Adversarial Search

10S3001 – Artificial Intelligence

Samuel I. G. Situmeang

Faculty of Informatics and Electrical Engineering

Objectives

Students are able:

- to explain the adversarial search background,
- to understand how the minimax algorithm works and how to use it,
- to understand how alpha-beta pruning algorithm works and how to use it, and
- to understand the game theory of stochastic games.

Siswa mampu:

- untuk menjelaskan latar belakang pencarian *adversarial*,
- untuk memahami cara kerja algoritma *minimax* dan cara menggunakannya,
- untuk memahami cara kerja algoritma *alpha-beta pruning* dan cara menggunakannya, dan
- untuk memahami teori permainan permainan stokastik.

Adversarial Search

Adversarial Search

- Adversarial search problems ≡ games, occur in multi-agent competitive environments.
- There is an **opponent** we can't control planning against us!
- Game vs. search: optimal solution is not a sequence of actions but a **strategy** (policy) If opponent does a, agent does b, else if opponent does c, agent does d, etc.
- Tedious and fragile if hard-coded (i.e., implemented with rules).
- Good news: Games are modeled as **search problems** and use **heuristic evaluation** functions.
- Games are interesting to AI because they are too hard to solve.
  - Chess has a branching factor of 35, with 35^{100} nodes $\approx 10^{154}$.
  - Need to make some decision even when the optimal decision is infeasible.

Pencarian *adversarial* adalah pencarian yang diterapkan pada situasi di mana agen melakukan perencanaan (*planning*) sementara agen lain melawan atau menentang agen tersebut. Perencanaan (alias perencanaan otomatis atau perencanaan dengan kecerdasan buatan) adalah proses mencari rencana, yang merupakan urutan tindakan yang membawa agen/dunia dari keadaan awal (*initial state*) ke satu atau beberapa keadaan tujuan (*goal state*), atau kebijakan (*policy*). Perencanaan tidak hanya menggunakan algoritma pencarian, misalnya algoritma pencarian *state-space* seperti **A*** untuk menemukan rencana, tetapi perencanaan juga melibatkan pemodelan masalah dengan menggambarkannya dalam beberapa cara.

Perlu juga dipahami, bahwa teknik pencarian yang kita bahas di perkuliahan yang lalu, seperti *uninformed search*, *informed search*, dan *local search* tidak memperhitungkan *state* dari pihak lawan. Sementara *adversarial search* peduli pada *state* lawan. Contoh Implementasinya adalah pada permainan catur. Dalam permainan ini tentu saja banyak sekali kemungkinan bagi sebuah pion catur bergerak dan juga banyak konfigurasi keadaan papan catur, dan agen (pemain) dapat menentukan strategi yang efektif dalam menghadapi lawannya berdasarkan berbagai kemungkinan pergerakan yang mungkin ditempuh.

Ada beberapa algoritma yang dapat kita gunakan dalam rangka penerapan *adversarial search*, misalnya *minimax algorithm* dan *alpha-beta pruning*.

Types of Games

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

We are mostly interested in deterministic games, fully observable environments, zero-sum, where two agents act alternately.

Dalam kecerdasan buatan, kita mengklasifikasikan game dalam dua dimensi yang berbeda. Jika permainan memiliki informasi yang sempurna, misalnya agen dapat melihat seluruh papan, maka kita menyebutnya permainan dengan informasi yang sempurna (*perfect information*). Contohnya Catur, *Checkers*, *Go*, *Othello*, *Backgammon*, dan Monopoli. Jika agen tidak memiliki informasi yang sempurna (*imperfect information*), berarti agen tidak memiliki gambaran utuh tentang apa yang terjadi, misalnya agen tidak dapat melihat kartu lawan. Contohnya *Battleship*, *Blind Tic-tac-toe, Bridge*, *Poker*, *Scrabble*, *Nuclear War*, dll.

Dimensi yang kedua adalah permainan deterministik versus permainan non-deterministik, yang sebenarnya disebutkan di sini sebagai peluang. Permainan non-deterministik juga disebut permainan stokastik, stokastik berarti ada faktor peluang yang terlibat dalam jalannya permainan. Dan faktor peluang ini diindikasikan dengan adanya pengocokan kartu, melempar dadu, dll. dalam permainan.

