Programmeerimise põhikursus Javas

Loeng 13

http://courses.cs.ttu.ee/pages/ITI0011

Outline

- Homework stuff

- Test next week
- Exam times (doodle)

- codingbat?

- IV homework - Gomoku

- Recursion
- Minimax, alpha-beta

Homework submission

- https://courses.cs.ttu.ee/pages/ITI0011:git

- Homeworks into HW1, HW2, HW3 and HW4 folders
- **Check your score table to see git status**
- Homework 4 to be pushed into git latest **December 14th 23:59**
  - into folder "HW4"
- **The last option to defend HW4 is in December**

- **Course code example in git:** http://firstname.lastname@git.ttu.ee/kursused/iti0011/materjalid.git
- Use UNI-ID to access materials (not visible in browser)

Test, Exam

- Exam times (to be fixed):
  - **16.01.2015 (Friday) 10:00 and 13:00 U06A-229**
  - ??.01.2015 (?) 10:00 and 13:00
- Exam is written on paper, 2h time

- Test
  - during the lecture on **02.12.2014**
  - very short version of exam (3 "questions")
    - reading code
    - questions with options
    - writing code
  - gives you some feedback about your skills and preparation
  - on paper
  - about 45 min

Recursion

- Recursion calls itself
- Must have a stopping case
- Problem can be divided into sub problems
- All recursive algorithms/methods can be written without recursion

- Classic example: factorial

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ (n-1)! \times n, & \text{if } n > 0 \end{cases}$$

Recursion

- Factorial example

$$\begin{aligned} 4! &= 3! \times 4 \\ &= \overbrace{(2! \times 3)} \times 4 \\ &= (\overbrace{(1! \times 2)} \times 3) \times 4 \\ &= ((\overbrace{(0! \times 1)} \times 2) \times 3) \times 4 \\ &= (((1 \times 1) \times 2) \times 3) \times 4 \\ &= 24 \end{aligned}$$

Recursion vs. iterative

- Fibonacci numbers are defined:
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$ for n > 1
- Example numbers:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
- Recursive algorithm:

```
public static int fiboRec(int num) {
        if (num < 2) {
                return num;
        }
        return fiboRec(num - 1) + fiboRec(num - 2);
}
```

Recursion vs. iterative

- To get 10th Fibonacci number, we have to call **fiboRec** 177 times!
- Linear approach:

```
public static int fiboIt(int num) {
      int fib = 0; // the number we are looking for
      int prev = 1; // previous fibo number
      for (int i = 1; i <= num; i++) {
            fib = fib + prev;
            prev = fib - prev;
      }
      return fib;
}
```

- Only 10 times of for-loop execution
- See also **fiboLinRec** method for linear recursion.

Linear recursion, tail recursion

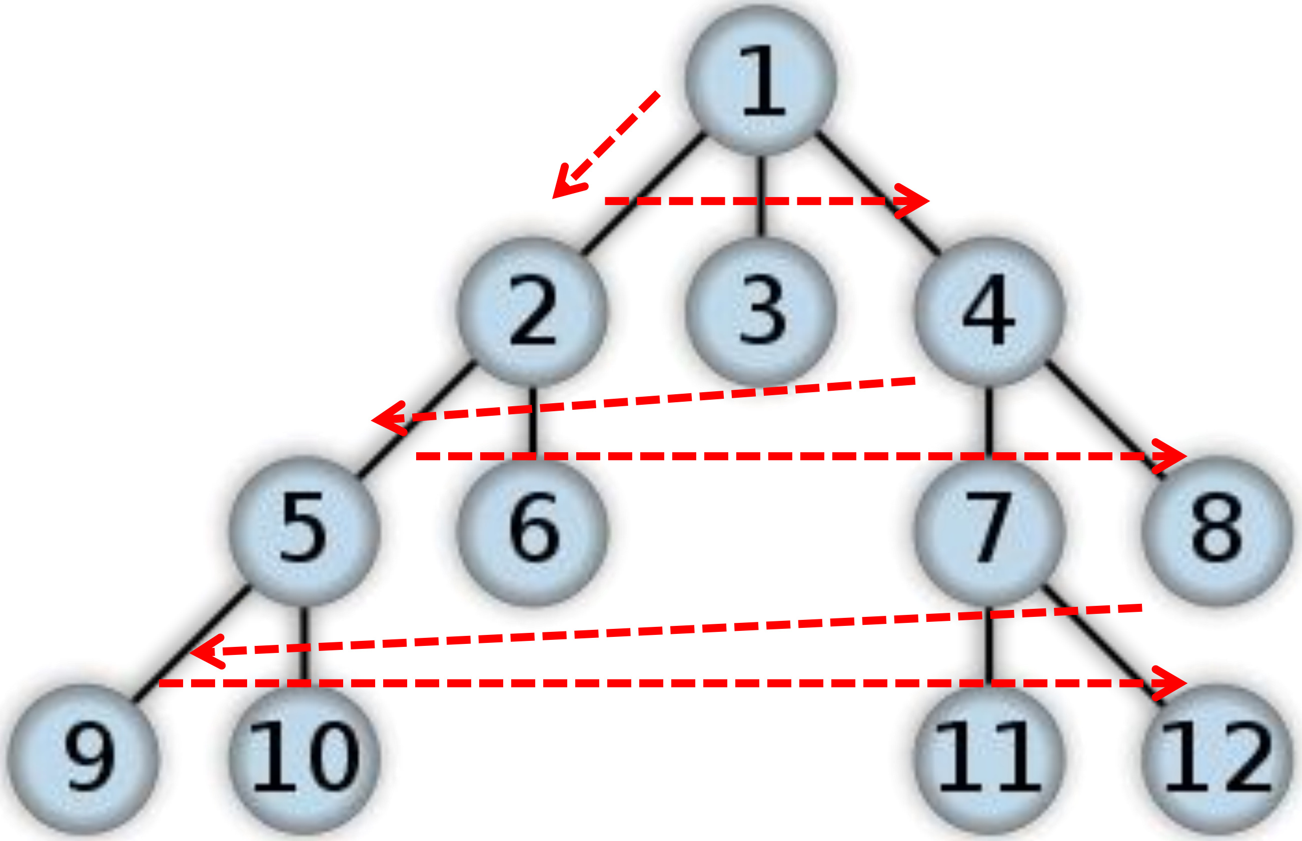
- Linear recursion - a single recursive call is made
- Tail recursion - linear recursion where recursive call is the last step

