loss/draw) of all possible resulting positions from a move (all from the poin of view of the player to move in the position π. Then we know the value of the position:

- If one of the moves leads to a win, then the position is a win: we can simply choose (one of) the winning moves. (In particular, we should choose the move that has the shortest path to a win. This will be ...)
- If none of the moves leads to a win, but one of the moves leads to a draw, then the position is a draw: we choose a move leading to a draw.
- Finally, if none of the moves leads to a win or a draw, then this means that every move leads to a loss, hence the position is a loss.

Theoretical aside:

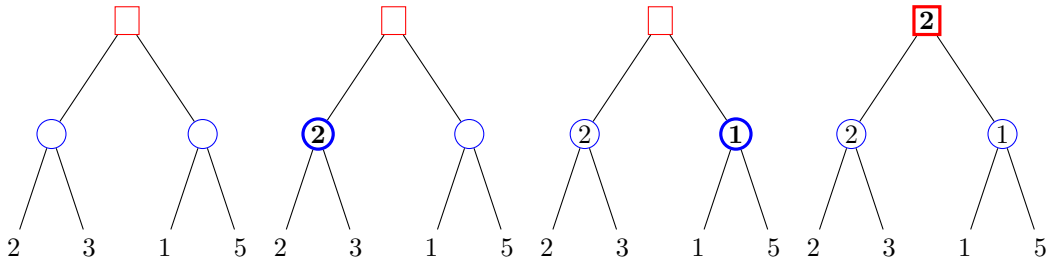
First define a *finite game* to be a game such that at any position there is a finite number of possible moves and such that there are no infinite sequence of plays (every game has finite length). We include in a game "position", any necessary state to determine the result of that position during the play of a game. For example, for chess, where a third repetition of a position is ruled a draw (with the game ending), then we have to include the previous game up to that point in the game "state" of the position.

First, we prove that each possible game position π has a maximum possible length of play $l(\pi)$ from that position on. suppose, towards a contradiction, that there were a position π with no bound on the length of play from that position on, i.e. $l(\pi) = \infty$. Now, there are finitely many possible moves from that position, say $m_1, \ldots, m_k$, and if each move m_i resulted in a position $m_i : \pi$ with a maximum game length $l(m_i : \pi) = n_i$, then $n = 1 + \max\{ n_1, \ldots, n_k \}$ would be a bound on the length of any possible game from π. Thus there must be at least one of the moves m_i with unbounded game length after making the move m_i, i.e. $l(m_i : m_i) = \infty$. But we can then repeat this argument to find a move m_{i_1} in the position $m_i : \pi$ with $l(m_{i_1} : m_i : \pi) = \infty$. If we keep repeating this, we find an infinite sequence of moves $m_i, m_{i_1}, m_{i_2}, \ldots, m_{i_n}, \ldots$ with each move a possible move in the resulting position of moving all previous moves. But this then gives us an infinitely long sequence of plays in the game, contradicting the assumption that we are dealing with a finite game. This contradiction shows that our assumption on the existence of position π must be false, and hence every possible position π has a maximum possible game length $l(\pi)$.

(Note that this means that the number of possible positions is finite and the total number of possible games is also finite, though usually mind-bogglingly astronomical.)

Next, we prove that any finite game has a definite result (either win for one of the players or draw.) In fact, we will prove that every game position has a definite result. We prove this by (total) induction on the maximum game length from a position. For the base case we take all game-end positions π_e: the game has ended, so the game length from that position on is $l(\pi_e) = 0$, and we of course know the definite game result (since the game is ended, the result $\rho(\pi_e)$ is clearly known!) Next, suppose we have a game position π such that for every other game position π' with shorter maximum game length (i.e. $l(\pi) > l(\pi')$), we know, by the induction hypothesis, the result $\rho(\pi')$. Then we apply the fundamental observation from above: let $m_1, \ldots, m_k$ be the possible moves from π, then we know the result of the position after making the move m_i, $\rho(m_i : \pi)$ for each move. So if one of the results is a win (for the player to move in π), then the result of π is a win. If no result is a win, but one is a draw, then the result is a draw, and finally, if all results are losses, then the result is a loss for the player to move. Thus by induction, we have shown that every game position has a definite result (depending on the result of subsequent positions.)

Now for almost any game we'll find that actually searching to the end of the game is infeasible, due to the combinatorial explosion of the game tree.




```
pub fn search_negamax_pruning<Board:GameBoard,Eval:Evaluator<Board>>(b: &mut Board, eval: &eval, ss: &mut SearchStats, depth: isize) -> Board::Sca
    ss.nodes += 1;
    // check if game is over
    match b.is_terminal() {
        Some(s) => { return s; }
        None => { }
    }
    // check if hit depth bound
    if depth<=0 {
        let e = eval.evaluate(b, ss, depth);
        return Some(e);
    }
    let tomove = b.to_move();
    let other = tomove.other();
    let ml = b.moves();
    // initialize to loss
    let mut best_so_far = Board::Scale::worst(tomove);
    for m in ml {
        b.make_move(m);
        let result = search_negamax_pruning(b, eval, ss, depth - 100);
        b.unmake_move(m);
        best_so_far = Board::Scale::max(tomove, best_so_far, result);
    }
    return best_so_far;
}
```

```
pub fn search_negamax_ttable<Board:GameBoard,Eval:Evaluator<Board>>(b: &mut Board, eval: &eval, ss: &mut SearchStats, depth: isize) -> Board::Scal
```

```
template <class TMove, bool tt>
result minimax(Board<TMove>& b) {
  hash h;
  if (tt) {
    h = b.Hash();
    if (ttable[h].type == EXACT_BD) {
      ++tthits;
      return ttable[h].res;
    }
```

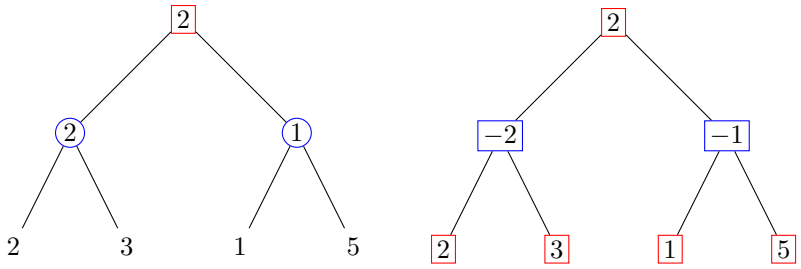
```
  }
  result eval = b.Result();
  ++nodes_searched;
  if (IS_TERMINAL(eval)) {
    if (tt) { ttable[h].res = eval; ttable[h].type = EXACT_BD; }
    return eval;
  }
  result best_v;
  if (b.ToMove() == XX) {
    std::vector<TMove> moves;
    best_v = OO_WIN_IN(-1);
    int num_m = b.Moves(moves);
    for (int m=0; m<num_m; ++m) {
      b.MakeMove(moves[m]);
      result res = minimax<TMove,tt>(b);
      b.UnMakeMove(moves[m]);
      res = PLY_UP_RESULT(res);
      if (res > best_v) best_v = res;
    }
  } else {
    std::vector<TMove> moves;
    best_v = XX_WIN_IN(-1);
    int num_m = b.Moves(moves);
    for (int m=0; m<num_m; ++m) {
      b.MakeMove(moves[m]);
      result res = minimax<TMove,tt>(b);
      b.UnMakeMove(moves[m]);
      res = PLY_UP_RESULT(res);
      if (res < best_v) best_v = res;
    }
  }
  if (tt) { ttable[h].res = best_v; ttable[h].type = EXACT_BD; }
  return best_v;
}
```

3.2 Negamax search

Most real game-playing programs don't use the min-max formulation for their search, but rather use the so-called "negamax" form. This gives a function which is equivalent (meaning it will search exactly the same nodes and will return exactly the same result) but requires less code. Instead of requiring two different functions (which are nearly identical), one for the Min player and one for the Max player, negamax search uses a single search routine which is always trying to maximize the outcome from the point of view of the player whose turn it is (rather than only player consistently trying to maximize and the other player trying to minimize).

Another way to think about it is that min-max search uses an absolute evaluation whereas negamax uses a relative evaluation: evaluating the position from the point-of-view of the person to move. The relative evaluation is closer to how we might actually think about the position if we were playing a game: we will think that it is good for us or bad for us. The absolute evaluation is locked to thinking in terms of a single player.

Generally negamax formulations, though sometimes confusing when first getting into this area, tend to lead to cleaner and more natural code. It eliminates a lot of if-then checks for which side is moving in the search code. (It can add a tiny complication to the board evaluation which is often naturally coded in absolute terms.)



3.3 Transposition tables

We can reduce significantly the amount of effort spent in min-max style searching by recognizing that many positions will arise multiple times in the course of our search — for example, one might move a piece twice over the course of two turns, ending up at the same position, but with a different intermediate step.