Maka dengan demikian, kita dapat simpulkan sebagai berikut.

- Contoh permainan deterministik yang memiliki informasi sempurna contohnya adalah Catur, *Checkers*, *Go*, dan *Othello*.
- Contoh permainan deterministik yang memiliki informasi tidak sempurna adalah

Battleship dan *Blind Tic-tac-toe*.

- Contoh permainan non-deterministik yang memiliki informasi sempurna adalah *Backgammon* dan Monopoli.
- Contoh permainan non-deterministik yang memiliki informasi tidak sempurna adalah *Bridge*, *Poker*, *Scrabble*, dan *Nuclear War*.

Dalam kuliah ini, kita akan fokus pada permainan deterministik yang memiliki informasi yang sempurna.

Zero-sum Games

- Adversarial: Pure competition.
- Agents have different values on the outcomes.
- One agent maximizes one single value, while the other minimizes it.
- Each move by one of the players is called a “ply.”

One function: one agents maximizes it and one minimizes it!

Each move in the game by one of the players is called a ply. So we have here one objective function, we want one of them is maximizing it, and one of them is minimizing it.

Minimax Algorithm

Embedded Thinking...

Embedded thinking or backward reasoning!

- One agent is trying to figure out what to do.
- How to decide? He thinks about the consequences of the possible actions.
- He needs to think about his opponent as well...
- The opponent is also thinking about what to do etc.
- Each will imagine what would be the response from the opponent to their actions.
- This entails an embedded thinking.

Zero-sum game adalah representasi matematis dari suatu situasi di mana keuntungan satu orang akan menjadi kerugian orang lain. Jika total keuntungan peserta dikurangi kerugian, maka jumlah tersebut akan menjadi nol.

Zero-sum game melibatkan *embedded thinking* di mana satu agen atau pemain mencoba untuk mencari tahu:

- Apa yang harus dilakukan.
- Bagaimana memutuskan langkah?
- Perlu memikirkan lawannya juga
- Lawan juga berpikir apa yang harus dilakukan

Masing-masing pemain berusaha untuk mengetahui respon lawannya terhadap tindakan mereka. Ini membutuhkan pemikiran tertanam (*embedded thinking*) atau penalaran mundur (*backward reasoning*) untuk menyelesaikan masalah *game* dengan kecerdasan buatan.

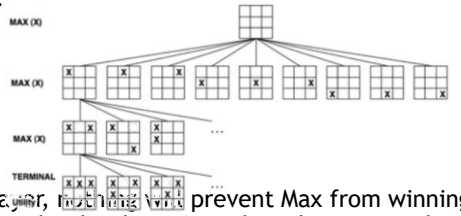
Formalization

- The **initial state**
- **Player(s)**: defines which player has the move in state s. Usually taking turns.
- **Actions(s)**: returns the set of legal moves in s.
- **Transition function**: $S \times A \rightarrow S$ defines the result of a move.
- Terminal_test: True when the game is over, False otherwise. States where game ends are called **terminal states**.
- $Utility(s, p)$: **utility function** or objective function for a game that ends in terminal state s for player p. In Chess, the outcome is a win, loss, or draw with values $+1$, 0, $1/2$. For tic-tac-toe we can use a utility of $+1$, -1, 0.

Kita dapat memformalkan masalah dalam bentuk sebagai berikut. Jadi kita mulai dengan beberapa keadaan (*state*) awal, sebut saja nol, misalnya.

Single Player…

- Assume we have a tic-tac-toe with one player.
- Let's call him Max and have him play three moves only for the sake of the example.

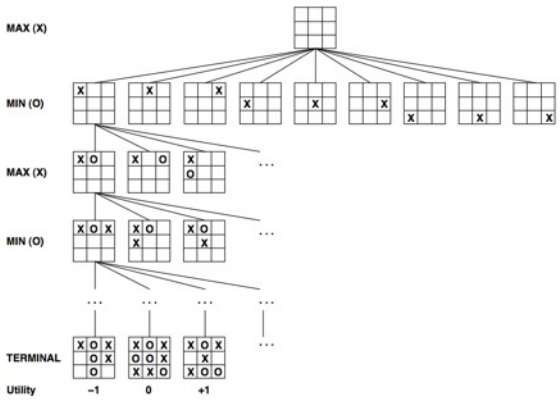


- In the case of one player, nothing will prevent Max from winning (choose the path that leads to the desired utility here 1), unless there is another player who will do everything to make Max lose.

