Artificial Intelligence Techniques (MSc)

Lecture 2: Adversarial Search and Game Playing

Delft University of Technology

Dr. Nils Bulling

September 20, 2016

Outline

1. Games and Search: Introduction
2. Optimal Decision Making
3. Non-Optimal Decision Making: Resource Constraints
4. Stochastic Games
5. Partially Observable Games
6. Summary

Lecturer: Dr. Nils Bulling

Teaching Assistant: Chris Rozemuller, MSc

Course email: ai@ii.tudelft.nl

Lecture:

- Wednesday, September 21st, 08:45-10:45

Tutorial and **Test**:

- Tutorial: Thursday, September 22nd
- Test: Thursday, September 29th

Please note the following points

1. The **exercises** and **assignments** are very **important** for the understanding and for passing the exam!
2. I would appreciate it to hear about any flaws, typos, comments and any other feedback which helps to improve the lecture.

Reading Material I

- The slides are quite detailed and should in general be sufficient.
- General reading :

Russel, S. and Norvig, P. (2010).
Artificial Intelligence: a Modern Approach.
Prentice Hall, 3 edition. Chapter 4.5, 5

Acknowledgement and Copyright

The lecture is based on and uses material of:

Russel, S. and Norvig, P. (2010).
Artificial Intelligence: a Modern Approach.
Prentice Hall, 3 edition. Chapter 4.4 and 5.

The author gratefully acknowledges material and slides provided by Carholijn Jonker developed in previous editions of this course at TU Delft.

Next Section

Topics for today: **algorithms for playing (board) games**

- formalization of games as search problems
- finding optimal decisions in games (e.g Tic-Tac-Toe)
- complexity: pruning and heuristic methods (e.g Chess)
- partially observable games (e.g Poker)
- stochastic games (e.g Backgammon)

Games

- setting:
  - a player's outcome depends on actions of others
  - often self-interested players
- unpredictability of other players and incomplete information
  $\rightsquigarrow$**contigencies** How to deal with them?
- Players use **strategies**. A strategy is a conditional plan that specifies how to reply to an opponent.
- Can we use classical search algorithms like breath-first and depth-first search to find optimal strategies?
- Extension of classical search algorithms: **adversarial search** (and-or search) to find winning strategies.
- Problem: Huge **states-spaces** often prevent exact solutions
  $\rightsquigarrow$ clever techniques and heuristics methods are needed

Brief Historical Overview

1846 (Babbage): discussion on computer's solving games

1912/44 (Zermelo/van Neumann/Morgenstern):
algorithm for perfect play (Minimax), foundation of game theory

1945-1950 (Zuse/Wiener/Shannon):
chess: finite horizon, approximate evaluation

1948 (Turing) chess program (paper machine)

1952/59 (Samuel) checkers: machine learning to improve heuristic

1955 (McCarthy) pruning of search tree to allow deeper search

1997 (IBM) IBM's Deep Blue wins over chess world champion Garry Kasparov

Properties of Games

Example 1.1 (Chess)

Which "game properties" has chess?

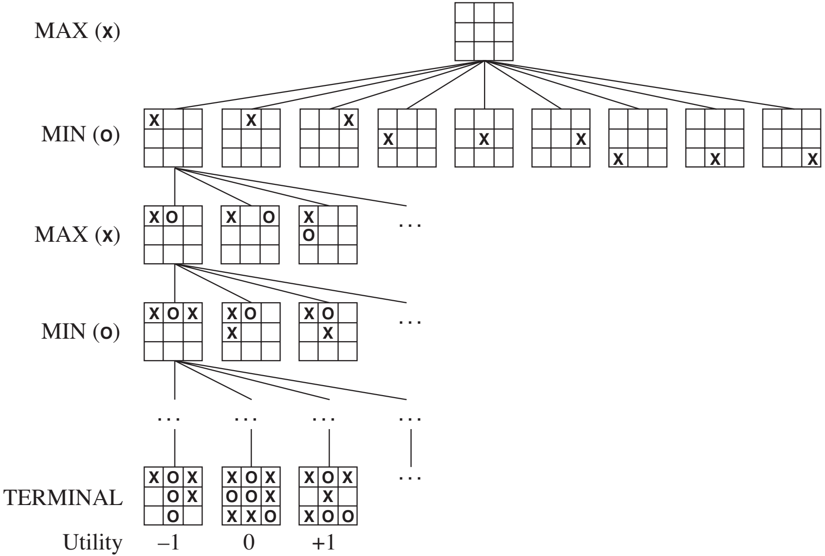
- How many players?
- Which information do players have about the state of the game?
- Is the outcome of an action completely deterministic?

We can classify games along the following dimensions:

Types of Games	deterministic	stochastic
perfect information	chess, checkers, go, othello	backgammon, monopoly
imperfect information	battleship, blind tictactoe, Kriegspiel	bridge, poker, scrabble, nuclear war

In the following, we often consider 2-player (Max and Min), turn-based, deterministic, strictly competetive, perfect information

The (finite) rules of a game induce a **game tree**, consisting of all possible plays (is it always finite?):



We give a semi-formal definition of an adversarial search problem.

Definition 1.2 (Formulation as Search Problem)

An **adversarial search problem** consists of the following elements:

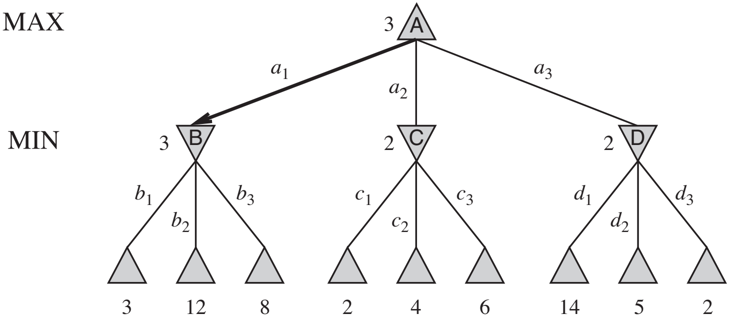
- s_0: initial (game) state of the game
- Player(s): defines the player whose turn it is at state s
- Action(s): returns set of legal actions at state s
- Result(s, a): (game) state which results if action a is being executed in s
- TerminalTest(s): returns true if the game is over; false otherwise.
- Utility(s, p): returns the utility of player p at terminal node s.

