CSCI 3230

Fundamentals of Artificial Intelligence

Chapter 3 (Sects 3.1–3.4)

PROBLEM SOLVING BY SEARCHING

importance

Outline

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

Problem-solving agents

Intelligent agents act to make the environment go through a sequence of states that maximizes the performance measure:

- 1st step: Goal formation, based on the current situation.
  - A set of world states that satisfy the goal. Actions cause transitions between world states, so the agent has to search for appropriate actions. (within constraints) e.g. goto Shatin
- 2nd step: Problem formulation is to decide what actions and states to consider. i.e. modeling -> problem (search) space
- 3rd step: Search – An agent with several immediate options chooses the best one by examining different possible sequences of actions. A search algorithm takes a problem as input and returns a solution (e.g. an action sequence or an optimal state).
- 4th step: Execution – The actions recommended are carried out.

Problem-solving agents 2

A simple problem-solving agent:

function Simple-Problem-Solving-Agent (*percept*) **returns** an action
  static: *seq*, an action sequence, initially empty
    state, some description of the current world state
    goal, a goal, initially null
    problem, a problem formulation
 state ← Update-State (*state*, *percept*)   //e.g. at Eng Bldg…at University Station….
 if *seq* is empty **then do**
  goal ← Formulate-Goal (*state*)
  problem ← Formulate-Problem (*state*, *goal*)
  seq ← Search (*problem*)
 action ← First (*seq*)
 seq ← Rest (*seq*)
 return *action*

Note: this is offline problem solving: solution executed “eyes closed”. Online problem solving involves acting interactively without complete knowledge.

Example: Romania

On holiday in Romania; currently in Arad.

Flight leaves tomorrow from Bucharest.

Formulate goal:

be in Bucharest

Formulate problem:

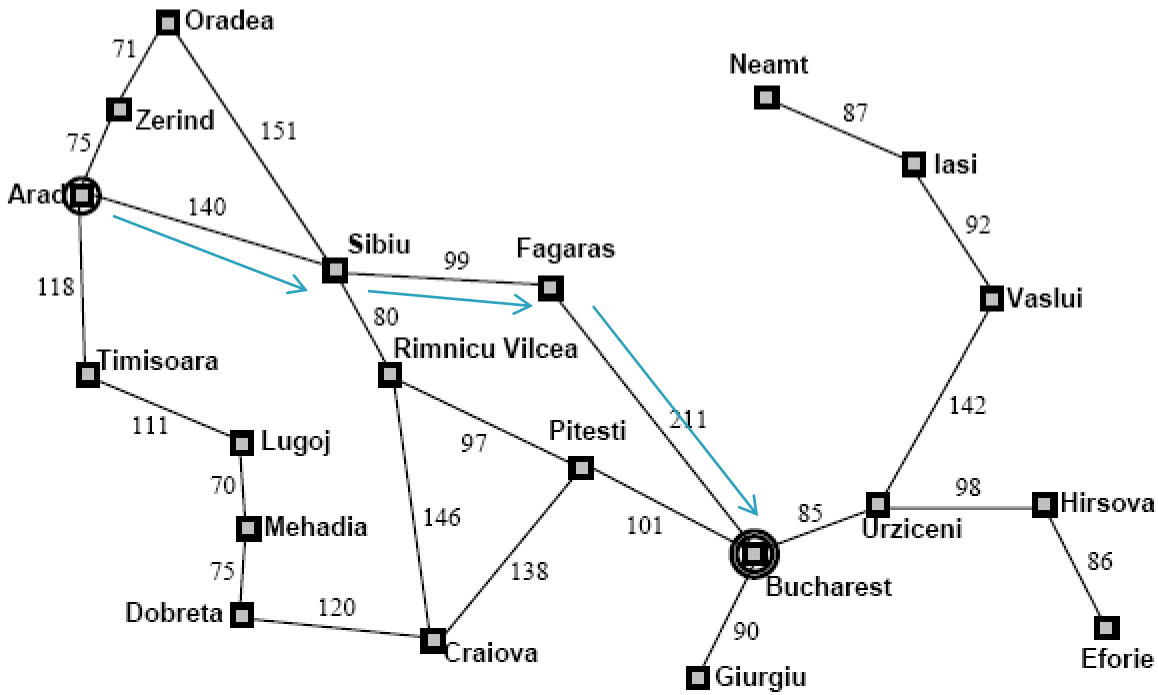
states: various cities

actions: drive between cities

Find solution:

sequence of cities, e.g. Arad, Sibiu, Fagaras, Bucharest

Example: Romania



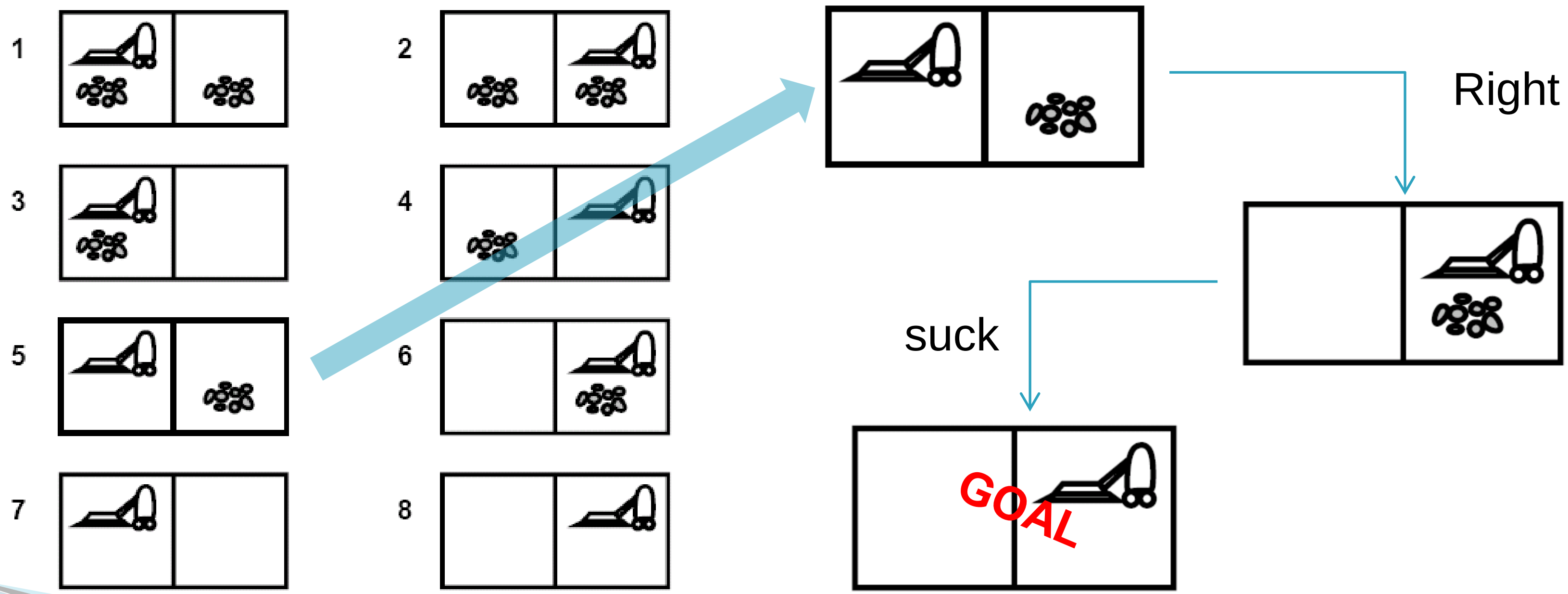
Sequences: Arad → Sibiu → Fagaras → Bucharest

?optimal

Back to Slide 9

4 Problem types (1)

- Single-state problems:
  - Deterministic, fully observable:
    - Consider a single state at a time
      - E.g. at State 5, [Right, suck] → goal state



8-possible states in a simple vacuum world

4 Problem types (2, 3)

- Multiple-state (conformant) problems:
  - Non-observable
    - The agent must reason about sets of states that it might get to. E.g. Without sensor, [Right] → {2, 4, 6, 8}
    - Sometimes there is no fixed action sequence that guarantees a solution to this problem.
- Contingency problems:
  - Nondeterministic and/or partially observable
    - The agent calculates a whole tree of actions rather than a single action sequence. In general, each branch of the tree deals with a possible contingency that might arise. Many real-world are contingency problems, because exact prediction is impossible

4 Problem types (4)

- Exploration problems:
  - Unknown state space
    - The agent must experiment, gradually discovering what its actions do and what sorts of states exist. ?”door”?