Part II

The Classical Model

Chapter 4

Board representation, move generation

Here

4.1 Array

The most obvious representation of games set on a grid of cells is to use an array representation, with each element of the array indicating which piece, if any, in the cell. This can be a two-dimensional array, matching the layout of the game-board, for example. Though, it is often more convenient to use a one-dimensional array, with the mapping

Example, chess:

```
// basic setup
pub enum Piece { Pawn, Knight, Bishop, Rook, Queen, King };
pub enum Color { White, Black };
```

Matrix or two-dimensional representation:

```
// definition
pub struct Board_matrix { cells : [[Option<(Color,Piece)>; 8]; 8], };
// creation
let mut b = Board_matrix { cells : [[None; 8]; 8], };
// put white knight on c2 square
b.cells[1][2] = Some((Color::White,Piece::Knight));

// definition
pub struct Board_array { cells : [Option<(Piece,Color)>; 8 * 8], };
// creation
let mut b = Board_array { cells : [None; 8*8], };
// put white knight on c2 square
b.cells[1*8 + 2] = Some((Color::White,Piece::Knight));
```

4.2 Piece lists

Piece lists are useful for larger games where the the board density (ratio of number of pieces to squares on the board) is low. Instead of directly representing the board, we simply record for each piece, where it is on the board. It is often convenient to also represent the board explicitly, but can still use piece-lists in addition to speed up move generation or other calculations.

The advantage here is that when doing move generation we don't have to scan the entire board looking for pieces to move — we've got the pieces directly accessible and ready to iterate over. If we organize the lists correctly, we can also get the advantage of generating moves for certain kinds of pieces before or after other kinds of pieces.

For example, in 5x6 minichess there are 30 squares on the board and the starting position has 20 pieces, for a starting density of $\frac{20}{30} = 0.\bar{6}$, about 67%. On the other hand, in 10x10 amazons, there are 100 squares on the board and there are only 8 active pieces on the board, with a density of only $\frac{8}{100} = 0.08$, exactly 8%. Thus, it would be quite wasteful in amazons for move generation to scan through 100 squares to find the mere 4 pieces for that side out of the full 100 squares on the board. Better would be to simply store directly the locations of the queens for each side in a short list.

4.3 0x88

This is a trick which works for 8×8 boards but does not nicely generalize to other sizes of boards (or to non-rectangular boards). It is fairly popular in chess playing programs. The idea is to note that representing a number between 0 and 7 requires exactly 3 bits and that when generating moves for chess, any putative moves that would take a piece off of the board would take them to a coordinate which sets the fourth (or "8") bit. Thus, if we represent an x-y coordinate in bits as "0rrr0ccc" then we can detect an illegal coordinate simply by xor'ing with 0x88 = b10001000.

Note that if we then use a grid of cells of size 8x16 = 128, then the valid squares lie on the "left hand" 8x8 grid, so we can use the bit representation as direct index into the array.

For example, using 8-bit integer arithmetic, we have the following, (we mark any invalid squares as **):

Square = 0x88	right 1	left 1	down 1	up 1	knight (up 2, left 1)
Δ	+1 = 0x01	-1 = 0xFF	-16 = 0xF0	+16 = 0x10	+31 = 0x1F
a1 = 0x00	b1 = 0x01	*** = 0xFF*	*** = 0xF0*	a2 = 0x10	*** = 0x1F*
b7 = 0x61	c7 = 0x62	a7 = 0x60	b6 = 0x51	b8 = 0x71	*** = 0x80*
g2 = 0x16	h2 = 0x17	f2 = 0x15	g1 = 0x06	g3 = 0x26	f4 = 0x35
h8 = 0x77	*** = 0x78*	g8 = 0x76	h7 = 0x67	*** = 0x87*	*** = 0x96*

Notice that all invalid moves end up with an index with highest bit set in one of its two nibbles (4-bit segment).

4.4 Mailbox

4.5 Bit-boards

Bit-boards are a popular approach which are designed to take advantage of the fact that bit-wise operations are performed in quickly and in parallel by the computer. The match of the 64 squares of the chess-board, for example, match very nicely with the 64 bits in a machine work in a modern 64-bit processor.

The basic idea is that for each piece type, we maintain a vector of bits, one bit for each square on the board, indicating if there is a piece of that type on the board.

4.5.1 Rotated bit-boards

4.5.2 Kindergarten bit-boards

4.5.3 Magic bit-boards

Chapter 5

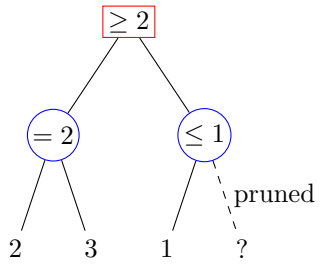
Search

In this chapter we investigate more deeply the issue of searching,

5.1 Alpha-beta pruning

Alpha-beta ($\alpha\beta$) pruning is a way to get the same result as min-max (or negamax) searching but with significantly reduced amount of search. The idea is that we we don't really need to search every node, but can ignore significant parts of the game-tree without affecting the accuracy of our result.

The basic idea is illustrated in the following tree:



where we can ignore the (subtree off the) final branch by the following reasoning: the max player can guarantee a result of *at least 2* by making the first move, while if he makes the second move, the min player can ensure a result of *no better than 1* by taking her first move, irrespective of the value of her second move. Thus the max player would not take his second move and we do not need to search any further.

The effectiveness of $\alpha\beta$-pruning is extremely sensitive to the order in which we search the different possibilities at a given node. The best case is when the first node we search at each node is the best possible move from that position. Unfortunately, we typically do not know the best move (if we know the best move, there would be no need to search in the first place!)

Worst-case scenario (9 leaf nodes evaluated):

Best-case scenario (5 leaf nodes evaluated):

The theoretical improvement we can expect from full width search to optimal $\alpha\beta$-search was computed by Knuth and [...] which takes a tree with uniform branching factor b of depth L from b^L nodes for complete search to $b^{\lceil L/2 \rceil} + b^{\lfloor L/2 \rfloor}$ which translates roughly into allowing us to search to *twice* the depth for the same amount of time!

5.2 Principal variation

The *principal variant* or PV is the expected optimal line of play, where both players are ...

5.3 Transposition tables

Transposition tables are critical for efficient performance.

A crucial element of using transposition tables is the hash function — the function which takes a position and

5.3.1 Zobrist hashing

The standard approach to hashing was ... by Zobrist in ... It is a clever approach which applies easily to many different games and implementation styles.

The core idea is to assign a (pseudo-)random number to each individual color, piece, and square combination. For example, in chess, we'll have a number (code) for white knight at a1, white knight at a2, ..., black rook at h8. Then, to compute the hash of a position, we simply take the "exclusive or" (or "xor" or $\oplus$) of this code for each piece on the board. [1]

Now, the xor operator has the beautiful property that $(a \oplus b) \oplus b = a$ for any a and b. In other words, if I take a number and xor any other number with it twice, I end up with my original number. Additionally, xor operates bit-by-bit on an integer, so it is a very fast operation to execute by the CPU.

5.3.2 BCH hashing

5.4 Iterative deepening

In iterative-deepening, we do a series of searches to increasing depth, rather than deciding beforehand a fixed depth to search

As we will see later, using iterative deepening makes for good use of time — allowing us to do the best search we can in a given time interval, even though ...

Although re-searching the same position multiple times may seem like a ridiculous waste of time, it is actually quite efficient and it is possible for an iterative-deepened search to a given depth to actually take *less* time than a single search directly to that depth! The main reason for this is that with the use of a transposition table, we are able to use the results of shallower searches to sort the moves when doing a deeper search, thus generally achieving better results for $\alpha\beta$-pruning. Another point to note is that often shallower search results stored in the transposition table are re-used in the deeper search allowing for savings in the deeper search; that is, sub-trees can often be reached via different move sequences of different lengths.

5.5 Quiescence search

The idea behind quiescence search is to end search lines only at "quiet" positions. This stabilizes evalations and helps to avoid the so-called "horizon effect". It will also add to the tactical strength.

As an example of the horizon effect, consider the following chess position:

...

If we use a standard fixed-depth alpha-beta search we get the following lines ... notice that it isn't until we reach a search depth of ... that the search finally realizes that snagging the pawn isn't safe. On the other hand using quiescence extensions we get ...

The definition of quiet or quiescent position depends on the game and the nature of the search can vary.

For CHESS, a common approach to quiescence search is to search all captures, checks, and check responses. Note that it is important to allow a "stand pat" option also when searching (where a side can just accept the static evaluation instead of following further in the quiescence search.

Following captures is a relatively safe and for many games, is a self-limiting approach, since each capture can reduce the number of pieces on the board. However, this is not universally true. For example, in SHOGI, doing a quiescence search which searched captures as well as drops would lead to horrendous search explosions

[1] The xor of individual bits is defined by $0 \oplus 1 = 1 \oplus 0 = 1$ and $0 \oplus 0 = 1 \oplus 1 = 0$. The xor of two n-bit integers is simply the bit-by-bit xor of the individual bits in the integer representations. You may check that, for example, $1 \oplus 3 = 2$ and $1 \oplus 4 = 5$.

as capture-drop sequences can be unending (with each captured piece being reused as a possible drop by the capturing side.)

5.6 Null-move pruning

Null-move pruning is a forward-pruning technique based on the assumption that it is always better to make a move than to "pass". It can significantly reduce the number of nodes searched with a minor reduction in the strength (due to the occasional ...)

5.6.1 Adaptive null-move pruning

For chess it was ... $R = 3$ was somewhat over-aggressive in pruning, while $R = 2$ is

5.6.2 Verified null-move pruning

5.7 Other heuristics

Here we cover some heuristic pruning techniques which are "unsound" — unlike $\alpha\beta$-pruning, they are not guaranteed to return the exact min-max value. However, in practice they are extremely effective.