The **game tree** of the search problem is the tree induced by functions Action and Result in the initial state s_0. We often use S to denote the set of all game states.

Note the differences between: **adversarial search problem**, **game tree** and a **search tree** over the adversarial search problem!

Notation of ply

The execution of each action in a game is also called a **ply**.
For example, the following game tree consists of **two plies**, or **2-plies**.



In chess a **move** consists of 2-plies.

Next Section

Computing Optimal Decisions

How to find an **optimal strategy**?

How to **search** through the game tree? Can we use standard search algorithms like depth-first search or breath-first search?

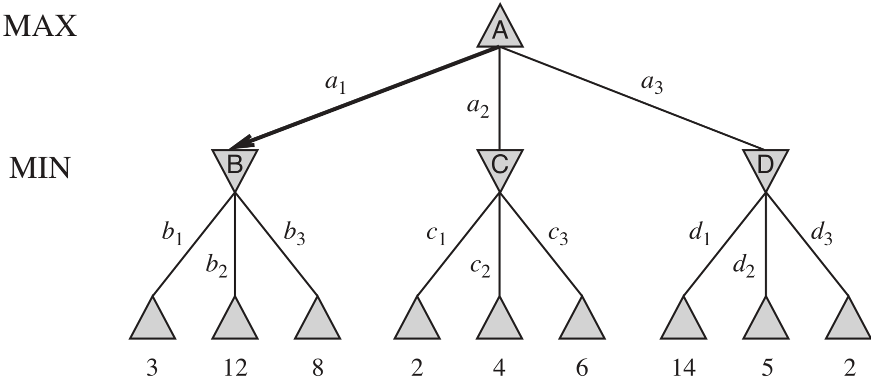
We discuss the following:

- optimal decision making: find the optimal strategy of a player$\rightsquigarrow$**Minimax algorithm**
- improve performance of Minimax algorithm $\rightsquigarrow$**alpha-beta Minimax**

Subsection I

2 Optimal Decision Making

Consider the following 2-player game tree. Players are denoted by Max and Min. Terminal nodes are the leaf nodes; they are labelled with Max's utility value. What is the **optimal** strategy of Max?



⇝ FeedbackFruits Idea: We assume that the opponent plays optimally and compute the best possible action under this assumption.

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

$Minimax(s)$ recursively computes the utility of the optimal action at state s. It is called the **minimax value** at s.

Exercise 2.1 (Non-optimal opponents)

Minimax assumes that the opponent plays optimally. Suppose the opponent does not play optimally. Show the following.

1. *For non-optimally playing opponents the Minimax value is at least as good as if played against an optimal opponent.*
2. *For non-optimally playing opponents, the Minimax value may not be optimal (i.e. give a concrete game tree as counterexample).*

The **Minimax algorithm** performs a depth-first search through the game tree. We assume that Max is the starting player.

function MINIMAX-DECISION(*state*) **returns** *an action*
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(*state*, *a*))

---

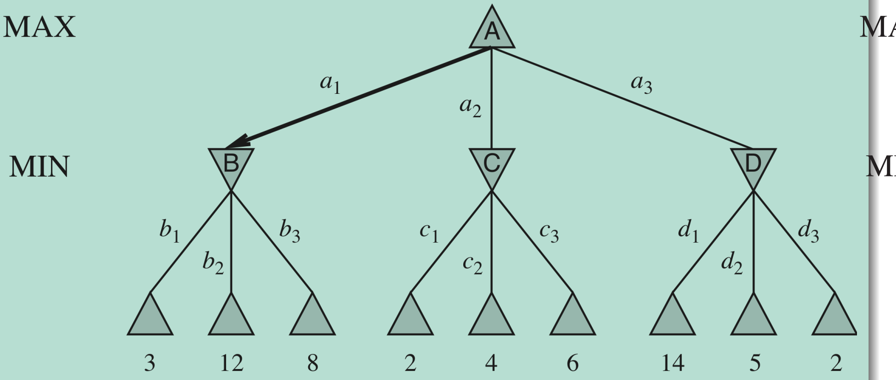
function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

---

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return v

Example 2.1 (Minimax)

Apply the Minimax algorithm to the following game tree. Compute the minimax value and the optimal action:



Properties of Minimax

Definition 2.2 (Branching factor)

The **branching factor** at a node is the number of legal moves at that node. The **maximal branching factor** is the maximal branching factor at some node. The **average branching factor** is the average branching factor across all nodes.

We will often use branching factor to refer to the average branching factor.

Example 2.3

What is the average branching factor of the following games?

- TicTacToe $\rightsquigarrow$ 4
- Backgammon $\rightsquigarrow$ $\sim$400, maximal one can be much higher 400
- Chess $\rightsquigarrow$35
- Go $\rightsquigarrow$250

Recall: A **search strategy** prescribes how a state space/game tree is searched.

Definition 2.4 (Completeness, optimality)

A search strategy is called

- **complete**, if it finds a solution, provided there exists one at all.
- **optimal**, if whenever it produces an output, this output is an optimal solution, i.e. one with the best utility.

It is assumed that you know the following classical search strategies and their properties:

- depth-first search
- breath-first search
- iterative deepening search
- etc.

We analyse the time complexity of the Minimax algorithm. How many notes does the Minimax algorithm examine?

Suppose the game tree has the (finite) maximal branching factor b and a **depth** of at most m **levels** (starting from 0), i.e. an m-ply game tree.