- Fibonacci with tail recursion:

```
public static int fiboTailRec(int depth, int val, int prev) {
      if (depth == 0) return prev;
      return fiboTailRec(depth - 1, val + prev, val);
}
```

Breadth-first search

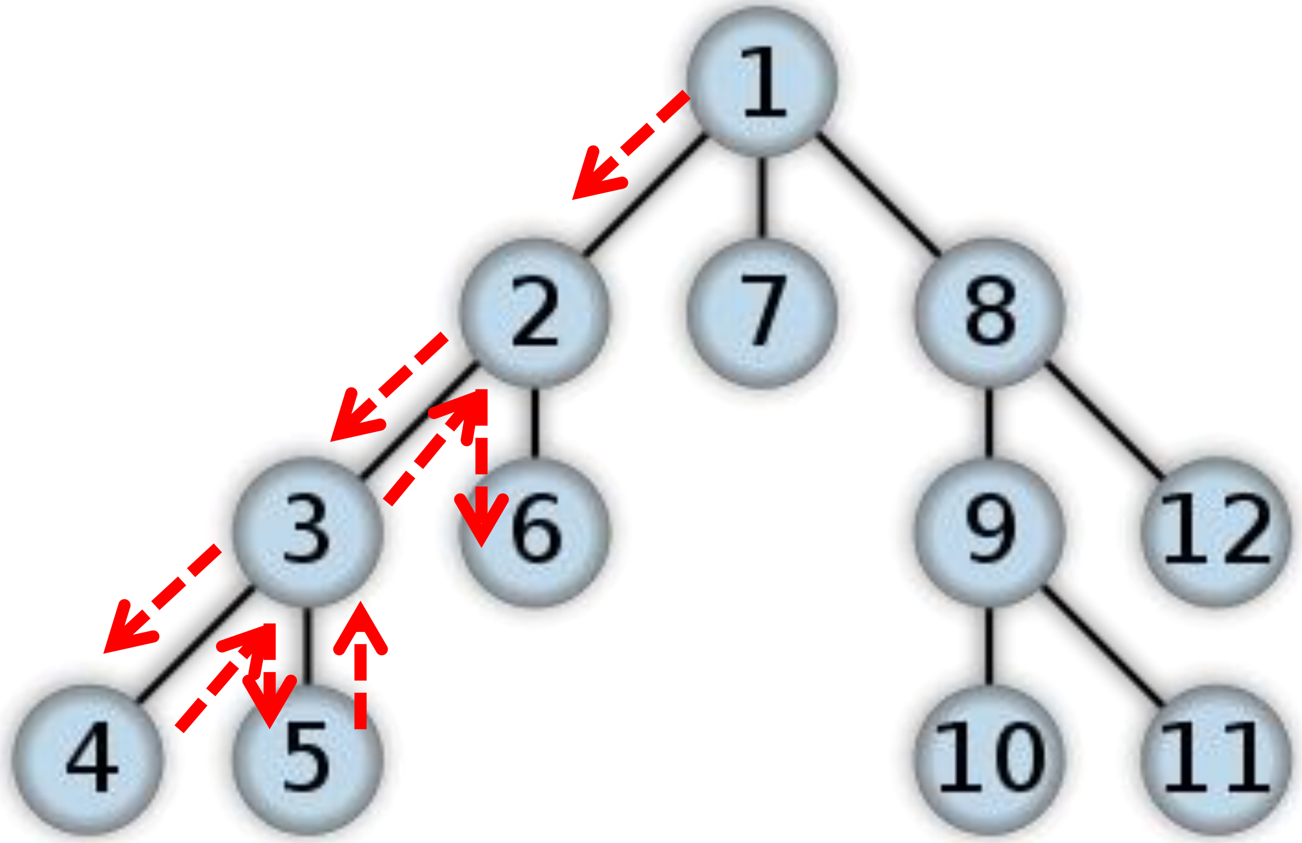
- Advantage: find the goal closest to the initial state
- Problems: Exponential space requirements



