VNU-HUS MAT1206E/3508: Introduction to AI

Search, Games and Problem Solving
In-class Discussion

Hoàng Anh Đức

Bộ môn Tin học, Khoa Toán-Cơ-Tin học
Đại học KHTN, ĐHQG Hà Nội
`hoanganhduc@hus.edu.vn`

Contents

Additional Materials

Prof. Ertel's Lectures at Ravensburg-Weingarten University in 2011

- `https://youtu.be/RRO9-QXROss&t=2210` (Introduction)
- `https://youtu.be/rwefoi__Fk4` (Uninformed Search: Breadth-First Search, Depth-First Search, Iterative Deepening)
- `https://youtu.be/THZ3YxHAwno` (Heuristic Search: Greedy Search, A*-Search, IDA*-Search)
- `https://youtu.be/IW-HIOPqgsk` (Games with Opponents, Heuristic Evaluation Functions)

Introduction

- By the 1950s, as computers advanced to the point where experimenting with practical AI algorithms became feasible, *games* emerged as one of the most distinctive AI challenges (aside from tasks like cracking Nazi codes).
- Games offered a *well-defined and constrained domain* that could be *easily formalized and studied*.
- Board games such as checkers, chess, and more recently the highly complex strategy game Go (originating in China over 2500 years ago) have inspired countless researchers and continue to drive advancements in AI.
- Closely tied to games, *search and planning techniques* became a major area of progress in AI during the 1960s.
- Algorithms like the Minimax algorithm and Alpha-Beta Pruning, developed during this period, remain foundational for game-playing AI, though they have since been refined and extended with more advanced variants.
- We will explore the fundamental concepts of search, games, and problem-solving.

Introduction

Search Problems

A *search problem* is defined by the following values

State:	Description of the state of the world in which the search agent finds itself.
Starting state:	The initial state in which the search agent is started.
Goal state:	If the agent reaches a goal state, then it terminates and outputs a solution (if desired).
Actions:	All of the agents allowed actions.
Solution:	The path in the search tree from the starting state to the goal state.
Cost function:	Assigns a cost value to every action. Necessary for finding a cost-optimal solution.
State space:	Set of all states.
Search tree:	States are nodes, actions are edges.

Introduction

Example 2 (8-Puzzle, cont.)

Apply the definition to the 8-puzzle, we get

State:	3×3 matrix S with the values $1, 2, 3, 4, 5, 6, 7, 8$ (once each) and one empty square.
Starting state:	An arbitrary state.
Goal state:	An arbitrary state.
Actions:	Movement of the empty square S_{ij} to the left (if $j \neq 1$), right (if $j \neq 3$), up (if $i \neq 1$), down (if $i \neq 3$).
Solution:	The path in the search tree from the starting state to the goal state.
Cost function:	The constant function 1, since all actions have equal cost.
State space:	The state space is degenerate in domains that are mutually unreachable. (Thus there are unsolvable 8-puzzle problems.)

Introduction

Example 3 (8-Puzzle [Nilsson 1998]; [Russell and Norvig 2010])

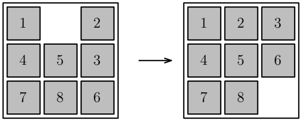


Figure: Possible starting and goal states of the 8-puzzle

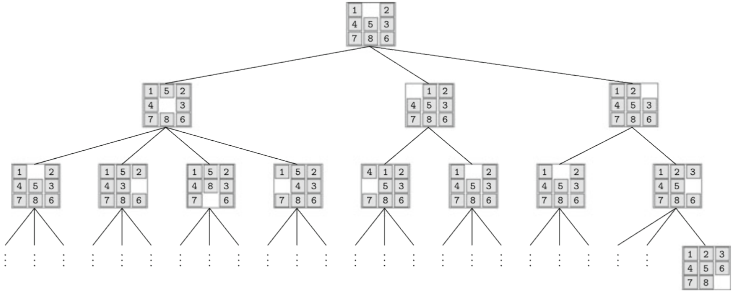


Figure: Search tree for the 8-puzzle. Bottom right a goal state in depth 3 is represented. To save space the other nodes at this level have been omitted

Introduction

For analysis of the search algorithms, the following terms are needed:

Branching factor

The *number of successor states of a state s* is called *the branching factor* $b(s)$, or b if the branching factor is constant.

Effective branching factor

The *effective branching factor* of a tree of depth d with n total nodes is defined as *the branching factor that a tree with constant branching factor, equal depth, and equal n would have*.

Complete Search Algorithms

A search algorithm is called *complete* if it *finds a solution for every solvable problem*. If a complete search algorithm terminates without finding a solution, then the problem is unsolvable.

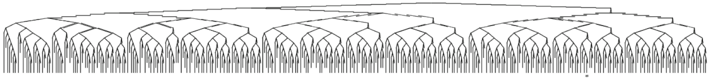
Introduction

- The *search* for a solution in an *extremely large search tree* presents a problem for nearly all inference systems.
  - From the starting state there are many possibilities for the first inference step.
  - For each of these possibilities there are again many possibilities in the next step, and so on.
- Some ideas about “large” numbers:
  - The *age of our universe* is about 1.4×10^{10} years $\approx 4.4 \times 10^{17}$ seconds.
  - The number of cells in the human body (estimated at 3.72×10^{13}, i.e., about 37.2 trillion)
  - The number of neuronal connections in the human brain (estimated at 10^{14}, i.e., about 100 trillion)

Introduction

Example 4

The search tree for SLD resolution proof of *a very simple formula* from [Ertel 1993] with *three Horn clauses*, *each with at most three literals*:



- The tree was cut off at a depth of 14 and has a solution in the leaf node marked by $\star$.
- It is only *possible to represent* it at all because of the *small branching factor (= number of children at each node) of at most two* and *a cutoff at depth 14*.

Exercise 1

(a) At most how many leaf nodes does the cut-off search tree have?

(b) At most how many inference steps are there in total in the cut-off search tree?

Introduction

Exercise 2

Assume the branching factor is a constant equal to 30 and the first solution is at depth 50. (These assumptions are completely realistic. In chess for example, there are over 30 possible moves for a typical situation, and a game lasting 50 half-turns is relatively short. (Each player has 25 moves.) In Go, the average branching factor is estimated to be about 250.)

(a) How many leaf nodes does the search tree have?

(b) How many inference steps are there in total in the search tree?

(c) Assume we had $10,000$ computers which can each perform a billion inferences per second, and that we could distribute the work over all of the computers with no cost. What would be the total computation time for all inferences?

Introduction

Questions

- Why do good chess players exist – and nowadays also good chess computers?
- Why do mathematicians find proofs for propositions in which the search space is even larger?

- Evidently we humans use *intelligent strategies* which *dramatically reduce the search space*.
  - The experienced chess player, just like the experienced mathematician, will, by mere observation of the situation, immediately rule out many actions as senseless. Through his *experience*, he has the *ability to evaluate various actions for their utility in reaching the goal*.
  - Often a person will go by *feel*. If one asks a mathematician how he found a proof, he may answer that the intuition came to him in a dream.
- In everyday problems, *intuition plays a big role*. We will later deal with this kind of *heuristic search method* and additionally *describe processes with which computers can*, similarly to humans, *improve their heuristic search strategies by learning*.

Introduction

- A tree with constant branching factor b and depth d has total

$$n = \sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1}$$

nodes.

Theorem 1

For heavily branching finite search trees with a large constant branching factor, almost all nodes are on the last level.

Introduction

Example 5 (Shortest Path from City A to City B)

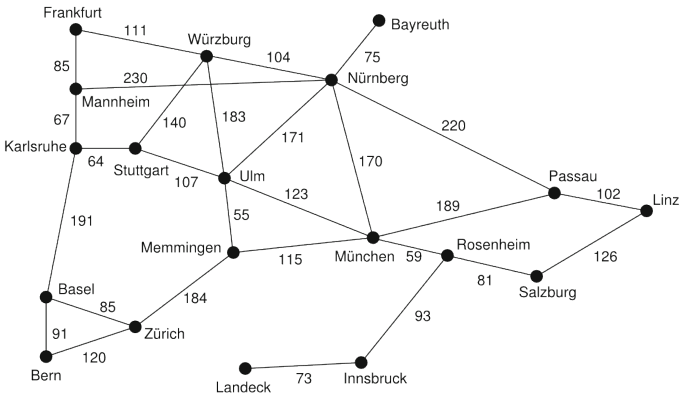


Figure: The graph of southern Germany as an example of a search task with a cost function.

Introduction

Example 3 (cont.)

State: A city as the current location of the traveler.

Starting state: An arbitrary city A.