Minimax performs a full depth-first search: $\leadsto$ FeedbackFruits

Time complexity : $b + b^2 + \ldots + b^m = \frac{1-b^{m+1}}{1-b} - 1 = O(b^m)$

Space complexity : $O(bm)$

Can be improved to $O(m)$ if only one successor is kept in memory, which is then currently updated.

Complete : yes (if m is finite)

Optimal : yes, against an optimal opponent

Exact values Do Not Matter

The exact value of the terminal nodes of a game tree do not matter as long as it is ensured, that the relative ordering is kept
$\rightsquigarrow$**ordinal utility function**.

Let t be a transformation function on terminal nodes to real numbers such that for all terminal nodes n_1 and n_2 with values $x, y \in \mathbb{R}$, respectively, in the original game tree it holds that:

- $x \leq y$ iff $t(n_1) \leq t(n_2)$, and
- $x < y$ iff $t(n_1) < t(n_2)$.

Then, the actions computed by the Minimax algorithm on the original game tree and to the game tree obtained by updating the utility values according to the transformation function t are identical.

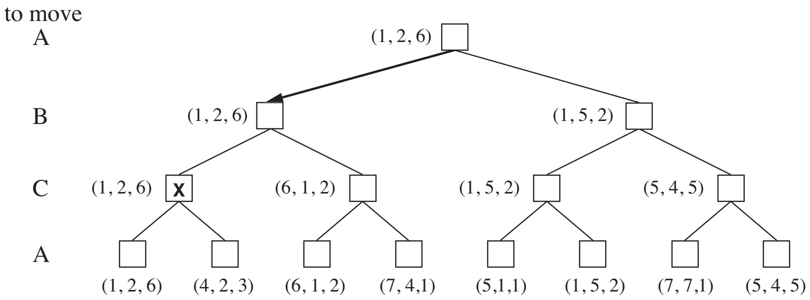
Exercise 2.2 (Exact values do not matter)

State the above more formally and give a formal proof.

Subsection I

2 Optimal Decision Making

We have considered a two player setting. This can be generalized to arbitrary players and non-zero sum games:



Exercise 2.3 (Minimax in multi-player turn-based games)

Extend the Minimax algorithm to the multi-player setting and compute the optimal strategy of player A in the game tree shown above.

Subsection I

2 Optimal Decision Making

The Mimimax algorithm computes an **optimal stratgey**, but it has to search the whole tree. This is often not possible.

Example 2.5 (Chess)

How many nodes to search in Chess up to the following depth:

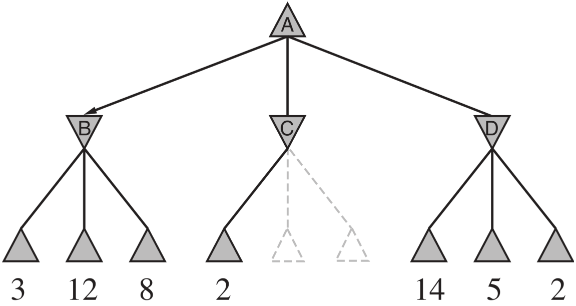
- 1-ply: $b = 35$
- 2-ply (1 move): $b^2 = 1225$
- 3-ply: $b^3 = 42875$
- 4-ply (2 moves): $b^4 = 1.5$ millions

The number of nodes to search grows too fast—exponential in the depth. Can we **prune** the search tree and at the same time **retain optimality**?

Yes, using the following idea:

- Prune subtrees which cannot yield optimal solutions.
- In order to decide which subtrees to prune, store the value of the best "choices" (α resp. β) of the players (Max resp. Min) encountered so far.

The soundness of the pruning follows from the observation that the minimax value is independent of the minimax value of pruned subtrees:



$$
\begin{aligned}
Minimax(s_0) &= \max(\min(3,12,8), \min(\mathbf{2}, \mathbf{x}, \mathbf{y}), \min(14,5,2)) \\
&= \max(3, \min(\mathbf{2}, \mathbf{x}, \mathbf{y}), 2)) \\
&= \max(3, z, 2) \quad \text{for } z = \min(2, x, y) \leq 2 \\
&= 3
\end{aligned}
$$

The Alpha-Beta Minimax algorithm implements the idea of pruning, by updating two values for each node:

α-value: value of best action for *Max* found so far at any choice point along the path (i.e. highest value)

β-value: value of best choice for *Min* found so far at any choice point along the path (i.e. lowest value)

In the search tree, the values are represented as interval $[\alpha, \beta]$.

Now, pruning works as follows:

1. **Update the values** along a path.
2. If the computed value of the current node would give a worse solution for the other player then prune remaining subtrees. (As the other player would not choose this node.)

Suppose it is Min's turn and it can ensure a value v with $v \leq \alpha$. Then, Max has a different action at least as good $\rightsquigarrow$ prune the tree.

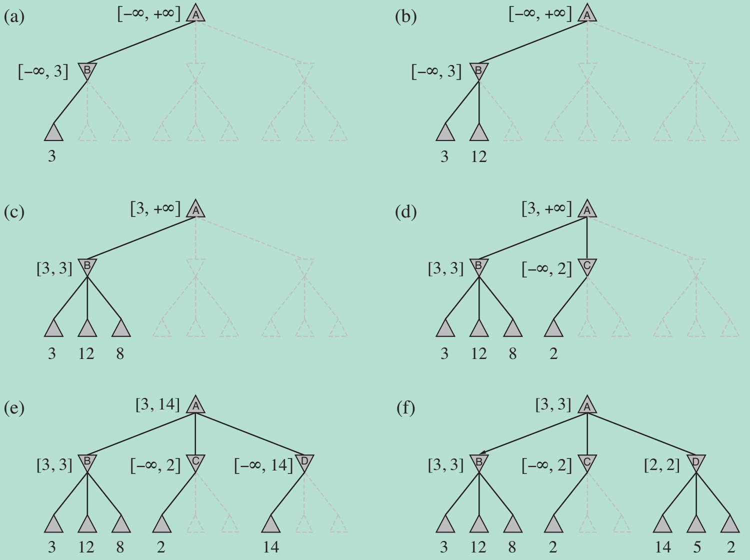
function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