Breadth-first

- Advantage: find the goal closest to the initial state
  - Find the route from A to B with the fewest stops
- Problems: Exponential space requirements

Depth-first search



Depth-first search

- Advantages: limited space requirements
- Problems:
  - does not always find the goal
  - does not always find the goal closest to the initial state
  - may take very long time before finding a goal that may be close to the initial state

Limited-depth depth-first search

- Like ordinary DFS, but the search is limited to a predefined depth
- the depth of each state is recorded as it is generated. When picking the next state to expand to, only those with depth less or equal to the predefined depth are expanded.

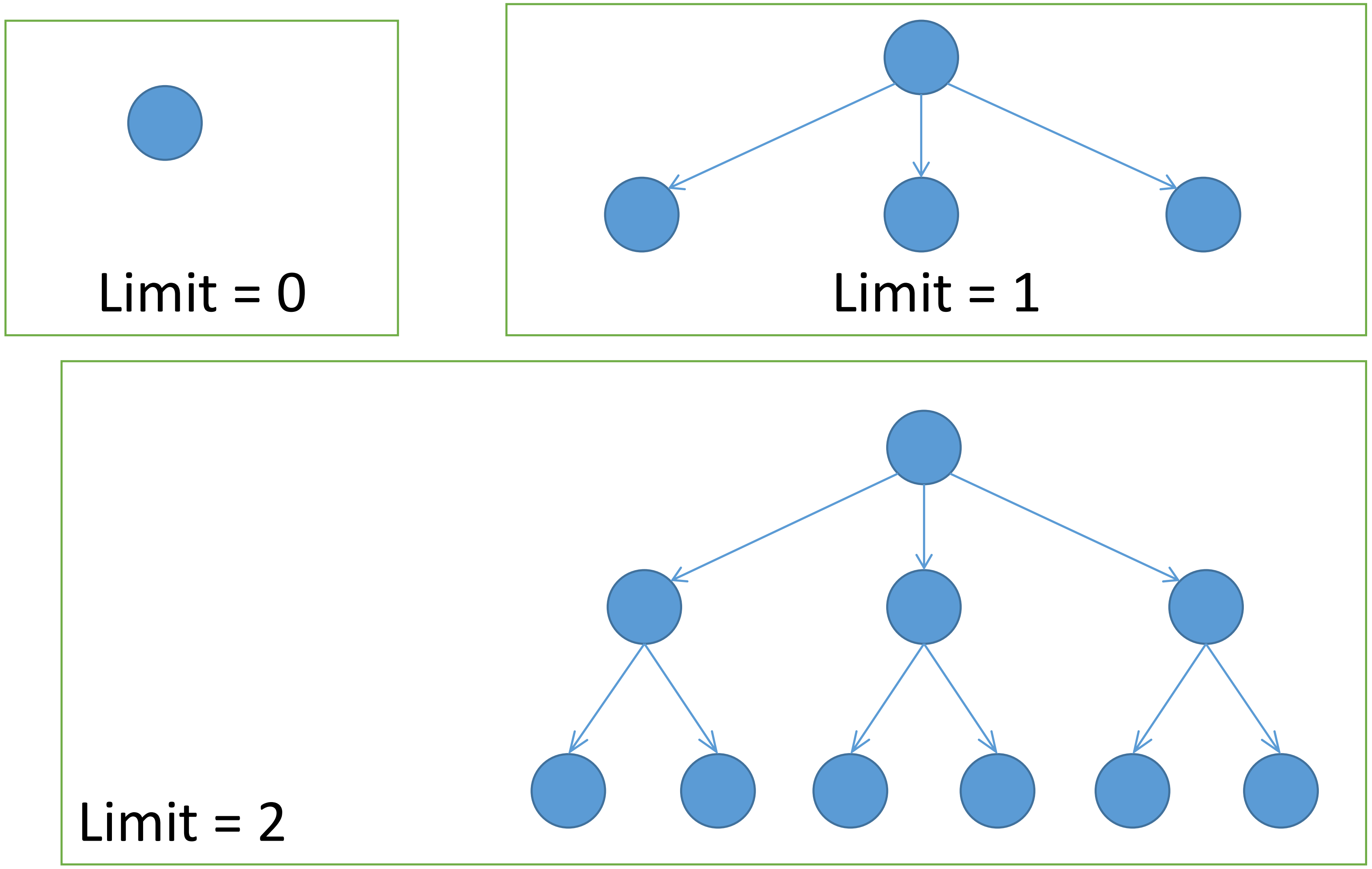
- **Problem: if we pick a too small depth limit, we may fail to find the answer**
- **what is a good depth limit?**

Iteratively deepening depth-first search

- Limited depth DFS where depth limit is increased with each iteration
  - Generate solutions with depth limit 1
  - Generate solutions with depth limit 2 etc.

- Once all the nodes of a given depth are explored, the current depth limit is incremented.

