COSC1125/1127 Artificial Intelligence

Week 3: Adversarial Search

Readings: [RN2] Chapter 6; or [RN3] Chapter 5

Assumptions so far…

So far: assumed agent has complete control of environment.

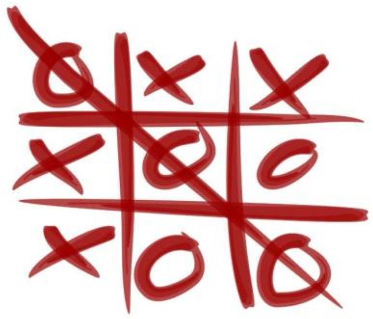
- State does not change unless the agent changes it.
  - All we need to compute is a single path to a goal state.
- Assumption not always reasonable!
  - Stochastic environment (e.g., the weather, traffic accidents).
  - Other agents whose interests conflict with yours.

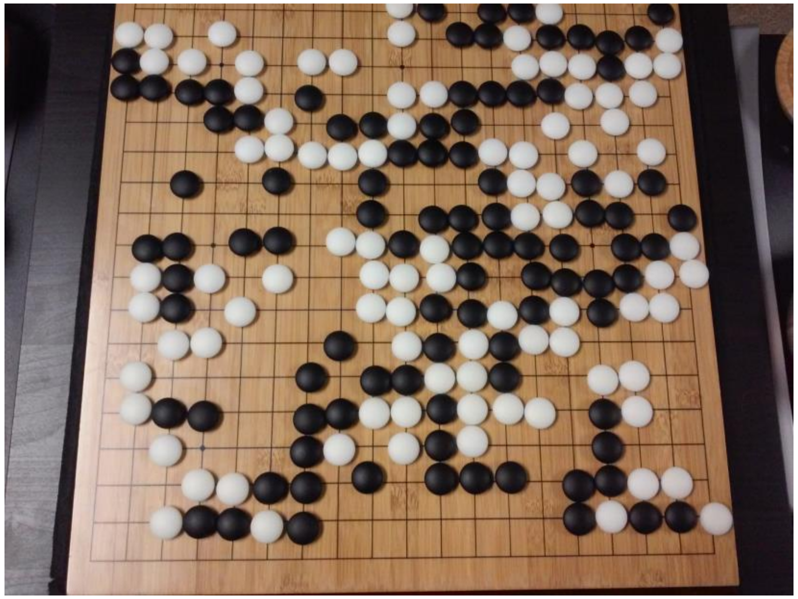
However, in a game setting:

- There may be two (or more) agents making changes to the world (the state).
- Each agent has their own interests
- e.g., each agent has a different goal; or assigns different costs to different paths/states
- Each agent tries to alter the world so as to best benefit itself.

Games considered here

- **Zero-sum games**: Fully competitive
  - If one play wins, the others lose (e.g. Poker) you win what the other player lose
- **Deterministic**: no chance involved
  - No dice, or random deals of cards, or coin flips, etc.
- **Perfect information**
  - All aspects of the state are fully observable e.g no hidden cards



