Chapters 3–5: More search

DIT410/TIN173 Artificial Intelligence

Peter Ljunglöf

(inspired by slides by Poole & Mackworth, Russell & Norvig, et al)

12 April, 2016

Outline

1. *More search strategies (Russel & Norvig 3.4)*
   - Iterative deepening
   - Bidirectional search
2. *Local search (Russell & Norvig 4.1, Poole & Mackworth 4.8)*
   - Iterative best improvement
   - Randomized algorithms
   - Population-based methods
   - Evaluating randomized algorithms (Poole & Mackworth 4.8.3)
3. *Games (Russell & Norvig 5.1–5.5)*
   - Minimax search
   - Alpha-beta pruning
   - Nondeterministic games

Outline

Outline

Iterative deepening

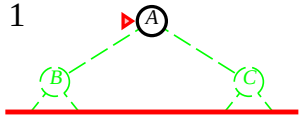
- BFS is guaranteed to halt but uses exponential space.
- DFS uses linear space, but is not guaranteed to halt.
- Idea: take the best from BFS and DFS –
  recompute elements of the frontier rather than saving them.
- Look for paths of depth 0, then 1, then 2, then 3, etc.
- Depth-bounded depth-first search can do this in linear space.
- If a path cannot be found at *depth bound*, look for a path at *depth bound* + 1. Increase *depth bound* when the search fails unnaturally (*depth bound* was reached).

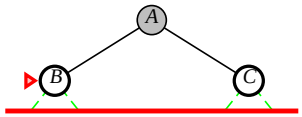
Iterative deepening search, bound = 0

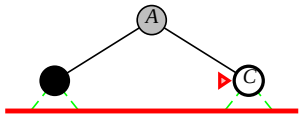
Limit = 0

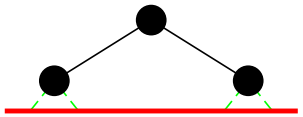
Iterative deepening search, bound = 1

Limit = 1

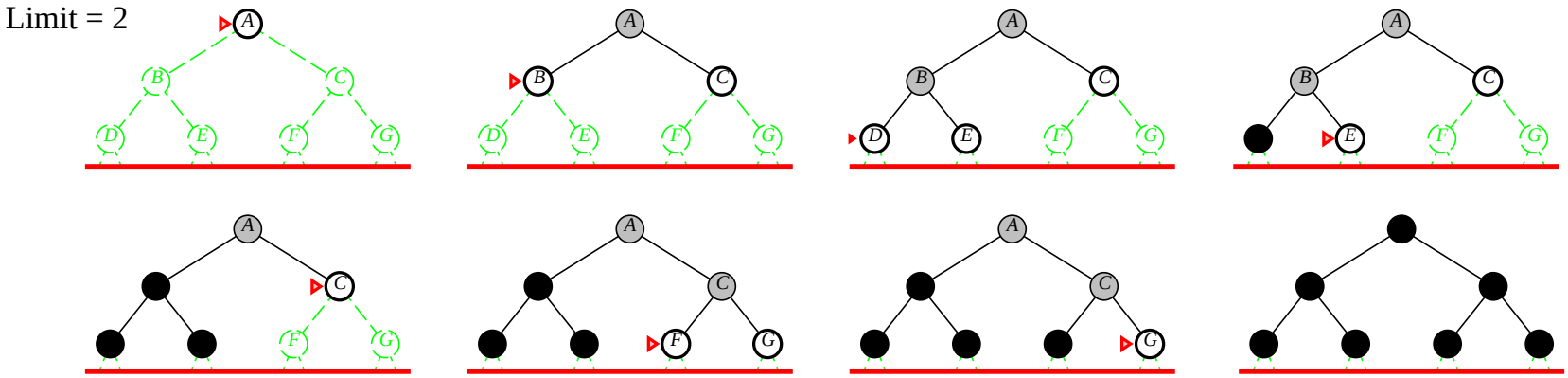




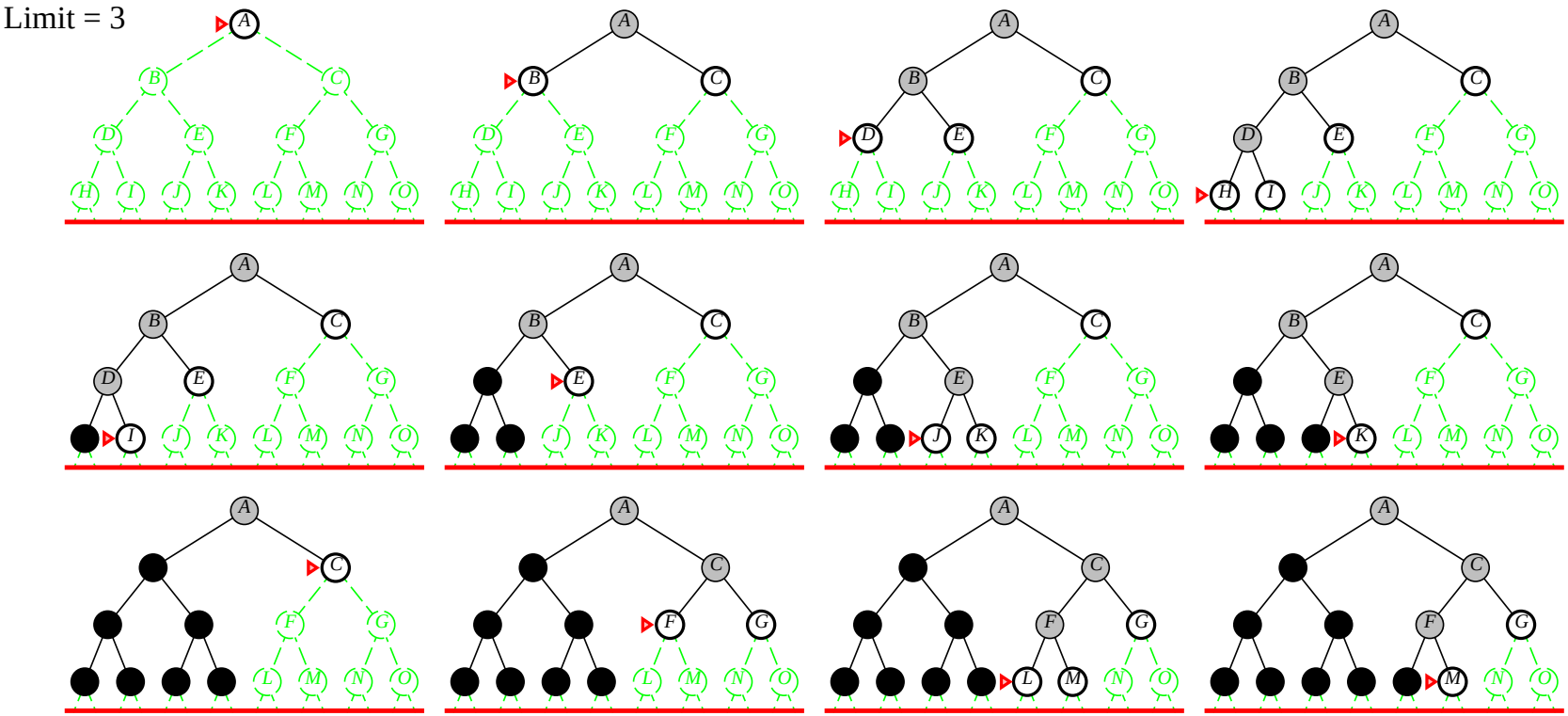




Iterative deepening search, bound = 2



Iterative deepening search, bound = 3



Iterative-deepening search

procedure IDSearch(*graph*, *initial-state*, *goal-state*):
    for *limit* **in** 0, 1, 2, . . . :
        result := DepthLimitedSearch($\langle$*initial-state*$\rangle$, *limit*)
        if *result* $\neq$ cutoff **then return** *result*

