

# Artificial Intelligence Course

<del>9/11</del>	<del>Introduction</del>
<b>10/11</b>	<b>Uninformed Search, Heuristic Search</b>
<b>11/11</b>	<b>Local Search, Search in Games</b>
17/11	Constraint Satisfaction
18/11	Propositional Logics as Knowledge Representation
24/11	Predicate Logics, rules, prolog, resolution
25/11	Planning
01/12	Case-based reasoning and clustering
02/12	Decision Trees Learning
8/12	Bayes' Networks
15/12	Reinforcement Learning, Connectionist Approaches
05/01	Wrap up and A&Q for exam

# Water Jug Problem

- **Given:** 1 jug (3L) and 1 jug (4L) and a tap for filling jugs
- **Goal:** Get exactly 2L in the 4L jug.



4L



3L

- Don't give a solution,  
but think about how to solve that problem with natural intelligence?

# Problem Analysis and Modeling

- State

(x, y) with x liquid in the 4l jug and y is filling in the 3l jug  
(0, 0) is the start state

- What is the goal?

(2, 0) is the goal state

- What actions can we do?

*Fill 4l Jug:*  $(a,b) \rightarrow (4,b)$

*Fill 3l Jug:*  $(a,b) \rightarrow (a, 3)$

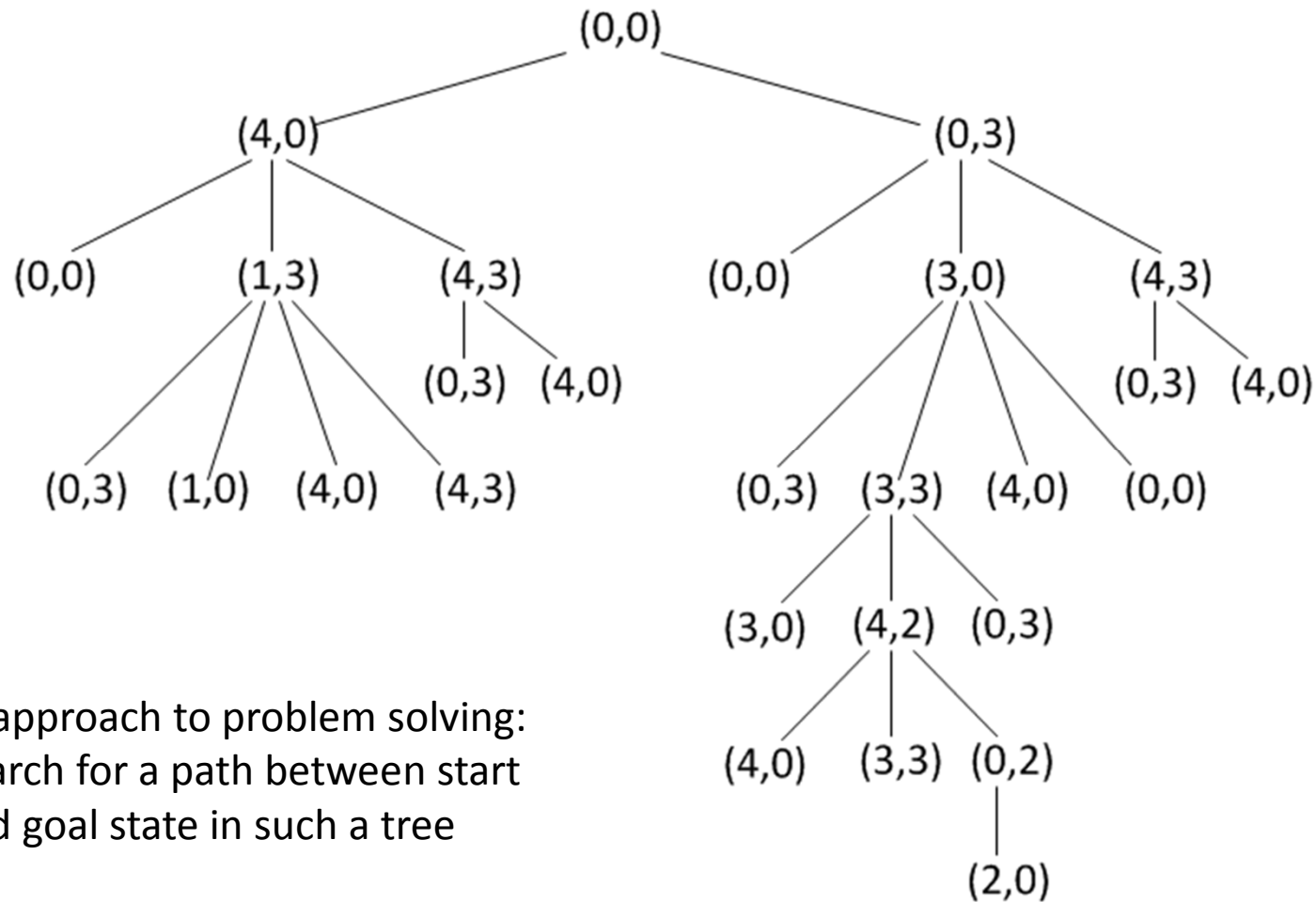
*Empty 4l Jug:*  $(a,b) \rightarrow (0,b)$

*Empty 3L Jug:*  $(a,b) \rightarrow (a,0)$

*Fill 4L from 3L Jug:*  $(a,b) \rightarrow (4, b-(4-a))$ , if  $b > 4-a$ ,  $(a+b, 0)$  otherwise

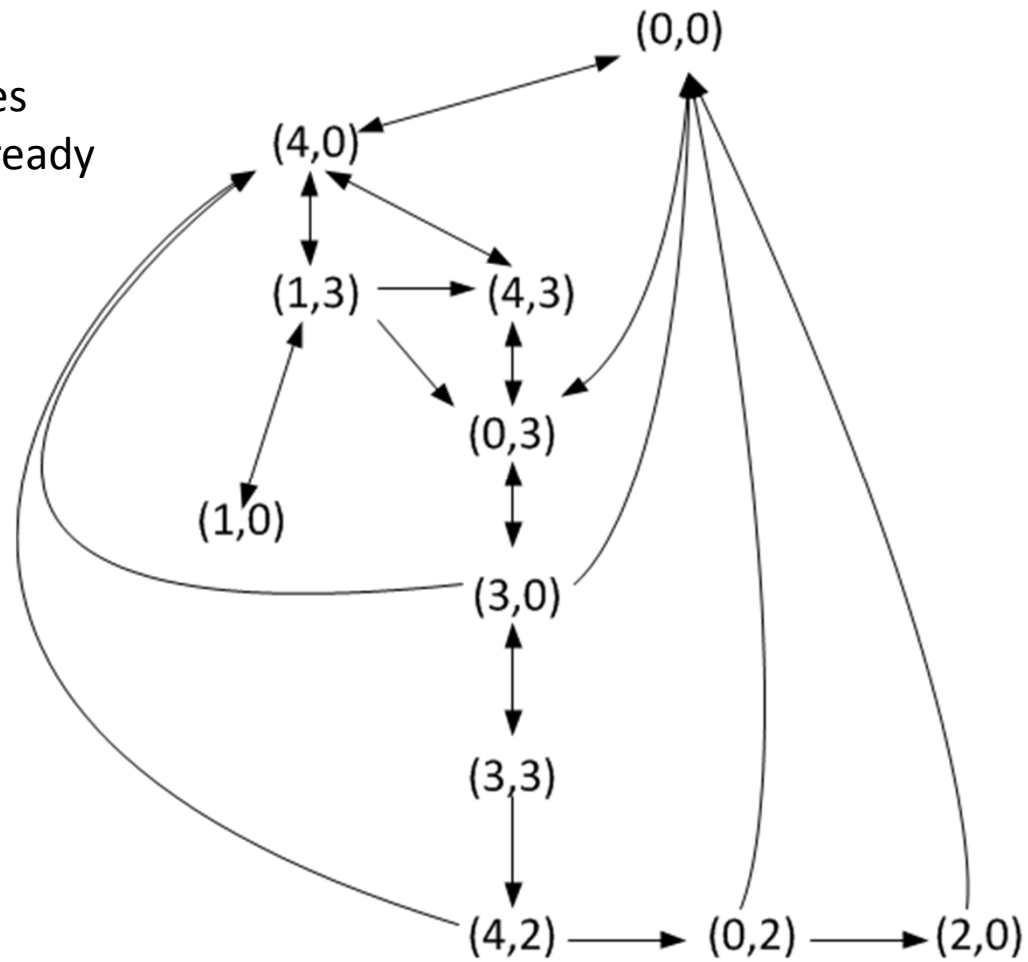
*Fill 3l from 4l Jug:*  $(a,b) \rightarrow (a-(3-b), 3)$ , if  $a > 3-b$ ,  $(0, a+b)$  otherwise

# From States and Actions to a Tree



# Search Graph

If we identify states  
that have been already  
there



# Search Problem Formulation

A **SEARCH** problem can be defined by *five* items:

- **Initial State**
- Description of possible **Actions**:  
given a state  $s$ ,  $\text{ACTIONS}(s)$  set of actions that are applicable in  $s$
- **Transition model**, a description of what the action does:  $\text{RESULT}(s, a)$  returns the state that results from doing action  $a$  in state  $s$ :  
→ **Actions and Transition Model determine State Space**,  
forms a graph/network with states as nodes, actions as links. A path in state space is a sequence of states connected by a sequence of actions
- **Goal Test** determines whether a state is a goal state
- **Path Cost Function** assigns numeric cost to each path. Cost function reflects performance measure;  
Assumption: Costs are additive, sums of non-negative step costs:  $c(s, a, s')$

# Problem Formulation

- **States:** Tuple with the amount of water in each jug: (X,Y)

**Initial state:** (0,0)

**Actions:** All actions: *Fill4LJug*, *Fill3LJug*, *Empty4LJug*,  
*Empty3LJug*, *Fill4Lfrom3LJug*, *Fill3LFrom4LJug*

$\text{Actions}((0,0)) = \{\text{Fill4LJug}, \text{Fill3LJug}\}$

$\text{Actions}((4,0)) = \{\text{Empty4LJug}, \text{Fill3LJug}, \text{Fill3LFrom4LJug}\}$

...

**Transition Modell:**

$\text{Result}((a,b), \text{Fill4LJug}) = (4,b)$

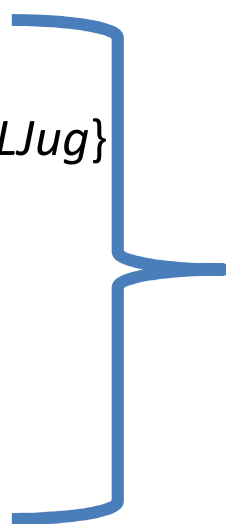
$\text{Result}((a,b), \text{Fill3LJug}) = (a,3)$

$\text{Result}((a,b), \text{Empty4LJug}) = (0,b)$

...

**Goal Test:** (2,\*)

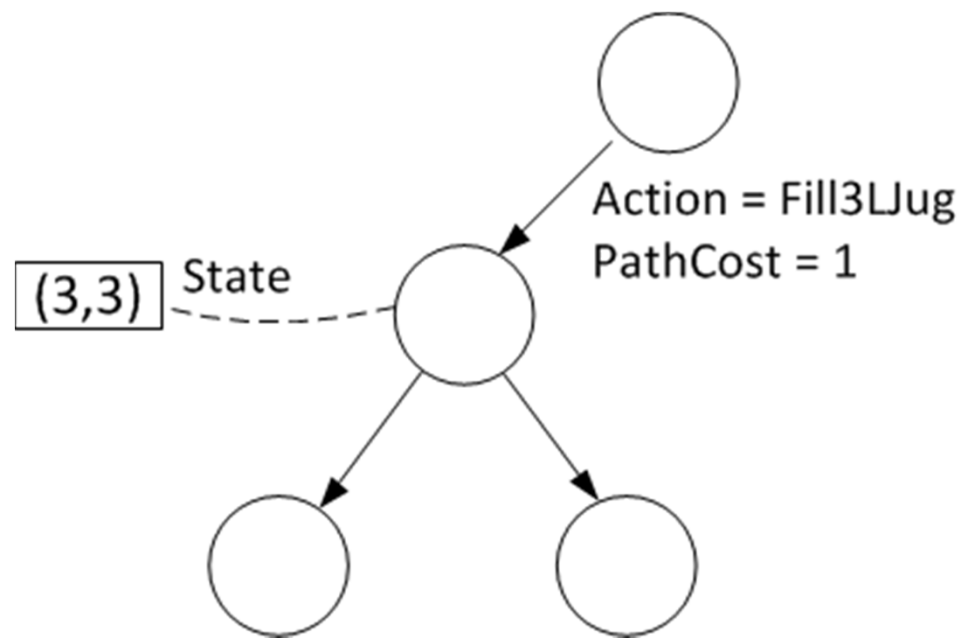
**Path Costs:** each step cost = 1



This defines  
**State Space**  
represented  
in the tree/graph

# States and Nodes in Search Tree

- **State** is a (representation of) physical configuration
- **Node** is a data structure, a constituting part of a search tree
  - includes: *parent*, *children*, *action*, *path-cost*
- Expand function creates new nodes, filling the various fields





# Searching for Solutions

- The **Search Tree** represents possible action sequences with initial state as root; branches are actions, nodes are states
- "Expanding" the current state = applying all legal actions to the current state generating a new set of states
- **Essence of search:** following up one option now and putting the others aside for later in case the first choice does not lead to a solution  
→ Set of current leaf-nodes available for expansion at any given point:  
"frontier"
- **Loopy paths critical**, but considering non-negative, non-zero path-costs, a loopy path is never better than a path without loops → redundant paths (algorithms that forget their history, are doomed to repeat it) → remembering already tested states

# Generic **Tree** Search Algorithm

```
function TreeSearch (Problem P) returns a path
  frontier = [[start]]
  Loop:
    if frontier is empty, then return FAIL
    path = remove-choice (frontier)
    s = path.end
    if goaltest(s), then return path
    for a in actions
      add [path+a→result(s,a)] to frontier
```

# Generic **Tree** Search Algorithm

```
function TreeSearch (Problem) returns a path  
    frontier = [[start]]
```

```
    while frontier:  
        path, node = pop(frontier)  
        s = parent(path)  
        if goal_test(node):  
            return path  
        for a in actions(node):  
            add [path, a]
```

A tree has no loops, a  
graph has – so if we want  
to tackle a graph, we need  
to keep track of the visited  
nodes

```
            frontier
```

# Generic **Graph** Search Algorithm

```
function GraphSearch (Problem P) returns a path
  frontier = [start]
  visited = []
  cameFrom = {start:null}
  Loop:
    if frontier is empty, then return FAIL
    node = remove-choice (frontier)
    add node to visited
    if goaltest(node), then
      return
      reconstruct-path (cameFrom, start, end)
    for next in actions
      if not in visisted or in frontier
        add next to frontier
        add next→node to cameFrom
```

# Reconstruct Path

```
function reconstruct-path(came-from, s, e) returns a path
  path = []
  node = end
  Loop:
    if node == e
      then return path
    else
      push node to path
      node = came-from[node]
             # to entry for node in came-from
```

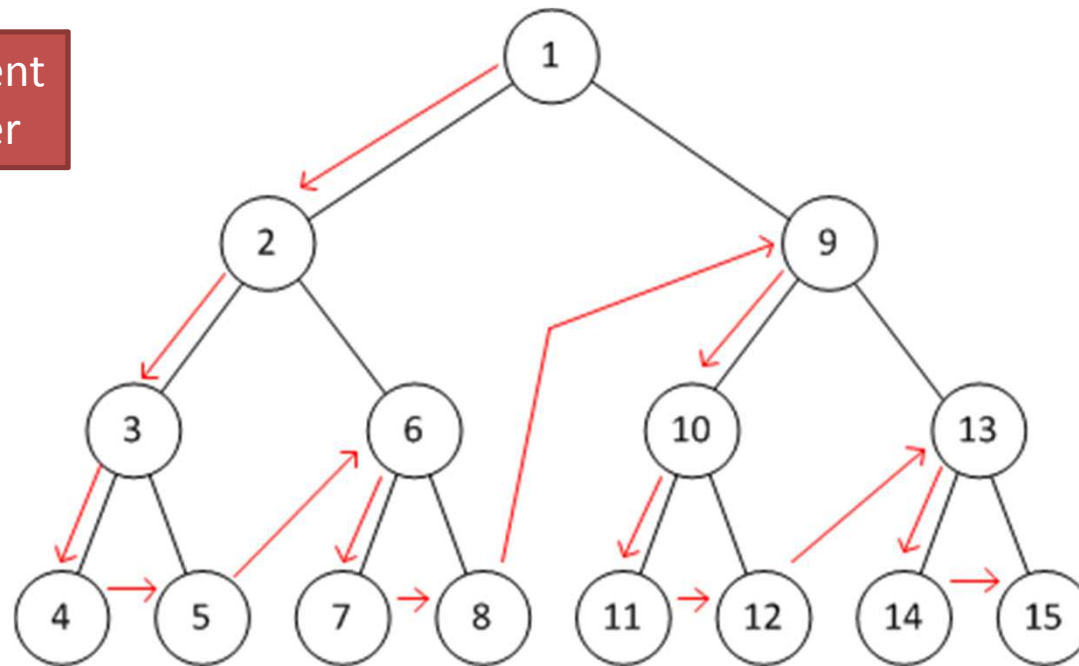
We assume that this function is called at the end of a successful search. For each node on our shortest path, there is a predecessor in the came-from. This is an iterative solution.

# Basic Search Algorithms

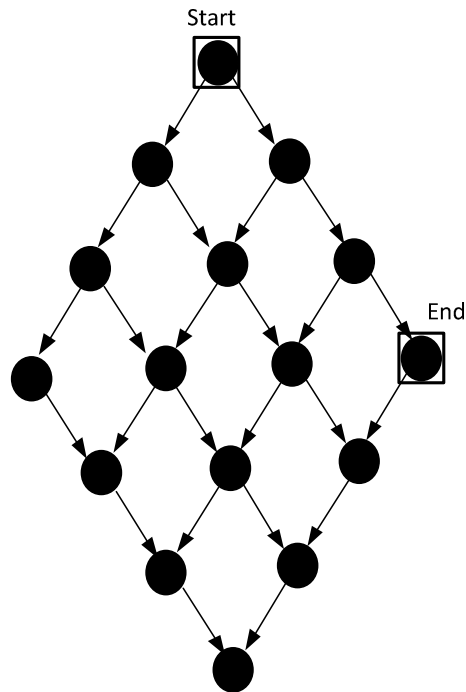
- This is a family of algorithms
- We need to define: **remove-choice** and **add**
- The two most prominent basic search techniques
  - **Depth-First Search**: Strategy: expand deepest unexpanded node
    - Remove-choice takes the **first** entry of frontier
    - Adds new entries to the **beginning** of frontier
  - **Breadth-First Search**: Strategy: expand shallowest unexpanded node
    - Remove-choice takes the **first** entry of the frontier
    - Appends new entries at the **end** of the frontier
- Remarks: these are **iterative** versions.