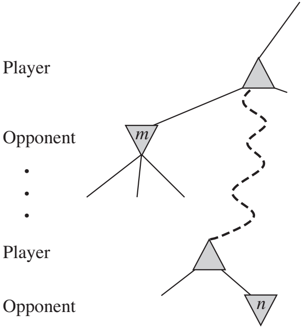
function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS($state$) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) ,α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Example 2.7 (Alpha-Beta Pruning)



Proposition 2.1 (Correctness of Alpha-Beta Minimax)

Let $P \in \{\text{Max}, \text{Min}\}$ *and* O *be the opponent. Let* n *be a node in the game tree and suppose that* P *has an action to reach* n *in the next step. If* P *has a better action* m *along the path to* n *(i.e. either at the parent node of* n *or at any choice point further up in the path (preventing from reaching* n*)), then node* n *will never be reached if* P *plays a strategy induced by the* Minimax *algorithm.*



Proof.

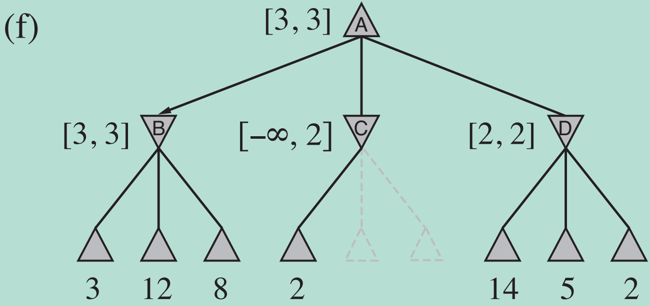
⇝ Exercise ☐

Performance

The improve in the **performace** is sensitive to the action ordering (i.e. the order in which the game tree is traversed).

Example 2.8 (Order important for efficiency)

In Example 2.7, (e) and (f) no terminal node can be pruned. What if the 3rd successor of D had been the first child of D?



Remark 2.1 (Alpha-Beta Minimax)

- *"optimal action ordering": time complexity* $\approx O(b^{m/2}) = O(\sqrt{b}^m) \rightsquigarrow$ *Exercise*
- *Thus, Alpha-Beta Minimax can examine a game tree roughly twice as deep as Minimax.*
- *"random generation of action": time complexity* $\approx O(b^{3m/4})$
- *Conclusion: a good ordering is very important in order to search large game trees.*

Exercise 2.4 (Alpha-Beta Minimax with optimal move ordering)

Show that in the case of an "optimal action ordering" the Alpha-Beta Minimax algorithm has a worst-case time complexity of $O(b^{m/2})$ *where* m *is the maximal depth of the game tree.*

Next Section

So, far we considered optimal decisions. Often, there are **resource limits** that do not allow to reach optimal decisions.

Example 3.1 (Search under time constraints)

Suppose we play a chess and decisions must be made within 100 seconds. Moreover, the computer can explore 10^4 nodes per second. (Recall, the average branching factor of chess is $b \approx 35$.)

- How many plies can *MiniMax* (optimally) explore? About 4: $O(b^m) \leq 10^6 \rightsquigarrow 35^{3.9} \approx 10^6 \rightsquigarrow m \approx 4$ (for the exact value, solve: $\frac{1-b^{m+1}}{1-b} \leq 10^6$)
- ... and alpha-beta search (with a good action ordering)? About 8
- This is considered the level of a good chess player.

What if **time limit** but **no terminal node** is reached? How to make a decision? Possibly a non-optimal one?

Heuristic Minimax Value

Instead of a fixed cutoff after a predefined number of steps, we consider a quality-based approach.

- A **cutoff test** replaces the terminal test in the Minimax algorithm. It decides when to stop the search, possibly using an evaluation function.
- An **evaluation function** determines how good the current state is. It returns a **heuristic utility value**.

The **heuristic Minimax value** value is defined as follows:

$$H{-}Minimax(s) =$$
$$\begin{cases} Eval(s) & \text{if } CutoffTest(s,d) \\ \max_{a \in Actions}(H{-}Minimax(Result(s,a))) & \text{if } Player(s) = Max \\ \min_{a \in Actions}(H{-}Minimax(Result(s,a))) & \text{if } Player(s) = Min \end{cases}$$

The $CutoffTest(s)$ may also take into consideration the current depth d.

Subsection I

3 Non-Optimal Decision Making: Resource Constraints

Evaluation Functions

Minimax with Cutoffs

Deterministic Games in Practice

An **evaluation function** estimates the worth of the current state. This is also how human chess players evaluate the current state of the game. How to define such a function ?

Basic requirements on the evaluation function:

1. *Eval*(s) should give the same value as *Utility*(s) for terminal states s (e.g. don't turn win into loose).
2. Evaluation of the function must not be too complex: time limits!
3. At non-terminal states, the evaluation value should be strongly correlated with the chance of winning.

Note that uncertainty is introduced by **computational limitations** not by the game itself.

How to define **good evaluation functions**? This is a difficult task. For example, player experience, static features, sub-solutions.