procedure DepthLimitedSearch($\langle n_0, \ldots, n_k \rangle$, *limit*):
    if n_k is a *goal-state* **then return** path $\langle n_0, \ldots, n_k \rangle$
    else if *limit* = 0 **then return** cutoff
    else:
        failure-type := failure
        for every neighbor n of n_k:
            result := DepthLimitedSearch($\langle n_0, \ldots, n_k, n \rangle$, *limit*-1)
            if *result* is a path **then return** *result*
            else if *result* = cutoff **then** *failure-type* := cutoff
        return *failure-type*

Iterative deepening complexity

Complexity with solution at depth k and branching factor b:

level	breadth-first	iterative deepening	# nodes
1	1	k	b
2	1	$k-1$	b^2
...	...	...	...
$k-1$	1	2	b^{k-1}
k	1	1	b^k
total	$\geq b^k$	$\leq b^k \left(\frac{b}{b-1}\right)^2$	

Numerical comparison for $k = 5$ and $b = 10$:

$$\#(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110$$
$$\#(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

Note: IDS recalculates shallow nodes several times, but this doesn't have a big effect compared to BFS!

Outline

Direction of search

- The definition of searching is symmetric: find path from start nodes to goal node or from goal node to start nodes.
- Forward branching factor: number of arcs out of a node.
- Backward branching factor: number of arcs into a node.
- Search complexity is b^n. Should use forward search if forward branching factor is less than backward branching factor, and vice versa.
- Note: when a graph is dynamically constructed, the backwards graph may not be available.

Bidirectional search

- Idea: search backward from the goal and forward from the start simultaneously.
- This wins as $2b^{k/2} \ll b^k$. This can result in an exponential saving in time and space.
- The main problem is making sure the frontiers meet.
- This is often used with one breadth-first method that builds a set of locations that can lead to the goal. In the other direction another method can be used to find a path to these interesting locations.

Island driven search

- Idea: find a set of islands between s and g.

$$s \longrightarrow i_1 \longrightarrow i_2 \longrightarrow \ldots \longrightarrow i_{m-1} \longrightarrow g$$

  This gives us m smaller problems rather than 1 big problem.
- This can win as $mb^{k/m} \ll b^k$.
- The problem is to identify the islands that the path must pass through. It is difficult to guarantee optimality.

Outline

1. *More search strategies (Russel & Norvig 3.4)*
   - Iterative deepening
   - Bidirectional search
2. *Local search (Russell & Norvig 4.1, Poole & Mackworth 4.8)*
   - Iterative best improvement
   - Randomized algorithms
   - Population-based methods
   - Evaluating randomized algorithms (Poole & Mackworth 4.8.3)
3. *Games (Russell & Norvig 5.1–5.5)*
   - Minimax search
   - Alpha-beta pruning
   - Nondeterministic games

Outline

Iterative improvement

In many optimization problems, the *path* is irrelevant

- the goal state itself is the solution

Then the state space can be the set of "complete" configurations

- e.g., for 8-queens, a configuration can be any board with 8 queens (it is irrelevant in which order the queens are added)

In such cases, we can use *iterative improvement* algorithms;
we keep a single "current" state, and try to improve it

- e.g., for 8-queens, we start with 8 queens on the board, and gradually move some queen to a better place

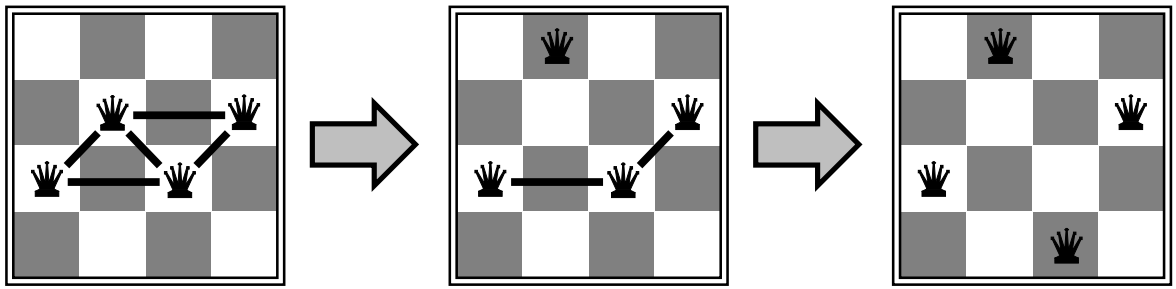
The goal would be to find an optimal configuration

- e.g., for 8-queens, where no queen is threatened

This takes constant space, and is suitable for online and offline search

Example: n-queens

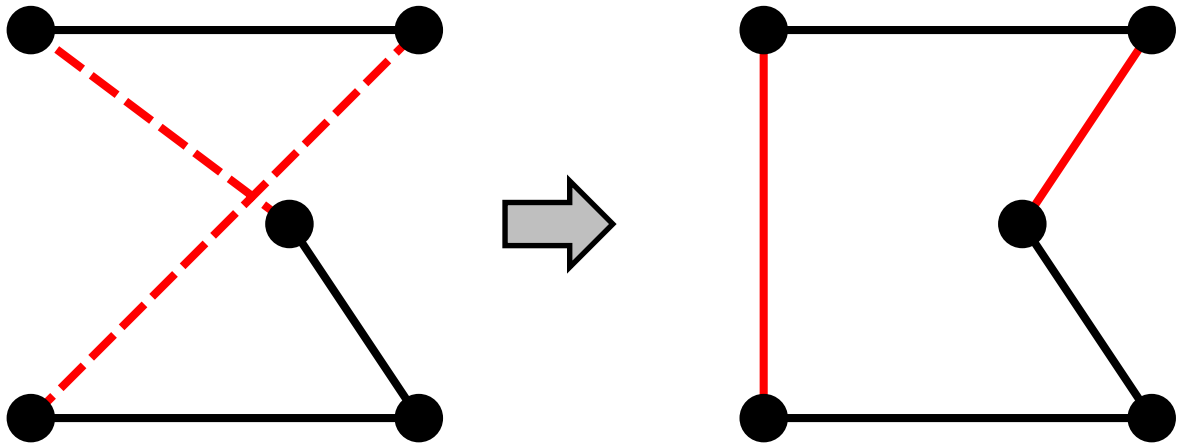
- Put n queens on an $n \times n$ board, in separate columns
- Move a queen to reduce the number of conflicts;
  repeat until we cannot move any queen anymore
  - then we are at a local maximum, hopefully it is global too