# Depth First Search

Get first element  
push to frontier



???



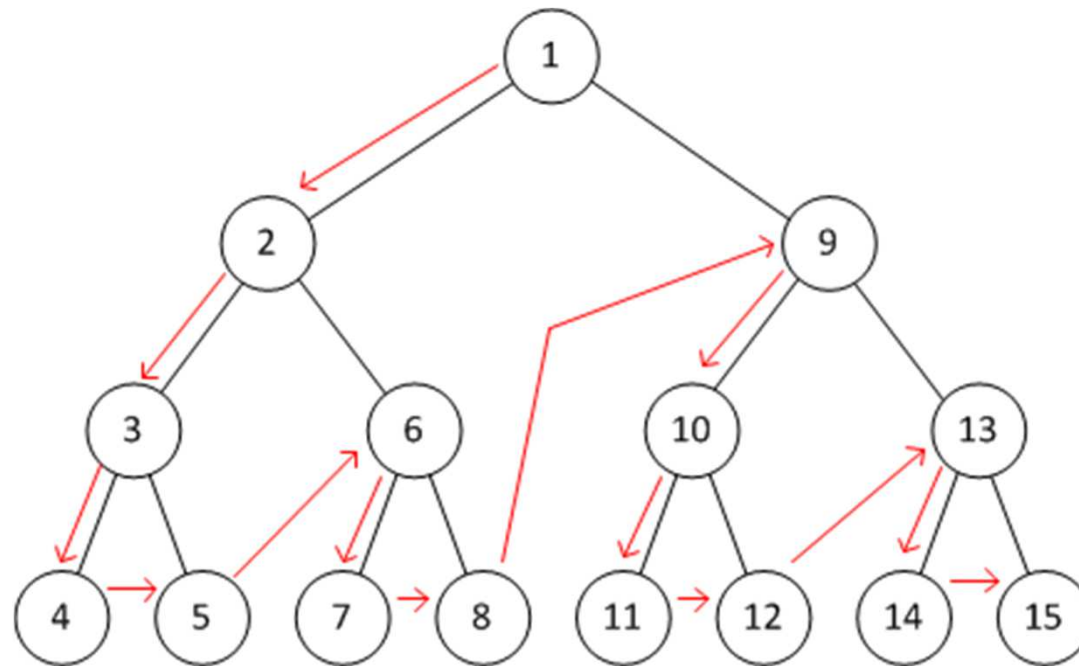
## Depth-First-Search

How many nodes do you need to expand, if nodes are expanded in a left-2-right way?

How many nodes do you need to expand, if nodes are expanded in a right-2-left way?



# Depth First Search



**Complete?**

**Optimal?**

**How much time?**

**How much space?**

**How to describe the performance of an algorithm?**

# Short Excursion: The Big-O-Notation

- Standard to describe **the complexity or performance of algorithms**
- $O(\dots)$  with ... as a characterization how the algorithm behaves, when the data set, to which it is applied, becomes very large ( $\rightarrow$  infinite)
- Describes upper bound
- Examples
  - $O(1)$ : No matter how large the data set becomes, using the algorithm costs always the same number of operations.
  - $O(n)$ : The algorithm's performance is directly proportional to the size of the data. Linear increase
  - $O(n^2)$ : effort that the algorithm needs raises proportional to the square of the data set
  - $O(N \cdot M)$ ,  $O(N \cdot \log N)$ ,  $O(\log N)$ ,  $O(2^N)$

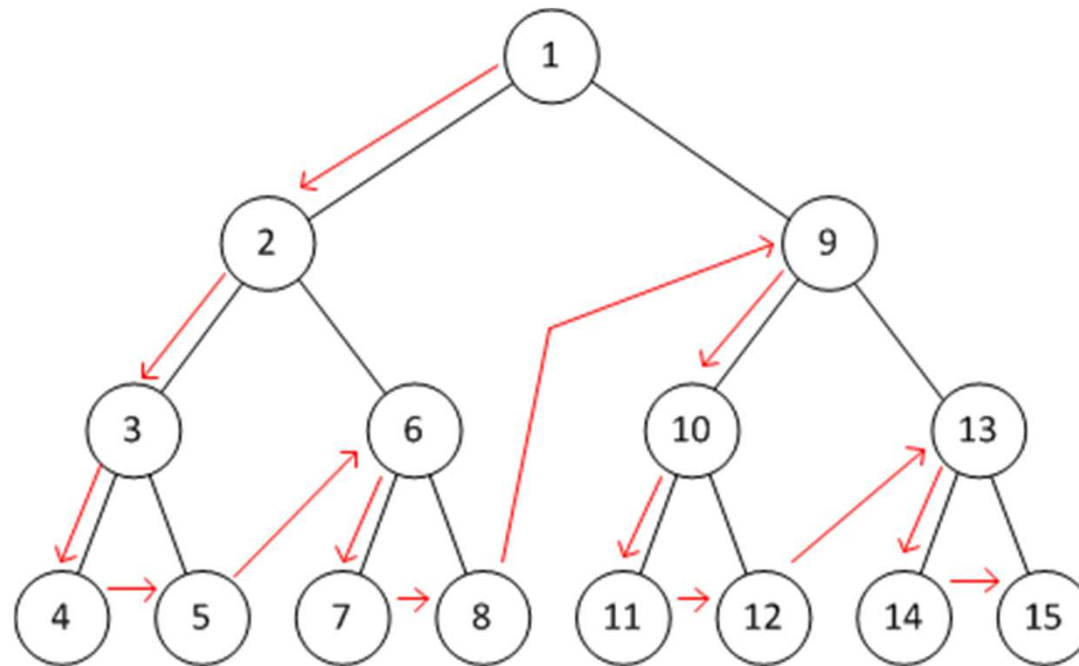
# What does that mean?

- Think about searching for an object in a very large **sorted** array. For finding the value, you need to compare your token to each entry in the array.
    - A. If you can calculate the index that belongs to the token directly → Costs for searching is always just one calculation, and thus independent from number of entries in the array →  $O(1)$
    - B. If you check all entries from index 0 on, in the worst case the entry that you search is the last one → you have checked each entry once →  $O(N)$  with  $N$  is the number of entries
    - C. More efficient binary search: Check the entry in the middle, is your token larger or smaller → Check the entry in the middle of the corresponding half array → ... Search like your array would be a tree → if the entry is always between → you search to the leaves →  $O(\log N)$
- A would be optimal, C is better than B

# Algorithm Complexity Analysis

- Identify costly atomic operations (comparisons)
- Count how often the operation is in the worst case  
→ identify relevant terms  
(no absolute numbers, if there is a  $n^x$ ,  $n^{x-1}$  is not relevant...)
- For example: an algorithm applied on  $n$  items:
  - 5 times loops through all items: for each loop, there is an inner loop which loops again →  $5*n*n$
  - After that, there is another operation which loops 2 times through all items again →  $2*n$
  - And there are 8 additional costly operations.  
→ Cost:  $5*n^2 + 2*n + 8$
  - For really large  $n$  → the important term is  $n^2$  →  $O(N^2)$

# Depth First Search



**Complete?** No, fails in infinite-depth trees, graphs with loop

**Optimal?** **No**

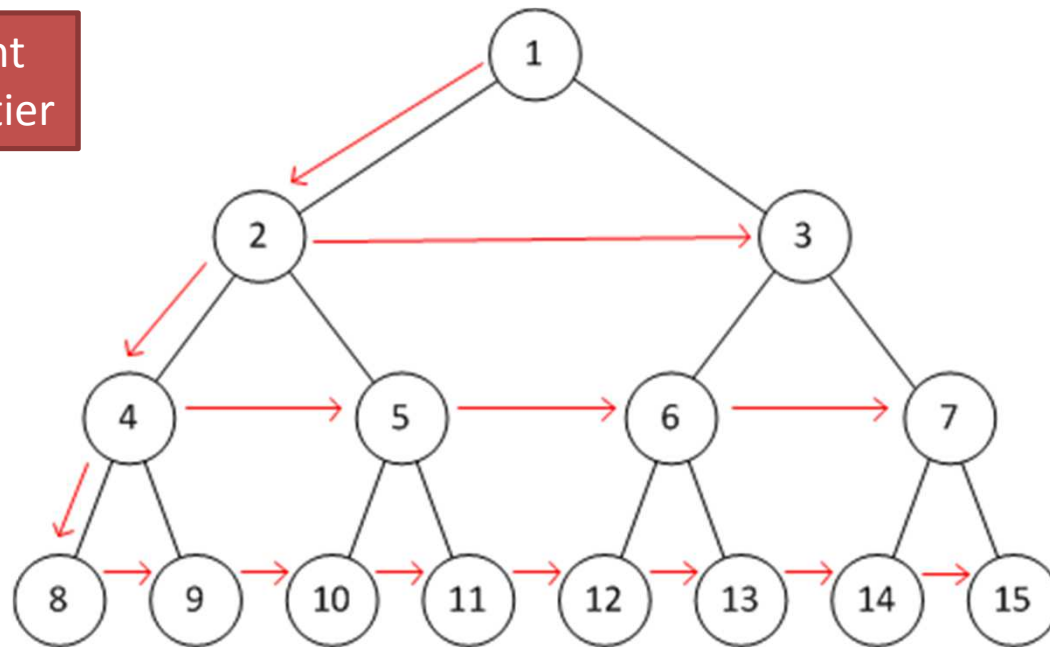
**How much time?** in worst case complete tree is searched

$O(N)$  if explicit or  $O(b^d)$  with branching factor  $b$ , and  $d$  depth searched

**How much space?**  $O(d)$  (recursive, depends on implementation)

# Breadth First Search

Get first element  
append to frontier



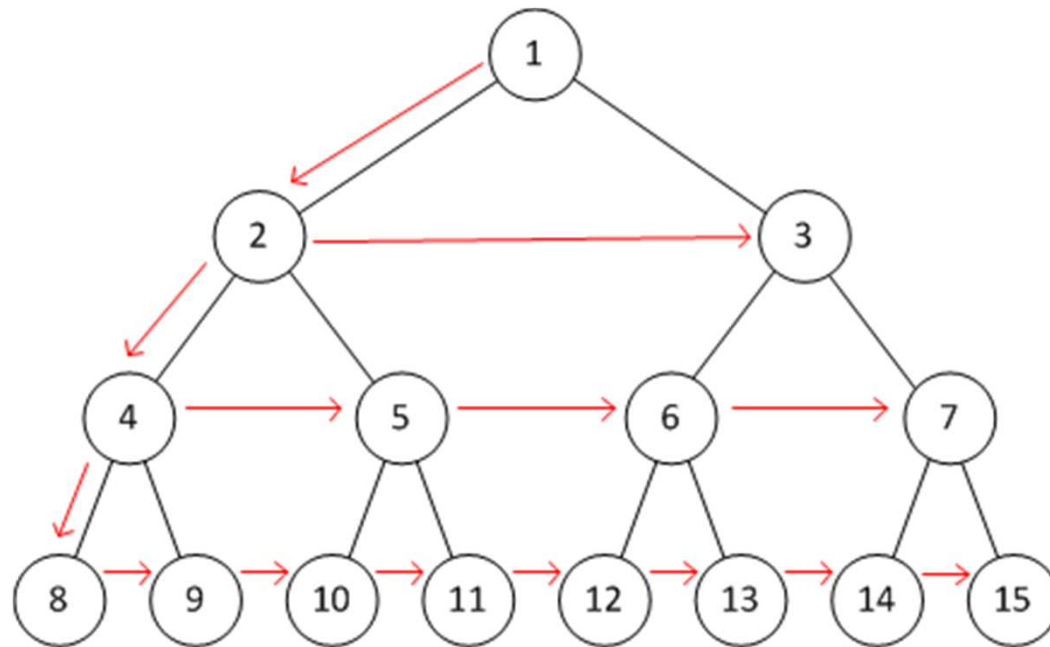
**Complete?**

**Optimal?**

**How much time?**

**How much space?**

# Breadth First Search



**Complete?** yes

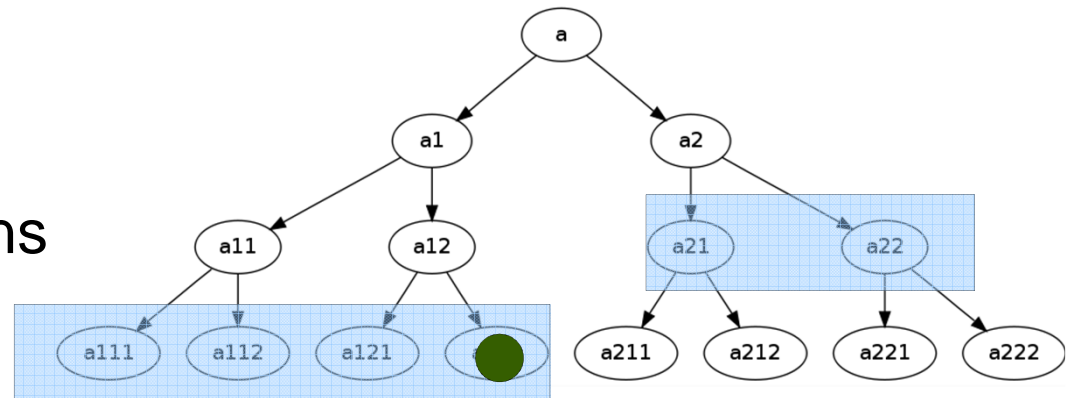
**Optimal?** yes

**How much time?** Same as depth first search  $O(b^d)$

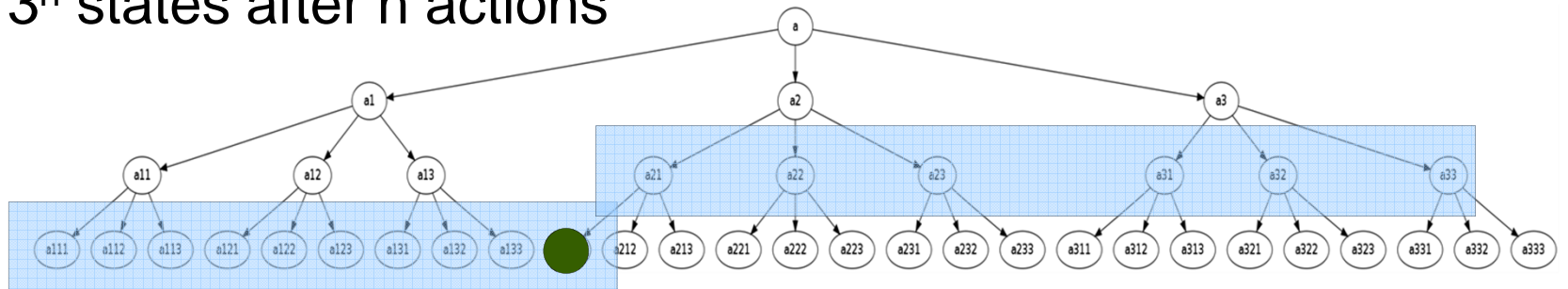
**How much space?** **Worst case:  $O(b^d)$**

# BfS and Branching Factor

- If we have 2 choices at each state  
→  $2^n$  states after  $n$  actions

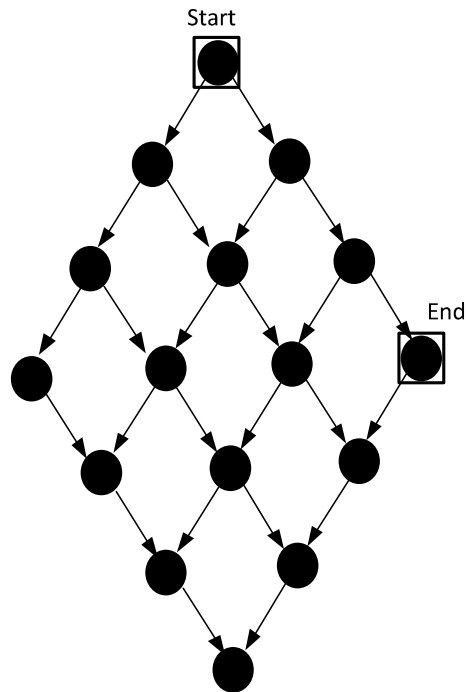


- If we have 3 choices at each state  
→  $3^n$  states after  $n$  actions





???



### Breadth-First-Search

How many nodes do you need to expand, if nodes are expanded in a left-2-right way?

How many nodes do you need to expand, if nodes are expanded in right-2-left way?

# Combining approaches

- Depth first search: uses linear space, **but may not find optimal state**
- Breadth first search: finds optimal state, but uses exponential space
- Can we combine the advantages of both approaches?

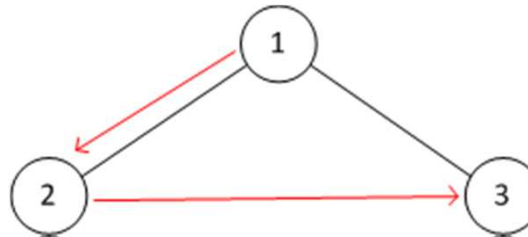
## → Iterative Deepening Search:

```
function Iterative-Deepening-Search(problem) returns a  
  path  
  for depth = 0 to  $\infty$   
    result  $\leftarrow$  DEPTH-LIMITED-SEARCH(problem, depth)  
    if result  $\neq$  Fail then return result
```

- Depth-limited Search is depth-first search with a limit on depth to avoid being trapped in endless loops
- One can show that this algorithm combines the advantages of both.