Basic construction of an evaluation function:

1. Define **features** $f_1, \dots, f_n$ of a state which induce a vector-based evaluation function $f(s) = (f_1(s), \dots, f_n(s))$.
2. Values of features induce equivalence classes, called **categories**, of states. Same values of features corresponds to same category, e.g.
$$C_1 = \{s \mid f_1(s) = c_1^1, \dots, f_n(s) = c_n^1\}, \quad \text{here } c_j^1 \text{ are fixed}$$
3. States in a category are usually of different quality. Define the evaluation function for a category as the **expected value** of the states in the category, e.g.
$$\mathbf{E}(C_1) = 0.7 \cdot \mathbf{1} + 0.1 \cdot \mathbf{0} + 0.2 \cdot (-\mathbf{1})$$
(where feature value $1, 0, -1$ corresponds to win, draw, loose, respectively).
4. Define the evaluation of a state as the evaluation of its category:
$$\text{Eval}(s) = \mathbf{E}(C) \qquad \text{if } s \in C$$

Remark 3.1 (Evaluation function)

- *Often, Minimax allows to consider 4-plies lookahead if classes of categories/features is not too big, and if the features can easily be computed.*
- *The construction requires a lot of experience and many categories; often too many to be useful.*

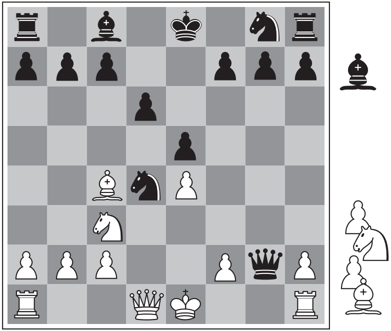
In practise: Compute **numerical values from each feature** and **combine them** into a single value (independent of the notation of category).

Example 3.2 (Evaluation of a Chess state)

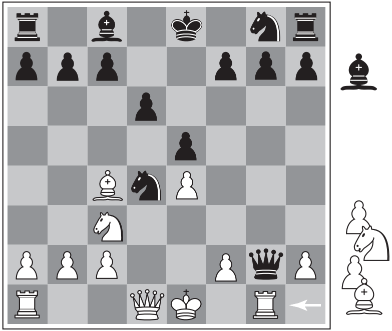
In chess, e.g., a queen could be assigned a worth of 10 and a rook 6.

Another feature could be defined as:

$$\mathbf{f_1(s)} = (\text{number of white queens}) - (\text{number of black queens})$$



(a) White to move



(b) White to move

- One could use a **linear weighted sum** of features (w_i are weights):
$$Eval(s) = w_1 f_1(s) + \ldots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s)$$

This assumes that features are **independent** of each other. Therefore, **non-linear** functions can be used as well.

Subsection I

Is it rather straight-forward how to incorporate evaluation functions into Minimax and Alpha-Beta Minimax. The *TerminalTest* in the Alpha-Beta Minimax algorithm is simply replaced by:

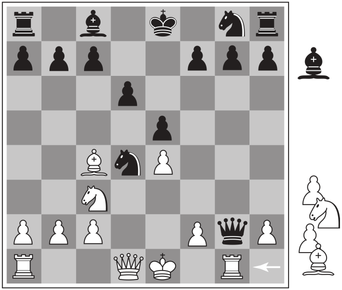
$$\boxed{\text{if } \mathit{CutoffTest}(s, d) \text{ then return } \mathit{Eval}(s)}$$

where d represents the current depth.

A depth limit is used to ensure that the answer is given within a given time limit. How to determine a good depth limit?

- Simple approach: introduce a fixed depth limit d.
- More robust: combine with iterative deeping depth-first search $\rightsquigarrow$ when time is up, return value of deepest completed search

This simple implementation can lead to errors. Crucial changes in the player situation need to be taken into account. Let us reconsider the following situation:



White to move

- Seems to be a winning situation for Black.
- But: White can capture the black queen, indicating a winning situation for White.
- It is said that the state of the game is **not quiescent**.

Some notes:

- A situation is **quiescent** if an action cannot yield fundamental changes in the state evaluation.
  $\rightsquigarrow$ **Cutoffs should only be made in quiescent states.**
- More improvements are possible/necessary: e.g. horizon effect/singular extensions

Subsection I

Overview: Deterministic Games in Practice

Checkers: Jonathan Schaeffer's Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used a precomputed endgame database defining **perfect play for all positions involving 8 or fewer pieces** on the board, a total of 444 billion positions. 2007: **solved**:

`http://www.newscientist.com/article/dn12296-checkers-solved-after-years-of-number-c`
`html#.VHzc8_mG98E`

Chess: Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997,

http://www.youtube.com/watch?v=3EQA679DFRg.

Deep Blue searches 200 million positions per second and 12-plies, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 plies.

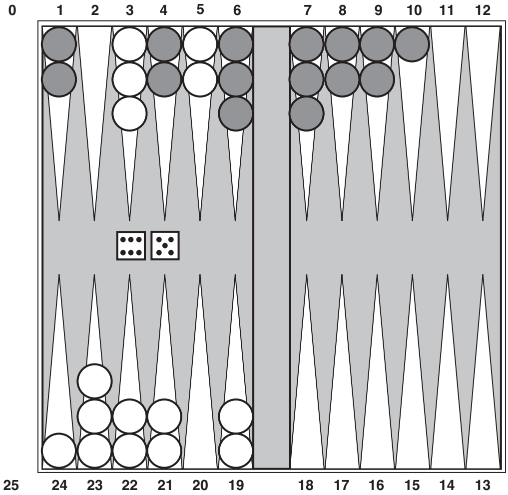
Null move heuristic: shallow search where opponent can move twice at the beginning (gives a lower bound)

Go: Difficult for Computers. Problem: **branching factor** > 300, so most programs use pattern knowledge bases to suggest plausible actions.

Recent progress: In 2016 AlphaGo (Google) won against a professional Go master (9 dan)

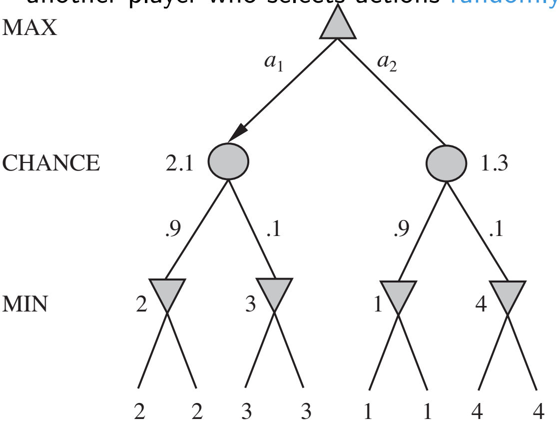
Next Section

So far, we considered **deterministic** and fully observable games.
What if a game includes **dice throws** or **random moves**?