Adversarial Search: Minimax

- Two players: Max and Min
- Players alternate turns
- Max moves first
- Max maximizes results
- Min minimizes the result
- Compute each node's minimax value's the best achievable utility against an optimal adversary
- Minimax value ≡ best achievable payoff against best play

Minimax Example



Adversarial Search: Minimax

- Find the optimal strategy for Max:
  - Depth-first search of the game tree
  - An optimal leaf node could appear at any depth of the tree
  - Minimax principle: compute the utility of being in a state assuming both players play optimally from there until the end of the game
  - Propagate minimax values up the tree once terminal nodes are discovered

Adversarial Search: Minimax

- If state is terminal node: Value is utility(state)
- If state is MAX node: Value is highest value of all successor node values (children)
- If state is MIN node: Value is lowest value of all successor node values (children)

Adversarial Search: Minimax

- For a state s minimax(s) =

$$\begin{cases} Utility(s) & \text{if Terminal-test(s)} \\ max_{a \in Actions(s)} \text{ minimax(Result(s,a))} & \text{if Player(s) = Max} \\ min_{a \in Actions(s)} \text{ minimax(Result(s,a))} & \text{if Player(s) = Min} \end{cases}$$

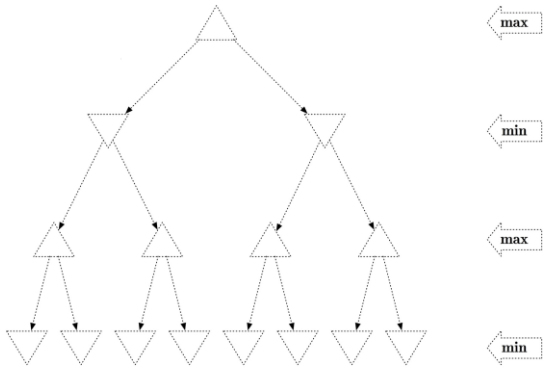
The Minimax Algorithm

```
/* Find the child state with the lowest utility value */

function MINIMIZE(state)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨minChild, minUtility⟩ = ⟨NULL, ∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MAXIMIZE(child)

        if utility < minUtility:
            ⟨minChild, minUtility⟩ = ⟨child, utility⟩

    return ⟨minChild, minUtility⟩
```

```
/* Find the child state with the highest utility value */

function MAXIMIZE(state)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨maxChild, maxUtility⟩ = ⟨NULL, −∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MINIMIZE(child)

        if utility > maxUtility:
            ⟨maxChild, maxUtility⟩ = ⟨child, utility⟩

    return ⟨maxChild, maxUtility⟩
```

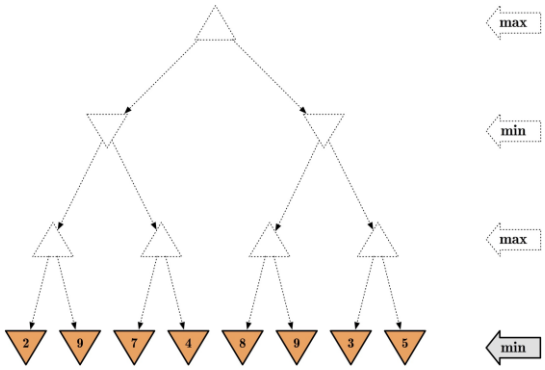
```
/* Find the child state with the highest utility value */

function DECISION(state)
    returns STATE :

    ⟨child, _⟩ = MAXIMIZE(state)

    return child
```