This almost always solves n-queens problems almost instantaneously for very large n (e.g., $n = 1$ *million*)

Example: Travelling salesperson

Start with any complete tour, and perform pairwise exchanges

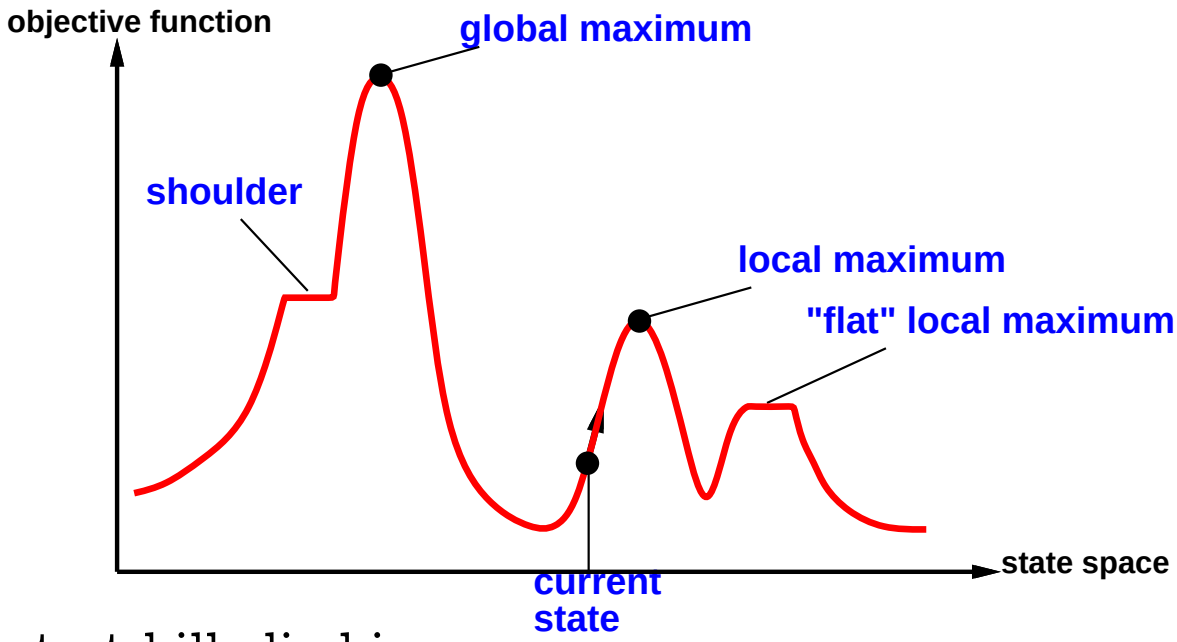


Variants of this approach get within 1% of optimal very quickly with thousands of cities

Hill climbing (or gradient ascent/descent)

procedure HillClimbing(*graph*, *initial-state*):
    current := *initial-state*
    loop:
        neighbor := a highest-valued successor of *current*
        if *neighbor* value $\leq$ *current* value **then return** *current*
        child := *neighbor*

The state space landscape



Random-restart hill climbing:

- overcomes local maxima
- always finds the global maximum (given enough time)

Random sideways moves:

- escapes from shoulder, but loops on flat maxima

Outline

Randomized algorithms

- Consider two methods to find a minimum value:
  - Greedy ascent, starting from some position,
    keep moving upwards,
    and report maximum value found
  - Pick values at random, and report maximum value found
- Which do you expect to work better to find a global maximum?
- Can a mix work better?

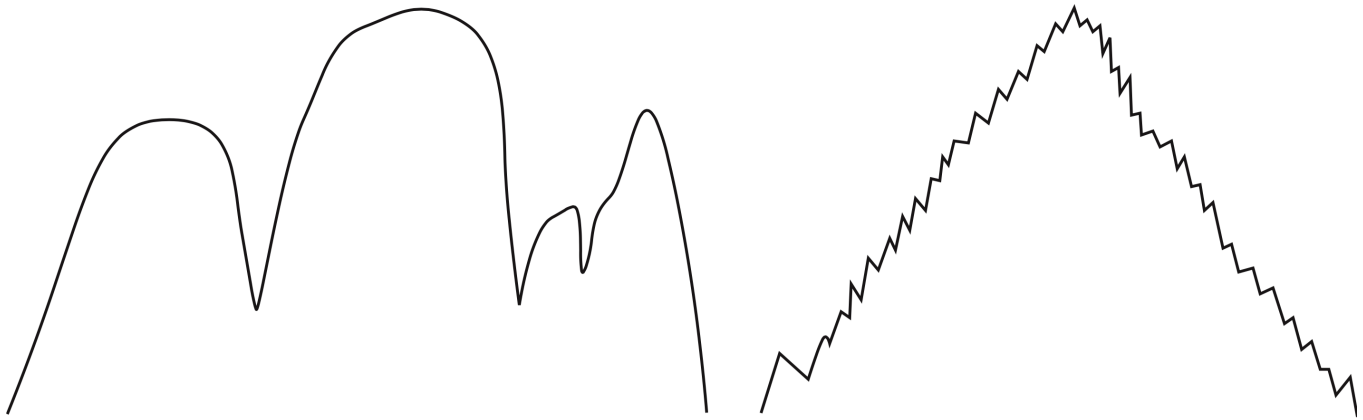
Randomized hill climbing

As well as upward steps we can allow for:

- Random steps: move to a random neighbor.
- Random restart: reassign random values to all variables.

1-dimensional illustrative example

Two 1-dimensional search spaces; step right or left:



- Which method would most easily find the global maximum?
  - random steps or random restarts?
- What happens in hundreds or thousands of dimensions?
  - e.g., where different dimensions have different structure?

Stochastic local search

Stochastic local search is a mix of:

- Greedy ascent: move to a highest neighbor
- Random walk: taking some random steps
- Random restart: reassigning values to all variables

One example of this is simulated annealing:

- the probability for taking a random step is gradually decreased while the search algorithm is running

Outline

Beam search

Idea: maintain a population of k states in parallel,
instead of one:

- At every stage, choose the k best out of all of the neighbors.
- When $k = 1$, it is greedy descent.
- When $k = \infty$, it is breadth-first search.
- The value of k lets us limit space and parallelism.
- *Note*: this is not the same as k searches run in parallel!
- *Problem*: quite often, all k states end up on the same local hill.

