Universidad de Alcalá

# Search

Intelligent Systems Group

# Outline

- Introduction
- Problem-solving agents
- Problem formulation
- Problem types
- Example problems
- Basic search algorithms
- Conclusions

# Introduction

- Early AI works were directed to:
  - Proof of theorems
  - Solving crosswords
  - Games
- All in AI is search
  - Not entirely true (obviously) but more than you can imagine
  - Finding a good/best solution to a problem among several possible solutions

# Outline

- Introduction
- **Problem-solving agents**
- Problem formulation
- Problem types
- Example problems
- Basic search algorithms
- Conclusions

# Problem-solving agents

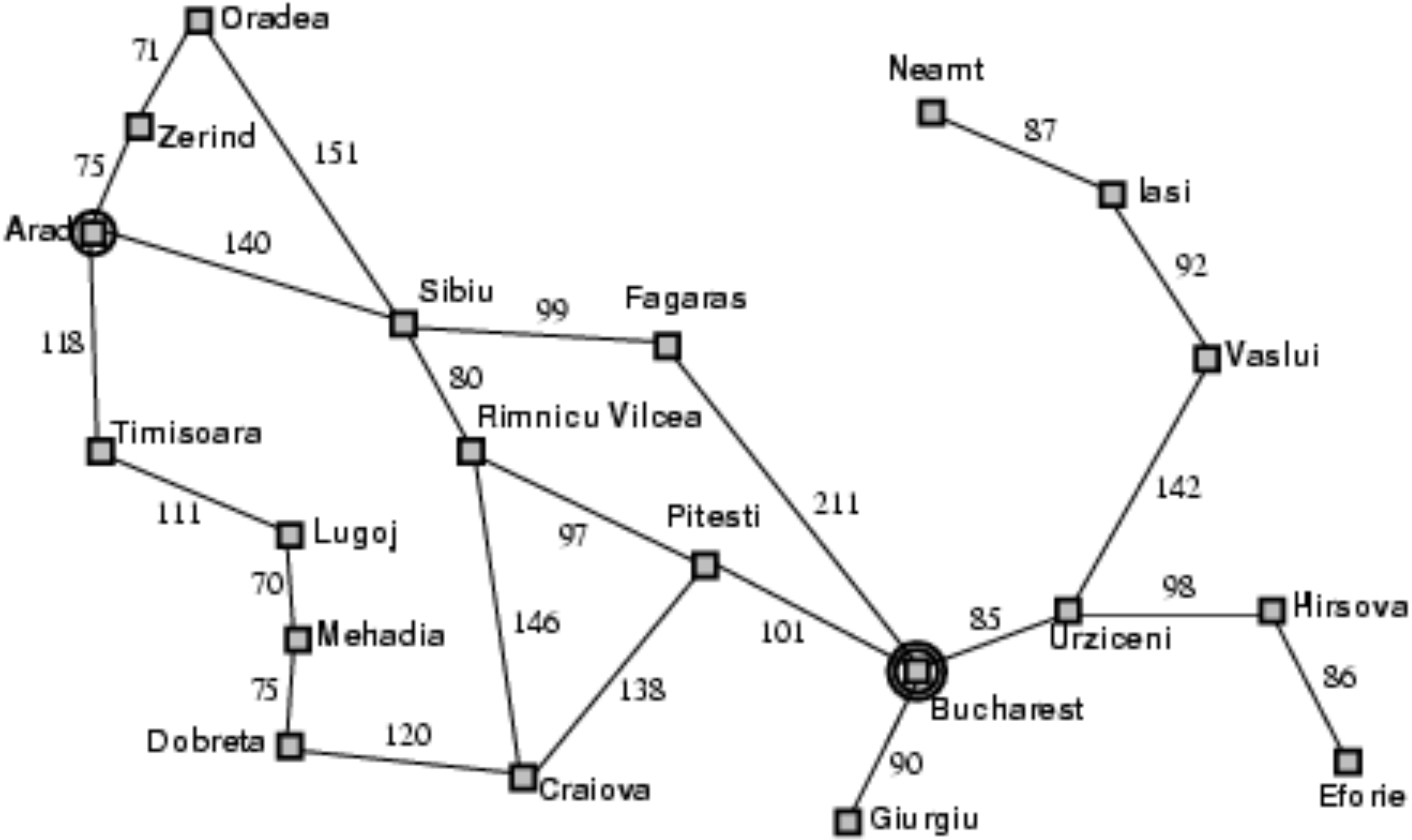
- Agents must maximize its performance measure
- Example: On holiday in Romania; currently in Arad
  Flight leaves tomorrow from Bucharest
- Formulate goal:
  - be in Bucharest
- Formulating the problem:
  - states: multiple cities
  - actions: drive between cities
- Finding a solution:
  - Sequence cities, eg., Arad, Sibiu, Fagaras, Bucharest

- The process of finding such a solution is called search

# Problem-solving agents

- Assumptions of the environment:
  - Static: search and formulation is done without considering changes in the environment
  - Observable: the initial state is known
  - Discrete: the alternative locations are known
  - Deterministic: each state is determined by the current state and the action executed
- The solutions are simple sequences of actions, they are executed without considering perceptions

# Problem-solving agents



# Outline

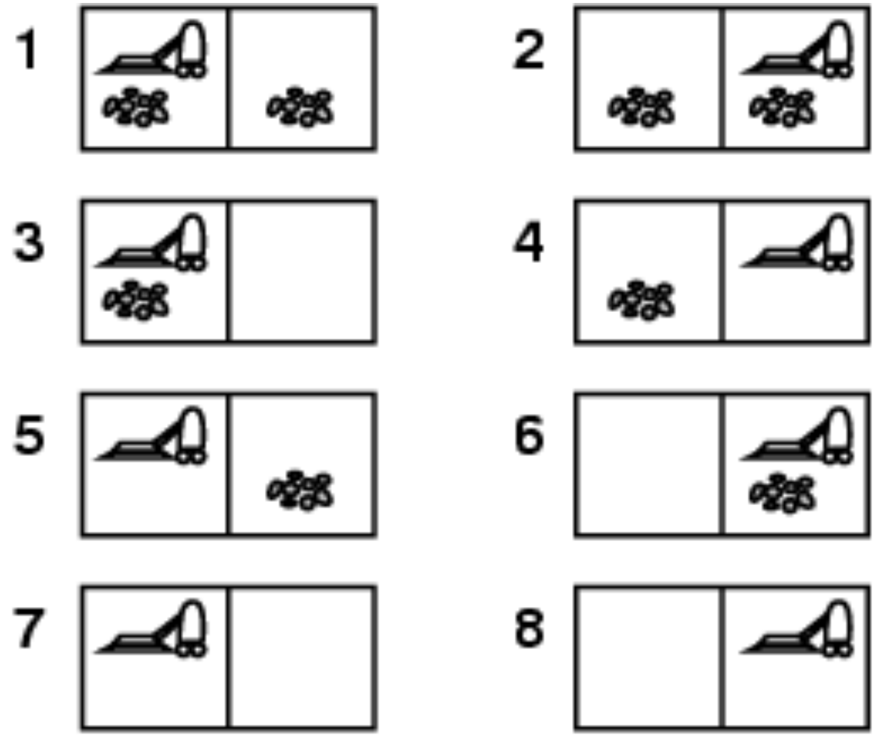
- Introduction
- Problem-solving agents
- **Problem types**
- Problem formulation
- Example problems
- Basic search algorithms
- Conclusions

# Problem types

- Deterministic, fully observable → single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence
- Non-observable → sensorless problem (conformant problem)
  - Agent may have no idea where it is; solution is a sequence
- Nondeterministic and/or partially observable → contingency problem
  - percepts provide new information about current state
  - often interleave search with execution
- Unknown state space → exploration problem

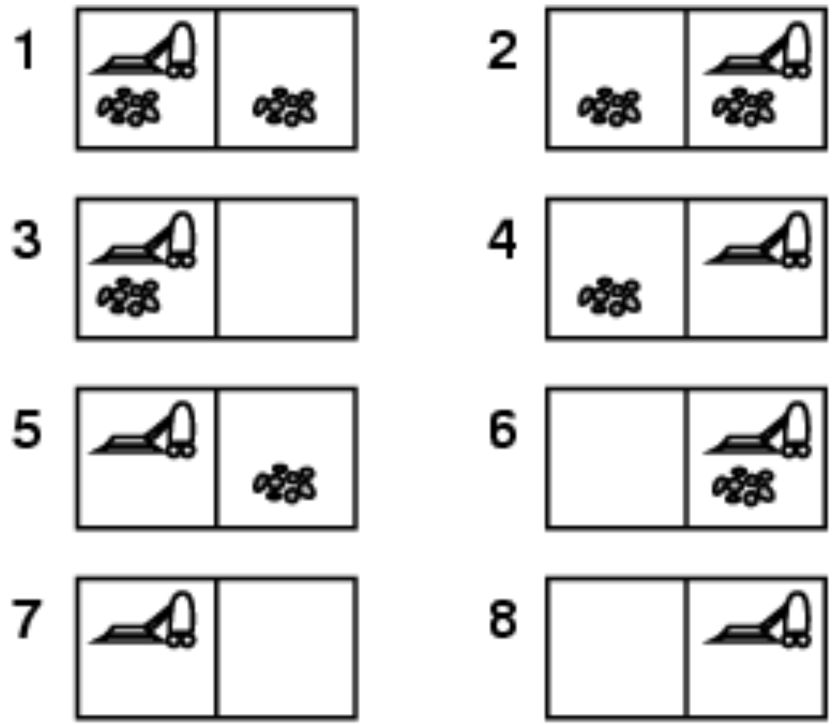
# Example: vacuum world

- **Single-state**, start in #5.
  Solution?



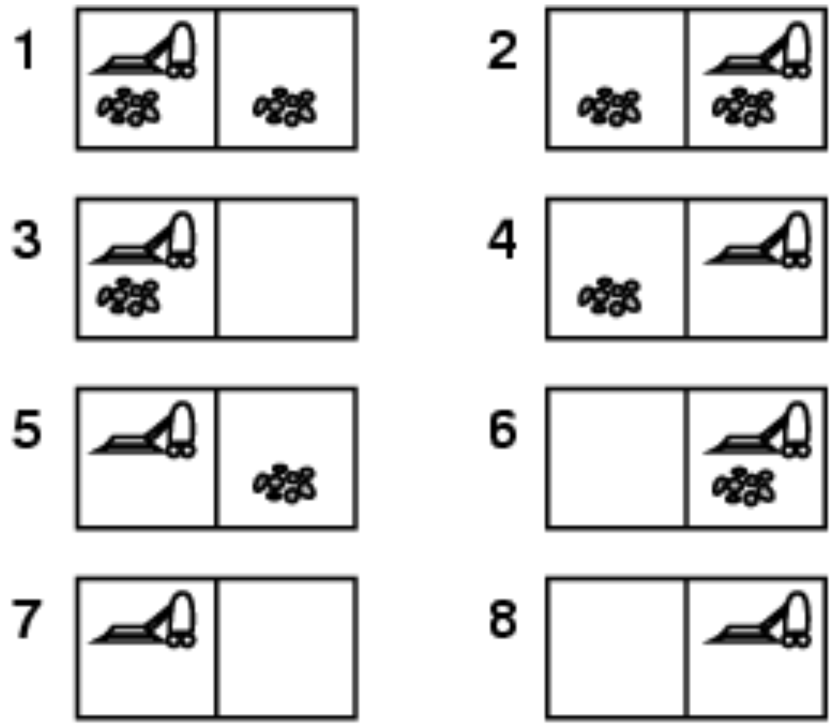
# Example: vacuum world

- **Single-state**, start in #5.
  Solution? *[Right, Suck]*
- **Sensorless,** start in {*1,2,3,4,5,6,7,8*} e.g.,
  *Right* goes to {*2,4,6,8*}
  Solution?



# Example: vacuum world

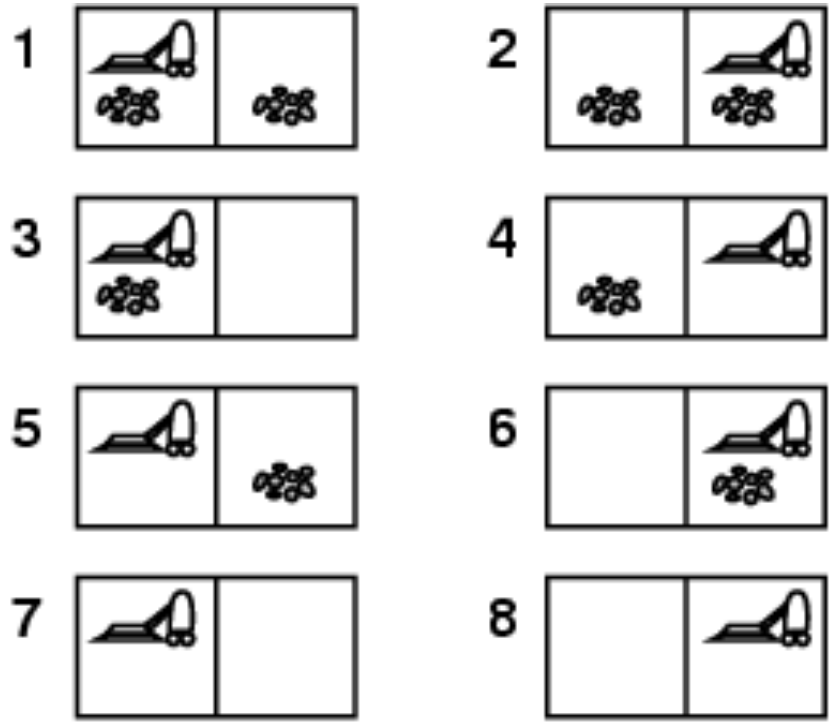
- **Sensorless,** start in {*1,2,3,4,5,6,7,8*} e.g., *Right* goes to {*2,4,6,8*} Solution? *[Right,Suck,Left,Suck]*




- **Contingency**
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: *[L, Clean],* i.e., start in #5 or #7 Solution?

# Example: vacuum world

- **Sensorless**, start in {*1,2,3,4,5,6,7,8*} e.g., *Right* goes to {*2,4,6,8*} Solution? *[Right,Suck,Left,Suck]*




- **Contingency**
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: *[L, Clean],* i.e., start in #5 or #7
    Solution? *[Right, **if** dirt **then** Suck]*

# Outline

- Introduction
- Problem-solving agents
- Problem types
- **Problem formulation**
- Example problems
- Basic search algorithms
- Conclusions

# Problem formulation

- A problem is defined by four items:

1. initial state e.g., "at Arad"
2. actions or successor function *S(x)* = set of action–state pairs
   - e.g., *S(Arad)* = {<*Arad* → *Zerind, Zerind*>, … }
3. goal test, can be
   - explicit, e.g., *x* = "at Bucharest"
   - implicit, e.g., *Checkmate(x)*
4. path cost (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - *c(x,a,y)* is the step cost, assumed to be ≥ 0

- A solution is a sequence of actions leading from the initial state to a goal state

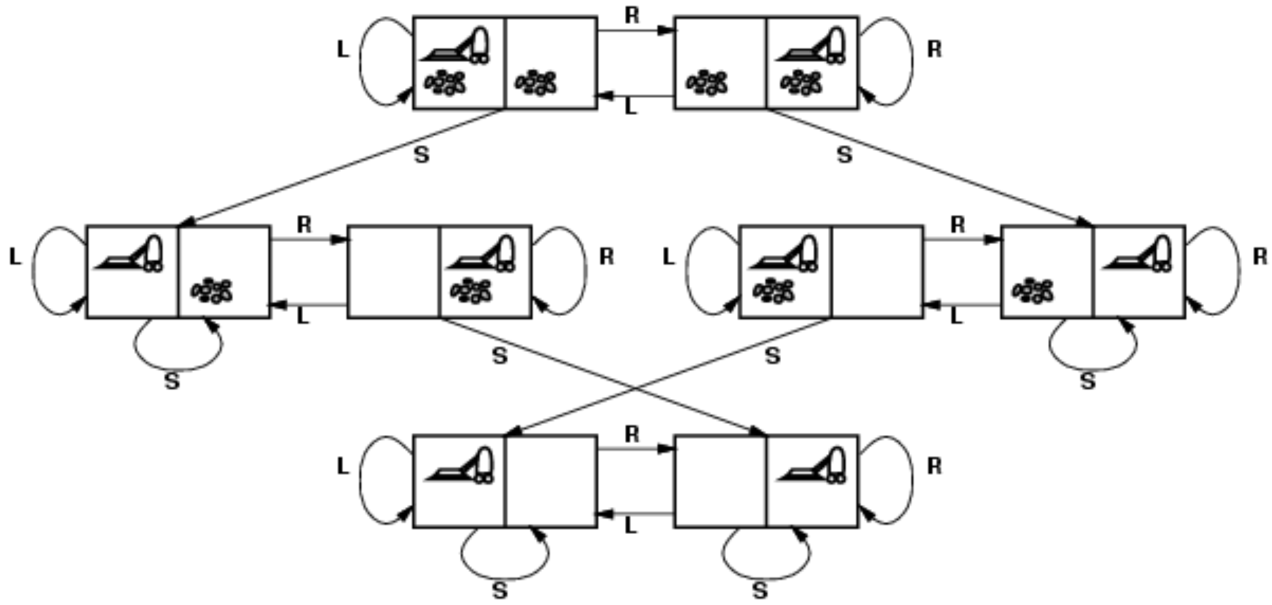
# Problem formulation

- Real world is absurdly complex
  → state space must be abstracted for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, any real state "in Arad“ must get to some real state "in Zerind”
- (Abstract) solution =
  - set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem

# Outline

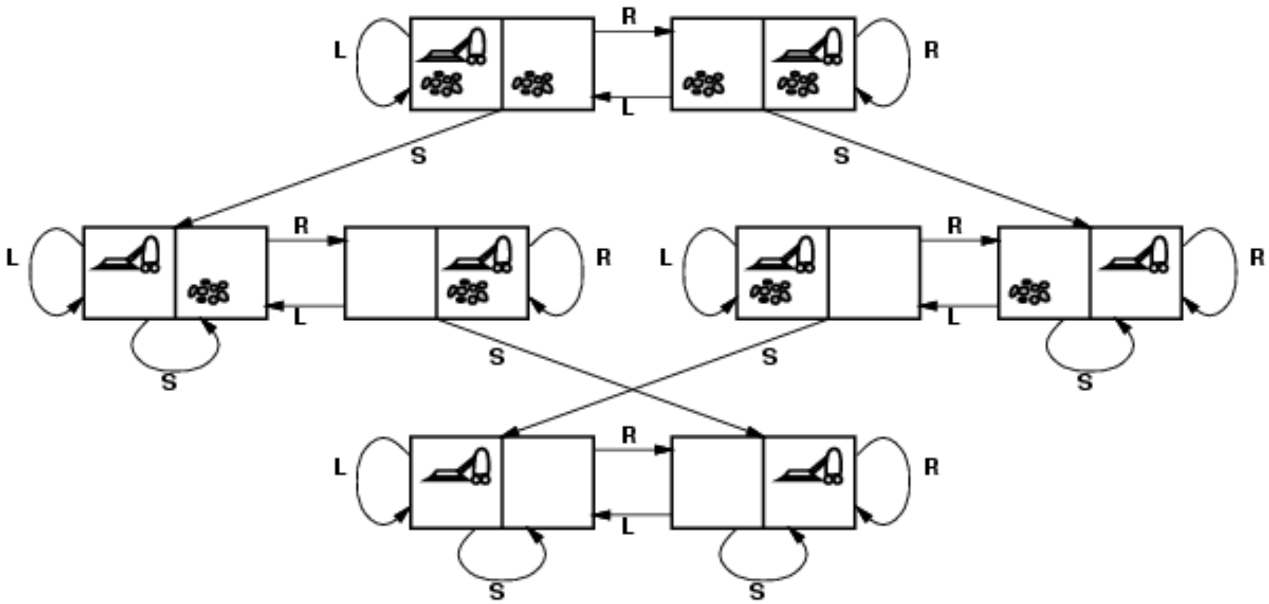
- Introduction
- Problem-solving agents
- Problem types
- Problem formulation
- **Example problems**
- Basic search algorithms
- Conclusions

# Vacuum world state space graph



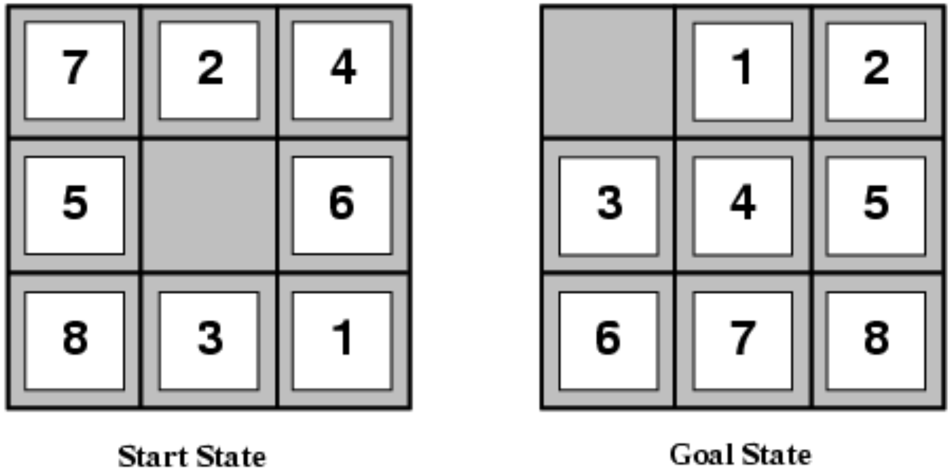

- states?
- actions?
- goal test?
- path cost?

# Vacuum world state space graph



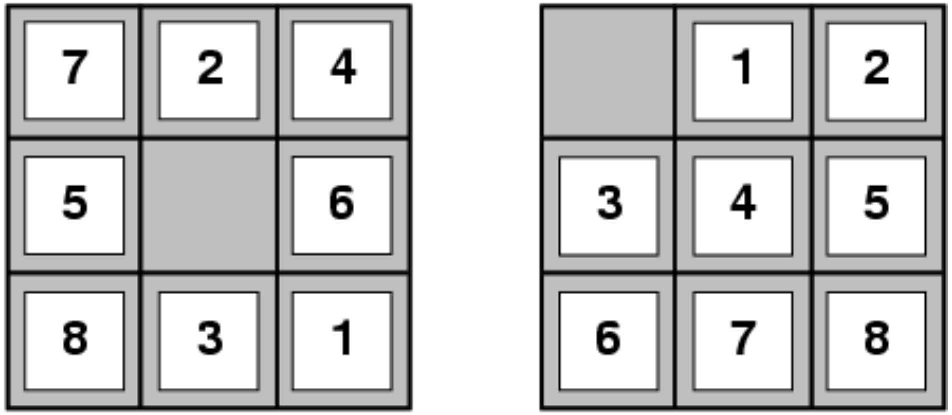

- states? integer dirt and robot location
- actions? *Left*, *Right*, *Suck*
- goal test? no dirt at all locations
- path cost? 1 per action

# Example: The 8-puzzle




- states?
- actions?
- goal test?
- path cost?

# Example: The 8-puzzle

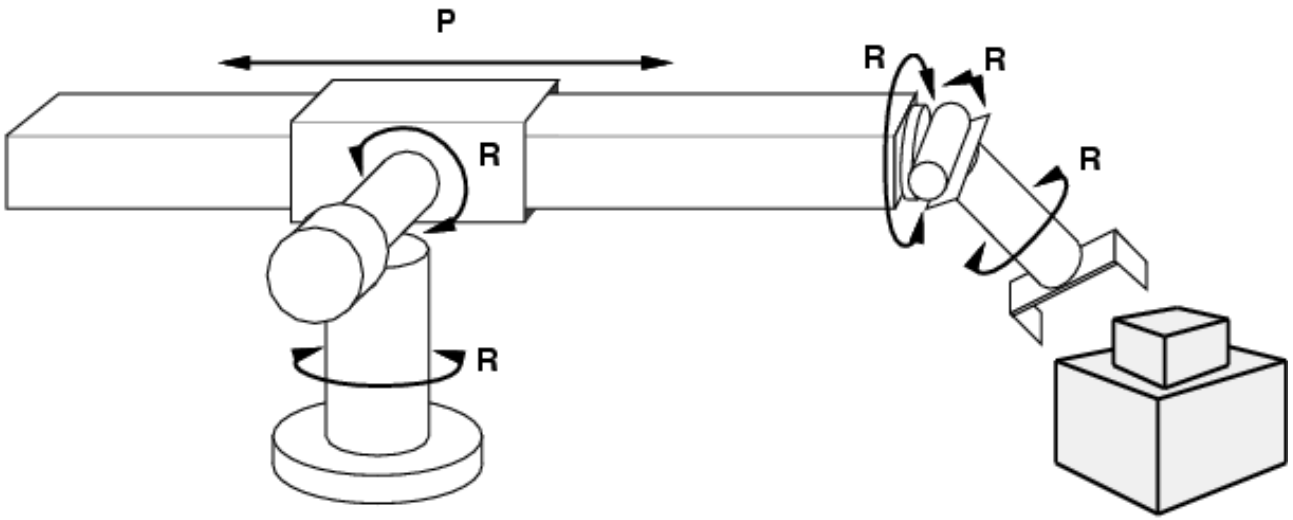



Start State Goal State

- states? locations of tiles
- actions? move blank left, right, up, down
- goal test? = goal state (given)
- path cost? 1 per move

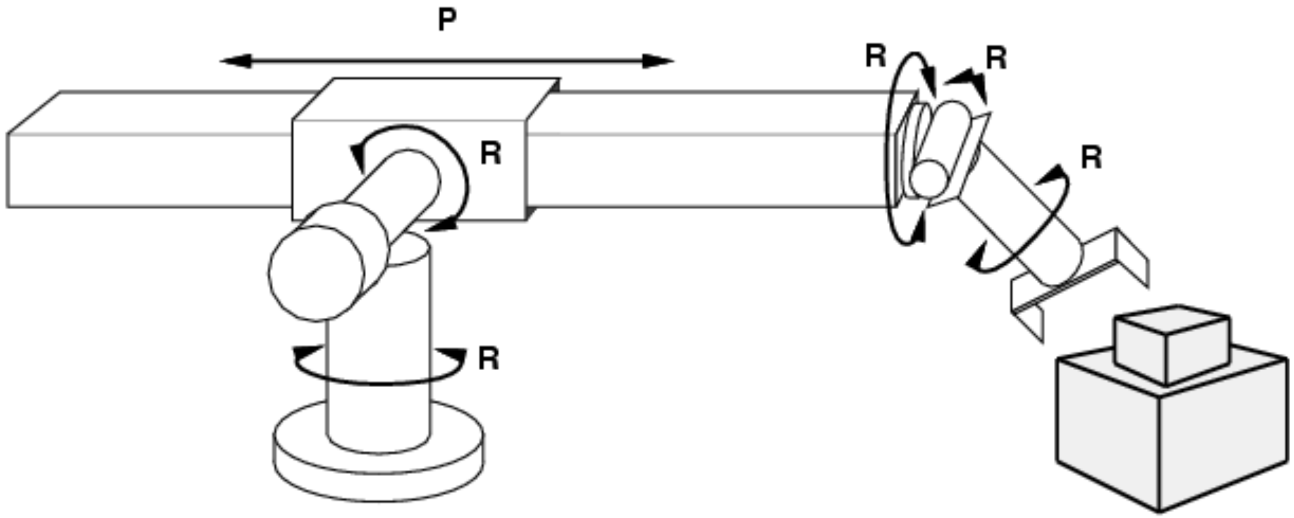
[Note: optimal solution of *n*-Puzzle family is NP-hard]

# Example: robotic assembly




- states?
- actions?
- goal test?
- path cost?

# Example: robotic assembly




- states?: real-valued coordinates of robot joint angles parts of the object to be assembled
- actions?: continuous motions of robot joints
- goal test?: complete assembly
- path cost?: time to execute

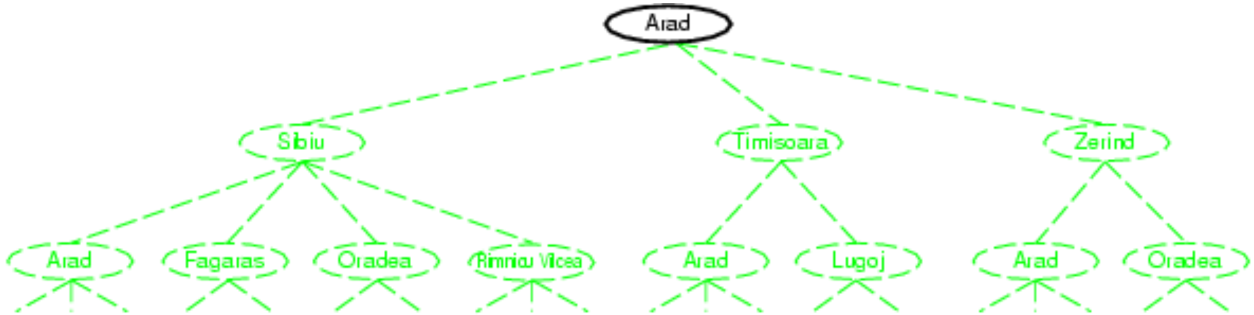
# Outline

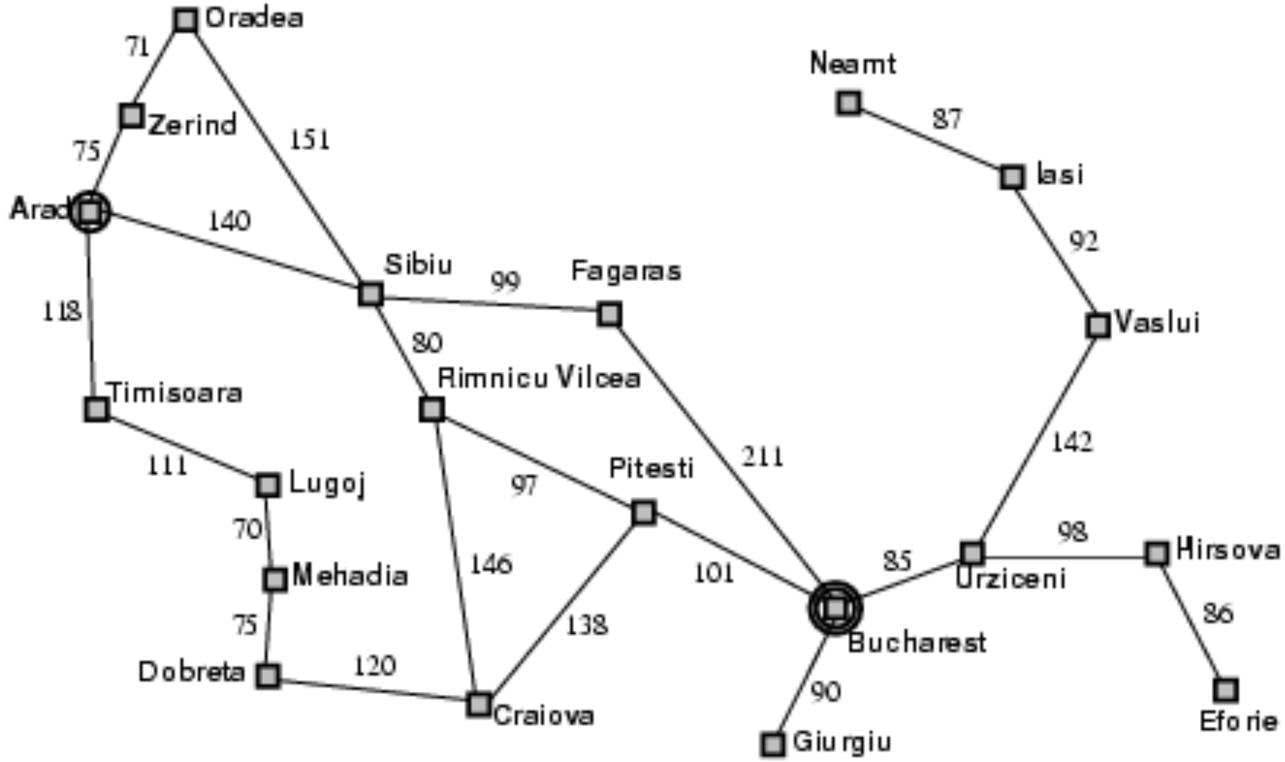
- Introduction
- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- **Basic search algorithms**
- Conclusions

# Search algorithms

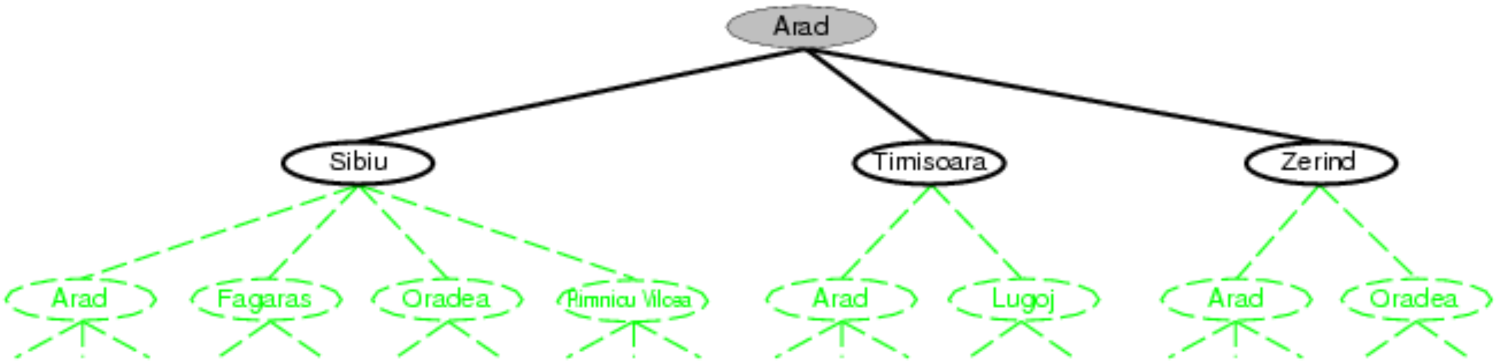
- We have formulated problems, we now need to solve them: search tree
- In general we can have a search graph rather than a tree when the state can be reached from multiple paths
- Basic idea:
  - offline, simulated exploration of state space by generating successors of already-explored states (a.k.a.~expanding states)