Minimax Example

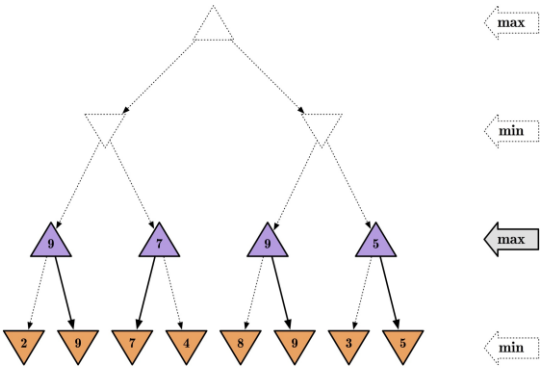


Game Tree

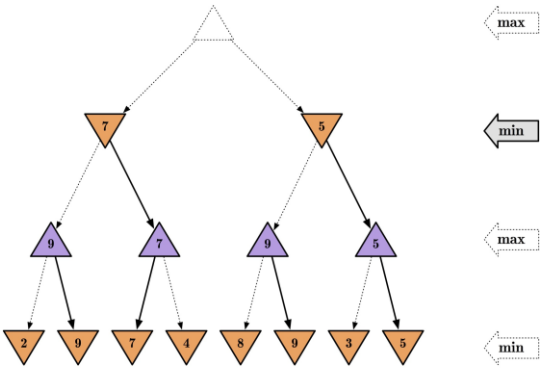
Minimax Example



Minimax Example

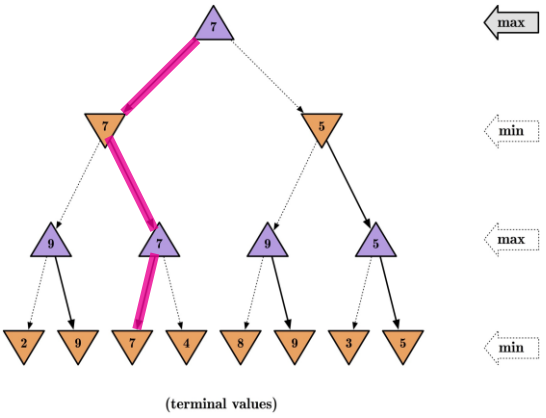


Minimax Example



(terminal values)

Minimax Example



This is the-- when both of them play optimally **this is the best choice for max to get maximum utility.**

Properties of Minimax

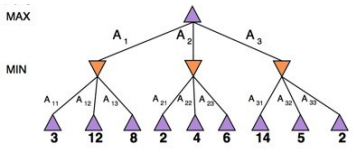
- Optimal (opponent plays optimally) and complete (finite tree)
- DFS time: $O(b^m)$
- DFS space: $O(bm)$
  - Tic-Tac-Toe
    - ≈ 5 legal moves on average, total of 9 moves (9 plies).
    - $5^9 = 1,953,125$
    - $9! = 362,880$ terminal nodes
  - Chess
    - $b \approx 35$ (average branching factor)
    - $d \approx 100$ (depth of game tree for a typical game)
    - $b^d \approx 35^{100} \approx 10^{154}$ nodes
  - Go branching factor starts at $361 (19 \times 19 \text{ board})$

Case of Limited Resources

- **Problem:** In real games, we are limited in time, so we can't search the leaves.
- To be practical and run in a reasonable amount of time, minimax can only search to some depth.
- More plies make a big difference.
- **Solution:**
  1. Replace terminal utilities with an evaluation function for non-terminal positions.
  2. Use Iterative Deepening Search (IDS).
  3. Use pruning: eliminate large parts of the tree.

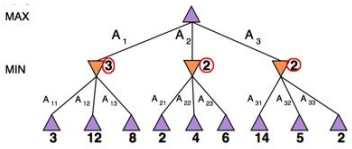
Alpha-Beta Pruning

$\alpha - \beta$ Pruning



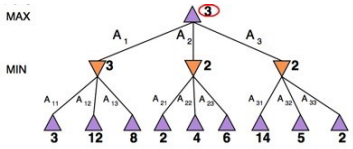
- A two-ply game tree.

$\alpha - \beta$ Pruning



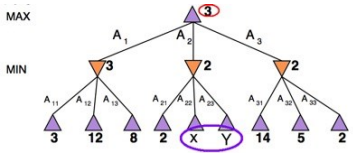
$\alpha - \beta$ Pruning

Which values are necessary?