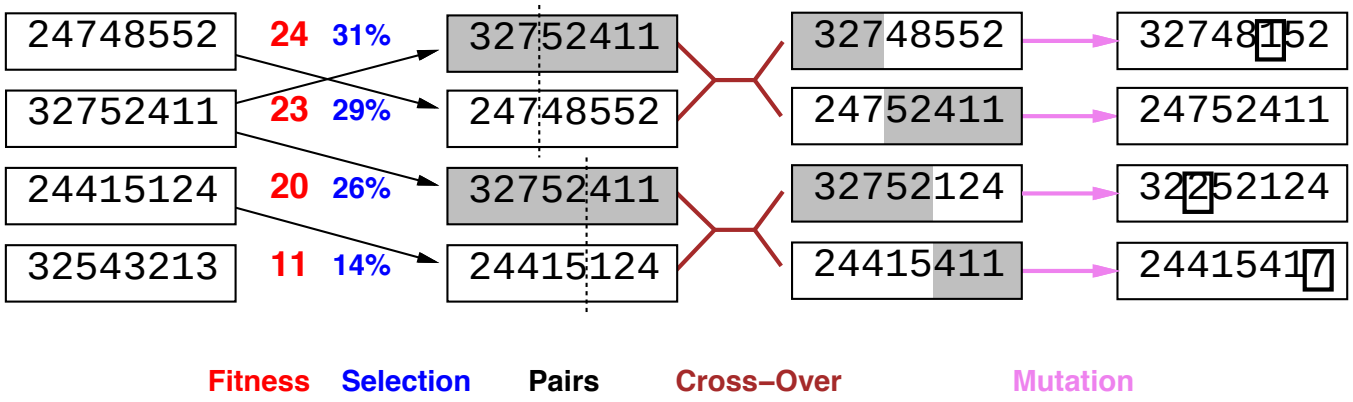
Stochastic beam search

Like beam search, but it probabilistically chooses the k individuals at the next generation:

- The probability that a neighbor is chosen is proportional to its heuristic value.
- This maintains diversity amongst the individuals.
- The heuristic value reflects the fitness of the individual.
- Like asexual reproduction: each individual mutates and the fittest ones survive.

Genetic algorithms

Like stochastic beam search, but pairs of individuals are combined to create the offspring.



- For each generation:
  - Randomly choose pairs of individuals where the fittest individuals are more likely to be chosen.
  - For each pair, perform a cross-over: form two offspring each taking different parts of their parents:
  - Mutate some values.
- Stop when a solution is found.

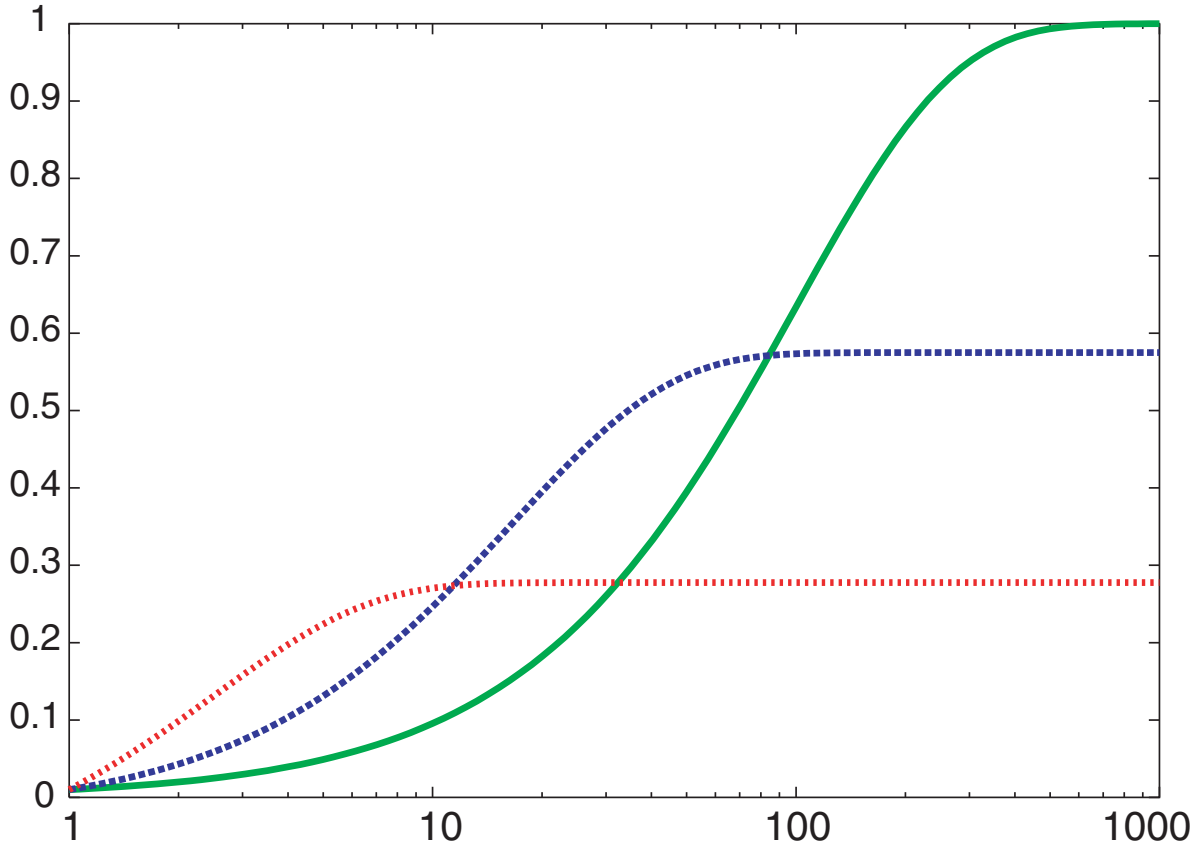
Outline

Comparing stochastic algorithms

- How can you compare three algorithms **A**, **B** and **C**, when
  - **A** solves the problem 30% of the time very quickly but doesn't halt for the other 70% of the cases
  - **B** solves 60% of the cases reasonably quickly but doesn't solve the rest
  - **C** solves the problem in 100% of the cases, but slowly?
- Summary statistics, such as mean run time, median run time, and mode run time don't make much sense.

Runtime distribution

Plots runtime (or number of steps) and the proportion (or number) of the runs that are solved within that runtime.



Outline

Multiple agents

Let's consider multiple agents, where:

- the agents select actions autonomously
- each agent has its own information state
  - they can have different information (even conflicting)
- the outcome depends on the actions of all agents
- each agent has its own utility function
  (that depends on the total outcome)

Types of agents

There are two extremes of multiagent systems:

Cooperative: The agents share the same utility function
Example: Automatic trucks in a warehouse

Competetive: When one agent wins all other agents lose
A common special case is when $\sum_a u_a(o) = 0$ for any outcome o. This is called a *zero-sum game*.
Example: Most board games

Many multiagent systems are between these two extremes.

Example: Long-distance bike races are usually both *cooperative* (the bikers usually form clusters where they take turns in leading the group), and *competetive* (only one of them can win in the end).