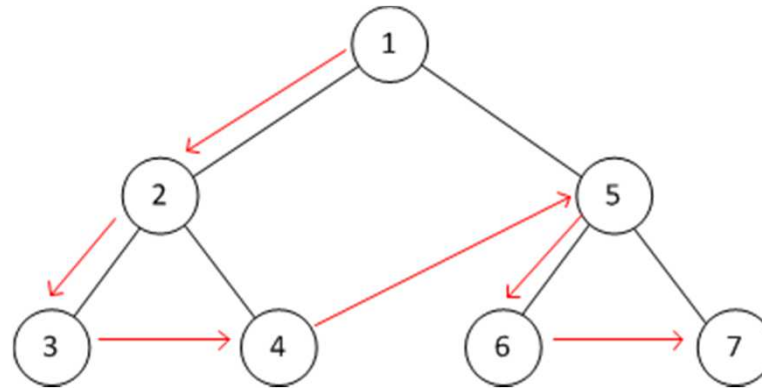
# Iterative Deepening Search

depth = 1



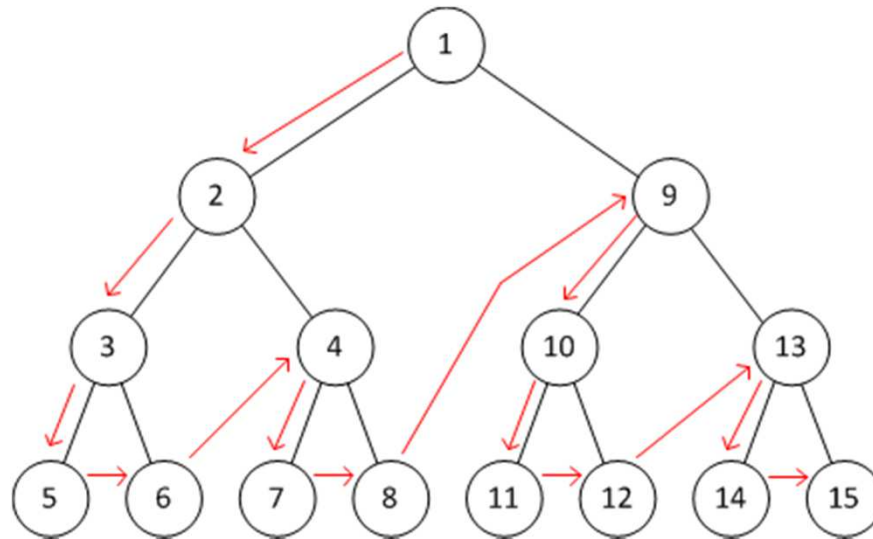
# Iterative Deepening Search

depth = 2



# Iterative Deepening Search

depth = 3



**Complete?** yes

**Optimal?** yes

**How much time?** Not so much more, as the most effort comes from the last row.

**How much space?** Same as depth first search

# Another Example: The 8-Puzzle

7	2	4
5		6
8	3	1

Start State

States?  
Actions?

	1	2
3	4	5
6	7	8

Goal State

Goal Test?  
Path Cost?

# General Remarks about State Space

- Real world is incredibly complex → state space must be **abstracted** for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions, e.g. "Center → University" represents a complex set of possible routes, detours, rest stops, steering actions...
- For guaranteed realizability, **any** real state "Center" must get to some real state "at University"
- Abstraction
  - is **valid**, if each solution in the abstract state space can be expanded into a solution in the more detailed world.
  - is **useful**, if carrying out each of the actions in the solution is easier than in the original problem





# Informed Search

- Informed Strategies use **problem-specific knowledge** in addition to the definition of the problem
- Problem-specific knowledge mostly in form of **heuristics** , e.g. How close a potential next node already is to the goal; → **Heuristic Search**
- Heuristic does not need to be perfect, give a rough guide how good the current node is
- Next node  $n$  for expansion is selected using an **evaluation function  $f(n)$**  → frontier is a sorted in descreasing order of desirability.  
That means if evaluation function gives future costs → take minimum

# Best-first Search

- "Greedy Search"
- Evaluation Function  $f(n)$ : **estimate of costs from node  $n$  to the goal**  
→ expands the node next that appears to be the closest to the goal
- Often used heuristic on maps:  $h_{sl}$ : Straight-line distance

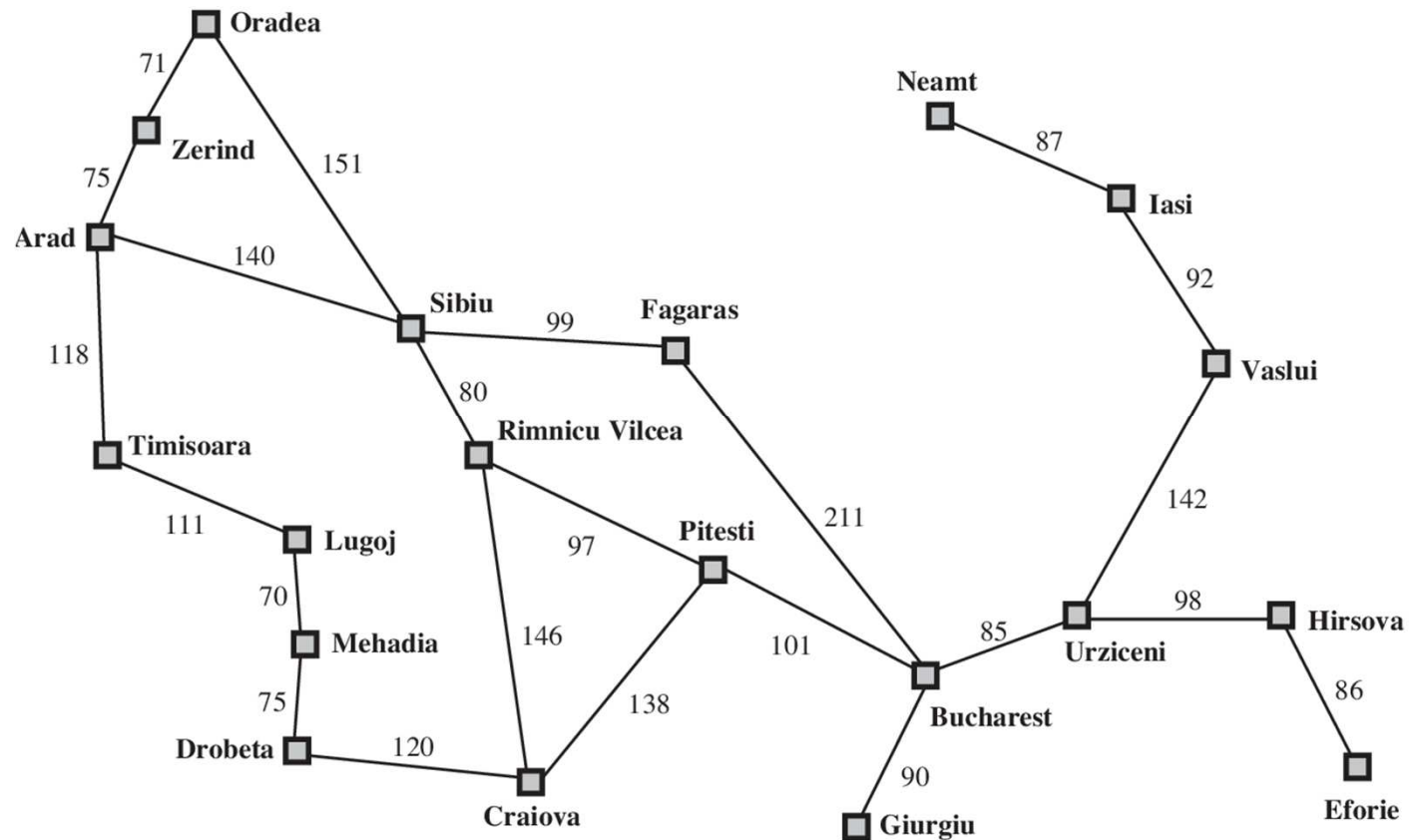
```
function Best-First Search (Problem P, heuristic h)
returns a path
    frontier = [start]
    closed = []
    Loop:
        if frontier is empty, then return FAIL
        node = pop best acc to h (frontier)
        add node to closed
        if goaltest(node), then return reconstructed path
        for next in actions(node)
            if next not in frontier or in closed
                add next to frontier
```

# Another Example: Romania

- On holiday in Romania
- Flight leaves tomorrow from Bucharest
- A number of potential routes for going to the airport
- **Start State:** in "Arad"
- **Goal State:** arrive in Bucharest (in Time)
  - States: various cities
  - Actions: drive from one to the other town
- Transition Model given by the road map: nodes are towns (state "at town"), actions: drive from town a to town b
- **Path costs: distance for driving between a and b**

Example from Russell&Norvig  
Ertel uses a similar map of Southern Germany

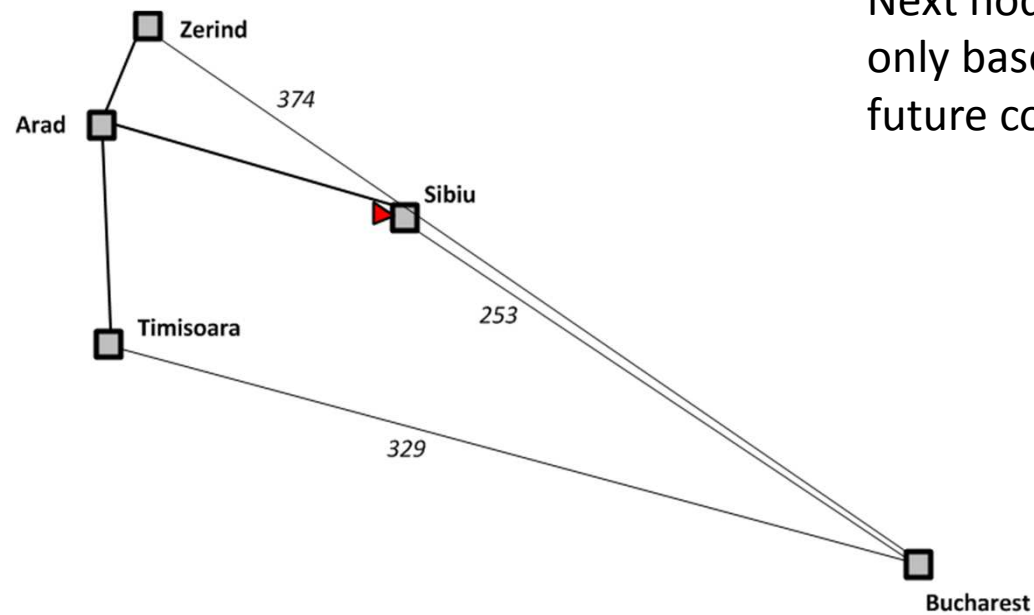
# A map as Search Graph



Start: Arad

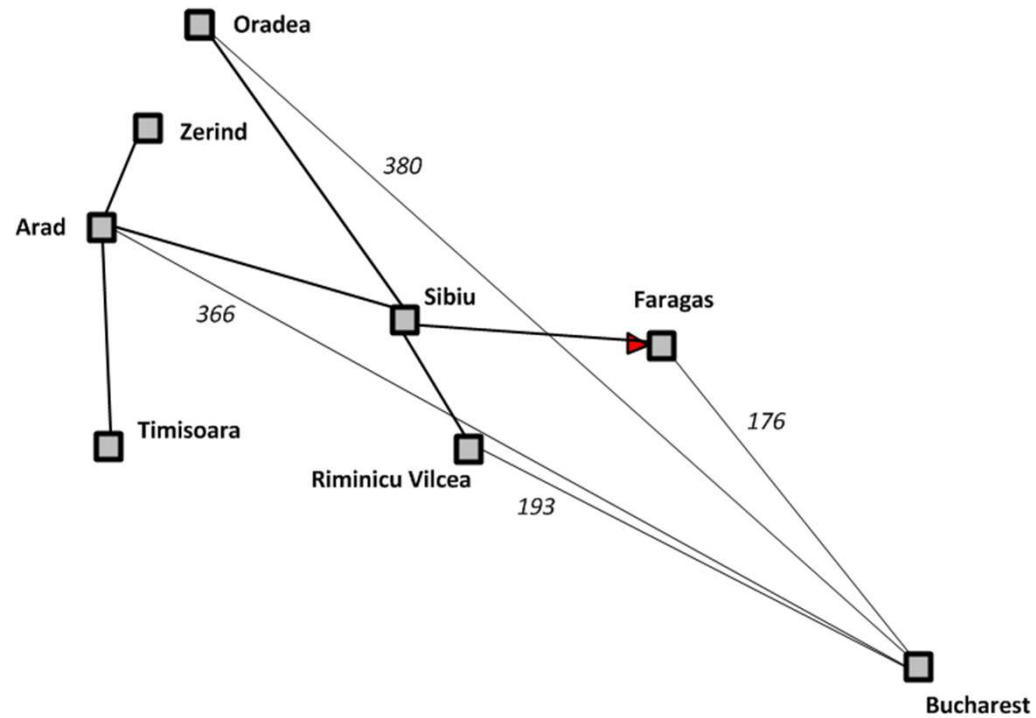
Goal: Bucharest

# Best-first search

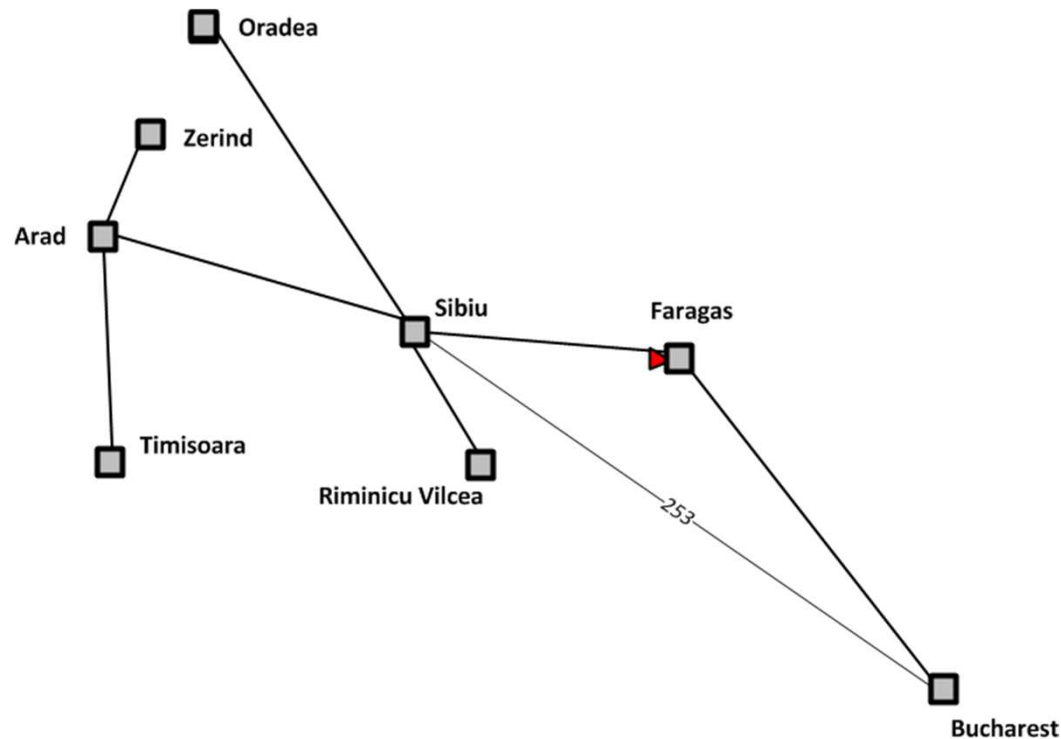


Next node to expand selected only based on heuristic about future costs

# Expanding from Sibiu



# Expanding from Faragas



The future cost estimation at Bucharest = 0  
we found our path: Arad → Sibiu → Faragas → Bucharest  
However, this is not the optimal one!