Goal state: An arbitrary city B.

Actions: Travel from the current city to a neighboring city.

Cost function: The distance between the cities. Each action corresponds to an edge in the graph with the distance as the weight.

State space: All cities, that is, nodes of the graph.

Optimal Search Algorithms

A search algorithm is called *optimal* if it *always finds the solution with the lowest cost*, provided that at least one solution exists.

Introduction

Deterministic Problem

Every action leads from a state to a unique successor state.

Observable Problem

The agent always knows which state it is in.

Example 4

- The 8-puzzle problem is deterministic and observable.
- In route planning in real applications, both characteristics are not always given.
  - The action “Drive from Munich to Ulm” may—for example because of an accident—lead to the successor state “Munich”.
  - It can also occur that the traveler no longer knows where he is because he got lost.

Introduction

- We want to *ignore all kinds of complications* similar to those in the route planning problems. Therefore, we will *only look at problems that are deterministic and observable*.
- Deterministic and observable problems *make action planning relatively simple* because, due to *having an abstract model*, it is possible to *find action sequences for the solution of the problem without actually carrying out the actions in the real world*. $\Rightarrow$ Offline algorithms.
- One faces *much different challenges* when, for example, *building robots that are supposed to play soccer*. *Here there will never be an exact abstract model of the actions.* $\Rightarrow$ Online algorithms (which make decisions based on sensor signals in every situation). (*Reinforcement learning* works toward optimization of these decisions based on experience.)

Uninformed Search
Breadth-First Search

```
BREADTHFIRSTSEARCH(NodeList, Goal)

1  NewNodes = $\emptyset$
2  **for all** Node $\in$ NodeList
3      **if** GoalReached(Node, Goal)
4          **return** ("Solution found", Node)
5      NewNodes = **append**(NewNodes, Successors(Node))
6  **if** NewNodes $\neq \emptyset$
7      **return** BREADTHFIRSTSEARCH(NewNodes, Goal)
8  **else**
9      **return** ("No solution")
```

Uninformed Search
Breadth-First Search

Key Concepts:

- Breadth-First Search (BFS) explores the search tree level by level.
- Uses a **First-In-First-Out (FIFO)** strategy for node expansion.
- The algorithm stops as soon as the goal node is found or all nodes are exhausted.

Algorithm Steps:

1. Initialize an empty list for new nodes.
2. For each node in the current list:
   - Check if it is the goal node.
   - If not, generate its successors and add them to the new list.
3. Recursively call BFS on the new list of nodes.
4. If no new nodes are generated, return “No solution.”

Important Functions:

- `GoalReached(Node, Goal)`: Checks if the current node matches the goal.
- `Successors(Node)`: Generates all possible child nodes of the current node

Uninformed Search
Breadth-First Search

Characteristics:

- **Complete:** Will find a solution if one exists.
- **Optimal:** Guarantees the optimal (lowest cost) solution when all step costs are equal.
  - Otherwise, BFS may not find an optimal solution.
- **Time Complexity:** $O(b^d)$, where b is the branching factor and d is the depth of the solution.
- **Space Complexity:** $O(b^d)$, as all nodes at the current depth are stored in memory.

When all costs are not the same, we use a variant of BFS.

Uniform Cost Search

= BFS + The node with the lowest cost from the list of nodes (which is sorted ascendingly by cost) is always expanded, and the new nodes sorted in. $\Rightarrow$ Always optimal!

Uninformed Search

Depth-First Search

```
DEPTHFIRSTSEARCH(Node, Goal)
1  if GoalReached(Node, Goal)
2       return (“Solution found”)
3  NewNodes = Successors(Node)
4  while NewNodes $\neq \emptyset$
5       Result = DEPTHFIRSTSEARCH(First(NewNodes), Goal)
6       if Result = “Solution found”
7            return (“Solution found”)
8       NewNodes = Rest(NewNodes)
9  return (“No solution”)
```

Uninformed Search

Depth-First Search

Key Concepts:

- Depth-First Search (DFS) explores as far as possible along each branch before backtracking.
- Uses a **Last-In-First-Out (LIFO)** strategy for node expansion.
- Only the successors of the current node are stored in memory.

Algorithm Steps:

1. Check if the current node is the goal node.
2. If not, generate its successors.
3. Recursively expand the first successor node.
4. If no solution is found, backtrack to the last branch and expand the next successor.

Important Functions:

- `GoalReached(Node, Goal)`: Checks if the current node matches the goal.
- `First`: Retrieves the first element of a list.
- `Rest`: Retrieves the remaining elements of a list.
- `Successors(Node)`: Generates all possible child nodes of the current node.

Characteristics:

- **Incomplete:** May enter infinite loops in infinite-depth trees.
- **Non-optimal:** Does not guarantee the optimal solution (e.g., when there is no solution in the far left branch and the tree has infinite depth).
- **Time Complexity:** $O(b^d)$, where b is the branching factor and d is the depth of the solution.
- **Space Complexity:** $O(bd)$, as only the current path and its successors are stored in memory.

To address the limitations of DFS:

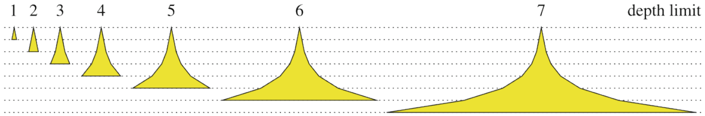
Depth-Limited Search

A variant of DFS with a depth limit ($\Rightarrow$ *Pruned search tree*). It avoids infinite loops but may miss solutions beyond the limit.

Uninformed Search
Iterative Deepening

Iterative Deepening

We begin the depth-first search with a depth limit of 1. If no solution is found, we raise the limit by 1 and start searching from the beginning, and so on.



Uninformed Search

Iterative Deepening

```
ITERATIVEDEEPENING(Node, Goal)
1  DepthLimit = 0
2  repeat
3      Result = DEPTHFIRSTSEARCH-B(Node, Goal, 0, DepthLimit)
4      DepthLimit = DepthLimit + 1
5  until Result = “Solution found”
```

```
DEPTHFIRSTSEARCH-B(Node, Goal, Depth, Limit)
1  if GoalReached(Node, Goal)
2      return (“Solution found”)
3  NewNodes = Successors(Node)
4  while NewNodes $\neq \emptyset$ and Depth $<$ Limit
5      Result = DEPTHFIRSTSEARCH-B(First(NewNodes),
       Goal, Depth+1, Limit)
6      if Result = “Solution found”
7          return (“Solution found”)
8      NewNodes = Rest(NewNodes)
9  return (“No solution”)
```

Uninformed Search

Iterative Deepening

Analysis:

- complete.
- optimal, if costs are constant and increment = 1
- Computation time = $O(b^d)$
- Memory requirement = $O(bd)$.
- One could argue that *repeatedly re-starting depth-first search at depth zero causes a lot of redundant work*. *For large branching factors this is not the case*. Indeed, *the computation time for all iterations besides the last can be ignored*. **[Why?]**

Uninformed Search
Comparison

	Breadth-first search	Uniform cost search	Depth-first search	Iterative deepening
Completeness	Yes	Yes	No	Yes
Optimal solution	Yes (*)	Yes	No	Yes (*)
Computation time	b^d	b^d	∞ or b^{d_s}	b^d
Memory use	b^d	b^d	bd	bd

- (*): only true with constant action cost.
- d_s is the maximal depth for a finite search tree.

Uninformed Search

Cycle Check

Exercise 3

Nodes may be repeatedly visited during a search. Such cycles can be prevented by *recording within each node all of its predecessors* and, *when expanding a node, comparing the newly created successor nodes with the predecessor nodes*. All of the *duplicates found can be removed from the list of successor nodes*.

How would a check on cycles of arbitrary length affect the search performance?

Heuristic Search

- *Heuristics* are problem-solving strategies which in many cases find a solution faster than uninformed search.
- There is no guarantee!
- In everyday life, heuristic methods are important.
- *Realtime-decisions under limited resources*.
- A good solution found quickly is preferred over a solution that is optimal, but very expensive to derive.

Mathematical Modeling:

- *Heuristic evalutation function* $f(s)$ for states
- **Goal:** Find, with little effort, a *solution* to the stated search problem with *minimal total cost*.
- Node = state + heuristic evaluation + ...

