Foundational Knowledge for AI

Fredrik Heintz

Dept. of Computer Science
Linköping University

fredrik.heintz@liu.se

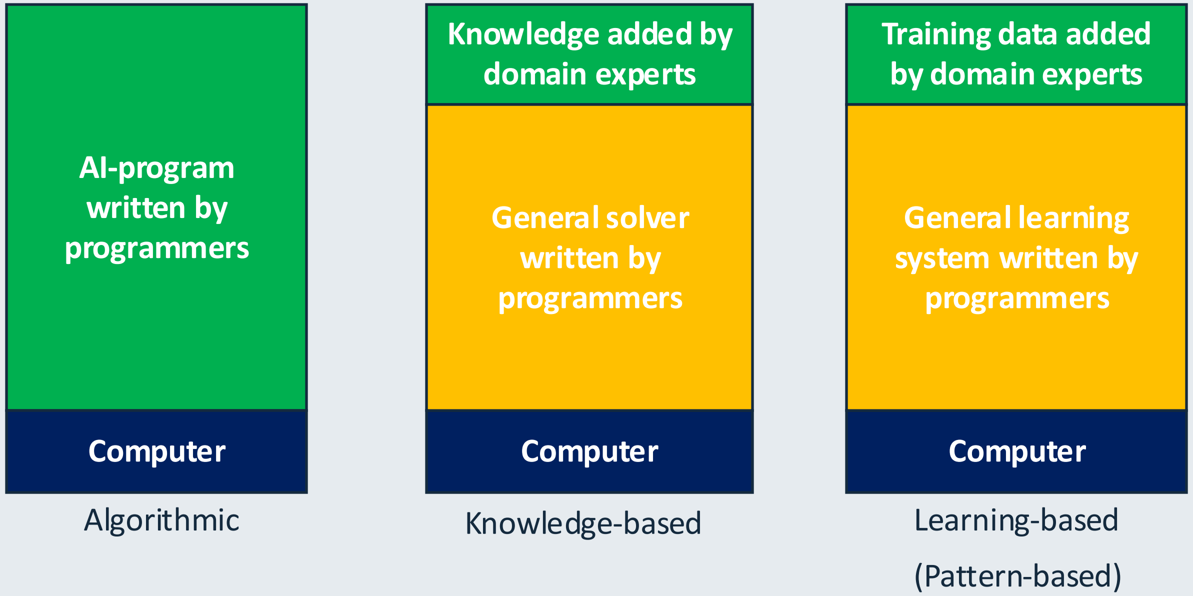
@FredrikHeintz

Outline:

- Uninformed Search Algorithms
- Informed Search Algorithms
- Search in Complex Environments
- Adversarial Search and Games

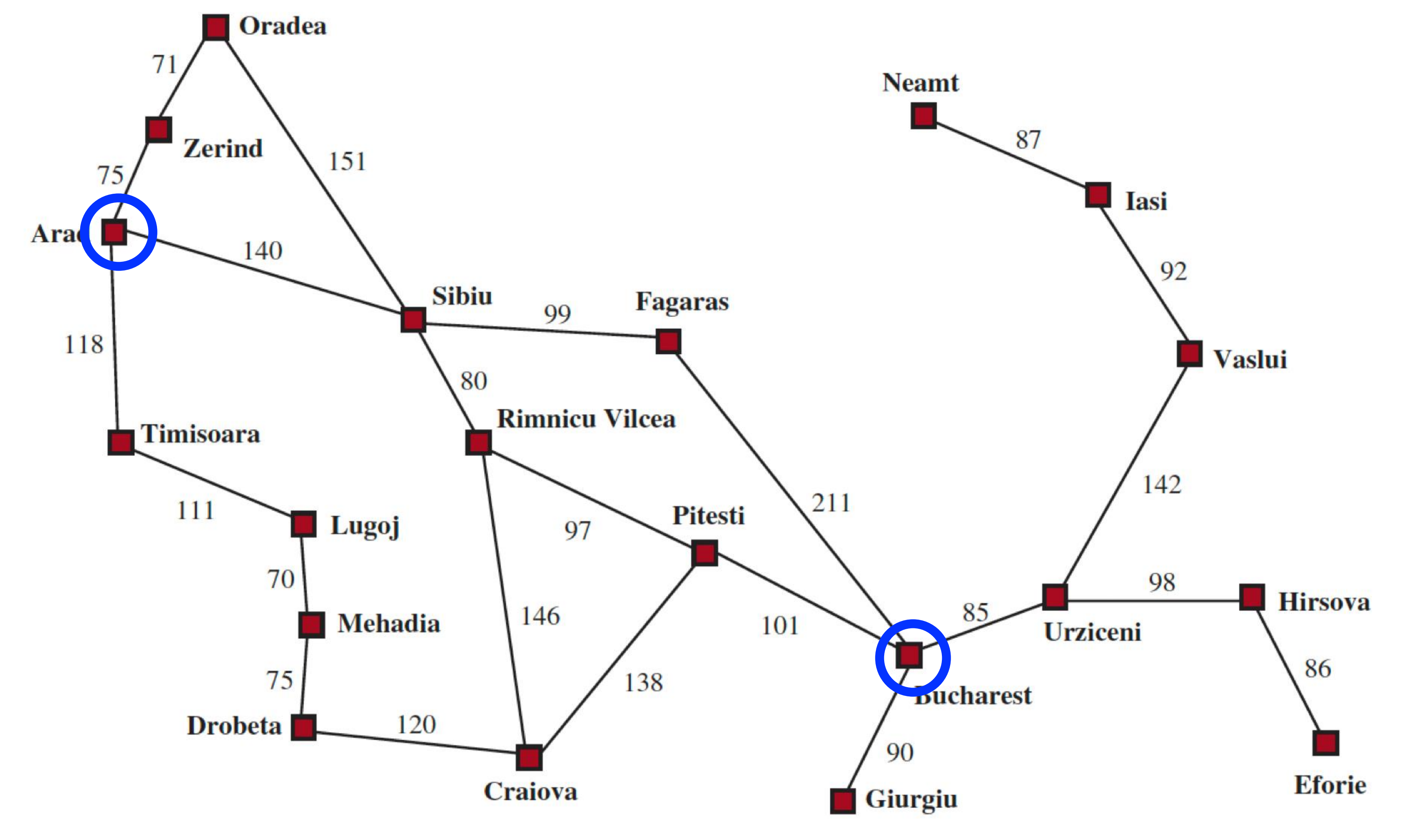
Co-funded by the European Union

Algorithmic, Knowledge-Based and Learning-Based AI



Co-funded by the European Union

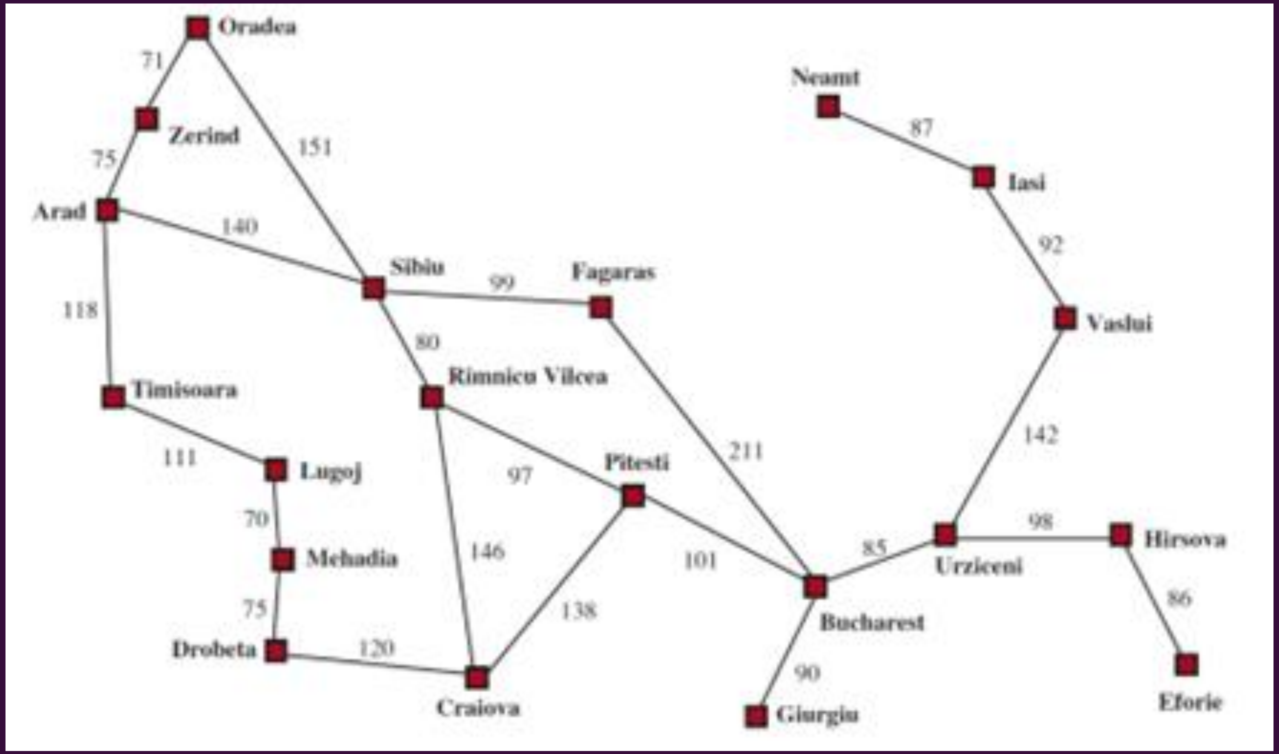
Romania Route Finding Problem



Co-funded by the European Union

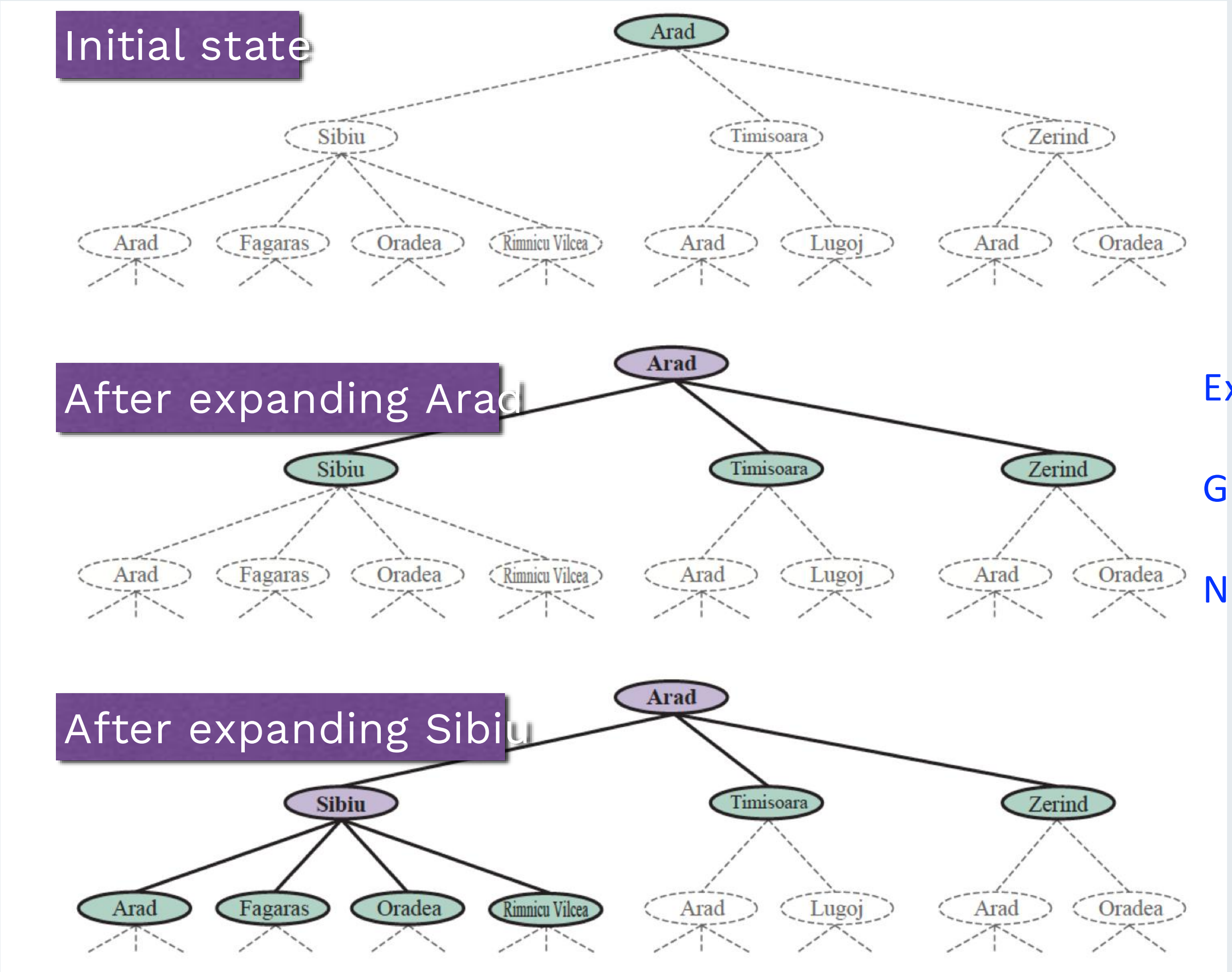
Problem Formulation

- State Space - A set of possible states that the environment can be in
  - Cities in the Romania map.
- Initial State - The state the agent starts in
  - Arad
- **ACTIONS**(*State*) - A description of what actions are applicable in each state.
  - **ACTIONS**(Arad)= {ToSibiu, ToTimisoara, ToZerind}
- **RESULT**(*State, Action*) - A description of what each action does (Transition Model)
  - **RESULT**(Arad, ToZerind)= Zerind
- Goal Test - Tests whether a given state is a goal
  - Often a set of states: { Bucharest }
- An Action Cost Function - denoted **ACTION-COST**(*s,a,s'*) when programming, or *c(s,a,s')* when doing math.
  - Gives the numeric cost of doing *a* in *s* to reach state *s'*.
  - Cost functions should reflect the agents performance measure: distance, time taken, etc.
- Solution - A path from the start state to the goal state