Iterative deepening search



Iterative deepening search

- Not so bad as it looks!
- Why bad: the root subtree is computed **every time** instead of storing it

- Most of the solutions are in the bottom leaves anyhow:

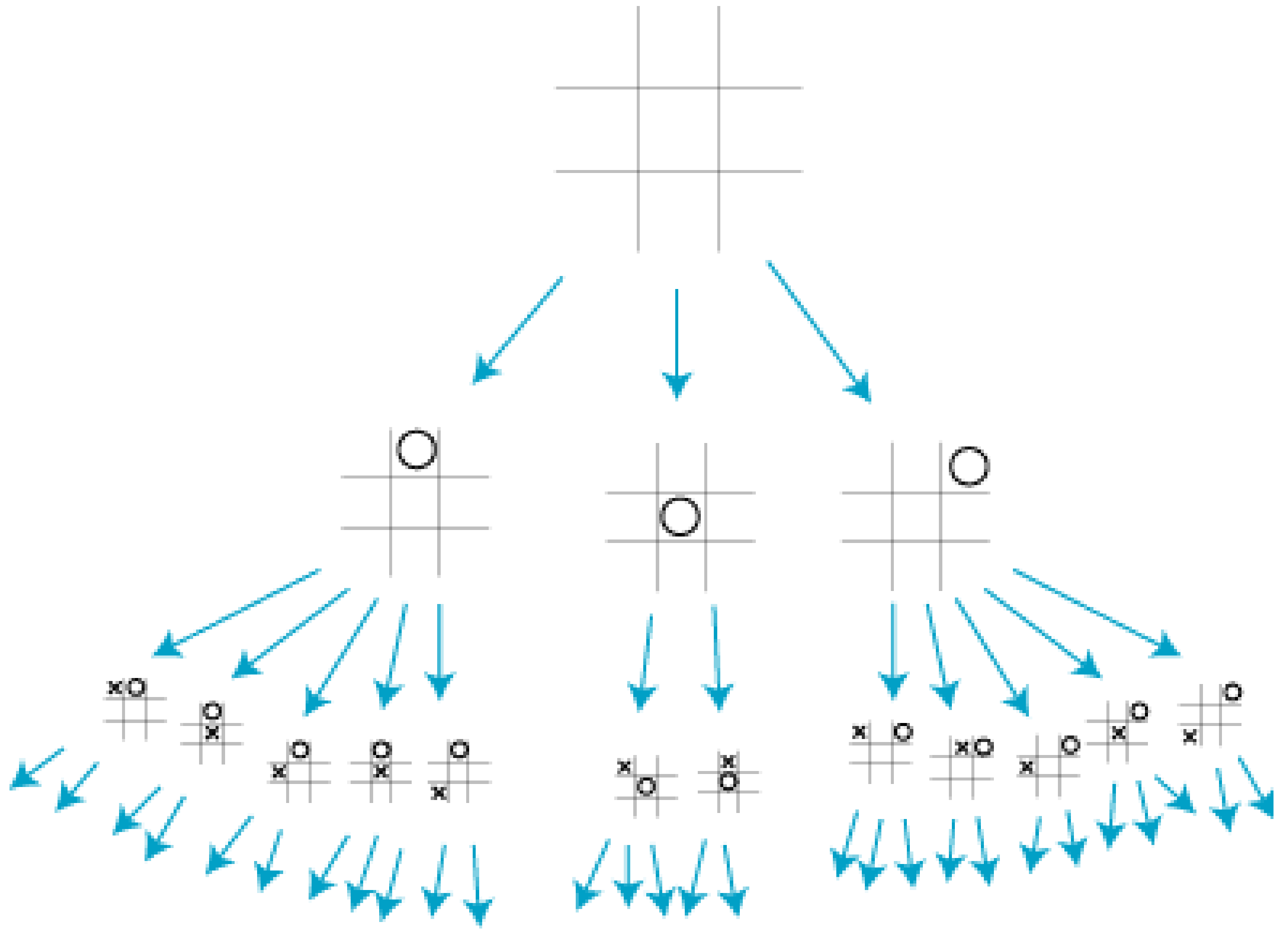
$b + b^2 + b^3 + \ldots + b^d = O(b^d)$

- Repeating the search takes:

$(d + 1)b + (d)b^2 + (d - 1)b^3 + \ldots + (1)b^d = O(b^d)$

- For $b = 10$ and $d = 5$ the number of nodes searched is 111,111 regular vs. 123,456 repeated (**only 11% more!**)

Tree of moves in Tic-Tac-Toe



http://www.ics.uci.edu/~eppstein/180a/970417.html

How to use the game tree?