Heuristic Search

```
HEURISTICSEARCH(Start, Goal)
1  NodeList = [Start]
2  while True
3      if NodeList = ∅
4          return ("No solution")
5      Node = First(NodeList)
6      NodeList = Rest(NodeList)
7      if GoalReached(Node, Goal)
8          return ("Solution found", Node)
9      NodeList = SortIn(Successors(Node), NodeList)
```

With appropriate evaluation functions, one can generate BFS and DFS from HEURISTICSEARCH

- SortIn(X,Y) inserts the elements from the unsorted list X into the ascendingly sorted list Y.
  - *The heuristic rating is used as the sorting key*. Thus it is guaranteed that *the best node* (that is, the one with the lowest heuristic value) is *always at the beginning of the list*.
  - *When sorting in a new node* from the node list, *it may be advantageous to check whether the node is already available* and, if so, to delete the duplicate.

Heuristic Search

Ideally, the *best heuristic* would be *a function that calculates the actual costs from each node to the goal*.

Question

How do we find a heuristic that is fast and simple to compute?

An idea for finding a heuristic *simplification of the problem*.

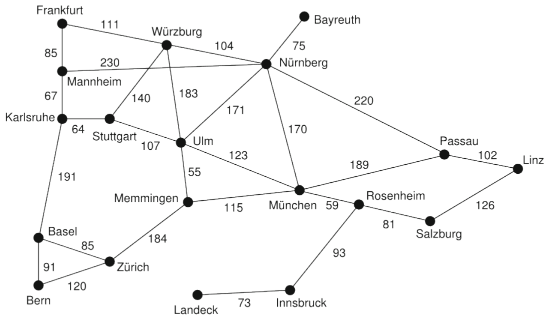
- The *original task* is *simplified enough* that *it can be solved with little computational cost*.
- The *costs from a state to the goal in the simplified problem* then *serve as an estimate for the actual problem*.
  $\Rightarrow$ *cost estimate function* h.

Heuristic Search

Greedy Search

Example 5 (Shortest Path from City A to City B)

- **Simplified Task:** finding the straight line path from city to city (that is, the flying distance).
- *Cost estimate function $h(s)$* = flying distance from city s to Ulm (given in the figure below).



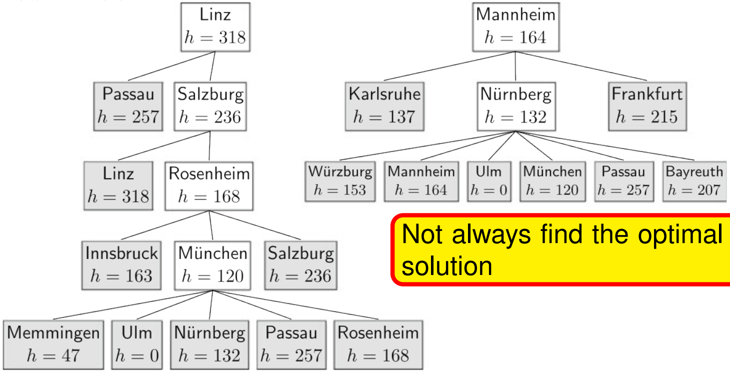
Basel	204
Bayreuth	207
Bern	247
Frankfurt	215
Innsbruck	163
Karlsruhe	137
Landeck	143
Linz	318
München	120
Mannheim	164
Memmingen	47
Nürnberg	132
Passau	257
Rosenheim	168
Stuttgart	75
Salzburg	236
Würzburg	153
Zürich	157

Figure: City graph with flying distances from all cities to Ulm

Heuristic Search
Greedy Search

Example 5 (cont.)

We use the cost estimate function $h(s)$ directly as the evaluation function in the HEURISTICSEARCH, i.e., we set $f(s) = h(s)$.

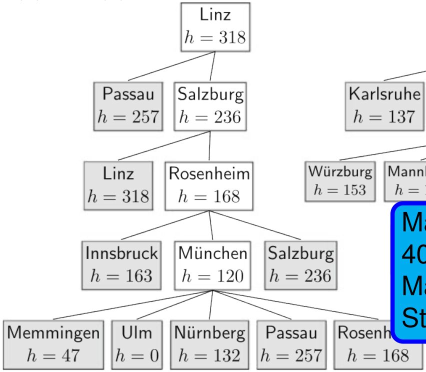


Not always find the optimal solution

Figure: Greedy search: from Linz to Ulm (left) and from Mannheim to Ulm (right).

Example 5 (cont.)

We use the cost estimate function $h(s)$ directly as the evaluation function in the HEURISTICSEARCH, i.e., we set $f(s) = h(s)$.



Mannheim–Nürnberg–Ulm: 401 km
Mannheim–Karlsruhe–Stuttgart–Ulm: 323 km

Figure: Greedy search: from Linz to Ulm (left) and from Mannheim to Ulm (right).

Heuristic Search

A*-Search

- *Cost function* $g(s)$ = Sum of accrued costs from root to current node s.
- *Heuristic cost estimate* $h(s)$ = Estimated cost from current node s to goal.
- *Heuristic evaluation function* $f(s) = g(s) + h(s)$.

Admissible heuristic cost estimate function

A heuristic cost estimate function $h(s)$ that *never overestimates the actual cost* from state s to the goal is called *admissible*.

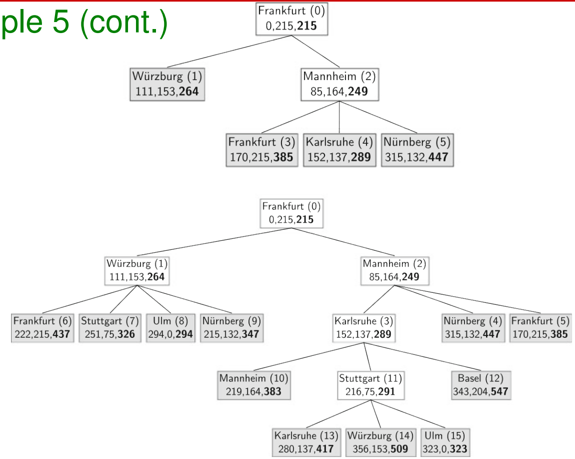
A*-algorithm

$= \boxed{\text{HEURISTICSEARCH}} + \boxed{f(s) = g(s) + h(s)} + \boxed{\text{admissible } h}$

Heuristic Search

A*-Search

Example 5 (cont.)



Frankfurt (0) 0,215,**215**
Würzburg (1) 111,153,**264**
Mannheim (2) 85,164,**249**
Frankfurt (3) 170,215,**385**
Karlsruhe (4) 152,137,**289**
Nürnberg (5) 315,132,**447**
Frankfurt (0) 0,215,**215**
Würzburg (1) 111,153,**264**
Mannheim (2) 85,164,**249**
Frankfurt (6) 222,215,**437**
Stuttgart (7) 251,75,**326**
Ulm (8) 294,0,**294**
Nürnberg (9) 215,132,**347**
Karlsruhe (3) 152,137,**289**
Nürnberg (4) 315,132,**447**
Frankfurt (5) 170,215,**385**
Mannheim (10) 219,164,**383**
Stuttgart (11) 216,75,**291**
Basel (12) 343,204,**547**
Karlsruhe (13) 280,137,**417**
Würzburg (14) 356,153,**509**
Ulm (15) 323,0,**323**

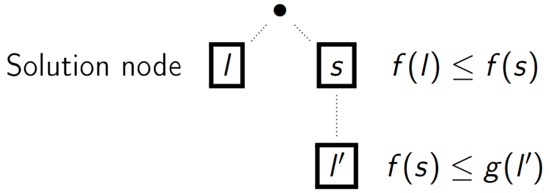
Figure: Two snapshots of the A* search tree for the optimal route from Frankfurt to Ulm. In the boxes below the name of the city s we show $g(s), h(s), \mathbf{f}(\mathbf{s})$. Numbers in parentheses after the city names show the order in which the nodes have been generated by the Successors function

Theorem 2

The A algorithm is optimal. That is, it always finds the solution with lowest total cost if the heuristic h is admissible.*

Proof.

The *first solution node l* found by A* *never has a higher cost than another arbitrary solution node l'*, i.e., $g(l) \leq g(l')$.



Definition of f $\left[f(s) = \text{estimated cost from start to goal via } s \right]$

$$\underbrace{g(l) = g(l) + h(l)}_{\substack{l \text{ is a solution node} \\ \Leftrightarrow h(l) = 0}} = \underbrace{f(l) \leq f(s)}_{\substack{\text{list of open nodes is sorted} \\ \text{in ascending order of } f}} = \underbrace{g(s) + h(s) \leq g(l')}_{\substack{\text{the heuristic is admissible} \\ \left(\text{estimated cost to the goal} \leq \text{actual cost to the goal}\right)}}.$$

- Weaknesses of A*:
  - High memory requirements
  - List of open nodes must be sorted $\Rightarrow$ Heapsort (logarithmic time complexity for insertion and removal of nodes)
- Solution: IDA*-algorithm [Korf 1985]
  - Iterative Deepening
  - The same as depth-first search, but limit for heuristic evaluation $f(s)$ (instead of depth limit).
    - Perform a DFS, cut off a branch when $f(s)$ exceeds a given threshold
    - This threshold starts at the estimate of the cost at the initial state, and increases for each iteration of the algorithm.
    - At each iteration, the threshold used for the next iteration is the minimum cost of all values that exceeded the current threshold.