Co-funded by the European Union

3 Partial Search Trees: Route Finding



Expanded

Generated

Not Expanded

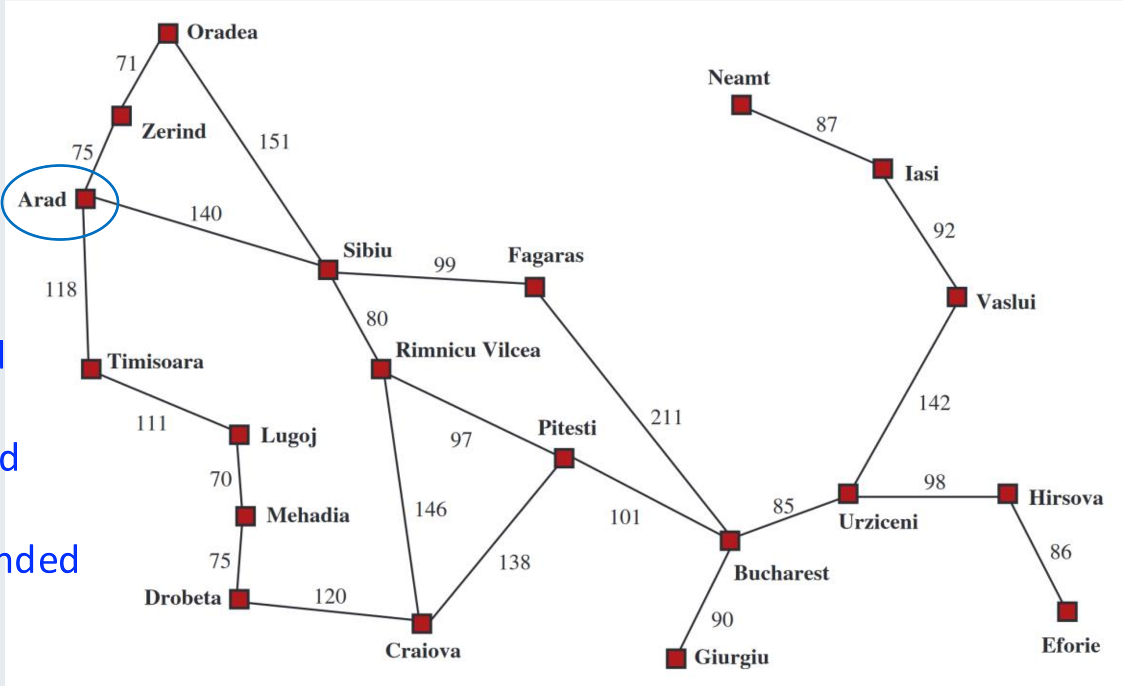
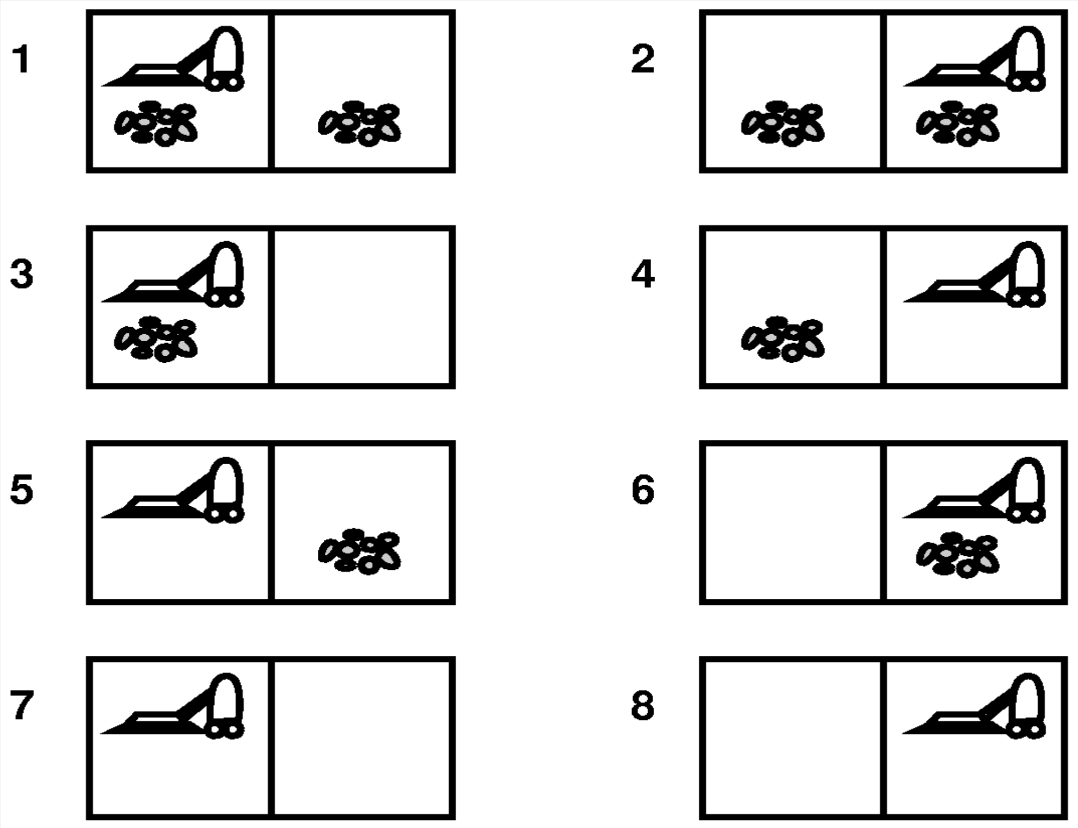


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

Co-funded by the European Union

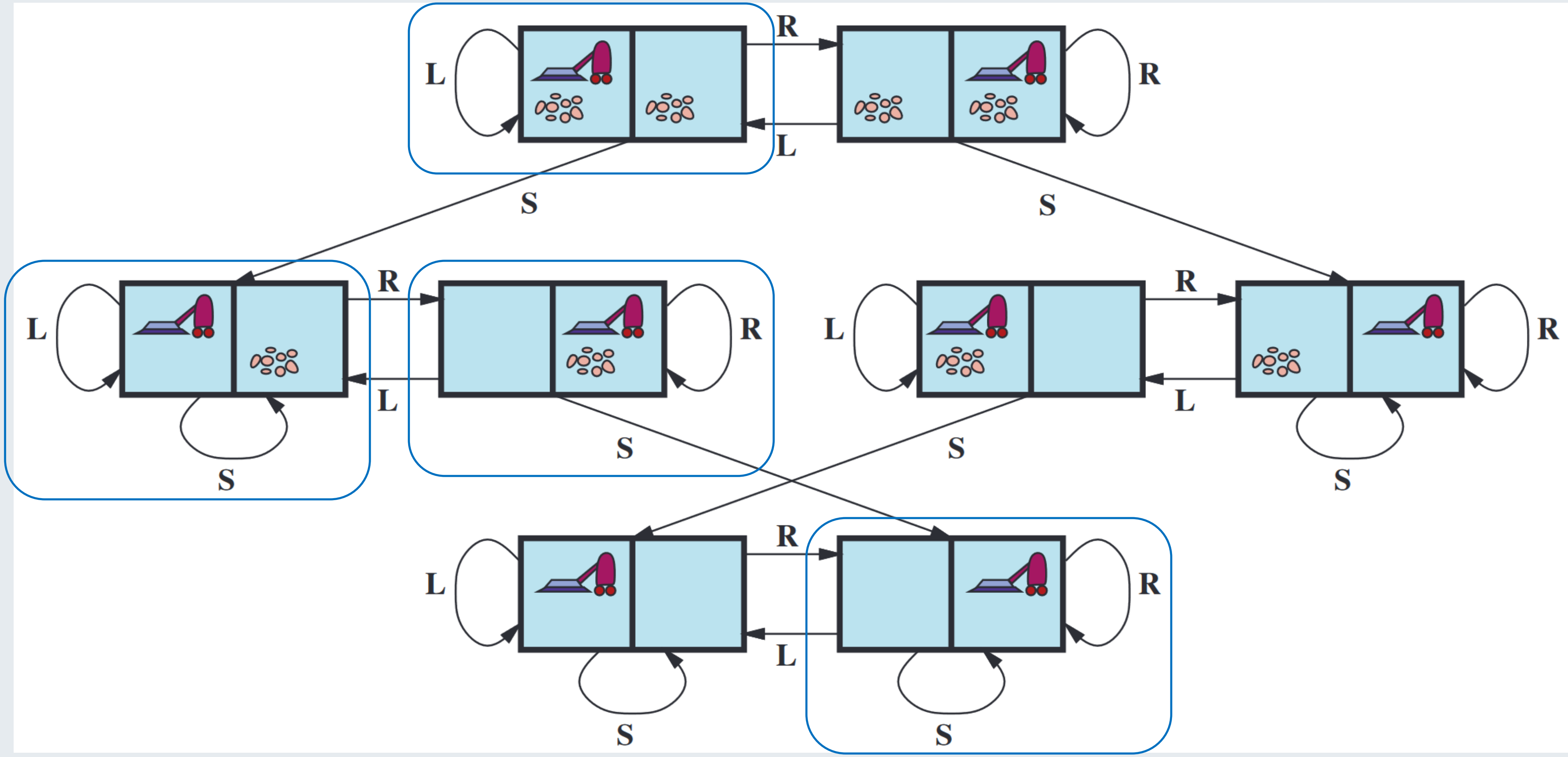
Problem Formulation: Vacuum cleaning world

- State Space:
  2 positions, dirt or no dirt
  -> 8 world states
- Actions:
  Left (L), Right (R), or Suck (S)
- Transition model: next slide
- Initial State: Choose.
- Goal States:
  States with no dirt in the rooms
- Action costs:
  one unit per action



Co-funded by the European Union

Solving the Vacuum World



Co-funded by
the European Union

Problem Formulation: Missionaries and Cannibals

Informal problem description:

- Three missionaries and three cannibals are on one side of a river that they wish to cross.
- A boat is available that can hold at most two people.
- You must never leave a group of missionaries outnumbered by cannibals on the same bank.
  - ➞ How should the state space be represented?
  - ➞ What is the initial state?
  - ➞ What is the goal state?
  - ➞ What are the actions?

Co-funded by the European Union

One Formalization

- Many other formalisations

State Space: triple (x,y,z) with 0 ≤ x,y,z ≤ 3, where x,y, and z represent the number of missionaries, cannibals and boats currently on the original bank.

Initial State: (3,3,1)

Actions: see transition model

Transition Model: from each state, either bring one missionary, one cannibal, two missionaries, two cannibals, or one of each type to the other bank.

Note: not all states are attainable (e.g., (0,0,1)), and some are illegal.

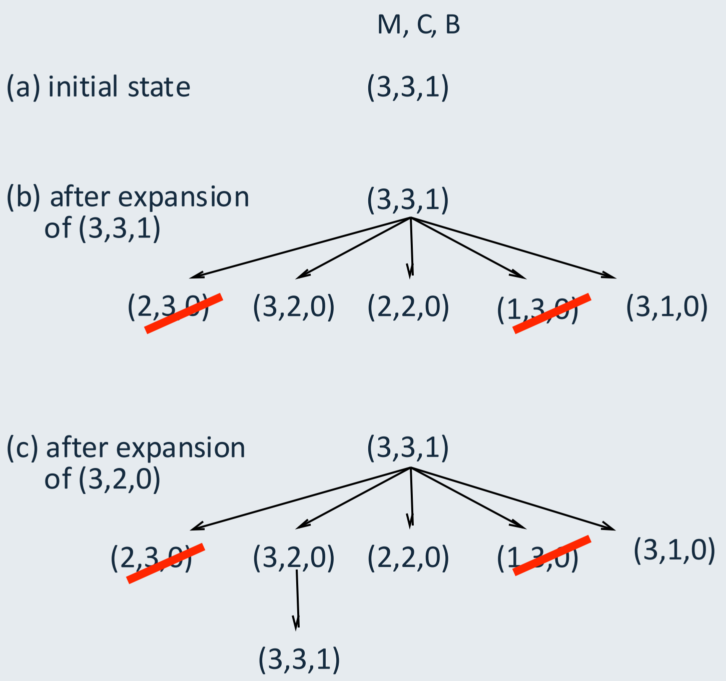
Goal States: {(0,0,0)}

Action Costs: 1 unit per crossing

Co-funded by the European Union

General Search

From the initial state, produce all successive states step by step —> search tree.



