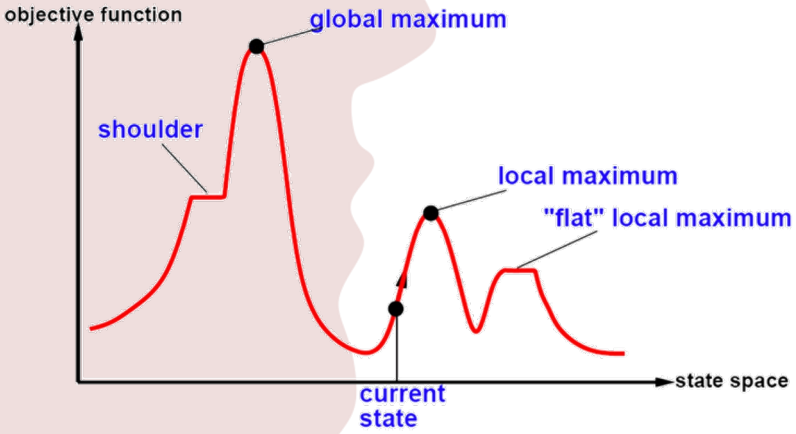
AI101

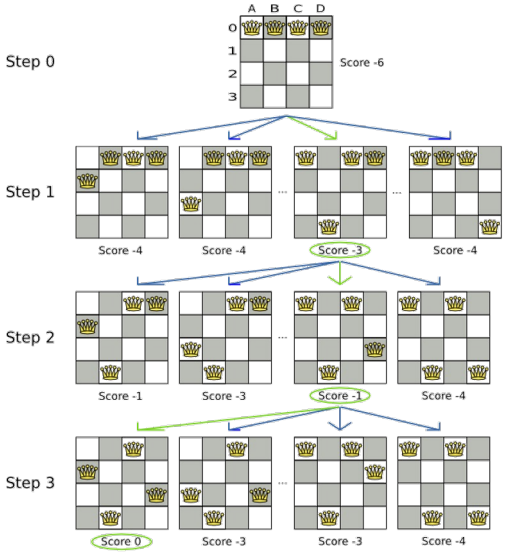
Lecture 4: Local Search and Adversarial Search

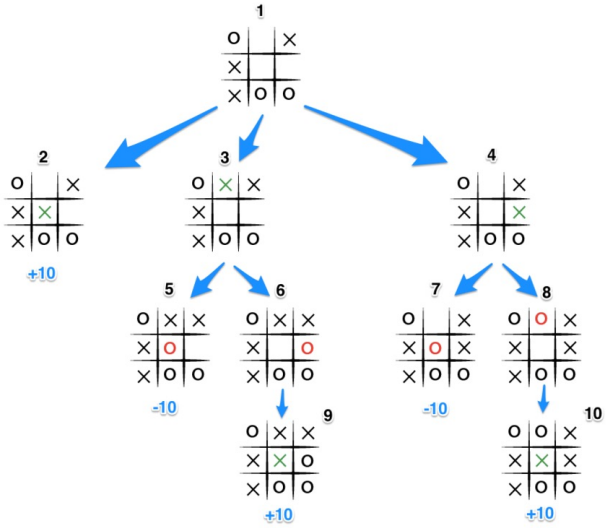
TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik

hessian.AI







Recap

Last Week: (Un)Informed Search

- Introduction into problem-solving via search algorithms
- In (Un-)informed search, nodes are expanded systematically in two ways:
  - Keeping different path in the memory
  - Selecting the best suitable path
- The solution of an (un-)informed search is an optimal sequence of actions/states leading to the goal
- Informed search uses additional knowledge in the search process
  - This knowledge is often represented in form of heuristics
- (Un-)informed search strategies are memory hungry

Outline

What are potential limits of informed uninformed search?

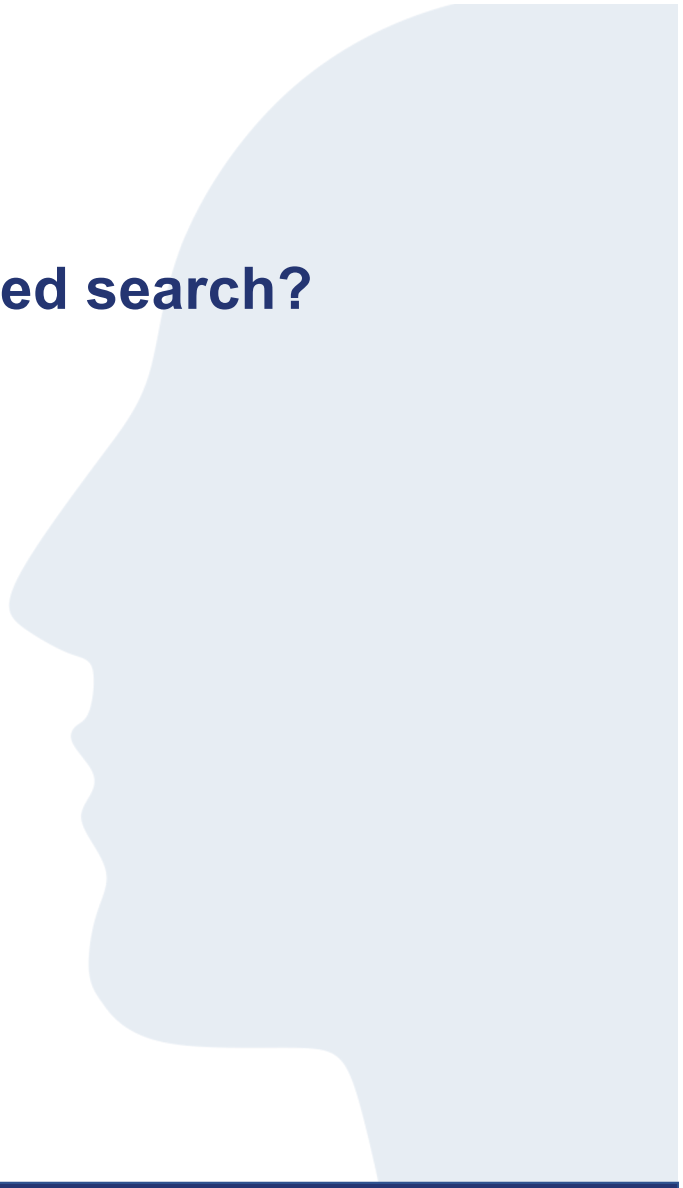
Local Search

- Hill Climbing
- Beam Search
- Simulated Annealing

Gradient Descent

Adversarial Search

- AI in Games
- Minimax
- Alpha-Beta Pruning



The Issue

So far

- Methods that systematically explore the search space, possibly using principled pruning (e.g., A*)
- Search algorithms can handle search spaces of up to 10^100 states

But

What if we have much larger search spaces?

Optimization Problems

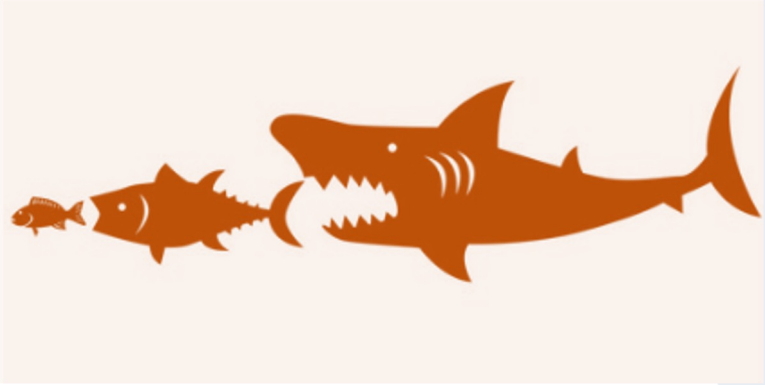
Optimization Problem

An optimization problem is one where all states/nodes can be a solution (to different degrees) but the target is to find the state that optimizes (min, max, …) the solution according to an objective function. There are no goal state and also no path costs.

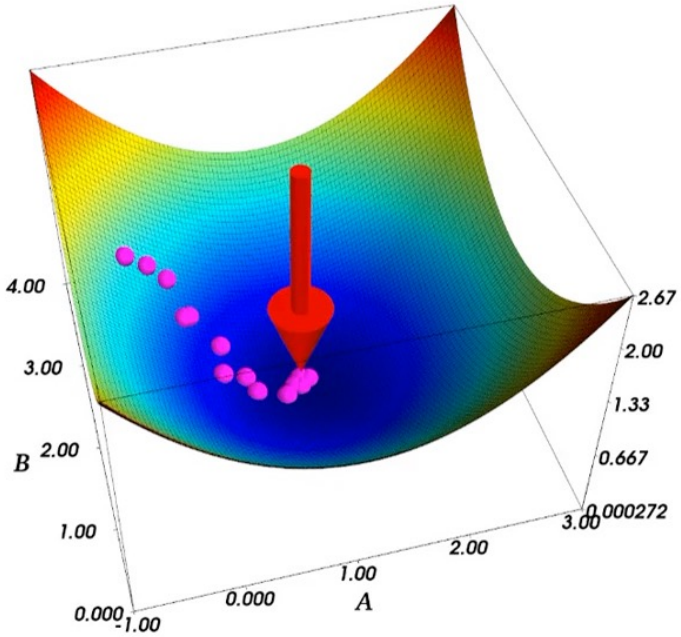
Objective/Evaluation Function

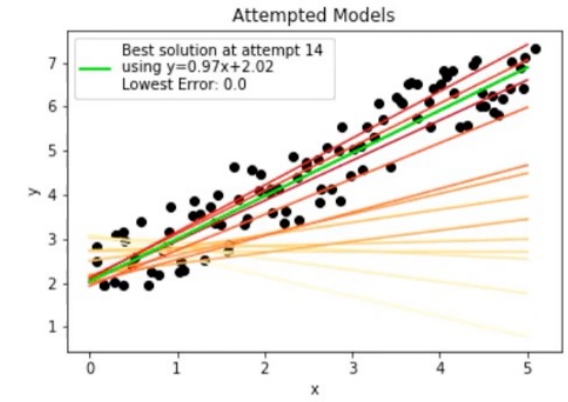
An objective function is a function whose value is either minimized or maximized depending on the optimization problem.

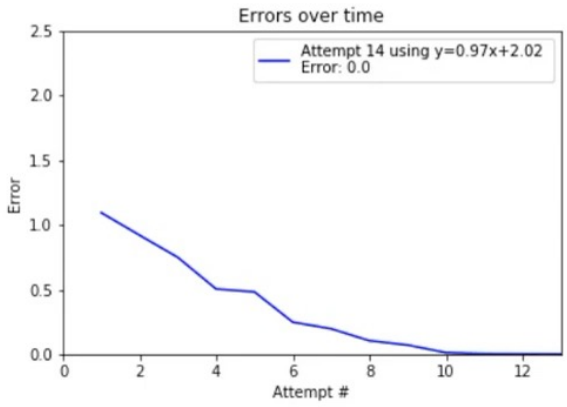
Example:
Darwinian evolution and survival of the fittest may be regarded as an optimization process



Optimization Problems







Frederik Hardervig https://www.youtube.com/watch?v=z3qOOJl-VSU

Key Terminologies

Convergence

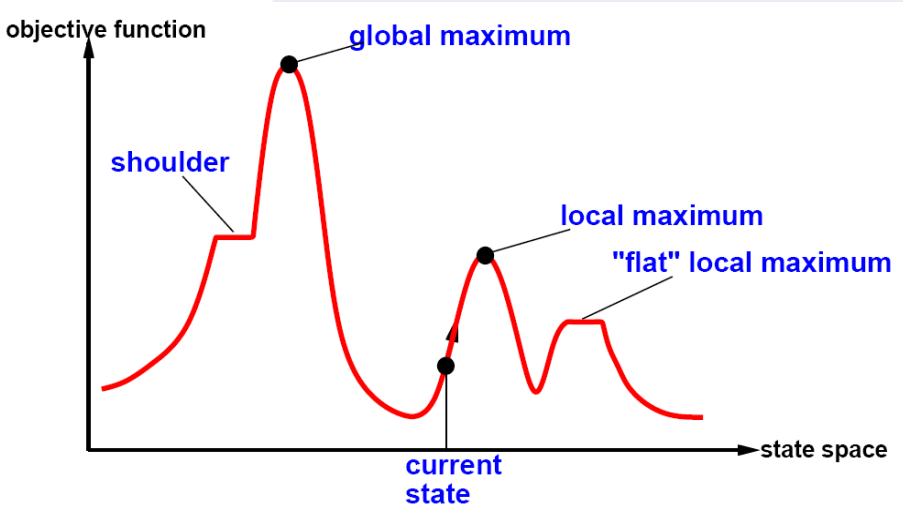
Describes the property of a sequence of (function) values to approach a limit/value more and more

Global Optimum

A global optimum is the extremum (minimum or maximum) of the objective function for the entire input search space.