State space:

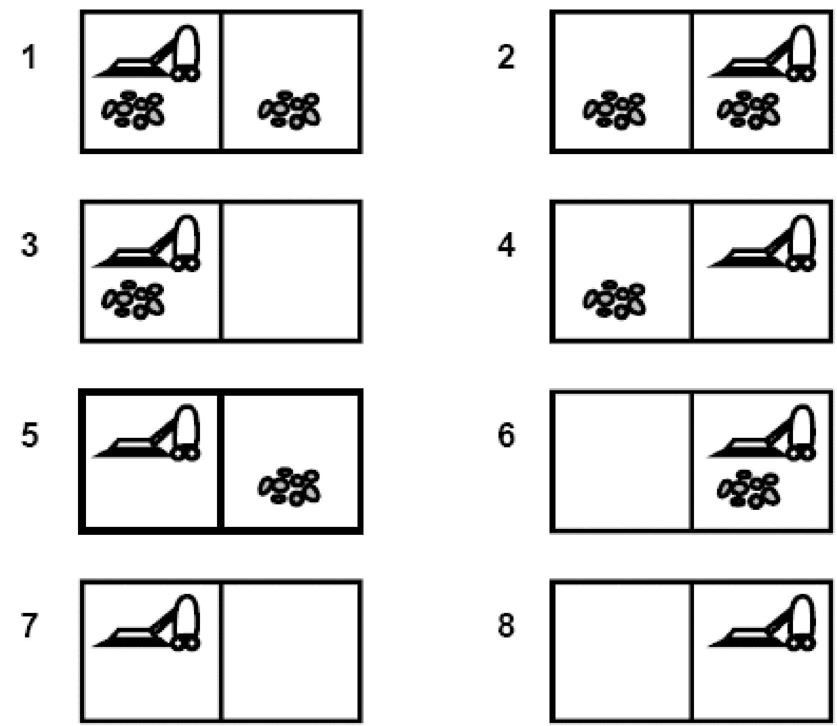
- The set of all states reachable from the initial state by any sequence of actions. A path in the state space is any sequence of actions leading from one state to another. (e.g. all cities in p.6)

? Is the path important

Example: vacuum world

What is the solution of the following state?

- Single-state, start in 5
  - [*Right, Suck*]
- Conformant, start in {1,2,3,4,5,6,7,8}
  e.g. *Right* goes to {2,4,6,8}
  - [*Right, Suck, Left, Suck*]
- Contingency, start in 5
  Murphy's Law: *Suck* can dirty a clean carpet
  Local Sensing: dirt, location only.
  - [*Right,* **if** *dirt* **then** *Suck*]



Single-state problem formulation

A (well-defined) problem is defined by four items:

Initial state – e.g. “at Arad” and other states

Successor function – S(x) = set of action–state pairs

e.g. S(Arad) = {[Arad→ Zerind, Zerind], …[action, state]…}

Initial state + Actions => state space

Goal test:

- Explicit, e.g. x = “at Bucharest”
- Implicit, e.g. NoDirt(x), need evaluation

Path cost (additive)

e.g. ∑c: 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 goal state.

Selecting a state space (modeling)

Real world is absurdly complex

$\Rightarrow$ 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 states "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!