Co-funded by the European Union

Examples of Real-World Problems

- Route Planning, Shortest Path Problem
  - Routing video streams in computer networks, airline travel planning, military operations planning...
- Travelling Salesperson Problem (TSP)
  - A common prototype for NP-complete problems
- VLSI (integrated circuits) Layout
  - Another NP-complete problem
- Robot Navigation (with high degrees of freedom)
  - Difficulty increases quickly with the number of degrees of freedom. Further possible complications: errors of perception, unknown environments
- Assembly Sequencing
  - Planning of the assembly of complex objects (by robots)

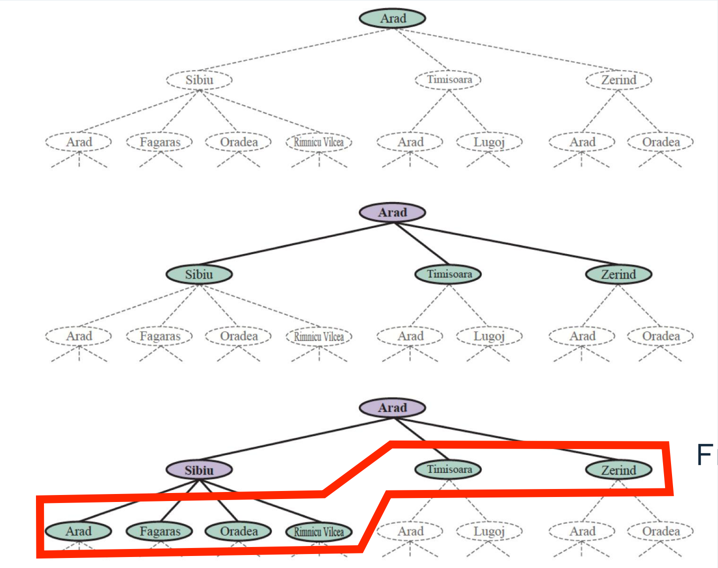
Co-funded by the European Union

Search Algorithms

We focus on algorithms that superimpose a <u>search tree</u> over the <u>state space graph</u>

Important distinction between search tree and state space!

Search tree Construction



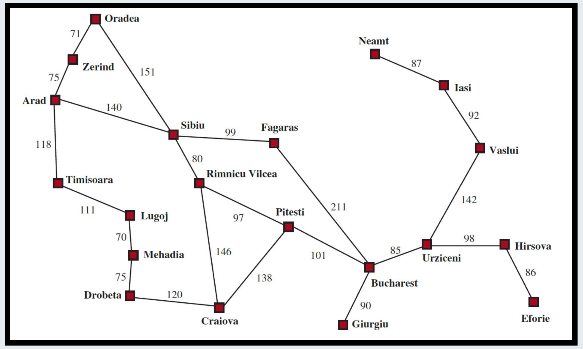
Expanded

Generated

Not Expanded

Frontier (choose from here)

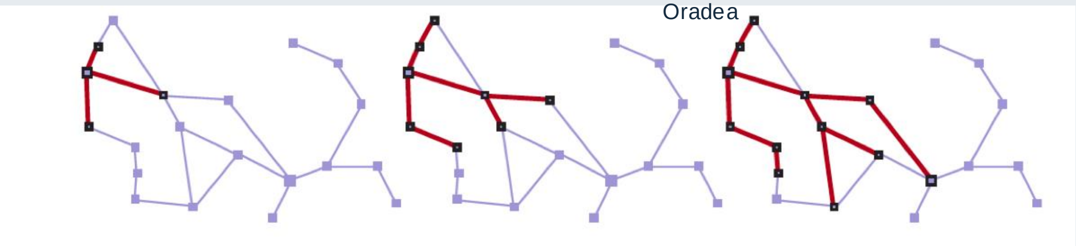
State Space Graph



Graph-Search: Only add a child if the state associated with it has not already been reached:

- Avoid cycles
- Avoid redundant paths

Oradea: has 2 successor states, but already reached by other paths, so do not expand.

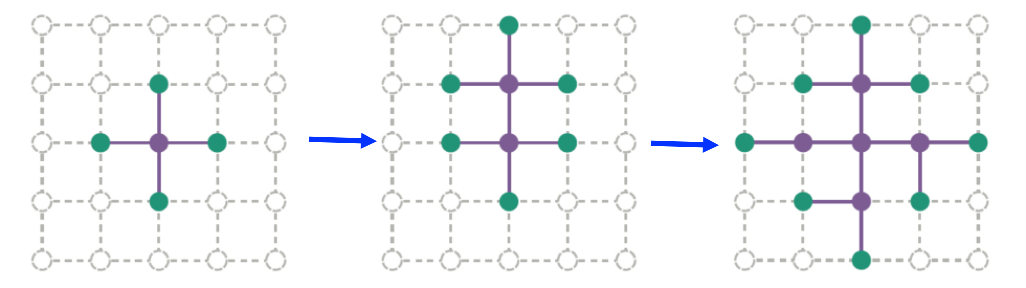


Sequence of search trees superimposed (each node only has one parent)

Co-funded by the European Union

Search Algorithms

Separation property of graph search



The frontier (green) separates the interior (lavender) form the exterior (faint dashed)

Any state with a node generated for it is said to be reached

What is the best way to search through a state space in a systematic manner in order to reach a goal state?

Co-funded by the European Union

Search Strategies

A *strategy* is defined by picking the order of node expansion.

Strategies can be *evaluated* along the following dimensions:

Completeness – does it find a solution if it exists?

Time Complexity – number of nodes generated/expanded

Space Complexity – maximum number of nodes in memory

Optimality – does it always find a least cost solution

Time and space complexity are *measured* in terms of:

b – *maximum branching* factor of search tree

d – *depth of the least cost solution* in the search tree

m – *maximum length* of any path in the state space (possibly infinite)

Co-funded by the European Union

Some Search Classes

- *Uninformed Search* (Blind Search)
  - No additional information about states besides that in the problem definition
  - Can only generate successors and compare against state.
  - Some examples:
    - Breadth-first search, Depth-first search, Iterative deepening DFS
- *Informed Search* (Heuristic Search)
  - Strategies have additional information as to whether non-goal states are more promising than others.
  - Some examples:
    - Greedy Best-First Search, A* Search

Co-funded by the European Union

Implementing the Search Tree

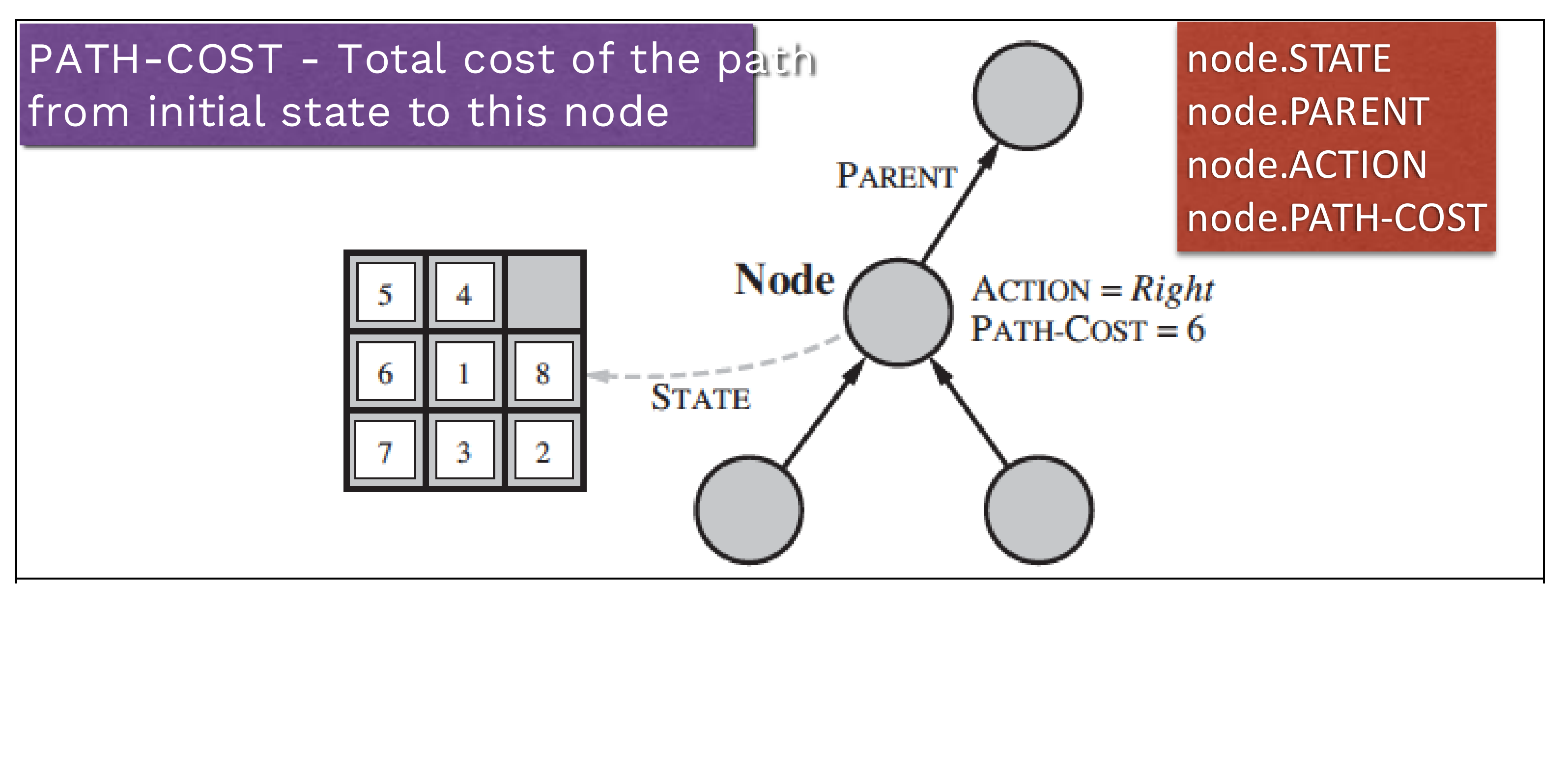
Three kinds of Queues:

- Priority Queue - min cost first
- FIFO Queue - first in, first out
- LIFO Queue - last in, first out

- Data structure for nodes in the search tree:
- STATE: state in the state space
- PARENT: Predecessor nodes
- ACTION: The operator that generated the node
- DEPTH: number of steps along the path from the initial state
- PATH-COST: Cost of the path from the initial state to the node
- Operations on a queue/frontier (4th Edition):
- IS-EMPTY(*frontier*): Empty test
- POP(*frontier*): Returns the first element of the queue
- TOP(*frontier*): Returns the first element
- ADD(*node, frontier*): Inserts new elements into the queue
- Operations on a queue (3rd Edition):
- Make-Queue(Elements): Creates a queue
- Empty?(Queue): Empty test
- First(Queue): Returns the first element of the queue
- Remove-First(Queue): Returns the first element
- Insert(Element, Queue): Inserts new elements into the queue
- (various possibilities)
- Insert-All(Elements, Queue): Inserts a set of elements into the queue

Co-funded by the European Union

States (in state space) and Nodes (in a search tree)



Can have several nodes with the same state due to multiple paths to the state

The search tree describes paths between states leading towards a goal

Co-funded by the European Union

Breadth-First Search

Assume all actions have the same cost

Search by Minimal Depth:

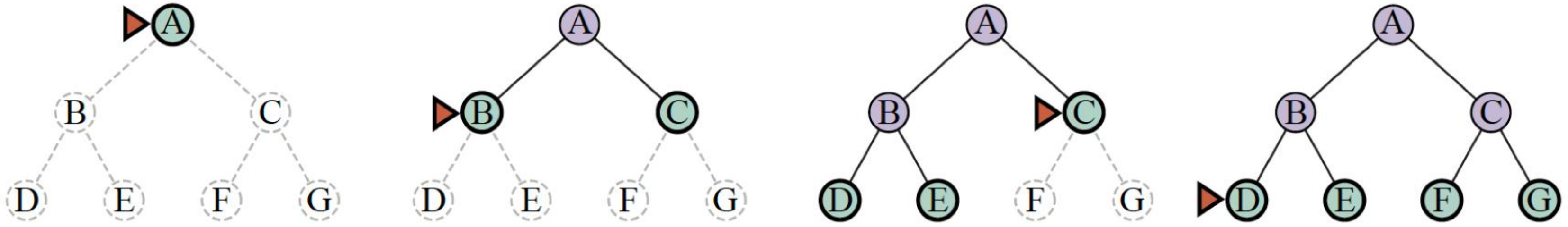


Figure 3.8 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by the triangular marker.

Could be implemented using Best-First-Search:

function BREADTH-FIRST-SEARCH *problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, DEPTH)

f(n) = n.DEPTH

Is a better way!