Local Optimum

A local optimum is the extremum (minimum or maximum) of the objective function for a given region of the input space, e.g., a basin in a minimization problem.



Local Search Algorithms
(or Iterative Improvement Methods)

Local Search

Local search algorithms traversing only a single state rather than saving multiple paths in memory. It modifies its state iterative, trying to improve a criteria.

In many optimization problems the sequence of actions and costs are irrelevant

The goal is to find a state fulfilling all goal constraints not the path to reach it!

Advantages:

- Uses a very little/constant amount of memory
- Find a reasonable solution in very large state spaces (where systematical approaches fail)

Disadvantages:

- No guarantees for completeness or optimality

try to improve it by maximizing a heuristic evaluation using only **„local“ improvements** (i.e., only modifies the current state)

Local Search Algorithms
Again there are several algorithm design considerations

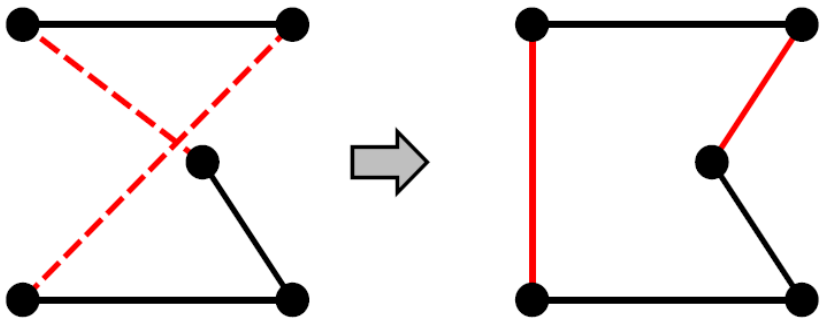
- How do you represent your problem
- What is your objective function
  - How to measure cost or value of a state
- What is a “neighbor” of a state
  - Which other states can be reached from the current state
  - How can one compute a neighbor or a step
- Are there any constraints you can exploit
- …

Local Search Algorithms

Example: Travelling Salesman Problem

Basic Idea:

- Start with a complete (but most likely suboptimal) tour
- perform pairwise exchanges



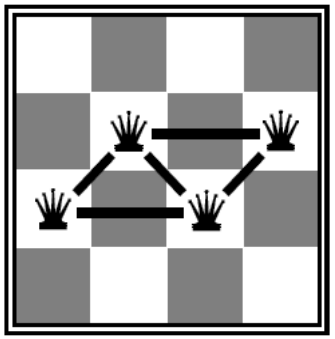
- variants of this approach get within 1% of an optimal solution very quickly with thousands of cities

Local Search Algorithms

Example: n-Queens Problem

Recall the problem: Place n Queens on a chess board so that no queen can capture any other queen (i.e. 2 queens can not be on the same line vertically, horizontally and in diagonal)

Idea: Move a queen so that it reduces the number of conflicts



h = 5

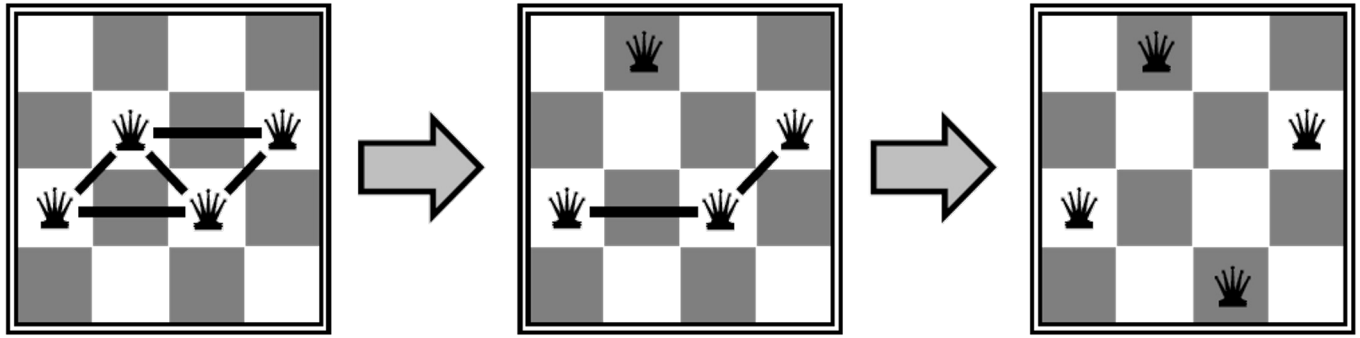
States: 4 queens in 4 columns (256 states)
Neighborhood Operators: move queen in column
Evaluation / Optimization function: h(n) = number of attacks / “conflicts”
Goal test: no attacks, i.e., h(G) = 0

Local Search Algorithms

Example: n-Queens Problem

Recall the problem: Place n Queens on a chess board so that no queen can capture any other queen (i.e. 2 queens can not be on the same line vertically, horizontally and in diagonal)

Idea: Move a queen so that it reduces the number of conflicts



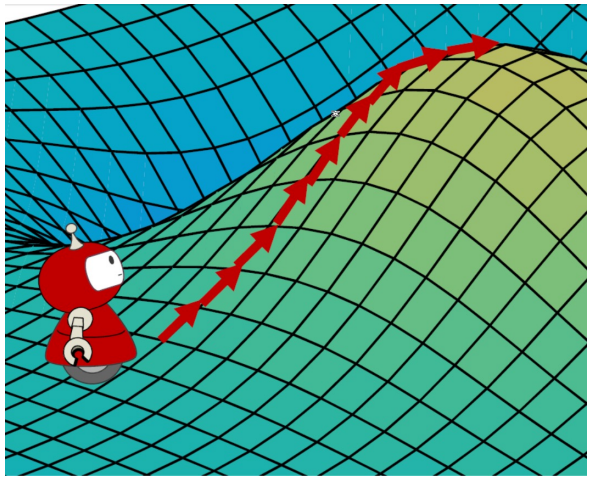
h = 5 h = 2 h = 0

Local Search: Because we only consider local changes to the state at each step. We generally make sure that series of local changes can reach all possible states.

Local Search Algorithms
Hill Climbing / Greed Local Search

Algorithm

- Expand the current state (generate all neighbours)
- Move to the one with the highest evaluation
- Do so until you reach a maximum (evaluation goes down again)



```
function HILL-CLIMBING( problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
    end
```

Local Search Algorithms
Hill Climbing / Greed Local Search

"Like climbing Mount Everest in thick fog with amnesia"

Algorithm

- Expand the current state (generate all neighbours)
- Move to the one with the highest evaluation
- Do so until you reach a maximum (evaluation goes down again)

Problem: Local Optima

- the algorithm will stop as soon as it is at the top of a hill
- But this hill does not have to be optimal (plateaus, ridges, shoulders,…)

"Like climbing Mount Everest in thick fog with amnesia„

Local Search Algorithms
Hill Climbing / Greed Local Search

"Like climbing Mount Everest in thick fog with amnesia"

Problem: Local Optima

Idea: Randomized Hill-Climbing Variants

1. Randomized Restart Hill Climbing

- Different initial positions result in different local optima
  → make several iterations with different starting positions

2. Stochastic Hill-Climbing

- select the successor node ramdomly
- better nodes have a higher probability of being selected

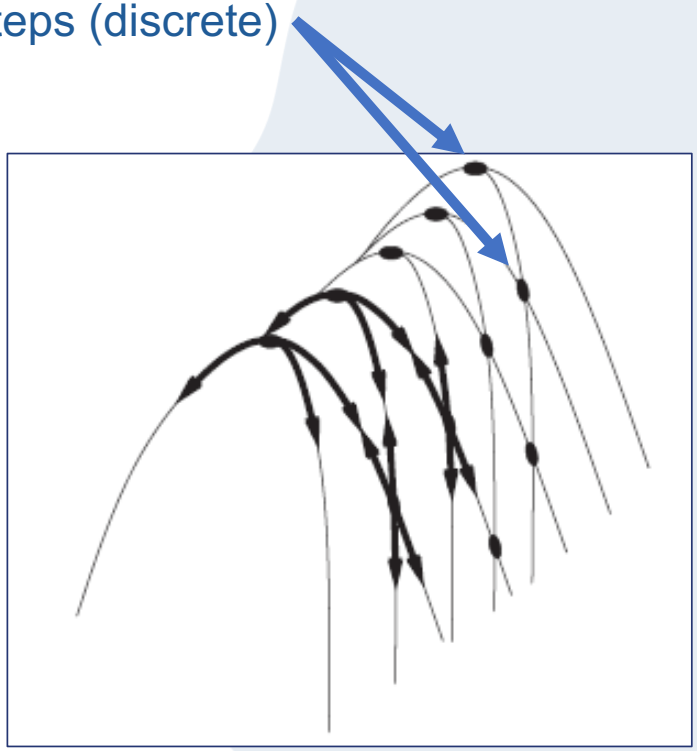
Image generated with Stable Diffusion using the prompt “Like climbing Mount Everest in thick fog with amnesia“

Optimization Problems
Ridge Problem/Issue

Problem:

Every neighbor state appears to be downhill.

- The search space has an uphill
- The neighbors not

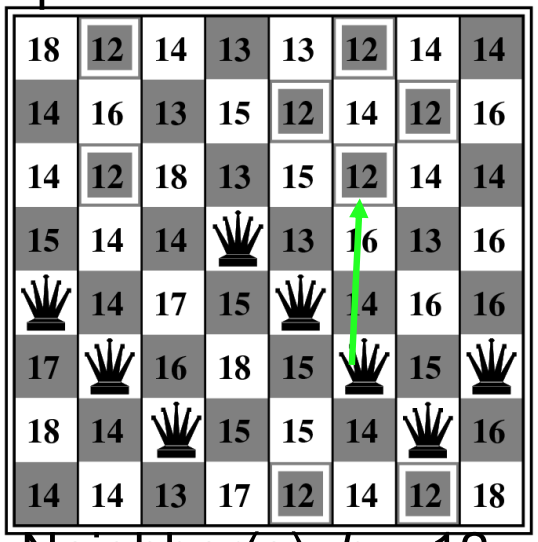


Ridge

Fold a piece of paper and hold it tilted up at an unfavorable angle to every possible search space step. Every step leads downhill; but the ridge leads uphill.