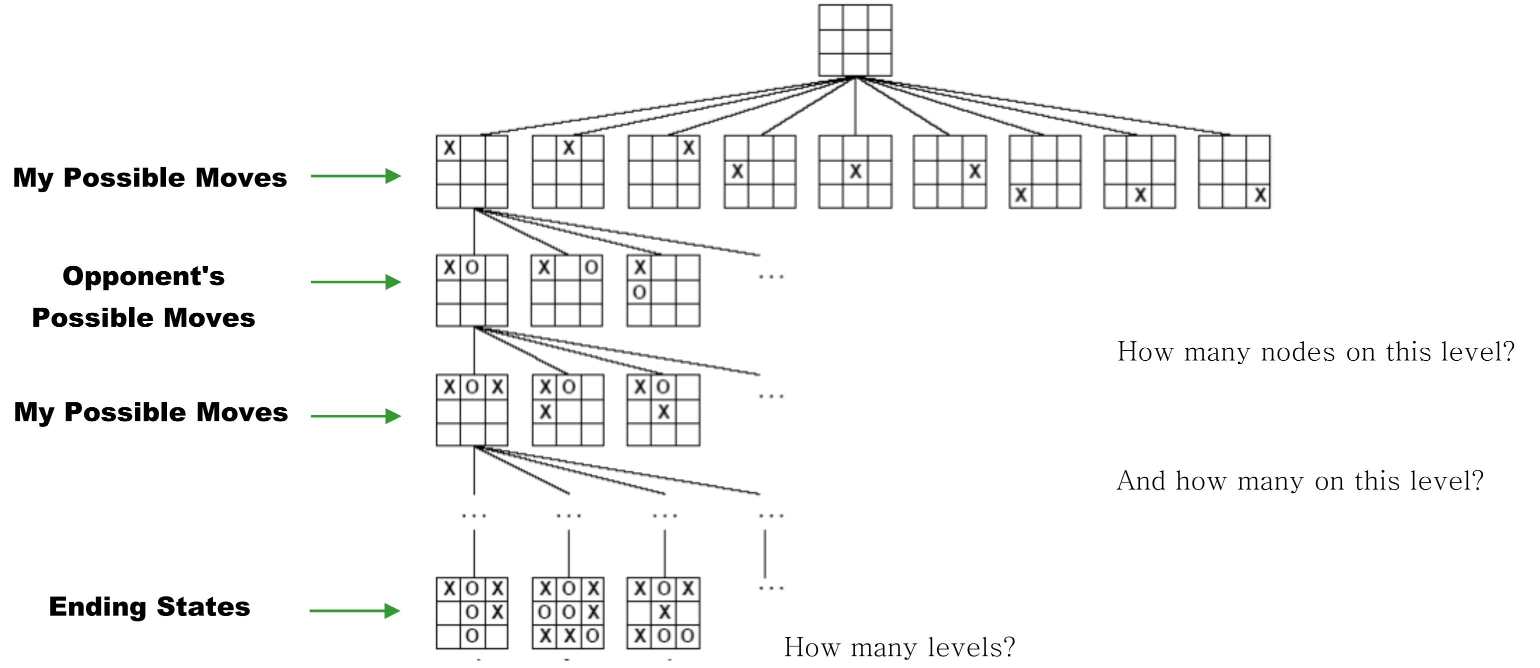


Game Playing

8-puzzle involves only one player. A game usually have two players, such as Chess, Tic-tac-toe, Checker, Othello etc.

Can we formulate game playing as search problems? The answer is yes.

However can we build a search tree to represent a game from the very beginning to every possible ending scenarios of the game? Such as the tree for tic-tac-toe:



My Possible Moves
Opponent's Possible Moves
My Possible Moves
Ending States
How many nodes on this level?
And how many on this level?
How many levels?

Game Playing…

It is almost impossible to build a search tree to represent the entire search space for a game. And it is not necessary to do so.

A realistic search method is to find the "best" next move based on the current state of the game. This move should ideally maximize the chance of winning.

To find this move, we need search through the search space starting from the current state, then choose the next move which is on the next level and on the path of "winning the game."

The opponent will reply against our move. Then we perform the search **again** based on the **new state** to find the next "best" move.

This process will repeat until the game is over.

In game playing, Minimax search is often used to find next moves.

PLY

Games usually have time limits and the search space is often enormous. So It is unlikely to perform a very deep search.

The depth of a search is measured as PLY (levels of the search).

2-ply: two levels, one move from me and one from the opponent.

7-ply: 7 levels of search.

For a chess player 4-ply look-ahead is hopeless.

4-ply	human novice
8-ply	human master, typical PC
12-ply	Kasparov, Deep Blue

Minimax Search

Minimax Search is good for deterministic, perfect-information games*.

Minimax search is different to the search strategies we discussed so far.

- At one level the search is to find the best move for me, a move with **MAX** outcome for me.

- At next level the search is to find the best move for opponent, a move with **MIN** outcome for me.

- It is unlikely to reach the ending states of a game (**Terminals**) due to the limited ply. So approximation of "how good a move is" must be used.

*[Games like Chess, Checkers, Go and Othello are deterministic, perfect-information games.]

*[Backgammon, Monopoly are not deterministic. Bridge and scrabble are imperfect-information games.]

Minimax Search...

