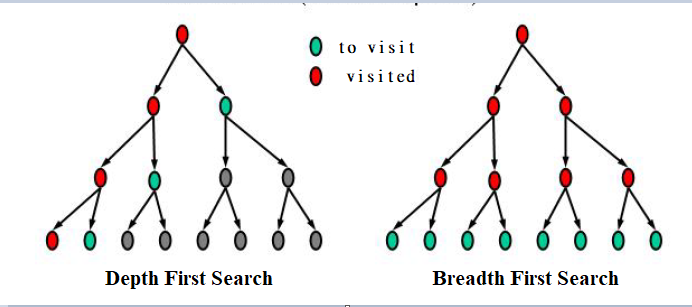
COM3103

Artificial Intelligence

1. Game Playing

Recall:

* Depth First Search
  + Search one branch completely to the end before other branches
* Breadth First Search
  + Search level by level



**Recall:**

**Heuristic** generally refers to a "rule of thumb", i.e., some "hints" that's helpful but not guaranteed to work.

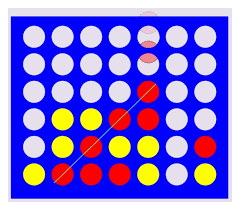
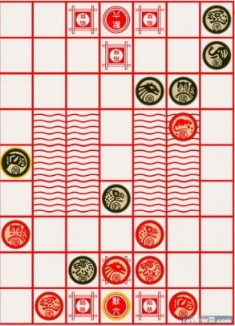
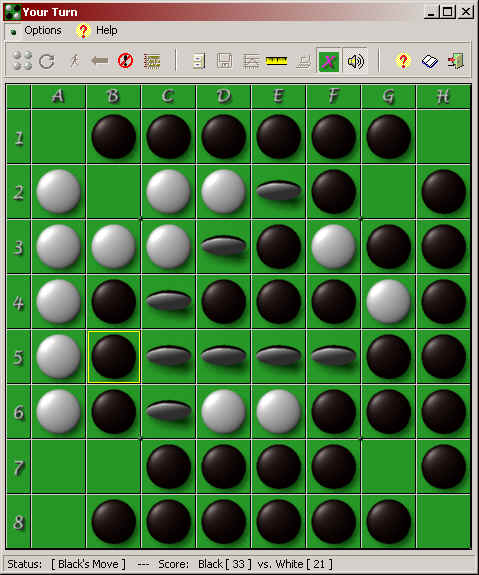
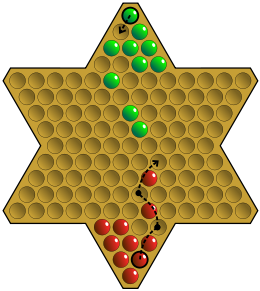
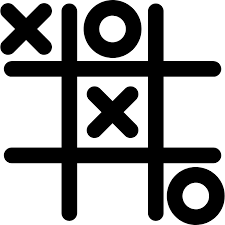
Use some simple functions to estimate how good is each node in the to-visit-list (i.e., estimate how close it is to the goal).

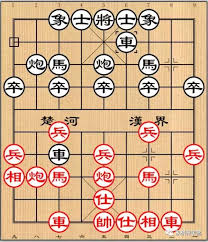
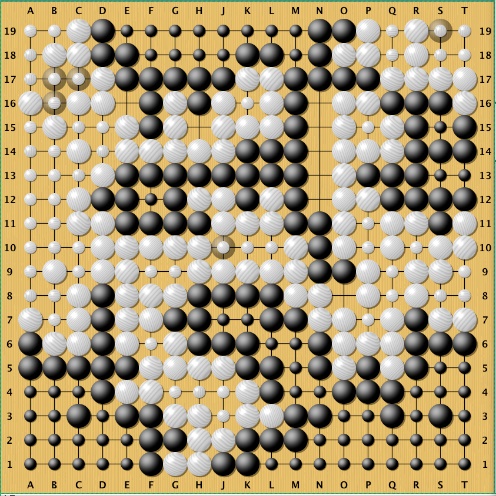
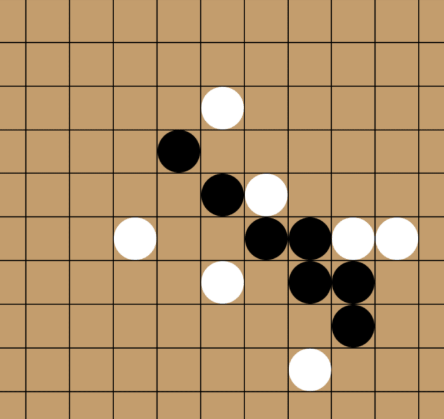
There is no guarantee that the estimation is always correct.