# Best-first Search

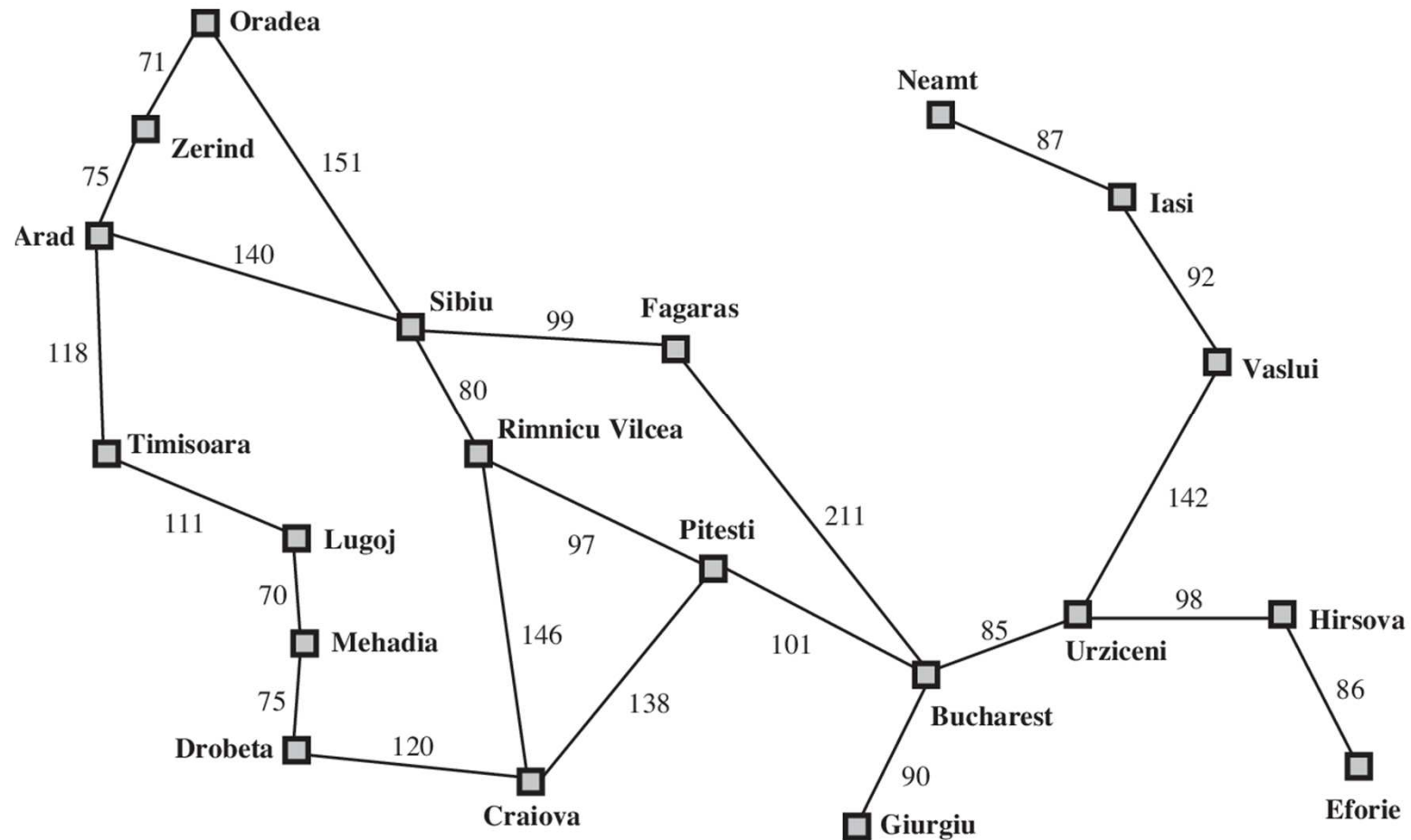
- **Complete?** no, can get stuck as in loops
- **Time?** In the worst case, everything needs to be expanded, but a good heuristic may give dramatic improvement
- **Space?** Similar to breadth first search
- **Optimal?** no



# A\* search

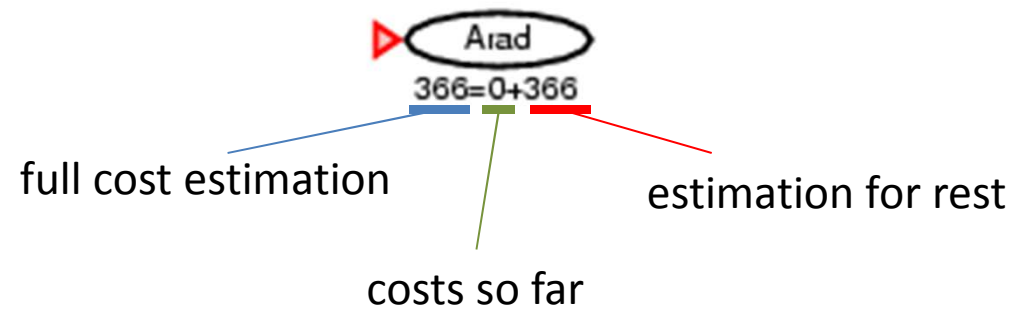
- Idea: do best-first search, but avoid expanding paths that are already expensive → **consider complete path costs**
- Evaluation function:  **$f(n) = g(n) + h(n)$** 
  - $g(n)$  = cost so far to reach node  $n$   
 $h(n)$  = estimated costs from  $n$  to goal  
 $f(n)$  = estimated total costs of path through  $n$  to goal
- A\* search uses an **admissible** heuristic:  $h(n) \leq h^*(n)$  where  $h^*(n)$  is the true cost from  $n$ , also require  $h(n) \geq 0$ ,  $h(G) = 0$  for any goal  $G$ .  
Heuristic is non-negative, and **must not overestimate real costs**
- Example:  $h_{\text{slid}}(n)$  does not overestimated the actual road distance.
- With admissible heuristic: **A\* search is optimal**

# Back to Romania

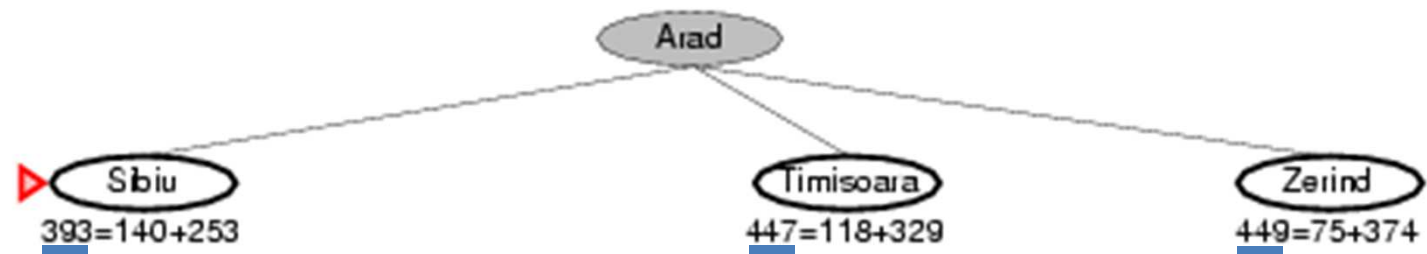


# A\* for search the optimal path

## Arad → Bucharest

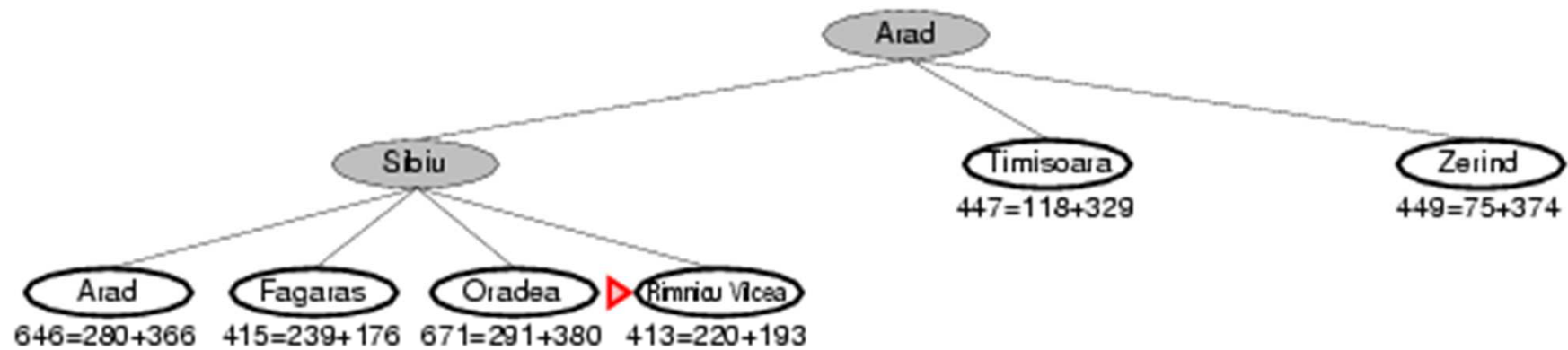


# A\* search example

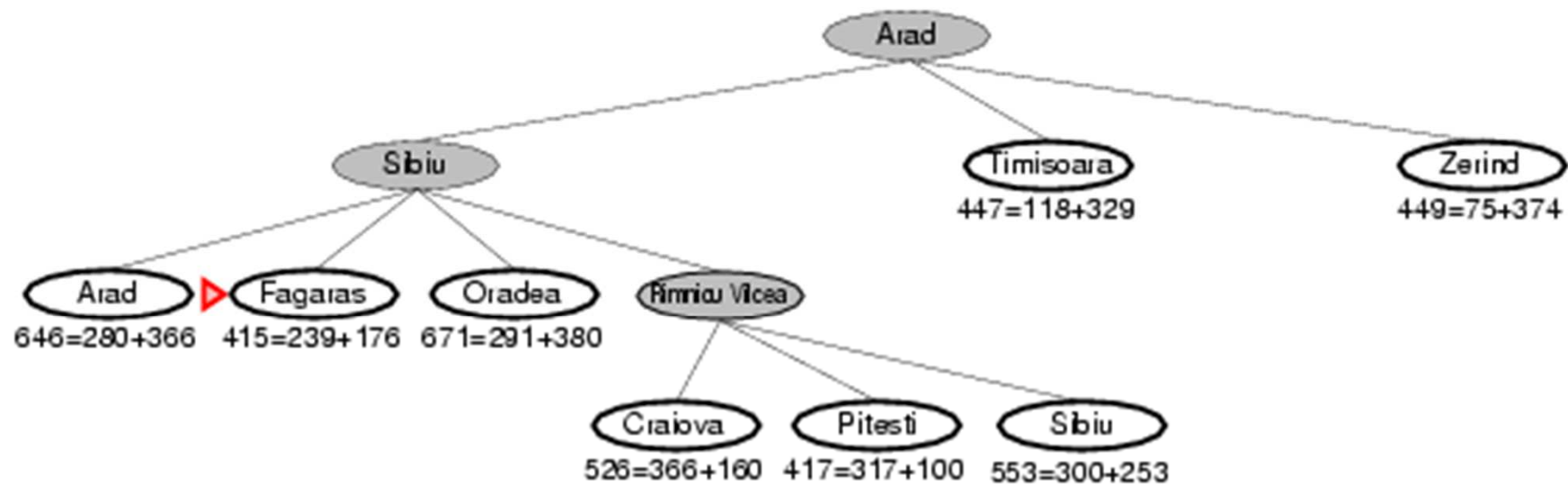


Node with minimal overall cost estimation is the next to expand

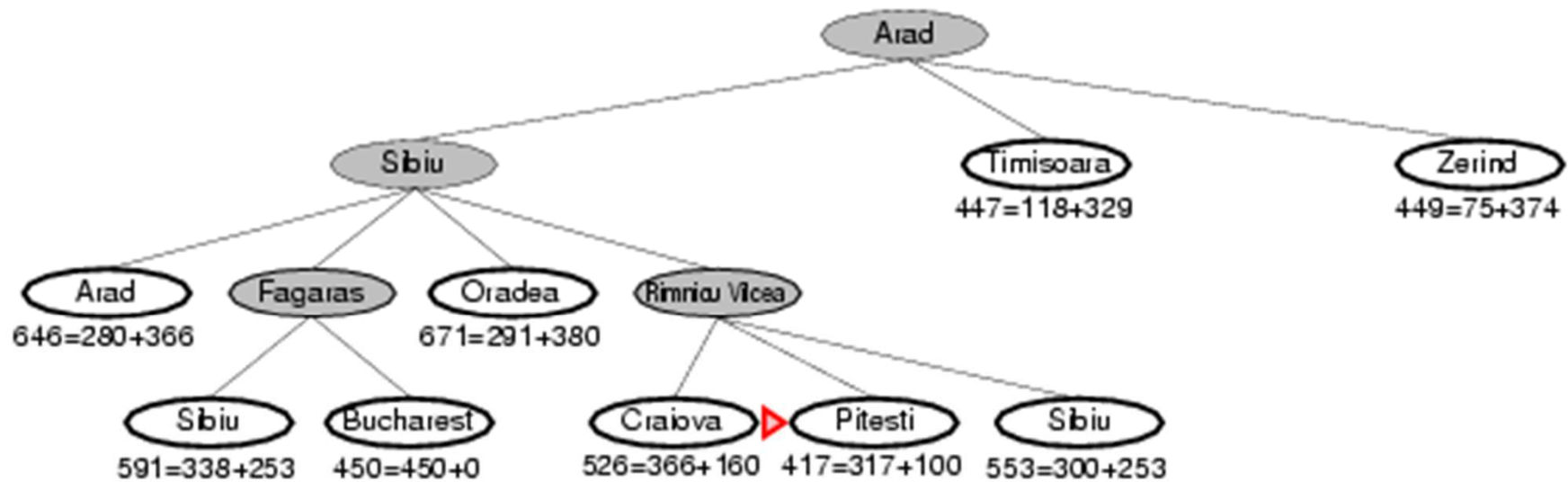
# A\* search example



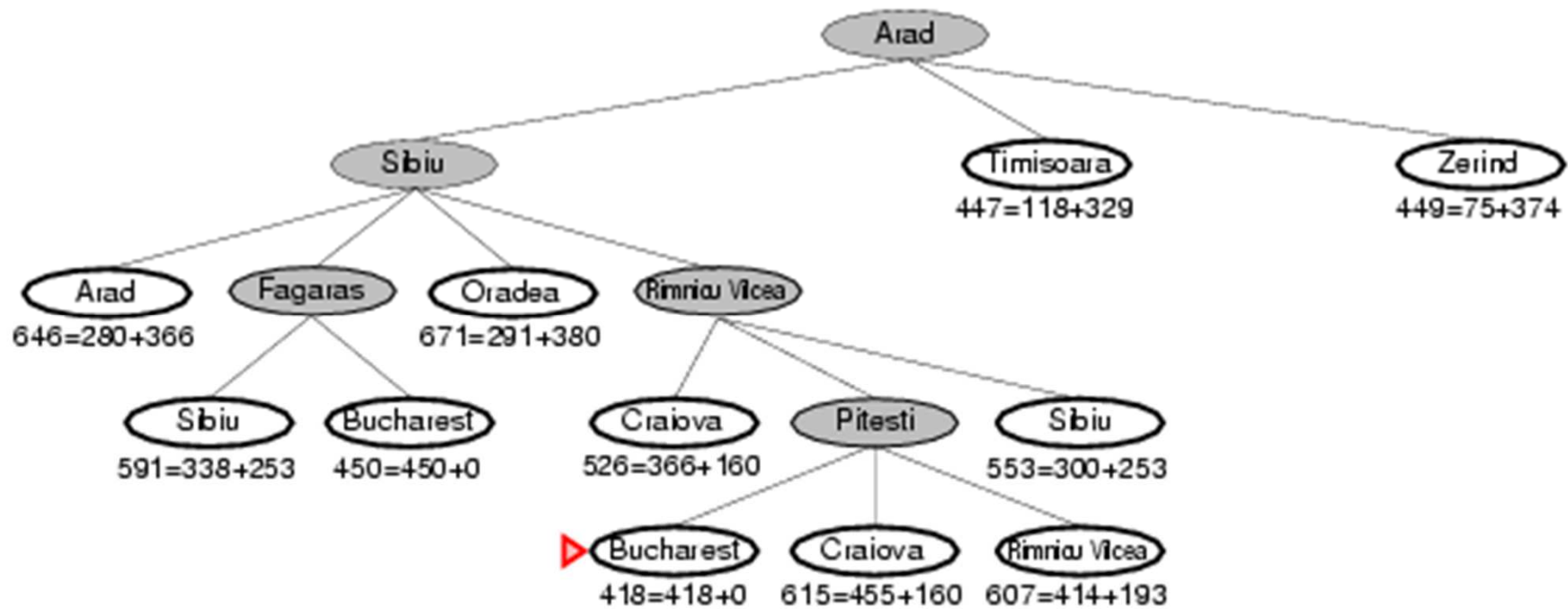
# A\* search example



# A\* search example



# A\* search example





# A\* Pseudo Code

```
function A*(Problem P) returns a path
  frontier = {init}
  explored = {}
  loop:
    if frontier is empty, then return FAIL
  node = select-best(frontier)
  if goaltest(node), then return node 2
  remove node from frontier
  add node to explored
  for next in successors(node)
    calculate score = f(next) 1,2
    if next in explored
      if score is better than before
        remove next from explored, 1,2
        add next to frontier
    else if next in frontier
      update score 1,2
    else
      add next to frontier
```

# For Implementation

- Keeping track of the costs 1
  - Add a data structure for keeping track of the costs so far ("costs-so-far" dictionary/hashtable)
  - Whenever you calculate the score of a node, update/add the information about the costs into this data structure
- Reconstructing the path 2
  - This algorithm is just for returning the searched node, if you are interested in the path, you need to
    - Add a data structure that allows you to keep track from where the node was reached ("came-from" dictionary/hashtable)
    - Update that data structure when you found a shorter path to a node

# Properties of A\*

- **A\* is guaranteed to return the optimal path, ONLY IF the heuristic never overestimates future costs!**
- A\* expands all nodes with  $f(n) < f(C^*)$   
A\* expands some nodes with  $f(n) = f(C^*)$   
A\* expands no nodes with  $f(n) > f(C^*)$
- Complete: Yes, unless there are infinitely many nodes with  $f \leq f(C^*)$
- Time: Exponential in [relative error in  $h$  x length of solution]
- Space: Keeps all nodes in memory
- Optimal: Yes
- Different variants of A\* improving weak points:  
Iterative Deepening A\*, etc.