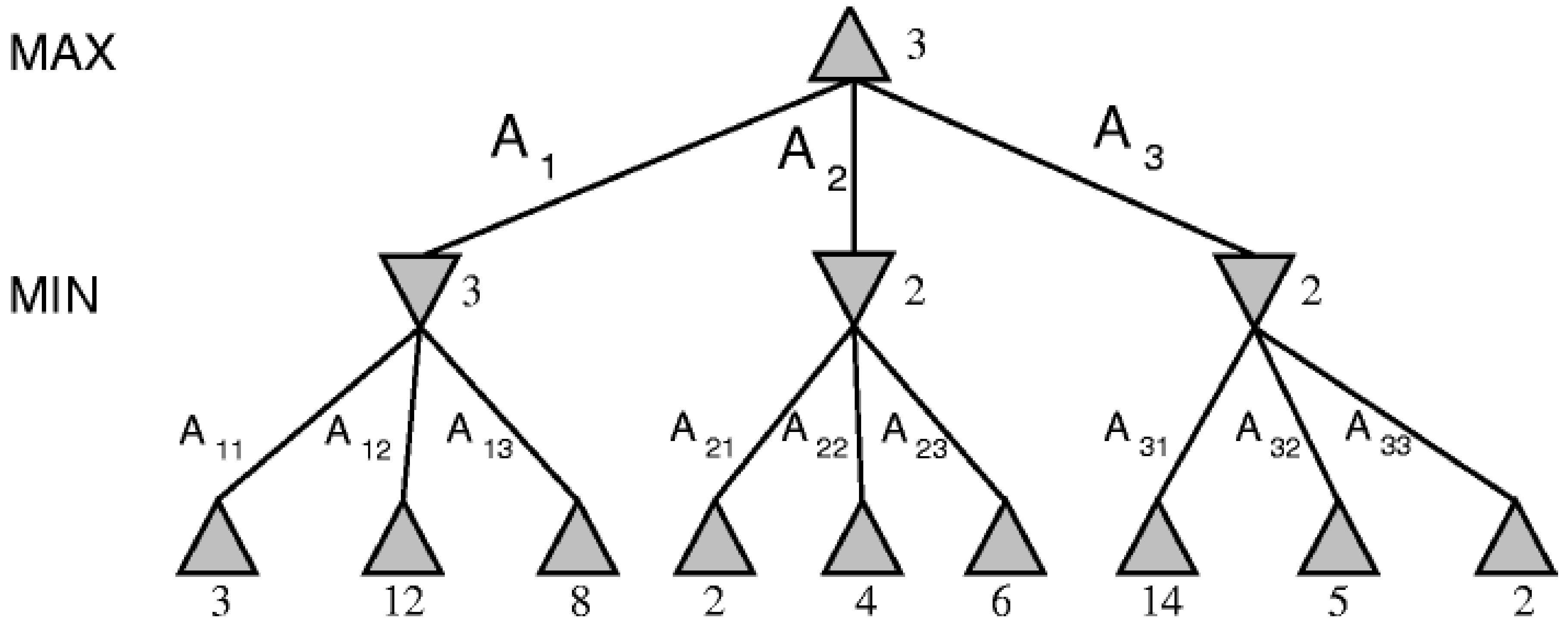
- Prerequisite:
  - Computer wants to make a move which gives the highest estimation score (the best move for computer)
  - Opponent wants to make a move which gives the lowest estimation score (best move for the opponent, worst move for the computer)

- Realization:
  - From every state of the game, computer selects the move which has the highest estimation score (for computer)
  - From every state of the game, the opponent selects the move which has the lowest estimation score (for computer)

- The idea:
  - Let's calculate the game tree with depth N (example N = 3)
  - On the lowest level (the deepest) we just calculate the estimation score
  - On every level above the lowest, we propagate estimation scores upwards