Co-funded by the European Union

Depth-First Search

Assume all actions have the same cost

Always expands
deepest node
in the frontier first

Usually implemented not as
graph search but as
tree-like search (without a
table of reached states)

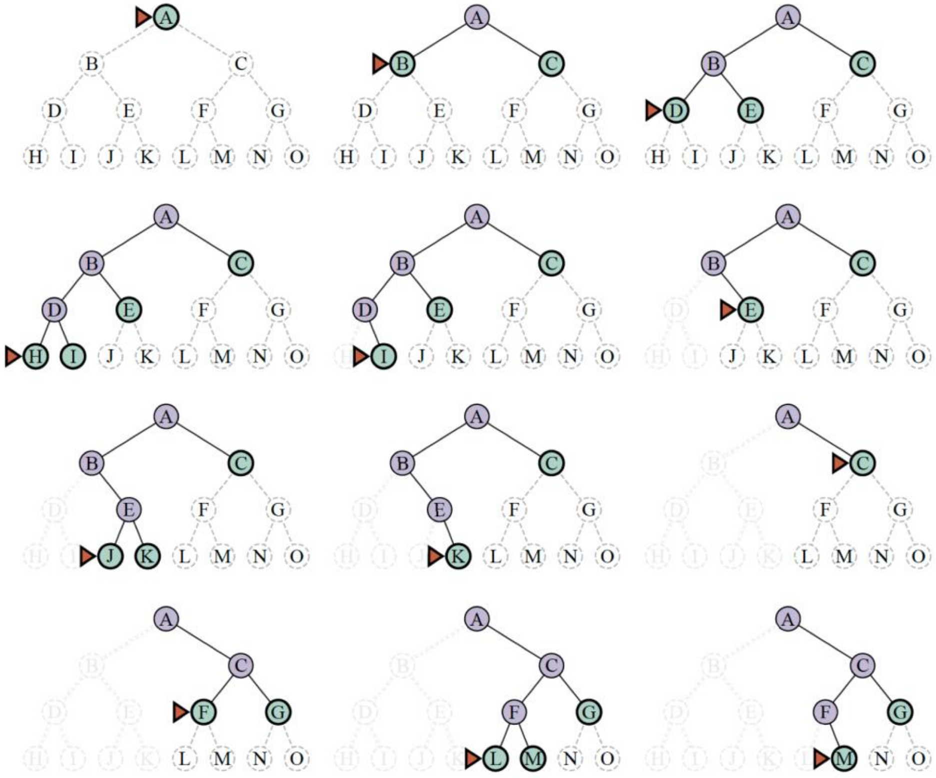


Figure 3.11 A dozen steps (left to right, top to bottom) in the progress of a depth-first search on a binary tree from start state A to goal M. The frontier is in green, with a triangle marking the node to be expanded next. Previously expanded nodes are lavender, and potential future nodes have faint dashed lines. Expanded nodes with no descendants in the frontier (very faint lines) can be discarded.

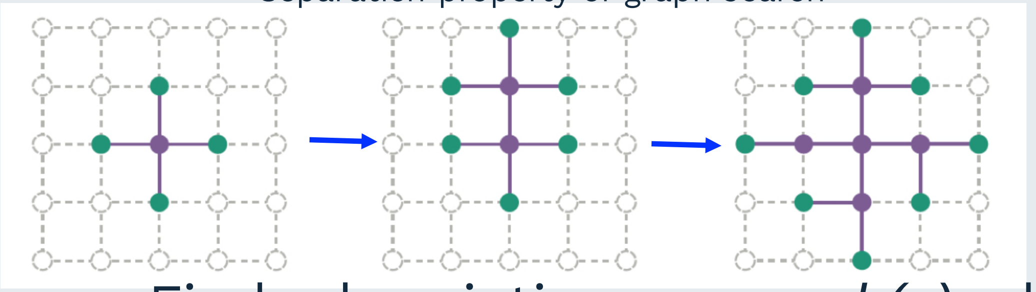
Co-funded by
the European Union

Heuristic Search

Co-funded by
the European Union

Intuitions behind Heuristic Search

Separation property of graph search



Systematic Search
through the state space

Find a heuristic measure *h(n)* which estimates how close
a node *n* in the frontier is to the nearest goal state and
then order the frontier queue accordingly relative to closeness.

The evaluation function *f(n)*, previously discussed will include *h(n)*:

$$f(n) = \ldots. + h(n)$$

h(n) is intended to provide domain specific hints about location of goals

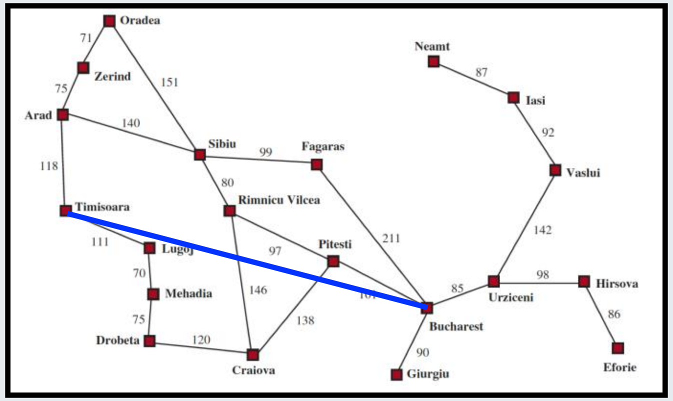
Co-funded by the European Union

Romania Travel Problem

Let's find a heuristic!

Straight line distance from city *n* to goal city *n'*

Assume the cost to get somewhere is a function of the distance traveled



Straight line distance to Bucharest from any city

$h_{SLD}()$

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Heuristic:

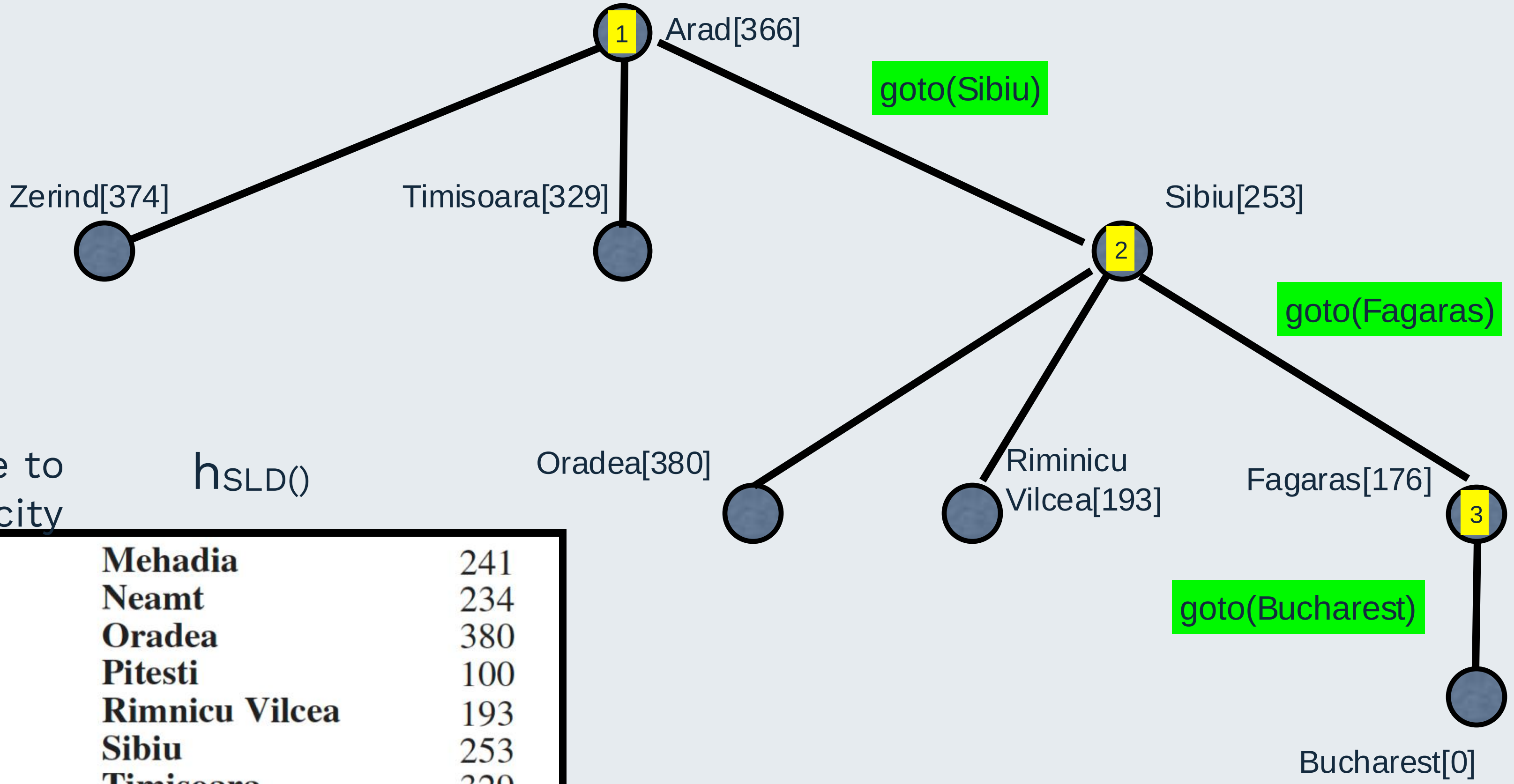
$$f(n) = h_{SLD}(n)$$

Notice the SLD under estimates the actual cost!

Co-funded by the European Union

Digital Business 4
Evolving your digital future

Greedy Best-First Search: Romania



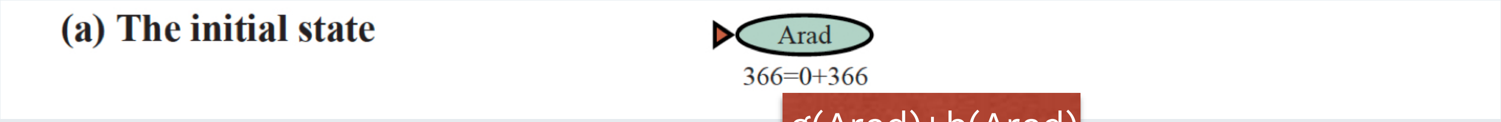
Straight line distance to Bucharest from any city

$h_{SLD}()$

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Co-funded by the European Union

A*-1

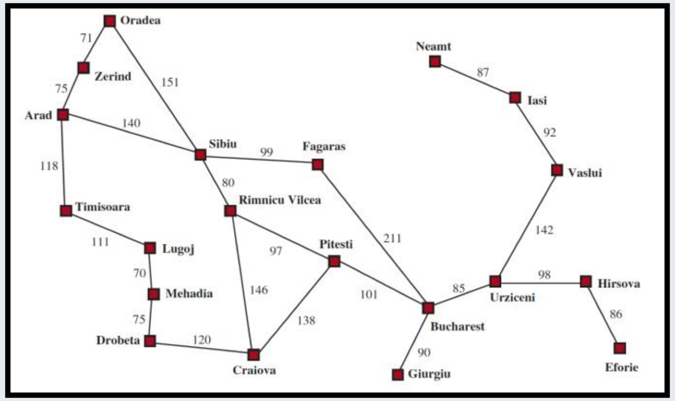


g(Arad)+h(Arad)

Heuristic (with Bucharest as goal):

f(n) = *g(n)* + *h(n)*
g(n) - Actual distance from root node to *n*
h(n) - $h_{SLD}(n)$ straight line distance from *n* to Bucharest

g(n)



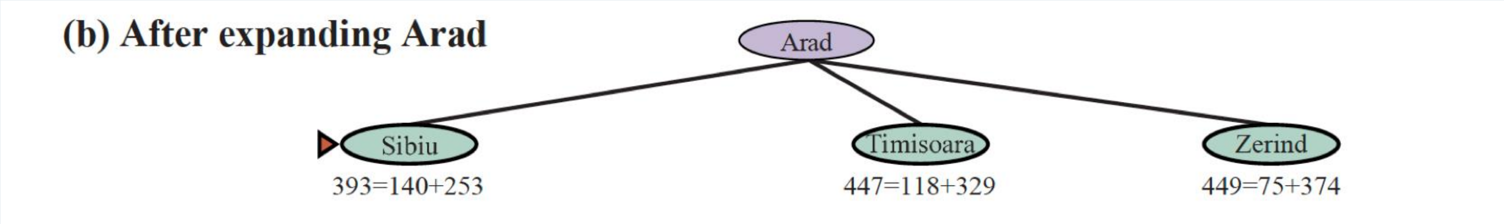
Straight line distance to Bucharest from any city

$h(n) = h_{SLD}(n)$

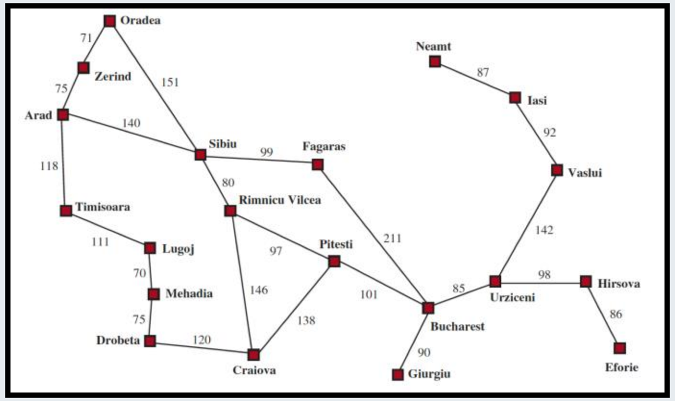
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Co-funded by the European Union