Measuring problem-solving performance

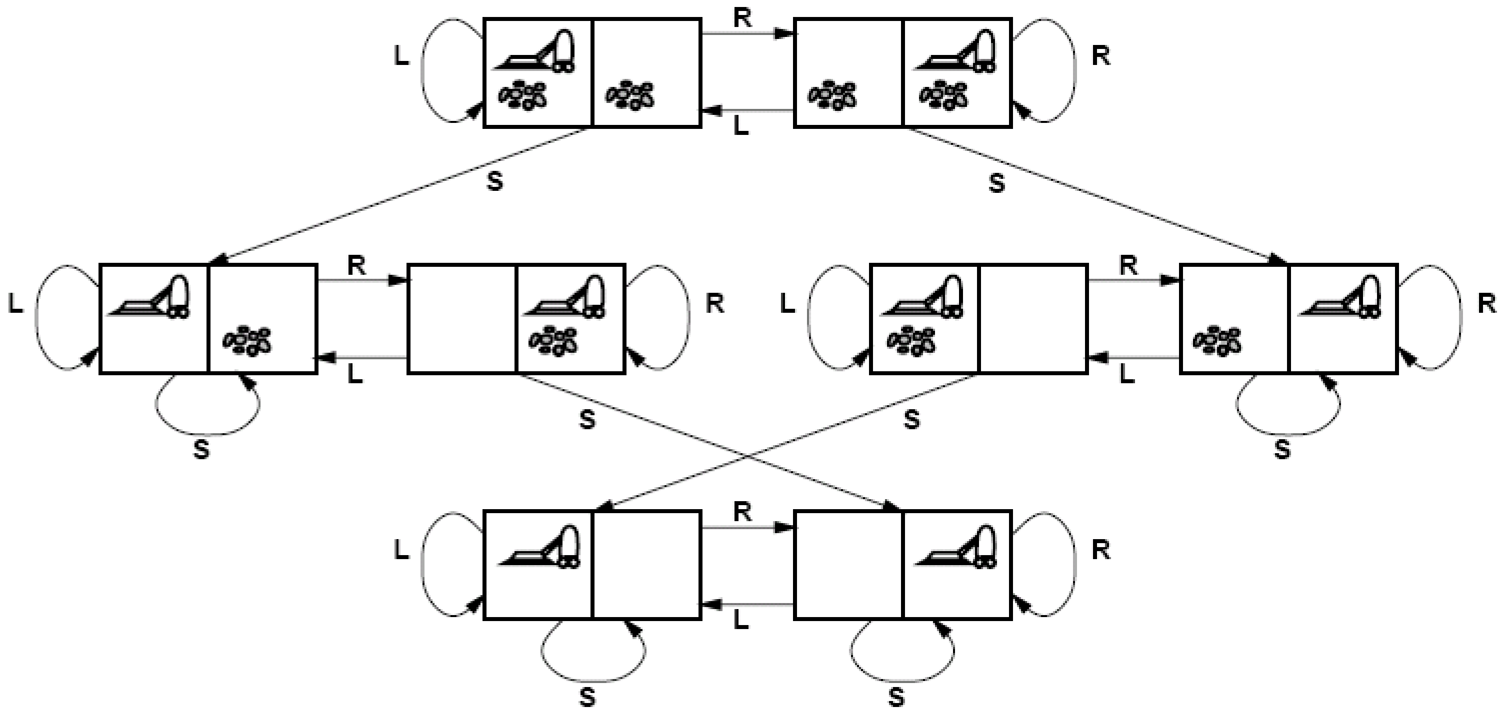
- (1) Does it find a solution at all? –complete
- (2) Is it a good solution (one with a low path cost)?–optimal, effective
- (3) What is the search cost associated with the time and memory required to find a solution? –efficient, complexity

The total cost of the search is the sum of the path cost and the search cost. (Trade off between them)

- The real art of problem solving is in deciding what goes into the description of the states and actions and what is left out.
- The process of removing detail from a representation is called **abstraction**.
- The choice of a good abstraction thus involves removing as much detail as possible while retaining (?) validity and ensuring that the abstract actions are easy to carry out.

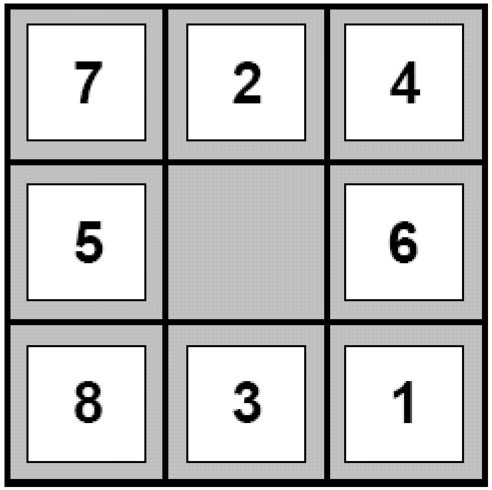
search space landscape??; transformation; feature selection; sensitivity analysis;