Heuristic Search
Summary

- Of the various search algorithms for uninformed search, *iterative deepening is the only practical one* because it is complete and can get by with very little memory.
- IDA* is complete, fast and memory efficient.
- *Good* heuristics greatly reduce the effective branching factor.
- *Heuristics have no performance advantage for unsolvable problems* because *the unsolvability of a problem can only be established when the complete search tree has been searched through*.
  - For solvable problems, heuristics often reduce computation time dramatically, but for unsolvable problems the cost can even be higher with heuristics.
  - For example, in the proof of PL1 formulas (which is undecidable), the search tree can be infinitely deep. This means that, in the unsolvable case, the search potentially never ends.

Heuristic Search

Summary

Question

How to find good heuristics?

- Manually, e.g. by simplification of the problem.
  - Simplify the original problem.
  - Solve the simplified (easier) version.
  - Optimal solutions from the simplified version $\Rightarrow$ heuristic functions.
- Automatic generation of heuristics by machine-learning techniques.

Games with Opponents

- Games for two players
- Chess, Checkers, Othello, Go
- deterministic (= every action (a move) results in the same child state given the same parent state), observable (= every player always knows the complete game state)
- Card games: only partially observable.
  - The player does not know the other players' cards, or only has partial knowledge about them.
- Zero-sum games: Win + Loss = 0
  - Every gain one player makes means a loss of the same value for the opponent.

Games with Opponents

Minimax Search

Characteristics of games:

- The *effective branching factor in chess* is *around* 30 *to* 35.
- 50 moves per player: $30^{100} \approx 10^{148}$ leaf nodes $\Rightarrow$ *no chance to fully explore the search tree*.
- *Real-time requirement* (chess is often played with a time limit) $\Rightarrow$ Limited search depth.
- Among the leaf nodes of this depth-limited tree there are normally no solution nodes (that is, nodes which terminate the game) $\Rightarrow$ *heuristic evaluation function* B *for board positions*.
  - The level of play strongly depends on the quality of the function B.
- Player: Max, Opponent: Min.
- **Assumption:** Opponent Min always makes the best move he can.
- The higher the evaluation $B(s)$ for position s, the better position s is for the player Max and the worse it is for his opponent Min.
  - Max maximizes the evaluation of his moves.
  - Min minimizes evaluation of his moves.

Games with Opponents

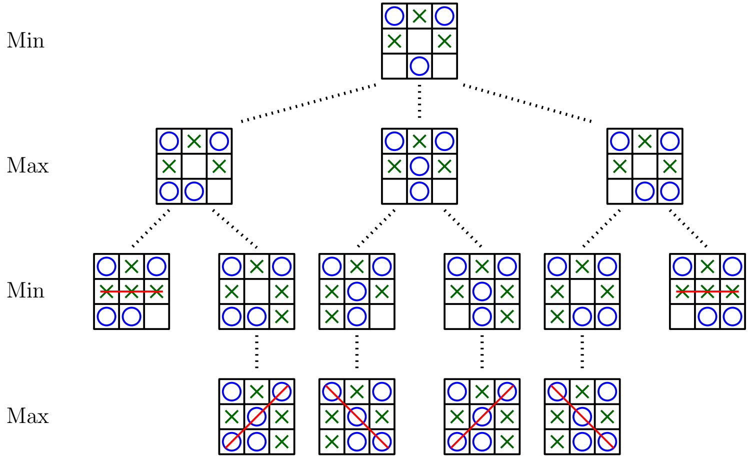
Minimax Search: Tic-Tac-Toe Example

- **Game Setup:** A 3x3 Tic-Tac-Toe board with two players: Max (X) and Min (O). Players alternate turns, with Max going first
- **Objective:** Max aims to win (+1), Min aims to win (-1), and a draw results in 0. A play wins by placing three of their marks in a horizontal, vertical, or diagonal row.
- **Minimax Algorithm:**
  - Max chooses the move with the highest score (= evaluation function $B(s)$ for position s).
  - Min chooses the move with the lowest score (= evaluation function $B(s)$ for position s).

Games with Opponents

Minimax Search: Tic-Tac-Toe Example

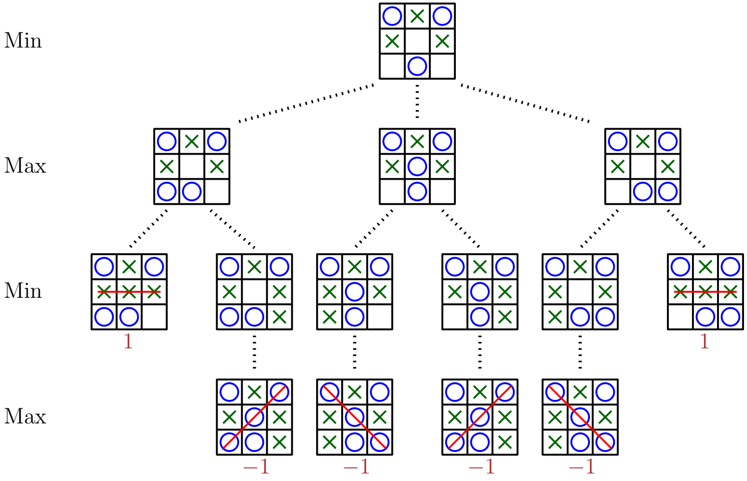
Example: A partial game tree starting with Min's turn. The tree alternates between Min placing ◯ and Max placing ×, with the current player shown on the left.



Games with Opponents

Minimax Search: Tic-Tac-Toe Example

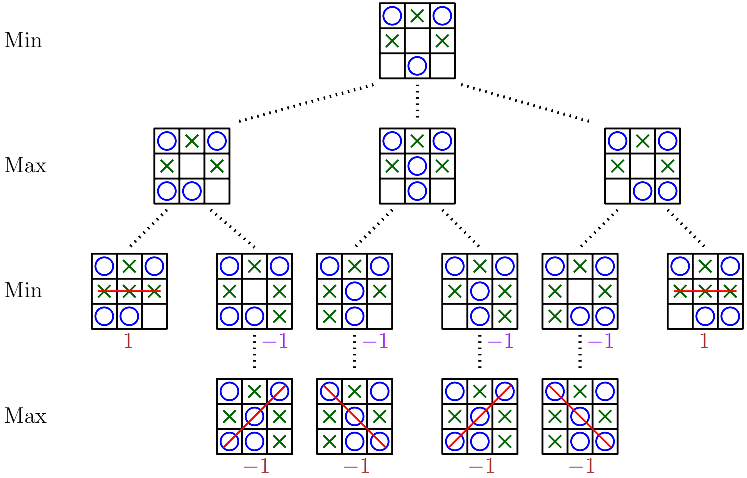
Example: A partial game tree starting with Min's turn. The tree alternates between Min placing ○ and Max placing ×, with the current player shown on the left.



Games with Opponents

Minimax Search: Tic-Tac-Toe Example

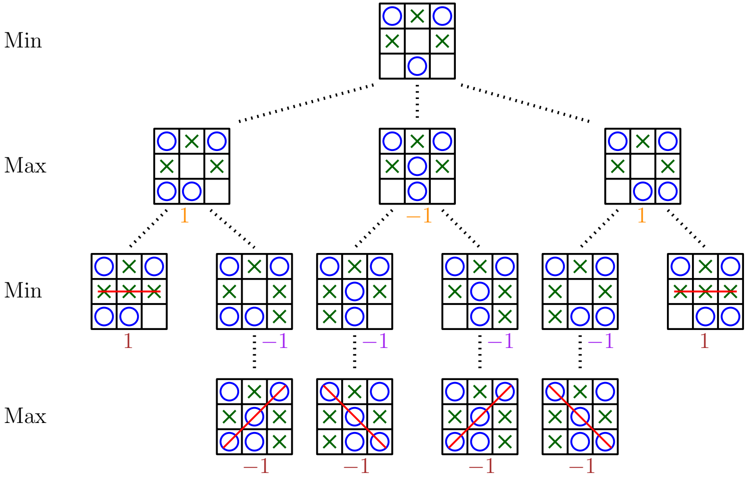
Example: A partial game tree starting with Min's turn. The tree alternates between Min placing ○ and Max placing ×, with the current player shown on the left.