**Example: Estimated distance to goal**

**Two-players Board Games**

* + Two players alternating moves, against each other
  + **Perfect information** (all moves are observables) and **deterministic** (no random factors)



********

**Two-players Board Game**

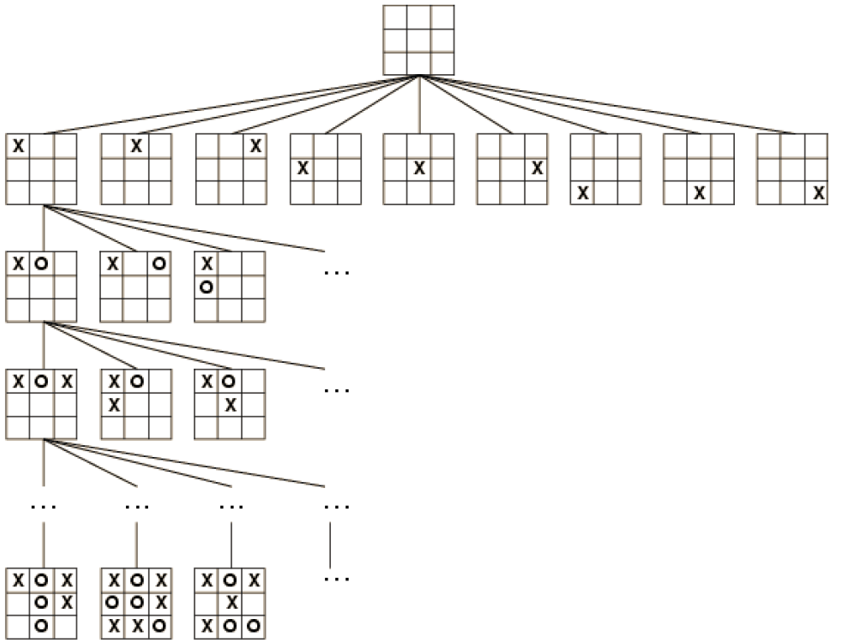
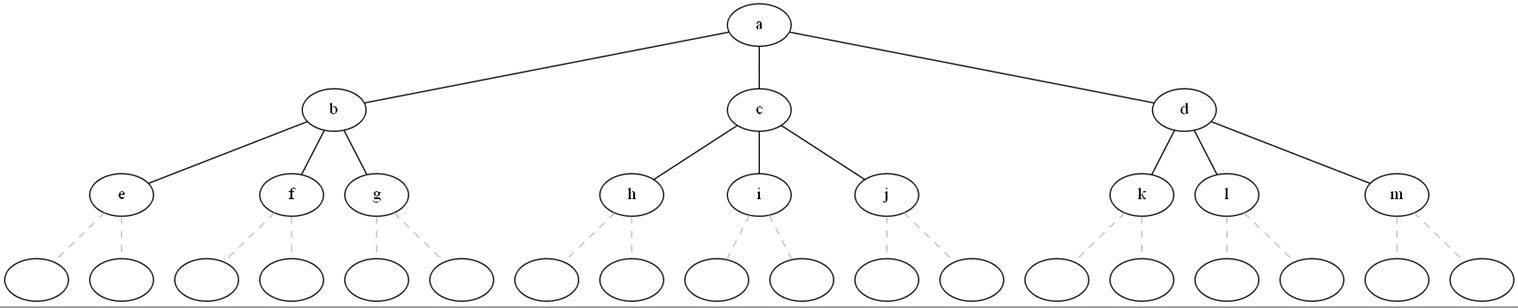
* + Starting state: starting board position
  + Goal state: winning board position
  + Operators: the legal moves
  + Scoring function (heuristic): a number for estimating the game situtation (who is winning)

Note:

* We do not need to return the complete path. We just need to decide the next move.
* It depends a lot on what the opponent will play!
* Need to "look ahead" multiple steps
  + "*If I take this move, how will the opponent respond, and then how can I counter …*"

**Game Tree**

* Every node represent a game state (e.g., the position of the pieces, and who to move next)
* Root node: the current state
* Goal: a winning state



Goal!