The question here is whether we need to really explore all those nodes in order to find this value 3 that represents the maximum utility for Max. And the answer is no.

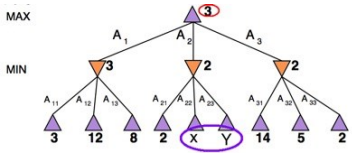
$\alpha - \beta$ Pruning



$$Minimax(root) = \max(\min(3, 12, 8), \min(2, X, Y), \min(14, 5, 2))$$

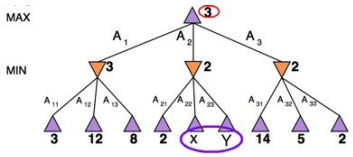
So for example maybe these x, y, here are not that really needed to be calculated.

$\alpha - \beta$ Pruning



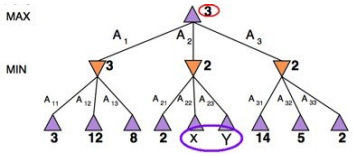
$$Minimax(root) = \max(\min(3, 12, 8), \min(2, X, Y), \min(14, 5, 2))$$
$$= \max(3, \min(2, X, Y), 2)$$

$\alpha - \beta$ Pruning



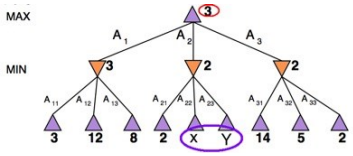
$$Minimax(root) = \max(\min(3,12,8), \min(2,X,Y), \min(14,5,2))$$
$$= \max(3, \min(2,X,Y), 2)$$
$$= \max(3,Z,2) \qquad where\ Z = \min(2,X,Y) \leq 2$$

$\alpha - \beta$ Pruning



$$Minimax(root) = \max(\min(3,12,8), \min(2,X,Y), \min(14,5,2))$$
$$= \max(3, \min(2,X,Y), 2)$$
$$= \max(3,Z,2) \qquad where\ Z = \min(2,X,Y) \leq 2$$
$$= 3$$

$\alpha - \beta$ Pruning



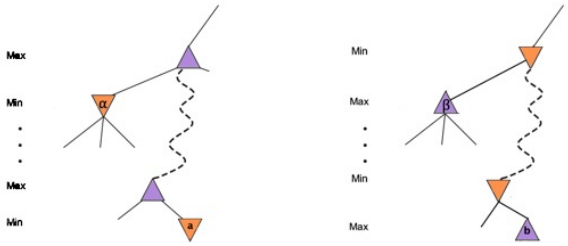
$$Minimax(root) = \max(\min(3,12,8), \min(2,X,Y), \min(14,5,2))$$
$$= \max(3, \min(2,X,Y), 2)$$
$$= \max(3,Z,2) \qquad where\ Z = \min(2,X,Y) \leq 2$$
$$= 3$$

Minimax decisions are independent of the values of X and Y.

$\alpha - \beta$ Pruning

- **Strategy**: Just like minimax, it performs a DFS.
- **Parameters**: Keep track of two bounds
  - α: largest value for Max across seen children (current lower bound on MAX's outcome).
  - β: lowest value for MIN across seen children (current upper bound on MIN's outcome).
- **Initialization**: $\alpha = -\infty, \beta = \infty$
- **Propagation**: Send α, β values *down* during the search to be used for pruning.
  - Update α, β values by *propagating upwards* values of terminal nodes.
  - Update α only at Max nodes and update β only at Min nodes.
- **Pruning**: Prune any remaining branches whenever $\alpha \geq \beta$.

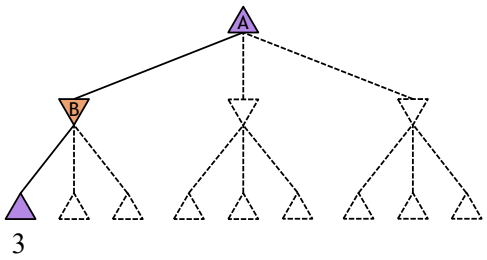
$\alpha - \beta$ Pruning



- If α is better than a for Max, then Max will avoid it, that is prune that branch.
- If β is better than b for Min, then Min will avoid it, that is prune that branch.