Minimax algorithm

- Demo: http://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html



http://cs-alb-pc3.massey.ac.nz/notes/59302/l05.html

The skill of the program

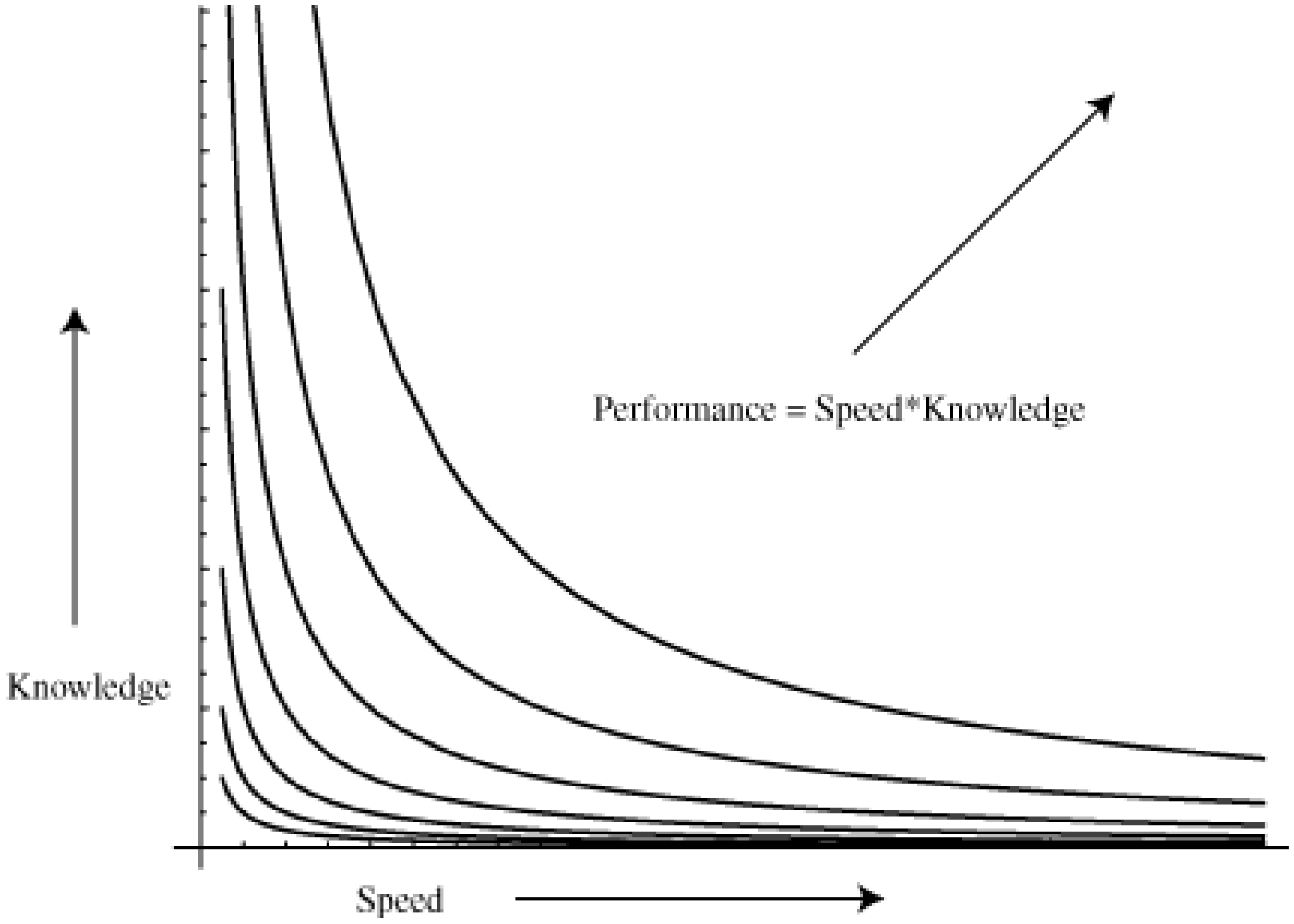
- The higher the depth of the tree is, the better the playing skill is

- The tree grows very quickly!

- In chess, there are about 30 moves on a state
  - The first level of the tree would have 30 nodes
  - The second level would have 30*30 nodes
  - ...
  - The N-th level would have 30^N

  - On the 50th level there would be about 30^{50} (10^{73}) nodes, which is about the same as the number of particles in the Universe (10^{82})

Performance of the game



http://www.ics.uci.edu/~eppstein/180a/990114.html

Improvements

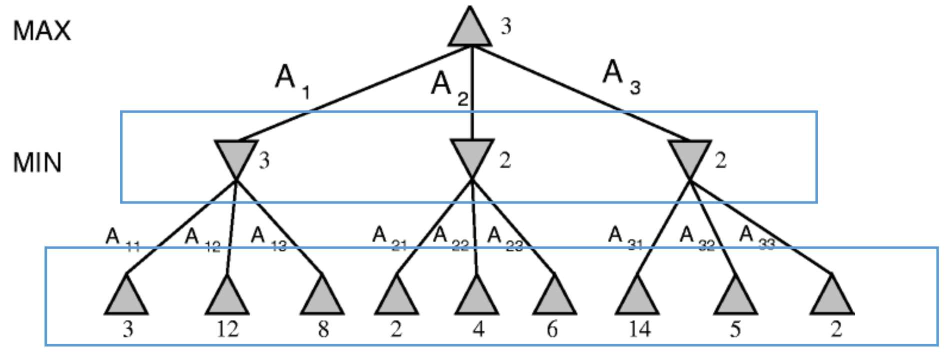
- There is no silver bullet!
- Many different improvements/tricks can be applied

- For example:
  - Let's improve the evaluation calculation
  - This in return makes our program slower and we can check less states

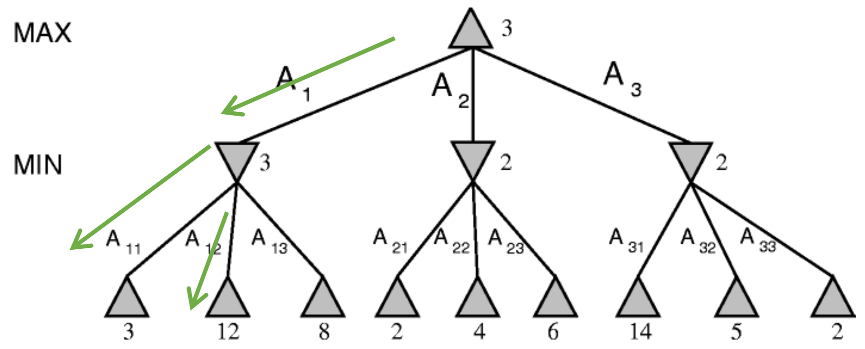
- Most of the improvements are achieved by **pruning the game tree**: **there is no need to search the whole tree**.