5.7.1 Killer moves

5.7.2 History heuristic

5.7.3 Razoring

5.7.4 Futility pruning

Extended futility pruning

5.7.5 Search extensions

Chapter 6

Evaluation

In this chapter we discuss the evaluation of

6.1 Material

The most significant of components for evaluation of positions for most chess-like games is material: the positions ...

6.2 Piece-square bonus

It is known that in CHESS, for example, it is generally advantageous for knights to occupy central squares and that usually it is bad to have a knight in a corner square. This can be generalized to giving a bonus or malus for each piece being in any particular position on the board.

Similarly, certain general tactics can be "encouraged" by giving bonuses and maluses:

- General piece centralization
- Pawn pushing towards promotion squares
- Bishops on long diagonals (fianchetto)
- ...

6.3 Mobility

Mobility describes the available moves by each side. The simplest approach would simply count the possible moves in the given position. More sophisticated measures include counting attacks differently than simple moves, "x-ray" moves (moves or attacks which are blocked by one's own pieces but which ... TODO...

6.4 Structure, Shape

For games such as CHESS or SHŌGI, we have "static" features of the position — features of the position which tend to persist over longer stretches of the game, features which involve the interaction of many pieces, ...

For CHESS, this would include pawn-structure (e.g. isolated pawns, double pawns, passed pawns, protected passed pawns, etc.) whereas

For SHOGI we have "shape" TODO...

6.5 Square-control bonus

6.6 Hanging pieces, defended pieces

6.6.1 SEE

6.6.2 SOMA

6.6.3 SUPER-SOMA

Part III

The Monte Carlo Revolution

Chapter 7

MCTS

Monte-carlo tree search (MCTS) is a relatively new approach to solving games that has become extremely effective at certain kinds of games, such as GO, where traditional $\alpha\beta$-approaches have difficulties, both due to the high branching factor and to the challenge of constructing a good evaluation function.

7.1 UCB

Chapter 8

Extensions

8.1 AMAF

8.2 RAVE

8.3 Incorporating domain knowledge

Part IV

Other

Chapter 9

UI

In this chapter we

9.1 GGS

9.2 UCI

9.3 UGI

9.4 UGE

9.5 XBoard

9.6 Fritz

Chapter 10

Time management

In this chapter we

10.1 Time controls

There are many different time-controls that exists for different games. For example, there may be a fixed amount of time for each side for all moves made in the game, or there may be a set amount of time for each move made (where the unused time may be accumulated for later (with a possible cap) or may simply be lost.)

- Chess: In chess, there are a variety of popular time-controls.
- Go:
- Arimaa: from `http://arimaa.com/arimaa/learn/matchRules.html` and where the 2011 Arimaa Open Classic and World Championship is being played to `60s/5m/75/0/4h/4m` for the Swiss preliminary and `90s/5m/75/0/6h/5m` for the FDE final.

  The time specification `60s/5m/75/0/4h/4m` means, for example,

  - 60 seconds per move
  - 5 minutes in the starting reserve
  - 75% of remaining move time gets added to reserve
  - 4 hours maximum limit for the game
  - 4 minutes maximum turn time

  In other words: On each turn a player gets 60 seconds per move. If a move is not completed in this time then the reserve time may be used. There is a starting reserve of 5 minutes. If the reserve time is used up and the player has still not made a move then the player will lose on time. If the move is made in less than 60 seconds then 75% of the remaining move time is added to the reserve. A players turn may not exceed more than 4 minutes regardless of how much time is in reserve. If the game is not completed within 4 hours, it will be stopped and the winner determined by score.
- Computer Olympiad: in the ICGA Computer Olympiads, the most common time control is ...

10.1.1 Arimaa Time Controls Specification

The time control used for Arimaa is given by a 6-part specification `M/R/P/L/G/T` where

- `M` is the number of minutes:seconds per move; required
- `R` is the number of minutes:seconds in reserve; required
- `P` is the percent of unused move time that gets added to the reserve; optional defaults to 100

- `L` is the number of minutes:seconds to limit the reserve; 0 means no limit; optional; defaults to 0
- `G` is the number of hours:minutes after which time the game is halted and the winner is determined by score. `G` can also be specified as the maximum number of moves by ending with 't'; 0 means no limit; optional; defaults to 0
- `T` is the number of minutes:seconds within which a player must make the move; 0 means no limit; optional; defaults to 0

Different time units for any of the time control fields can be specifed by adding one of the following letters after the numbers. In such cases the letter serves as the separator and `:` should not be used.

- `s` - seconds
- `m` - minutes
- `h` - hours
- `d` - days

For example: `24h5m10s/0/0/0/60d` means 24 hours, 5 minutes and 10 seconds per move and the game must end after 60 days. Such a time control may be used in a postal type match.

- The game time parameter (`G`) can also be specified in terms of maximum number of turns each player can make by adding the letter `t` after the number.
- On each turn a player gets a fixed amount of time per move (M) and there may be some amount of time left in the reserve (R).
- If a player does not complete the move within the move time (M) then the time in reserve (R) is used. If there is no more time remaining in reserve and the player has not completed the move then the player automatically loses. Even if there is time left in the move or reserve, but the player has not made the move within the maximum time allowed for moves (T) then the player automatically loses.
- If a player completes the move in less than the time allowed for the move (M), then a percentage (P) of the remaining time is added to the players reserve. The result is rounded to the nearest second. This parameter is optional and if not specified, it is assumed to be 100%.
- An upper limit (L) can be given for the reserve so that the reserve does not exceed L when more time is added to the reserve. If the initial reserve already exceeds this limit then more time is not added to the reserve until it falls below this limit. The upper limit for the reserve is optional and if not given or set to 0 then it implies that there is no limit on how much time can be added to the reserve.
- For practical reasons a total game time (G) may be set. If the game is not finished within this allotted time then the game is halted and the winner is determined by scoring the game. This parameter is optional and if not given (or set to 0) it means there is no limit on the game time. Also instead of an upper limit for the total game time, an upper limit for the total number of turns each player can make may be specified by adding the letter 't' after the number. After both players have taken this many turns and the game is not finished the winner is determined by scoring the game.
- For games which use a time per move of less than 1 minute, both players are always given 1 minute of time to setup the initial position in the first move of the game. If the setup is not completed in 1 minute then the reserve time (R) is also used. The unused time from the setup move is not added to the reserve time.

Some examples:

- Example 1: `0/5` means 0 minutes per move with 5 minutes in reserve (per player). This is equivalent to G/5 in Chess; it means each player has a total of 5 minutes of time to play. If a player runs out of time before the game is over, the player loses. This is known as Blitz or "Sudden Death" time control in Chess.

- Example 2: `0:12/5` means 12 seconds per move with 5 minutes in reserve and all of the unused time from each move is added to the reserve time. It is similar to "5 12" in Chess which means "Game in 5 minutes with a 12 second increment". After each move 12 seconds is added to the remaining time. This is known as Incremental time control in Chess.
- Example 3: `3/0` means 3 minute per move and no reserve time, but 100 percent of the unused time for each move is added to the reserve. This guarantees that each player will make at least 40 moves in 2 hours. This is similar to the "40/2" Quota System time control used in Chess.
- Example 4: `0:30/5/100/3` means 30 seconds per move with 5 minutes in reserve and 100% of the unused time from each move is added to the reserve time. When the reserve already exceeds the limit, more time is not added to it. When the reserve falls below 3 minutes more time can be added to it, but the reserve is capped at 3 minutes.
- Example 5: `4/2/50/10/6` This means 4 minutes per move with a starting reserve of 2 minutes. After the move 50% of the time remaining for the move (rounded to the nearest second) is added to the reserve such that it does not exceed 10 minutes. There is a limit of 6 hours for the game after which time the game is halted and the winner is determined by score.
- Example 6: `4/4/100/4/6` This means 4 minutes per move and a starting reserve of 4 minutess. 100% percent of the unused move time gets added to the reserve such that it does not exceed 4 minutes. There is a time limit of 6 hour for the game after which the winner is determined by score.
- Example 7: `4/4/100/4/90t` This is the same as above, but the game ends after both players have made 90 moves. Thus it ends after move 90 of silver is completed.
- Example 8: `4/4/100/4/90t/5` This is the same as above, but the players may not take more than 5 minutes for each turn even if there is still time remaining in reserve.

10.2 Estimation of remaining time

10.3 Control

10.4 Pondering

Pondering is where the computer uses the time during the opponent's move to think.

The standard approach is to assume that the opponent will make the move given as the response in the principal variation (PV) and to start "thinking" based on this assumption. As long as this correctly predicts the move 50% or more it will pay off well. (If the opponent makes a different move, we have to restart the search and the effort is lost.)

10.5 Example: Crafty

Here we describe the method for time-control used in Crafty (version xx.xxx).

10.6 Example: Gerbil

The open-source chess engine Gerbil supports a basic sudden-death time-control with or without an increment; a fixed time per move; and moves-per-period time-control.

It has variables `base` and `extra` in the engine.

For a fixed time t per move, it is quite simple: use all the given time `base` $= t$ and there is no extra time `extra` $= 0$.