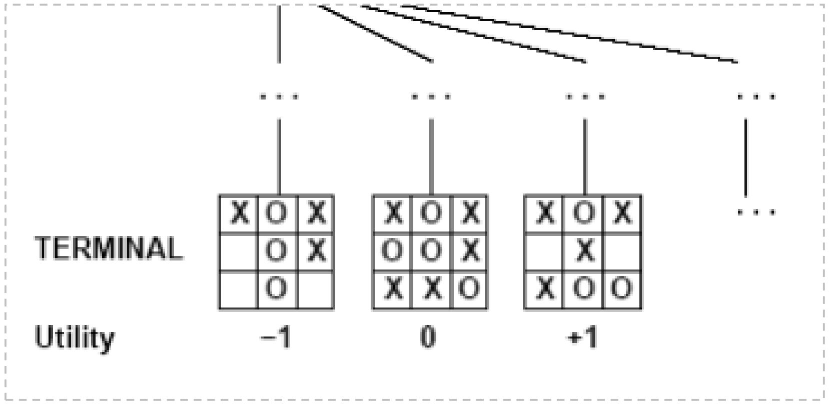
The components of MINIMAX Search are:

Initial state: the current state of game for which we want to pick a move.

Operators: each indicates a legal move (also called successor function).

Terminal test: test whether the end of game (terminals) are reached.

Utility/evaluation function: gives a numeric value for outcome of game.



Example of terminals in tic-tac-toe.

The utility value of states

+1 is win for me

0 is draw

−1 is loss for me.

Utility function applies on terminals, evaluation function applies on non-terminals. The higher return value the better for me. More details later.

Algorithm of Minimax Search

Minimax expands a game tree from the current state like **a depth-first search.**

Each **leave** nodes on the tree will be assigned a value by utility/evaluation function.

Assigning (minimax) values to each internal node from lower level to upper level.

- If the node is a **MAX** node, assign it with the **highest** value of its children.
- If the node is a **MIN** node, assign it with the **lowest** value of its children.

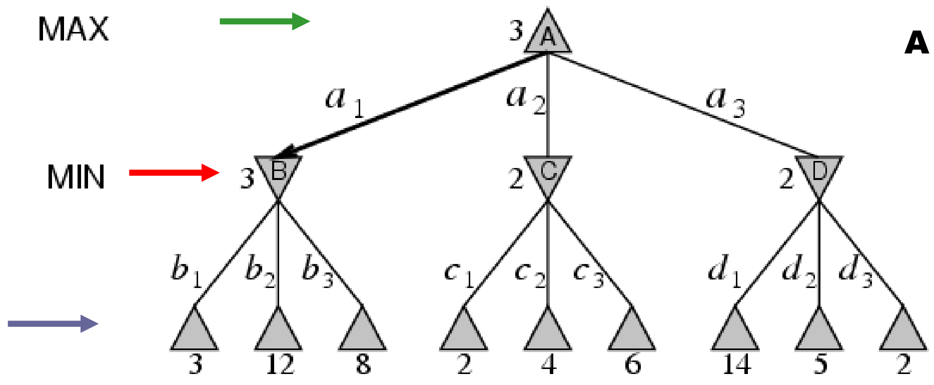
After assigning values, MAX chooses a **move at the top level** which lead to the highest value as the action.

The algorithm can be implemented by using recursion.

[Pseudo code in Chapter 6, the Text Book]

Algorithm of Minimax Search

Use a 2-ply game tree as example:



A two-ply game tree

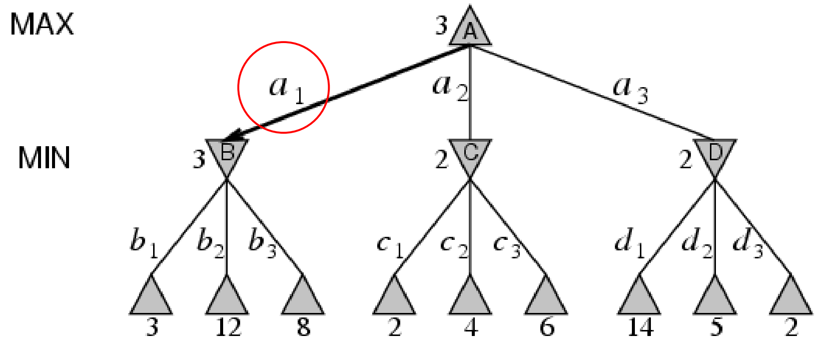
▲ is a "MAX node". It is *MAX*'s turn to move (to find the max value).
▼ is a "MIN node". It is *MIN*'s turn to move (to find the min value).