Subsection I

We extend game trees with **chance moves**. This can be seen as another player who selects actions randomly.

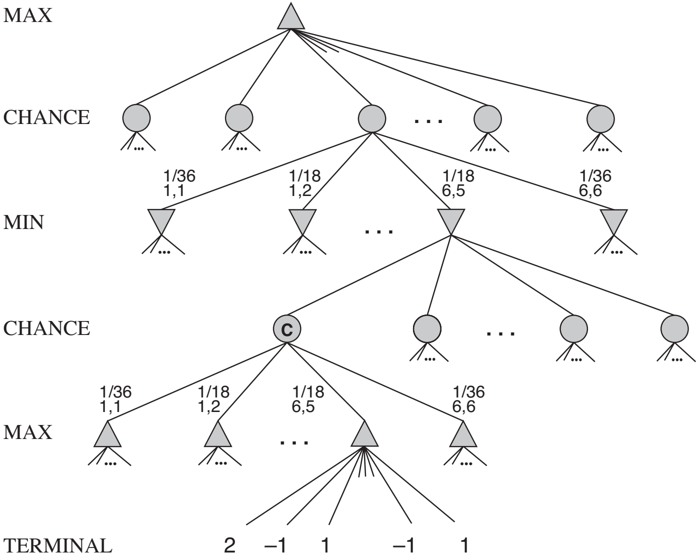


Sources of randomness:

- dice throws
- card shuffling
- coin flipping
- drawing pieces
- etc.

Attention: We merge the actions of the player and the associated chance move into a **single ply** (here Min's action). This affects the branching factor: in the above example, it is $b = 4$ (rather than $b = 2$) for Min's ply.

Backgammon has the following excerpt of a game tree. The situation is that Max moved its pieces, thereafter, Min throws the dices:



Subsection I

4 Stochastic Games

- Game Trees with Chance Moves
- Minimax Algorithm for Stochastic Games

Expeceted Minimax Value

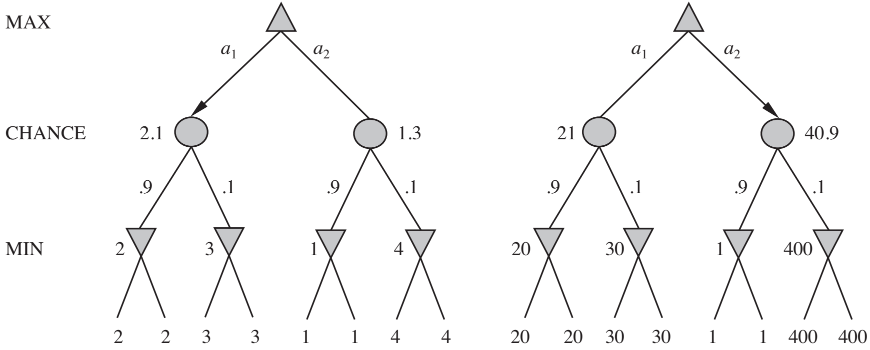
$$
Expectiminimax(s) = \begin{cases} Utility(s) & \text{if } TerminalTest(s) \\ \max_{\mathbf{a} \in Actions}(Expectiminimax(Result(s,a))) & \text{if } Player(s) = Max \\ \min_{\mathbf{a} \in Actions}(Expectiminimax(Result(s,a))) & \text{if } Player(s) = Min \\ \sum_{r \in \sigma}(\Pr(\mathbf{r}) Expectiminimax(Result(s,r))) & \text{if } Player(s) = \\ & \quad = Chance \end{cases}
$$

where

- σ represents a **stochastic action** (like a dice role) and $r \in \sigma$ a possible outcome of the action
- $Result(s, r)$ is the same state as s, together with the information that the outcome of the stochastic action is r.

The **ExpectiMiniMax algorithm** is defined analogously.

As before, **evaluation functions** can be used to define a cut-off version of the Expectiminimax algorithm. But note, that with chance moves the evaluation is **sensitive to the actual values**:



Here, the optimal action is different in both game trees.
Behavior preserved if positive linear transformations are considered.

Remark 4.1 (Practical Aspects)

- *Chance nodes increase significantly the branching factor b*
  - **time complexity**: $O(b^m n^m)$ *where n is number of* **distinct dice rolls** $\rightsquigarrow$ *FeedbackFruits*
  - **Backgammon**: *21 is the number of distinct dice rolls (with two dice);* ≈ 20 *moves, average branching factor about 420 (but much more with doubles, e.g. 1-1 means: move* 4 *pieces each 1 positions* $\rightsquigarrow$ *branching factor of 4000 and more)*
  - *4-plies*: $(21 \cdot 20)^4 \approx 3.1 \cdot 10^{10}$ *nodes*
  - *limited value in look-ahead for increased depth*
  - *alpha-beta pruning less effective (***Why?***)*
- *TD-Gammon Gammon (G. Tesauro, 1992, IBM)*
  - *artificial neural net trained by a form of temporal-difference learning*
  - *Depth 2 search + very good evaluation function: world-champion level*

Exercise 4.1 (Alpha Beta Pruning for Stochastic games)

Give an intuitive argument why alpha-beta search may not be very effective in stochastic games.

Next Section

In games like chess and backgammon, players have **perfect information** about the world. There are many games in which players have **incomplete information**; they cannot see the complete state of the game. Examples are:

- poker
- bridge
- scrabble
- etc.

How can we model these games appropriately?

We consider **belief states** rather than actual states of the game. Such sates encodes all situation an agent cannot distinguish from the current situation.