For sudden-death with no increment, it uses

```
    case tctINCREMENT:
        if (pcon->tc.tmIncr) {
            TM    tmTarget;

            //    An increment time control tries not to go below the increment
            //    plus 30 seconds, or 6 times the increment, whichever is more.
            //
            //    If it finds itself below that, it will the increment minus
            //    1/10 of the time by which it is under the target.
            //
            //    If it is over that, it will use the increment plus 1/20 of
            //    the time by which it is over the target.
            //
            //    So the program will drift slowly down to the target, but if
            //    it gets a ways below it, it will tend to come up quickly.
            //
            //    There are a zillion different increment time controls, and
            //    sometimes I could end up with a stupid base time.  I try hard
            //    to make sure the base time doesn't go below zero, which would
            //    be very bad since my time quantity is an unsigned value.
            //
            //    I will try hard to use >= half of the increment.  I will also
            //    make sure to not use more than half the remaining time (which
            //    is taken care of by the code at "lblSet".
            //
            tmTarget = pcon->tc.tmIncr + 30000;
            if (tmTarget < 6 * pcon->tc.tmIncr)
                tmTarget = 6 * pcon->tc.tmIncr;
            if (pcon->ss.tmUs >= tmTarget)
                tmBase = pcon->tc.tmIncr + (pcon->ss.tmUs - tmTarget) / 20;
            else    // The following statement can go negative on an unsigned.
                tmBase = pcon->tc.tmIncr - (tmTarget - pcon->ss.tmUs) / 10;
            Assert(sizeof(long) == sizeof(TM));
            if ((long)tmBase < (long)pcon->tc.tmIncr / 2)
                tmBase = pcon->tc.tmIncr / 2;
        } else {
            //
            //    A zero-increment time control will use 1/30 of the remaining
            //    time down to 10 minutes, 1/40 down to one minute, and 1/60
            //    after that.
            //
            if (pcon->ss.tmUs >= 600000)        // 10 minutes.
                tmBase = pcon->ss.tmUs / 30;
            else if (pcon->ss.tmUs > 60000)    // 1 minute.
                tmBase = pcon->ss.tmUs / 40;
            else
                tmBase = pcon->ss.tmUs / 60;
        }
        //    This code dummy-checks "tmBase", assigns "tmExtra", then assigns
        //    an end-time based upon "pcon->ss.tmStart".
        //
        //    First I check to see if I'm scheduled to eat more than half of
        //    my remaining time.  I put a ceiling at that amount, right off.
        //
        //    Next, I'll assign some emergency time.  This is 3x the base time,
        //    with a ceiling on base + extra of 1/2 the remaining time.  I will
        //    also allocate no emergency time if I have < 30 seconds on the
        //    clock.
        //
        //    So there will be plenty of "extra" time if there is time left,
        //    otherwise there is little or none.
        //
lblSet:    if (tmBase > pcon->ss.tmUs / 2)
            tmBase = pcon->ss.tmUs / 2;
        tmExtra = (pcon->ss.tmUs < 20000) ? 0 : tmBase * 3;
        if (tmBase + tmExtra > pcon->ss.tmUs / 2)
            tmExtra = pcon->ss.tmUs / 2 - tmBase;
        pcon->ss.tmEnd = pcon->ss.tmStart + tmBase;
        pcon->ss.tmExtra = tmExtra;
        break;
    case tctFIXED_TIME:
        tmBase = pcon->tc.tmBase;        // Use *all* the time.
        pcon->ss.tmEnd = pcon->ss.tmStart + tmBase;
        pcon->ss.tmExtra = 0;            // No emergency time.
        break;
    case tctTOURNEY:
        movMade = pcon->gc.ccm / 2 + pcon->gc.movStart - 1;
        while (movMade >= pcon->tc.cMoves)
            movMade -= pcon->tc.cMoves;
        movLeft = pcon->tc.cMoves - movMade;
        Assert(movLeft >= 1);
```

```
        //
        //    This expression is kind of nasty.  It is:
        //
        //        Time / (3/4 x Moves + 3)
        //
        //    The point of the "3" is to make sure there's some extra time pad
        //    at the beginning, and the point of the 3/4 is to make sure that
        //    earlier moves take a little longer per move.
        //
        //    This expression has been reduced to ...
        //
        //        (4 * Time) / (3 * Moves + 12)
        //
        //    .. by multiplying the top and the bottom by 4.  This helps out
        //    with the integer math.
        //
        tmBase = (4 * pcon->ss.tmUs) / (3 * movLeft + 12);
        goto lblSet;
    }

//    This is called a whole bunch of times via the engine/interface protocol.
//    Given what is known about the time control, the current time, and the
//    search depth, this may set "ptcon->fTimeout", which will eventually finish
//    the search.

//    This routine has a built-in defense against cases where someone might try
//    to force a move before the first ply has been fully considered.  It will
//    simply ignore such cases.

void VCheckTime(PCON pcon)
{
    TM    tmNow;

    if (pcon->ss.plyDepth == 1)    // Can't time out before a one-ply
        return;                    //  search is finished.
    if (pcon->smode != smodeTHINK)
        return;
    if ((tmNow = TmNow()) >= pcon->ss.tmEnd) {
        //
        //    Time has expired.  Check for a fail-high or fail low and try to
        //    add some time.  Don't add any time if we're already dead lost and
        //    just slowly collapsing, or if we are won and are probably failing
        //    high repeatedly.  The standard for "won" is a little greater than
        //    for "lost".
        //
        if ((pcon->ss.prsa != prsaNORMAL) && (pcon->ss.val < valROOK) &&
            (pcon->ss.val >= -valMINOR))
            if (pcon->ss.tmExtra < 1000) {
                pcon->ss.tmEnd += pcon->ss.tmExtra;
                pcon->ss.tmExtra = 0;
            } else {
                pcon->ss.tmEnd += pcon->ss.tmExtra / 2;
                pcon->ss.tmExtra -= pcon->ss.tmExtra / 2;
            }
        if (tmNow >= pcon->ss.tmEnd)
            pcon->fTimeout = fTRUE;
    }
}
```

10.7 Example: Glaurung

As an interesting case-study, we examine the logic used by the open-source chess engine Glaurung. (From examining the source code.)

The basic timing method involves three variables to control the time spent on a search: `max_time` — the target time to spend on the search, `extra_time` — a possible amount of extra time to search in certain cases, and `absolute_max_time` — an absolute maximum, which will abort the search when exceeded. It supports sudden-death (with or without increment) or moves-per-period time-controls.

For sudden-death time-control with t remaining time (in milliseconds) and no increment, it sets:

$$\begin{aligned} \texttt{max\_time} &= t/40 \\ \texttt{absolute\_max\_time} &= t/8 \end{aligned}$$

If there is an increment of i milliseconds, then it uses

$$\begin{aligned} \texttt{max\_time} &= t/30 + i \\ \texttt{absolute\_max\_time} &= \max(t/4, i - 100) \end{aligned}$$

For moves-per-period style of time-control, it sets

$$\begin{aligned} \texttt{max\_time} &= t/\min(\texttt{moves\_to\_go}, 20) \\ \texttt{absolute\_max\_time} &= \min(4t/\texttt{moves\_to\_go}, t/3) \end{aligned}$$

unless there is only a single move left to go in which case it uses

$$\begin{aligned} \texttt{max\_time} &= t/2 \\ \texttt{absolute\_max\_time} &= \min(t/2, t - 500) \end{aligned}$$

Glaurung uses an iterative-deepening search strategy, and for the first 5 iterations, it sets `extra_time` to zero, otherwise (for the 6th and higher iterations) if the best move has changed in the last two iterations it will set

$$\texttt{extra\_time} = \texttt{bmcc}_{\text{it}} \times (\texttt{max\_time}/3) + \texttt{bmcc}_{\text{it}-1} \times (\texttt{max\_time}/6)$$

where $\texttt{bmcc}_{\text{it}}$ is the "best-move change-count" — the number of times that the best move has changed in the it'th iteration.

Glaurung will stop a search early if there seems to be an "easy" move (a move which is clearly better than the rest). If it has done at least 5 iterations and:

```
// Stop search early if one move seems much better than the rest:
if(RSI->iteration >= 5 && RSI->easymove &&
   ((RSI->root_moves.moves[0].cumulative_nodes > (RSI->nodes*85)/100 &&
     get_time() - RSI->start_time > RSI->max_time/16) ||
    (RSI->root_moves.moves[0].cumulative_nodes > (RSI->nodes*99)/100 &&
     get_time()-RSI->start_time > RSI->max_time/32)) &&
   !RSI->infinite && RSI->thinking_status!=PONDERING) {
  break;
}
```

The search will be ended if most of `max_time` has been used at the end of an iteration (because there wouldn't be time to complete the next iteration in any case.) Specifically if the time used so far exceeds $(\texttt{max\_time} + \texttt{extra\_time}) \times 80/128$.