A*-2



g(n)



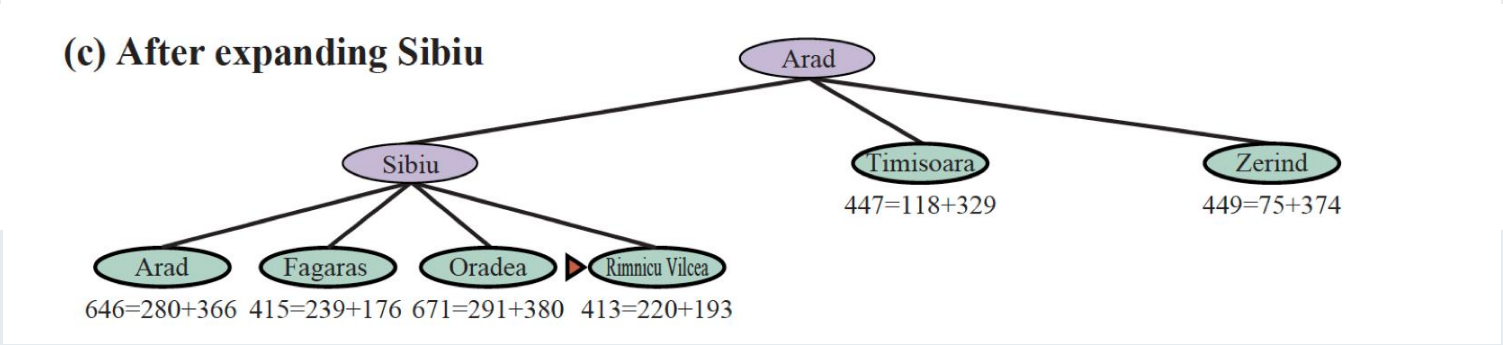
Straight line distance to Bucharest from any city

$h(n) = h_{SLD}(n)$

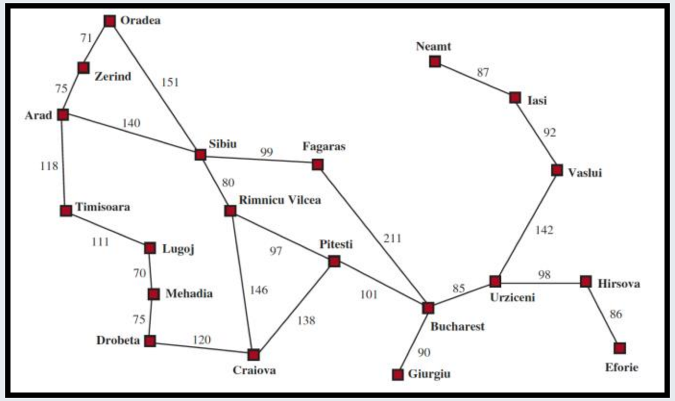
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Co-funded by the European Union

A*-3



g(n)



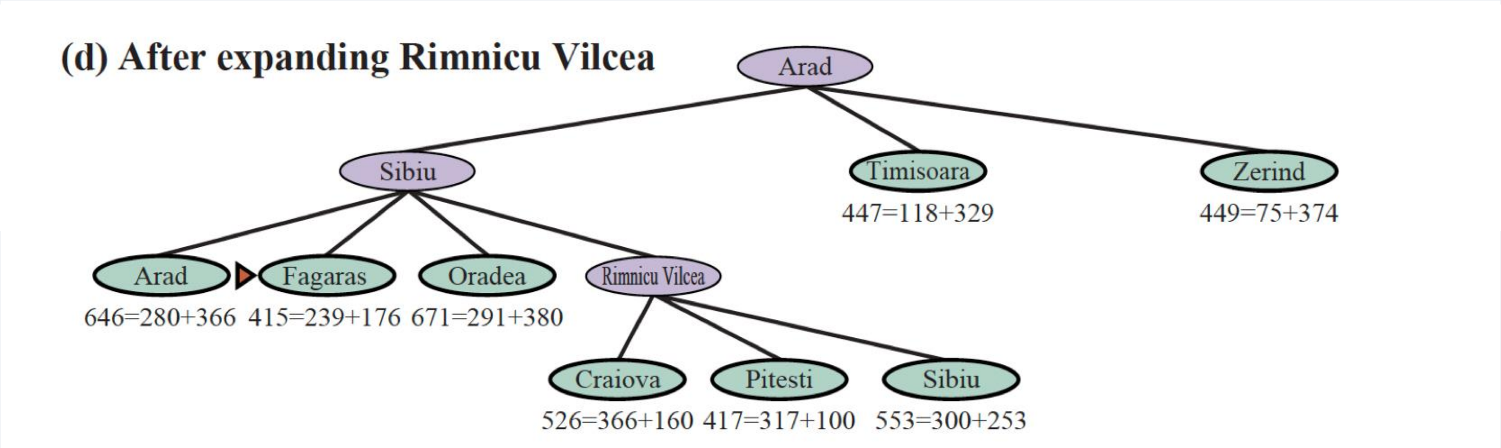
Straight line distance to Bucharest from any city

$h(n) = h_{SLD}(n)$

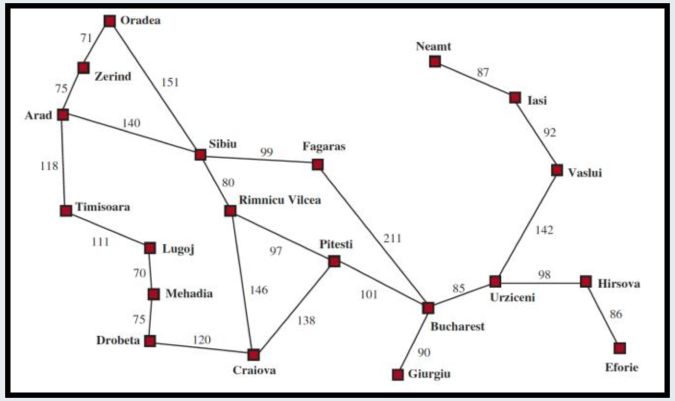
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Co-funded by the European Union

A*-4



g(n)



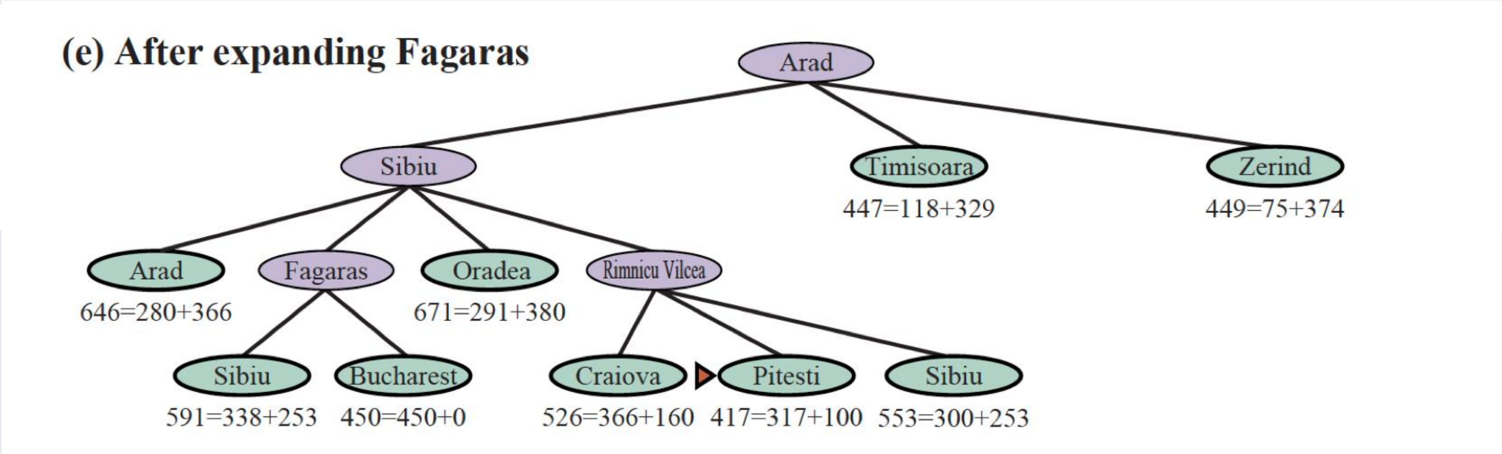
Straight line distance to Bucharest from any city

$h(n) = h_{SLD}(n)$

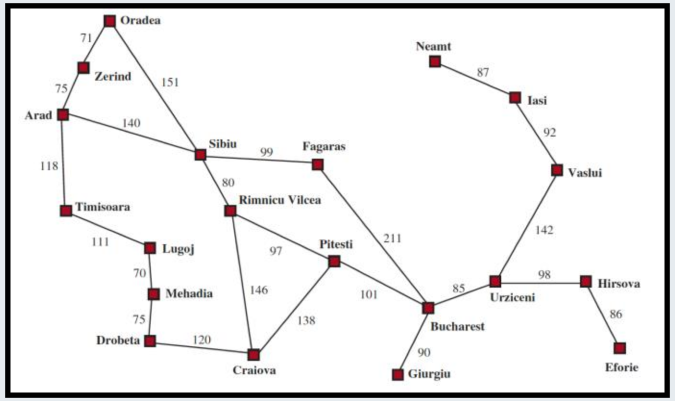
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Co-funded by the European Union

A*-4



g(n)

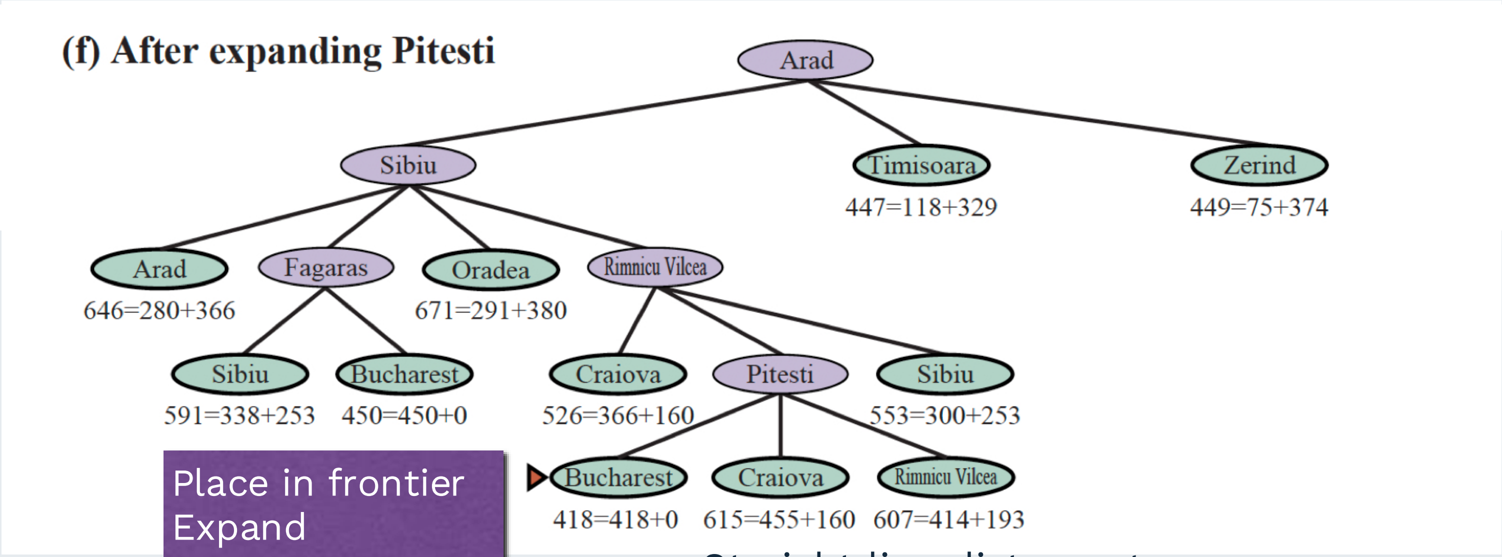


Straight line distance to Bucharest from any city

$h(n) = h_{SLD}(n)$

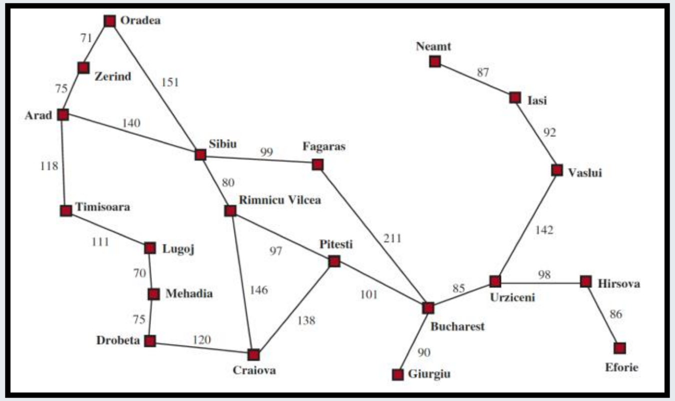
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

A*-6



Place in frontier
Expand
Late Testing Goal

g(n)



Straight line distance to Bucharest from any city

h(n) = $h_{SLD}(n)$

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Co-funded by
the European Union

Some properties of A*

- Cost-Optimal -
  - for a given admissible heuristic (tree-like search)
  - for a given consistent heuristic (tree-like, graph-search)
  - Consistent heuristics are admissible heuristics but not vice-versa.
- Complete - Eventually reach a contour equal to the path of the least-cost to the goal state.
- Optimally efficient - No other algorithm, that extends search paths from a root is guaranteed to expand fewer nodes than A* for a given heuristic function.
- The exponential growth for most practical heuristics will eventually overtake the computer (run out of memory)
  - The number of states within the goal contour is still exponential in the length of the solution.
  - There are variations of A* that bound memory....