Games with Opponents

Minimax Search: Tic-Tac-Toe Example

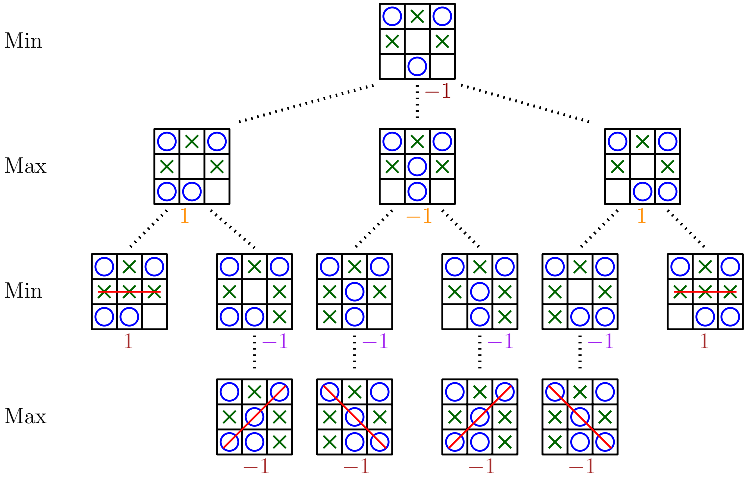
Example: A partial game tree starting with Min's turn. The tree alternates between Min placing ○ and Max placing ×, with the current player shown on the left.



Games with Opponents

Minimax Search: Tic-Tac-Toe Example

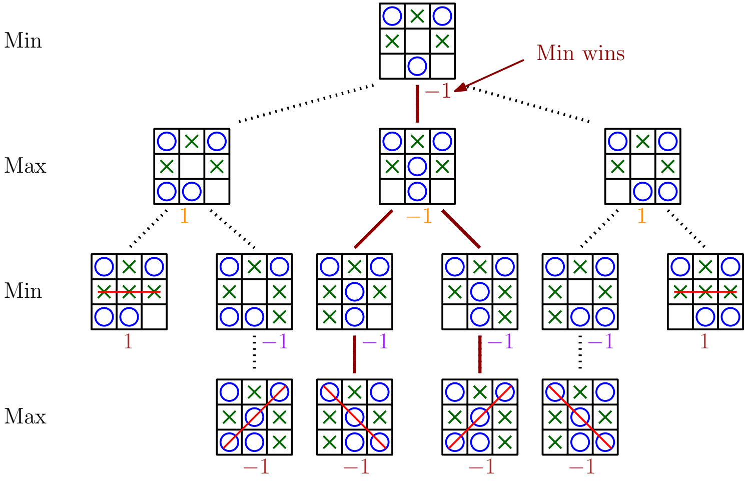
Example: A partial game tree starting with Min's turn. The tree alternates between Min placing ○ and Max placing ×, with the current player shown on the left.



Games with Opponents

Minimax Search: Tic-Tac-Toe Example

Example: A partial game tree starting with Min's turn. The tree alternates between Min placing ○ and Max placing ×, with the current player shown on the left.

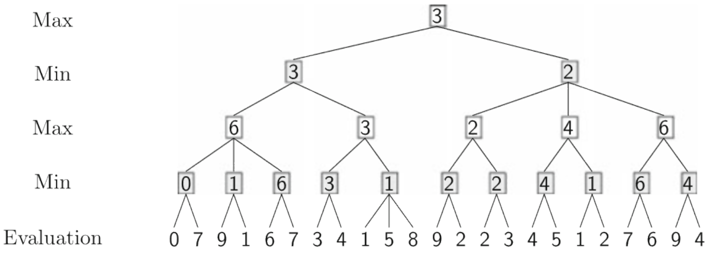


Games with Opponents

Minimax Search

Exercise 4

The minimax evaluation of a game tree is shown in the figure below [Ertel 2025], Fig. 6.19, p. 117. The look-ahead is four half-moves, and the evaluations of all leaves are given. Explain how the evaluations of the inner nodes are derived.



Games with Opponents

Alpha-Beta Pruning

- **Key Idea:** Alpha-Beta Pruning avoids evaluating branches of the game tree that cannot influence the final decision based on the values already discovered during the search.
- It uses two values: *Alpha* α and *Beta* β.

Alpha and Beta Values

- **Alpha:** Represents the best (highest-value) score that the maximizing player (usually the AI) can guarantee so far. It acts as a *lower bound*. The initial value of Alpha is $-\infty$.
- **Beta:** Represents the best (lowest-value) score that the minimizing player (the opponent) can guarantee so far. It acts as an *upper bound*. The initial value of Beta is $+\infty$.

Games with Opponents

Alpha-Beta Pruning Process

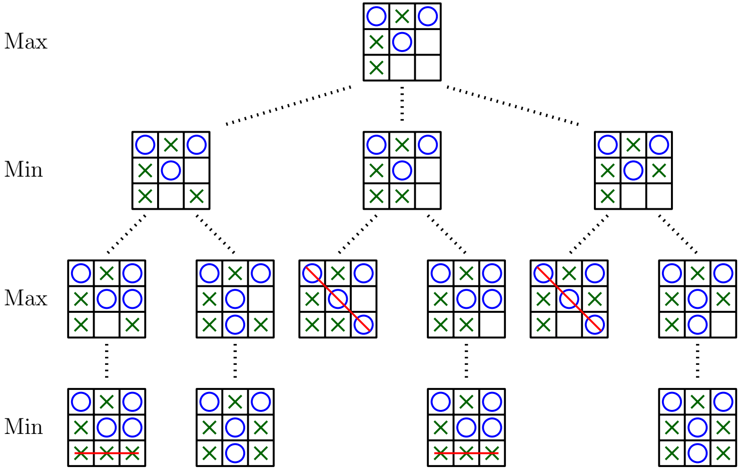
- As the tree is explored, the Alpha and Beta values are updated.
- When exploring a node, the algorithm compares the node's value against these bounds:
  - If *Alpha* $\geq$ *Beta*, the current branch will not affect the final decision.
  - This is because the opponent will avoid this path in favor of a better one.
- When this condition is met, the branch is *pruned*, and the algorithm moves on to the next branch.

Benefits of Alpha-Beta Pruning

- Skips large parts of the tree.
- Significantly reduces the number of nodes to be evaluated.
- Maintains the same optimal decision as the full Minimax algorithm.