# Tree search example

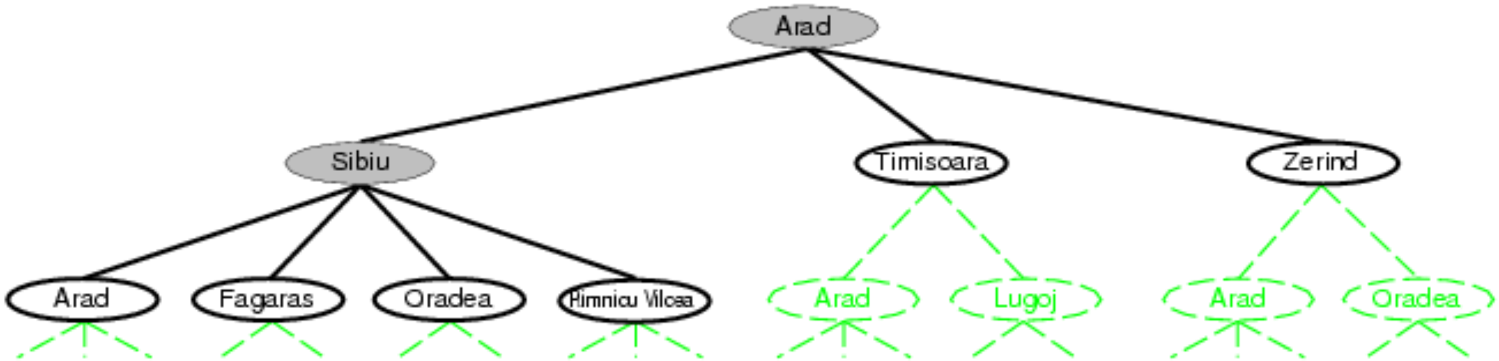


# Tree search example

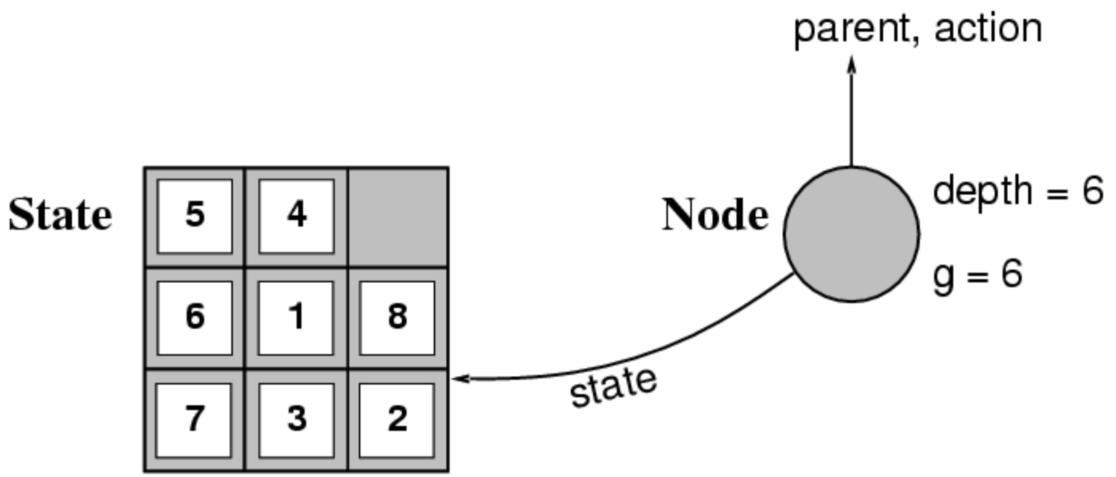


# Tree search example



# Implementation: states vs. nodes

- A state is a (representation of) a physical configuration
- A node is a data structure constituting part of a search tree includes state, parent node, action, path cost *g(x)*, depth




- The `Expand` function creates new nodes, filling in the various fields and using the `SuccessorFn` of the problem to create the corresponding states.

# Search strategies

- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - 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 the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of the state space (may be $\infty$)

# Outline

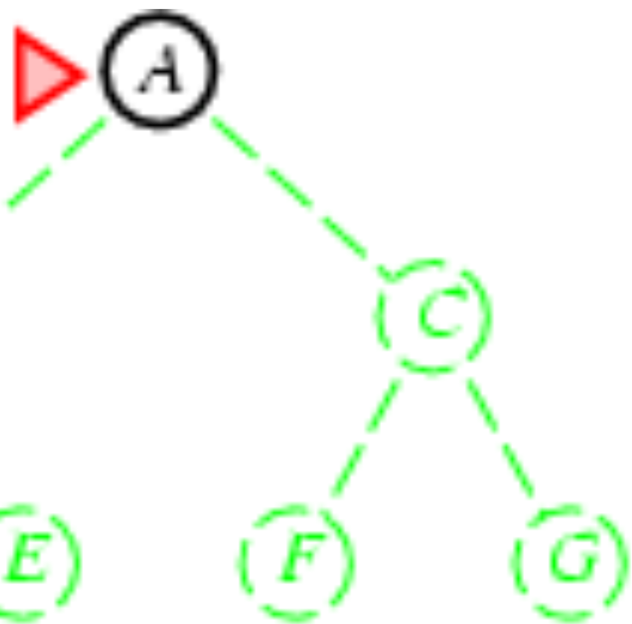
- Introduction
- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms
  - **Uninformed search strategies**
  - Informed search strategies
- Conclusions

# Uninformed search strategies

- Uninformed search strategies use only the information available in the problem definition
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search

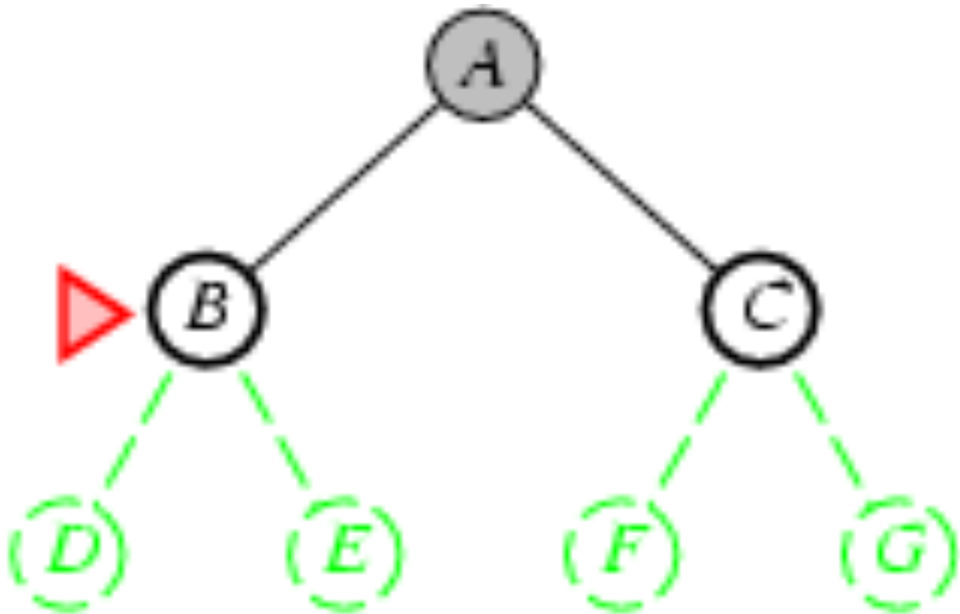
# Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



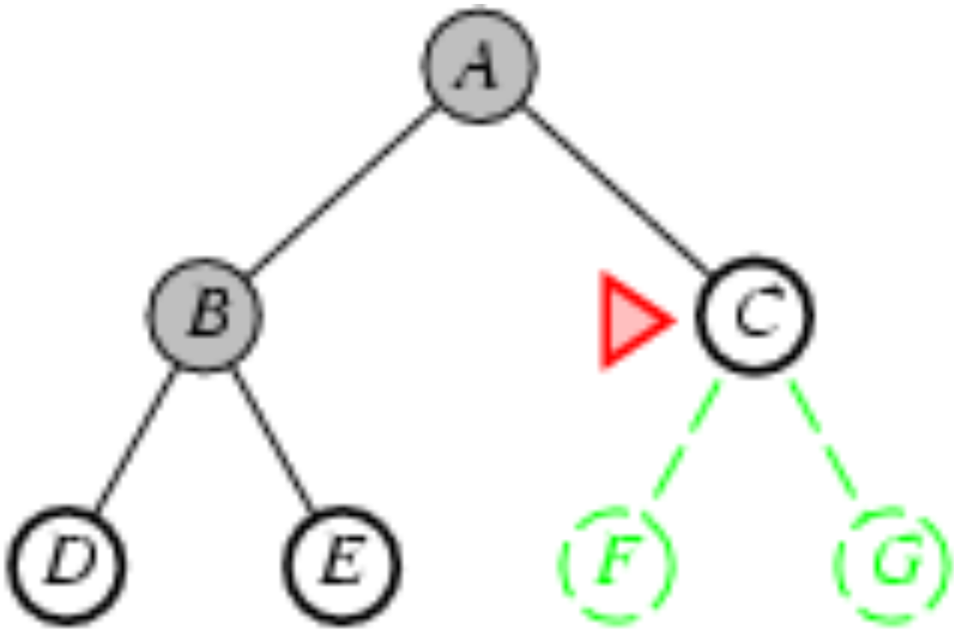
# Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



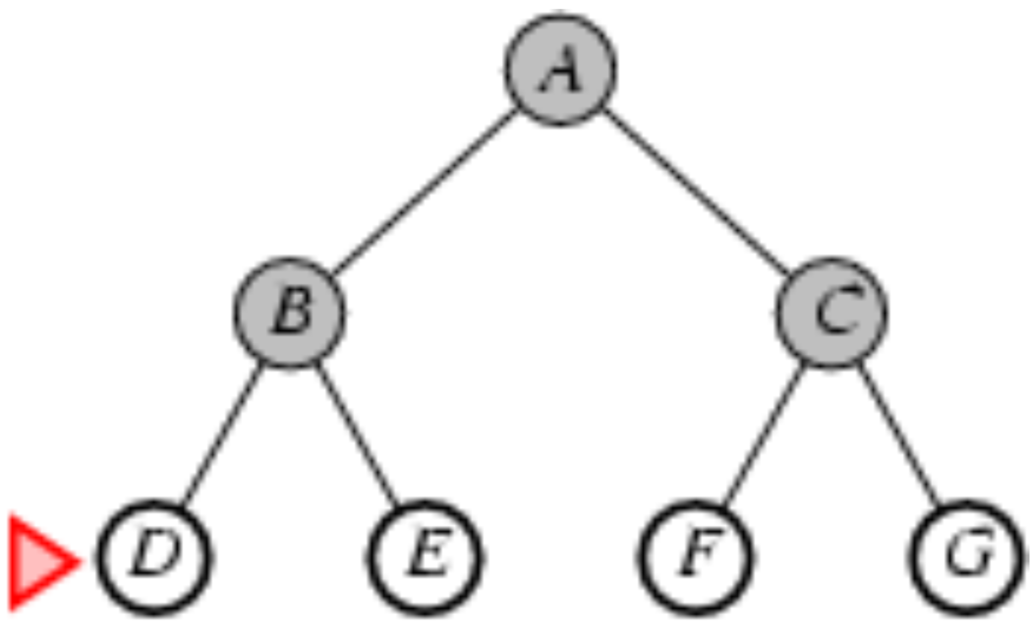
# Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



# Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end



# Properties of breadth-first search

- Complete? Yes (if $b$ is finite)
- Time? $1+b+b^2+b^3+\ldots +b^d + b(b^d-1) = O(b^{d+1})$
- Space? $O(b^{d+1})$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)

Space is the bigger problem (more than time)

Each state has b successors (branching factor)
d is the shallower depth

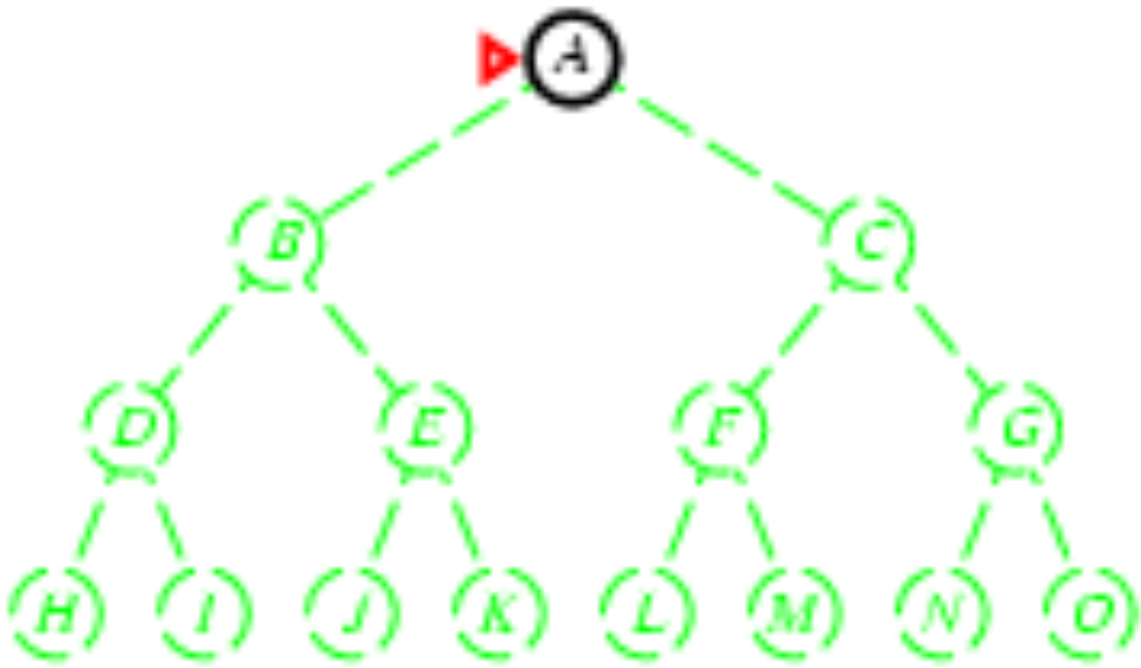
# Uniform-cost search

- Expand least-cost unexpanded node
- Implementation:
  - *fringe* = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- Complete? Yes, if step cost ≥ ε
- Time? # of nodes with *g* ≤ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$ where $C^*$ is the cost of the optimal solution
- Space? # of nodes with *g* ≤ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$
- Optimal? Yes – nodes expanded in increasing order of *g(n)*

*If all costs are equal* → $O(b^d)$

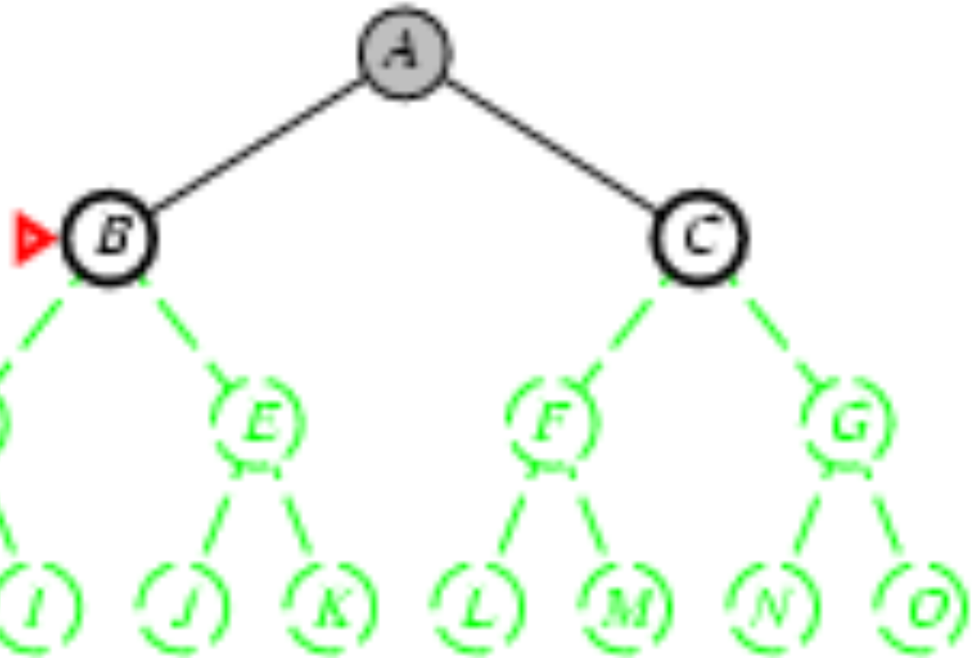
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