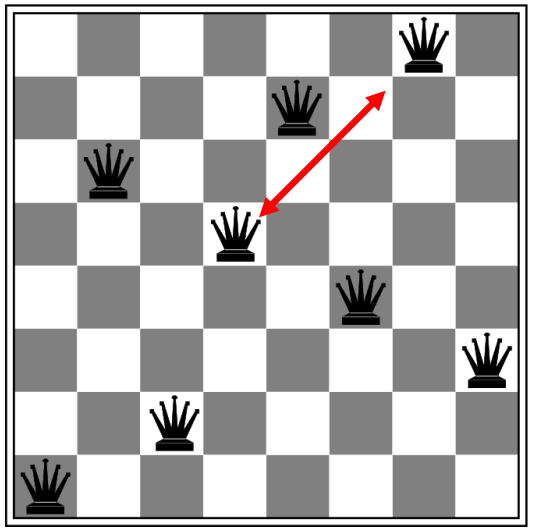
Example: 8-Queens Problem

- Heuristic *h*:
  - number of pairs of queens that attack each other, if we move a queen in ist column to that square
- Example state: *h* = 17



- Best Neighbor(s): *h* = 12

- Local optimum with $h = 1$



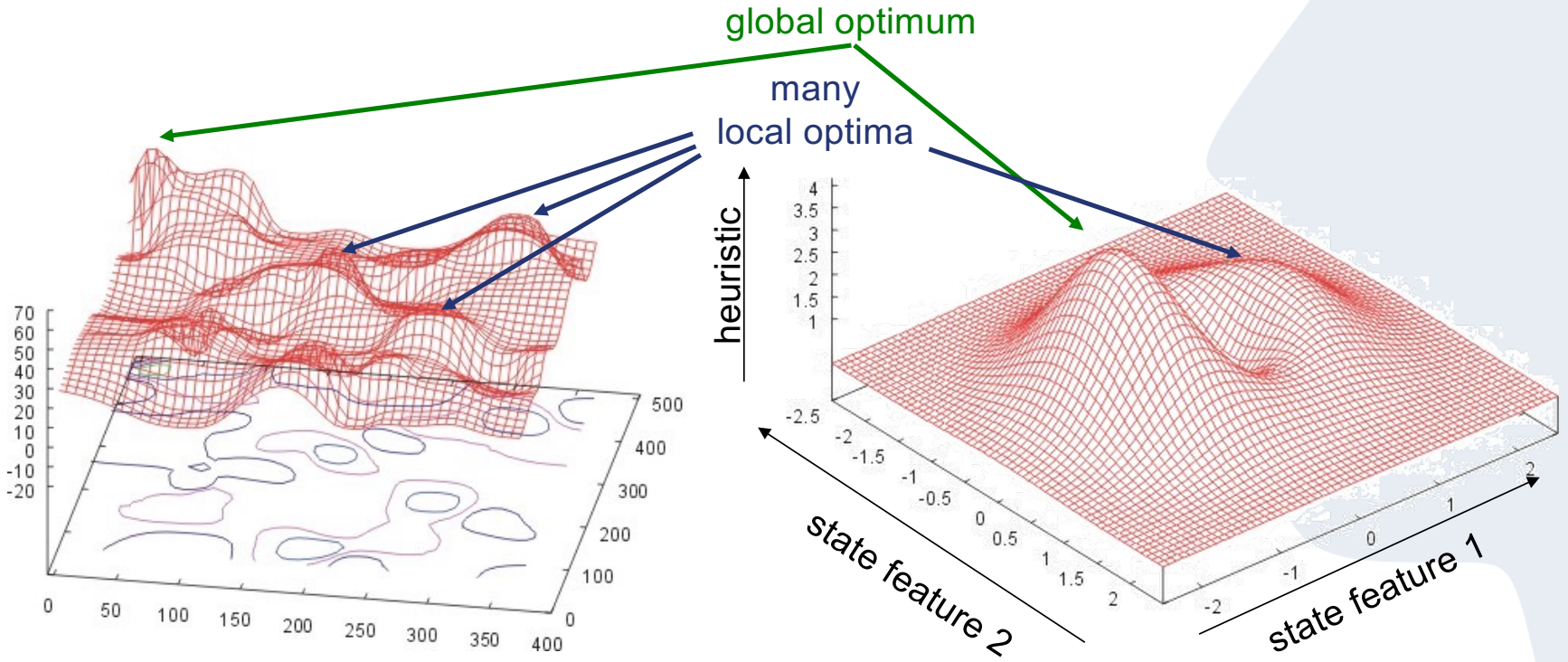
  - no queen can move without increasing the number of attacked pairs
  - What can you do to get out of this local minimum?

Optimization Problems

Multi-Dimensional State-Landscape (as we somehow have already seen)

State-Landscape: States are presented as locations with the heuristic value as evelation

Problem: Finding the global optimum (local search) becomes worse with more dimensions



Gradient Descent

The "landscape view allows one to use methods from mathematical analysis

Gradient

A gradient is a derivative of a function that has more than one input variable. Known as the slope of a function in mathematical terms, the gradient simply measures the change in all weights with regard to the change in error.

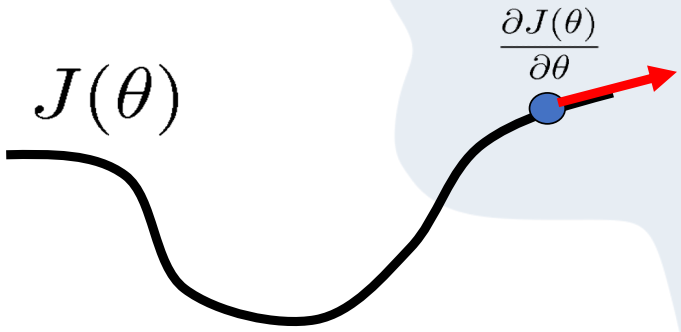
Gradient Descent

Gradient Descent is a very popular optimization strategy. It can be described as Hill-climbing in a continuous state space.

Analogy: Instead of climbing up a hill, think of gradient descent as hiking down to the bottom of a valley

Goal: Minimize cost-function $J(\theta)$

Derivative $\frac{\partial J(\theta)}{\partial \theta}$

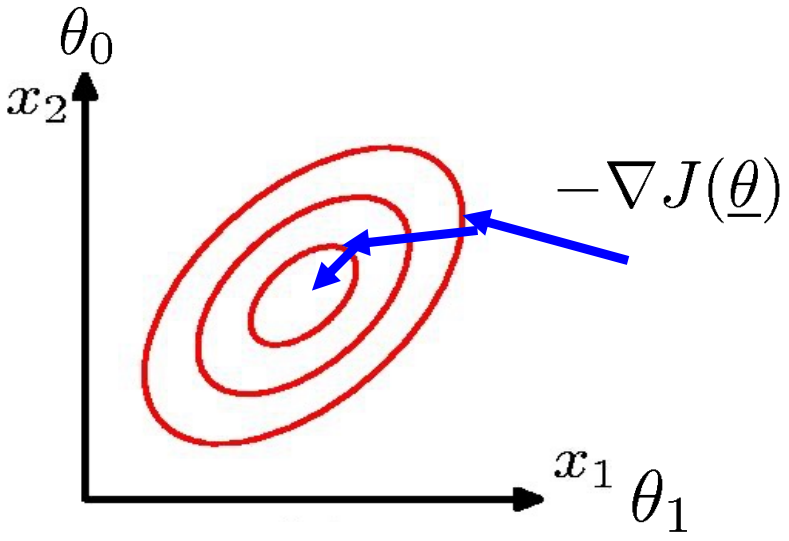


Gradient Descent

Hill-climbing in continuous spaces

Gradient vector $\nabla J(\underline{\theta}) = \left[\frac{\partial J(\underline{\theta})}{\partial \theta_0} \quad \frac{\partial J(\underline{\theta})}{\partial \theta_1} \quad \dots \right]$

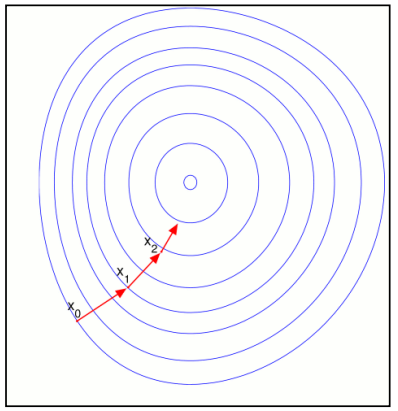
Analogy: Think of a gradient in this context as a vector that contains the direction of the steepest step you can take and also how long that step should be.



Gradient Descent

Hill-climbing in continuous spaces

Gradient vector: $\nabla J(\underline{\theta}) = \left[\frac{\partial J(\underline{\theta})}{\partial \theta_0} \quad \frac{\partial J(\underline{\theta})}{\partial \theta_1} \quad \ldots \right]$



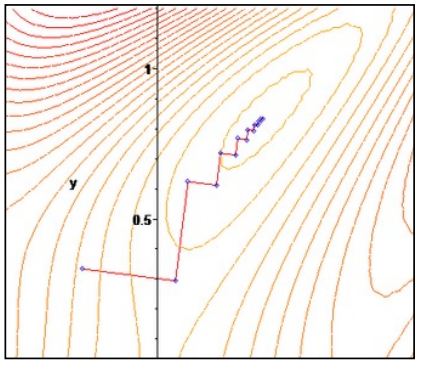
Assume we have some cost-function: $J(x_1, x_2, \ldots, x_n)$

and we want minimize over continuous variables: $x_1, x_2, .., x_n$

1. Compute the gradient : $\frac{\partial}{\partial x_i} J(x_1, \ldots, x_n) \qquad \forall i$
2. Take a small step downhill in the direction of the gradient:

   $$x_i' = x_i - \lambda \frac{\partial}{\partial x_i} J(x_1, \ldots, x_n)$$

3. Check if $J(x_1', \ldots, x_n') < J(x_1, \ldots, x_n)$
4. If true then accept move, if not “reject”.
5. Repeat.



Gradient Descent
The Importance of the Learning Rate

Learning Rate

The learning rate is a hyperparameter, controlling how quickly the model is adapted to the problem.

Analogy: How big should we choose the steps the gradient descent takes ?

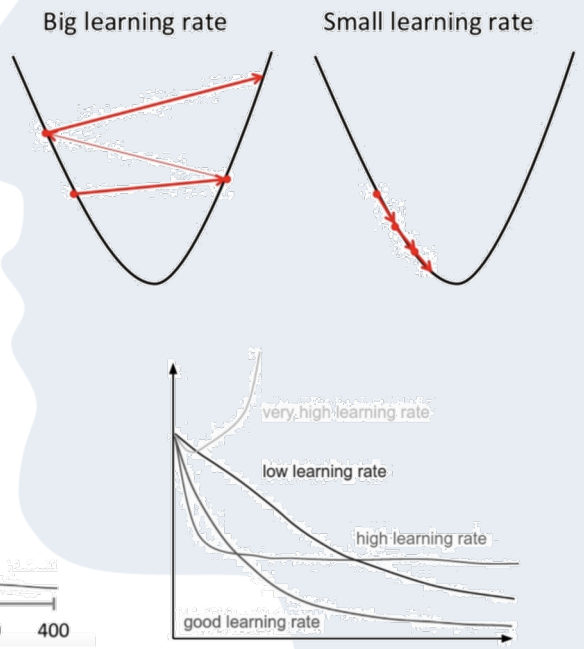
Finding the right Learning Rate:

Smaller learning rates

- Smaller changes, require more training epochs

Larger learning rates

- More rapid changes, needing fewer epochs
- Can converge to a local optima or not at all



Gradient Descent

How do I determine the gradient?

- Derive formula using multivariate calculus.
- Ask a mathematician or a domain expert.
- Do a literature search.
- Automatic Differentiation

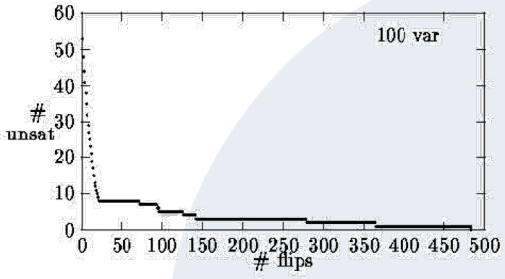
Variations of gradient descent can improve performance for this or that special case.

- Simulated Annealing, Linear Programming too

Works well in smooth spaces; poorly in rough.