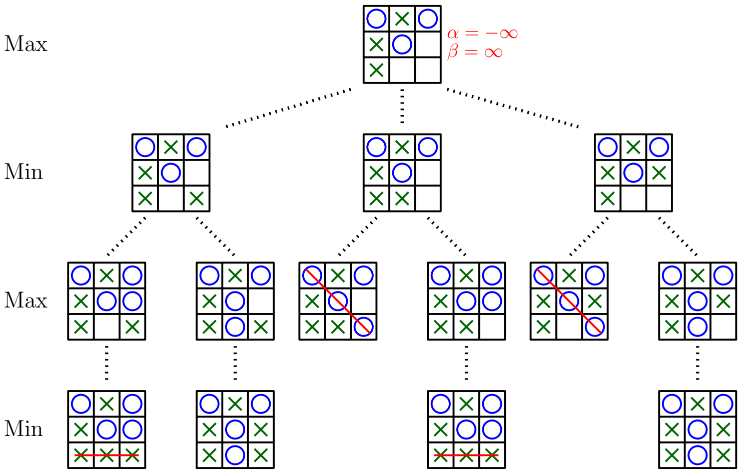
Games with Opponents

Alpha-Beta Pruning Process



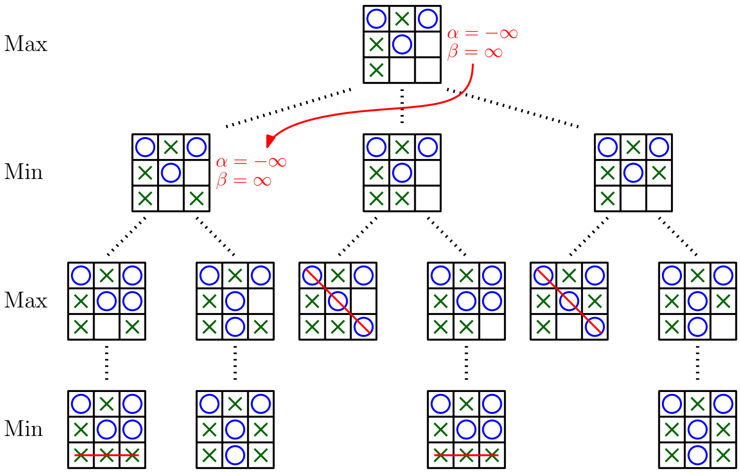
Games with Opponents

Alpha-Beta Pruning Process



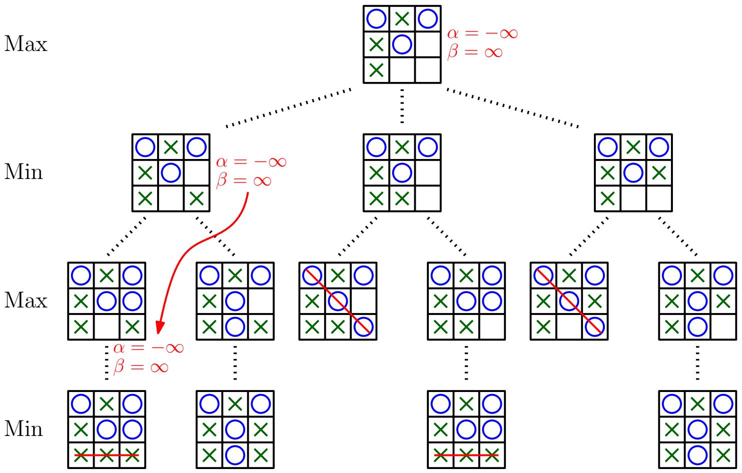
Games with Opponents

Alpha-Beta Pruning Process

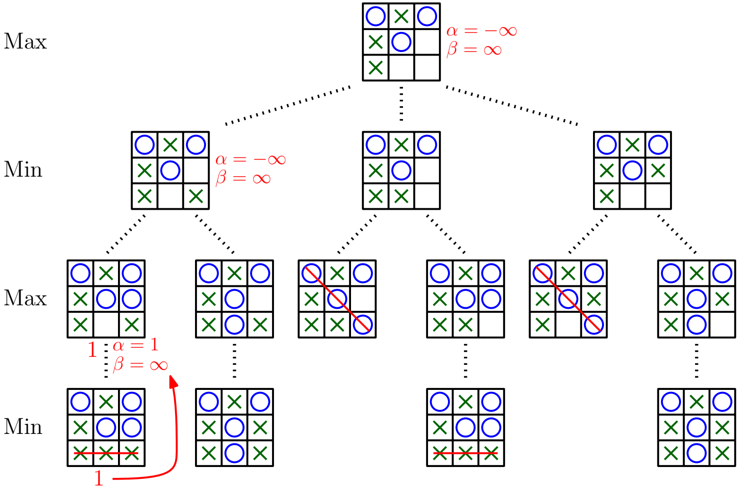


Games with Opponents

Alpha-Beta Pruning Process

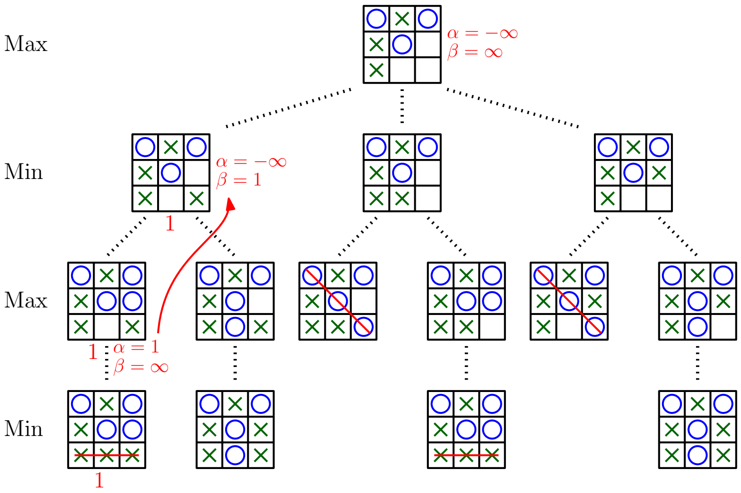


Games with Opponents
Alpha-Beta Pruning Process



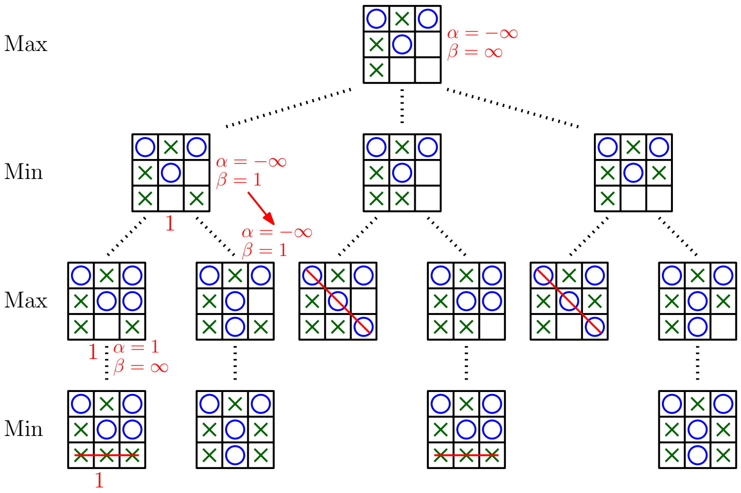
Games with Opponents

Alpha-Beta Pruning Process



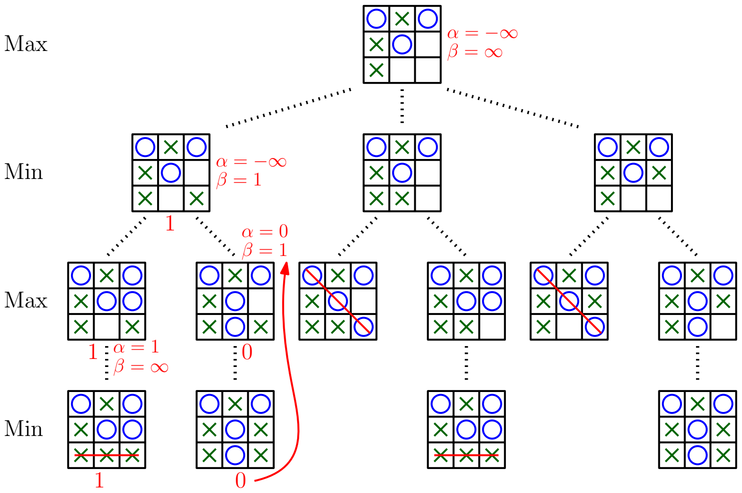
Games with Opponents

Alpha-Beta Pruning Process

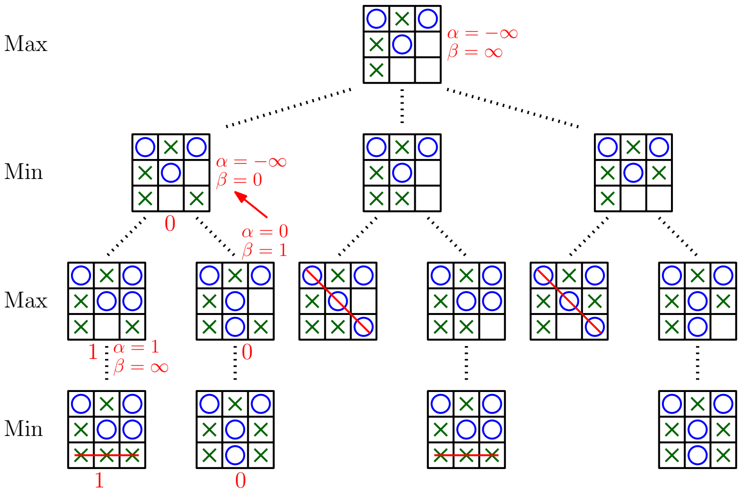


Games with Opponents

Alpha-Beta Pruning Process

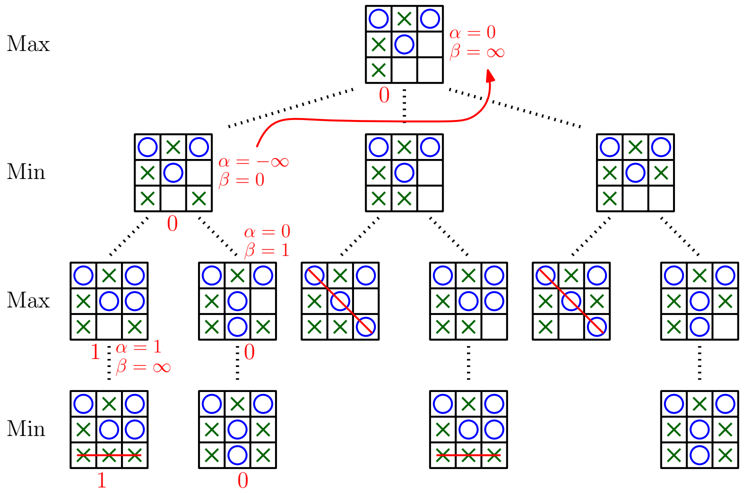


Games with Opponents
Alpha-Beta Pruning Process



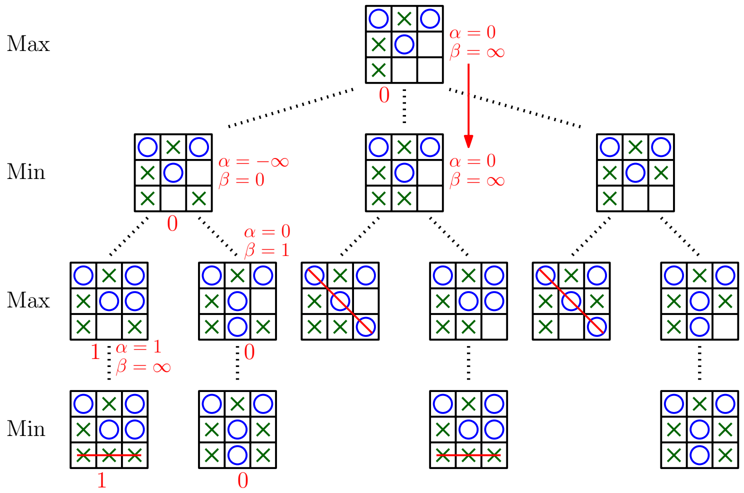
Games with Opponents

Alpha-Beta Pruning Process



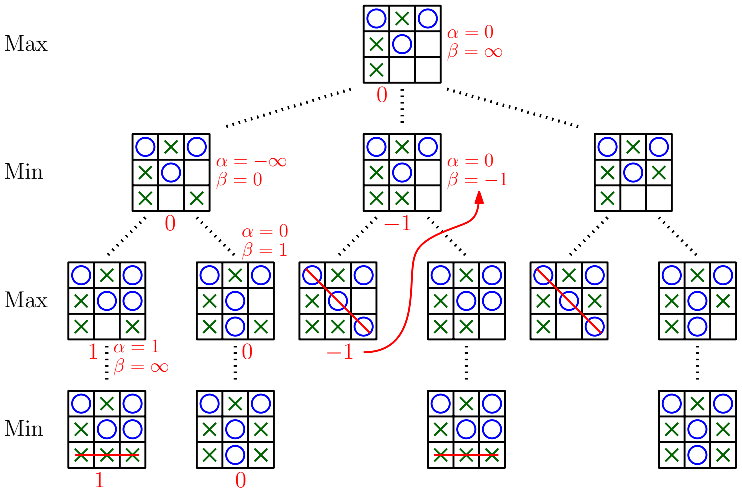
Games with Opponents

Alpha-Beta Pruning Process



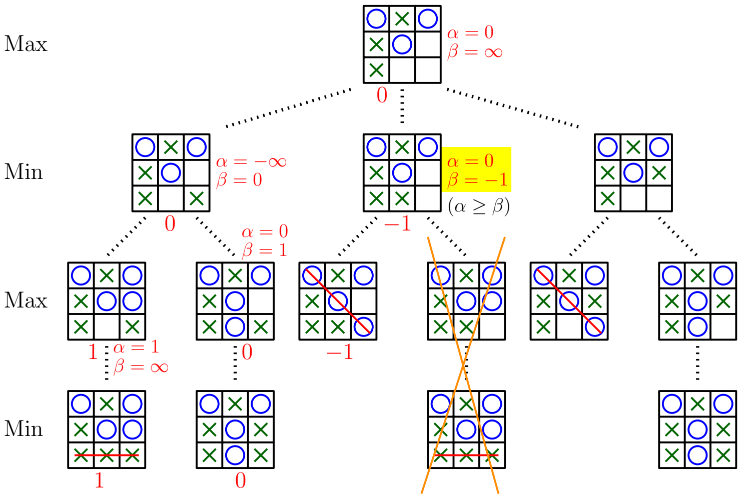
Games with Opponents

Alpha-Beta Pruning Process



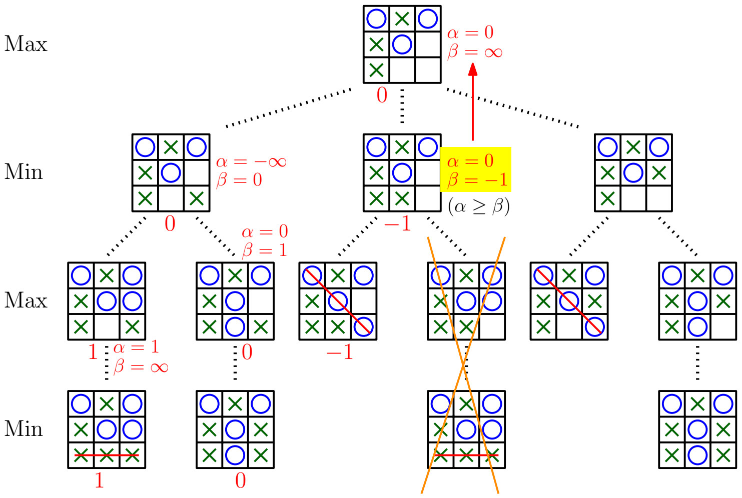
Games with Opponents

Alpha-Beta Pruning Process



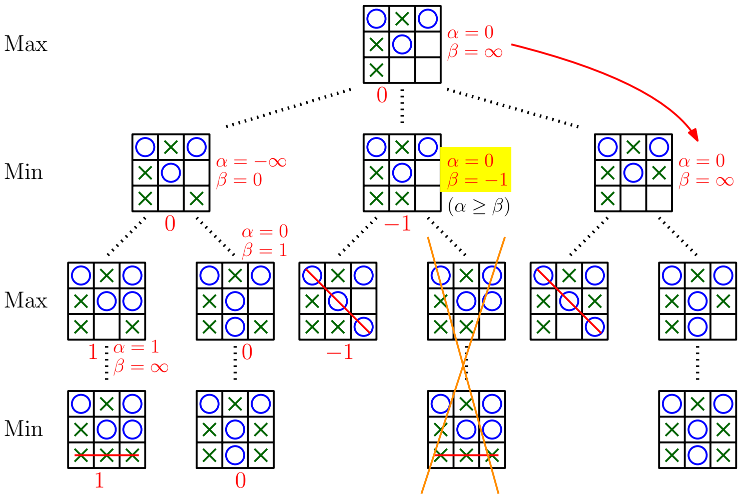
Games with Opponents

Alpha-Beta Pruning Process



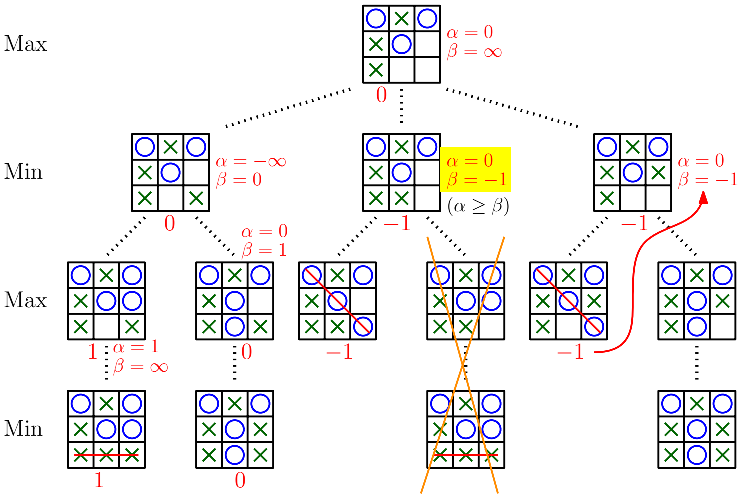
Games with Opponents

Alpha-Beta Pruning Process



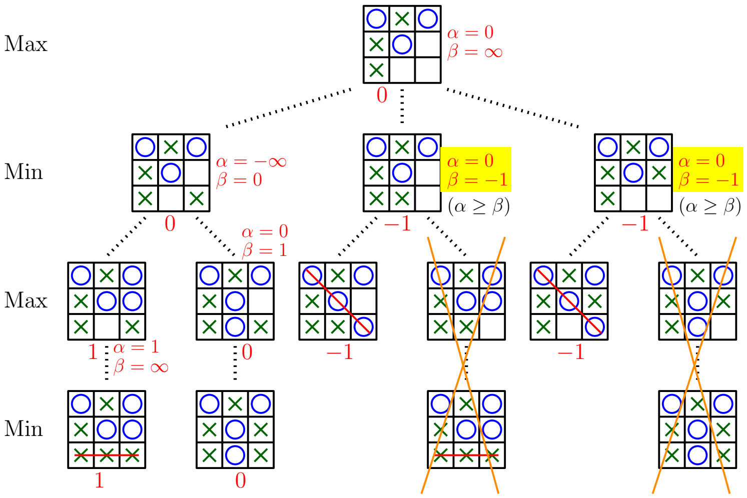
Games with Opponents

Alpha-Beta Pruning Process



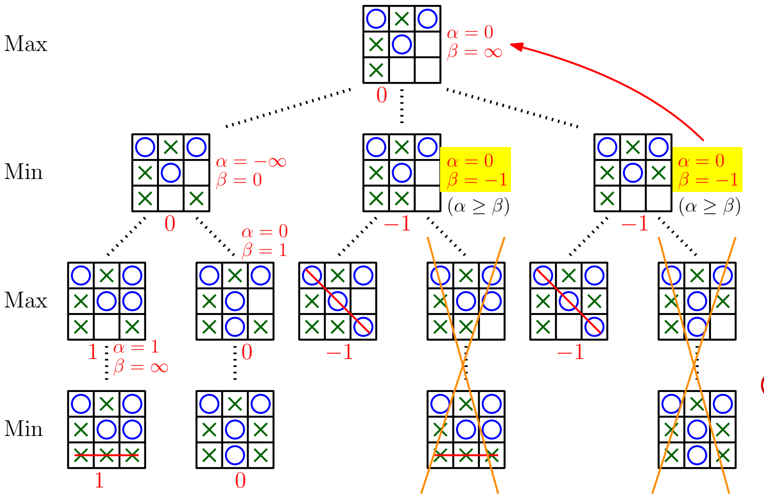
Games with Opponents

Alpha-Beta Pruning Process



Games with Opponents

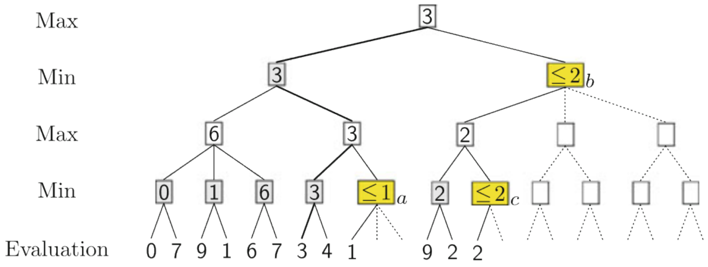
Alpha-Beta Pruning Process



Games with Opponents

Alpha-Beta Pruning Process

The alpha-beta minimax evaluation of a game tree is shown in the figure below [Ertel 2025], Fig. 6.20, p. 118. The look-ahead is four half-moves, and the evaluations of all leaves are given. Explain how the evaluations of the inner nodes are derived and how to decide which branch to prune.



Games with Opponents
Alpha-Beta-Pruning

Analysis: (see [Pearl 1984])

- *Computation time* heavily depends on *the order in which child nodes are traversed*
- *Worst-Case:* does not offer any advantage.
  - *Successors of maximum nodes are sorted in ascending order, successors of minimum nodes are sorted in descending order*.
  - With constant branching factor b, the number n_d of leaf nodes to evaluate at depth d is $n_d = b^d$.
- *Best-Case:*
  - *Successors of maximum nodes are sorted in descending order, successors of minimum nodes are sorted in ascending order*.