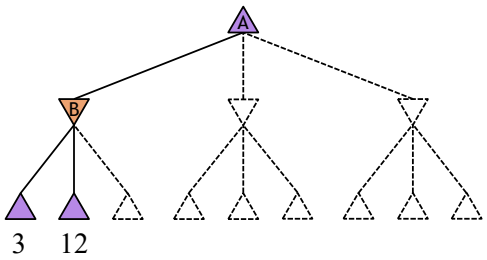
$\alpha - \beta$ Pruning

(a)



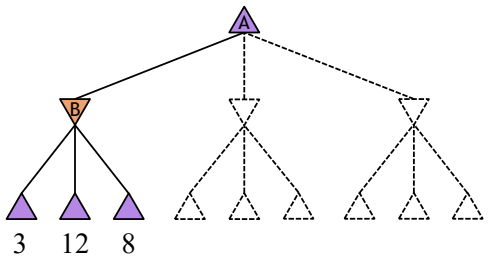
$\alpha - \beta$ Pruning

(b)



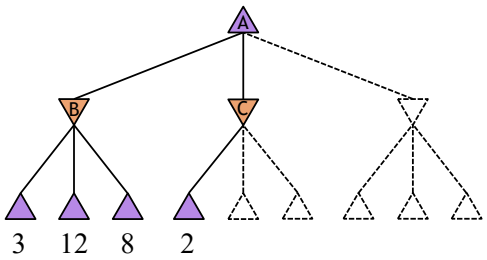
$\alpha - \beta$ Pruning

(c)



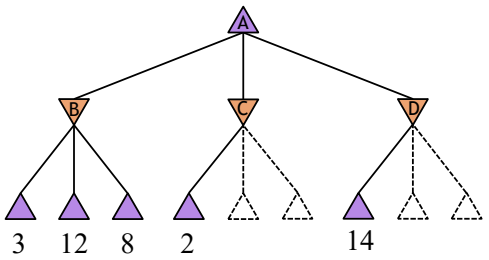
$\alpha - \beta$ Pruning

(d)



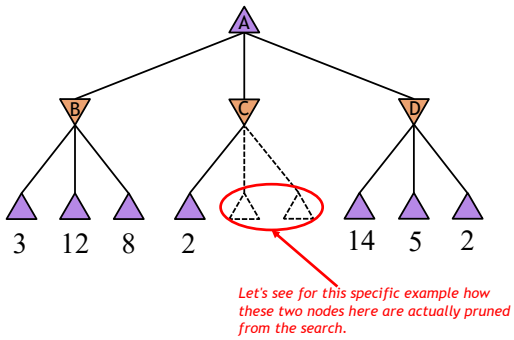
$\alpha - \beta$ Pruning

(e)

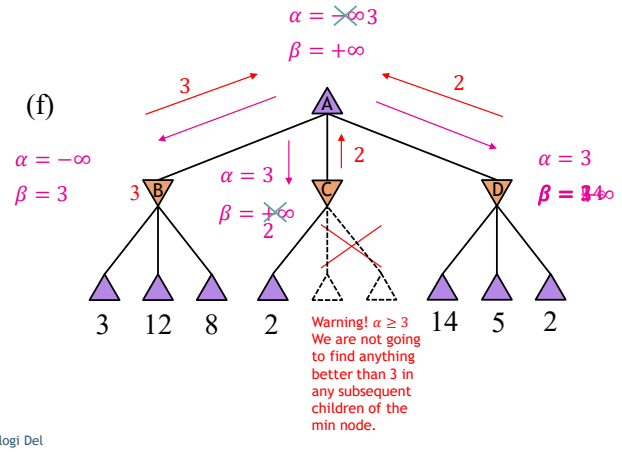


$\alpha - \beta$ Pruning

(f)