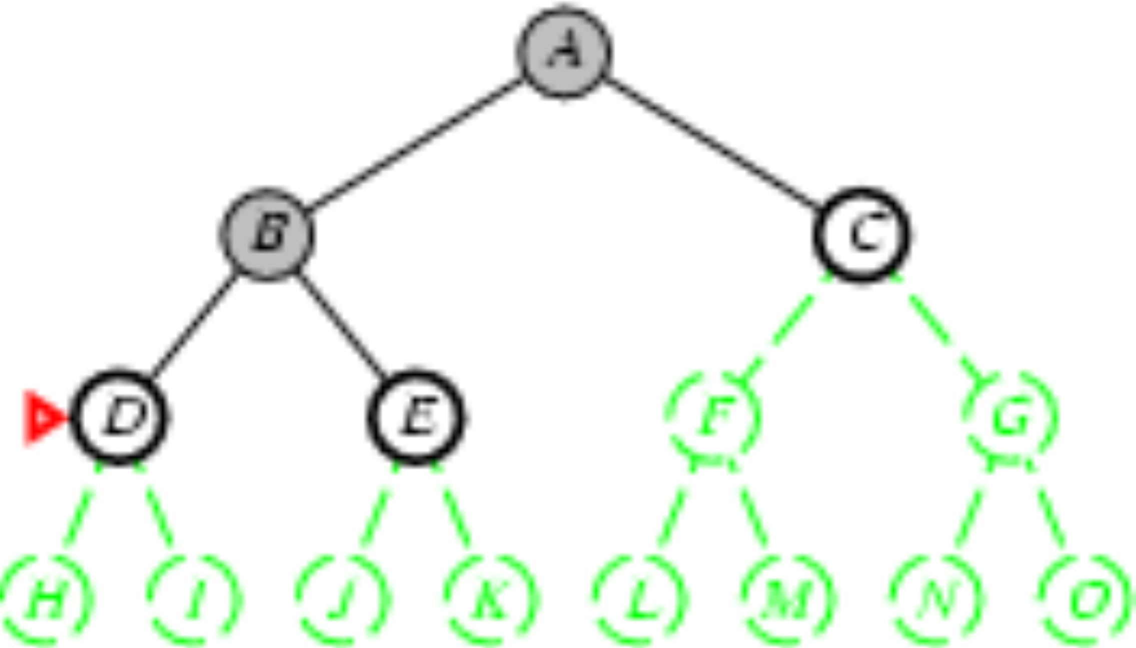
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

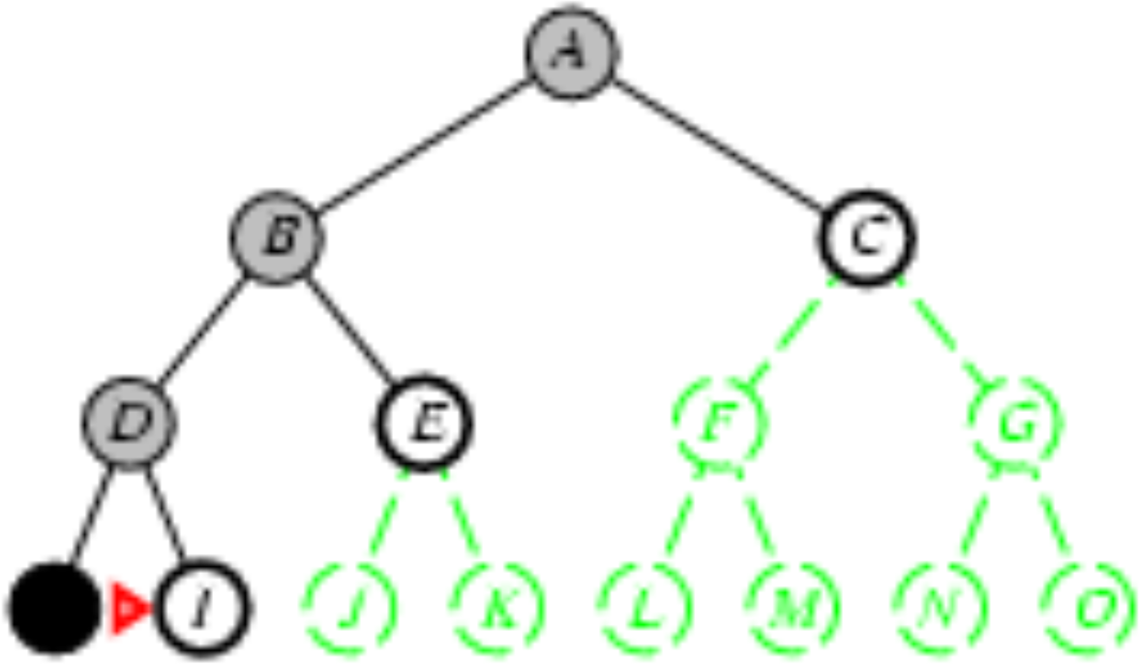


# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

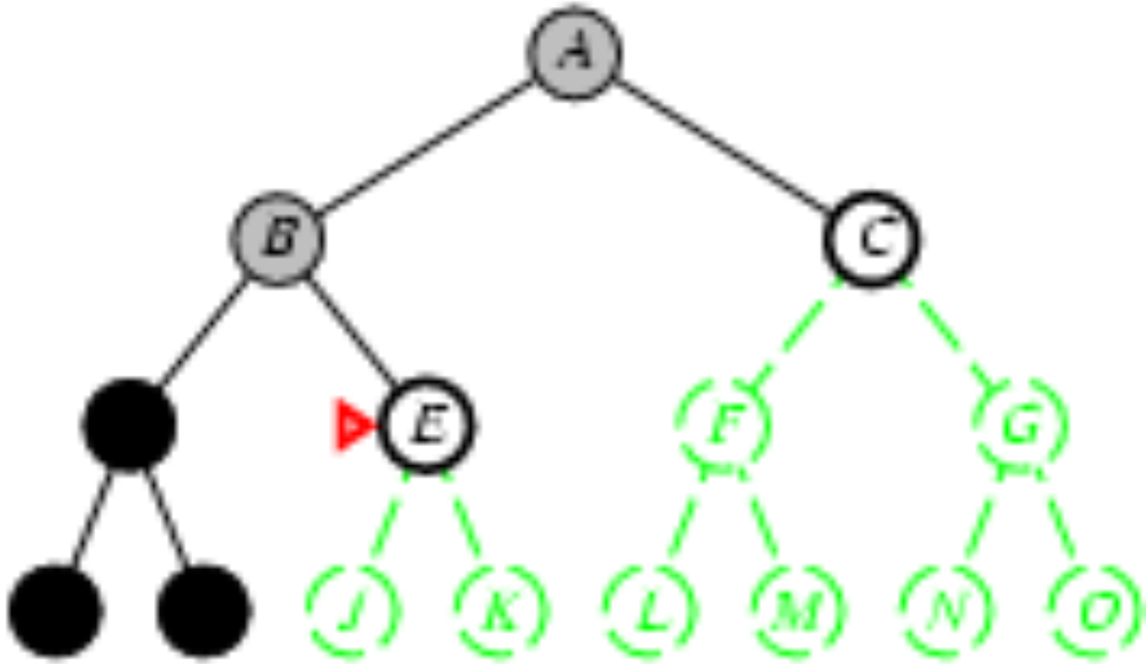
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



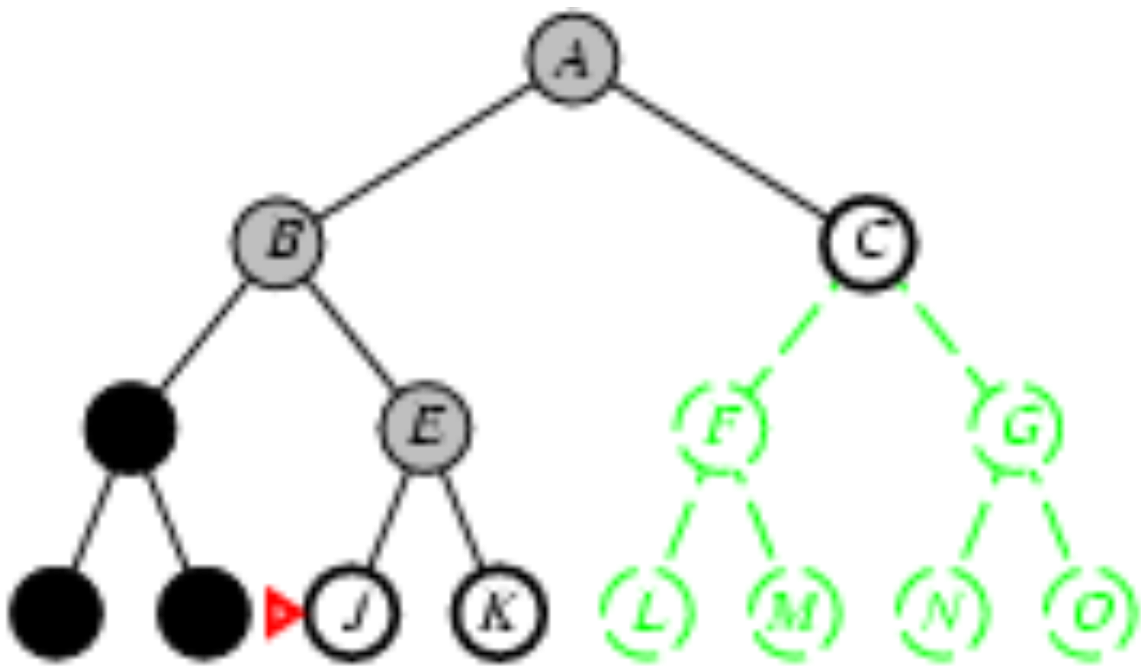
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



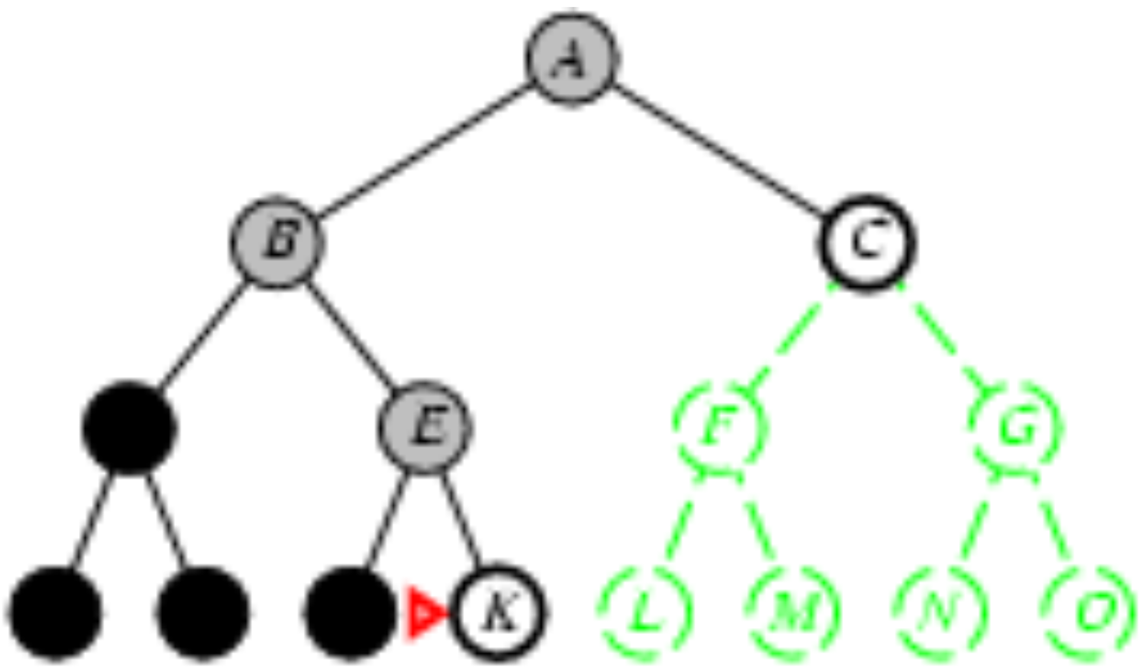
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



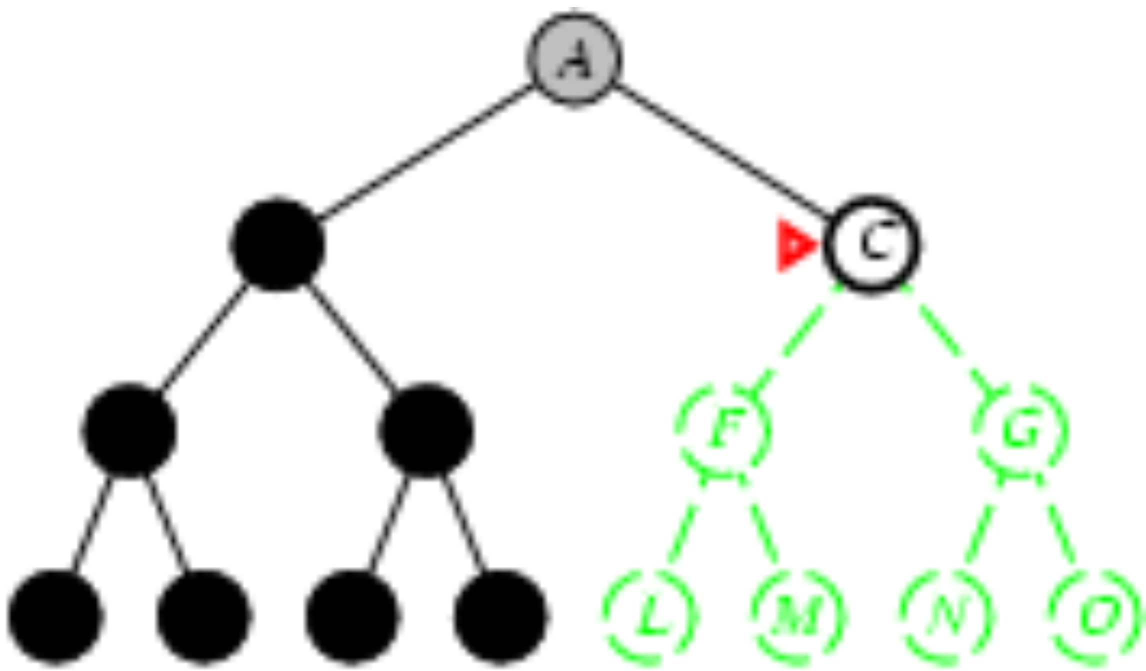
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



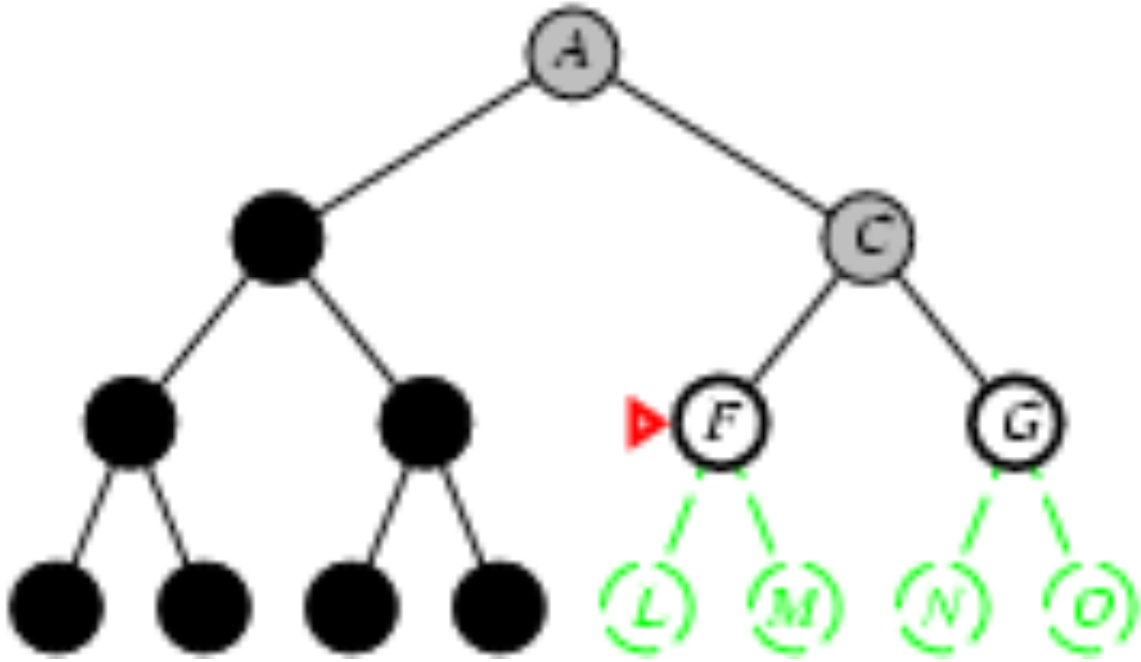
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