During the search, the timeout is

```
void check_for_timeout(void) {
  int t;
  static int last_info_time;

  t = get_time() - RSI->start_time;
  if(t < 1000) last_info_time = 0;
  else if(t - last_info_time >= 1000) {
    mutex_lock(IOLock);
    printf("info nodes " llu_format " nps " llu_format " time %d hashfull %d\n",
           RSI->nodes, (RSI->nodes*1000ULL)/((uint64) t), t, hashfull());
    mutex_unlock(IOLock);
    last_info_time = t;
    if(Options->currline) Threads[0].print_currline = true;
  }
  if(RSI->thinking_status != PONDERING && RSI->iteration >= 2 &&
     ((!RSI->infinite &&
       (t > RSI->absolute_max_time
        || (RSI->root_moves.current == 1 && t > RSI->max_time+RSI->extra_time)
        || (!RSI->fail_high && t > 6 * (RSI->max_time+RSI->extra_time)))) ||
      (RSI->node_limit && RSI->nodes >= RSI->node_limit) ||
      (RSI->exact_time && t >= RSI->exact_time)))
    RSI->thinking_status = ABORTED;
}
```

Chapter 11

Advanced Search

In this chapter we investigate more deeply the issue of searching,

11.1 Trappy min-max

This is a game-independent method described in [GR] for automatically playing in a way to set traps for the opponent — based on the idea that if you search more deeply than your opponent you can identify misleading lines of play (when the best-move to play at a shallower search depth is given a much lower evaluation at a deeper search depth). You may then choose a non-optimal move as long as it is not too much worse than the best move, but for which the obvious reply by the opponent is actually much worse ...TODO...

This kind of approach is best used against "fallible" opponents (human players, for example) or by brute-force deep-search based programs against shallower searching ... blah ...

11.2 Negascout

11.3 Null-window search

11.4 Aspiration search

11.5 PVS

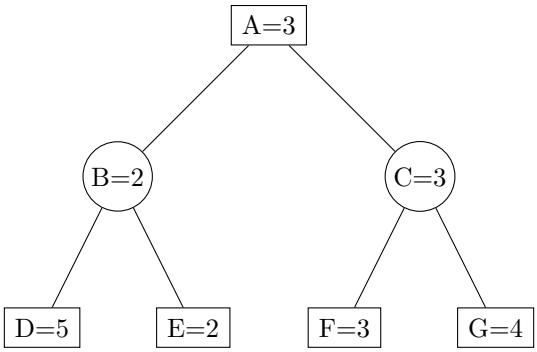
11.6 MTD(f)

11.7 Conspiracy number

Conspiracy number search is a best-first search method. It is based on the idea of searching to a given level of confidence, measured in the number of nodes we've evaluated which would need to "conspire" in being wrongly evaluated in order to change our evaluation of the root. (We may also think of this as a measure of robustness — how many nodes could be systematically mis-evaluated without affecting our result.)

If we have a search tree, there will be a certain number of "leaf" nodes that we have reached in our search. (That is, nodes where we use the evaluation function to get an evaluation of the position.) Define the conspiracy number of *any* node in the tree to be the minimum number of *leaf* nodes that would need to change their value in order to change the value of that given node. In fact, for different evaluation values, there may be different numbers ... blah blah....

Standard example repeated *ad nauseum*:



where for node A we have

value	cn	nodes which have to change
1	2	(D or E) and (F or G)
2	1	F or G
3	0	-
4	1	E or F
5	1	E
6	2	(D and E) or (F and G)

11.7.1 Alpha-beta conspiracy search

In [**?**], McAllester and Yuret describe a variant of $\alpha\beta$-search which is derived from conspiracy number search but is a depth-first rather than best-first search as in the original conspiracy numbers algorithm.

The idea incorporates two "depth" numbers $d_{\max}$ and $d_{\min}$ instead of a single one as in standard $\alpha\beta$-search. It also includes an additional list of lists of static values of siblings of a given node and its ancestors.

We also define a number `delta`, the "singular margin", as well as a real-valued, increasing function $S(k)$ such that $S(0) = 0$ and $\lim_{k\to\infty} S(k) = 1$ and which gets close to 1 before the typical branching factor b. (For example, we might take

$$S(k) = \begin{cases} 1 & \text{if } k \geq b \\ 1 - \left(\frac{b-k}{b}\right)^{\gamma} & \text{otherwise} \end{cases}$$

where b is the typical branching factor of the game we are analyzing. The parameter γ can be tuned from 1 to b to move continuously from conspiracy depth to classical ply depth.)

To start the search, one calls

```
ABC(root, -infinity, +infinity, d_max, d_min, [], [])
```

where

```
ABC(node, alpha, beta, d_max, d_min, max_options, min_options) {
  if (terminate?(node, alpha, beta, d_max, d_min, max_options, min_options)) {
    return the static value of node;
  }
  if (node is a max-node) {
    v := alpha;
    for each child c of node {
      sib_options = list of static values of siblings of c;
      next_max_options = [sib_options] ++ max_options;
      v = max(v, ABC(c, v, beta, d_max, d_min, next_max_options, min_options));
      if (v >= beta) { return v; }
    }
    return v;
  } else {
    ... dual of the max-node case ...
  }
}
```

and

```
terminate?(node, alpha, beta, d_max, d_min, max_options, min_options) {
  v = static value of node;
  if ((v >= beta) && (max_depth(beta, max_options) >= d_max)) {
    return true;
  }
```

```
  if ((v <= alpha) && (min_depth(alpha, min_options) >= d_min)) {
    return true;
  }
  if ((alpha < v < beta) && (max_depth(v, max_options) >= d_max) && (min_depth(v, min_options) >= d_min)) {
    return true;
  }
  return false;
}
```

and

```
max_depth(v, max_options) {
  return sum over all value lists L in max-options of S(k)
    where k is the number of elements of L > v-delta
}
```

and

```
min_depth(v, min_options) {
  return sum over all value lists L in min-options of S(k)
    where k is the number of elements of L < v+delta
}
```

11.8 Proof number

Implementation idea: to reduce space, assume pre-computed max number of nodes to search. Then we can pre-allocate huge array of elements. Children of node are allocated in adjacent sweep of elements from array - so parent just needs index of first child (and number of children). Note that this doesn't work well with strategies that free nodes etc.

Proof number search is a best-first search method which, as the name suggests, attempts to "prove" a result for a position. It does not rely on an evaluation function *per se*, but rather on a binary proposition or goal, for example, "Win for white" or "Not a loss for black". One can also imagine other propositions such as "White wins a queen". The search process then attempts to prove or disprove the proposition. In the discussion below, we will simply assume that the goal is "Win" for the max player for ease of exposition.

The *proof number* of a node is the number of nodes below it that need to be shown to be wins for the node itself to be a win and the *disproof number* is the number of its descendents that need to be proved losses for the node to be a loss. For a non-terminal leaf-node, we set both the proof and disproof numbers to 1. Thus, for a terminal node which is a win, we have a proof number of zero (0) and a disproof number of infinity (∞). Similarly, for a terminal node which is a loss, the proof number is ∞ and the disproof number is 0. For internal nodes, we can compute inductively their (dis)proof numbers: if a node X is a Max node, then the proof number $X.p$ and disproof number are given by

$$\begin{aligned} X.p &= \min_{C \in \text{Child}(X)} C.p \\ X.d &= \sum_{C \in \text{child}(X)} C.d \end{aligned}$$

since to show a win we only need to find *one* winning move, and to prove a loss *every* move must be shown to lose. Similarly, for a Min node Y, we have

$$\begin{aligned} Y.p &= \sum_{C \in \text{Child}(Y)} C.p \\ Y.d &= \min_{C \in \text{child}(Y)} C.d \end{aligned}$$

since our opponent needs to find only one move she can make that spoils our win to prove a loss, and for us to have a forced win, every possible move by the opponent must lead to a win.

Consider the following simple example: We see that the proof number of the root node is 1 — this makes sense, since if either i or j are wins, then the rightmost move at the root node leads to a win. Similarly, the disproof number of the root node is 4 — this also makes sense, since for the root to be losing, each of the three possible moves must be losing, this requires at least 1 losing move for the first move (a or b) and at least 1 for the second (f) and both i and j must be losing.

Note that there are different sets of moves that would lead to a loss for the root node, for example if c, d, e, g, h, i, j are all losses, then the root is a loss, but that is not the *smallest* set of nodes which could disprove the root. The smallest possible disproof sets are a, f, i, j, or b, f, i, j. Similarly, if a, b, c are all wins, then the root is a win, but that set is not the smallest proving set.

...

The next step is to expand the "most-proving" node — this will be a node which is in both the smallest proving set and the smallest disproving set. In our case, either node i or node j are candidates. Thus one of them would be the next node expanded in the search. Its children would be expanded and the proof-numbers updated.

...

```
pub struct Node {
    internal : boolean,
    max_node : boolean,
    children : Vec<Box<Node>>,
    pn       : isize,
    dp       : isize,
}

pub fn determine_most_proving_node(v : &mut Box<Node>) -> &mut Box<Node> {
    while v.internal {
        if v.max_node {
            // v = leftmost child with equal proof number;
            v = v.find(|s|s.pn==v.pn).unwrap();
        } else {
            //v = leftmost child with equal disproof number;
            v = v.find(|s|s.dp==v.dp).unwrap();
        }
    }
    return v;
}

pub fn update(v: &mut Box<Node>) {
    if v.max_node {
        // pn = minimum of pn of children
        v.pn = v.children.iter().map{|s|s.pn}.min();
        // dp = sum of dp of children
        v.dp = v.children.iter().map{|s|s.dp}.sum();
    } else {
        // pn = sum of pn of children
        v.pn = v.children.iter().map{|s|s.pn}.sum();
        // dp = minimum of dp of children
        v.dp = v.children.iter().map{|s|s.dp}.min();
    }
    if (V has parent) {
        update(parent(V));
    }
}
```

Note that we can actually stop this update procedure when the values stop changing — and can begin the search for the most-proving node from this stopping point also!

```
pub fn expand(v: &mut Box<Node>) {
    children(V) = generate_children(V);
    //for all S in children(V)
    for s : v.children.iter() {
        create_node(S);
        make_edge(V,S);
        value = evaluation(S);
        if (value=="Win") {
            s.pn = 0; s.dp = infinity;
        } else if (value=="Not win") {
            s.pn = infinity; s.dp = 0;
        } else {
            s.pn = s.dp = 1;
        }
    }
}
```

Could initialize leaves with other than 1/1 based on some heuristic (or even with the count of moves — it may be faster to just count moves than to generate them...)

```
func pn_search(position) {
  root = create_root(position);
  while (root.pn!=0 && root.dp!=0) {
    mpn = determine_most_proving_node(root);
    expand(mpn);
    update(mpn);
  }
  if (root.pn==0) return "Win";
  else            return "Not win";
}
```

... step-by-step example of search & updated ...