Another example: Greedy SAT

- Task: Find a satifying configuration I of a propositional formula **Δ**
- Hill-climbing for SAT in a discrete space



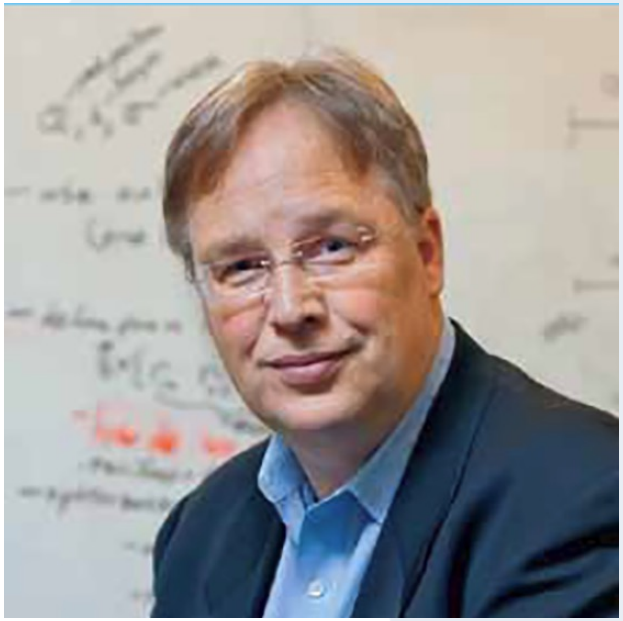
auxiliary functions:

- violated(Δ, *I*): number of clauses in Δ not satisfied by *I*
- flip(*I*, *v*): assignment that results from *I* when changing the valuation of proposition *v*

function GSAT(Δ):

repeat *max-tries* **times**:
    I := a random assignment
    repeat *max-flips* **times**:
        if *I* ⊨ Δ:
            return *I*
        V_{greedy} := the set of variables *v* occurring in Δ
                for which violated(Δ, flip(*I*, *v*)) is minimal
        randomly select $v \in V_{\text{greedy}}$
        I := flip(*I*, *v*)
return no solution found

Bart Selman (Cornell CS)



Satisfiablity (SAT) Problem

SAT Problem: given a propositional formula F in CNF, return a model (solution) to F or prove that none exists

Example: $F = (a + b)\ (a + \neg c)\ (\neg a + c)$

F has 3 models:

- $M_1 = \{a=0, b=1, c=0\}$
- $M_2 = \{a=1, b=0, c=1\}$
- $M_3 = \{a=1, b=1, c=1\}$

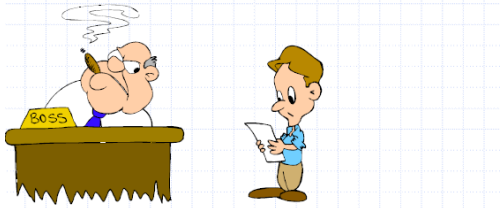
A SAT solver will return either of the 3 models

Example: $F = (a)\ (\neg a)$

F is not satisfiable

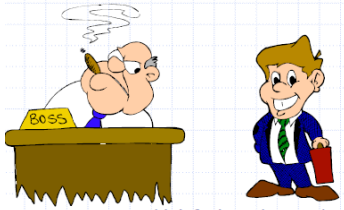
SAT has myriads of applications

◆What to do when we find a problem that looks hard...



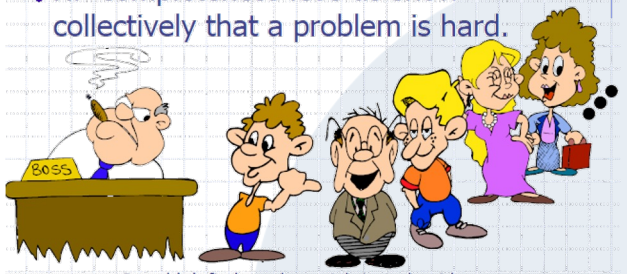
I couldn't find a polynomial-time algorithm;
I guess I'm too dumb.

◆Sometimes we can prove a strong lower bound... (but not usually)



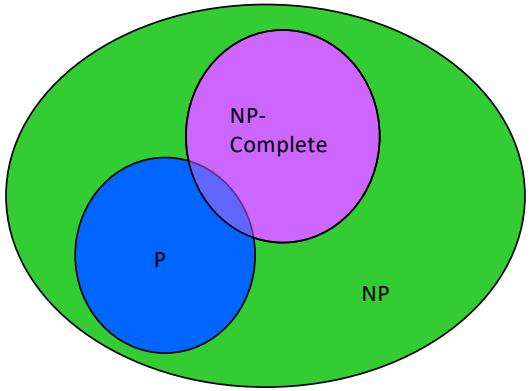
I couldn't find a polynomial-time algorithm,
because no such algorithm exists!

◆NP-completeness let's us show collectively that a problem is hard.



I couldn't find a polynomial-time algorithm,
but neither could all these other smart people.

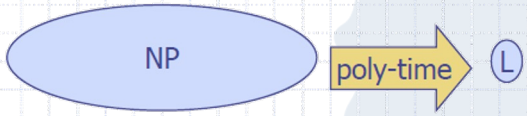
- P = { L | L is accepted by a deterministic Turing Machine in polynomial time }
- NP = { L | L is accepted by a non-deterministic Turing Machine in polynomial time }

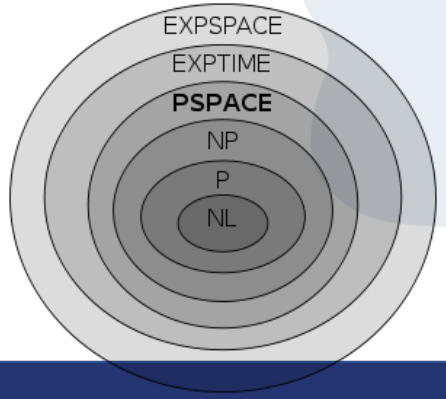


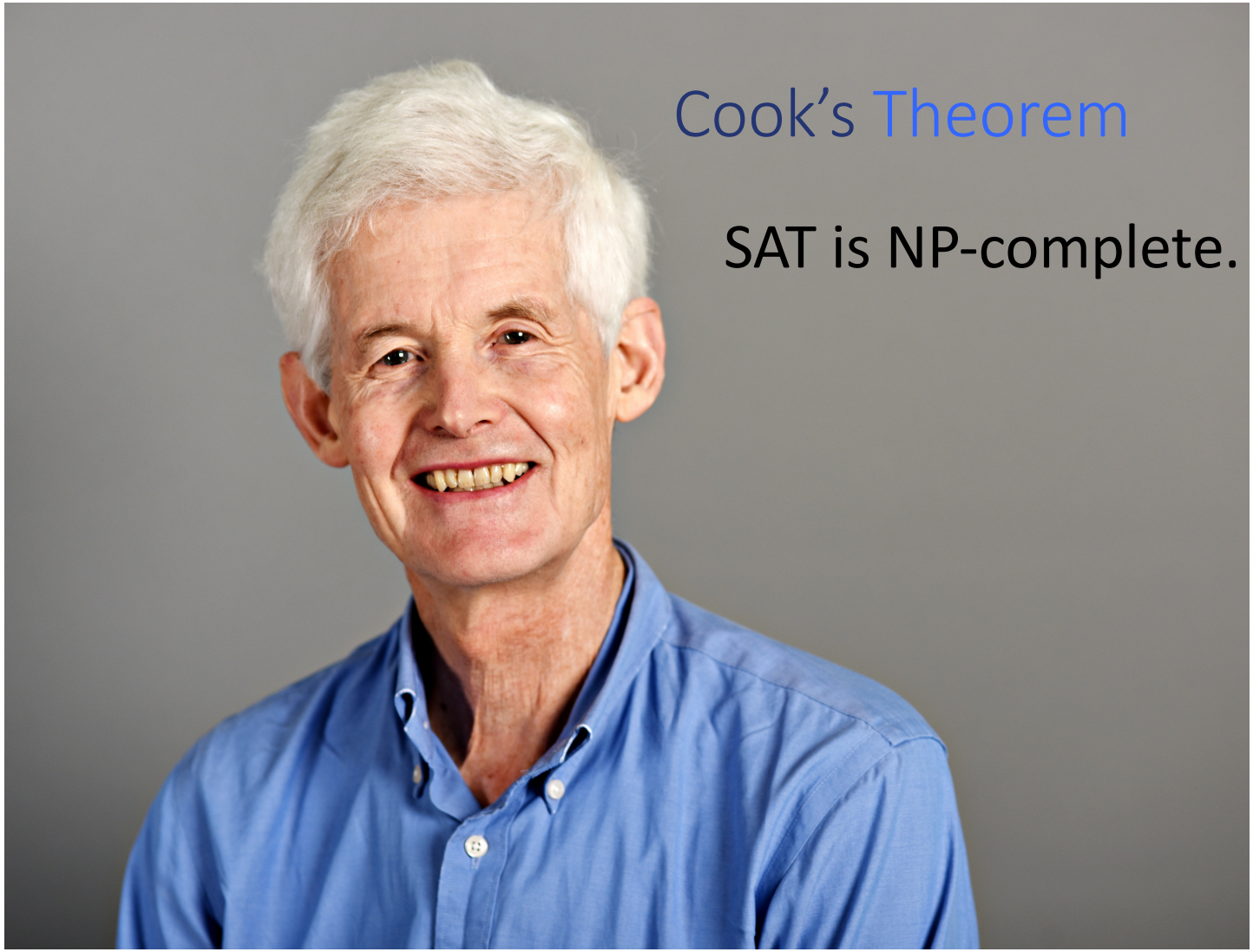
◆A problem (language) L is **NP-hard** if every problem in NP can be reduced to L in polynomial time.

◆That is, for each language M in NP, we can take an input x for M, **transform** it in polynomial time to an input x' for L such that x is in M if and only if x' is in L.

◆L is **NP-complete** if it's in NP and is NP-hard.







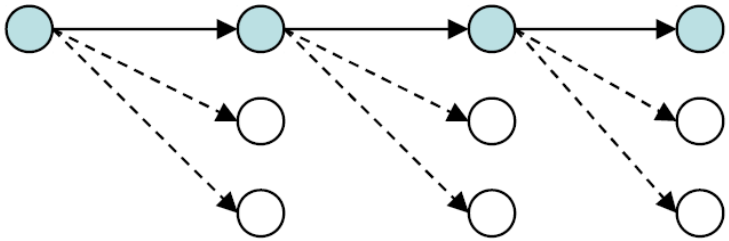
Stephen A. Cook became famous in theoretical computer science for his theorem: "SAT is NP-complete". In 1982 he received the Turing Award for this discovery.

Local Search Algorithms
Beam Search

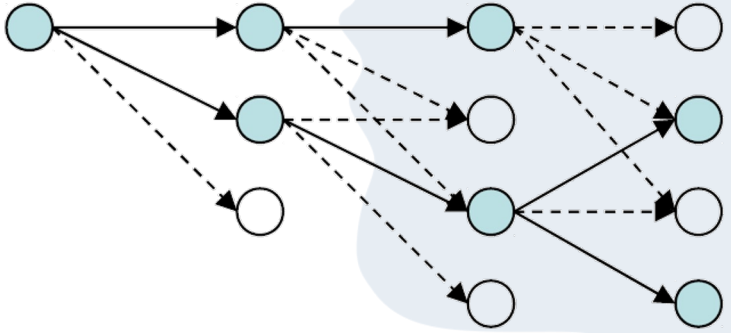
Idea: Keep track of k states rather than just one (k is calles beam size)

Algorithm

- Start with k randomly generated states
- At each iteration, all the successors of all k states are generated
- Select the k best successors from the complete list and repeat



Hill-Climbing Search



Beam Search ($k = 2$)

Local Search Algorithms
Simulated Annealing

Idea:

Use conventional hill-climbing style techniques, but occasionally take a step in a direction other than that in which there is improvement (downhill moves; away from solution).

As time passes, the probability that a down-hill step is taken is gradually reduced and the size of any down-hill step taken is decreased.