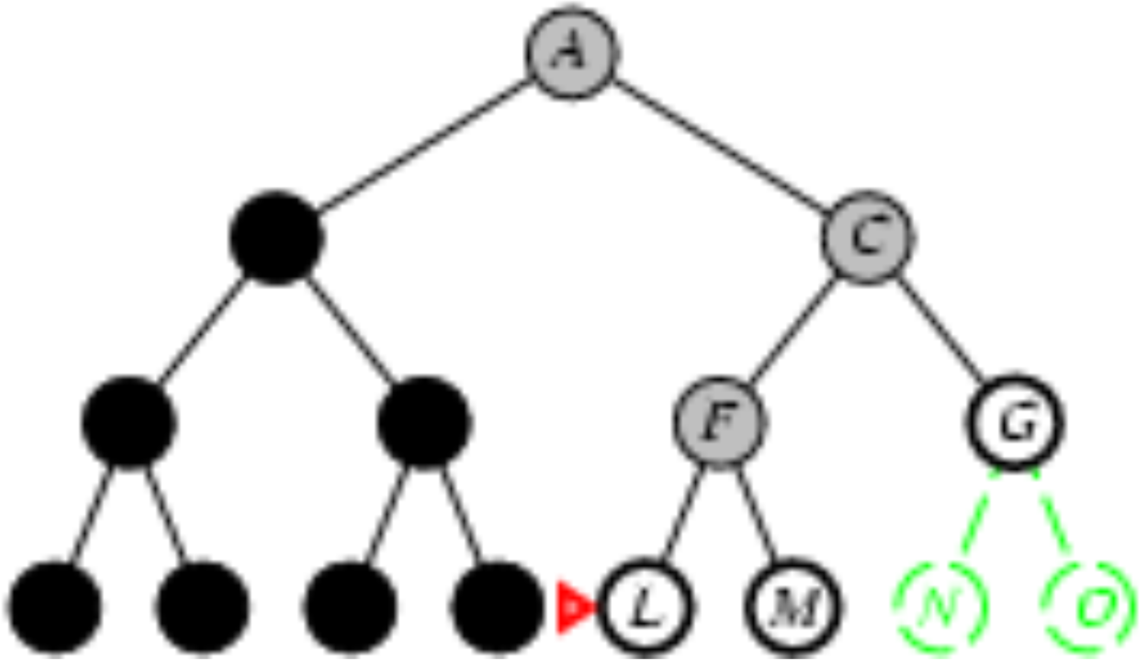
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



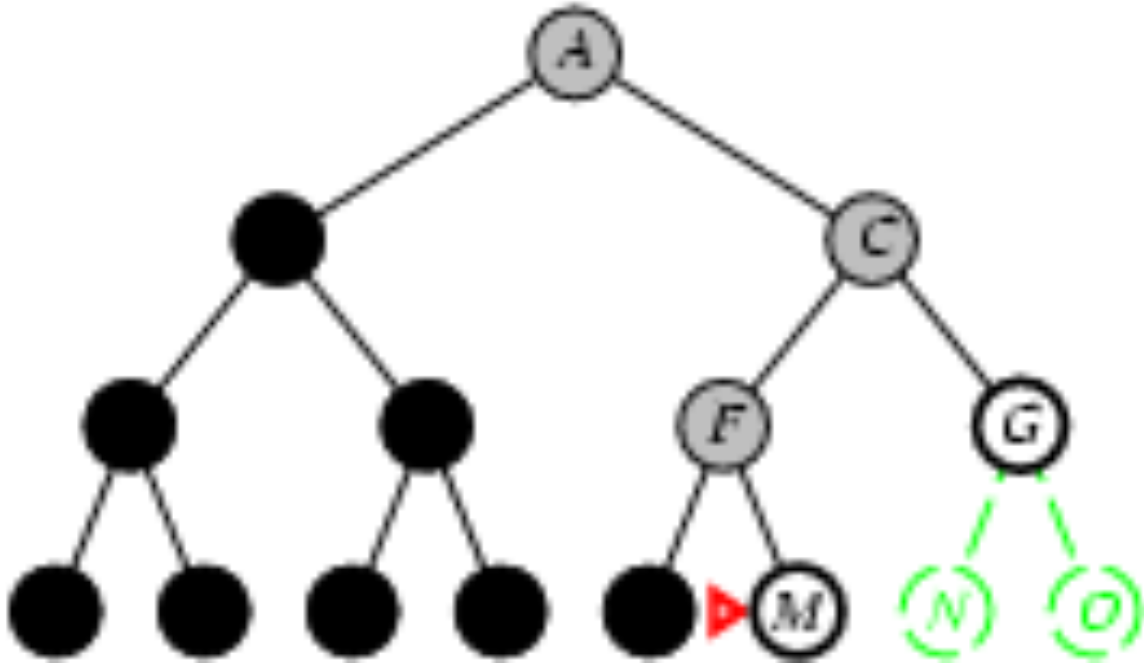
# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



# Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front



# Properties of depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
    → complete in finite spaces
- Time? $O(b^m)$: terrible if $m$ is much larger than $d$
  - but if solutions are dense, may be much faster than breadth-first
- Space? $O(bm)$
- Optimal? No

# Depth-limited search

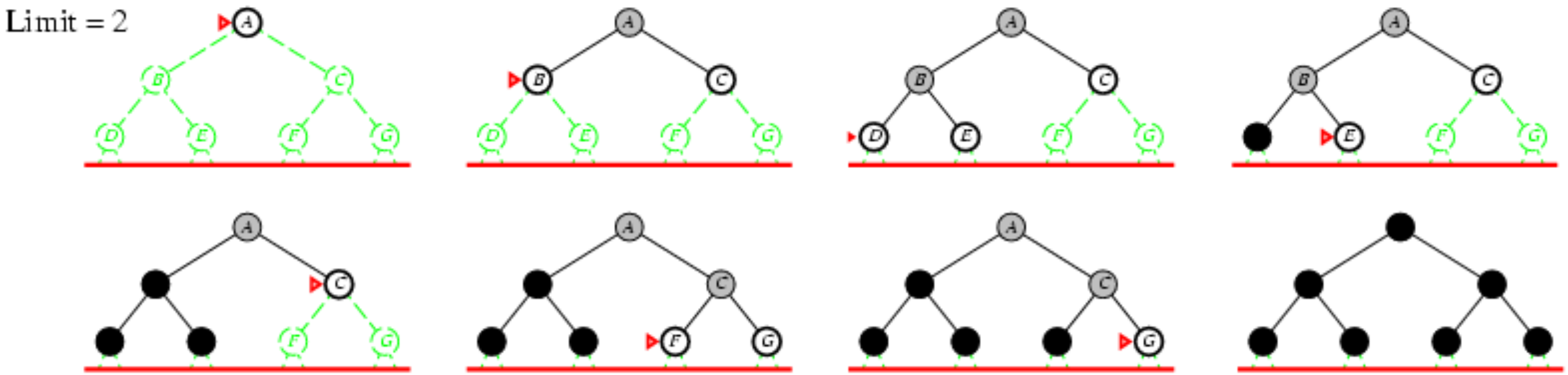
- = depth-first search with depth limit l,

i.e., nodes at depth l have no successor

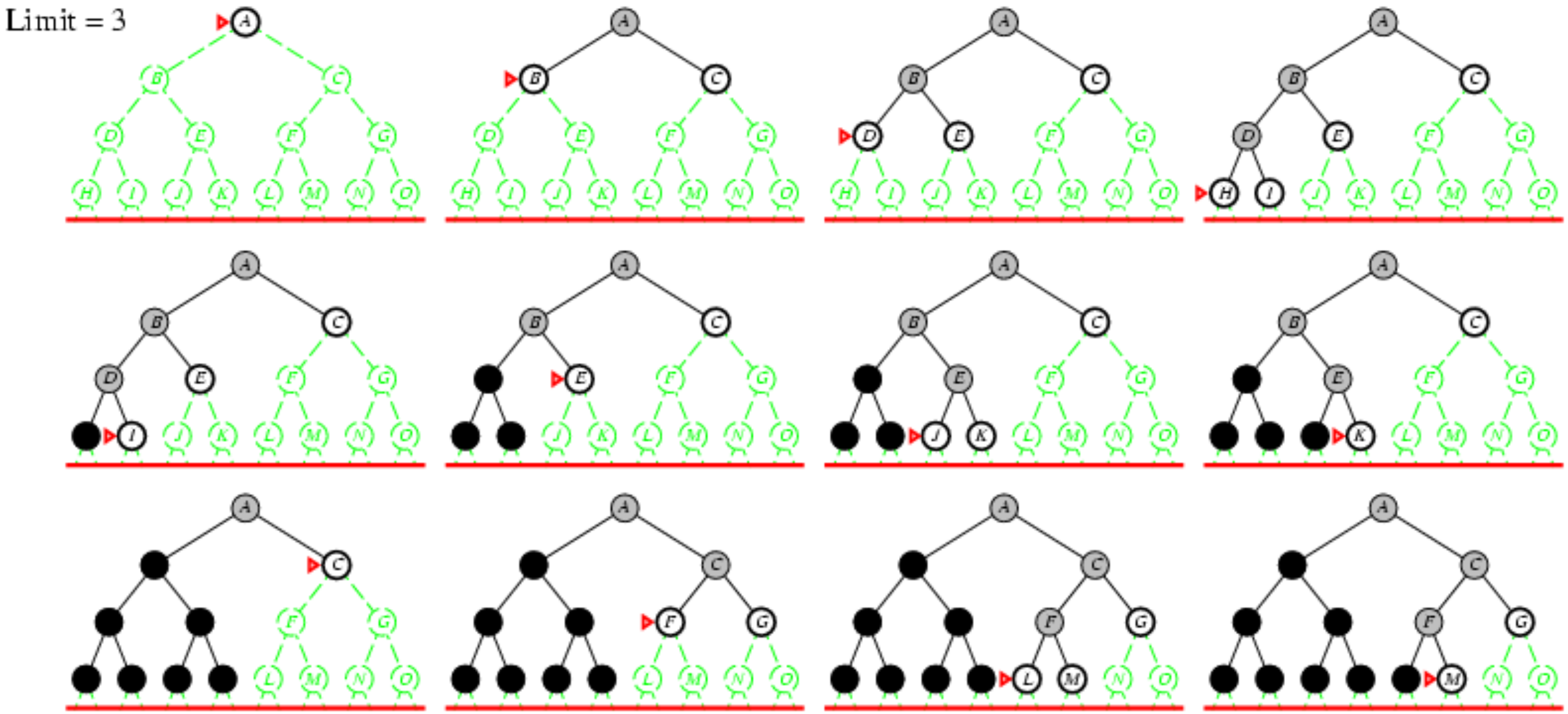
# Iterative deepening search $l$ =0

# Iterative deepening search *l* =1

# Iterative deepening search *l* =2



# Iterative deepening search *l* =3



# Iterative deepening search

- Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:

$$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

$$N_{IDS} = (d+1)b^0 + d\ b\text{^}^1 + (d-1)b\text{^}^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For $b = 10$, $d = 5$,
  - $N_{DLS}$ = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111
  - $N_{IDS}$ = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456

- Overhead = (123,456 - 111,111)/111,111 = 11%

# Properties of iterative deepening search

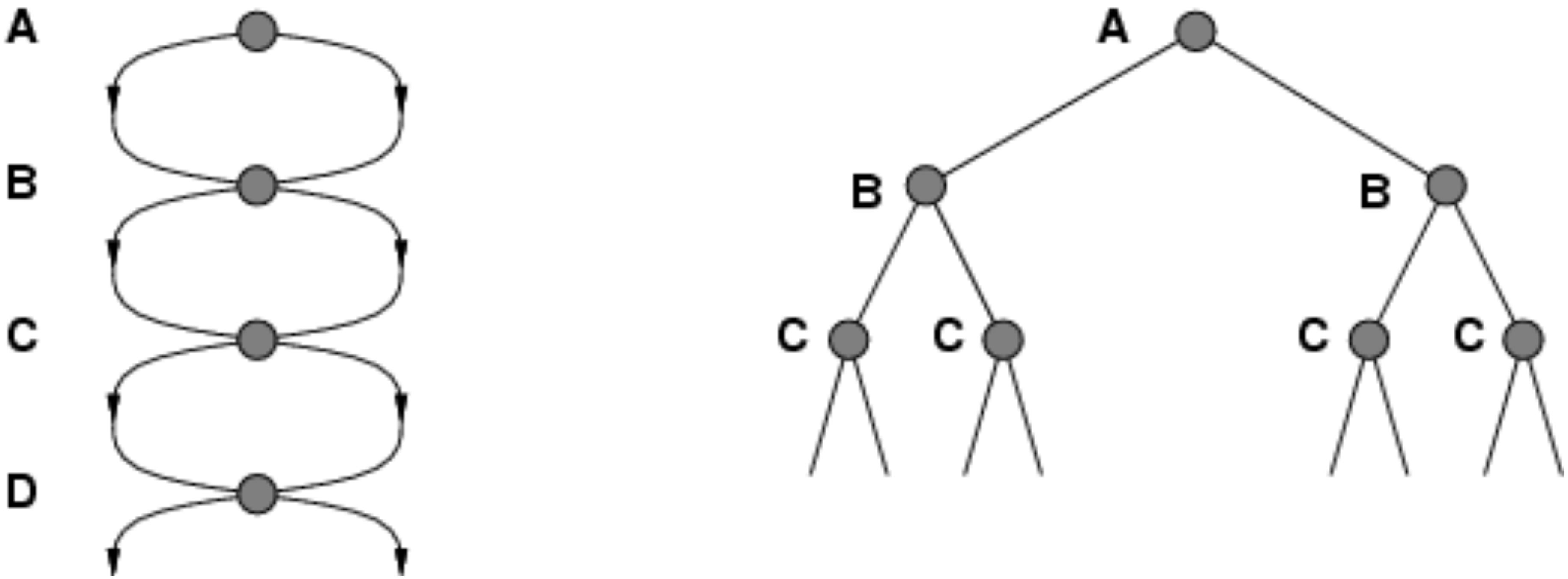
- Complete? Yes
- Time? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1

# Summary of algorithms

| Criterion | Breadth-First | Uniform-Cost | Depth-First | Depth-Limited | Iterative Deepening |
|---|---|---|---|---|---|
| Complete? | Yes | Yes | No | No | Yes |
| Time | $O(b^{d+1})$ | $O(b^{\lceil C^*/\epsilon \rceil})$ | $O(b^m)$ | $O(b^l)$ | $O(b^d)$ |
| Space | $O(b^{d+1})$ | $O(b^{\lceil C^*/\epsilon \rceil})$ | $O(bm)$ | $O(bl)$ | $O(bd)$ |
| Optimal? | Yes | Yes | No | No | Yes |

# Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!



# Outline

- Introduction
- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms
  - Uninformed search strategies
  - **Informed search strategies**
- Conclusions

# Informed search strategies

- Use the problem-specific knowledge beyond the definition of the problem itself to find more efficient solutions that uninformed strategy
  - Best-first search
    - Greedy best-first search
    - $A^*$ search
  - Heuristics
  - Local search algorithms
    - Hill-climbing search
    - Simulated annealing search
    - Local beam search
    - Genetic algorithms

# Best-first search

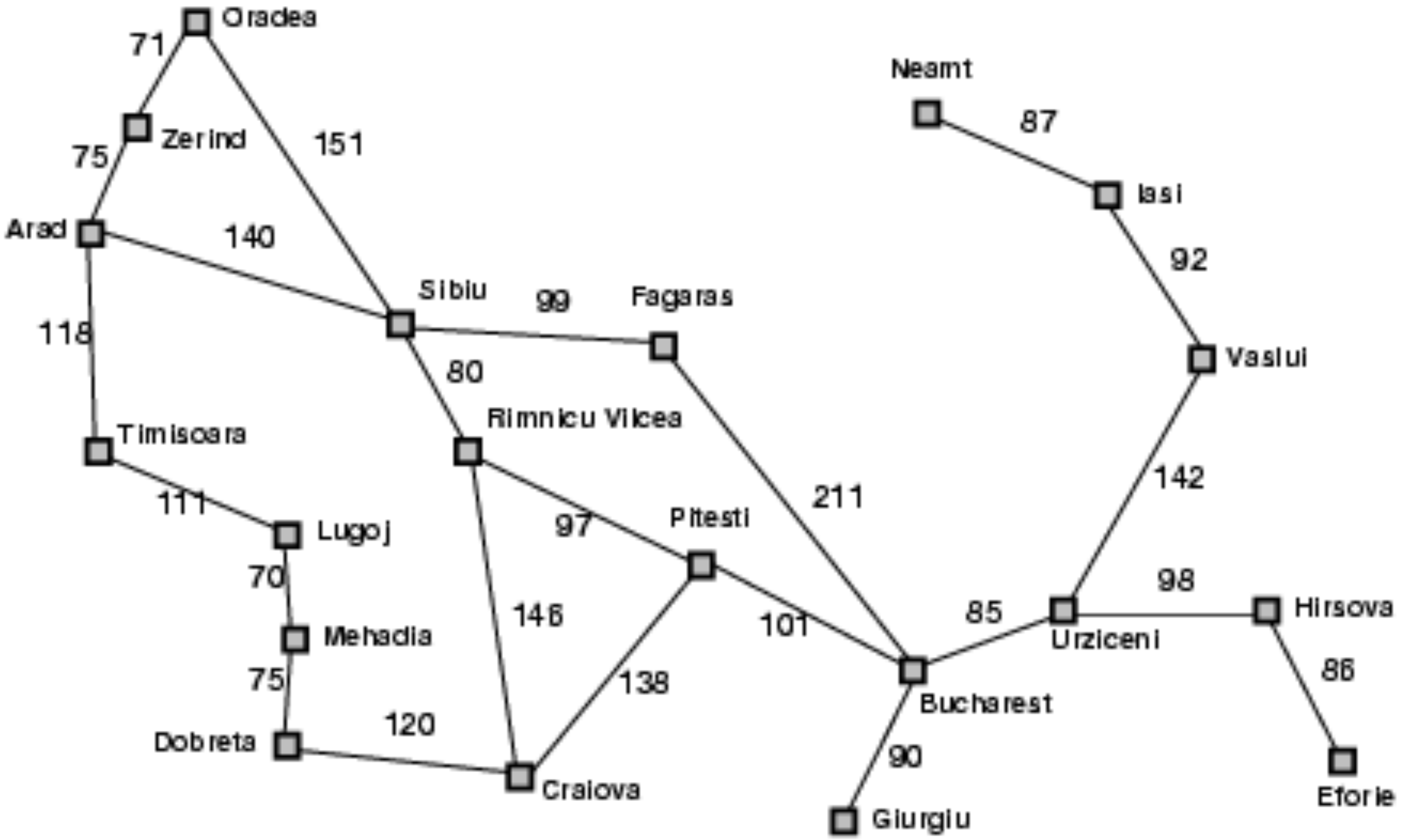
- Idea: use an evaluation function *f(n)* for each node
  - estimate of "desirability"
  - Expand most desirable unexpanded node