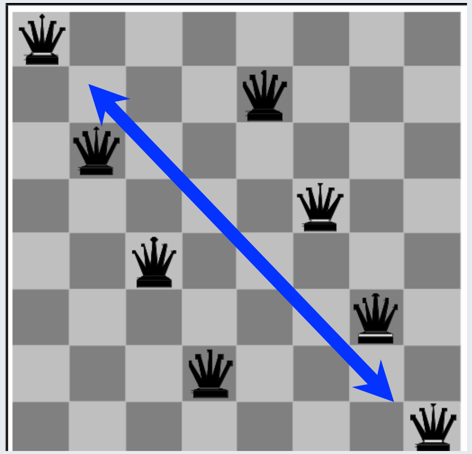
Co-funded by the European Union

Search in Complex Environments

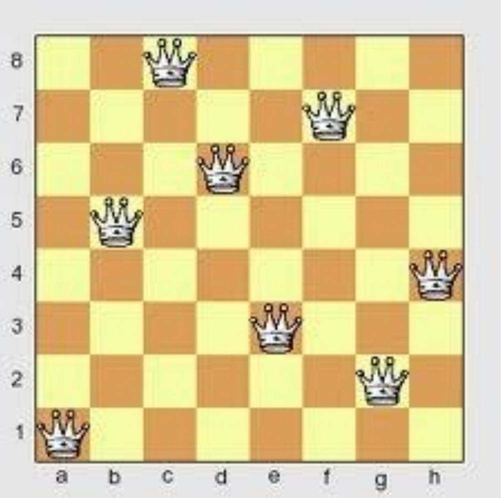
Digital 4 Business
Evolving your digital future

Co-funded by the European Union

Local Search: 8 Queens Problem



Bad Solution



Good Solution

Problem:
Place 8 queens on a chessboard such that
No queens attacks another

- Local Search:
  - the path to the goal is irrelevant!
  - we do not care about reached states
  - complete state formulation is a straightforward representation:
    - 8 queens, one in each column
  - operate by searching from start state to neighbouring states, choose the best neighbour so far, repeat

8 Queens is a candidate for use of local search!

8^8 (about 16 million configurations)

Co-funded by the European Union

Local Search Techniques

- Advantages:
  - They use very little memory
  - Often find solutions in large/infinite search spaces where systematic algorithms would be unreasonable
  - Can be used to solve optimisation problems
- Disadvantages
  - Since they are not systematic they may not find solutions because they leave parts of the search space unexplored.
  - Performance is dependent on the topology of the search space
  - Search may get stuck in local optima

Global Optimum: The best possible solution to a problem.

Local Optimum: A solution to a problem that is better than all other solutions that are slightly different, but worse than the global optimum

Greedy Local Search: A search algorithm that always takes the best immediate, or local, solution while finding an answer. Greedy algorithms find the overall, or globally optimal solution for some optimization problems, but may find less-than-optimal solutions for some instances of other problems. (They may also get stuck!)

Co-funded by the European Union

Hill-Climbing Algorithm (steepest ascent version)

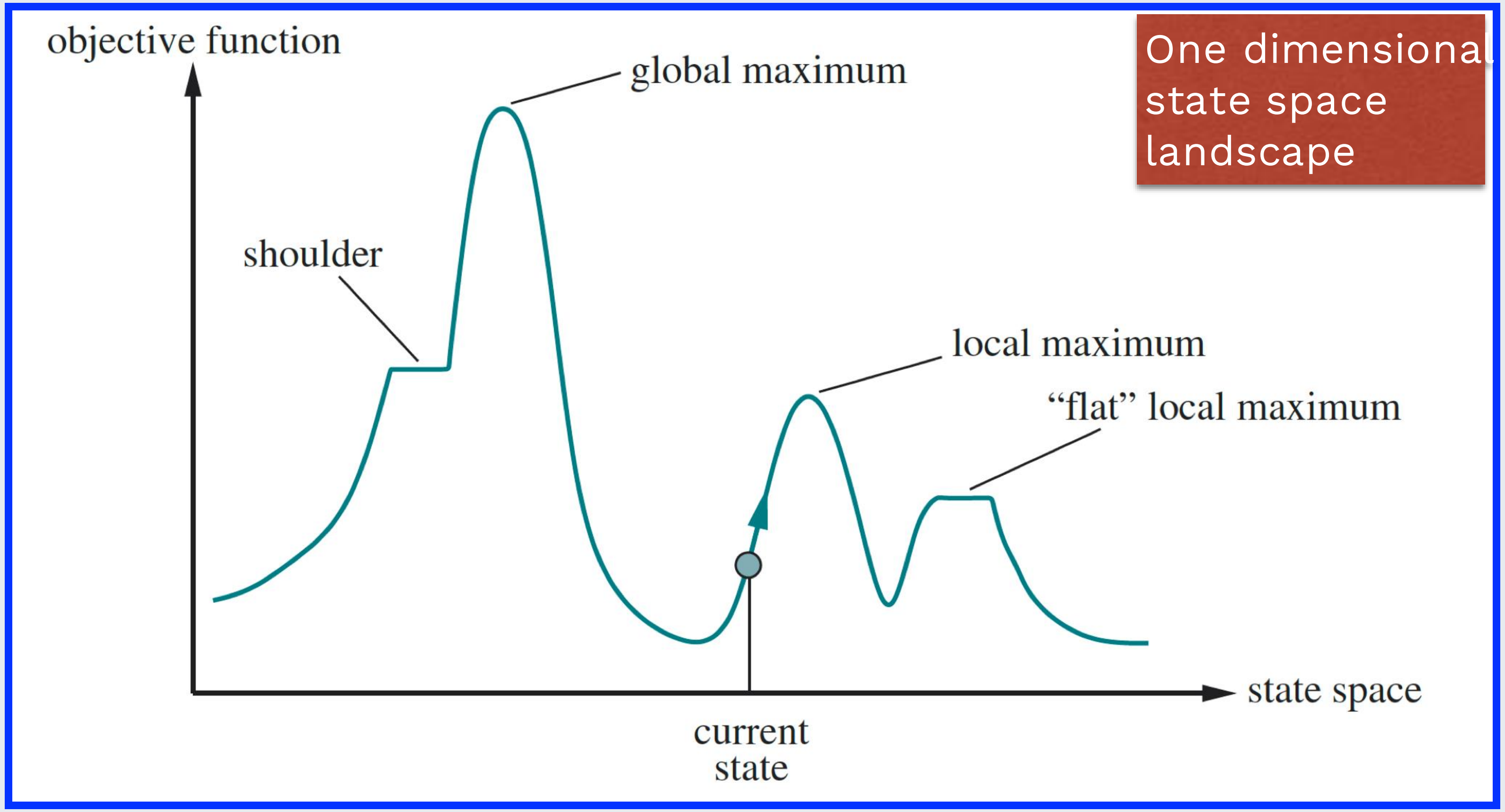
function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
    current ← *problem*.INITIAL
    while *true* **do**
        neighbor ← a highest-valued successor state of *current*
        if VALUE(*neighbor*) ≤ VALUE(*current*) **then return** *current*
        current ← *neighbor*

When using heuristic functions: steepest descent version

Co-funded by the European Union

Greedy Progress: Hill Climbing

Aim: Find the global maximum



One dimensional state space landscape

Hill Climbing: Modify the current state to try and improve it

Co-funded by the European Union

Variations on Hill Climbing

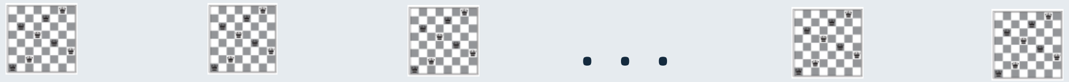
Digital 4 Business
Evolving your digital future

- Stochastic Hill Climbing
  - Choose among uphill moves at random, weighting choice by probability with the steepness of the move
- First Choice Hill Climbing
  - Implements stochastic hill climbing by randomly generating successors until one is generated that is better than the current state.
- Random-Restart Hill Climbing
  - Conducts a series of hill-climbing searches from randomly generated initial states until a goal is found.

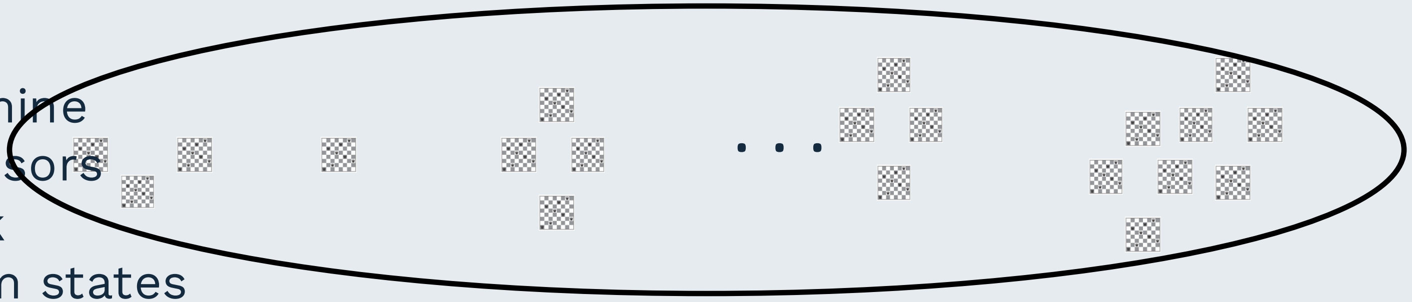
Co-funded by the European Union

Local Beam Search

Start with k random states

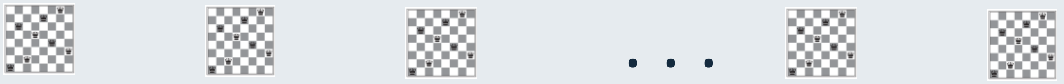


Determine successors of all k random states



If any successors are goal states then finished

Else select k best states from union of successors and repeat



Can suffer from lack of diversity among the k states (concentrated in small region of search space).

Stochastic variant: choose k successors at random with probability of choosing the successor being an increasing function of its value.