# Admissible Heuristics

- E.g. for the 8-Puzzle
  - $h_1(n)$  = number of misplaced tiles
  - $h_2(n)$  = total Manhattan distance (for each tile, add number of squares it is away from desired location)

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

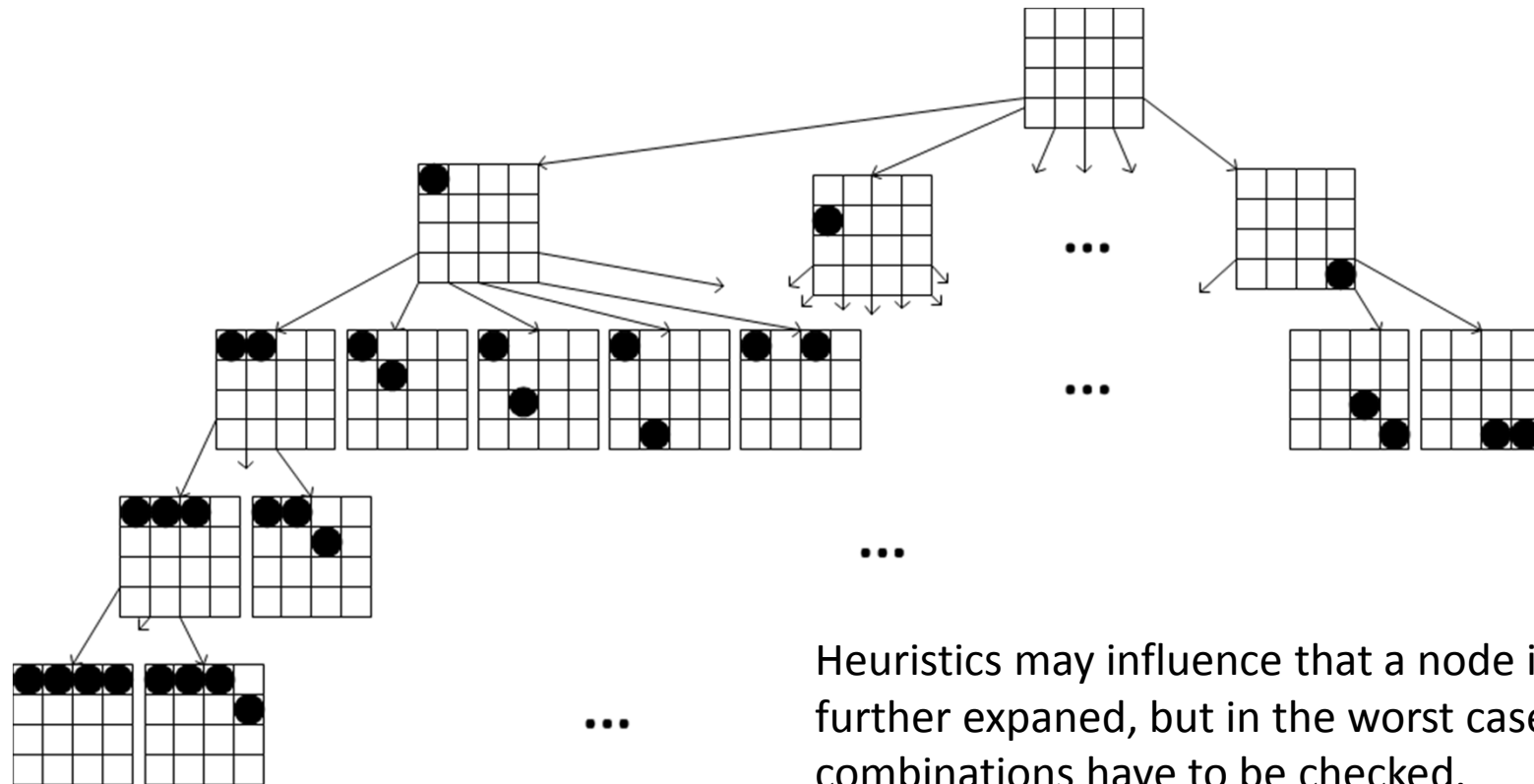
- $h_1(n)=?$                        $h_2(n)=?$
- *Admissible (and consistent) heuristics may found looking at relaxed problem formulations*



# Local Search Algorithms

- There are problems, in which the **path** to the goal is irrelevant; **the goal state itself is the solution**
  - Typical Examples:
    - **Traveling Salesman Problem (TSP)**: Find the shortest tour connecting different locations and return to the start position
    - **N-queens**: Find a configuration of n queens on a nxn chess board with no two queens on the same row, column or diagonal
  - Generic approach in such cases: **iterative local search algorithms**: keep a single "current" state, try to improve it until the optimal (or sufficiently good) solution is found.
  - **state space = set of "complete" configurations with actions modifying a fully elaborated state**
- Heuristic optimization

# Different from State-space Search

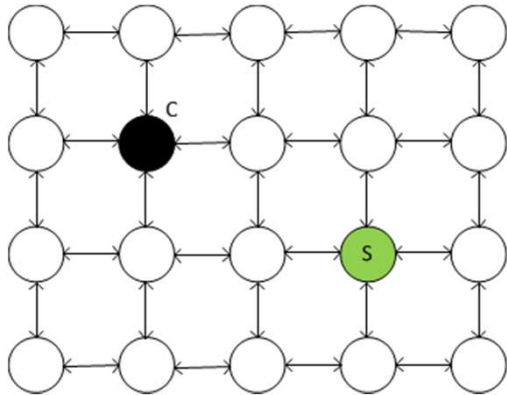


Heuristics may influence that a node is not further expanded, but in the worst case all combinations have to be checked.

# Local Search

## *Global Perspective:*

How to find solution S with current C ?



**Nodes:** candidate solutions  
(c=current, s=solution)

**Edges:** connections between  
neighbouring candidates

## *Local Perspective:*

What is the next step towards S?



Next position is searched from  
local neighbourhood based on  
local information, e.g. heuristics

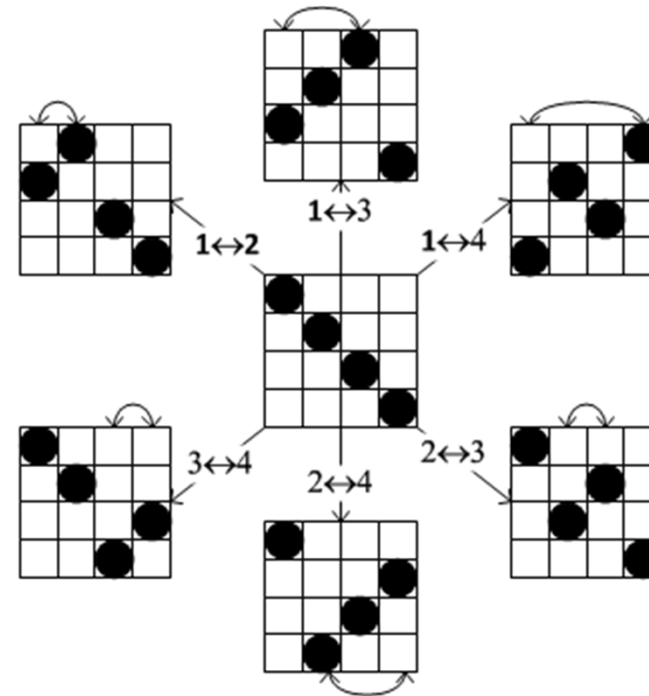
Taken from:

H. Hoos and T. Stützle: Stochastic Local Search, Fundamentals  
and Applications, Introduction Tutorial at AAAI 2004



# Local Operations

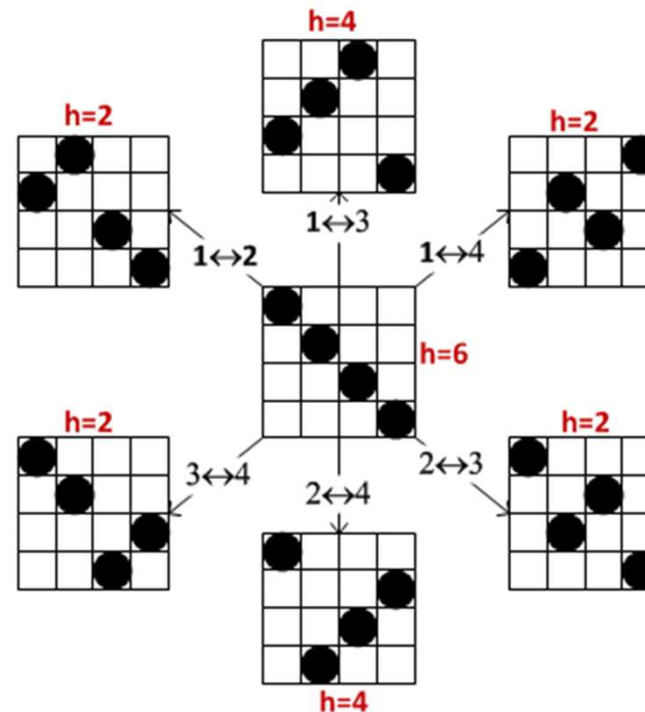
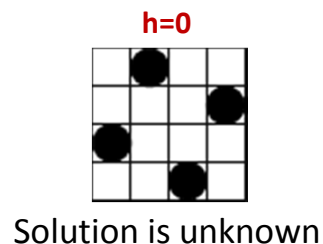
- In the n-queens problem, local operations can be:
  - *Move* one queen to another position
  - *Swap* the positions of two queens
- Operations connect two fully specified candidate solutions, not partial solutions.
- **But how to choose the next candidates**
  - Different particular algorithms



# State Evaluation

- What could be a **heuristic** that describes how good/bad or how far a candidate is from the solution?

→ Number of conflicts



→ Alternative operation for swapping:  
**Move one queen to reduce conflicts**

→ With the move operation and the conflicts heuristic, the n-queen problem can be solved for large n (~1Mio)

# Hill-Climbing

- Corresponds to **Best-first = greedy local search**
- "Like climbing Everest in thick fog with amnesia"

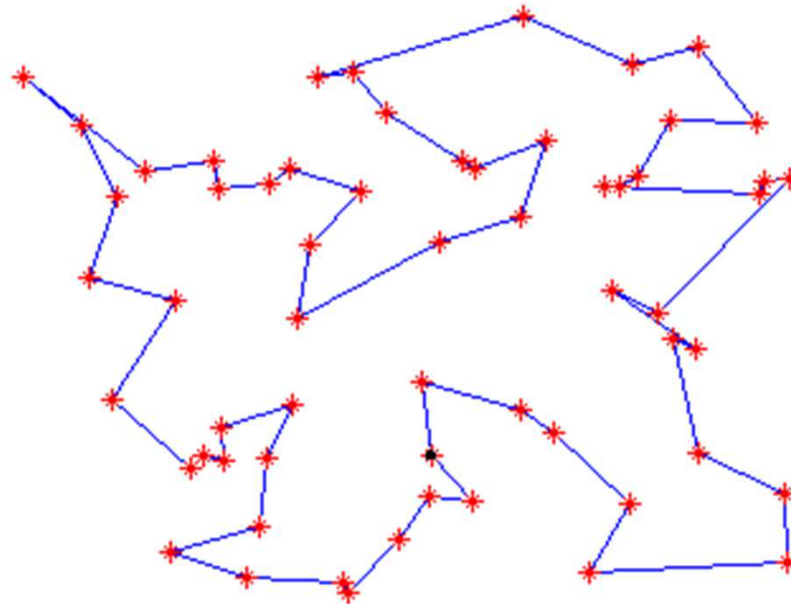
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  inputs: problem
  local variables:   current, a node
                      neighbor, a node

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

- "Gradient Search" in continuous case:  $x \leftarrow x + b * df(x)/dx$

# Traveling Salesman Problem

- Given a list of cities to visit, find an order to visit them that minimizes the total length to travel
- Each city is given by a name, and distances to the other cities.
- Many applications not just for salesmen, but also in (chip) production for optimal sequences of brazing, etc.

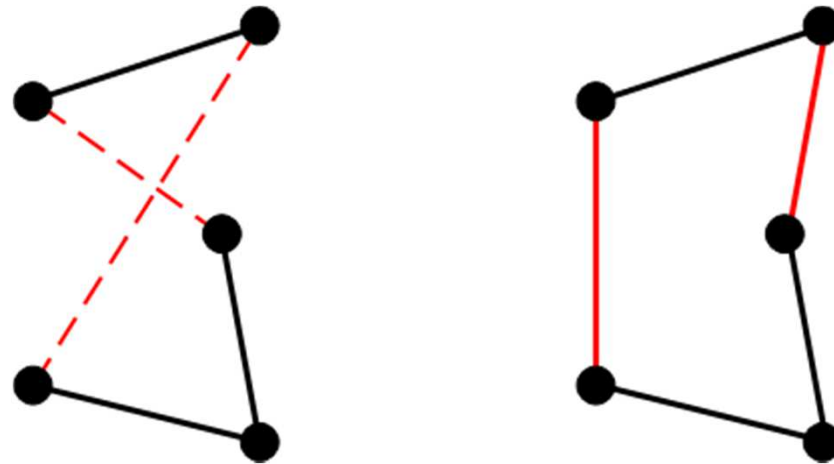


# Hill-Climbing for TPS

- How to represent a solution/state?
- What is the initial solution?
- How to calculate the neighbour-states?
- How to evaluate a state?

# Local Search → Iterative Improvement

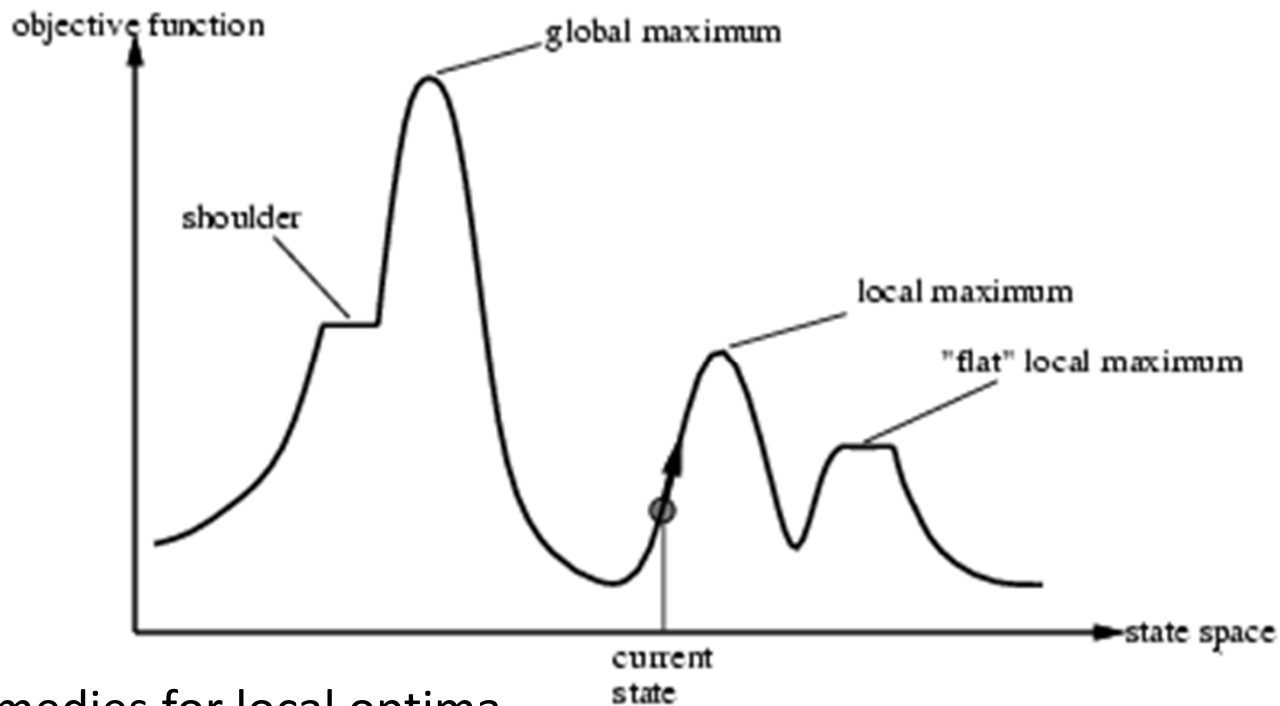
- Start with random tour as the first current
- Perform pairwise exchanges



- Continue with best changes
- Variants with this approach reach a solution within 1% deviation from the optimal one very quick for thousands of cities

# Problems with Hill-Climbing (and local search in general)

- Evaluation function may look like:



- Remedies for local optima
  - Random restart hill climbing
  - Random sideways moves escape from shoulders, loop on flat maxima

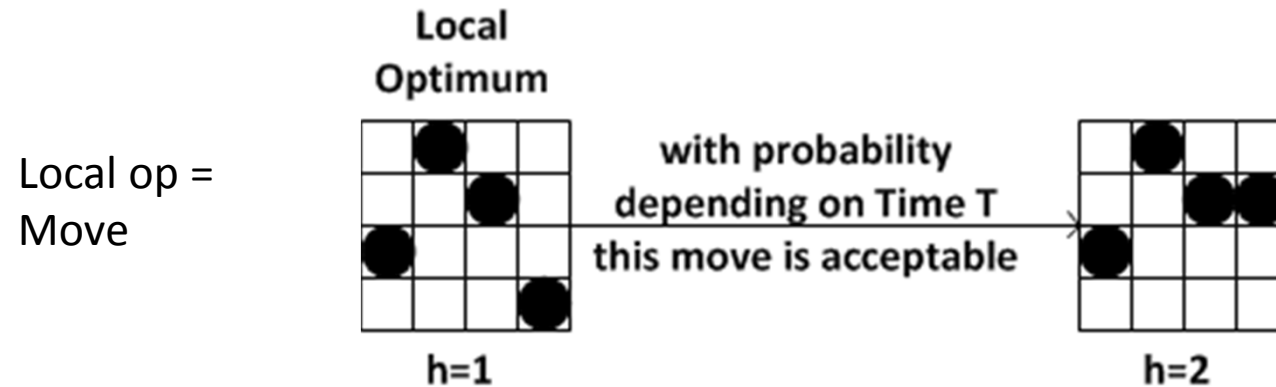
# Simulated Annealing

- Idea: escape local optima by allowing some "bad" moves but gradually decrease their size and frequency

```
function SIMULATED-ANNEALING (problem, schedule) returns a solution
  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  $\leftarrow$  MAKE-NODE(INITIAL-STATE[problem])
  for  $t \leftarrow 1$  to  $\infty$  do
    T  $\leftarrow$  schedule[t]
    if T=0 then return current
    next  $\leftarrow$  a randomly selected successor of current
     $\Delta E \leftarrow$  VALUE[next]-VALUE[current]
    if  $\Delta E > 0$  then current  $\leftarrow$  next
    else current  $\leftarrow$  next only with probability  $e^{\Delta E/T}$ 
```



# Illustration



- When  $T$  is hot, the solution can jump around quite freely; in the beginning almost random search  
→ also worse solutions can be accepted (how worse depends on  $T$ )
- When it becomes colder, the atom is forced to fit into the lattice more and more → only improvements are acceptable, e.g. almost greedy search

See: annealing processes in **physics**, e.g. a crystal lattice cools down from a soup in which atoms jump around to a fixed grid

# Genetic Algorithms

- Basic idea
  - ***Population-based*** Algorithm inspired by evolution
  - Solutions represented as ***Genomes***
  - *Evaluation* of genomes represent performance
  - The fittest individuals (best solutions) *survive* and create *offspring* (slightly different solutions)
  - Neighbors (“offspring”) generated using genetic operators such as mutation or cross-over
- Typical problems
  - Optimization problems , circuit layout, electric design ...
- Several variants
  - Genetic programming (each genome is a functional program)
  - Evolutionary Algorithms