The bottom nodes are leave nodes on which utility/evaluation function applies.

The sequence of assigning values to nodes is:

$3 \rightarrow b_1$, $12 \rightarrow b_2$, $8 \rightarrow b_3$, $3 \rightarrow B$, $2 \rightarrow c_1$, $4 \rightarrow c_2$, $6 \rightarrow c_3$, $2 \rightarrow C$, $14 \rightarrow d_1$, $5 \rightarrow d_2$, $2 \rightarrow d_3$, $2 \rightarrow D$, $3 \rightarrow A$.

Algorithm of Minimax Search…



In this example MAX's best move is a_1, because it leads to the successor with the highest value. MIN's best reply is b_1, because it leads to the successor with the lowest value. Based on MIN's reply, MAX will perform a search again.

Notes:

- Always assume that the opponent will play perfectly (no blunders).
- Minimax search is applied every time when there is a move to be chosen.
- The start node of minimax search is hardly ever the starting position of the game.

Imperfect Decisions

For "interesting games" such as chess, there is no hope of generating a full tree from the current state to every possible terminals (end of game).

We could only expand the tree *n*-ply where *n* is determined by available time.

The leave nodes of the *n*-ply search tree unlikely are terminals. Without utility value how to know how "good" these nodes are in terms of winning the game?

We could apply an evaluation function to (non-terminal) leaves.
- The evaluation function must somehow distinguish between good and bad positions for me.
- The better the position for me, the higher the return value from the function.

Evaluation values are just ***estimates***, like heuristic values from heuristic functions.

Evaluation Function

Evaluation function just like heuristic function, which is to estimate the desirability of a state.

Typically it is a weighted sum of factors:

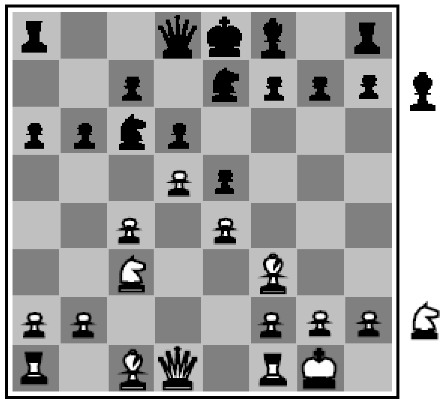
$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

Selecting factors and weights can be very tricky. It affects the performance of game playing.

Usually constructing an evaluation function needs input from human expert.

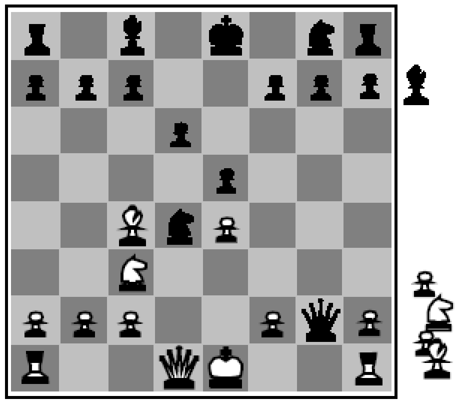
Also it needs experimentation to test and to adjust the function.

Evaluation Function…



Black to move

White slightly better



White to move

Black winning

For chess the weights and factors could be:

$w_1 f_1(s)$ = 9 * (number of white Queen – number of black queen)

$w_2 f_2(s)$ = 4 * (number of white Bishop – number of black Bishop)

$w_3 f_3(s)$ = 3 * (number of white knight – number of black knight)

$w_4 f_4(s)$ = 1 * (number of white pawn – number of black pawn)

… so on and so forth…

By this simple function we can estimate which player is closer to winning.

Alpha–beta Pruning

Even with a good evaluation function, search for 8–ply game tree (to match with human master) could be computationally very costly.