Games as search problems

The main difference to chapters 3–4:
now we have more than one agent that have different goals.

- All possible game sequences are represented in a game tree.
- The nodes are states of the game, e.g. board positions in chess.
- Initial state and terminal nodes.
- States are connected if there is a legal move/ply.
- Utility function (payoff function).
- Terminal nodes have utility values −1, 0 or +1.
  - (some authors use 0, $^1/_2$, and 1, confusingly)

Types of games

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

Outline

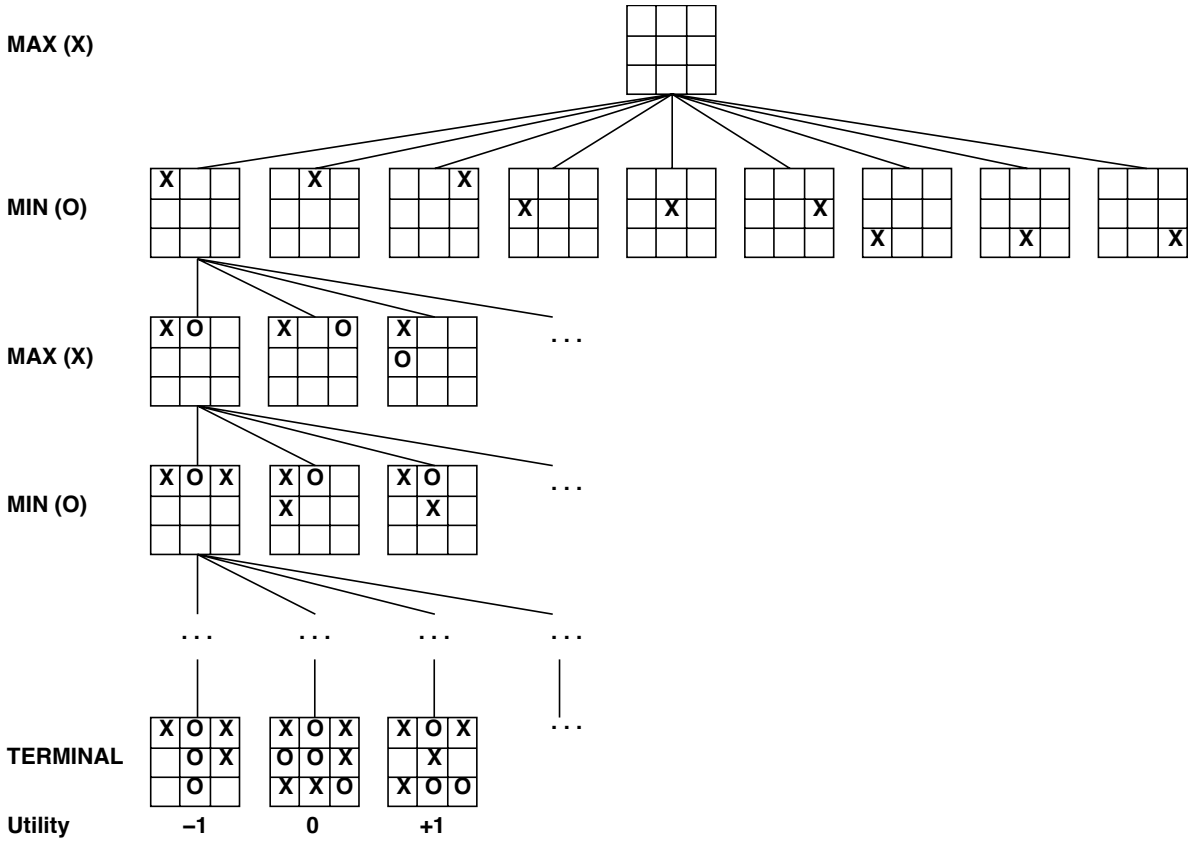
Strategies for 2-player games

Given two players called MAX and MIN, MAX wants to maximize the utility value. Since MIN wants to minimize the same value, MAX should choose the alternative that maximizes given that MIN minimized.

The Minimax algorithm gives perfect play for deterministic, perfect-information games:

procedure Minimax(*state*):
    if TerminalTest(*state*):
        return Utility(*state*)
    $A :=$ Actions(*state*)
    if *state* is a MAX node:
        return $\max_{a \in A}$ Minimax(*state*, a)
    else if *state* is a MIN node:
        return $\min_{a \in A}$ Minimax(*state*, a)

Minimax search: tic-tac-toe



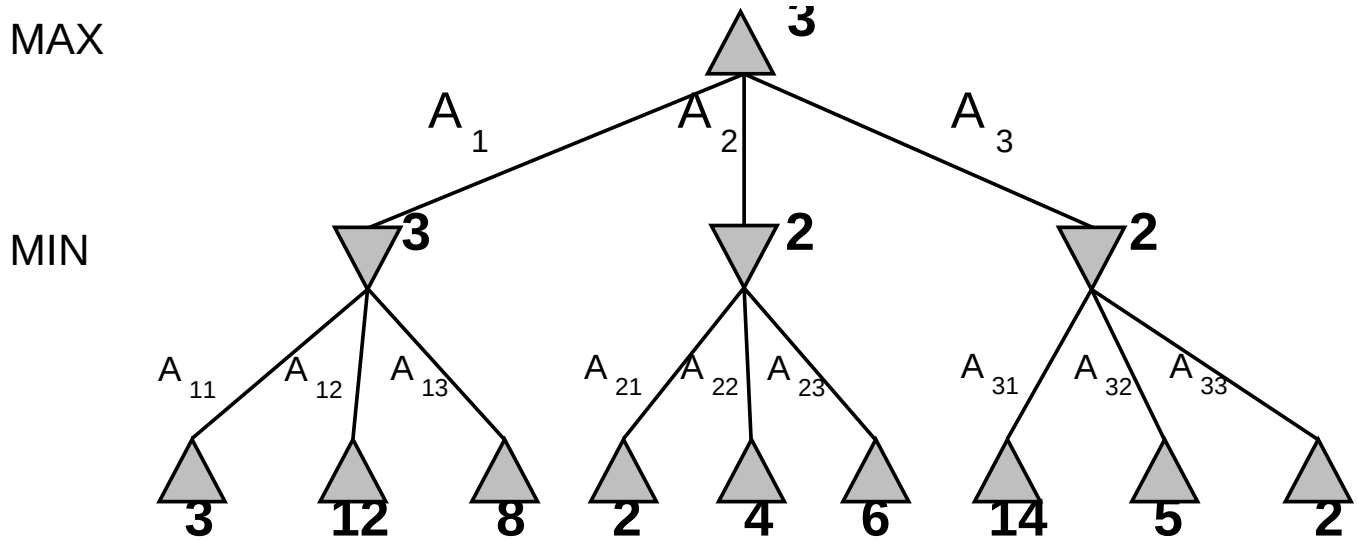
Outline

$\alpha - \beta$ pruning

- Suppose, we reach a node t in the game tree which has leaves $t_1, \ldots, t_k$ corresponding to moves of player MIN.
- Let α be the best value of a position on a path from the root node to t.
- Then, if any of the leaves evaluates to $u(t_i) \leq \alpha$, we can discard t, because any further evaluation will not improve the value of t.
- Analogously, define β values for evaluating MAX moves.