**Opponent moves**

**My moves**

**My moves**

**My moves**

**Opponent moves**

**My moves**

**Problem: game trees can be huge!**

* Recall: worst case time requirement is proportional to *bm*
  + *b*: branching factor, *m*: number of levels in the tree

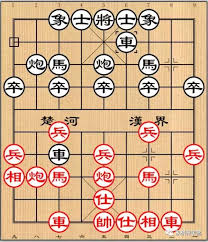
|  |  |  |
| --- | --- | --- |
| **Game** | **b** | **m** |
| Tic Tac Toe | 4 | 9 |
| Connect 4 | 4 | 39 |
| Reversi (黑白棋) | 10 | 58 |
| Gomuku (五子棋) | 30 | 210 |
| (Western) Chess | 35 | 80 |
| Chinese Chess (Xiangqi) | 38 | 95 |
| Weiqi (Go) | 150 | 250 |

* It is not practical to search the trees toward the end (except for very simply game like tic-tac-toe)
* We can only search the three up to a certain depth (before we can see a goal node)!

**Scoring Function**

* Assign a numeric value for evaluating a board position
  + High values - advantage for the first player
  + Low values - advantage for the second player
* Like a heuristic to evaluate a node.

Example:

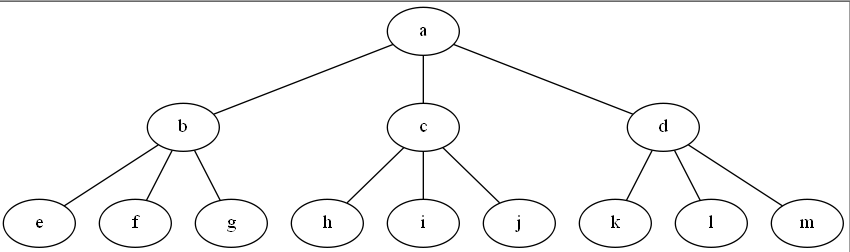
* Assign a number to each piece in chess games:
  + (Western) Chess: Queen – 9, rook – 5, bishop – 3, knight 3, pawn –1
  + Xiangqi (Chinese Chess): Chariot (車) – 9, Cannon (砲) – 4.5, Horse (馬) – 4, Elephant (象) – 2, Advisor (士) – 2, Soldier (兵) – 1 (2 after promotion)

**Look ahead and MiniMax**

* *Look ahead* the the possible outcomes after a certain numbers of moves (e.g., *p* moves)
* The number of levels to search is called "ply".
* For each outcome after p moves, evaluate the board situations using the scoring functions
* Pick the current move that will lead to the most advantages
* Problem: cannot control what the opponent will do next!
* Solution: Always assume the opponent will take the most unfavourable move. (*MiniMax*)

**MiniMax**

* Recall that the first player wants to maximize the score. The second players want to minimize it.
* Let's try to maximize our score after looking ahead two moves (i.e., ply = 2 )
* What move (b/c/d) would you play?



**My moves**

**Minimax**

**Opponent moves**

2

5

15

3

-3

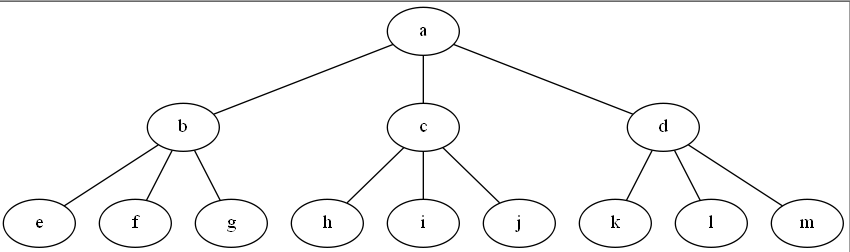
-2

8

3

12

* Consider who would the opponent respond for each move.
* For example, what would the opponent play if I play *b*? What about *c* and *d*?



**My moves**

8

12

3

**Opponent moves**

2

5

15

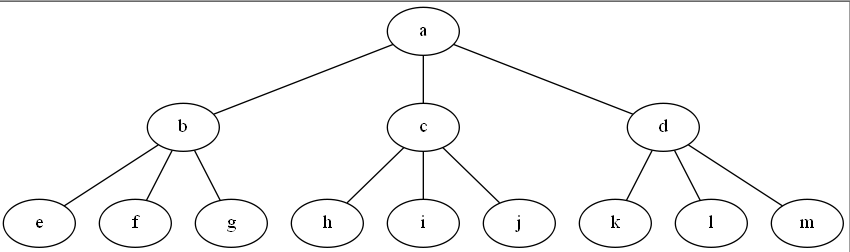
3

-3

-2

**Minimax**

* So which move (b/c/d) would you play?