A **belief state** b consists of all physical states the player considers possible, together with a probability.

Definition 5.1 (Belief state)

Let S be a set of game states of a game. A **belief states b over S** is a probability distribution over S.

We also represent a belief state b as a subset $X = \{s_1, \ldots, s_n\} \subseteq S$ where b is defined as follows (clearly not all belief states can be represented in such a way):

$$b(s) = \begin{cases} \frac{1}{n} & \text{if } s \in X \\ 0 & \text{otherwise.} \end{cases}$$

In that case we also identify b with X. How is the **(expected) utility of a belief state** defined?

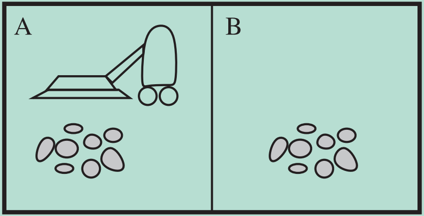
$$\text{Utility}(b) = \sum_{s \in S} b(s) \cdot \text{Utility}(s)$$

Example 5.2 (Vacuum world)

The **vacuum world** is defined as follows:

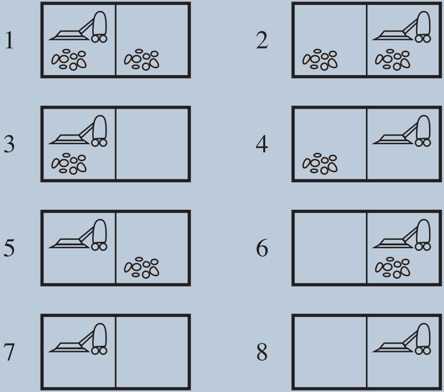
- It consists of two rooms.
- Each room may be dirty.
- A vacuum agent can move from one room to the other, or suck the dirt. Actions are *Left*, *Right*, *Suck*.
- The vacuum agent does not know its location, nor can it see whether a room is dirty or not.

The **goal** is to ensure that both rooms are clean. A game state:



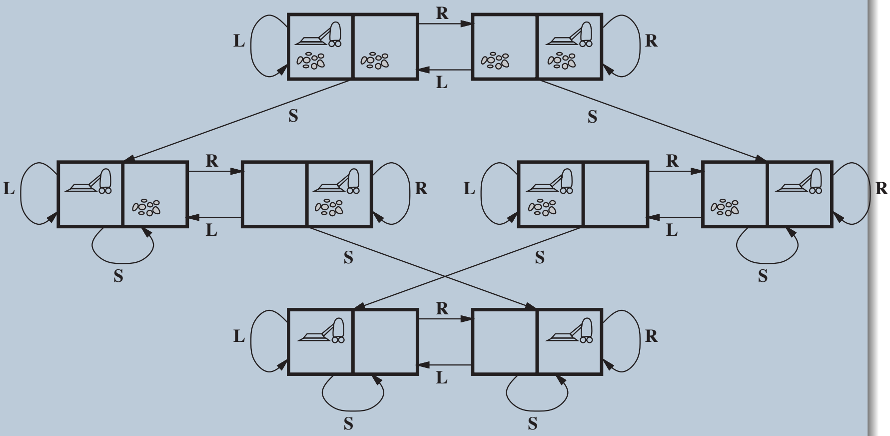
Example continued

The game states of the vacuum world are:



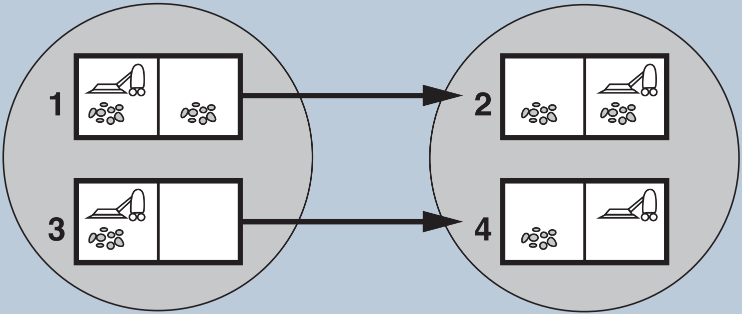
Example continued

The transitions in the physical world look at follows:



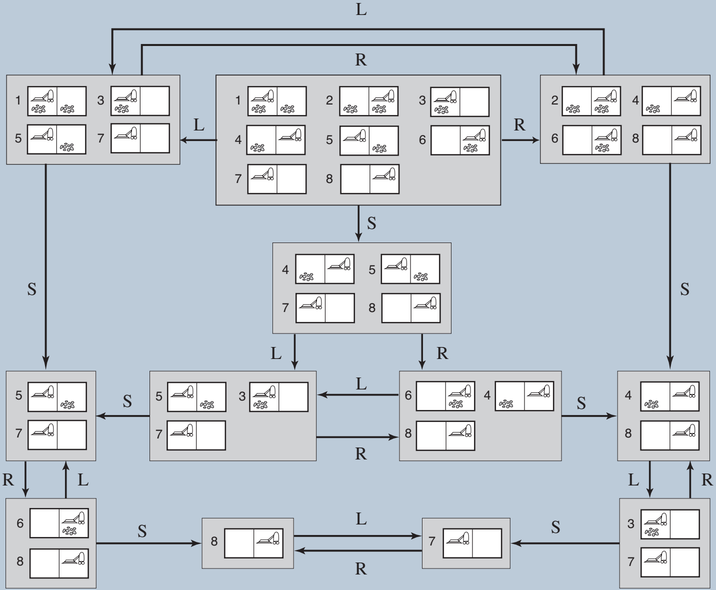
Example continued

What is a belief state in the world? Each of the two game states in a belief state can be associate probability 0.5. A state transition in the belief space (which action is being performed?):



The following picture shows the reachable part of the belief space.