- Implementation:

  Order the nodes in fringe in decreasing order of desirability

- Special cases:
  - greedy best-first search
  - $A^*$ search

# Romania with step costs in km




Straight–line distance
to Bucharest

| | |
|---|---|
| **Arad** | 366 |
| **Bucharest** | 0 |
| **Craiova** | 160 |
| **Dobreta** | 242 |
| **Eforie** | 161 |
| **Fagaras** | 176 |
| **Giurgiu** | 77 |
| **Hirsova** | 151 |
| **Iasi** | 226 |
| **Lugoj** | 244 |
| **Mehadia** | 241 |
| **Neamt** | 234 |
| **Oradea** | 380 |
| **Pitesti** | 10 |
| **Rimnicu Vilcea** | 193 |
| **Sibiu** | 253 |
| **Timisoara** | 329 |
| **Urziceni** | 80 |
| **Vaslui** | 199 |
| **Zerind** | 374 |

# Informed search strategies

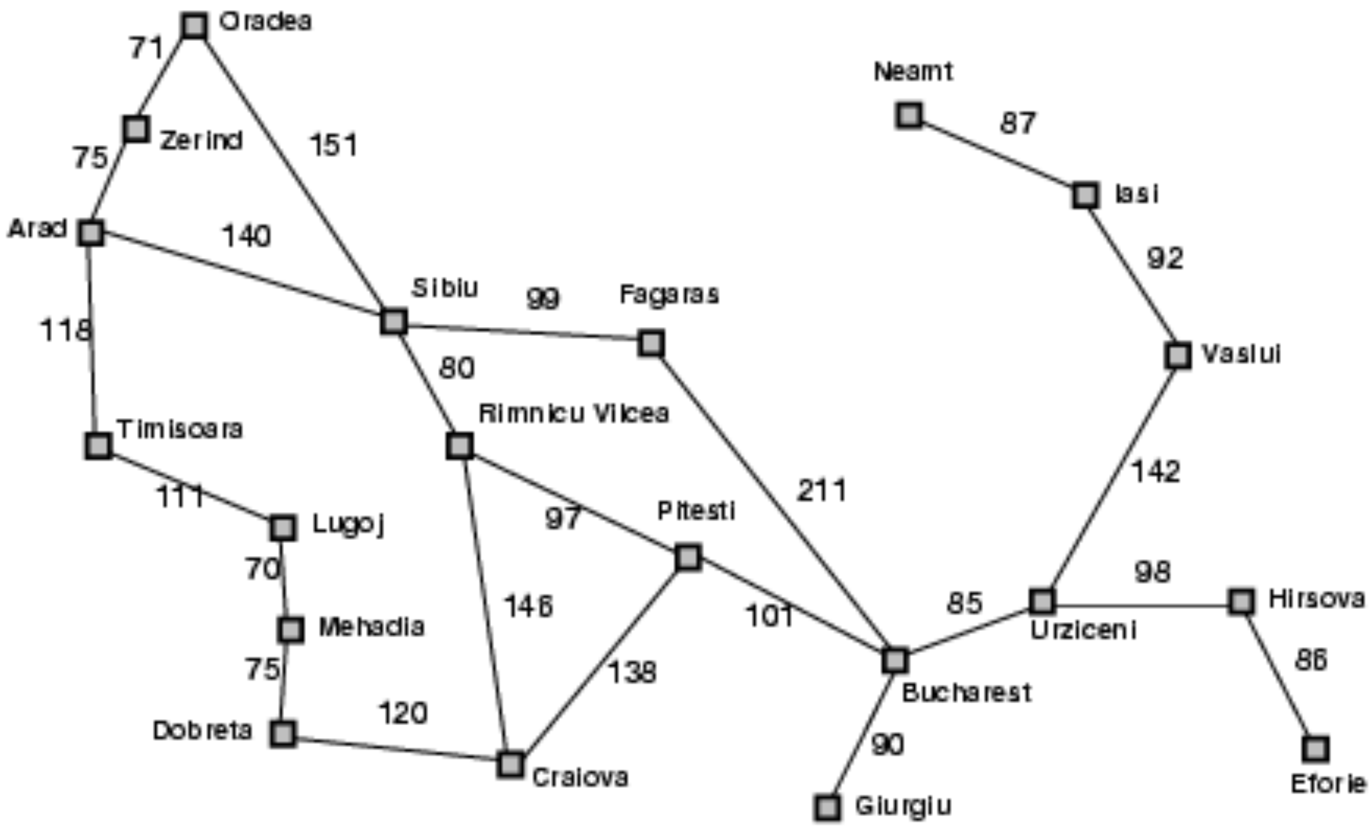
- Best-first search
  - **Greedy best-first search**
  - $A^*$ search
- Heuristics
- Local search algorithms
  - Hill-climbing search
  - Simulated annealing search
  - Local beam search
  - Genetic algorithms

# Greedy best-first search

- Evaluation function $f(n) = h(n)$ (heuristic) = estimate of cost from $n$ to *goal*
- Greedy best-first search expands the node that appears to be closest to the goal
- Implementation: as a priority queue to keep the fringe in ascending order of f-values
- e.g., $h_{SLD}(n)$ = straight-line distance from $n$ to Bucharest

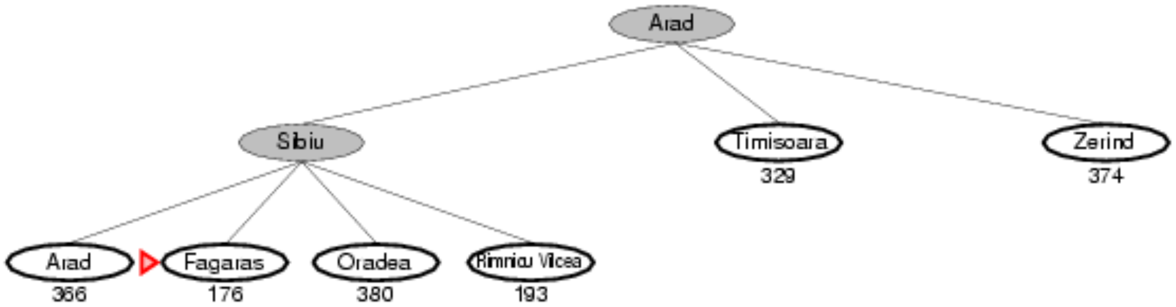
# Greedy best-first search example

# Greedy best-first search example

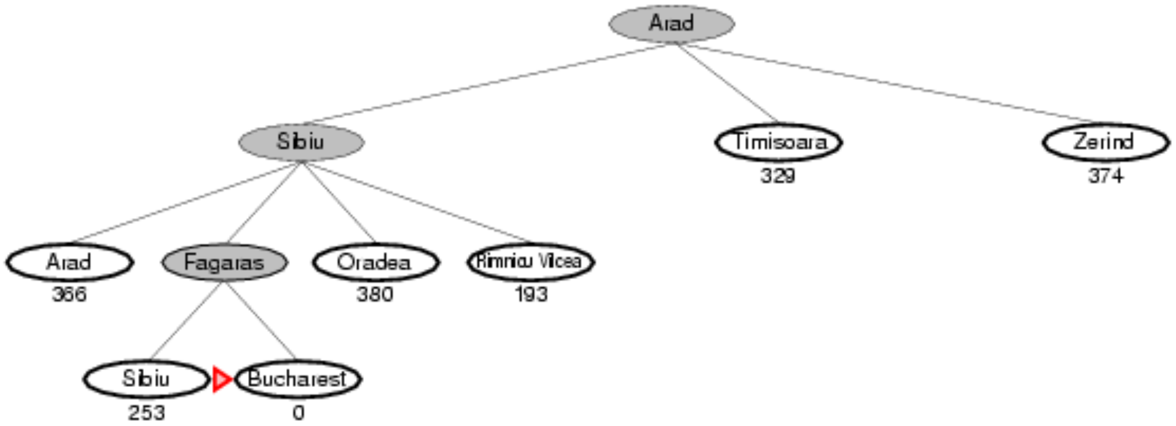




| Straight-line distance to Bucharest | |
|---|---|
| **Arad** | 366 |
| **Bucharest** | 0 |
| **Craiova** | 160 |
| **Dobreta** | 242 |
| **Eforie** | 161 |
| **Fagaras** | 176 |
| **Giurgiu** | 77 |
| **Hirsova** | 151 |
| **Iasi** | 226 |
| **Lugoj** | 244 |
| **Mehadia** | 241 |
| **Neamt** | 234 |
| **Oradea** | 380 |
| **Pitesti** | 10 |
| **Rimnicu Vilcea** | 193 |
| **Sibiu** | 253 |
| **Timisoara** | 329 |
| **Urziceni** | 80 |
| **Vaslui** | 199 |
| **Zerind** | 374 |

# Greedy best-first search example



# Greedy best-first search example



# Properties of greedy best-first search

- Complete? No – can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt →
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$ -- keeps all nodes in memory
- Optimal? No

Similar to depth-first search

Each state has b successors (branching factor)

d is the depth of the shallowest solution

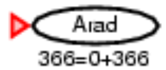
# Informed search strategies

- Best-first search
  - Greedy best-first search
  - **$A^*$ search**
- Heuristics
- Local search algorithms
  - Hill-climbing search
  - Simulated annealing search
  - Local beam search
  - Genetic algorithms

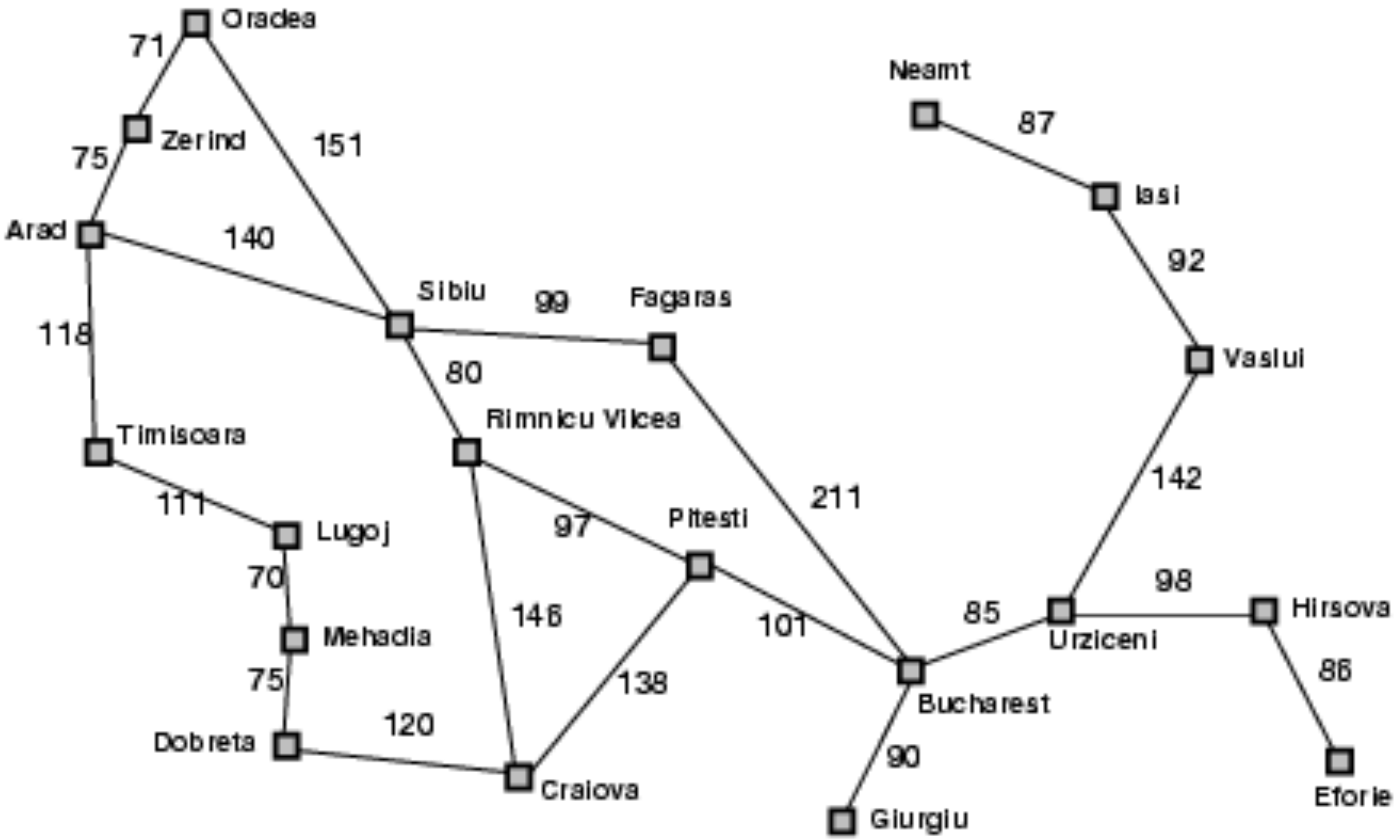
# A$^*$ search

- Idea: avoid expanding paths that are already expensive
- Evaluation function $f(n) = g(n) + h(n)$
  - $g(n)$ = cost so far to reach $n$
  - $h(n)$ = estimated cost from $n$ to goal
  - $f(n)$ = estimated total cost of path through $n$ to goal
- A * is optimal if h(n) is an admissible heuristic such that h(n) never overestimates the cost to reach the goal