[The branching factor of chess $\approx$35. How many nodes to search on a 8–ply game tree?]

Fortunately it is possible to compute the correct decision without looking at every node in the search tree.

The particular technique is called **alpha–beta pruning**, which returns the same move as minimax would but prunes away branches that cannot possibly influence the final outcome.

Actually the unnecessary branches are not pruned, but never expanded.

Alpha-beta Pruning…

Alpha (α) is a value for MAX, the best (highest) score along the path to the state.

Beta (β) is a value for MIN, the best (lowest) score along the path to the state.

Alpha and beta value update along the search.

During the search, discontinue expanding a node if

- it is a MAX and its value is higher than or equal to the value of its parent (β of a MIN). That means this MAX cannot result a MIN with lower β. **[Alpha pruning]**

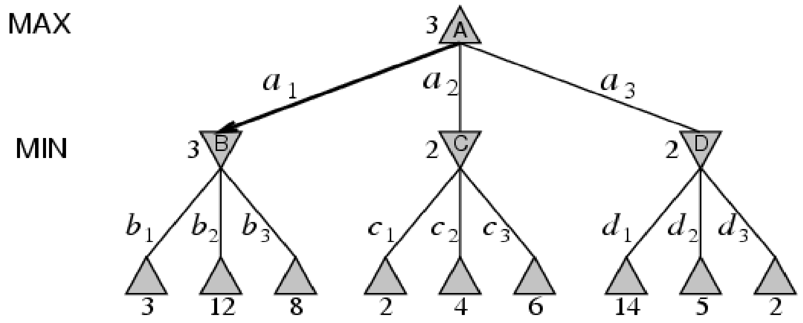
- It is a MIN and its value is lower than or equal to the value of its parent (α of a MAX). That means this MIN cannot result a MAX with higher α. **[Beta pruning]**

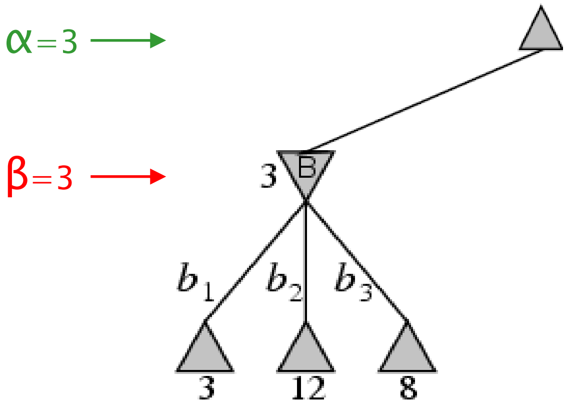
In both of above scenarios the game search can not be improved. Further expansion of these nodes is unnecessary.

So the algorithm is called alpha-beta pruning.

α–β Pruning Example 1

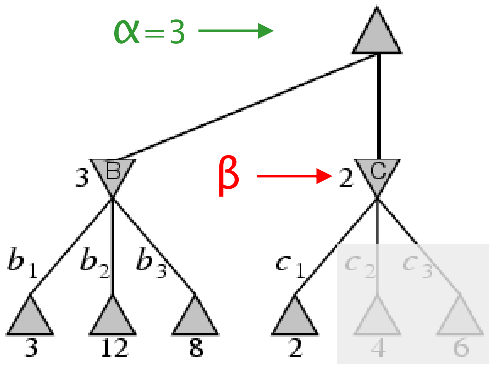
We use this 2-ply game tree again as example for α–β pruning.





- Expand the first branch.
- Get evaluation values for each leave node.
- So the value for B (β) is 3.
- So the root value (α) is currently 3.

α−β Pruning Example 2



- Expand node C and c_1
- Get evaluation values for c_1
- So the current value for C is 2.
- Value of C $<$ Value of root, 3.
- So **no further expansion** on C.