Example continued



- In partially observable games, **strategies** are defined over belief states, or rather **sequences of belief states**/ **sequences of percepts**.
- A **winning strategy** must yield a belief state in which all physical states are winning.
- These games can also be combined with non-deterministic actions, chance moves, etc.
- In general the reachable belief states are **exponetial** wrt. the set of game states. Finding optimal solutions becomes even more challenging.

Example 5.3 (Vacuum world)

Is there a winning strategy (**sure winning**) in the vacuum world, i.e. one in which both rooms are clean and the robot knows this? $\rightsquigarrow$ FeedbackFruits

Action sequence: Left, Suck, Right, Suck

Example 5.4 (Kriegspiel)

Kriegspiel is a partially observable variant of chess.

- Players only see their own pieces.
- They announce their actions and a referee announces whether each move is legal or illegal.
- The referee also informs about captures, checks, checkmates or stalemates, among other things.

So, the player **learns** from the announcements of the referee.

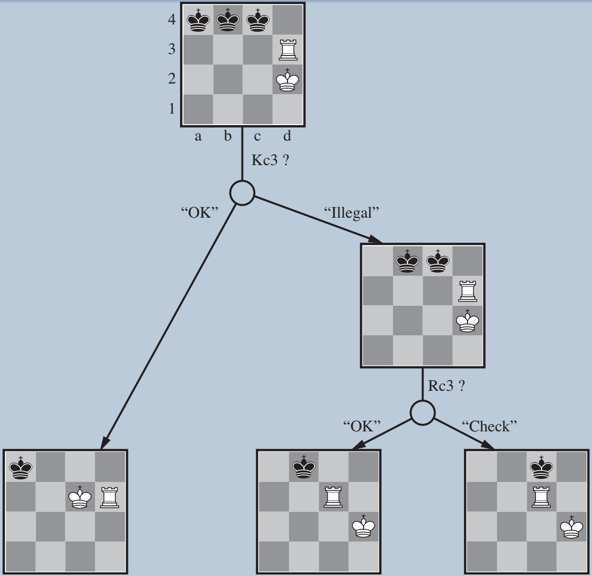
A winning strategy corresponds to a strategy which ensures a **guaranteed checkmate**, one which leads to a checkmate for every possible board state in the current belief state.

Example continued

black only has a king left

figure encode **belief states**

strategy to find out about **true position of king**



Partially observable games offers different notions of winning strategies. For example, in Kriegspiel:

- **guaranteed checkmate**: guaranteed winning
- **probabilistic checkmate**: winning with probability 1. E.g. finding the lonely king on the board. (In general this is not always possible.)
- **accidental checkmate**: player wins but could not know this from its belief state.
- It may not always be optimal to perform the "best action" as it may provide too much information $\rightsquigarrow$ concept of **equilibrium**

As another example of partially observable games, we discuss **card games**. Cards are dealt randomly at the beginning of the game, and are of course invisible to other players. How could we compute the best action? Now how can we solve such games?

What about the following idea: does it work?

1. Consider all possible deals $s_1, \ldots, s_r$ of the invisible cards.
2. Compute the optimal action a_i wrt. to each s_i assuming that s_i describes the actual distribution of the cards. Use (a variant of) Minimax.
3. Take the action that maximizes the expected utility wrt. the games induced by s_i: (note that a is the same action for all s_i!)

$$\mathrm{argmax}_a \sum_{i=1}^{\mathbf{r}} \mathsf{Pr}(\mathbf{s_i})\mathsf{Minimax}(\mathsf{Result}(\mathbf{s_i}, \mathsf{a}))$$

What can be problematic with this approach?

Note, that we assume that the game is **fully observable** for each fixed guessed deal s. Thus, we average over the result of the game if we knew the correct state. Therefore, it is also called **averaging over clairvoyance**.

The following example illustrates why **averaging over clairvoyance does not work** for (many) games with incomplete information.

Example 5.5 (Problem with "averaging over clairvoyance")

Day 1:
- **Road A** leads to a heap of gold; **Road B** leads to a fork
- left fork: bigger heap of gold; right fork: you will be run over by a bus.

Day 2:
- **Road A** leads to a heap of gold; **Road B** leads to a fork
- right fork: bigger heap of gold; left fork: you will be run over by a bus.

Day 3:
- **Road A** leads to a heap of gold; **Road B** leads to a fork
- one fork: bigger heap of gold; other fork: you will be run over by a bus.
- You cannot distinguish the two forks.

What happens if averaging over clairvoyance is applied? $\rightsquigarrow$ FeedbackFruits

Example continued

- In averaging over clairvoyance, *B* **is taken** because:
  - *B* is the best choice at Day 1.
  - *B* is the best choice at Day 2.
  - *B* is the best choice as Day 3 as it is either Day 1 or Day 2.

  The possible "deals" are instantiated, and Road *B* is always the best move, followed by different actions at the fork.

- In day three the **belief state** of the agent should have been considered and not an (instantiated) game state. Note that the flawed approach assigns different actions in states which the agent cannot distinguish.
- The averaging over clairvoyance approach always yields states of perfect knowledge which is not always appropriate.
- The right way to solve partially observable games is to consider **belief states** $\rightsquigarrow$ later in the course

Next Section

Summary and Conclusions

- Solving games is challenging
- and important, e.g. it plays a role in verification tasks (multi-agent systems)
- Solving games is often (theoretically) possible, but in practice, limited resources prevent optimal solutions.
- Many “simple” games have a huge branching factor which makes them hard to solve (e.g. Go); so far...
- Heuristic approaches are needed.
- Partially observable and stochastic games are even harder to solve.
- Some games with incomplete information are impossible to solve algorithmically: they are undecidable.

References I

Russel, S. and Norvig, P. (2010).
Artificial Intelligence: a Modern Approach.
Prentice Hall, 3 edition.

References I

Russel, S. and Norvig, P. (2010).
Artificial Intelligence: a Modern Approach.
Prentice Hall, 3 edition.