  - Effective branching factor $\approx \sqrt{b} \Rightarrow n_d \approx \sqrt{b}^d = b^{d/2}$ leaf nodes. $\Rightarrow$ Search horizon is doubled. (E.g., if you have computational resources to do a full search with depth d_1 then in the best case, with that same resources, you can search with depth $2d_1$ using alpha-beta pruning.)
  - In chess, this means effective branching factor reduces from 35 to about $6 \approx \sqrt{35}$.

Games with Opponents

Alpha-Beta-Pruning

Analysis (cont.): (see [Pearl 1984])

- *Average-Case:*
  - *The child nodes are randomly sorted*.
  - Effective branching factor $\approx b^{3/4} \Rightarrow n_d \approx b^{3d/4}$ leaf nodes.
  - In chess, this means effective branching factor reduces from 35 to about 14.
- *Heuristic node order:*
  - Connect alpha-beta pruning with iterative deepening over the depth limit $\Rightarrow$ *At every new depth limit we can access the ratings of all nodes of previous levels and order the successors at every branch*.
  - Effective branching factor of roughly 7 to 8, which is close to the optimal value $\sqrt{35} \approx 6$.

Games with Opponents

Non-deterministic Games

- e.g. dice games
- In the game tree, there are three types of levels in the sequence Max, dice, Min, dice, ..., where each dice roll node branches six ways.
- Average the values of all rolls

Heuristic Evaluation Functions

The following example illustrates how to *find good heuristic evaluation functions using the knowledge of human experts* in a chess program [Frayn 2005].

Example 6

- Experts are questioned about the *most important factors in the selection of a move* $\Rightarrow$ these factors are *quantified* $\Rightarrow$ *list of relevant features or attributes*.
- These are then (in the simplest case) combined into a *linear evaluation function $B(s)$ for positions*, which could look like

$$\begin{aligned} B(s) &= a_1 \cdot \text{material} + a_2 \cdot \text{pawn\_structure} + \\ &+ a_3 \cdot \text{king\_safety} + a_4 \cdot \text{knight\_in\_center} + \\ &+ a_5 \cdot \text{bishop\_diagonal\_coverage} + \cdots \\ \text{material} &= \text{material}(\text{own\_team}) - \text{material}(\text{opponent}) \\ \text{material}(\text{team}) &= \text{num\_pawns}(\text{team}) \cdot 100 + \text{num\_knights}(\text{team}) \cdot 300 + \\ &+ \text{num\_bishops}(\text{team}) \cdot 300 + \text{num\_rooks}(\text{team}) \cdot 500 + \\ &+ \text{num\_queens}(\text{team}) \cdot 900 \end{aligned}$$

- *Weights a_i are set intuitively* after discussion with experts + *changed after each game* based on positive and negative experience. *Better:* optimizing weights by *machine-learning methods*.