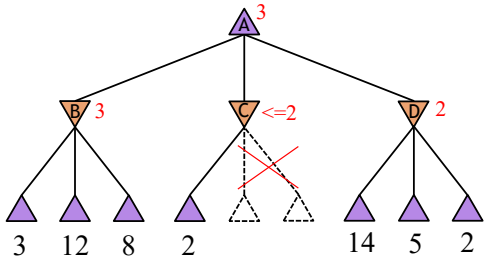


$\alpha - \beta$ Pruning



So let's start with the initial values for alpha and beta being **minus infinity for alpha, and beta being plus infinity.**
The first child is 3, then beta will change its mind. The value is no longer plus infinity, **we have a better min value for Min, which would** be, in this case, equal to 3.
The second child explored would be 12. Now, 12 is not better than 3 for Min and 8 is not better than 3 for Min. So in this case, Min declares that its minimum value **based on its children is 3**.
And we will send back this 3 value to the Max that's called the child its first child.
…

$\alpha - \beta$ Pruning



$\alpha - \beta$ Pruning

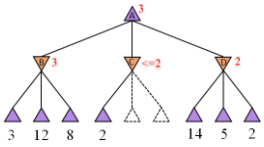
```
/* Find the child state with the lowest utility value */
function MINIMIZE(state, α, β)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨minChild, minUtility⟩ = ⟨NULL, ∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MAXIMIZE(child, α, β)

        if utility < minUtility:
            ⟨minChild, minUtility⟩ = ⟨child, utility⟩

        if minUtility ≤ α:
            break

        if minUtility < β:
            β = minUtility

    return ⟨minChild, minUtility⟩
```

```
/* Find the child state with the highest utility value */
function MAXIMIZE(state, α, β)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨maxChild, maxUtility⟩ = ⟨NULL, −∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MINIMIZE(child, α, β)

        if utility > maxUtility:
            ⟨maxChild, maxUtility⟩ = ⟨child, utility⟩

        if maxUtility ≥ β:
            break

        if maxUtility > α:
            α = maxUtility

    return ⟨maxChild, maxUtility⟩
```

```
/* Find the child state with the highest utility value */
function DECISION(state)
    returns STATE :

    ⟨child, _⟩ = MAXIMIZE(state, −∞, ∞)

    return child
```


So now, here's the alpha-beta pruning algorithm that's looks very similar to, as you can see, to minimax algorithm. **We have again a decision function**

Move Ordering



- It does matter as it affects the effectiveness of $\alpha - \beta$ pruning.
- Example: We could not prune any successor of D because the worst successors for Min were generated first. If the third one (leaf 2) was generated first we would have pruned the two others (14 and 5).
- Idea of ordering: examine first successors that are likely best.

Move Ordering

- **Worst ordering**: no pruning happens (best moves are on the right of the game tree). Complexity $O(b^m)$.
- **Ideal ordering**: lots of pruning happens (best moves are on the left of the game tree). This solves tree twice as deep as minimax in the same amount of time. Complexity $O(b^{m/2})$ (in practice). The search can go deeper in the game tree.
- **How to find a good ordering?**
  - Remember the best moves from shallowest nodes.
  - Order the nodes so as the best are checked first.
  - Use domain knowledge: e.g., for chess, try order: captures first, then threats, then forward moves, backward moves.
  - Bookkeep the states, they may repeat!

Real-time Decisions

- Minimax: generates the entire game search space
- $\alpha - \beta$ algorithm: prune large chunks of the trees
- BUT $\alpha - \beta$ still has to go all the way to the leaves
- Impractical in real-time (moves has to be done in a reasonable amount of time)
- Solution: bound the depth of search (cut o↵ search) and **replace** $utiliy(s)$ **with** $eval(s)$, an evaluation function to **estimate** value of current board configurations

Real-time Decisions

- $eval(s)$ is a heuristic at state s

  E.g., Othello: white pieces - black pieces

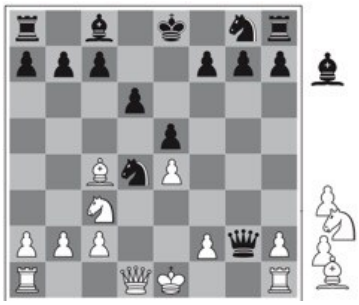
  E.g., Chess: Value of all white pieces Value of all black pieces turn non-terminal nodes into terminal leaves!
- An ideal evaluation function would rank terminal states in the same way as the true utility function; but must be fast
- Typical to define features, make the function a linear weighted sum of the features
- Use domain knowledge to craft the best and useful features.