**My moves**

2

-3

3

2

-3

4

12

3

**Opponent moves**

5

15

3

-2

**MiniMax (Idea)**

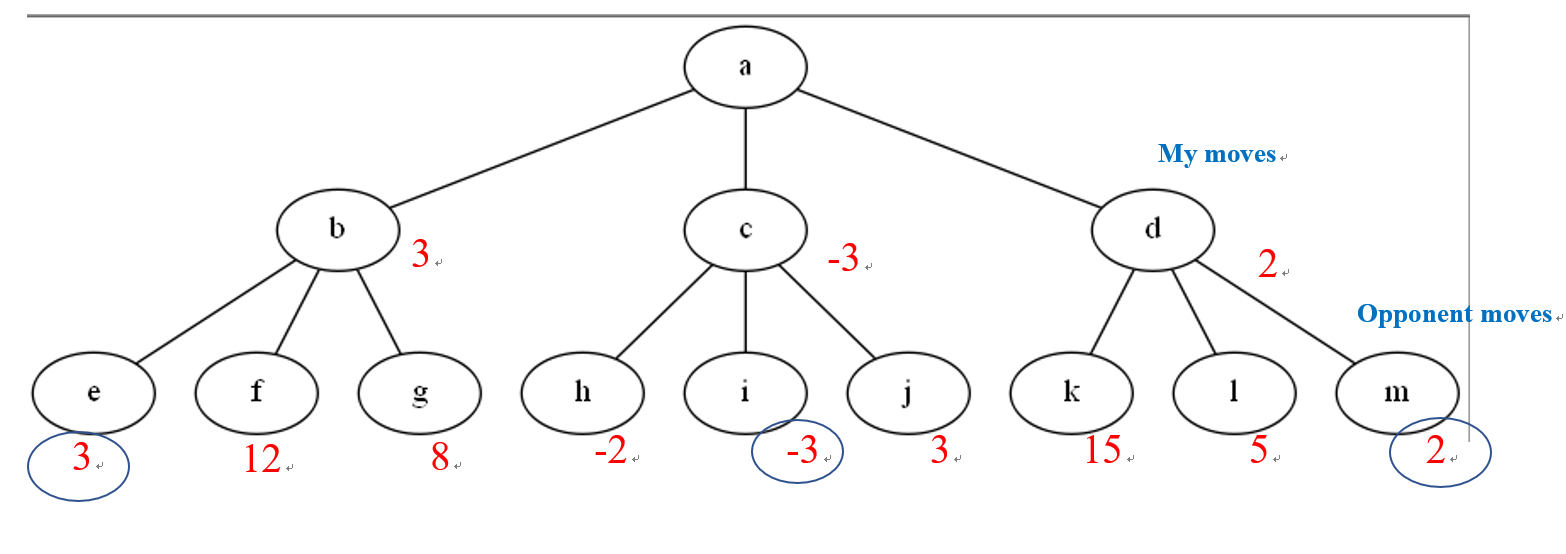
Objective: find out each of the current moves lead to the highest score (after max-ply moves)

Search the tree up to *max-ply* levels. For each node in the next levels, do:

If *max-ply* levels reached, or no more moves, or either player has won:

* + - Compute and return the score

Else:

* + - If the player 1's turn: report the maximum scores from the children nodes
    - If the player 2's turn: report the minimum scores from the children nodes