Heuristic Evaluation Functions

Example 7 (cont.)

Optimizing weights by *machine-learning methods*.

- Expert is only asked about relevant features $f_1(s), f_2(s), \ldots$
- A machine learning process is used to find an evaluation function that is as close to optimal as possible.
  - Start with an initial pre-set evaluation function (determined by the learning process).
  - Let the chess program play.
  - At the end of the game a rating is derived from the result (victory, defeat, or draw).
  - Based on this rating, the evaluation function is changed with the goal of making fewer mistakes next time.
- Problems:
  - *Credit Assignment*
  - positive or negative feedback only at the end
  - no ratings for individual moves
  - Feedback for actions of the past? $\Rightarrow$ *Reinforcement Learning*
- *Most of the world-best chess computers still work without machine-learning techniques*. Reasons:
  - Reinforcement learning has large computation times
  - Manually created heuristics are already heavily optimized.

Latest Research

Exercise 5

Do your own research on the latest developments in computer chess. You may consider the following aspects:

- Historical milestones in computer chess (e.g., Deep Blue vs. Kasparov).
- Current state-of-the-art chess engines (e.g., Stockfish, AlphaZero).
- Techniques used in modern chess engines (e.g., neural networks, reinforcement learning).
- Impact of AI on human chess players and the chess community.
- Future trends and potential developments in computer chess.
- Include references to relevant articles, papers, or websites.

References

- Ertel, Wolfgang (2025). *Introduction to Artificial Intelligence*. 3rd. Springer. DOI: 10.1007/978-3-658-43102-0.
- Russell, Stuart J. and Peter Norvig (2010). *Artificial Intelligence: A Modern Approach*. 3rd. Pearson.
- Frayn, Colin (2005). *Computer Chess Programming Theory*. URL: http://www.frayn.net/beowulf/theory.html.
- Nilsson, Nils J. (1998). *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann.
- Ertel, Wolfgang (1993). "Parallele Suche mit randomisiertem Wettbewerb in Inferenzsystemen." PhD thesis. Technische Universität München.

References (cont.)

Korf, Richard E (1985). “Depth-first iterative-deepening: An optimal admissible tree search.” In: *Artificial intelligence* 27.1, pp. 97–109. DOI: 10.1016/0004-3702(85)90084-0.

Pearl, Judea (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Series in Artificial Intelligence. Addison-Wesley Publishing Company.