Example: vacuum world state space graph

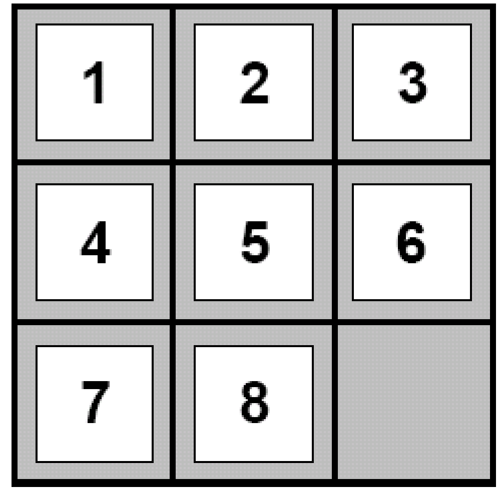


States:	Integer dirt and robot locations (ignore dirt amounts)
Actions:	Left, Right, Suck, NoOp (branch factor *b*)
Goal test:	No dirt
Path cost:	1 per action (0 for NoOp)

Example: The 8-puzzle



Start State

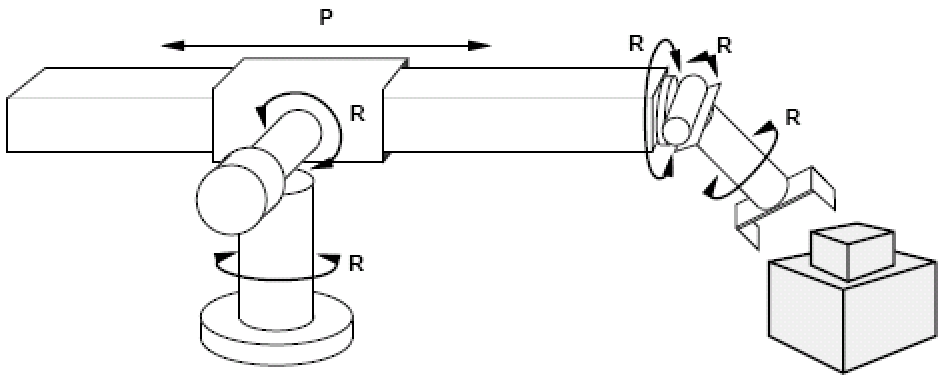


Goal State

States:	Integer location of tiles (ignore intermediate positions)
Actions:	Move blank left, right, up, down (ignore un-jamming etc.)
Goal test:	Goal state (given) B=?
Path cost:	1 per move

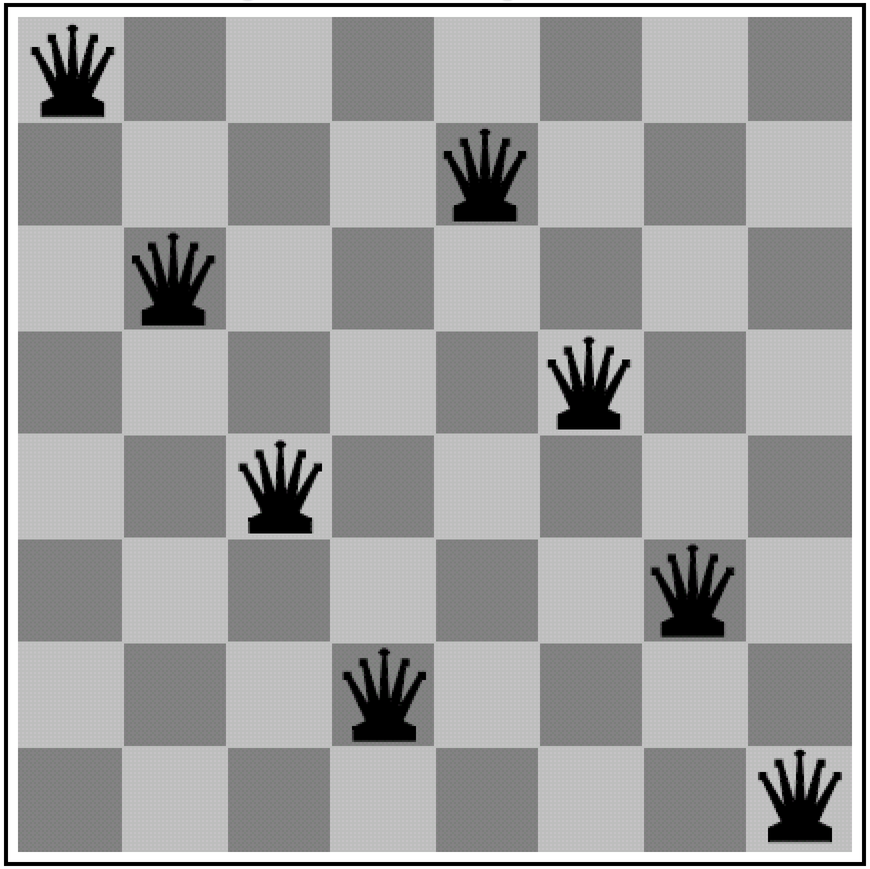
[Note: optimal solution of n-puzzle family is NP-hard, n=x*y-1]