Depth First VS Breadth First Search

- We can look through the tree:
  - level by level
    (breadth first search)



  - depth first search:



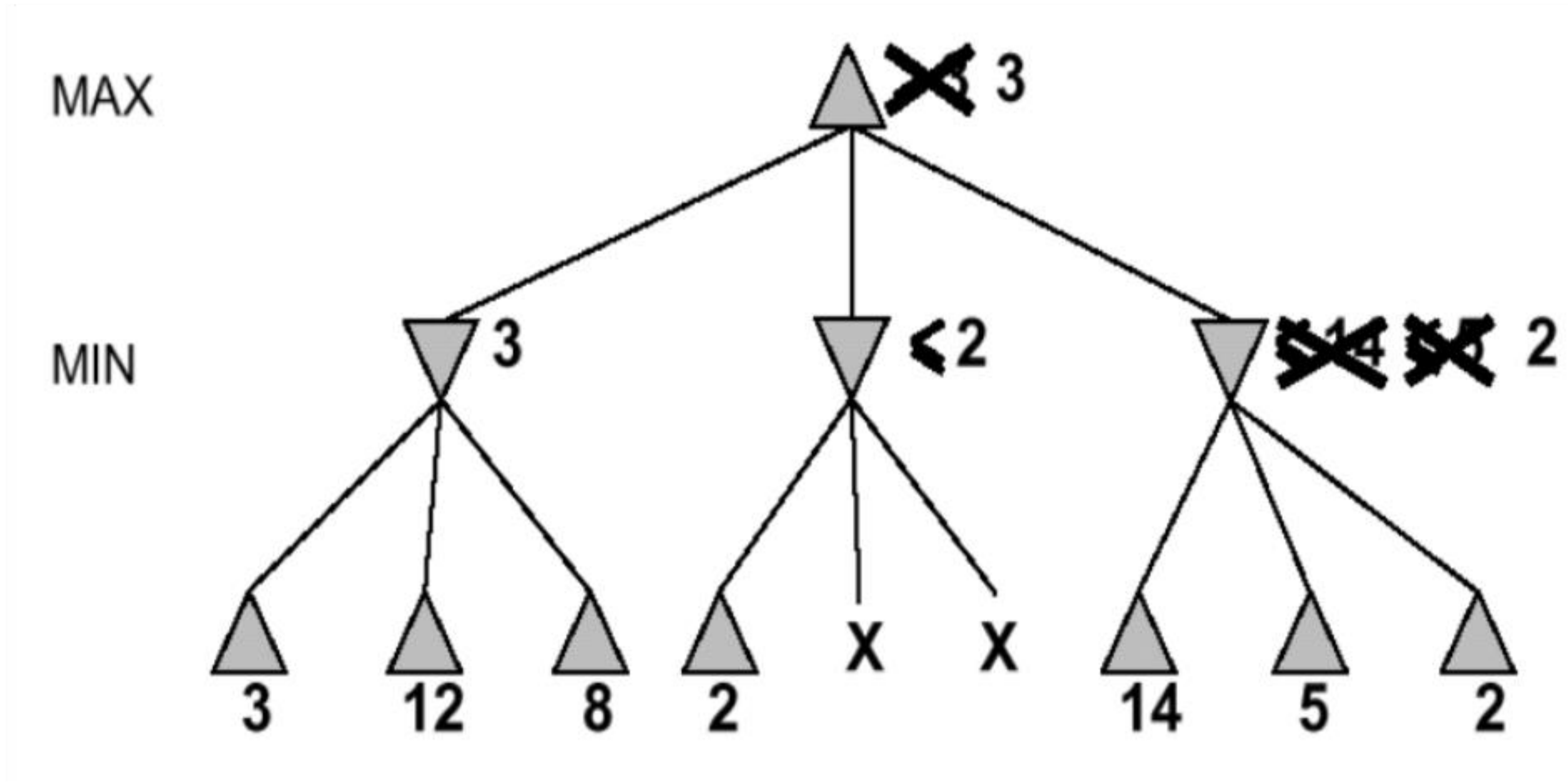
- Depth first search
  is preferred:
  - Uses less memory

- More about tree search techniques in the next lecture

Minimax algorithm

```
int minimax(position p, int depth) {
    struct move list[MAXMOVES];
    int i, n, bestValue, value;

    if (checkWin(p)) { // check for win-lose, game-over
        if (p.color == WHITE) return -INFINITY;
        else return INFINITY;
    }

    if (depth == 0) return evaluation(p); // search depth reached?

    if (p.color == WHITE) bestValue = -INFINITY; // helpers
    else bestValue = INFINITY;

    n = makeMoveList(p, list); // legal moves, total n moves
    if (n == 0) return handleNoMove(p); // no moves, tie?

    for (i = 0; i < n; i++) {           // iterate over legal moves
        doMove(list[i], p);             // try a move on the board
        value = minimax(p, depth - 1); // evaluate the position
        undoMove(list[i], p);           // take the move back for next iteration
        if (p.color == WHITE) bestValue = max(value, bestValue); // max own score
        else bestValue = min(value, bestValue); // min opponent
    }
    return bestValue; // recursive return
}
```