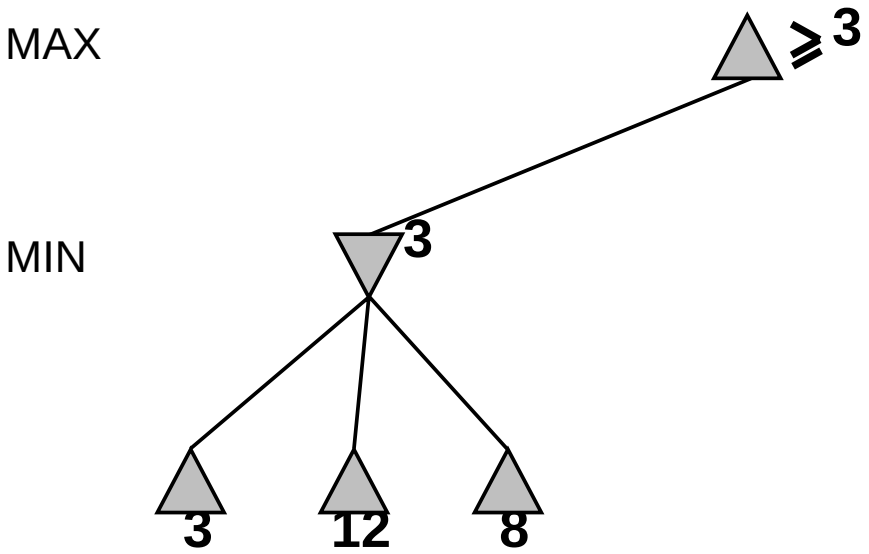
Minimax example

The Minimax algorithm gives perfect play for deterministic, perfect-information games.



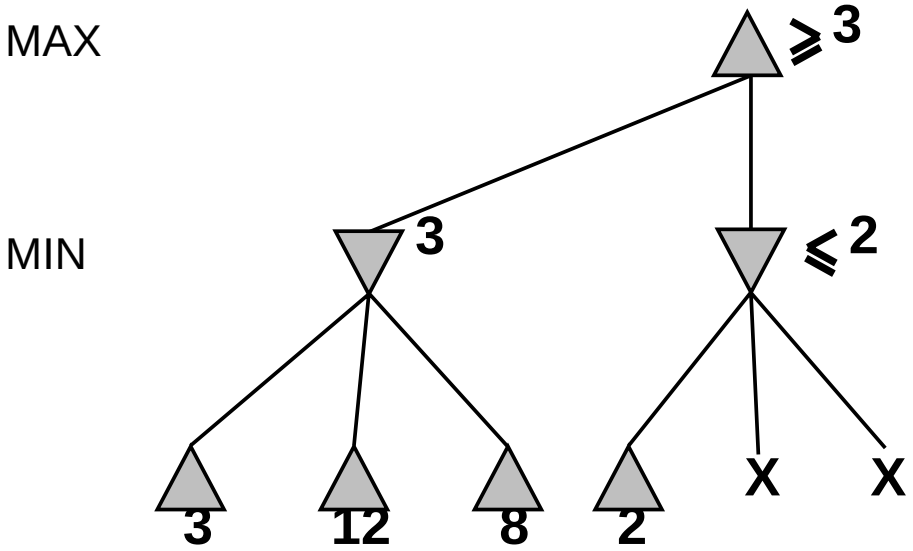
Minimax example, with $\alpha - \beta$ pruning

This is how $\alpha - \beta$ pruning might work:



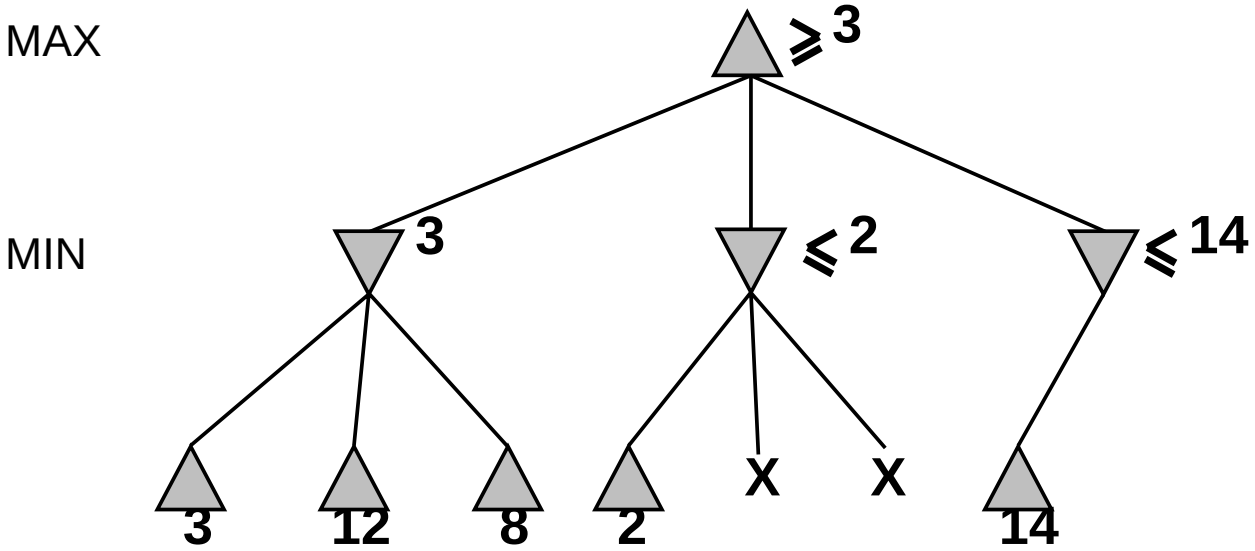
Minimax example, with $\alpha - \beta$ pruning

This is how $\alpha - \beta$ pruning might work:



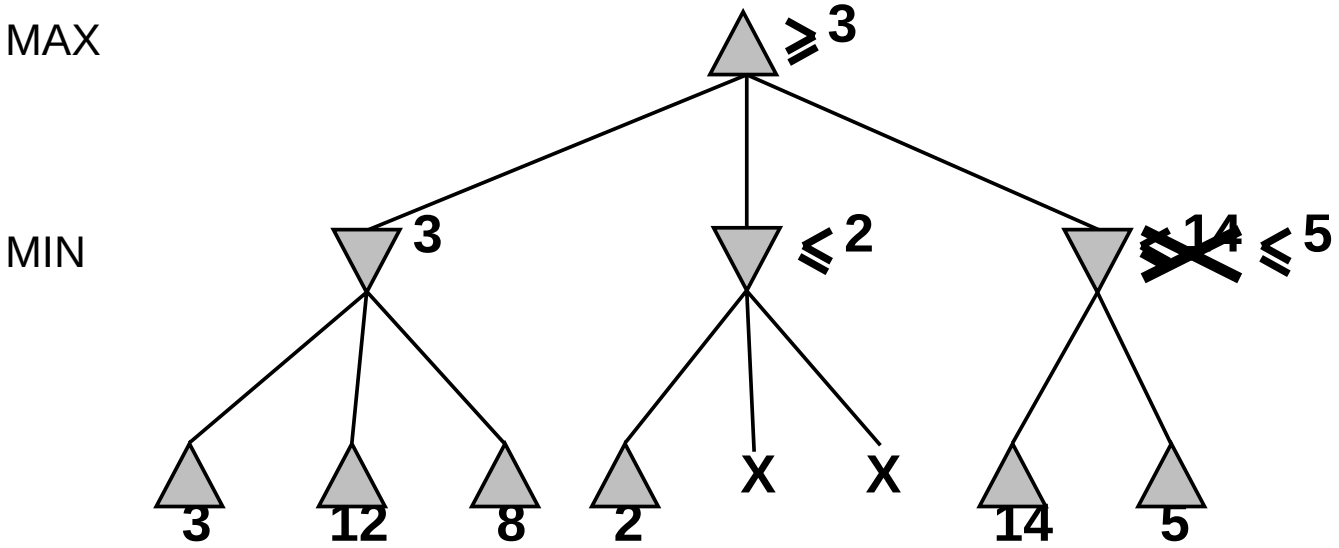
Minimax example, with $\alpha - \beta$ pruning

This is how $\alpha - \beta$ pruning might work:



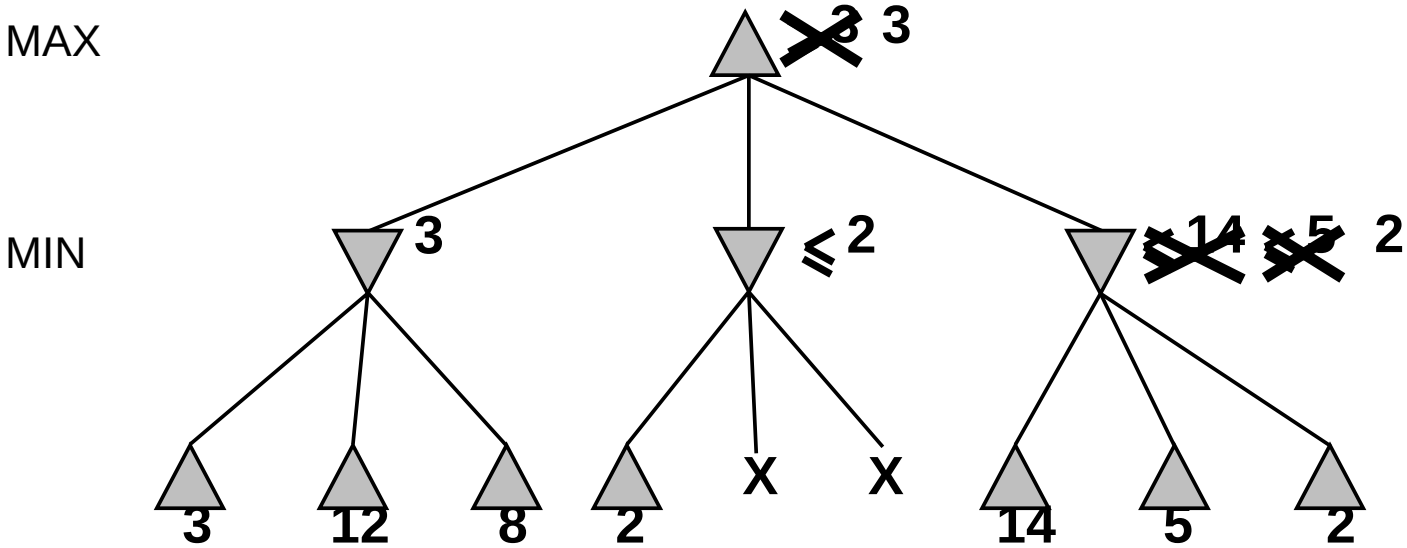
Minimax example, with $\alpha - \beta$ pruning

This is how $\alpha - \beta$ pruning might work:



Minimax example, with $\alpha - \beta$ pruning

This is how $\alpha - \beta$ pruning might work:



How efficient is $\alpha - \beta$ pruning?

The amount of pruning provided by the $\alpha - \beta$ algorithm depends on the ordering of the children of each node.

- It works best if a highest-valued child of a MAX node is selected first and if a lowest-valued child of a MIN node is returned first.
- In implementations of real games, much of the effort is made to try to ensure this outcome.
- With a "perfect ordering", the time complexity becomes $O(b^{m/2})$
  - this *doubles* the solvable search depth
  - however, 35^{40} (for chess) or 250^{80} is still impossible...

Minimax and real games

Most real games are too big to carry out minimax search, even with $\alpha - \beta$ pruning.

- For these games, instead of stopping at leaf nodes, we have to use a cutoff test to decide when to stop.
- The value returned at the node where the algorithm stops is an estimate of the value for this node.
- The function used to estimate the value is an evaluation function.
- Much work goes into finding good evaluation functions.
- There is a trade-off between the amount of computation required to compute the evaluation function and the size of the search space that can be explored in any given time.

Deterministic games in practice

Chess:

- DeepBlue (IBM) beats world champion Garry Kasparov, 1997.
- Modern chess programs: Houdini, Critter, Stockfish.

Checkers/Othello/Reversi:

- Human champions refuse to compete – computers are too good.
- Logistello beats the world champion in Othello/Reversi, 1997.
- Chinook plays checkers perfectly, 2007. It uses an endgame database defining perfect play for all 8-piece positions.

Go:

- Human champions refuse to compete – computers are too bad.
- Modern programs: MoGo, Zen, GNU Go.

Deterministic games in practice

Chess:

- DeepBlue (IBM) beats world champion Garry Kasparov, 1997.
- Modern chess programs: Houdini, Critter, Stockfish.

Checkers/Othello/Reversi:

- Human champions refuse to compete – computers are too good.
- Logistello beats the world champion in Othello/Reversi, 1997.
- Chinook plays checkers perfectly, 2007. It uses an endgame database defining perfect play for all 8-piece positions.