11.8.1 Modifications and enhancements

Note that the proof-number algorithm (like the conspiracy numbers algorithm) works best on irregular and non-uniformly branching trees — if every node has exactly the same number of children and the game-end nodes are all at the same distance from root, then the algorithm degenerates into a breadth-first brute-force search.

Transposition tables... GHI problem...

11.9 Other heuristics

Chapter 12

Advanced Evaluation

In this chapter we discuss the evaluation of

12.1 TDleaf(λ)

One of the most difficult aspects of constructing an evaluation function can be "tuning" the weighting of different features.

Here we discuss a way for the computer to tune numeric parameters (such as piece values) automatically.

1. Let $J(x, w)$ be a heuristic evaluation function parametrized by $w = (w_1, \ldots, w_k) \in \mathbf{R}^k$.
2. Let $x_1, \ldots, x_N$ be the positions that occurred in the game and let x'_i be the leaf node of the principal variation of the search starting at x_i.
3. Let $r(x_N) \in \{ -1, 0, 1 \}$ be the outcome of the game.
4. Define $r(x'_i, w) = \tanh(\beta \cdot J(x'_i, w))$, where $\beta = 0.255$ for knightcap, (where J is measured in units of 1 pawn).
5. For $i = 1, \ldots, N-1$ compute $d_i = r(x'_{i+1}, w) - r(x'_i, w)$ and partial derivatives $\nabla r(x'_i, w) = (\ldots \frac{\partial}{\partial w_j} r(x'_i, w) \ldots)$.
6. If opponent rating is less than computer rating then set $d_i = 0$ if the move leading to x'_{i+1} was not predicted by the computer.
7. Update the parameters according to

$$w'_j = w_j + \alpha \sum_{i=1}^{N-1} \frac{\partial}{\partial w_j} r(x'_i, w) \left(\sum_{m=i}^{N-1} \lambda^{m-i} d_m \right)$$

   where λ is around 0.3–0.7 if the evaluation function is already reliable and > 0.95 otherwise. Choose α so that updates are about 1/100 of a pawn.

Remarks on evaluating the partial-derivative of the evaluation function: in some cases, we may know the form of the evaluation and we can analytically derive the partial derivative. But suppose we cannot do this easily (or we simply can't be bothered to do it). A general approach to computing ...

$$\frac{\partial}{\partial z} f(..., z, ...) \approx \frac{f(..., z + h, ...) - f(..., z - h, ...)}{2h}$$

for some small h (but not too small). This costs two computations of the evaluation function, but works without special knowledge of the evaluation function ...

12.1.1 TD(μ)

The TD(λ) algorithm suffers from the fact that it can learn to play badly from bad examples. The TD(μ) algorithm is a modification which is designed to allow learning to happen even with bad examples. This is especially useful in the case of self-play; when there is no good opponent to practice and learn from.

12.2 Neural networks

Neural networks provide a way

Neural networks have been used as components in various well-known and high-quality game-engines, including NeuroGammon by ...,

The down-side to neural networks is that executing them tends to be relatively slow, meaning that they are not generally amenable to being used in a deep-searching approach which requires evaluating millions of positions.

12.3 Genetic algorithms

An interesting, but ... Co-evolution ...

12.4 Pattern database / brains

Chapter 13

GUI

In this chapter we

Chapter 14

Parallel Processing

Parallel processing is a technique that is important for taking advantage of the processing power we have in current computers: from multi-core chips to massively-parallel GPUs to networked workstations. It is not straightforward to extend techniques such as $\alpha\beta$-search to work efficiently in a parallel context. There have been a number of

The biggest issue with parallel extensions of $\alpha\beta$-search is that the search is inherently sequential: the results of search of one part of the tree is used when searching another, namely the $\alpha\beta$-bounds.

14.1 Youngest Brothers Wait

Chapter 15

Stuff

In this chapter we

15.1 Notation

15.1.1 Forsythe-Edwards Notation (FEN)

15.1.2 GBR code

15.1.3 PGN

15.2 Typesetting

15.3 Ratings

15.3.1 Elo

The Elo system (developed by Arpad Elo (?!)) is based on assigning numerical ratings to players under the statistical assumption that the probability that a player with rating A will beat a player with rating B is given by

$$\frac{10^{A/400}}{10^{A/400} + 10^{B/400}} = \frac{1}{1 + 10^{(B-A)/400}}$$

which means that a difference of 400 rating points corresponds to 10:1 odds in winning.

15.3.2 Glicko

15.4 Computer analysis

15.4.1 Blunder check

15.4.2 Auto-annotation

15.4.3 Combination finding

15.4.4 Automated problem construction

Chapter 16

Opening Book

In this chapter we discuss the usage and construction of an opening book. [Bur], [DL]

16.1 Basic format

16.2 Basic usage

16.2.1 Hash approach

One method (useful for small books with only positive entries) is to simply store all positions reachable from the book into the hash-table and when searching, if there is a move which reaches a position in the table, to take one of such moves (at random). This approach will break down when you want to store moves which should not be played, or if you want to associate probabilities or other more sophisticated ...

16.3 Automatic construction

16.4 Learning, updating

Chapter 17

Endgame Tablebase

In this chapter we discuss the usage and construction of an endgame tablebase.

Note that the approach of endgame tablebases is not applicable to every game. It works well in the cases where the game "simplifies" towards the end of the game. For example, if pieces tend to get captured as the game goes on and there are fewer and fewer pieces on the board (e.g. chess, checkers, mancala) then this approach works well. If the position gets more complicated as the game goes on (e.g. go, hex) or captured pieces are recycled (e.g. shogi) then it may have limited utility.

17.1 Building

The idea behind ...

The reason we use this backward approach of "un-moving" is to drastically reduce the amount of computation we need to do. Most positions in a given pass of the database will not be relevant, so rather than evaluating all possible moves from all possible positions in a database to see if they are a mate-in-n position, we quickly scan through to find the (relatively) few mate-in-$(n-1)$ positions and only for those positions we do a (un-)move generation step. Though un-move-generation may be complex to code in a game, this can reduce computation time by orders of magnitude (including reducing significantly the number of random-access hits to the database being constructed — this is important for cache behaviour in in-memory databases, as well as disk-access speed in very large databases on disk.)

TODO: simulate behaviour for naive "forward-moving" vs "backward-moving" for some simple game (or congo for example? — reusing existing code? — can be for simple sub-game LE+L, for example) — compute number of move-generations, random-accesses to database, overall clock-time.

17.1.1 Thompson approach

X. Thompson [**?**] gave an early algorithm for "retrograde analysis of certain endgames". It uses 5 active bit-files (for small enough endgames, they may fit entirely in memory and simply be bitmaps):

- W: known white-to-move-and-win
- B: known black-to-move-and-lose
- W_i: latest white-to-move-and-win-in-i-moves
- B_i: latest black-to-move-and-lose-in-i-moves
- J_i: temporary work-file

Then the algorithm proceeds as follows:

(Setup)
- Set B and B_0 to all positions with black to move but checkmated.
- Set $W = 0$.

– Set $i = 0$.

(A) For each B_i position, generate predecessors — add them to W and all positions not already in W form the W_{i+1} positions.

(B) For each W_{i+1} position, generate predecessors to J_{i+1}.

(C) For each J_{i+1} position, generate successors — if all of them are in W then the position is added to B_{i+1}. Finally, add B_{i+1} to B.