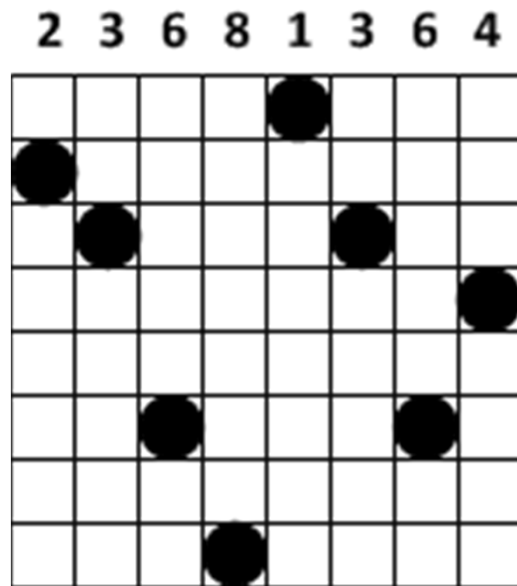
# GA: Basic Algorithm

- **function** GENETIC-ALGORITHM (*problem, schedule*) **returns** a solution  
  **inputs** *problem*, a problem  
  *populationSize, noGenerations*  
  **local variables:** *currentPopulation; selectedPopulation*  
  *currentPopulation*  $\leftarrow$   
    MAKE-POPULATION(INITIAL-STATE[*problem*], *populationSize*)  
  **for**  $t \leftarrow 1$  **to** *noGenerations* **do**  
    evaluate all members of *currentPopulation*  
    *selectedPopulation*  $\leftarrow$  selection (*currentPopulation*)  
    *currentPopulation*  $\leftarrow$  crossover (*selectedPopulation*)  
    *currentPopulation*  $\leftarrow$  mutate (*currentPopulation*)  
  **return** best Individuum(*currentPopulation*)

# Representation of Individual Solutions

- Assumption: a potential solution to the problem can be represented as **string of values**.
- Original GA: Binary values, but also other alphabets can be used
- Example: Search for the  $x, y, z$  for which the function  $f(x, y, z)$  returns max. value:
  - A solution is a triple with values for  $x$ ,  $y$  and  $z$ . For example using binary coding:  
 $x = 0110101101101011$   
 $y = 1000001011101110$   
 $z = 1100111100110110$
- Overall solution encoding:  
011010110110101110000010111011101100111100110110
- Fitness of the solution is  $f(x, y, z)$

# N-Queen Problem Individuum



- A queen per column, value is position in row
- Fitness value:  
Number of conflicts,  
The lower, the better

# Genetic Operators

- **Selection:** Driving force towards optimal solution
- Methods
  - Select the N fittest Individuals
  - Roulette Wheel Selection: Random selection biased by fitness
- Recombination of solutions by **Crossover**
  - For bigger steps in the search space
  - Simplest method:
    - Cut two strings AA and BB at the same position (crossover-point),
    - Generate two new individuals: AB and BA
- **Mutation** for bringing in new values
  - For local random steps in the search space
  - Simplest method:
    - Flip each letter of the genom string with a small probability

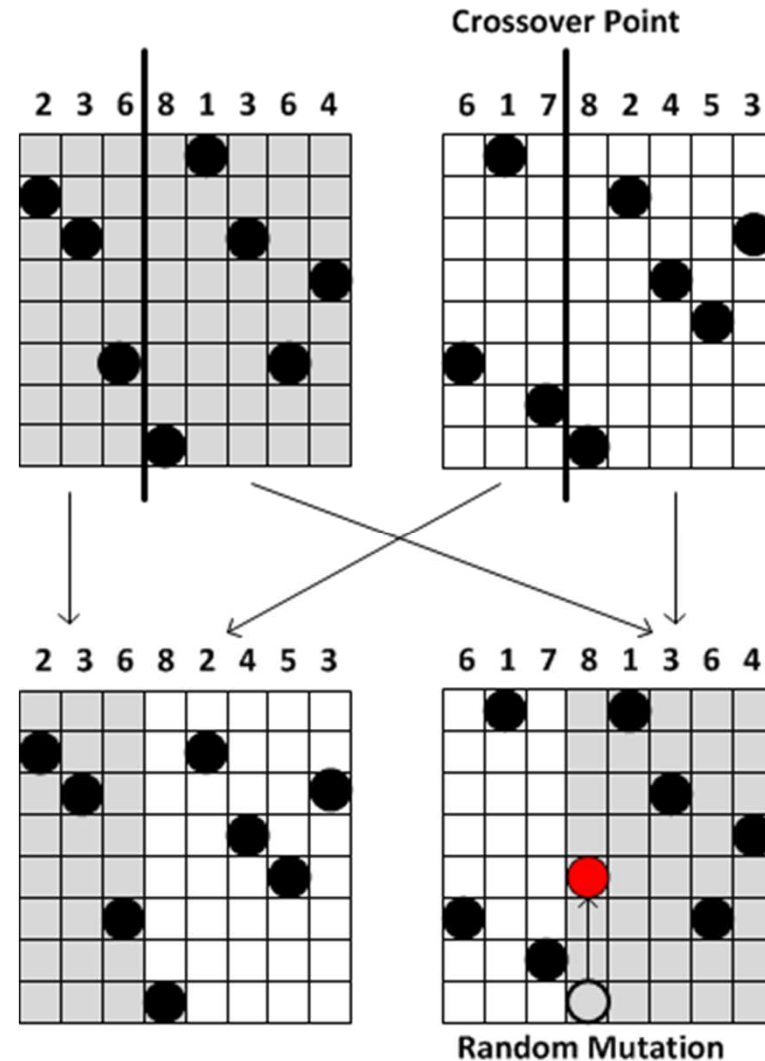
# Genetic Algorithm: Illustration

=



## ... in the N Queens Problem

- One-Point **Crossover** of two parents produces two offsprings.
- All offsprings form the new population → new generation
- **Mutation** eventually moves one queen randomly in its column





# Genetic Algorithm Properties

- Representation of the problem states/instances = individuum critical: genome as strings must support operators
- Genome for genetic operations needs "phenotype" represents how the genome is used.
- Crossover helps only if substrings are meaningful components;
- Problems when recombination destroys solution quality.
- Operators are non-deterministic  
→ random selection of crossover point(s), mutation

# Traveling Salesman Problem

- Adhoc solution: Genome represents sequence of cities
- *Crossover*
  - Combining the list from each parent with one crossover point
    - risk missing some cities in child solution!
    - Risk of visiting same city twice
- *Mutation*
  - Changing permutation of list – ok?
- How can the *genome* code only valid solutions?

# Alternative Traveling Salesman

- Genome as a *list of priorities*
    - London 10, Paris 20, Örebro 5, Stockholm 8 means  
visit Paris → London → Stockholm → Örebro → Paris
    - Initial Solution: Random priorities for each of the cities
  - Mutation
    - Add/decrease priority of random city
  - Crossover
    - [(L,10), (P, 20), (O, 5), (S, 8)] and  
[(L, 30), (P, 20), (O, 11), (S, 3)], crossover point at 2:
      - [(L, 10), (P, 20), (O, 11), (S, 3)] and
      - [(L, 30), (P, 20), (O, 5), (S, 8)]
- *only valid solutions can be generated*

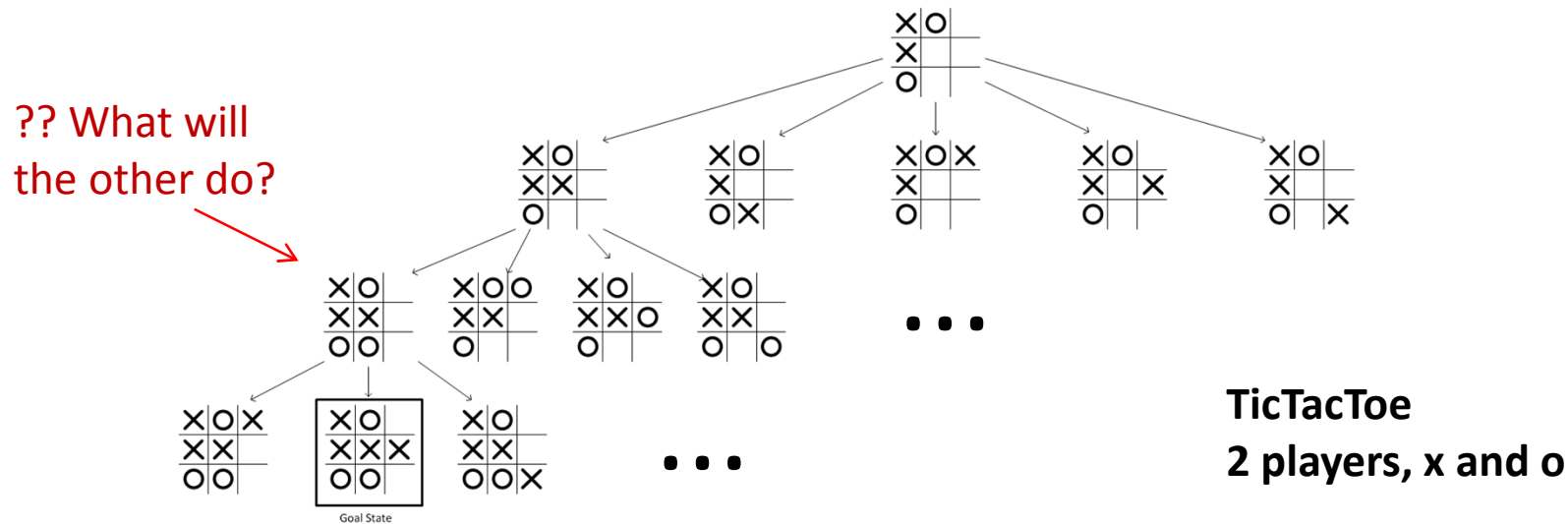
# More Genetic Algorithm Properties

- Minimum size of population needed, for preventing premature convergence of the population
  - Population-based: Evaluation of all solutions may be costly for large populations;
  - no guarantee for finding the optimal solution
  - GA is often chosen, if the rules governing the optimization problem are unknown; no standard solution is known
  - GA  $\neq$  Evolution! Real biology is much more complex
  - Instead of generation-based optimization, can also be embedded into simulation.
- other algorithms based on the example of natural evolution  
Artificial Life ( <http://www.karlsims.com/evolved-virtual-creatures.html> )



# Search for Games

- Many games can be solved by searching state space trees → Abstraction
- Deciding about next action by searching through all possible game positions

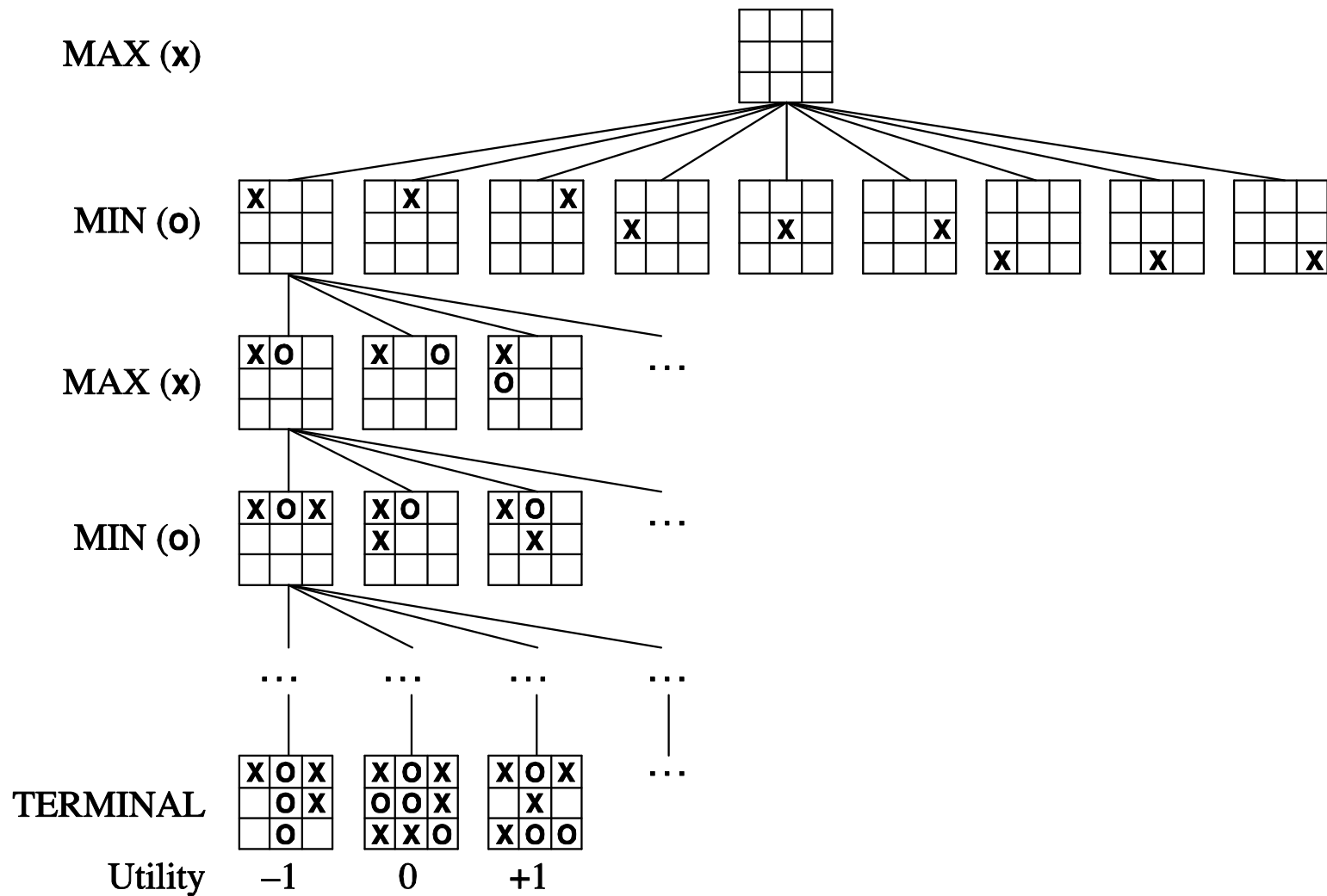


- **Problem: How can I decide which is the best action, unless I don't know how the other will act?**

# Game Search Tree for 2-Player Game

- Games with 2 players ("MAX", "MIN"), act alternately, utility values at the end are opposite
- Problem Formulation:
  - $S_0$ : initial state – initial game setup (board position and turn)
  - **PLAYER(s): which player has the move in state s**
  - ACTIONS(s), RESULT(s,a) as for standard search trees; defining legal moves
  - TERMINAL-TEST(s) is true, when game is over, states where the game has ended "terminal states"
  - **UTILITY(s) final numeric value for a game that ends in a terminal state: for example: -1 (loss), 0 (draw), 1 (win)**

# Game Search Tree in TicTackToe





# Optimal Decisions in Games

- The active player is always MAX and want to have:  
contingent strategy which specifies MAX move in the root/initial state and then Max's moves in the states resulting from every possible response by MIN, etc.
- Solution is based on node evaluation: assign **minimax value** to each node  $s$  assuming that both players play **optimally** from there to the end of the game:  
MAX will to move to the state of maximum value; MIN to minimum value:

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s), & \text{if } \text{TERMINAL-TEST}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if } \text{PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if } \text{PLAYER}(s) = \text{MIN} \end{cases}$$

MAX is always the player with the root  $\rightarrow$  "Me"

After each node is evaluated, MAX selects the action leading to the state with the best MINIMAX-value

# Minimax Algorithm Principle

- Algorithm:
  1. Generate game tree **completely**
  2. Determine utility of each **terminal** state
  3. Propagate the utility values upward in the tree by calculating the MINIMAX values for all nodes in the current level
  4. At the root node use MINIMAX values to select the move with the max (of the min) utility value
- Steps 2 and 3 in the algorithm assume that the opponent will play perfectly.

# Minimax Algorithm Pseudo Code

**function** MINIMAX-DECISION(*state*) *returns an action*

Return  $\arg \max_{a \in \text{ACTIONS}(\text{state})} \text{MIN-VALUE}(\text{RESULT}(\text{state}, a))$

---

**function** MAX-VALUE(*state*) *returns a utility value*

**If** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

$v \leftarrow -\infty$

**for each** *action* **in** ACTIONS(*state*) **do**

$v \leftarrow \max(v, \text{MIN-VALUE}(\text{RESULTS}(\text{state}, \text{action})))$

**return**  $v$

---

**function** MIN-VALUE(*state*) *returns a utility value*

**If** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

$v \leftarrow \infty$

**for each** *action* **in** ACTIONS(*state*) **do**

$v \leftarrow \min(v, \text{MAX-VALUE}(\text{RESULTS}(\text{state}, \text{action})))$