Can be seen of hill-climbing and random walk

- Allow some “bad moves” to escape local optima

Analogy: Annealing in metallurgy

Local Search Algorithms
Simulated Annealing

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] – VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability e^(Δ E/T)
```

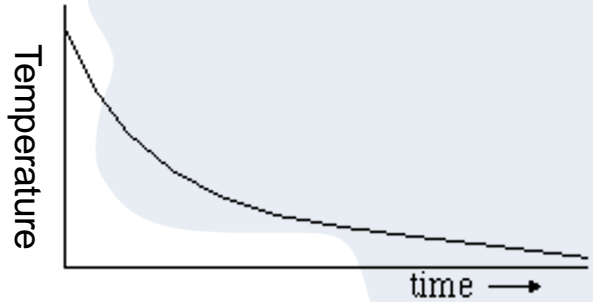
Local Search Algorithms
Simulated Annealing

Temperature

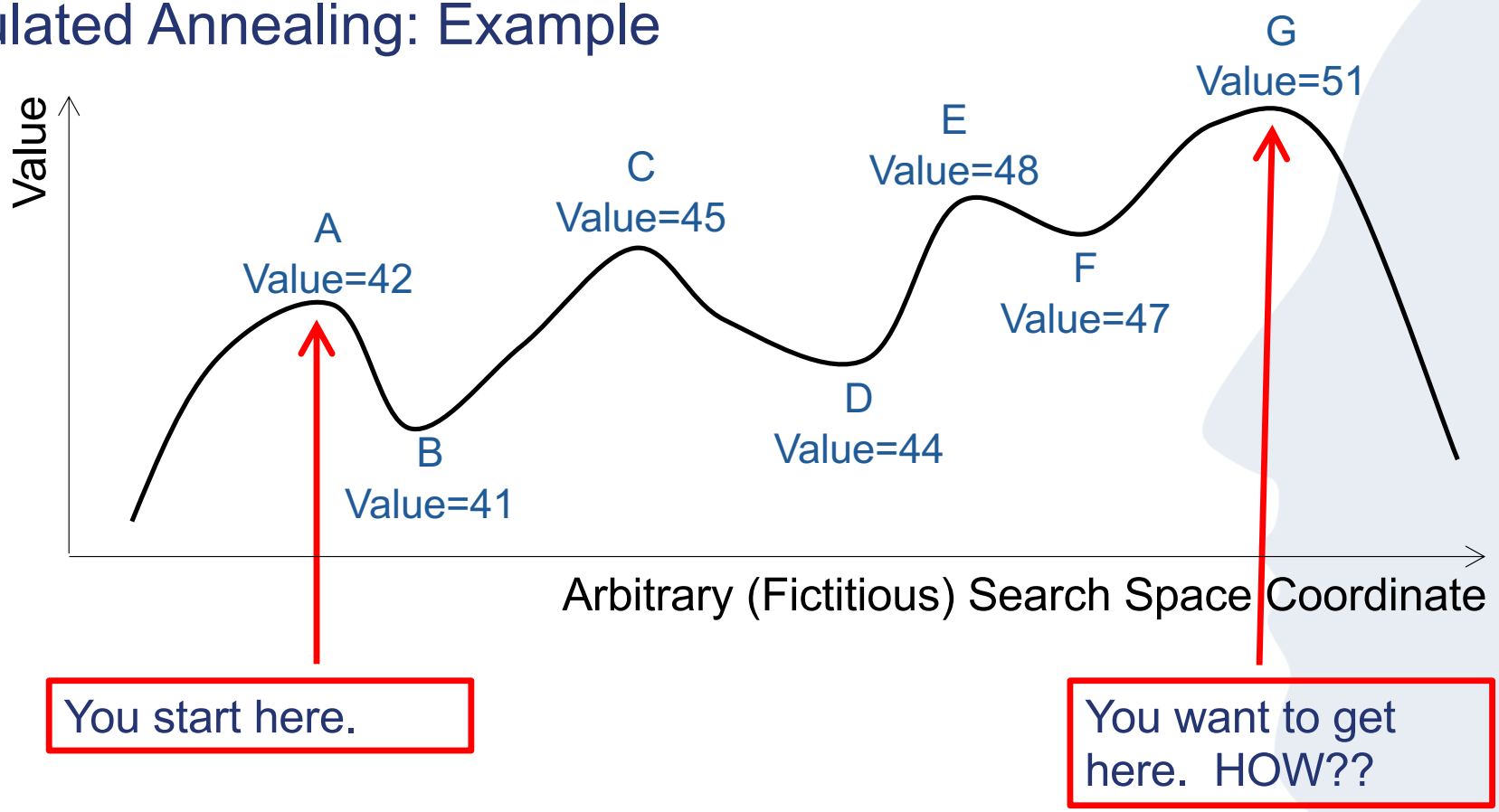
The temperature is a hyperparameter, controlling how frequently we allow “bad moves” to escape local optima. Usually the temperature decays **exponential** over the process

Effectiveness:

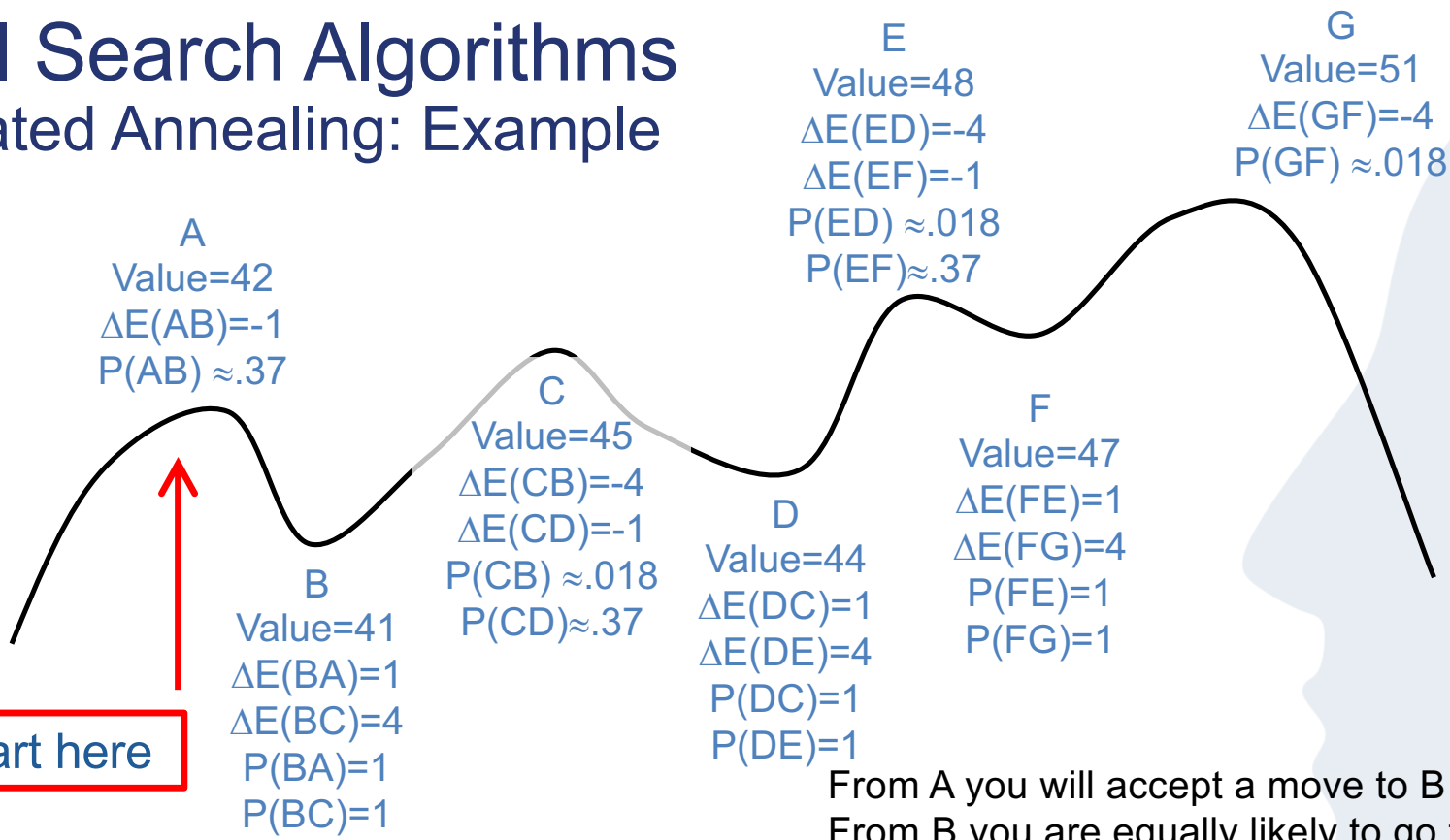
- If lowered slowly enough → converges to a global optimum
  (In any finite search space, random guessing also will find a global optimum → weak claim)
- Unfortunately this can take a VERY VERY long time



Local Search Algorithms
Simulated Annealing: Example



Local Search Algorithms
Simulated Annealing: Example



x	-1	-4
e^x	≈.37	≈.018

From A you will accept a move to B with P(AB) ≈.37.
From B you are equally likely to go to A or to C.
From C you are ≈20X more likely to go to D than to B.
From D you are equally likely to go to C or to E.
From E you are ≈20X more likely to go to F than to D.
From F you are equally likely to go to E or to G.
Remember best point you ever found (G or neighbor?).

Local Search Algorithms
Simulated Annealing: Conclusion

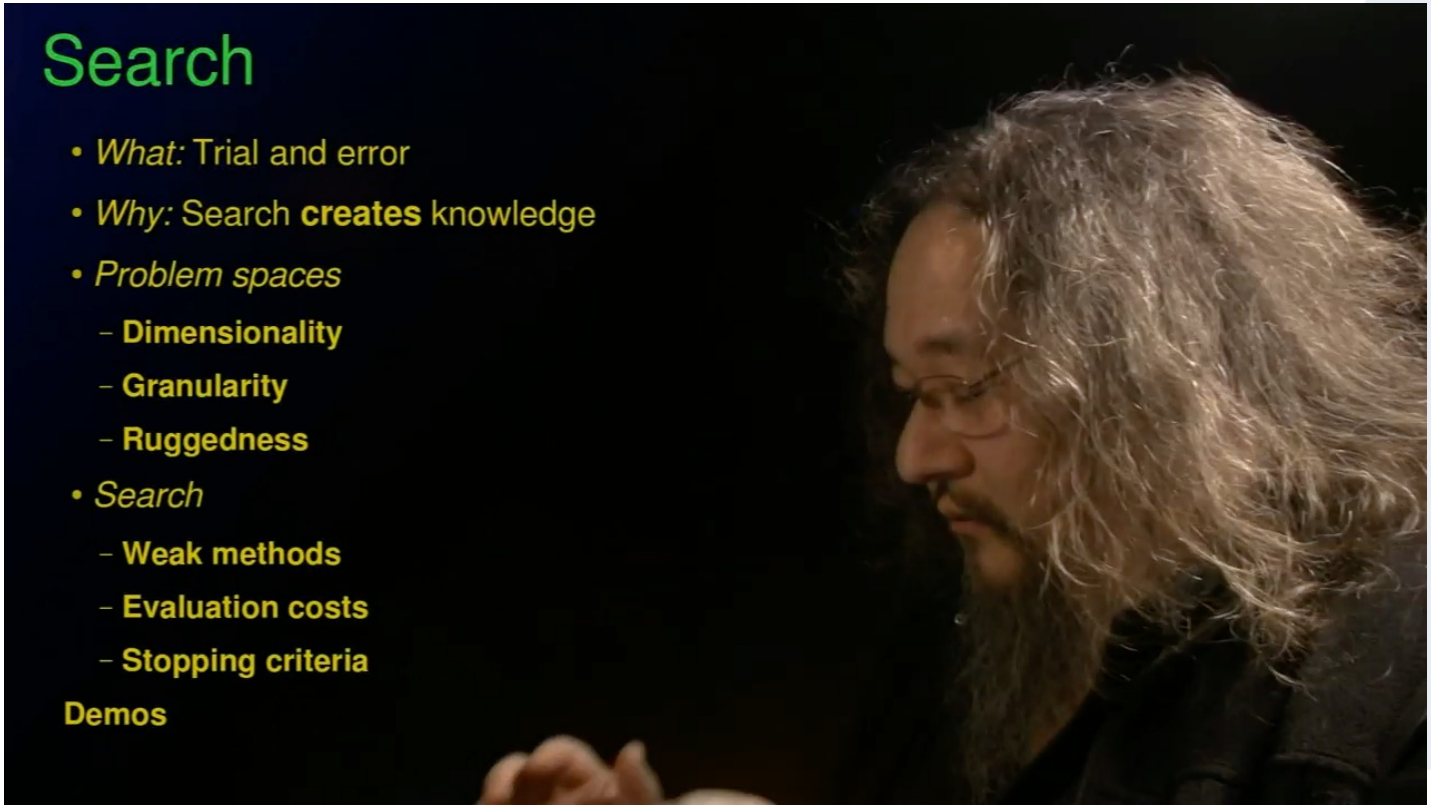
Superficially

- Simulated Annealing is local search with some noise added. Noise starts high and is slowly decreased.
- It models the way physical systems, such as gases, sample from their statistical equilibrium distributions. Order 10^23 particles. Studied in the field of statistical physics.

True story is much more principled:

- Simulated Annealing is a general sampling strategy to sample from a (combinatorial) space according to a well-defined probability distribution.
- The temperature has a strong effect on the quality of the Simulated Annealing process
- It's very general technique to sample from complex probability distributions by making local moves only. For optimization, we chose a clever probability distribution that concentrates on the optimum states for low temperature. *(Kirkpatrick et al. 1984)*

Local Search Algorithms
Illustration (in particular of steepest descent)



Dave Ackley, New Mexico CS for all, UNM CS https://www.youtube.com/watch?v=boTeFM-CVFw

AI and Games

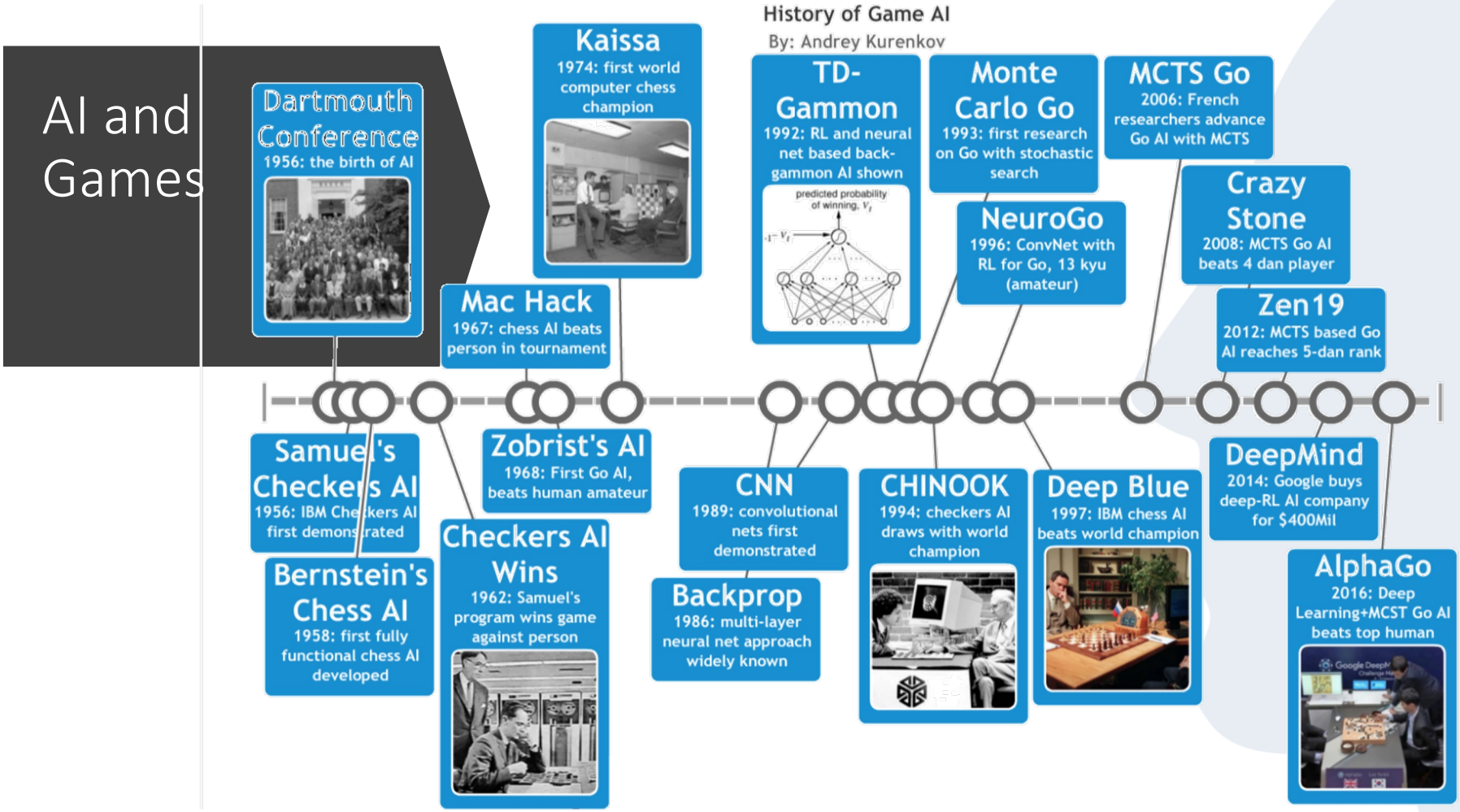


Image taken from https://www.andreykurenkov.com/writing/ai/a-brief-history-of-game-ai/

Adversarial Search
Is playing games just search?

Adversarial Search

Adversarial search is a search, where one examines the problem that arises when we try to plan ahead of the world and other players are planning against us or has conflicting goals to ours while sharing the same search space.

- Adversarial Search is used to model games as search problems
  - Here games are presented as game trees
- Each player has to consider the actions of the other players and the effect of their actions on their performance
- Often used in combination with a time limit → unlikely to find goal

Image taken from https://www.nytimes.com/2019/05/30/science/deep-mind-artificial-intelligence.html

Key Terminologies (2)
Recap

Environment

A optimization problem is one where all states/nodes can give a solution but the target is to find the stat that optimizes the solution according to an objective function.

Perfect Information vs. Imperfect Information

An objective function is a function whose value is either minimized or maximized depending on the optimization problem.

Deterministic vs Non-deterministic