(Loop) Increment i and iterate (A),(B),(C) until no changes to B and W

```
// Setup
W = 0;
B = B[0] = BlackCheckmated;
ChangedB = ChangedW = true;
for (i=0; ChangedB || ChangedW; ++i)
{
  // (A)
  J = Predecessors(B[i]);
  W[i+1] = J & ~W;
  ChangedW = NonEmpty(W[i+1]);
  W = W | W[i+1];
  // (B)
  J = Predecessors(W[i+1]);
  // (C)
  B[i+1] = ProveSuccessors(J, W);
  ChangedB = !Subset(B[i+1], B);
  B = B | B[i+1];
}
```

17.1.2 Wu-Beal approach

Here we describe the algorithm given in [**?**] for constructing endgame tablebases. It is designed for handling large databases, minimizing what must be kept in main memory ("random access") ... fast sequential access disks ... Thus we have the two full databases W/B accessed sequentially, one working bitmap S accessed sequentially, and only a single working bitmap R which must have random-access.

```
// Set random-access bitmap R to predecessors of elements in sequential bitmap S
R Predecessors(S);

// load into S or R elements of W or B satisfying predicate P
S/R Load(W/B, P);

// Add elements of S to W or B with predicate P
W/B Add(S, P);

// Only keep elements of S with all successors in R
S = ProveSuccessors(S,R)

// W, B: full depth-to-win databases, accessed sequentially
// S : sequential bitmap
// R : random-access bitmap
void toplevel()
{
  do_initialize();
```

```
  N = 0;                            // depth to mate or conversion
  while (!done_white || !done_black)
  {
    if (!done_white)                // last pass added new positions
    {
      S = Load(W, WIN_IN(N));       // S = WTM win_in_n
      R = Predecessors(S);          // R = BTM predecessors of S
      S = Load(B, UNKNOWN);         // S = BTM unknown
      S = S & R;                    // S = BTM may lose_in_n
      R = Load(W, WIN_IN(<=N));     // R = WTM win_in_(n or less)
      S = ProveSuccessors(S,R);     // S = BTM lose_in_n
      B = Add(S, LOSE_IN(N));       // B += S
      if (distance_to_mate)         // distance_to_mate?
        S = LOAD(B, LOSE_IN(N));    // S = BTM lose_in_n
      R = Predecessors(S);          // R = WTM maybe win_in_(n+1)
      S = Load(W, UNKNOWN);         // S = WTM unknown
      S = S & R;                    // S = WTM win_in_(n+1)
      W = Add(S, WIN_IN_N(n+1));    // W += S
    }
    if (!done_black)                // done for BTM?
    {
      S = Load(B, WIN_IN(N));       // S = BTM win_in_n
      R = Predecessors(S);          // R = WTM predecessors of S
      S = Load(W, UNKNOWN);         // S = WTM unknown
      S = S & R;                    // S = WTM may lose_in_n
      R = Load(B, WIN_IN(<=N));     // R = BTM win_in_(n or less)
      S = ProveSuccessors(S,R);     // S = WTM lose_in_n
      W = Add(S, LOSE_IN(N));       // W += S
      if (distance_to_mate)         // distance_to_mate?
        S = LOAD(W, LOSE_IN(N));    // S = WTM lose_in_n
      R = Predecessors(S);          // R = BTM maybe win_in_(n+1)
      S = Load(B, UNKNOWN);         // S = BTM unknown
      S = S & R;                    // S = BTM win_in_(n+1)
      B = Add(S, WIN_IN_N(n+1));    // B += S
    }
  }
}
```

17.2 Encoding: symmetries and numbering

An important technique for reducing the time and space complexity of the endgame tablebases is the use of symmetries to collapse "equivalent" positions into a single entry. A typical example would be the CHESS endgame of King vs. King + Queen: one can rotate or reflect the board without affecting the game-theoretic value of the positions, thus one could assum

Another technique to reduce the number of positions that needs to be considered is to take care to eliminate impossible positions. For example, in CHESS,

17.3 Compressing

Run-length encoding is a popular compression method that is well suited for sequential access to a table. It typically compresses significantly as many endgame tablebases consist of long runs of draws.

Random-access into the table is much slower as one must scan through sequentially to find the appropriate part of the table, though this can be sped up by the judicious use of indexes to point into the table.

17.4 Using

(internal nodes, tradeoffs in search, etc.)

The API for accessing the database needs to be designed with these symmetries in mind. And the search algorithm should be designed to interface with the endgame database in a transparent manner that allows ... Additionally, it is typically up to the ... Allow fast first check if database exists... then look up position

Part V

Specific Games

Chapter 18

Standard Models

There are a couple classes of standard approaches (or "models") that are applicable to

18.1 $\alpha\beta$-search

Our standard approach, which we will call the *$\alpha\beta$-search model*, (applicable to most chess-like games) is as follows: negamax search with $\alpha\beta$-pruning, iterative-deepening, transposition tables, quiescence search (where applicable), null-move pruning. [PVS search?] The static evaluation functions will depend heavily on the game in question but will be tuned using the TDLeaf(λ) algorithm.

Endgame tablebases will be built where applicable using retrograde analysis.

Opening books constructed using ...

The standard time-management approach will be used.

18.2 MCTS

Another standard approach, which has proven effective for some games, such as GO and HEX, is the *MCTS model*: a Monte-Carlo tree search based on UCT with modifications to incorporate domain-specific knowledge and with progressive widening.

18.3 Etc.

non-deterministic games

incomplete information

multi-player games

Arimaa...

Chapter 19

Congo

In this chapter we delve more deeply into the game of CONGO.

We will apply our standard approach of $\alpha\beta$-search with iterative deepening, transposition table and quiescence extensions (based on captures, checks, and drowning.) For static evaluation we'll use material values with a piece-square bonus as well as some CONGO-specific features such as TODO

We'll use the TDLeaf(λ) self-learning approach to arrive at the appropriate weightings.

For end-games, we'll go ahead and build 3, 4, and 5-piece table-bases.

Opening books will be built by the self-play and deep-search approach.

— lion, pawn, super-pawn, elephant, monkey, alligator, giraffe

19.1 Endgame Tablebase

We will build end-game tablebases for all endings with up to 4 pieces (plus lions). First, let's estimate the number of positions in endgames of the different types. A few remarks are in order here: first, since the lions are restricted to their 3×3 dens, there are at most 81 positions for the pairs of lions to be in, and we will assume that the white lion is in the left-most 6 squares in his den, giving us only $6 \times 9 = 54$ different lion dispositions. The pawn cannot occur in the last row (since it will promote when it reaches the last line), giving only $7 \times 6 - 2 = 40$ possible positions (2 of the squares are occupied by lions). For other piece types, there are no restrictions except for the general restrictions of only one piece per square, giving $49 - n$ possible squares, where n is the number of occupied.

L+L		54	54
LP+L	54×40	2160	2214
LX+L (x=S/E/M/A/G)	$54 \times (49 - 2)$	12 690(2538 each)	14 904
LP+LP	$54 \times 40 \times (40 - 1)$	84240	99 144
LX+LP (x=S/E/M/A/G)	$54 \times (49 - 2) \times (40 - 1) \times 5$	494 910(98 982 each)	594 054
LX+LX (x=S/E/M/A/G)	$54 \times (49 - 2) \times (49 - 3) \times 5$		
LX+LY (x,y=S/E/M/A/G;x$\neq y$)	$54 \times (49 - 2) \times (49 - 3) \times 5 \times 4$		
LXP+L (x=S/E/M/A/G)	$54 \times (49 - 2) \times (40 - 1) \times 5$	494 910(98 982 each)	
LXY+L (x,y=S/E/M/A/G)	$54 \times (49 - 2) \times (49 - 3) \times 5 \times 5$	2 918 700(116 748 each)	

The simplest encoding gives for each particular man-combination:

L+L	9×9	81	81
LX+L (x=P/S/E/M/A/G)	$9 \times 9 \times 49 \times 6$	23 814(3969 each)	23 895
LXY+L (x,y=P/S/E/M/A/G)	$9 \times 9 \times 49^2 \times 6^2$	7 001 316(194 481 each)	7 025 211
LX+LY (x,y=P/S/E/M/A/G)	$9 \times 9 \times 49^2 \times 6^2$	7 001 316(194 481 each)	14 026 527
LXY+LZ (x,y,z=P/S/E/M/A/G)	$9 \times 9 \times 49^3 \times 6^3$	2 058 386 904(9 529 569 each)	
LXYZ+L (x,y,z=P/S/E/M/A/G)	$9 \times 9 \times 49^3 \times 6^3$	2 058 386 904(9 529 569 each)	

19.1.1 Another attempt at counting endgame positions (10-20-2011)

Note that we can choose either a lion *or* a piece to be restricted to, say, the left side of the board. To indicate which piece is restricted, we use underlining below. Note also that pawns cannot be in the first or last rows.

$\underline{\text{L}}$L	6×9	54	54
L$\underline{\text{X}}$L (x=ACEGSZ)	81×28	$13,608$(2,268 each)	
L$\underline{\text{P}}$L	81×20	$1,620$	
L$\underline{\text{X}}$LY (x,y=ACEGSZ)	$81 \times 28 \times 46$	$2,190,888$(104,328 each)	
L$\underline{\text{X}}$YL (x,y=ACEGSZ; x¡y)	$81 \times 28 \times 46$	$1,564,920$(104,328 each)	
L$\underline{\text{X}}$XL (x=ACEGSZ)	$54 \times \frac{47 \times 46}{2}$	$350,244$(58,374 each)	
L$\underline{\text{X}}$LP (x=ACEGSZ)	$81 \times 28 \times 35$	$476,280$(79,380)	
L$\underline{\text{X}}$PL (x=ACEGSZ)	$81 \times 28 \times 35$	$476,280$(79,380)	
L$\underline{\text{P}}$LP	$81 \times 20 \times 34$	$55,080$	
L$\underline{\text{P}}$PL	$81 \times 20 \times 34$	$55,080$	
$\underline{\text{L}}$XXLY (x,y=ACEGSZ)	$54 \times \frac{47 \times 46}{2} \times 45$	$94,565,880$(2,626,830 each)	
L$\underline{\text{X}}$YLZ (x,y,z=ACEGSZ; x¡y)	$81 \times 28 \times 46 \times 45$	$422,528,400$(4,694,760 each)	

Chapter 20

Mancala

In this chapter we delve more deeply into the sowing game of MANCALA. Now, there are many different variants of this game; indeed, one might rather call this a *family* of games, rather than a game. Examples include OWARI, MANCALA, KALAH, BAO, ...