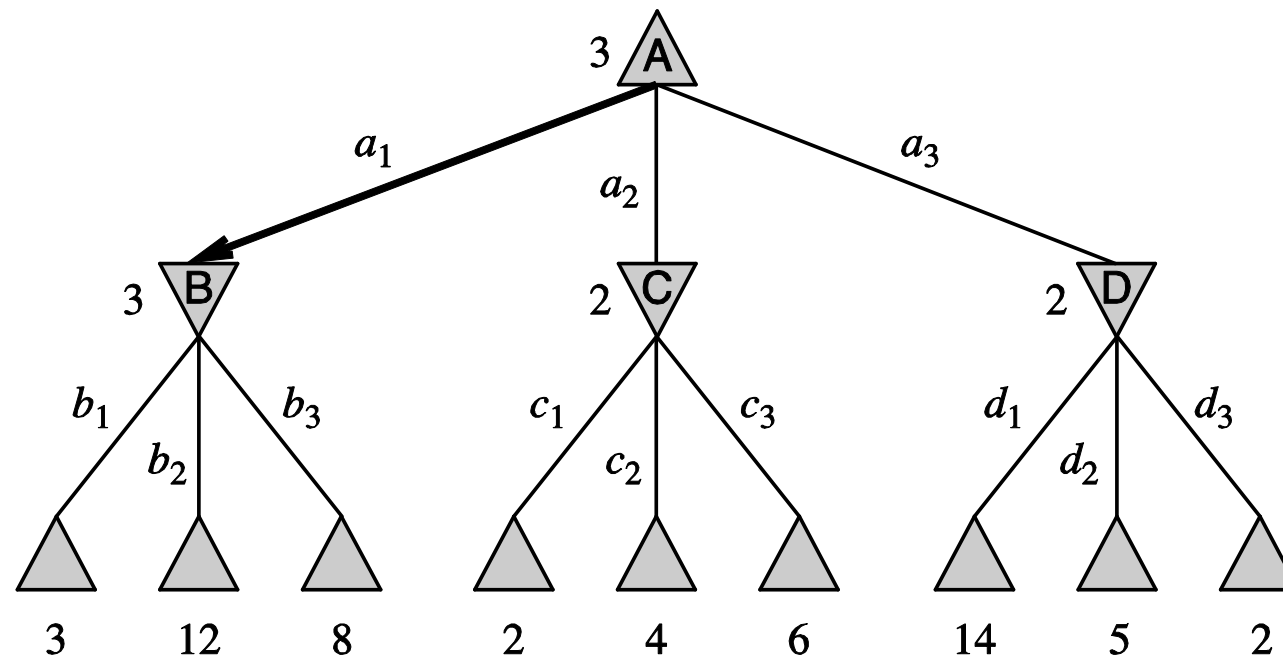
**return**  $v$

# Example

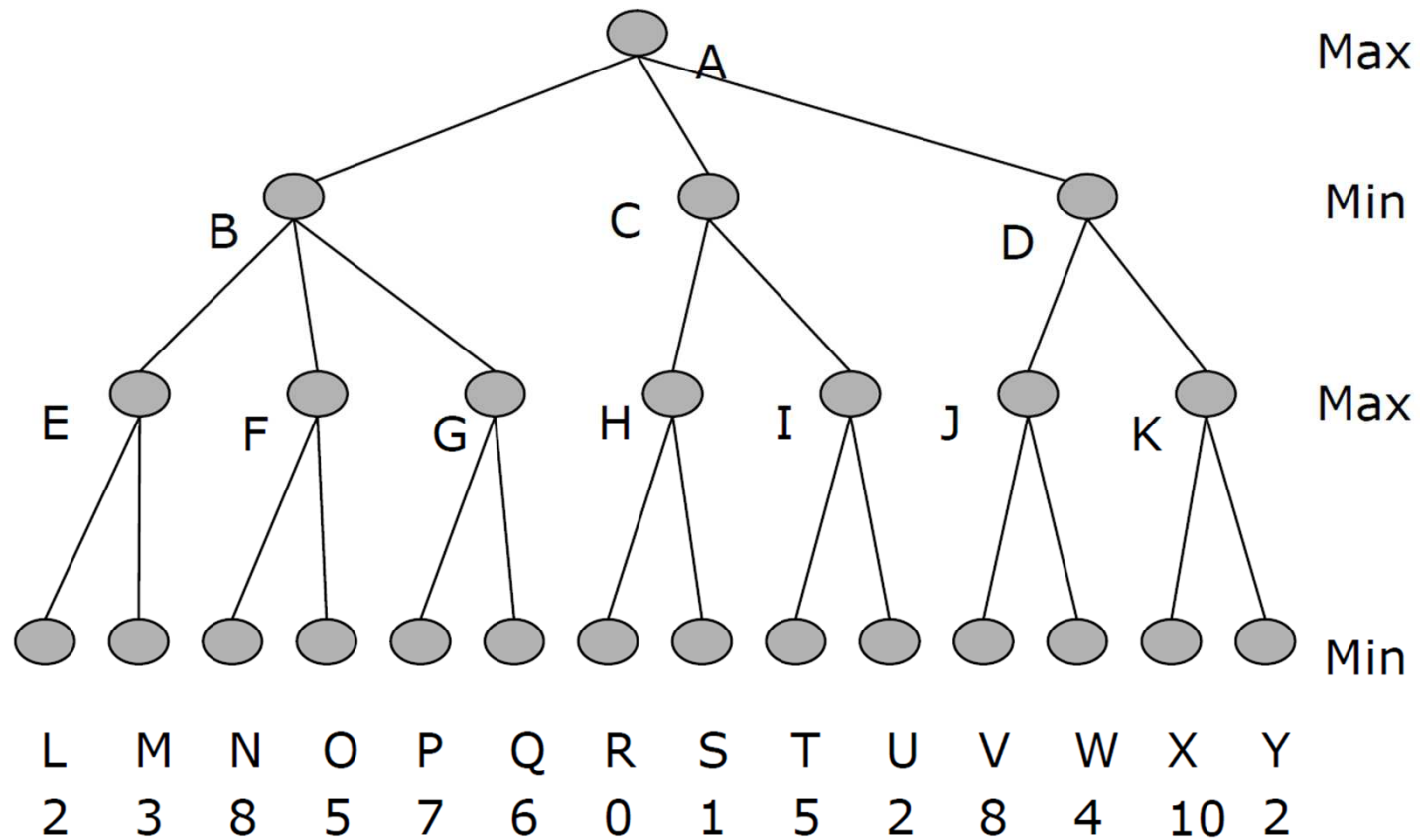
2 player game with 2 half-steps (2 "ply")

MAX

MIN



**An example for you:**  
**What action shall MAX select?**

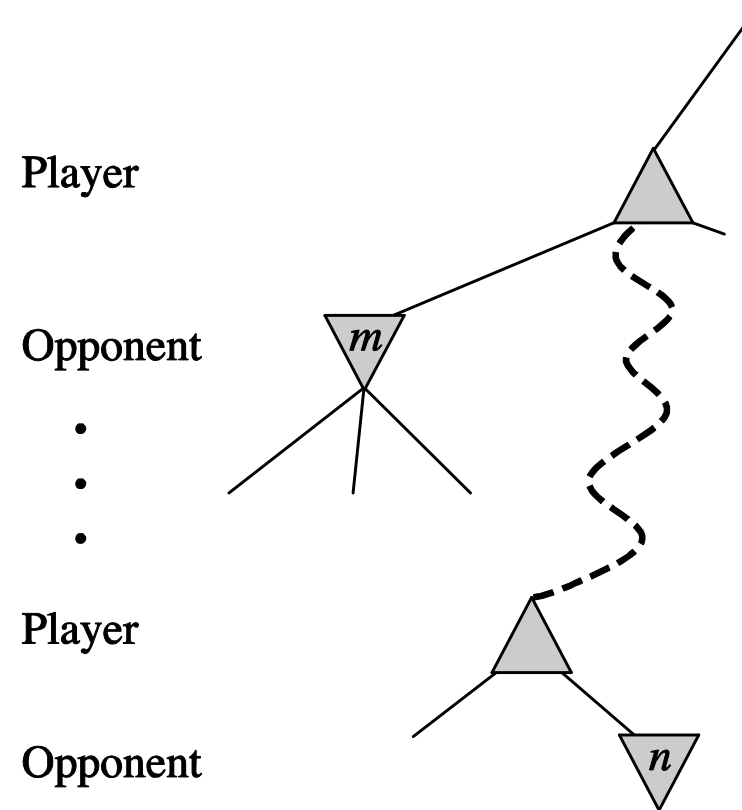


# Properties

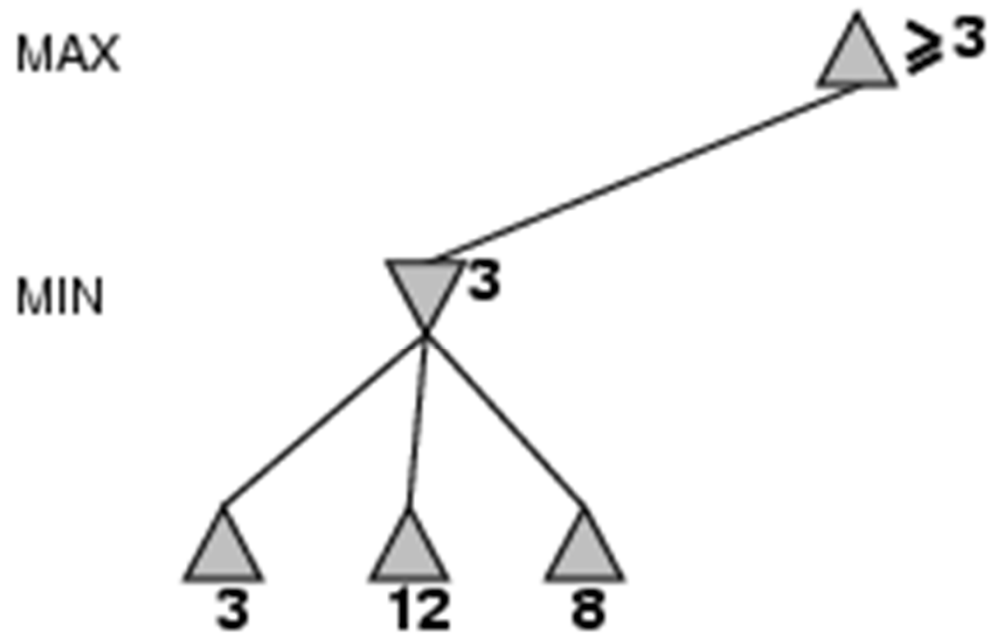
- **Completeness**, yes if the tree is finite
- **Optimal**: against an optimal opponent, otherwise?
- **Time complexity** needs to search through complete tree
- **Space complexity** like in depth-first exploration
- Directly **extendable** for games with more than 2 players
  
- for Chess branching factor  $\approx 35$ , maximum depth  $\approx 100 \rightarrow$  exact solution absolutely infeasible
  
- Idea: we do not need to explore every path  $\rightarrow$   **$\alpha$ - $\beta$ -Pruning**

# $\alpha$ - $\beta$ -Pruning

- Idea: If a player has a better choice **m** either at the parent node of **n** or at any choice point further up, then **n** will never be reached in an actual play
- $\alpha$  is the maximum lower bound of possible outcomes  
= **best choice for MAX so far**
- $\beta$  is the minimum upper bound of possible outcomes  
= **best choice for MIN so far**
- Value  $N$  of a node can only be:  
 $\alpha \leq N \leq \beta$
- $\alpha$ -pruning: search stops below a MIN node
- $\beta$ -pruning stops below a MAX node