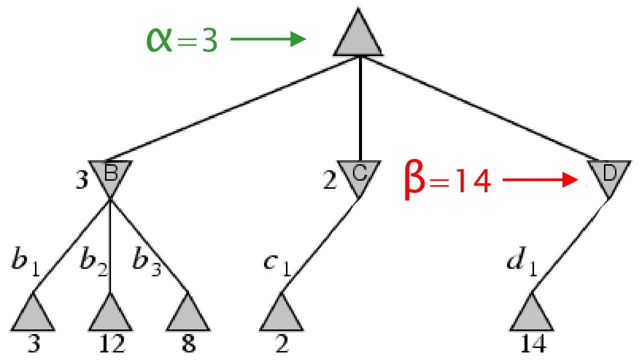
A beta-pruning is performed here – a MIN doesn't have a higher value than its MAX parent.

What if we expand c_2 and c_3? The values of c_2 and c_3 are 4, 6. So β of C will remain 2. Minimax search will still not choose C (because at least B has a higher value 3).

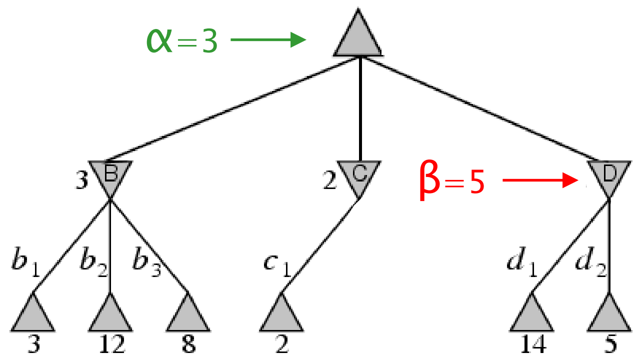
The search outcome is not affected by the value of c_2 and c_3.

So expanding c_2 and c_3 are unnecessary. They can be pruned.

α−β Pruning Example 3



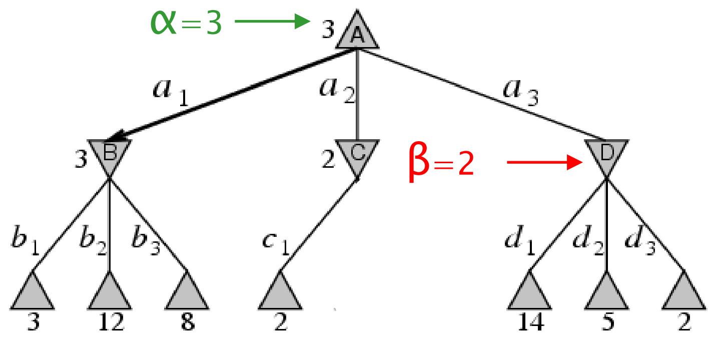
- Expand node D and d_1
- Get evaluation values for d_1
- So the current value for D is 14.
- Value of D > Value of root, 3.
- So **continue** the expansion on D.



- Expand and evaluate d_2
- So the current value for D is 5.
- Value of D > Value of root, 3.
- So **continue** the expansion on D.

Can we stop expansion at d_1 or d_2? No! What if the unexpanded node has a very high value, say $d_3 = 18$? Then we will miss an optimal path.

α−β Pruning Example 4



- Expand and evaluate d_3
- So the current value for D is 2.
- No more expansion (d_3 is the last child of D anyway).
- MAX chooses a_1 as the next move.

The outcome of this alpha-beta pruning is same with that from minimax.

Less nodes are expanded here. So alpha-beta pruning is more efficient.

However the expansion of d_1 or d_2 seems not worthwhile. The low value of d_3 bring down the value of D to 2.

Question: can we improve the pruning algorithm to avoid this kind of situations? How?

State-of-the-art Game Programs

Chess: Human world champion Gray Kasparov defeated by Deep Blue in six game match in 1997. Deep blue searches 200 million positions per second, uses very sophisticated evaluation function, and undisclosed methods for extending some line of search up to 40 ply. Currently a good program on PC can match with human world champion.

Othello: humans are no match for computers, who are too good.

Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. It runs on regular PCs and uses alpha-beta search.

Go: In October 2015, **AlphaGo** became the first computer Go program to beat a human professional Go player on a full-sized 19×19 board.

Summary

- Game playing, terminal, utility
- Minimax search and alpha-beta pruning
- imperfect decision & evaluation function in game playing

Start working on assignment#2 Connect4.

Acknowledgement: the slides were developed based on notes from Russell & Norvig, by several computer science staff members over the years.