# A* search example



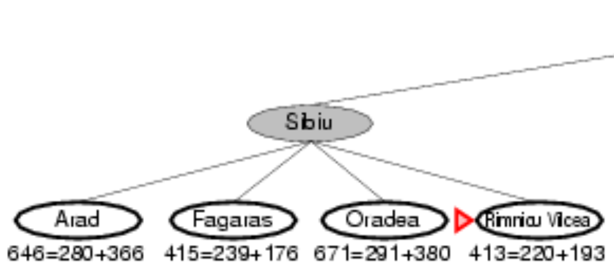
# A* search example

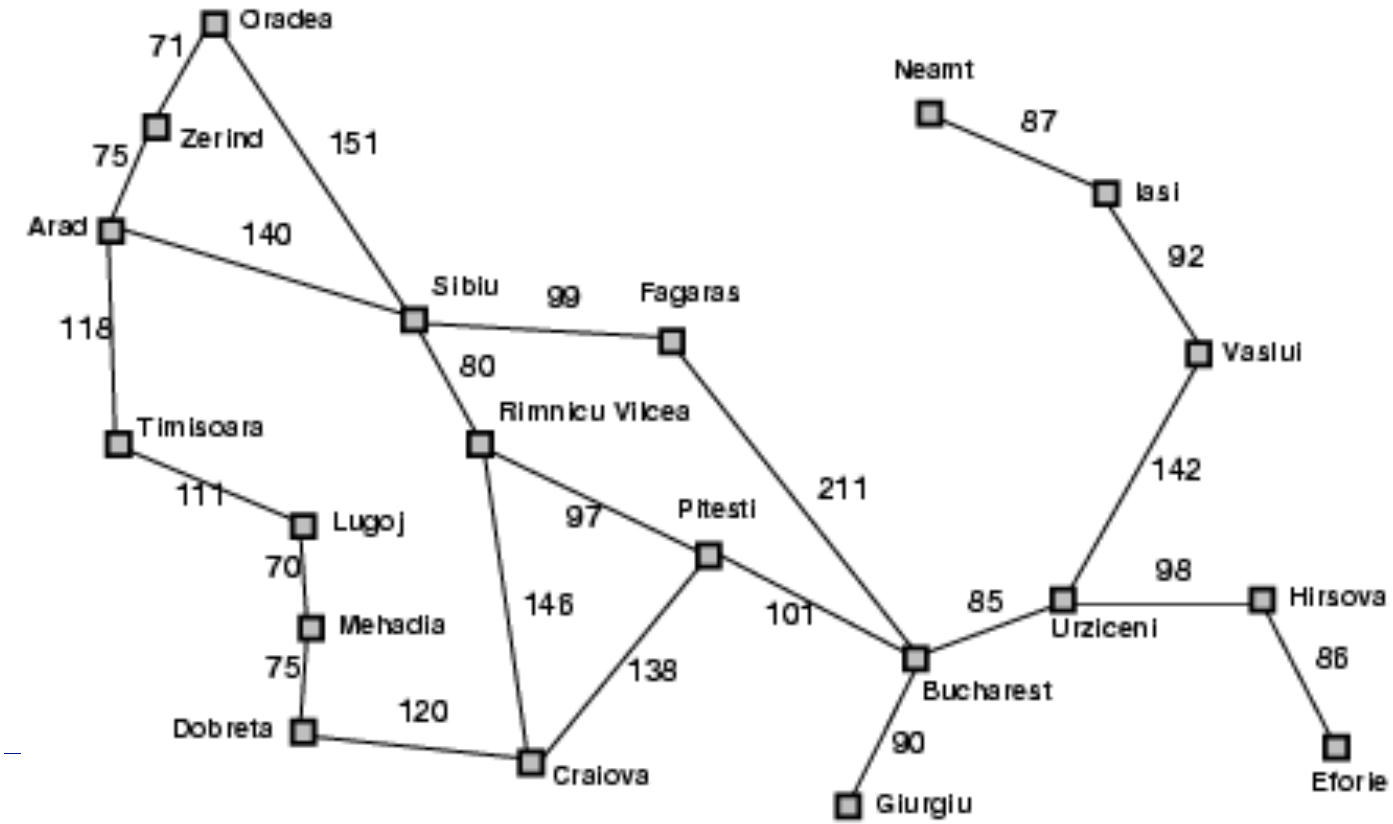




Straight–line distance to Bucharest

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

# A* search example

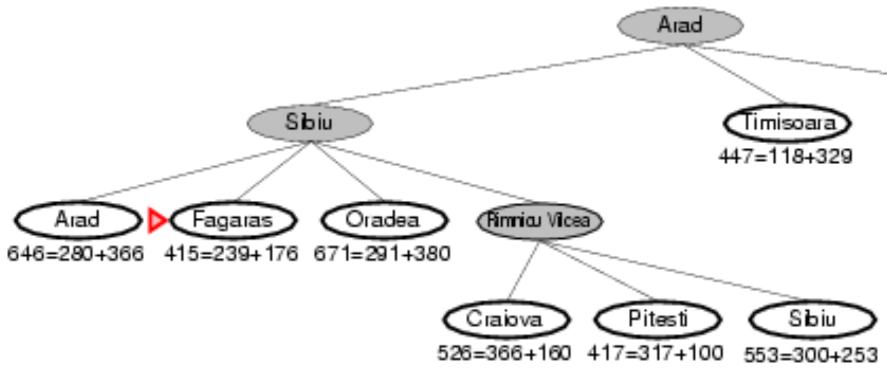


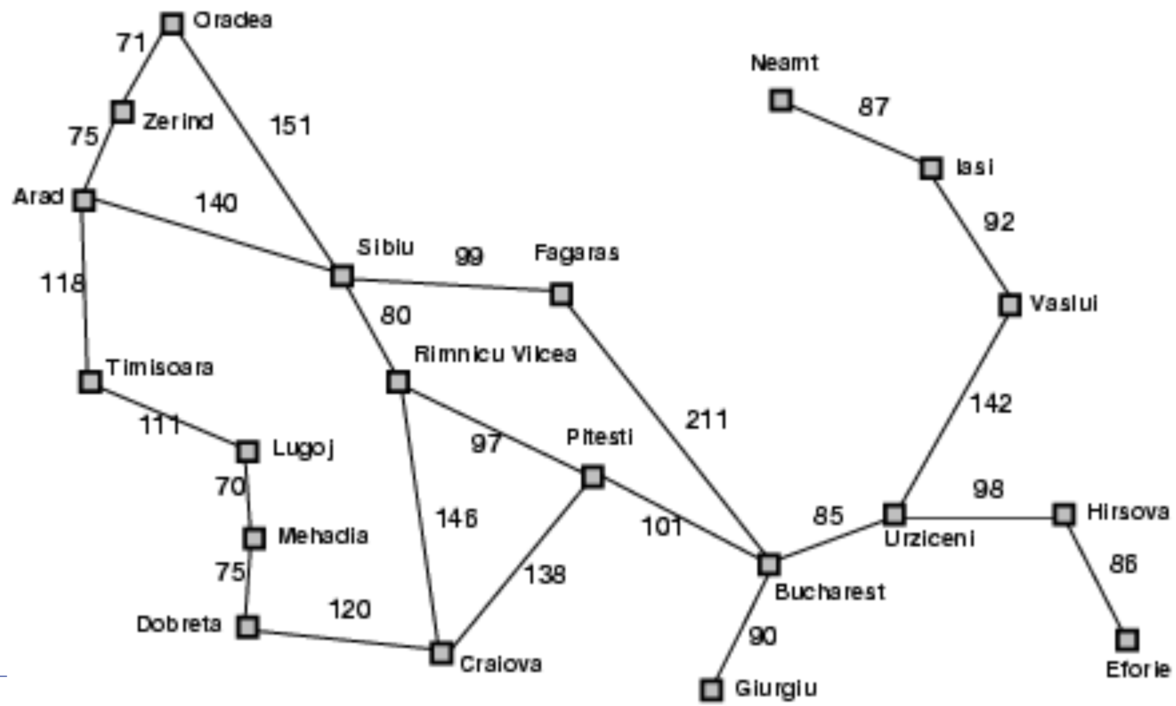




Straight–line distance to Bucharest

| | |
|---|---|
| **Arad** | 366 |
| **Bucharest** | 0 |
| **Craiova** | 160 |
| **Dobreta** | 242 |
| **Eforie** | 161 |
| **Fagaras** | 176 |
| **Giurgiu** | 77 |
| **Hirsova** | 151 |
| **Iasi** | 226 |
| **Lugoj** | 244 |
| **Mehadia** | 241 |
| **Neamt** | 234 |
| **Oradea** | 380 |
| **Pitesti** | 10 |
| **Rimnicu Vilcea** | 193 |
| **Sibiu** | 253 |
| **Timisoara** | 329 |
| **Urziceni** | 80 |
| **Vaslui** | 199 |
| **Zerind** | 374 |

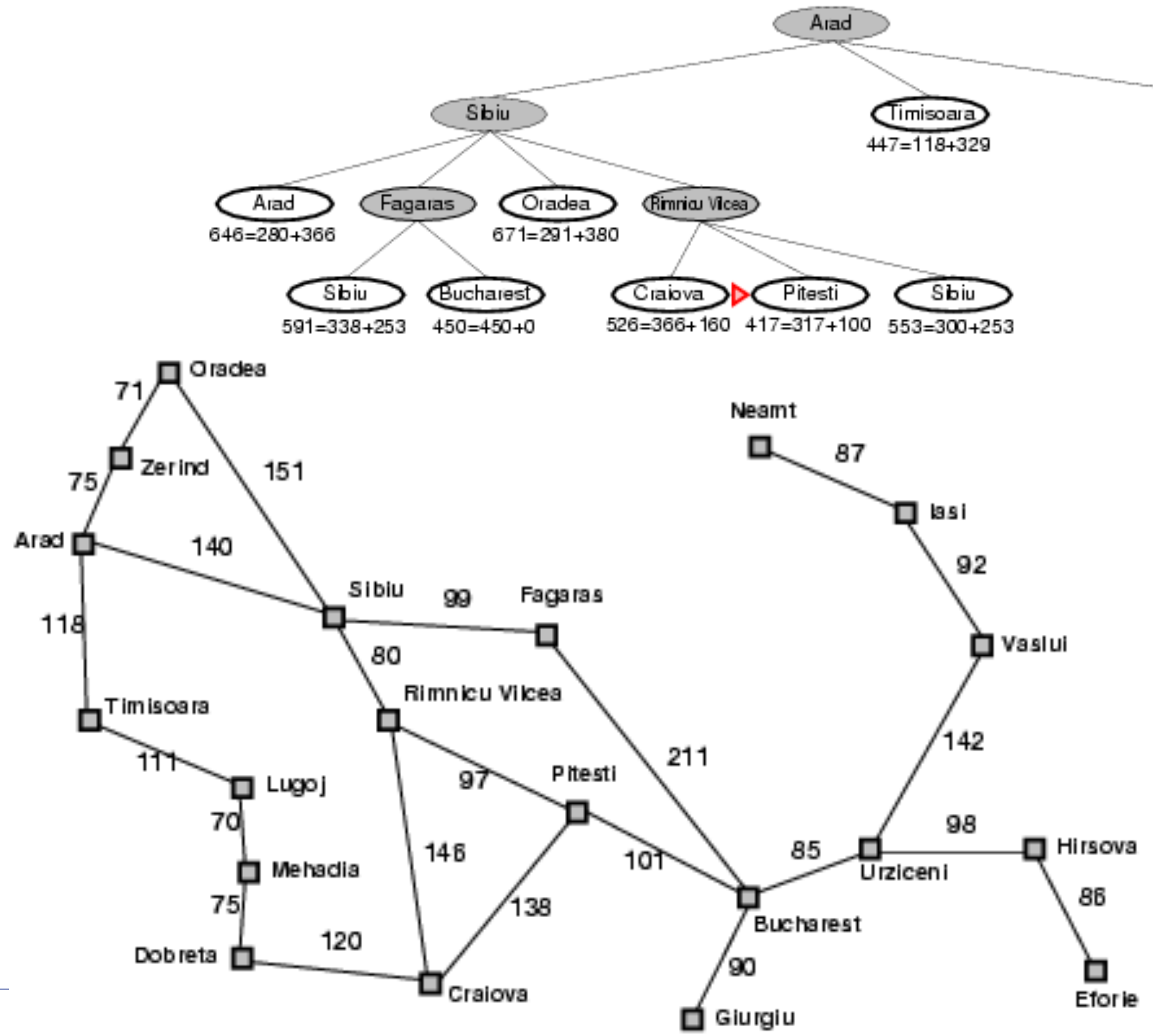
# A* search example



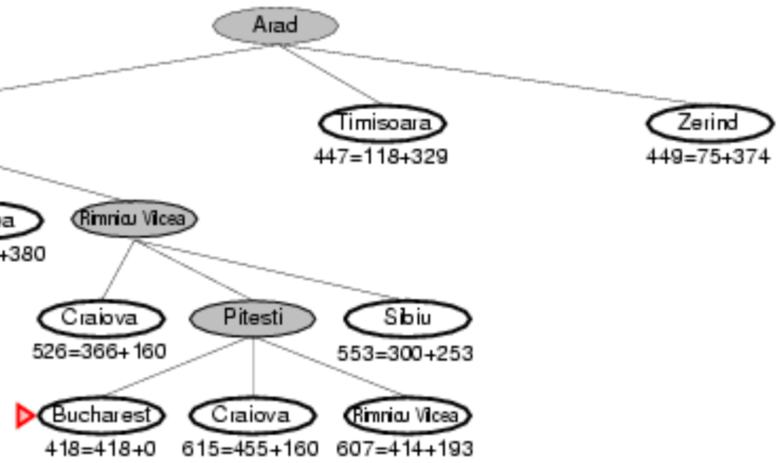

| Straight–line distance to Bucharest | |
|---|---|
| **Arad** | 366 |
| **Bucharest** | 0 |
| **Craiova** | 160 |
| **Dobreta** | 242 |
| **Eforie** | 161 |
| **Fagaras** | 176 |
| **Giurgiu** | 77 |
| **Hirsova** | 151 |
| **Iasi** | 226 |
| **Lugoj** | 244 |
| **Mehadia** | 241 |
| **Neamt** | 234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

# A* search example




| Straight–line distance to Bucharest | |
|---|---|
| **Arad** | 366 |
| **Bucharest** | 0 |
| **Craiova** | 160 |
| **Dobreta** | 242 |
| **Eforie** | 161 |
| **Fagaras** | 176 |
| **Giurgiu** | 77 |
| **Hirsova** | 151 |
| **Iasi** | 226 |
| **Lugoj** | 244 |
| **Mehadia** | 241 |
| **Neamt** | 234 |
| **Oradea** | 380 |
| **Pitesti** | 10 |
| **Rimnicu Vilcea** | 193 |
| **Sibiu** | 253 |
| **Timisoara** | 329 |
| **Urziceni** | 80 |
| **Vaslui** | 199 |
| **Zerind** | 374 |

# A* search example



# Informed search strategies

- Best-first search
- Greedy best-first search
- $A^*$ search
- **Heuristics**
- Local search algorithms
  - Hill-climbing search
  - Simulated annealing search
  - Local beam search
  - Genetic algorithms

# Admissible heuristics

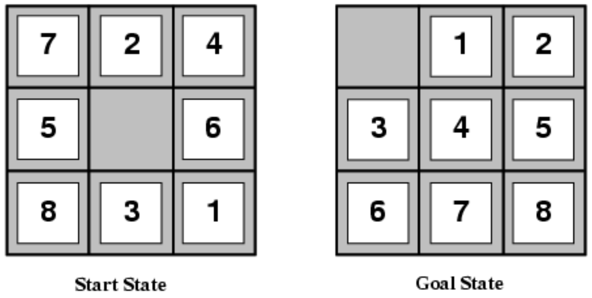
- A heuristic $h(n)$ is admissible if for every node $n$, $h(n) \le h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from $n$
- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
- Example: $h_{SLD}(n)$ (never overestimates the actual road distance)
- Theorem: If $h(n)$ is admissible, A* using `TREE-SEARCH` is optimal

# Properties of A*

- Complete? Yes (unless there are infinitely many nodes with f ≤ *f(G)* )
- Time? Exponential
- Space? Keeps all nodes in memory
- Optimal? Yes

# Admissible heuristics for 8-puzzle

- The average cost for the 8-puzzle are approx. 22 steps. Here are 26 steps.
- Branching factor is approx. 3
  - Empty in the middle, 4 mov
  - Empty in the corner, 2 mov
  - Rest cases, 3 mov
- Depth-first search will look $3^{22}$ states
- If we keep track of repeated states, we could reduce to 170.000
- In the15-puzle = $10^{13}$

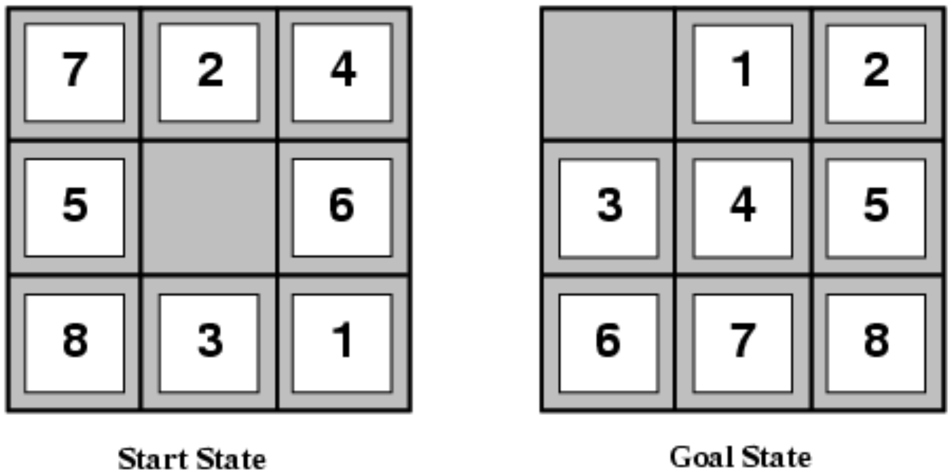


# Admissible heuristics for 8-puzzle

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance

(i.e., + horizontal and vertical distance from desired location of each tile)



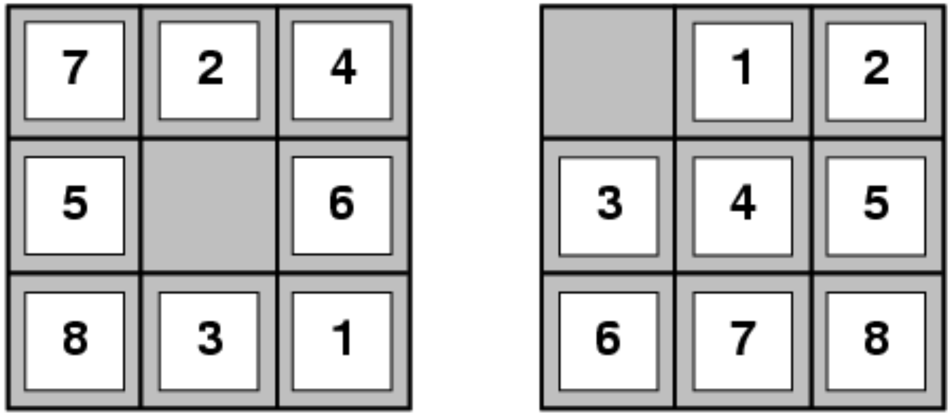

- $h_1(S) = ?$
- $h_2(S) = ?$

# Admissible heuristics

E.g., for the 8-puzzle:

- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)




Start State Goal State

- $h_1(S)$ = ? 8
- $h_2(S)$ = ? 3+1+2+2+2+3+3+2 = 18

# Dominance

- If $h_2(n) \geq h_1(n)$ for all $n$ (both admissible) then $h_2$ dominates $h_1$
- $h_2$ is better for search

- Typical search costs (average number of nodes expanded):

- $d=12$ IDS = 3,644,035 nodes
  $A^*(h_1)$ = 227 nodes
  $A^*(h_2)$ = 73 nodes
- $d=24$ IDS = too many nodes
  $A^*(h_1)$ = 39,135 nodes
  $A^*(h_2)$ = 1,641 nodes

# Relaxed problems

- A problem with fewer restrictions on the actions is called a relaxed problem
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then $h_1(n)$ gives the shortest solution
- If the rules are relaxed so that a tile can move to any adjacent square, then $h_2(n)$ gives the shortest solution