Advanced search algorithm: alpha-beta



Alpha-beta

```
alpha-beta(player, node, alpha, beta)
    if (game over in current board state) return winner

    children = all legal moves from player from this state
    if (max's turn)
        for each child
            score = alpha-beta(other player, child, alpha, beta)
            if score > alpha then alpha = score
            if alpha >= beta then return alpha (cut off)
        return alpha (this is our best move)
    else (min's turn)
        for each child
            score = alpha-beta(other player, child, alpha, beta)
            if score < beta then beta = score
            if alpha >= beta then return beta (cut off)
        return beta (this is the opponent's best move)
```

Negamax, alpha-beta algorithms

```
int negaMax(pos, depth) {
    if (depth == 0) return evaluate(pos);
    best = -INFINITY;
    succ = successors(pos); // legal moves

    while (not empty(succ)) {
        pos = removeOne(succ); // try one
        value = -negaMax(pos, depth - 1); //calc position
        if (value > best) best = value;
    }
    return best;
}
----------------------
int alphaBeta(pos, depth, alpha, beta) {
    if (depth == 0) return evaluate(pos);
    best = -INFINITY;
    succ = successors(pos);

    while (not empty(succ) && best < beta) {
        pos = removeOne(succ);
        if (best > alpha) alpha = best;
        value = -alphaBeta(pos, depth - 1, -alpha, -beta);
        if (value > best) best = value;
    }
    return best;
}
```

Improvement: sorting legal moves

- To improve the algorithm, it's wise to sort legal moves in a game state: start with the moves which are probably better

- This improves pruning effect
- How to sort?

- Iterative deepening:
    - Start full search with depth 2
    - Continue with full search with depth 4 etc.
    - Every time use the result of the last search for sorting

Other improvements

- **Killer moves**
  - Let's remember very good moves for both players
  - Try already stored very good moves first

- **Quiescence search**
  - Interesting positions are searched to a greater depth than "quiet" ones
  - High movement or capturing will be searched deeper

- **Null-move**
  - What happens if the opponent skips a move?
  - If the result is OK, it is a good indicator
  - If the result is bad, we store the "killer move"

The effect of pruning

- **The deeper the game tree, the more effect pruning has**.

- Game tree with **depth 5**:
  - MiniMax: 10,541,242 evaluation nodes
  - Alpha-Beta: 1,037,209 evaluation nodes
  - A-B + "killer moves": 530,587 evaluation nodes

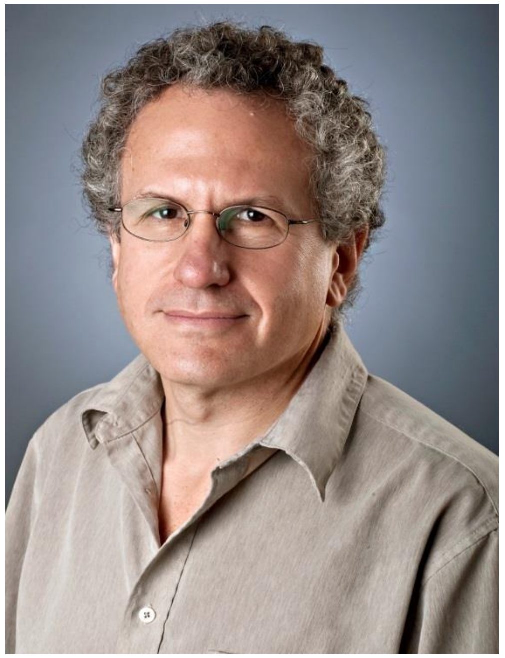
- Game tree with **depth 7**:
  - MiniMax: 8,100,000,000 evaluation nodes
  - Alpha-Beta: 162,662,568 evaluation nodes
  - A-B + "killer moves": 46,455,262 evaluation nodes

Additional idea: endgame databases

- Idea:
  - Let's build a huge endgame database
  - Every endgame state in the database has exact evaluation score (win, draw, loss)
  - First endgame states with one piece, then with two pieces etc.

- This was done in 8x8 checkers:
  - **Chinook**: **Jonathan Schaeffer**, Robert Blake, Paul Lu and Martin Bryant
  - University of Alberta: http://webdocs.cs.ualberta.ca/~chinook/
  - Endgame database with 10 or less pieces has more than **39,000,000,000,000** different positions
  - They also have solved checkers. Total number of different positions: **500,995,484,682,338,672,639** ($5 * 10^{20}$)

- Deep search usually **reach to the endgame database**, where precise evaluation can be given.
- **Search from both starting and ending positions**


http://www.science.ualberta.ca/UndergraduateStudents/StudentNewsletters/September2012Newsletter.aspx

Links

- Gomoku:
  - http://en.wikipedia.org/wiki/Gomoku
- Minimax:
  - http://en.wikipedia.org/wiki/Minimax
- Alpha-beta pruning:
  - http://en.wikipedia.org/wiki/Alpha-beta_pruning
  - http://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.html
  - http://www.youtube.com/watch?v=xBXHtz4Gbdo