Real-time Decisions

- How does it works?
  - Select useful features $f_1, \cdots, f_n$
    e.g., Chess: # pieces on board,
    value of pieces (1 for pawn, 3 for bishop, etc.)
  - Weighted linear function:

$$eval(s) = \sum_{i=1}^{n} w_i f_i(s)$$

  - Learn w_i from the examples
  - Deep blue uses about $6{,}000$ features!



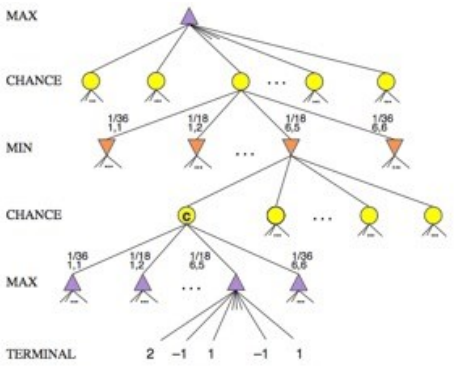
Stochastic Games

Stochastic Games

- Include a random element (e.g., throwing a dice).
- Include chance nodes.
- Backgammon: old board game combining skills and chance.
- The goal is that each player tries to move all of his pieces off the board before his opponent does.

Ptkfgs [Public domain], via Wikimedia Commons

Stochastic Games



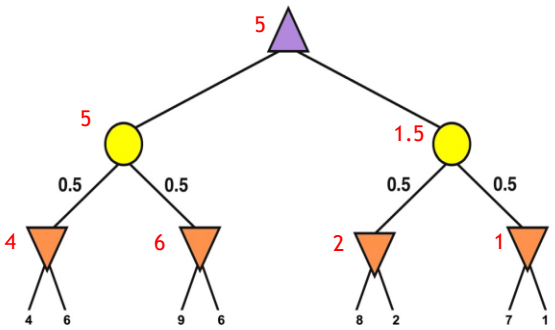
Partial game tree for Backgammon.

Stochastic Games

- Algorithm **Expectiminimax** generalized Minimax to handle chance nodes as follows:
  - If state is a Max node then
    return the highest Expectiminimax-Value of Successors(state)
  - If state is a Min node then
    return the lowest Expectiminimax-Value of Successors(state)
  - If state is a chance node then
    return average of Expectiminimax-Value of Successors(state)

Stochastic Games

- Example with coin-flipping:



Expectiminimax

- For a state s:

$$\text{Expectiminimax(s)} = \begin{cases} Utility(s) & \text{if Terminal-test(s)} \\ max_{a \in Actions(s)} \text{ Expectiminimax(Result(s,a))} & \text{if Player(s) = Max} \\ min_{a \in Actions(s)} \text{ Expectiminimax(Result(s,a))} & \text{if Player(s) = Min} \\ \sum_r P(r) \text{ Expectiminimax(Result(s,r))} & \text{if Player(s) = Chance} \end{cases}$$

- Where r represents all chance events (e.g., dice roll), and $Result(s,r)$ is the same state as s with the result of the chance event is r.

Summary

- Games are modeled in AI as a search problem and use heuristic to evaluate the game.
- Minimax algorithm choses the best move given an optimal play from the opponent.
- Minimax goes all the way down the tree which is not practical give game time constraints.
- Alpha-Beta pruning can reduce the game tree search which allow to go deeper in the tree within the time constraints.
- Pruning, bookkeeping, evaluation heuristics, node re-ordering and IDS are effective in practice.

Summary

- Games is an exciting and fun topic for AI.
- Devising adversarial search agents is challenging because of the huge state space.
- We have just scratched the surface of this topic.
- Further topics to explore include partially observable games (card games such as bridge, pocker, etc.).
- Except for robot football (a.k.a. soccer), there was no much interest from AI in physical games. (see http://www.robocup.org/).
- Interested in chess? check out the evaluation functions in Claude Shannon's paper.

References

- S. J. Russell and P. Borvig. (2020). *Artificial Intelligence: A Modern Approach (4th Edition)*, Prentice Hall International.
  - Chapter 5

eof