# Informed search strategies

- Best-first search
- Greedy best-first search
- $A^*$ search
- Heuristics
- **Local search algorithms**
  - Hill-climbing search
  - Simulated annealing search
  - Local beam search
  - Genetic algorithms

# Local search algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
- State space = set of "complete" configurations. Find configuration satisfying constraints, e.g., n-queens
- In such cases, we can use local search algorithms
- Keep a single "current" state, try to improve it
- Work with one current state and generally moves to the neighboring state
- The paths followed by the search are not retained
  - They use little memory
  - You can find reasonable solutions in large state spaces or infinite

# Example: *n*-queens

- Put $n$ queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

# Informed search strategies

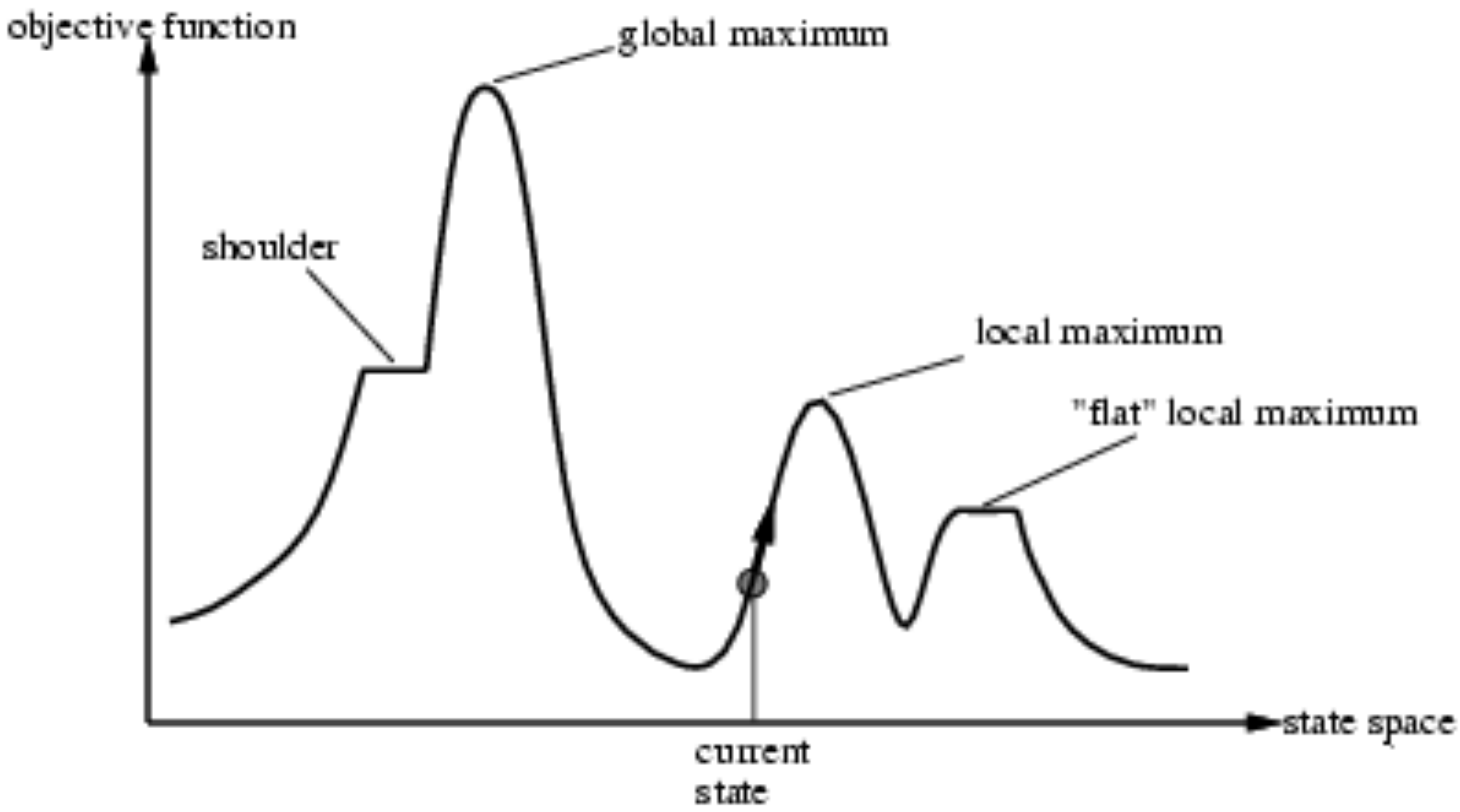
- Best-first search
- Greedy best-first search
- $A^*$ search
- Heuristics
- Local search algorithms
  - **Hill-climbing search**
  - Simulated annealing search
  - Local beam search
  - Genetic algorithms

# Hill-climbing search

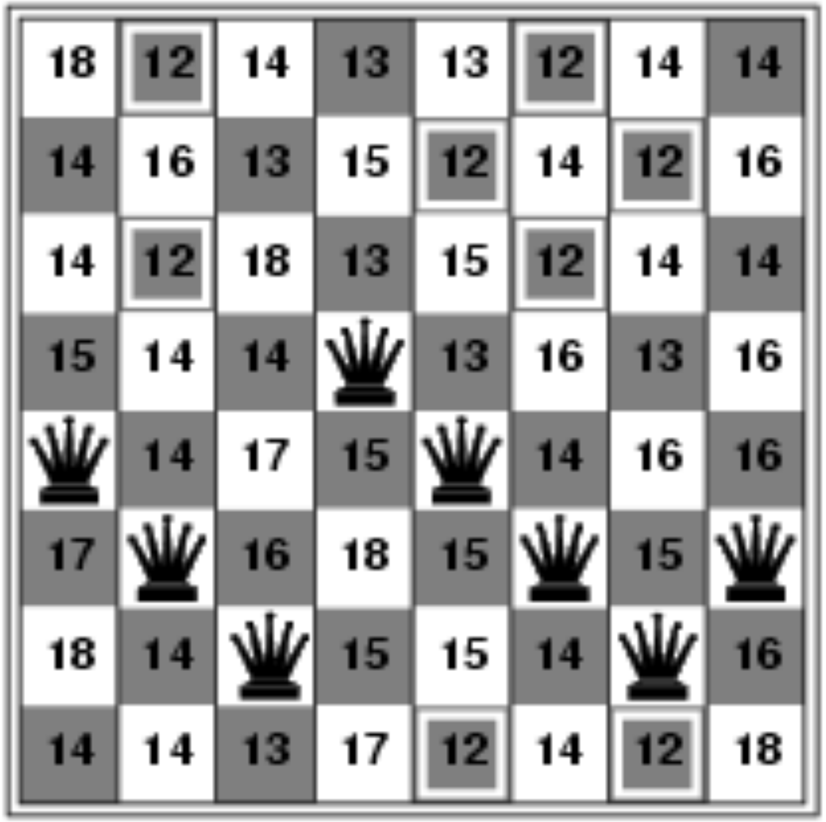
- It's just a loop that moves in the direction of increasing value
  - Ends when it reaches a peak where no neighbor has a higher value
  - The search tree is not kept, just a data structure of the current node to check the goal condition and its objective function value
- "Like climbing Everest in thick fog with amnesia"

# Hill-climbing search

- Problem: depending on initial state, can get stuck in local maxima

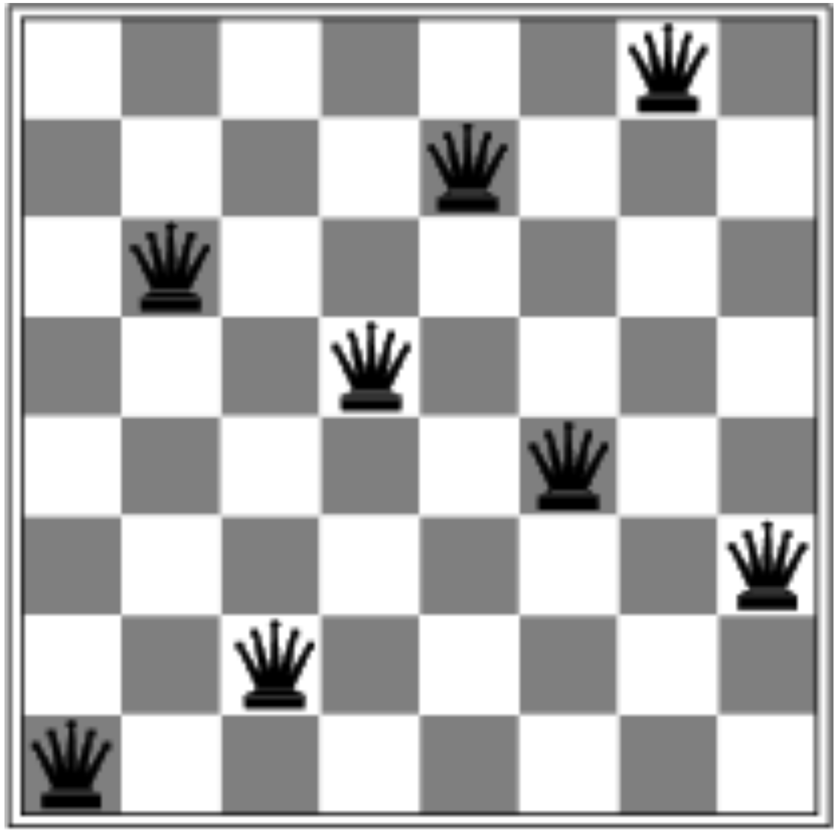


# Hill-climbing search: 8-queens problem




- *h* = number of pairs of queens that are attacking each other, either directly or indirectly
- *h = 17* for the above state
- The figure also shows the values of all successors, top successors have h = 12

# Hill-climbing search: 8-queens problem



- A local minimum with h = 1 (obtained in 5 steps)

# Hill-climbing search

- The algorithm gets stuck for several reasons::
  - Local Maximum: it is a peak that is higher than each of its neighbours, but lower than the maximum overall
  - Ridges: cause a sequence of local maxima that make navigation difficult
  - Plateau (flat): can lead to a local maximum  where there is no ascendant exit or a terrace to advance
- In the 8-queens, it gets stack in 86% and solve 14% cases
- If we allow lateral movements with the hope that we find a terrace (limiting them if reach a local maximum, e.g.100):→ 94% success
- Variants:
  - Stochastically (randomly chooses upward movements)
  - Random restart (the initial states are generated randomly)

# Informed search strategies

- Best-first search
- Greedy best-first search
- $A^*$ search
- Heuristics
- Local search algorithms
  - Hill-climbing search
  - **Simulated annealing search**
  - Local beam search
  - Genetic algorithms

# Simulated annealing search

- Simulated annealing is the process of tempering or hardening metals by heating and then cooling them gradually
- Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency
- It combines hill-climbing with random generation successor
- One can prove: If $T$ decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1
- Widely used in VLSI layout, airline scheduling, etc

# Informed search strategies

- Best-first search
- Greedy best-first search
- A$^*$ search
- Heuristics
- Local search algorithms
  - Hill-climbing search
  - Simulated annealing search
  - **Local beam search**
  - Genetic algorithms

# Local beam search

- Idea: Keep track of *k* states rather than just one
  - Start with *k* randomly generated states
  - At each iteration, all the successors of all *k* states are generated
  - If any one is a goal state, stop; else select the *k* best successors from the complete list and repeat
  - Alternatively stochastic LBS randomly choose k successors, with the probability of choosing a successor as an increasing function of its value

# Informed search strategies

- Best-first search
- Greedy best-first search
- $A^*$ search
- Heuristics
- Local search algorithms
  - Hill-climbing search
  - Simulated annealing search
  - Local beam search
  - **Genetic algorithms**

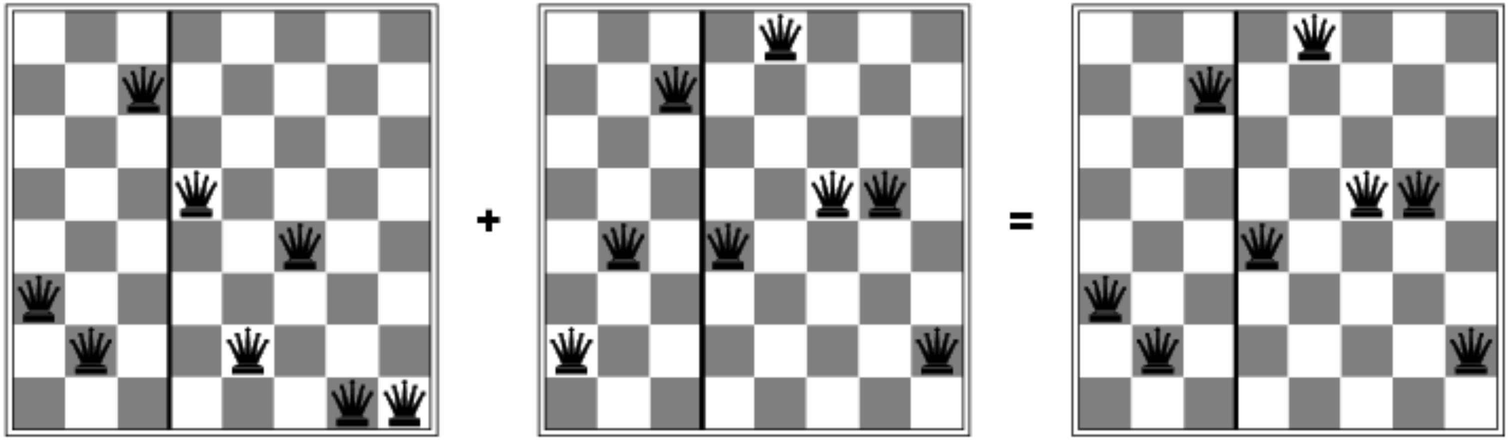
# Genetic algorithms

- A successor state is generated by combining two parent states
- Start with *k* randomly generated states (population)
- A state is represented as a string over a finite alphabet (often a string of 0s and 1s)
- Evaluation function (fitness function). Higher values for better states
- Produce the next generation of states by selection, crossover, and mutation

# Genetic algorithms

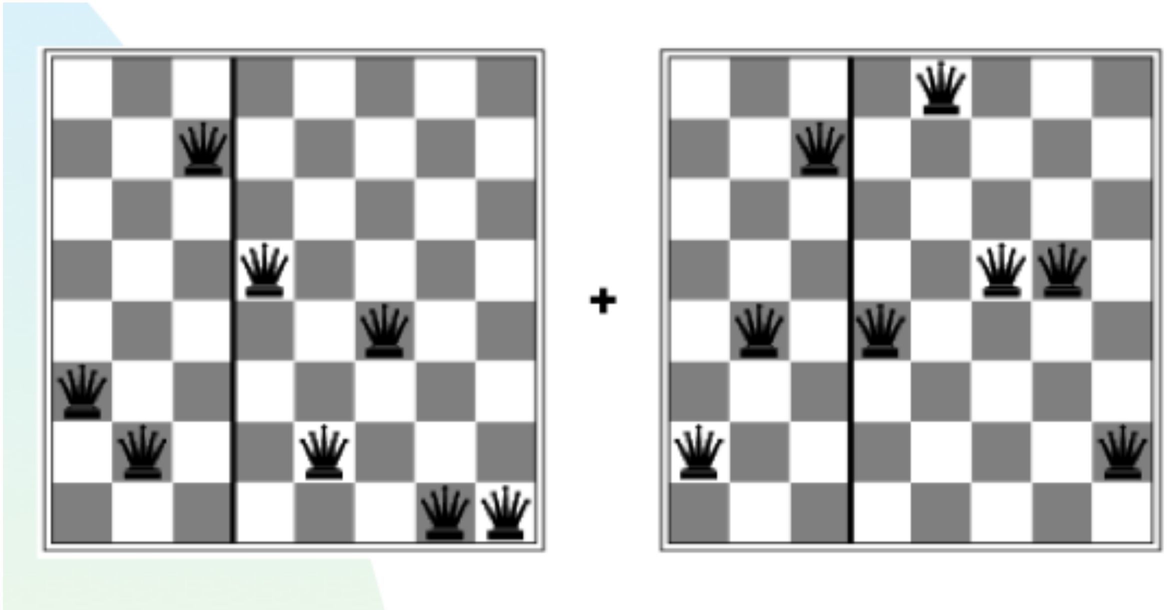

- The initial population is represented by chains of 8 digits representing states of 8 queens

# Genetic algorithms



- It corresponds to the 2 parents of figure (c)
  327 52411  (5 attacked queens)
  247 48552  (4 attacked queens)

# Genetic algorithms




- Fitness function: number of NON-attacking pairs of queens (min = 0, max = 8 × 7/2 = 28)
  - Right: 247 48552 (4 attacked queens) = 28 – 4 = 24

    24/(24+23+20+11) = 31%
  - Left: 327 52411 (5 attacked queens) = 28 – 5 = 23

    23/(24+23+20+11) = 29%

# Genetic algorithms


probabilities

- Crossing points are chosen and d) the offspring are created crossing the chain parental crosspoint
- In e) each position is subject to random mutation with a small probability

# Outline

- Introduction
- Problem-solving agents
- Problem formulation
- Problem types
- Example problems
- Basic search algorithms
- **Conclusions**

# Conclusions

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored

- Variety of uninformed and informed search strategies