An objective function is a function whose value is either minimized or maximized depending on the optimization problem.

Zero-Sum Games

An objective function is a function whose value is either minimized or maximized depending on the optimization problem.

Adversarial Search
Formalization of the Problem

A game can be defined as a type of search problem:

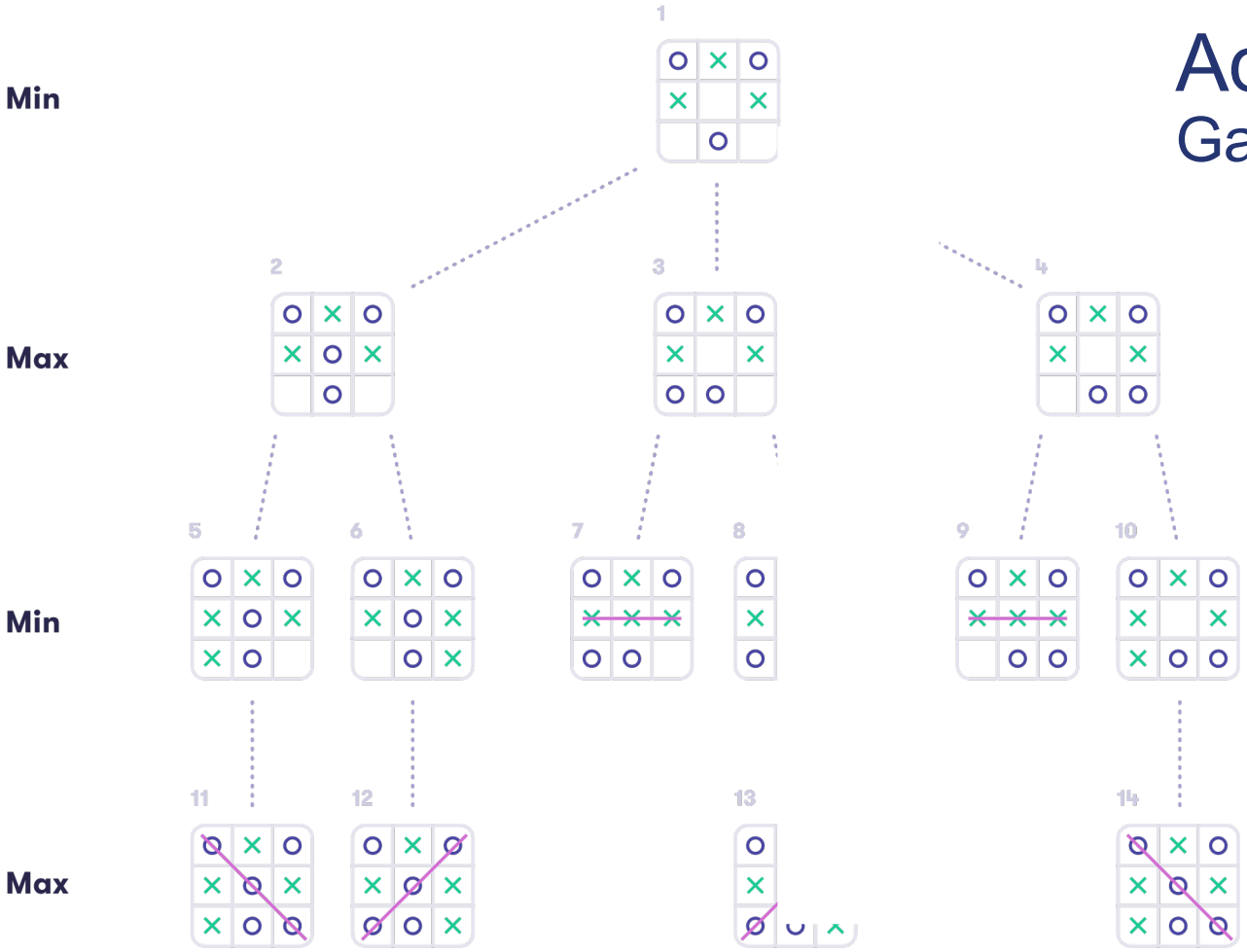
1. **Initial State**: Specifies how the game is set up at the start
2. **Player(*s*)**: Specifies which players turn it is
3. **Action(*s*)**: Returns a set of legal moves in the state *s*
4. **Result(*s*, *a*)**: Transition model, specifies the resulting state *s'* doing move *a* in state *s*
5. **Terminal(*s*)**: Tests if state *s* fulfills the goal/terminal constraints
6. **Utility(*s*, *p*)**: The utility function returns a numeric value for a terminal state *s* from the perspective of player *p*

Adversarial Search
Game Trees

- To solve games, we build so called game trees
- Different to search trees, game trees are arranged in levels that correspond to players
    - The root node is always the active/current player
- In game trees, leaf nodes are called terminal
- Each terminal node has a utility value corresponding to the outcome of the game

Adversarial Search
Tic-Tac-Toe

Adversarial Search
Game Tree: Tic-tac-toe



This is a tic-tac-toe game. The tree is supposedly small.
Fewer than 9! = 362,880 terminal nodes. In chess, this number is 10^{40}.

Adversarial Search Algorithm

Why do we teach AI to play games?

Games…

1. …are a useful metric
   - Allows us to track how much we improve over time
   - Are quantifiable (unlike the real world)
2. …provide a safe place to train and evolve
   - Games create low-stakes environments
   - Its saver then training in a real world situation
     - e.g. self-driving cars
3. …captures interest and imagination
   - In some ways it's a way to showcase advancements
   - Opens advancements to the public
     - See AlphaGo vs. Lee Sedol

Has its own Wikipedia article: https://en.wikipedia.org/wiki/Artificial_intelligence_in_video_games , Image: Demis Hassabis und Lee Sedol (Source: Deepmind)

Adversarial Search Algorithm
Games vs Search Problems?

1. "Unpredictable" opponent
    - Specify a move for every possible reply
    - Different goals per agent (my interest vs your interest)
2. Time limits
    - Unlikely to have enough time to find a goal
    - Needs approximation
3. Most games are…
    - Deterministic, turn-taking, two-player, zero-sum
4. But most “real” problems are
    - Stochastic, parallel, multi-agent, utility based

Adversarial Search
Minimax Algorithm

Idea: Build a game tree- where the nodes represent the states of the game and edges represent the moves made by the players in the game. The player are…

- **MIN:** Decrease the chances of **MAX** to win the game. (Opponent)
- **Max:** Increases his chances of winning the game.

Both play the game alternating, i.e., turn by turn and following the above strategy

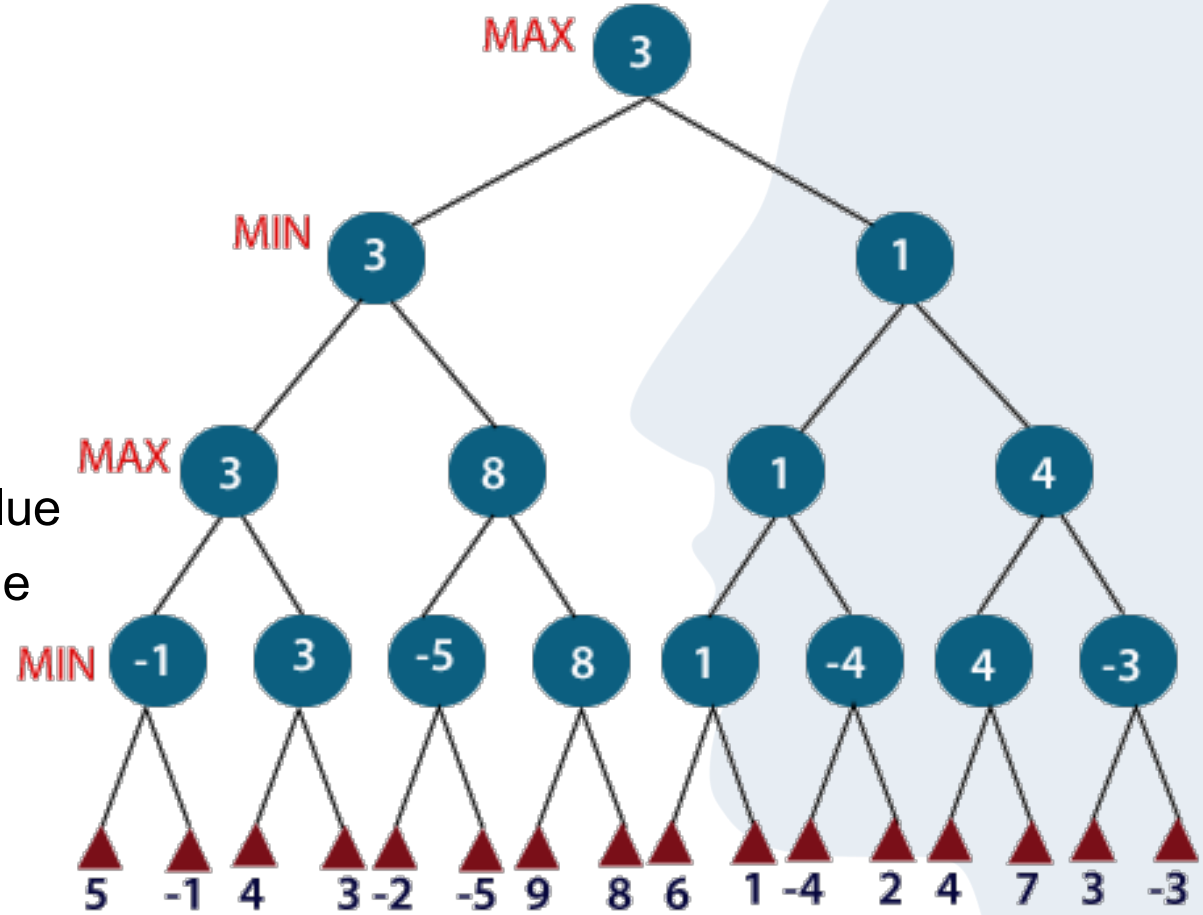
- Choose move to position with highest minimax value
    - Assuming opponent to play the best response to your action

Adversarial Search
Minimax Algorithm

https://youtu.be/trKjYdBASyQ?t=62

Adversarial Search
Minimax Algorithm

- MINIMAX strategy follows the DFS (Depth-first search) concept
- Each level is either MIN or MAX
- You play as MAX, opponent is MIN
- On a MAX level we maximize the value
- On a MIN level we minimize the value



Adversarial Search
Minimax Algorithm

Completeness

- Yes, if the tree is finite

Time Complexity

$O(b^m)$

Space Complexity

$O(b \cdot m)$

Optimal

- Yes, assuming an optimal opponent

(For chess b≈35, m≈100)

Minimax algorithm

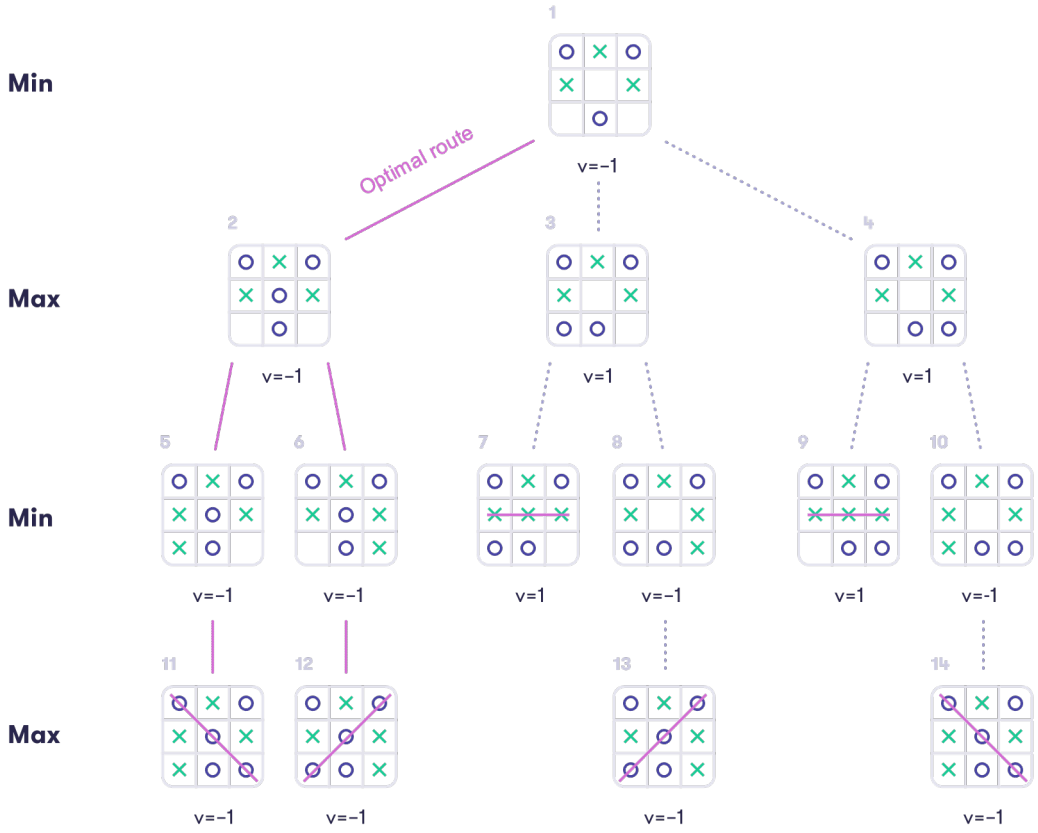
```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