**MiniMax (Idea)**

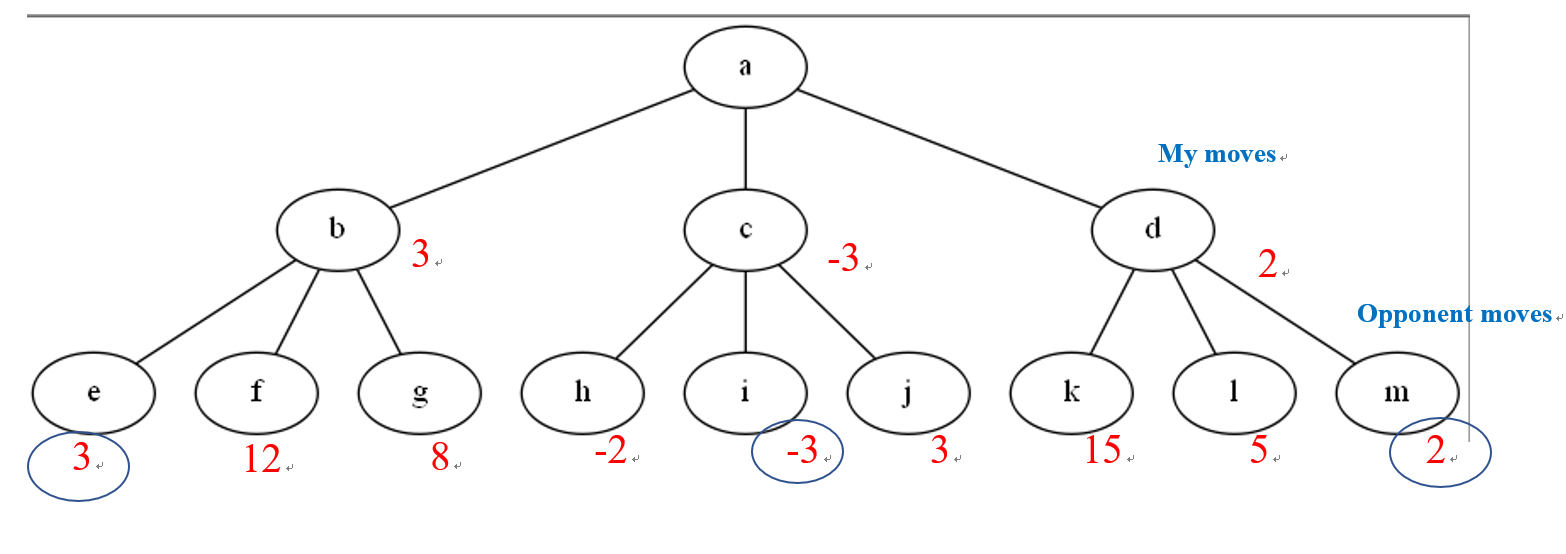
Objective: find out each of the current moves lead to the highest score (after max-ply moves)

Search the tree up to *max-ply* levels. For each node in the next levels, do:

If *max-ply* levels reached, or no more moves, or either player has won:

* + - Compute and return the score

Else:

* + - If the player 1's turn: report the maximum scores from the children nodes
    - If the player 2's turn: report the minimum scores from the children nodes

**MiniMax (Idea)**

Search the tree up to *max-ply* levels. For each node in the next levels, do:

If *max-ply* levels reached, or no more moves, or either player has won:

* + - Compute and return the score

Else:

* + - If the player 1's turn: report the maximum scores from the children nodes
    - If the player 2's turn: report the minimum scores from the children nodes

Question: How do we find the maximum/minimum score from the children in the next level?

Answer: call MiniMax again, but switch maximum to

minimum, and vice versa.

**MiniMax (We are player 1) (Idea)**

**We split it into two procedures as follows**

Max\_Value()

If *max-ply* levels reached, or no more moves, or either player has won:

* + - Compute and return the score

Else:

* + - For each child\_node:
      * call Min\_Value (child\_node)
    - Report the largest values reported above
* Min\_Value()

If *max-ply* levels reached, or no more moves, or either player has won:

* + - Compute and return the score

Else:

* + - For each child\_node:
      * call Max\_Value (child\_node)
    - Report the smallest values reported above

**More detailed version**

max\_value (current\_state, levels\_to\_search)

if levels\_to\_search is 0 or

no\_more\_moves\_possible or

one\_of\_the\_players\_ win:

return evaluation\_score( )

max\_score = -999999

for each child of current\_state

value = min\_value(child, levels\_to\_search -1 )

if value > max\_score

max\_score = value

return max\_score

(Note: we also need to return the best move if it is the top level)