Co-funded by the European Union

Digital 4 Business
Evolving your digital future

Simulated Annealing

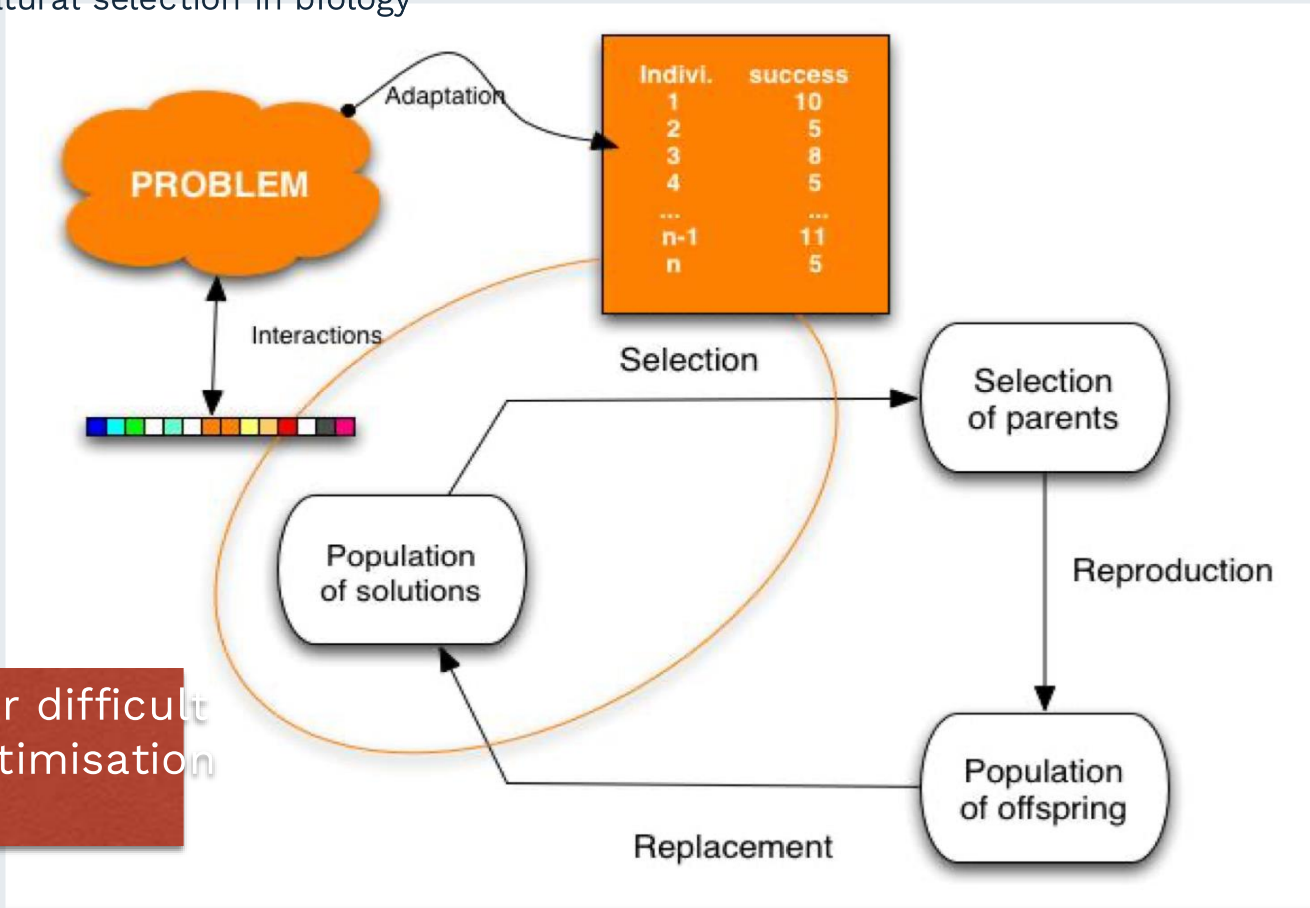
Hill Climbing + Random Walk

- Escape local maxima by allowing “bad” moves (random)
  - Idea: but gradually decrease their size and frequency
  - Origin of concept: metallurgical annealing
- Bouncing ball analogy (gradient descent):
  - Shaking hard (= high temperature)
  - Shaking less (= lower the temperature)
- If Temp decreases slowly enough, best state is reached

Co-funded by the European Union

Evolutionary Algorithms

Variants of Stochastic Beam Search using the metaphor of natural selection in biology



Often used for difficult non-linear optimisation problems

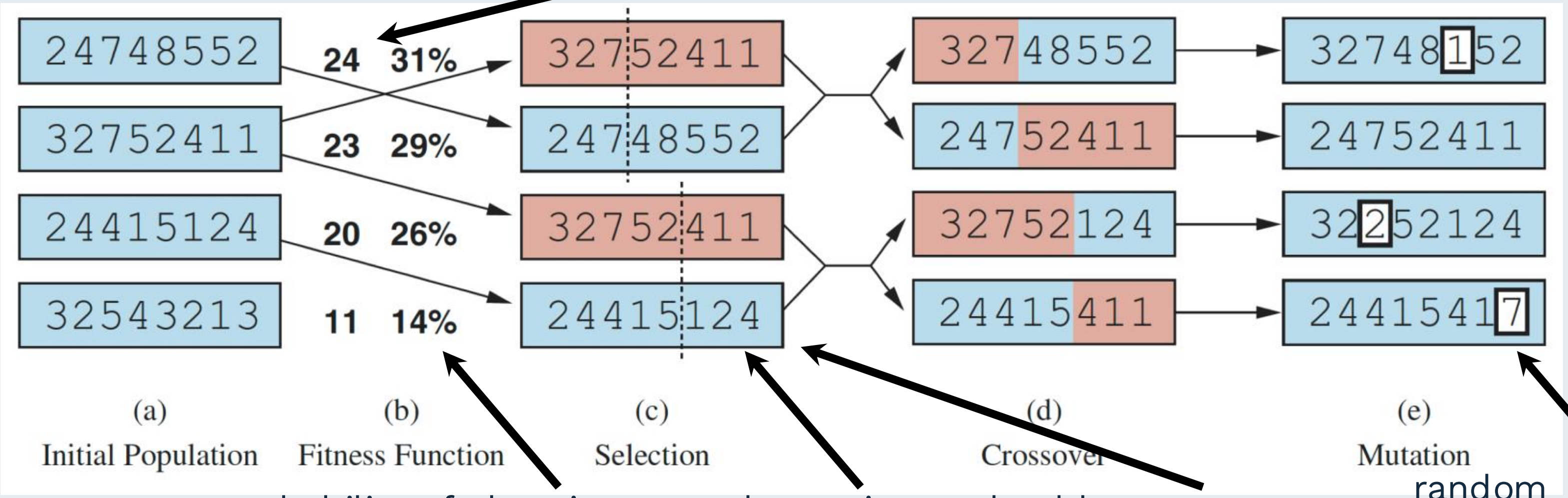
Co-funded by the European Union

Genetic Algorithms

Digital 4 Business
Evolving your digital future

Non-attacking pairs of queens

board encodings



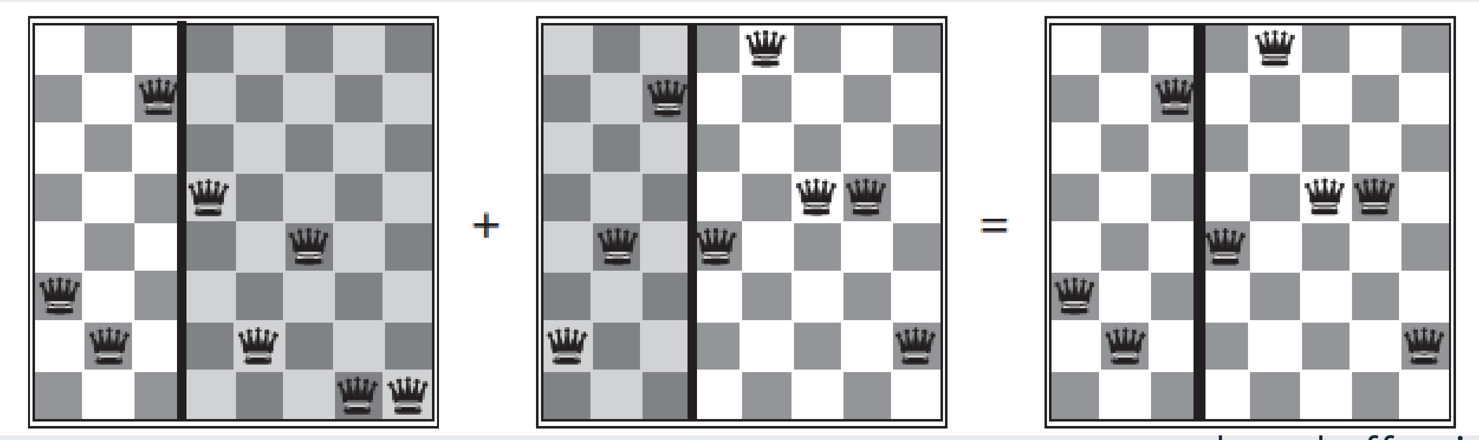
probability of choosing proportional to fitness

select pairs randomly for reproduction

choose crossover point randomly

random mutation small prob.

3 2 7 4 8 5 4 2



3 2 7 5 2 4 1 1

2 4 7 4 8 5 4 2

crossover: breed offspring in next generation

Co-funded by the European Union

Adversarial Search

Co-funded by the European Union

Why Study Board Games?

Board games are one of the oldest branches of AI (Shannon and Turing 1950).

- Board games present a very abstract and pure form of competition between two opponents and clearly require a form of "intelligence".
- The states of a game are easy to represent
- The possible actions of the players are well-defined
  - Realization of the game as a search problem
  - It is nonetheless a contingency problem, because the characteristics of the opponent are not known in advance

Co-funded by the European Union

Challenges

Board games are not only difficult because they are contingency problems, but also because the search trees can become astronomically large.

Examples:

- Chess: On average 35 possible actions from every position, 100 possible moves/ply (50 each player): $35^{100} \approx 10^{150}$ nodes in the search tree (with “only” 10^{40} distinct chess positions (states)).
- Go: On average 200 possible actions with circa 300 moves: $200^{300} \approx 10^{700}$ nodes.

Good game programs have the properties that they

- delete irrelevant branches of the game tree,
- use good evaluation functions for in-between states, and
- look ahead as many moves as possible.

Co-funded by the European Union

More generally: Adverserial Search

- Multi-Agent Environments
  - agents must consider the actions of other agents and how these agents affect or constrain their own actions.
  - environments can be cooperative or competitive.
  - One can view this interaction as a "game" and if the agents are competitive, their search strategies may be viewed as "adversarial".
- Most often studied: Two-agent, zero-sum games of perfect information
  - Each player has a complete and perfect model of the environment and of its own and other agents actions and effects
  - Each player moves until one wins and the other loses, or there is a draw.
  - The utility values at the end of the game are always equal and opposite, thus the name zero-sum.
  - Chess, checkers, Go, Backgammon (uncertainty)

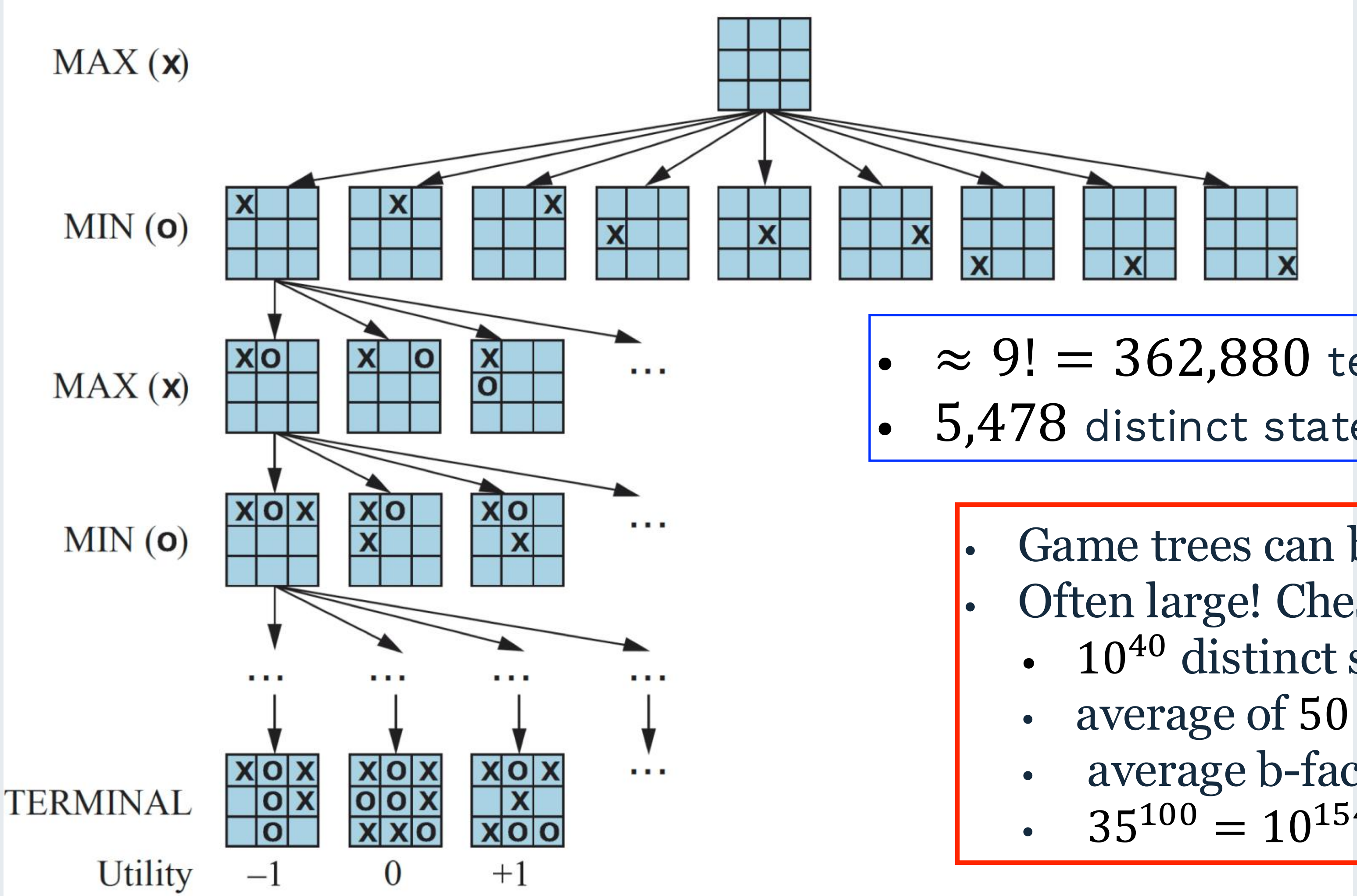
Co-funded by
the European Union

Games as Search

- The Game
  - Two players: One called MIN, the other MAX. MAX moves first.
  - Each player takes an alternate turn until the game is over.
  - At the end of the game points are awarded to the winner, penalties to the loser.
- Formal Problem Definition:
  - Initial State: S_0 – Initial board position
  - TO-MOVE(s) - The player whose turn it is to move in state s
  - ACTION(s) - The set of legal moves in state s
  - RESULT(s,a) - The transition model: the state resulting from taking action a in state s.
  - IS-TERMINAL(s) - A terminal test. True when game is over.
  - UTILITY(s,p) – A utility function. Gives final numeric value to player p when the game ends in terminal state s.
  - For example, in Chess: win (1), lose (-1), draw (0):