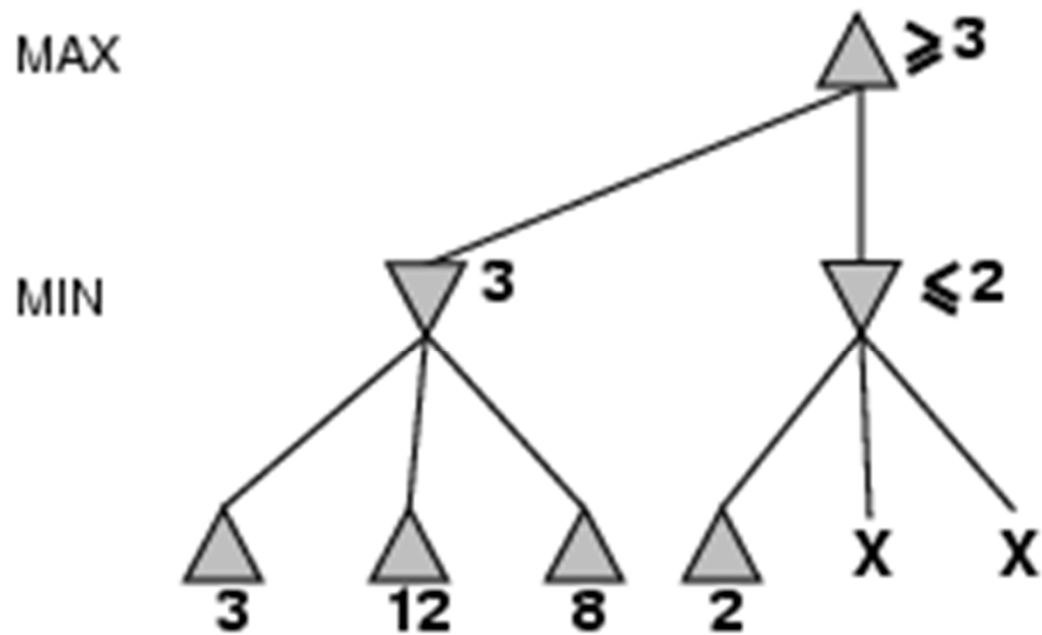


# $\alpha$ - $\beta$ -Pruning Example

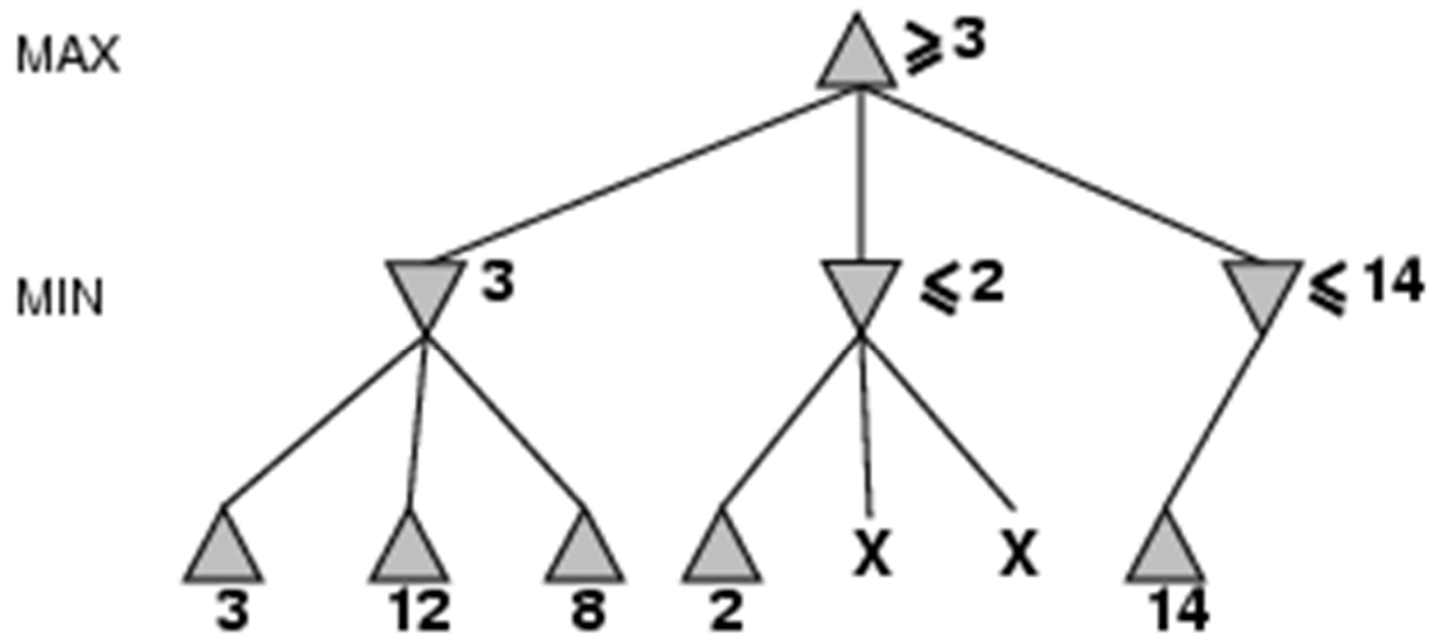




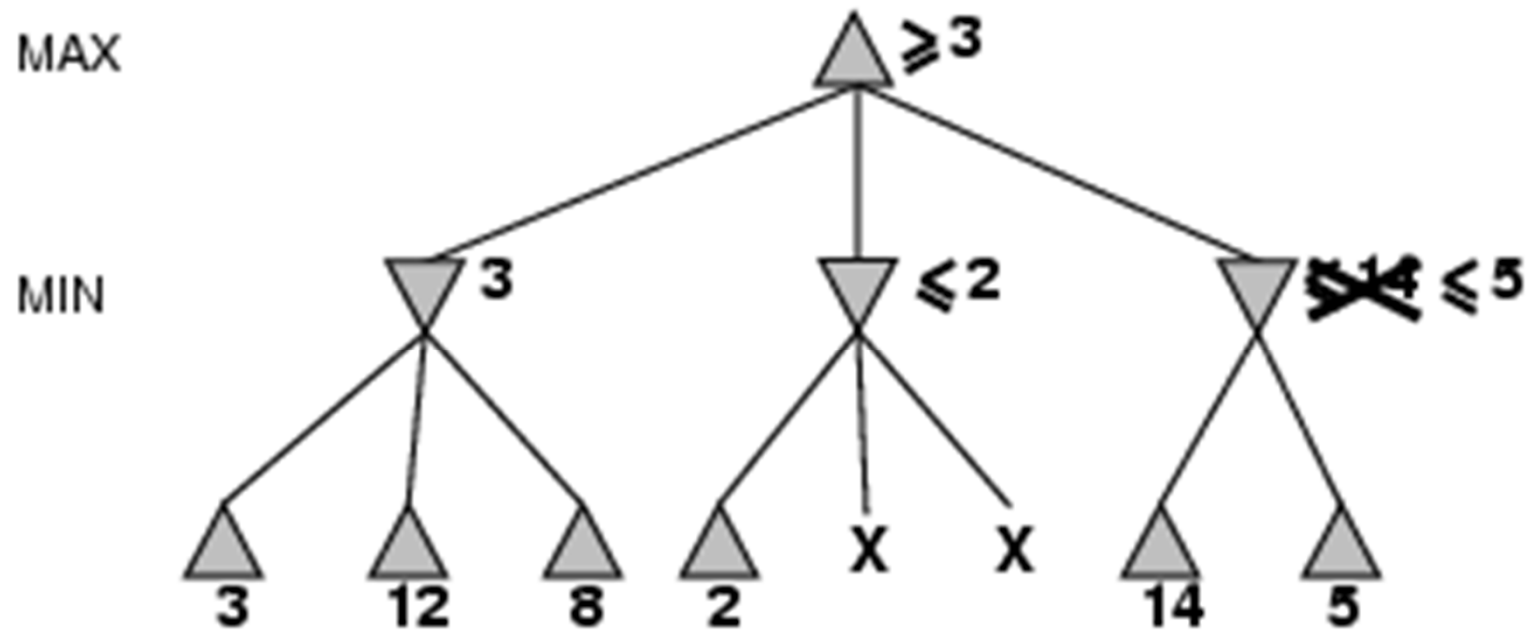
# $\alpha$ - $\beta$ -Pruning Example



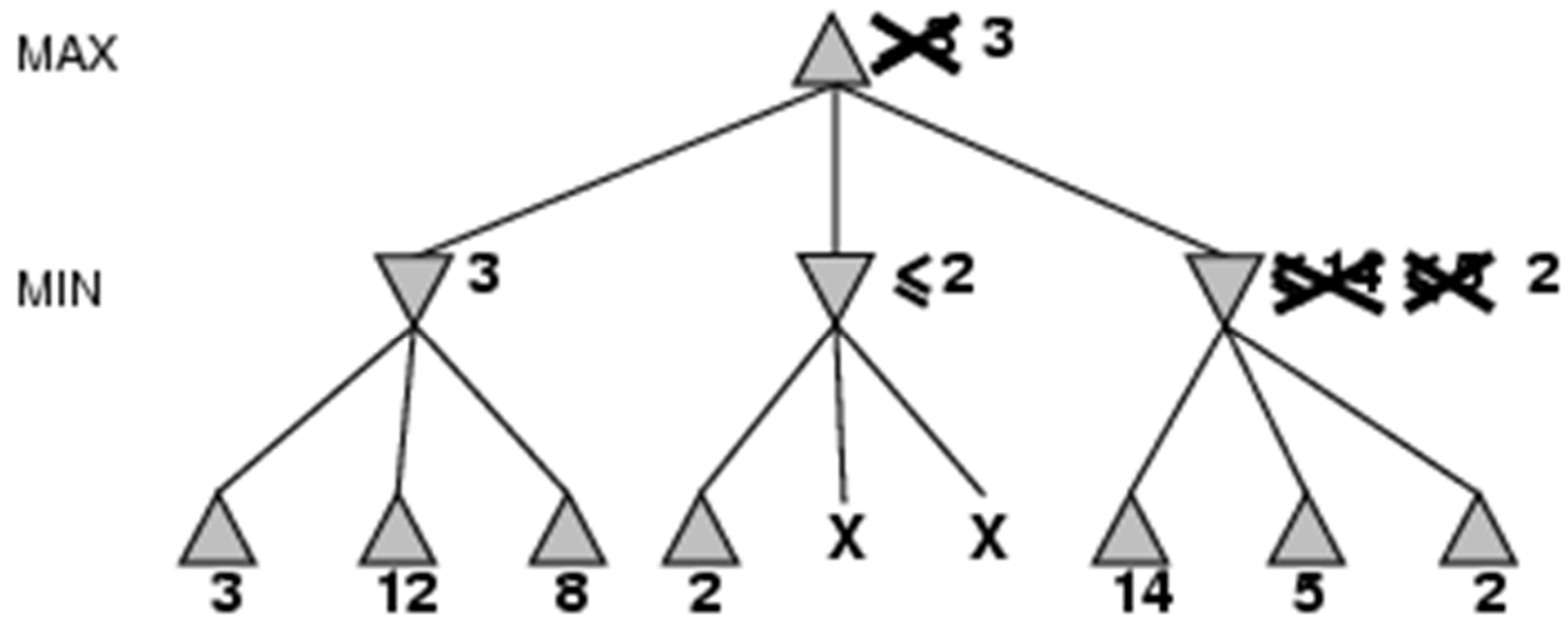
# $\alpha$ - $\beta$ -Pruning Example



# $\alpha$ - $\beta$ -Pruning Example



# $\alpha$ - $\beta$ -Pruning Example



# $\alpha$ - $\beta$ -Pruning Algorithm

**function** ALPHA-BETA-DECISION(*state*) *returns an action*

Return  $\arg \max_{a \in \text{ACTIONS}(\text{state})} \text{MIN-VALUE}(\text{RESULT}(\text{state}, a))$

---

**function** MAX-VALUE(*state*) *returns a utility value*

inputs: *state*, current state in game

$\alpha$ , value of best alternative for MAX along the path to *state*

$\beta$ , value of best alternative for MIN along the path to *state*

**If** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)

$v \leftarrow -\infty$

**for each** *action* **in** ACTIONS(*state*) **do**

$v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULTS}(\text{state}, \text{action})))$

**if**  $v \geq \beta$  **then return**  $v$

$\alpha \leftarrow \text{MAX}(\alpha, v)$

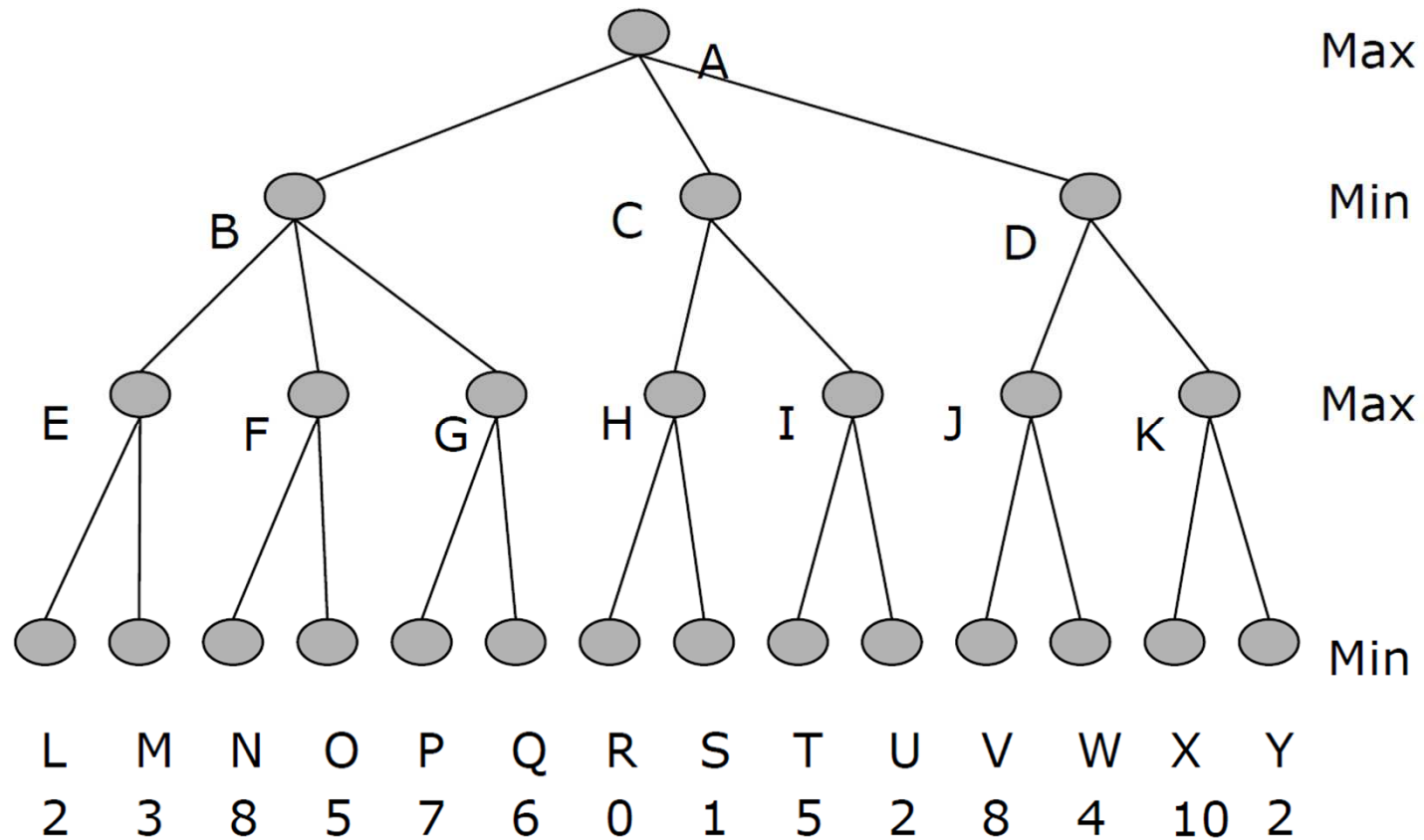
**return**  $v$

---

**function** MIN-VALUE(*state*) *returns a utility value*

Same as MAX-VALUE but with roles of  $\alpha, \beta$  reversed

# What can we prune?



# Properties of $\alpha$ - $\beta$ -Pruning

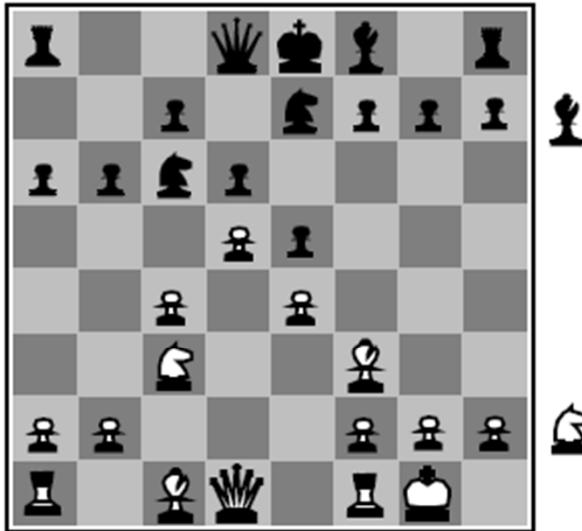
- Pruning does **not** affect final result
- In the optimal case, **doubles solvable depth**,
- but often search spaces are still **intractable**
- **Move ordering** essential  $\rightarrow$  Heuristics

# Ressource Limits

- Suppose in Chess:
  - 100 secs for deciding about next move
  - Computer can explore  $10^4$  nodes/sec
  - $10^6$  nodes per 100 sec  $\approx 35^{8/2}$
  - could do an  $\alpha$ - $\beta$  that reaches depth 8 (assuming a branching factor 35)
- Standard Approach
  - Instead of TERMINAL-TEST, use **CUTOFF-TEST**
    - **no definite evaluation, but heuristic**
  - Instead of UTILITY use **EVAL** - an evaluation function that estimates desirability of state (e.g. Position in chess); typically linear weighted sum of features
  - Cut off considering possible moves immediately (e.g. using statistics)

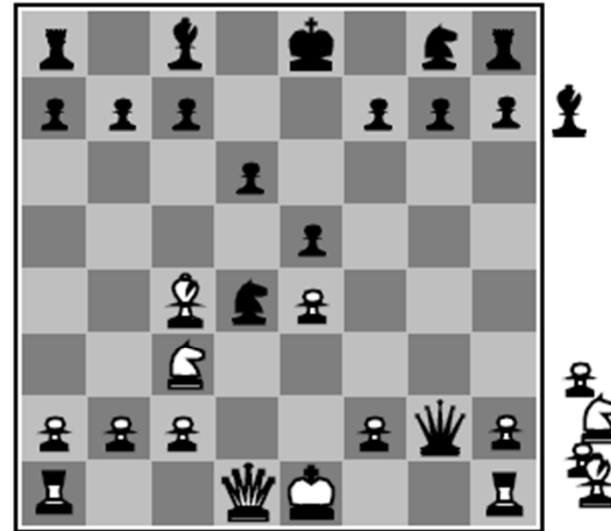


# Evaluation Functions



Black to move

White slightly better



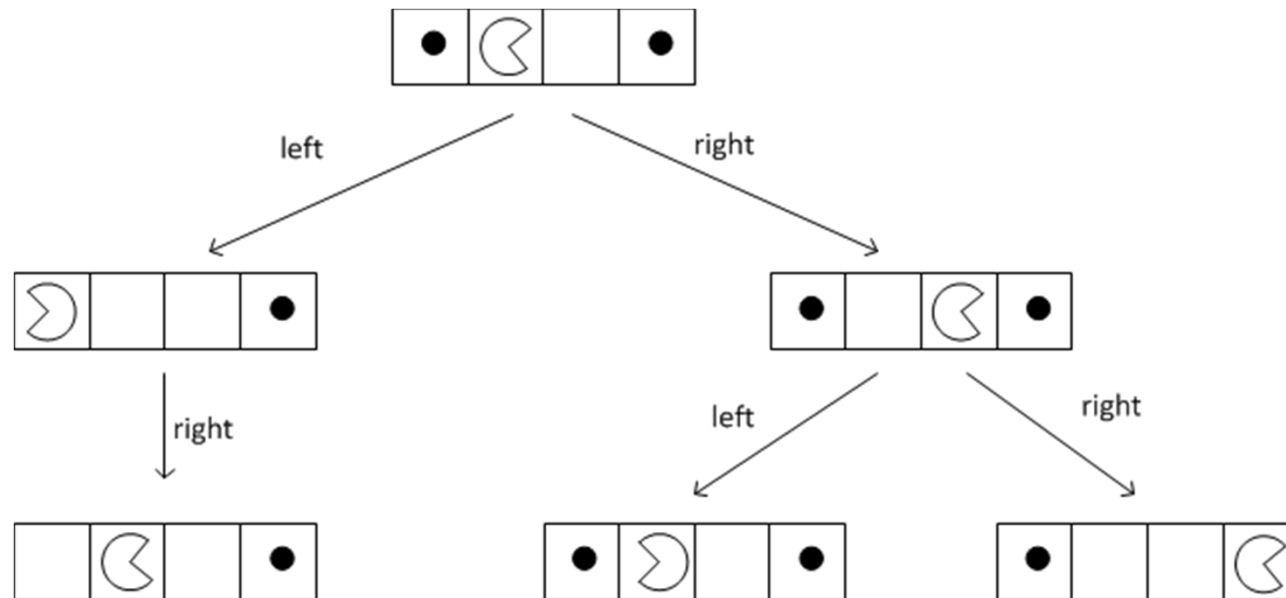
White to move

Black winning

- For Chess typically linear weighted sum of features:  
$$\text{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \dots$$
  
For example:  $w_1=9$ ;  $f_1(s)=\#(\text{white queens}) - \#(\text{black queens})$

# Horizon effect

- Depending on game and state evaluation heuristic, there is minimum depth which must be searched to get a good decision:



- If heuristic is number of eaten food items in the Pacman example (omitting a opponent) , what action to choose?

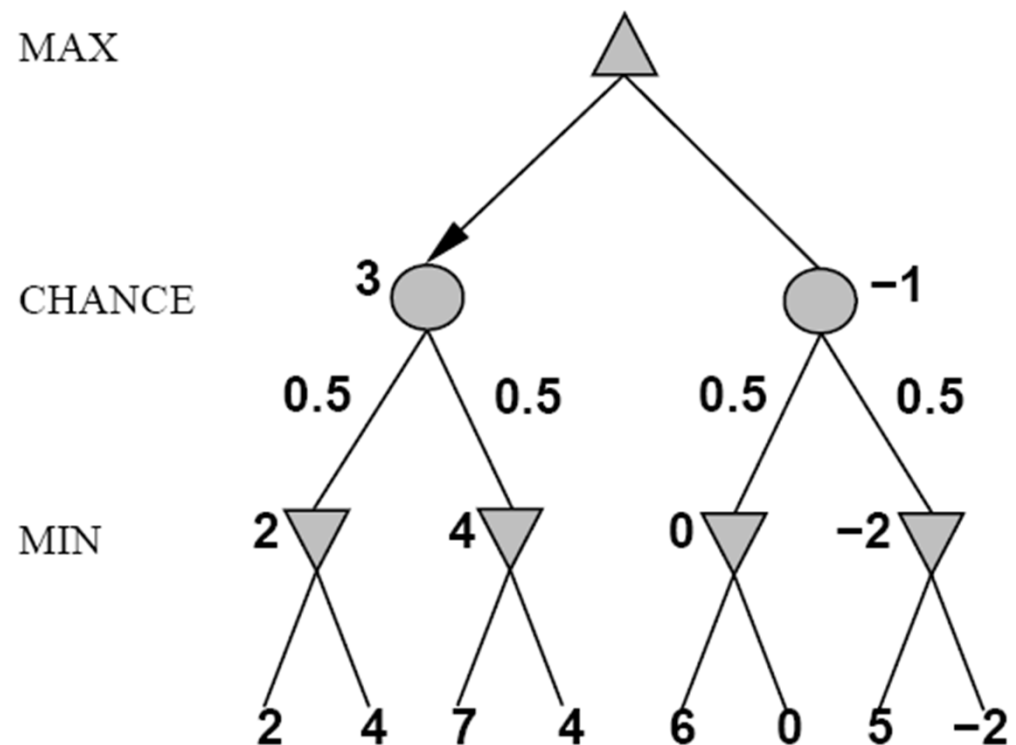
# Current State in (Deterministic) Games

- **Chess:** IBM's Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. It could search 200mio positions per second → depth 14; some interesting lines up to depth 40; EVAL function with >8000 features
- **Checkers:** CHINOOK (running on a regular PC, using  $\alpha$ - $\beta$  search), defeated the long-running human champion in an abbreviated match in 1990; Using a data base with  $\approx$  39Trillion end game positions
- **Othello** (Reversi) with  $\approx$  5-15 legal moves; in 1997 Logistrello defeated the world champion 6:0 → humans are no match for computers at Othello
- **Go;**  $b>300$ ; also definition of EVAL function difficult; top-programs avoid  $\alpha$ - $\beta$ , but use Monte Carlo Rollouts → advanced amateur level on full board; human champions refuse to compete against computers who are too bad.

# Non-deterministic Games

- For example: **Backgammon** as a combination of skill and chance
- "Chance" as **nondeterministic element comes from dices, card-shuffling,...**

→ Introduce random element as **extra node layer**, but how to handle them in the evaluation?



Example with coin-flipping

# MINIMAX → EXPECTIMINIMAX

- like MINIMAX, but explicitly handle chance nodes:

$$\text{EMINIMAX}(s) = \begin{cases} \text{UTILITY}(s), & \text{if } \text{TERMINAL-TEST}(s) \\ \max_{a \in \text{Actions}(s)} \text{EMINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{EMINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MIN} \\ \sum_R P(r) \cdot \text{EMINIMAX}(\text{RESULT}(s, r)) & \text{if } \text{PLAYER}(s) = \text{CHANCE} \end{cases}$$

- $P(r)$  is the probability of getting particular outcome: e.g. 1/6 when throwing a dice with 6 sides.
- But in practice:
  - 2 dices in Backgammon → Chance has 21 possible outcomes
  - as depth increases, probability of reaching a given node shrinks → value of lookahead is reduced →  $\alpha$ - $\beta$ -Pruning not so effective

# Current State in In-Deterministic Games

- **Backgammon:** as search depth is an expensive luxury, EVAL-functions are elaborated; G. Tesauro combined reinforcement learning with neural networks as evaluator with depth=2 or 3 → competitive with top human players
- **Bridge,** card game with imperfect information, multi-player game with teaming → programs are not winning championships, but are better than expected using a lot of AI techniques beyond search
- **Scrabble:** imperfect information and stochastic → top scoring move results in a good, not an expert player; nevertheless in 2006 QUACKLE defeated the former world champion 3-2

# Games and AI

- in 1965: A. Kronrod call chess the "drosophila" of AI (may be argued about, innovation in chess is not widely applicable) → better "Games are to AI what is the grand prix racing for automobile design"
- They illustrate several important points about AI
  - perfection is unattainable → must approximate
  - good idea to think about what to think about
  - uncertainty constrains the assignment of values to states
  - optimal decisions depend on information state, not real state