**More detailed version**

min\_value (current\_state, levels\_to\_search)

if levels\_to\_search is 0 or

no\_more\_moves\_possible or

one\_of\_the\_players\_ win:

return evaluation\_score( )

min\_score = 9999999

for each child of current\_state

value = max\_value(child, levels\_to\_search -1 )

if value < min\_score

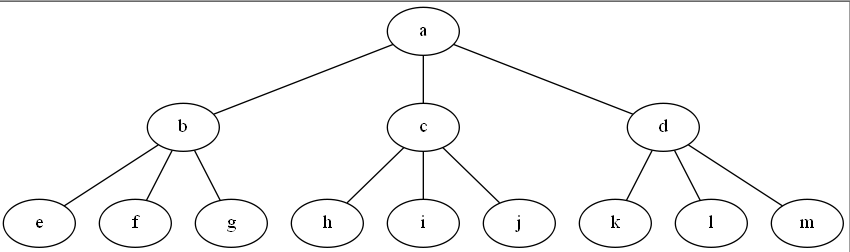
min\_score = value

return min\_score

(Note: we also need to return the best move if it is the top level)

**Alpha-Beta Pruning (further improvement on Minimax)**

* Suppose we are at node *a* (the maximizing player to move):  
  - node a has three children (*a, b, c*)  
  - *b* has reported a score of 3. we are now working on the children of *c*  
  - *h* has just reported as score of -2
  + Question: Do we really need to evaluate node *i* and *j*? Why or why not?



2

**max**

3

**min**

5

15

-3

3

-2

**min**

**min**

8

12

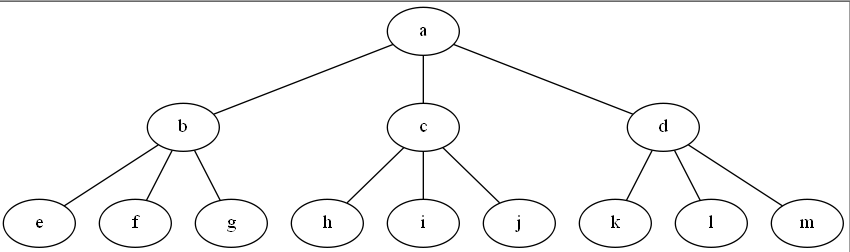
3

-2

**Alpha-Beta Pruning (further improvement on Minimax)**

* + Question: Do we really need to evaluate node *i* and *j*? Why or why not?

Answer: It is NOT necessary. The node *h* has already reported a value of -2. Therefore the score of node *c* cannot be higher than that. And since node *b* has a score of 3, node *c* will never be chosen by the higher level.



**max**

3

-2

**min**

**min**

**min**

2

-3

8

12

3

5

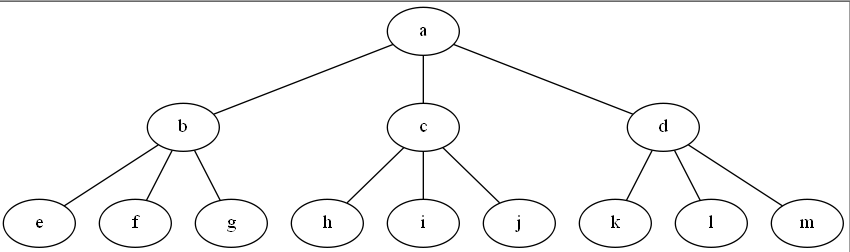
15

3

-2

**Alpha-Beta Pruning (further improvement on Minimax)**

* Maintain a best\_value we have seen so far
* Level levels can skip the rest of the node in the same branch of any of the nodes are already worse than the best value



**max**

3

2

3

-3

-2

**min**

**min**

**min**

2

8

12

3

5

15

3

-2

**Minimax with alpha-beta pruning**

max\_value (current\_state, num\_levels, previous\_best)

if num\_levels is 0 or

no\_more\_moves\_possible or

one\_of\_the\_players\_ win:

return evaluation\_score( )

max\_score = -999999

for each child of current\_state

value = min\_value(child, num\_levels -1, max\_score)

if value > max\_score

max\_score = value

if value > previous\_best

break

return max\_score

(Note: we also need to return the best move if it is the top level)

Minimax with alpha-beta pruning

min\_value (current\_state, num\_levels, previous\_best)

if num\_levels is 0 or

no\_more\_moves\_possible or

one\_of\_the\_players\_ win:

return evaluation\_score( )

min\_score = 9999999

for each child of current\_state

value = max\_value(child, num\_levels -1, min\_score )

if value < min\_score

min\_score = value

if value < previous\_best

break

return min\_score

(Note: we also need to return the best move if it is the top level)

Python Libraries Required for coming weeks

* numpy
* pygame

For IDLE, please go to your command prompt and type

pip install numpy

pip install pygame

(numpy should be included in Anaconda already)