Co-funded by the European Union

(Partial) Game Tree for Tic-Tac-Toe



- $\approx 9! = 362{,}880$ terminal nodes
- $5{,}478$ distinct states

- Game trees can be infinite
- Often large! Chess has:
  - 10^{40} distinct states
  - average of 50 moves
  - average b-factor of 35
  - $35^{100} = 10^{154}$ nodes

Co-funded by the European Union

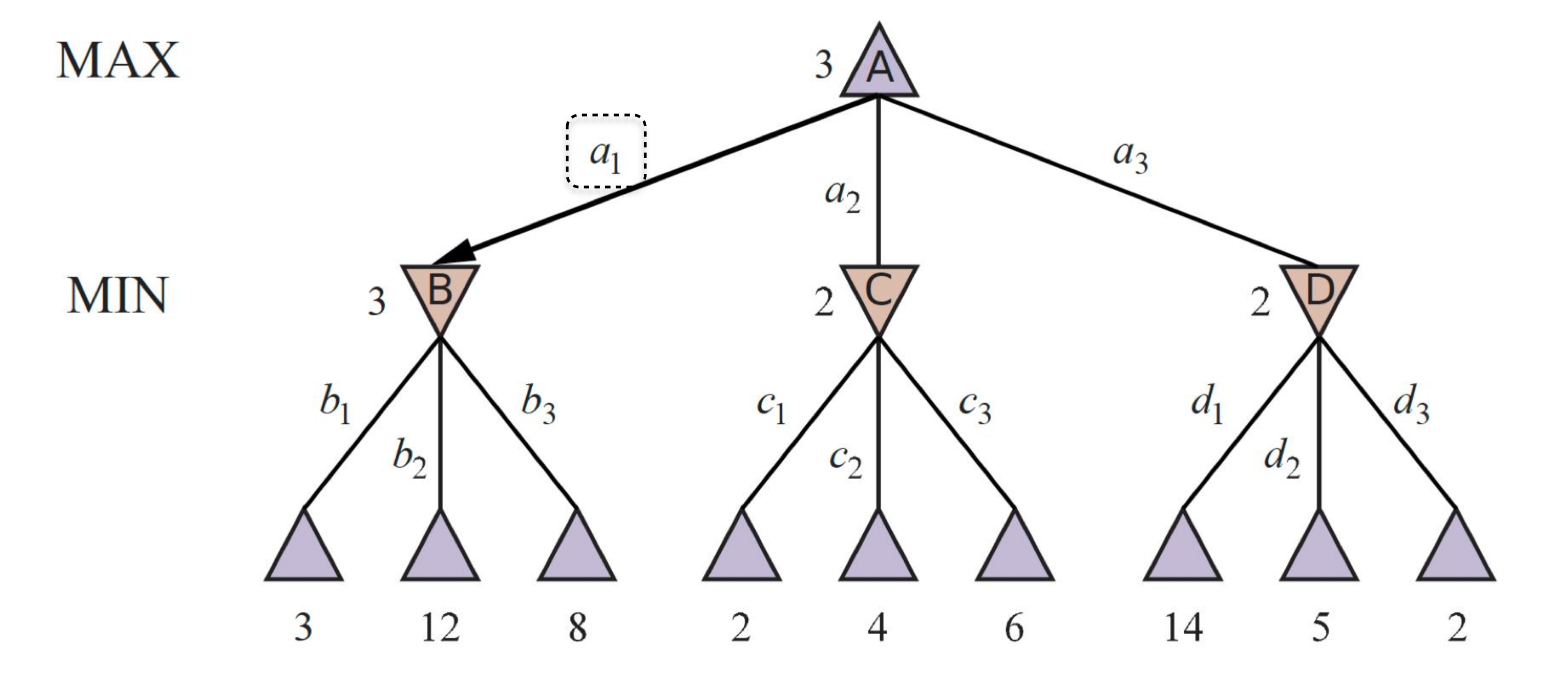
Optimal Decisions in Games: Minimax Search

1. Generate the complete game tree using depth-first search.
2. Apply the utility function to each terminal state.
3. Beginning with the terminal states, determine the utility of the predecessor nodes (parent nodes) as follows:
    1. Node is a MIN-node
       Value is the minimum of the successor nodes
    2. Node is a MAX-node
       Value is the maximum of the successor nodes
4. From the initial state (root of the game tree), MAX chooses the move that leads to the highest value (minimax decision).

Note: Minimax assumes that MIN plays perfectly. Every weakness (i.e. every mistake MIN makes) can only improve the result for MAX.

Co-funded by the European Union

Minimax Tree

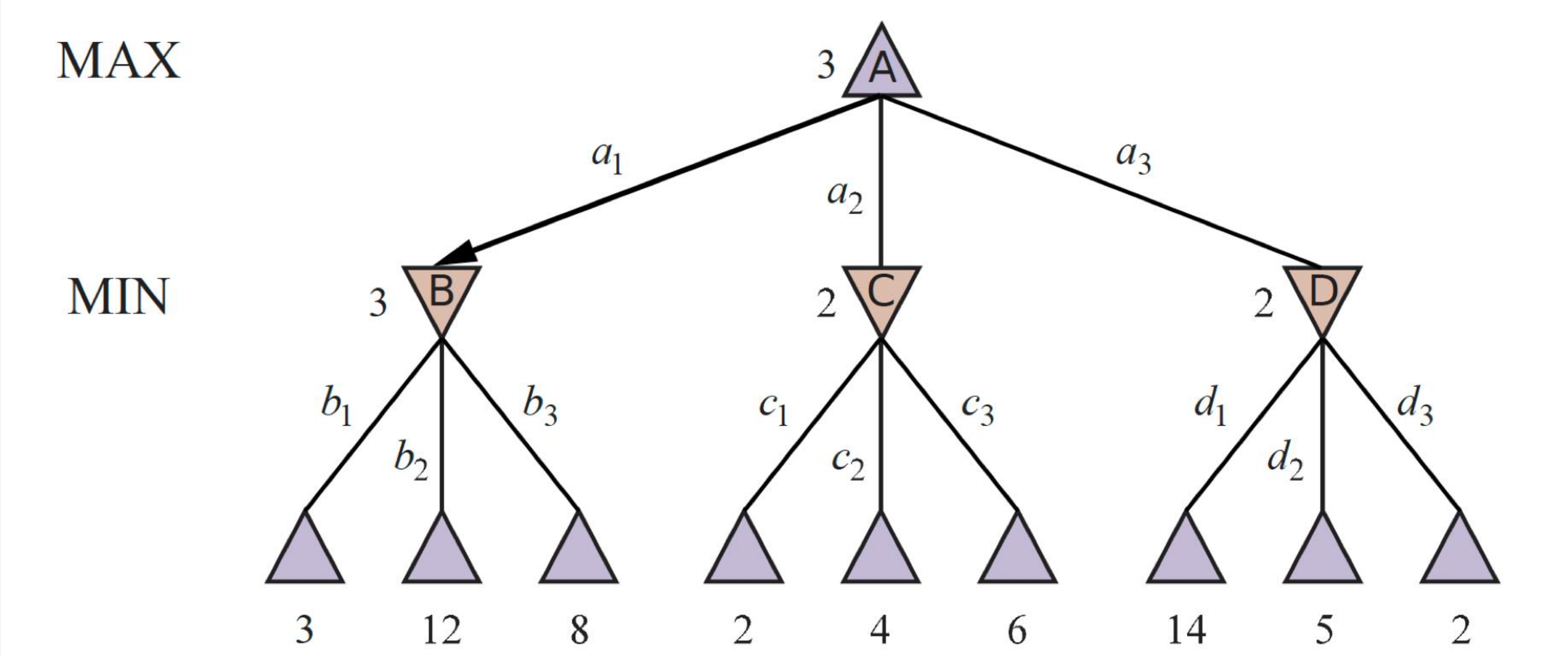


- Interpreted from MAX's perspective
- Assumption is that MIN plays optimally
- The minimax value of a node is the utility for MAX
- MAX prefers to move to a state of maximum value and MIN prefers minimum value

Co-funded by the European Union

What move should MAX make from the Initial state?

MAX utility values



$$MINIMAX(s) =$$

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

Co-funded by the European Union

Minimax Algortihm

```
function MINIMAX-SEARCH(game, state) returns an action
   player ← game.TO-MOVE(state)
   value, move ← MAX-VALUE(game, state)
   return move

function MAX-VALUE(game, state) returns a (utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   v ← −∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
      if v2 > v then
         v, move ← v2, a
   return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   v ← +∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
      if v2 < v then
         v, move ← v2, a
   return v, move
```

Assume max depth of the tree is m and b legal moves at each point:

- Time complexity: $\mathbf{O}(b^m)$
- Space complexity:
  - Actions generated at same time: $\mathbf{O}(bm)$
  - Actions generated one at a time: $\mathbf{O}(m)$

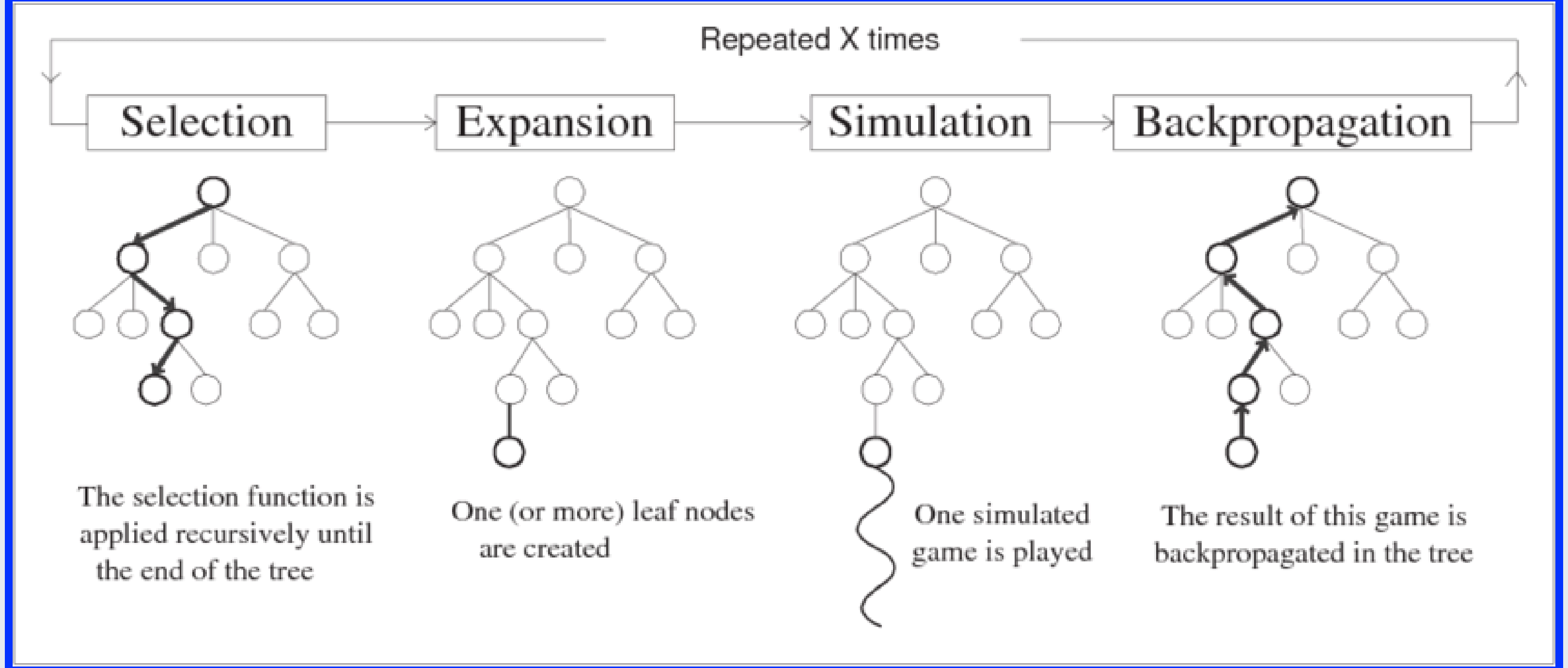
Serves as a basis for mathematical analysis of games and development of approximations to the minimax algorithm

Recursive algorithm that proceeds all the way down to the leaves of the tree and then backs up the minimax values through the tree as the recursion unwinds

Co-funded by the European Union

4 Steps in MCTS

MCTS maintains a search tree and grows it on each iteration using the following steps:



Starting at the root of the search tree, choose a move using the selection policy, repeating the process until a leaf node is reached

Grow the search tree by generating a new child/children.

Perform a playout from a child using the playout policy. These moves are not recorded in the search tree

Use the simulation result to update the utilities of the nodes going back up to the root.

After X times: Choose the best move from start state