Go:

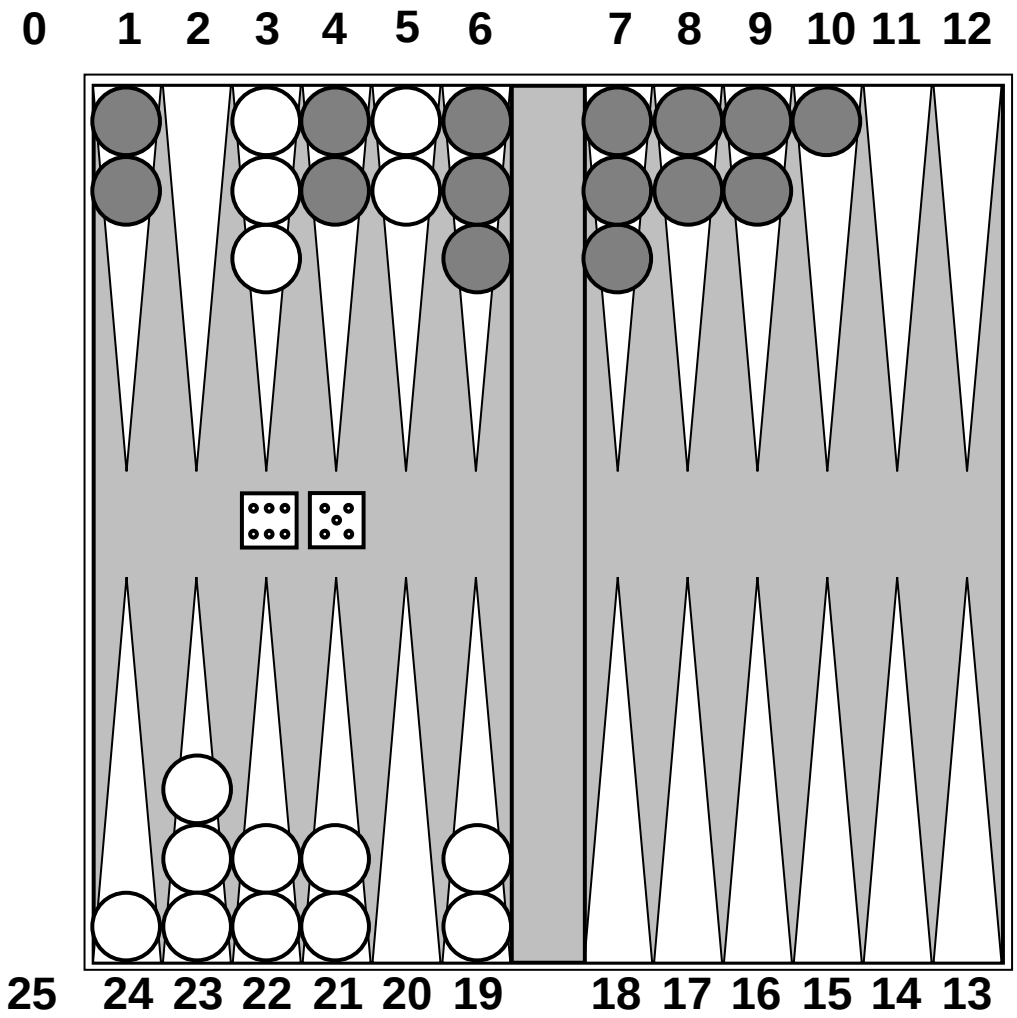
- Human champions refuse to compete – computers are too bad.
- Modern programs: MoGo, Zen, GNU Go.

Breaking news, 14th March 2016!

- AlphaGo (Google DeepMind) beats the world champion 4–1!

Outline

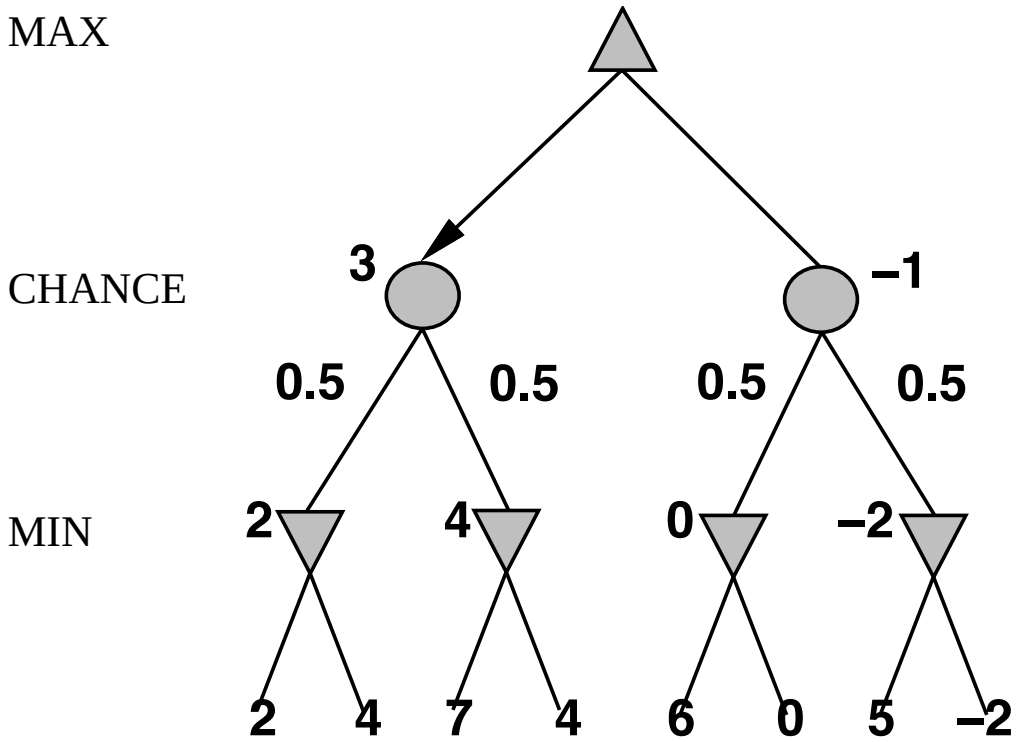
Nondeterministic games: backgammon



Nondeterministic games in general

In nondeterministic games, chance is introduced by dice, card-shuffling, etc.

Simplified example with coin-flipping:



Algorithm for nondeterministic games

The ExpectiMinimax algorithm gives perfect play

- it's just like Minimax, except we must also handle chance nodes

procedure ExpectiMinimax(*state*):
    if TerminalTest(*state*):
        return Utility(*state*)
    $A :=$ Actions(*state*)
    if *state* is a MAX node:
        return $\max_{a \in A}$ ExpectiMinimax(*state*, a)
    else if *state* is a MIN node:
        return $\min_{a \in A}$ ExpectiMinimax(*state*, a)
    else if *state* is a chance node:
        return $\sum_{a \in A} P(a)$ ExpectiMinimax(*state*, a)

where $P(a)$ is the probability that action a occurs.

Nondeterministic games in practice

Dice rolls increase the branching factor b:

- there are 21 possible rolls with 2 dice

Backgammon has ≈ 20 legal moves:

- depth 4 $\Rightarrow 20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$ nodes

As depth increases, the probability of reaching a given node shrinks:

- value of lookahead is diminished
- $\alpha - \beta$ pruning is much less effective

TDGammon (1995) used depth-2 search + very good Eval:

- the evaluation function was learned by self-play
- world-champion level