Adversarial Search

Minimax Algorithm Example for Tic-Tac-Toe



Adversarial Search
The problem of massive game trees

Problem: In many games, the game tree is simply way too big to traverse in full.

- E.g. in chess ($b \approx 35$) the tree grows like 1, 35, 1225, 42875, 1500625,… nodes.
  In Go we have $b \approx 250$.

"There are more possible Go positions than there are atoms in the universe."
Demis Hassabis

Image: NASA

Adversarial Search
The problem of massive game trees

Heuristic evaluation functions

- Instead of traversing the full tree, we limit the depth.
- For each leaf node we calculate a value for the current situation based on heuristics
    - Estimation of the likely outcome of the game if we continue from here

Pruning

- Means to cut back the tree
- Ignoring unwanted portions of a search tree which make no difference to its final result
- E.g. Alpha-Beta Pruning

- (or even just randomly select few subtrees only: stochastic search)

Adversarial Search
Alpha-Beta Pruning

Alphe-Beta Pruning

A modified, optimized version of the Minimax algorithm. It uses a technique called pruning to reduce the amount of exploration without losing the correctness of Minimax.

Alpha-Beta Pruning is based on two parameters:

- **Alpha:** The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is **-∞**.
- **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is **+∞**.

Idea

Remove all the nodes which are not really affecting the final decision but making algorithm slow, hence by pruning these nodes.

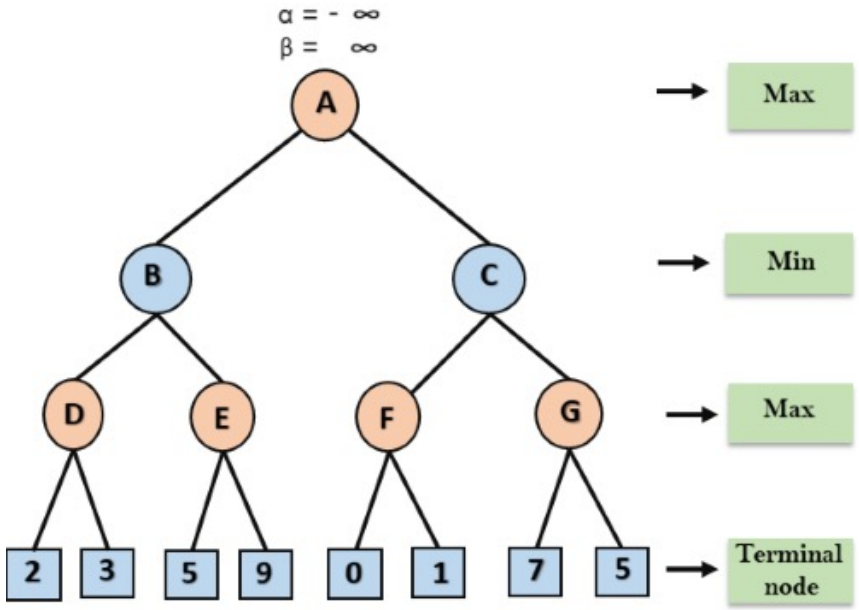
Adversarial Search
Alpha-Beta Pruning

Key differences to Minimax

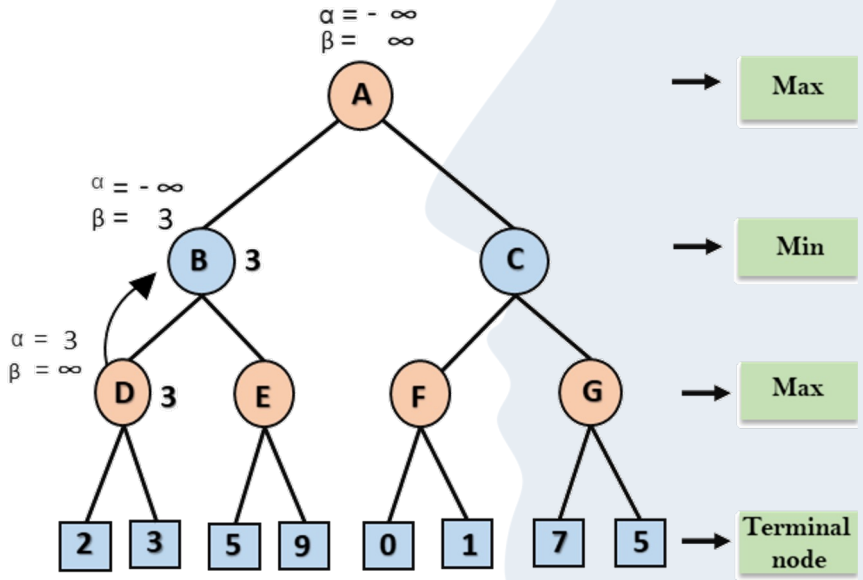
- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