20.1 Kalah

Kalah

20.2 Awari

20.3 Mancala

20.4 Dakon

20.5 Bao

Chapter 21

Chess

In this chapter we delve more deeply into the game of CHESS.

Chapter 22

Checkers

In this chapter we delve more deeply into the game of CHECKERS.

CHECKERS is an old game

CHECKERS is solved — that it is known to be a theoretical draw.

CHECKERS is

Chapter 23

Xiangqi

In this chapter we delve more deeply into the game of XIANGQI (or CHINESE CHESS).

Chapter 24

Shōgi

In this chapter we delve more deeply into the game of SHŌGI and some of its many variations.

24.1 Shōgi

the game of SHŌGI.

24.2 Dōbutsu shōgi

the game of DŌBUTSU SHŌGI.

24.3 Tori shōgi

the game of TORI SHŌGI.

24.4 Poppy shōgi

the game of POPPY SHŌGI.

24.5 Whale shōgi

the game of WHALE SHŌGI.

24.6 Chu shōgi

The game of CHU SHŌGI is a fascinating game that was ... china... 15th century... with ...

The biggest problem it will present is simply one of scale — the game is played on a 12×12 board and there are ... pieces on the board at the start of the game. This gives a large branching factor, implying that it will be difficult to search very deeply; but ...

24.7 Tenjiku shōgi

The game of TENJIKU SHŌGI is similar to CHU SHOŌGI but is even larger: it is played on

Despite its large size, it has a highly tactical flavor

Chapter 25

Tafl

In this chapter we delve more deeply into the game of HNEFATAFL. This is an ancient game and not without interest. Due to its venerability there is a high degree of variability in rules and many variations and variants.

The specific variation we will cover here is ...

25.1 Evaluation

One component of an evaluation function we will use for this game is the number of consecutive moves it would take the king to win.

25.2 Mate search

This game seems ideal for approaches such as proof-number search or λ-search.

Chapter 26

Go

In this chapter we delve more deeply into the game of GO (or WEIQI).

Chapter 27

Havannah

In this chapter we delve more deeply into the game of HAVANNAH.

For this game, MCTS (with various modifications) has proven to be highly effective.

Chapter 28

Hex

In this chapter we delve more deeply into connection games, in particular the game of HEX.

Chapter 29

Hive

In this chapter we delve more deeply into the game of HIVE.

HIVE presents some new complications for programming. It has no set board — in principle it is played on an infinite board. That is, since the pieces are placed ...

Chapter 30

Amazons

In this chapter we delve more deeply into the game of AMAZONS.

Chapter 31

Dots-and-boxes

In this chapter we delve more deeply into the game of DOTS-AND-BOXES.

Chapter 32

Exotica

32.1 Backgammon

The game of BACKGAMMON represents new challenges, the biggest being that of randomness — usage of dice complicates matters.

32.2 Poker

The game of POKER represents many new challenges. In addition to the randomness of the shuffled deck, we also have imperfect information: we don't know what cards the other players are holding.

32.3 Kriegspiel, Stratego

The games KRIEGSPIEL and STRATEGO represents the challenges of

32.4 Wargames, Advanced Squad Leader

The game ADVANCED SQUAD LEADER (ASL)

Chapter 33

Combinatorial game theory

In this chapter we discuss

Chapter 34

Zillions of Games

Zillions of Games is a game package for Windows that uses a "universal gaming engine" technology that allows you to play nearly any abstract board game or puzzle in the world.

Chapter 35

International Computer Olympiada (ICGA)

Game	Tournaments	Programs
ABALONE	1	2
AMAZONS	9	12
AWARI	5	9
BACKGAMMON	7	12
BRIDGE	5	9
CHECKERS	2	8
CHESS	60	245
CHINESE CHESS	13	36
CLOBBER	2	3
CONNECT6	4	17
CONNECT-FOUR	1	4
DOMINOES	1	3
DOTS AND BOXES	4	8
DRAUGHTS	9	21
GINRUMMY	1	2
GIPF	1	2
GO	25	59
GO-MOKU	4	16
HAVANNAH	1	2
HEX	6	9
KRIEGSPIEL	2	4
LINES OF ACTION	7	9
NINE MEN'S MORRIS	1	2
OCTI	1	2
OTHELLO	4	31
PHANTOM GO	3	4
POKER	1	2
POOL	3	6
QUBIC	2	4
RENJU	4	9
SCRABBLE	4	8
SHOGI	9	10
SURAKARTA	2	3

35.1 2008 Computer Olympiad (call for participation)

Official games:

ABALONE, AMAZONS, ARIMAA, BACKGAMMON, BAO, BRIDGE, CHINESE CHESS, CLOBBER, COMPUTATIONAL POOL, CONNECT-6, DIPLOMACY, DOMINOES, DOTS-AND-BOXES, 10X10 DRAUGHTS, GIPF, GO, 9X9 GO, HEX, KRIEGSPIEL, LINES OF ACTION, OCTI, OTHELLO, PHANTOM GO, POKER, SCRABBLE, STRATEGO, SHOGI, SURAKARTA

Willing to host more if there were strong engines, such as:

ATAXX, DVONN, MEDIOCRITY, ONYX, TAMSK, TWIXT, ZÈRTZ

35.2 2009 Computer Olympiad (competed)

AMAZONS, CHESS, CHINESE CHESS, CONNECT6, DRAUGHTS, GO, GO (9X9), HEX, LINES OF ACTION, SHOGI, HAVANNAH, KRIEGSPIEL, PHANTOM GO

35.3 2010 Computer Olympiad

35.3.1 call for participation

AMAZONS, BACKGAMMON, BRIDGE, CHESS, COMPUTATIONAL POOL, CONNECT6, CHINESE CHESS, DOTS AND BOXES, INTERNATIONAL DRAUGHTS, 19X19 GO, 9X9 GO, HEX, HAVANNAH, LINES OF ACTION, SHŌGI, and, SURAKARTA

35.3.2 participated

Chapter 36

Game rules, sample games, comparisons

Game	Average game length	Average number of moves	Estimated State Space	Classifications
Checkers				
Chess				
Awari				
Shogi				Clas-
Go				
Amazons				
Shogi				

sification codes: Endgame: convergent/divergent/stationary Information: perfect/imperfect(common)/imperfect(private) Determinism: deterministic/stochastic

CHESS, CHECKERS, CHINESE CHESS, SHOGI, CONGO, GO, HEX, DOTS-AND-BOXES, AMAZONS, MANCALA/AWARI, BACKGAMMON, CONNECT-4, PHUTBALL, TAFL/TABLUT, OTHELLO, KONANE, FOX-AND-GEESE, NINE-MAN-MORRIS, KHET, ABALONE,

YAVALATH, PENTALATH, PATAGONIA, (Browne's games)

GIPF, YINSH, TZAAR, ZÉRTZ, DVONN, PÜNCT,

BRIDGE, POKER, SCRABBLE, MONOPOLY, KRIEGSPIEL, YAHTZEE, STRATEGO,

settlers of cataan, carcasonne, puerto rico, yahtzee, parchisi, clue(do),

Encyclopedia of Chess Variants: progressive chess, alapo, alice chess, atomic chess, avalanche chess, barrier chess I, beirut chess, berolina chess, betza's chess, bird chess I, bomb chess, caissa, carnivore chess, chad, checkless chess, chess football, chezz, chu shogi, columbia common chess, congo, diana, dragon chess, dragonfly, eternity's children, extinction chess, grand chess, hexagonal chess (glinski's), jetan, kamikaze chess I, koopa chess, loonybird, losing chess, makruk, microchess I/II/III/IV, mimic chess I/II, minishogi, missile chess, rifle chess, null chess, one-dimensional chess, parallel time-stream chess (?), parallel world chess, the pawn's game I, petty chess, ploy, rotary, plague chess, push chess, quest-chess, quickchess, refusal chess, compromise chess, sjakti, smess, spite chess, star-trek chess, three kings chess, tori shogi, tenjiku shogi, ultima, wa shogi, whale shogi, yari shogi

[Hei00]

36.1 Congo

CONGO is a chess-like game played on an 7×7 board with

36.2 Chess

CHESS is played on an 8×8 board with

1	2	3
4	5	6
7	8	9

The pieces' basic movement is given as follows

- King - moves one step in any direction

×	×	×
×	K	×
×	×	×

- Queen - slides any number of steps in any direction

↖	↑	↗
←	Q	→
↙	↓	↘

- Rook - slides any number of steps in an orthogonal direction

	↑	
←	R	→
	↓	

- Bishop - slides any number of steps in a diagonal direction

↖		↗
	B	
↙		↘

- Knight - jumps one orthogonal and one diagonal step

	×		×	
×				×
		N		
×				×
	×		×	

- Pawn - moves one step forward (without the right of capturing) or captures diagonally forward

c	m	c
	P	

36.3 Shogi

Glossary

PNS — see proof-number search

proof-number search — A method of searching

Index

Bibliography

[Bur] Michael Buro, *Toward opening book learning.*

[DL] Chrilly Donninger and Ulf Lorenz, *Innovative opening-book handling.*

[GR] V. Scott Gordon and Ahmed Reda, *Trappy minimax - using iterative deepening to identify and set traps in two-player games.*

[Hei00] Ernst Heinz, *Scalable search in computer chess*, Vieweg, 2000.