Adversarial Search
Alpha-Beta Pruning

Step 1



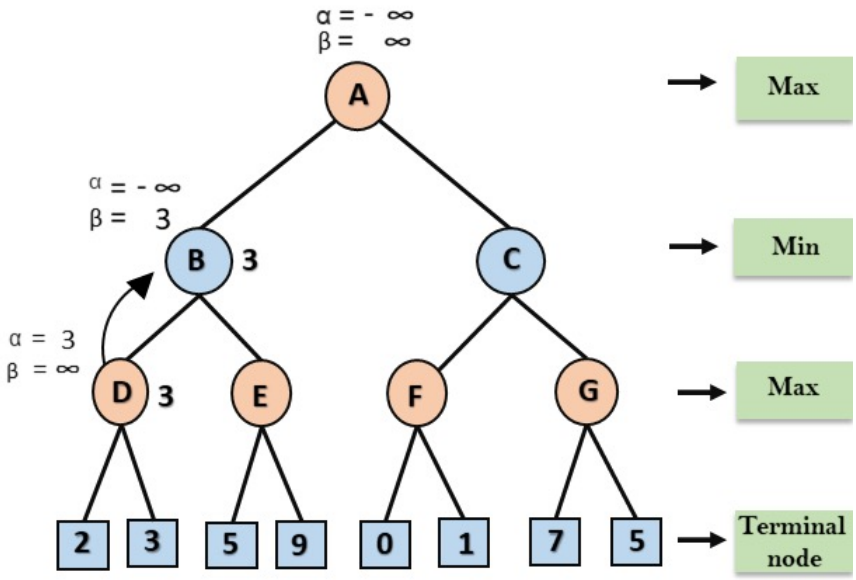
Step 2



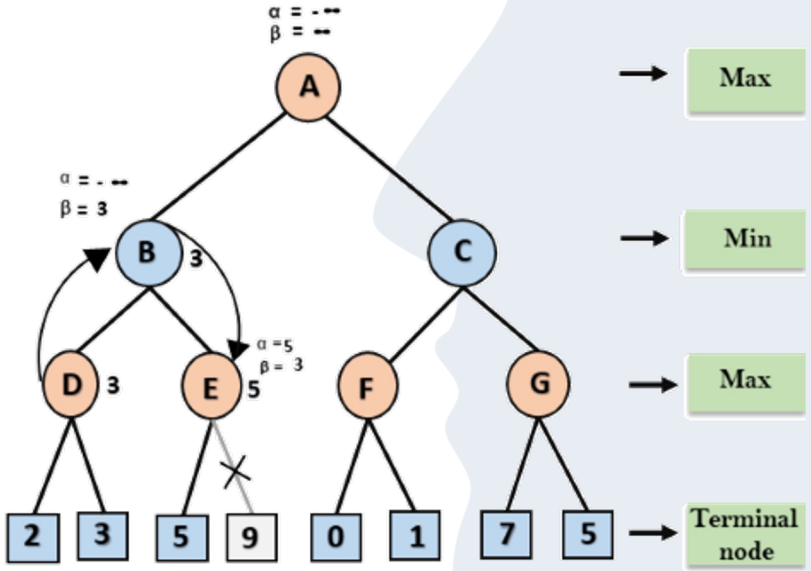
Images by https://www.javatpoint.com/ai-alpha-beta-pruning

Adversarial Search
Alpha-Beta Pruning

Step 2



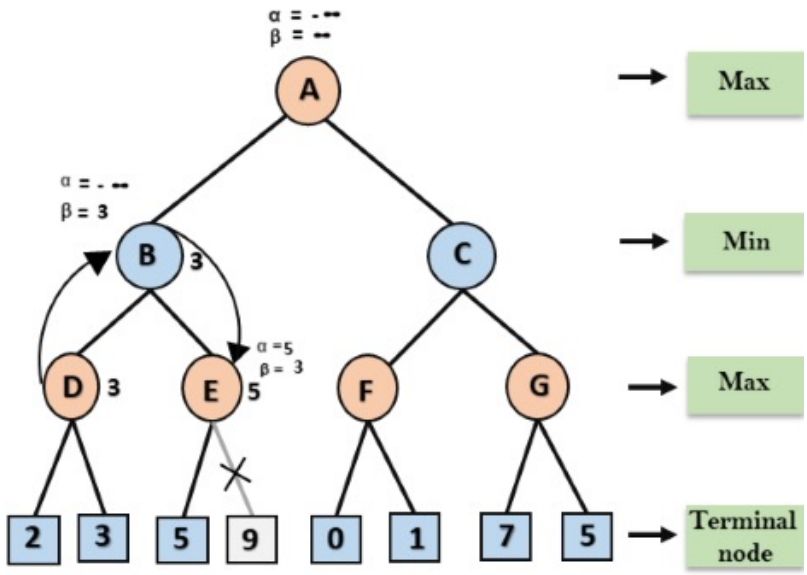
Step 3



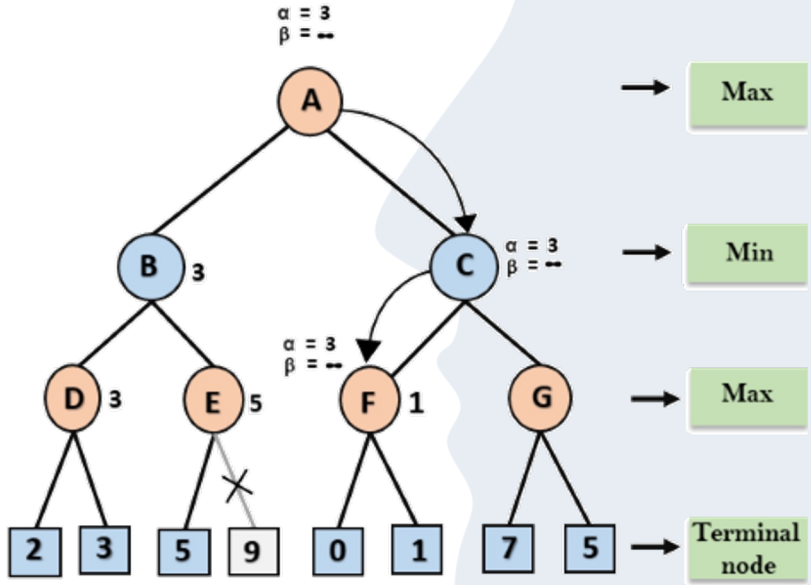
Images by https://www.javatpoint.com/ai-alpha-beta-pruning

Adversarial Search
Alpha-Beta Pruning

Step 3



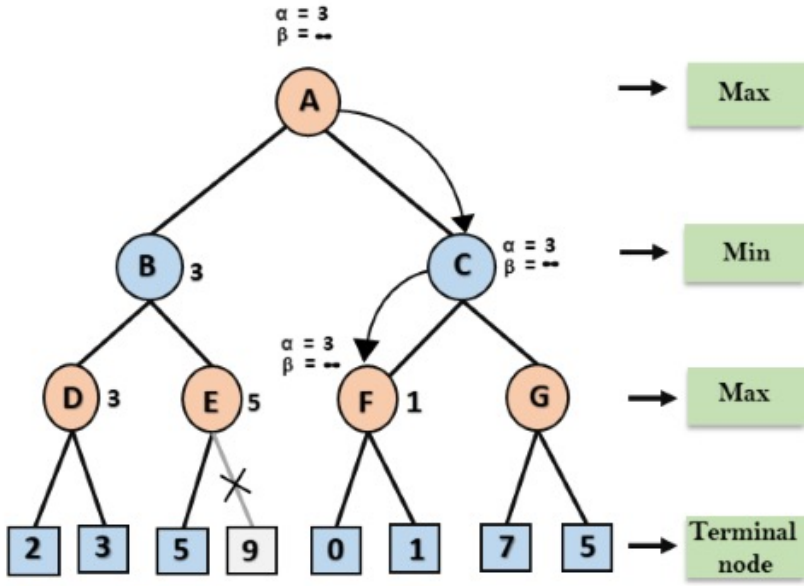
Step 4



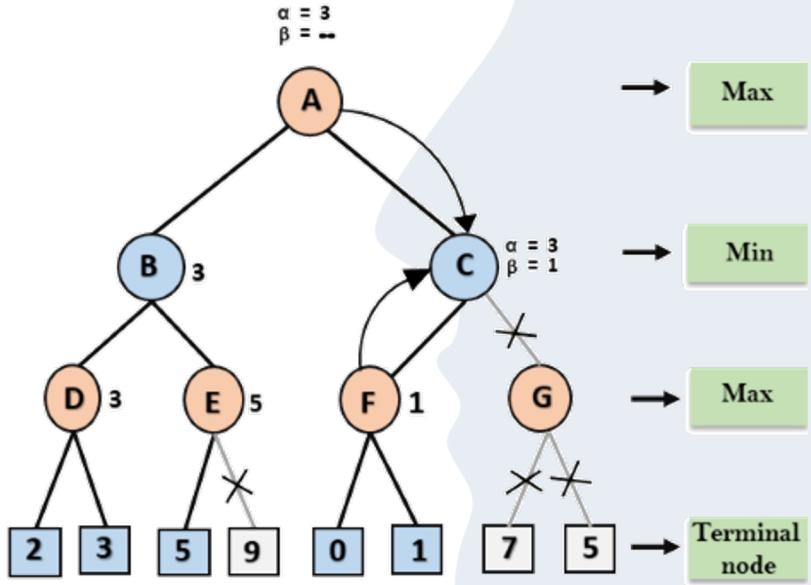
Images by https://www.javatpoint.com/ai-alpha-beta-pruning

Adversarial Search
Alpha-Beta Pruning

Step 4

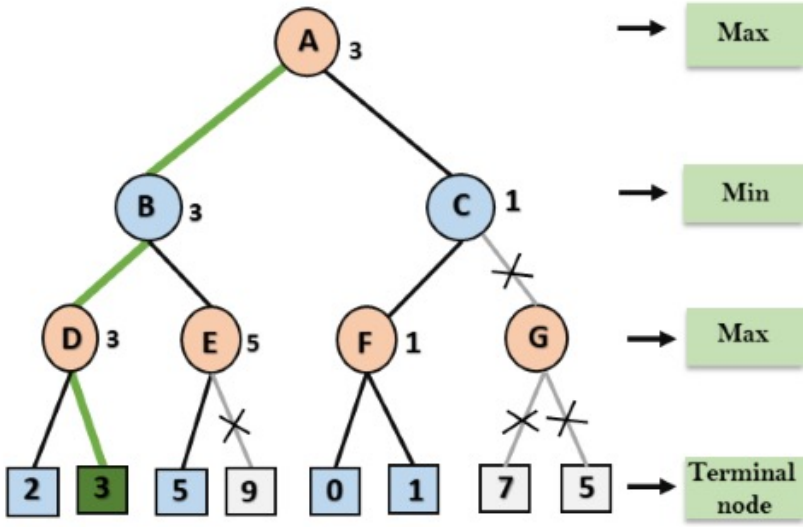


Step 5



Images by https://www.javatpoint.com/ai-alpha-beta-pruning

Adversarial Search
Alpha-Beta Pruning



- Alpha-Beta Pruning can save a lot of calculations, if the tree is well ordered
- In an ideal ordering (best move first) we can reduce the complexity to

$$O(b^{m/2}) \text{ instead of } O(b^m)$$

- To optimize Alpha-Beta Pruning you have to predict interesting path to increase the amount of cutoffs

Images by https://www.javatpoint.com/ai-alpha-beta-pruning

Adversarial Search
Go, the Destroyer of Tree Search?

Problems of Alpha-Beta Search:

- It needs a fast evaluation function
- In games with large branching factors (e.g. Go) the exploration of Alpha-Beta is simply not good enough
  - To explore the full tree minimax tree up to depth 3, we need 3003 = 27.000.000 evaluations.
  - Even with pruning it is too much to explore...

AlphaGo and AlphaZero

- In 2016, Deepmind introduced AlphaGo a combination of (Monte-Carlo) Tree Search and Neural Nets to beat Go

Adversarial Search
Outlook: AlphaGo

https://www.youtube.com/watch?v=g-dKXOlsf98

Search and AI
Why search has such a central role?

A lots of tasks in AI are intractable

- Search is the “only” way to handle them

Many application require some form of search

- e.g. Learning, Reasoning, Planning, Natural Language Understanding, Computer Vision,…
- Even in current State-of-the-Art algorithm, search has still a key role in their success

A lot of problems can be abstracted or displayed as a search problem!

Summary

Local Search and Optimization Problems

Gradient Descent

Adversarial Search

Minimax

Alpha-Beta Pruning

You should be able to:

- Explain how to find local and global maxima/minima
- Formulate a simple game as a game tree
- Use minimax to find optimal moves in a limited-size game tree

Next Week: CSPs