Example: robotic assembly



States:	Real-valued coordinates of robot join angles & parts of the object to be assembled
Actions:	Continuous motions of robot joints
Goal test:	Complete assembly with no robot included!
Path cost:	Time to execute

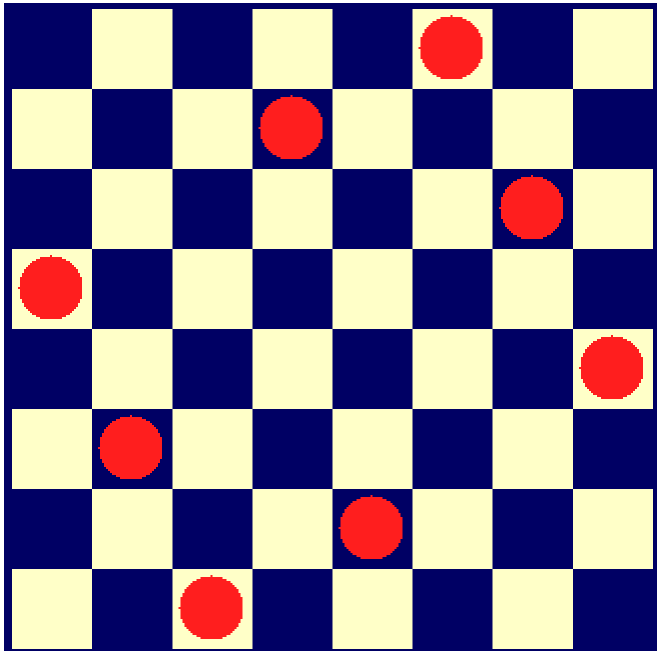
Example: The 8-queens problem



Rules: The 8 Queens cannot attack each other
This is one of the states of this problem

Back

- Solution



How would you find the solutions??

Example: 8-queens problem

2 main kinds of formulation (F): (1) The incremental formulation (F1, F2) places queens one by one. (2) The complete-state formulation (F3, F4) starts with all 8 queens on the board and moves them around.

- Goal test: 8 queens on board, none attacked.
- Path cost: zero.

They have different possible states and actions. Consider the following simple-minded formulation.

(F1) enumeration

- States: any arrangement of 0 to 8 queens on board
- Action: add a queen to any square.

In this formulation, we have 64^8 possible sequences to investigate. More sensible:

Example: 8-queens problem

(F2) DEMO

- States: arrangements of 0 to 8 queens with none attacked.
- Actions: place a queen in the left-most empty column such that it is not attacked by any other queen. (2057)

The right formulation makes a big difference to the size of the search space:

(F3)

- States: arrangements of 8 queens, one in each column.
- Actions: move any attacked queen to another square in the same column   Goto board

(F4)

- This formulation would allow the algorithm to find a solution eventually, but it would be better to move to an un-attacked square if possible

Example: Cryptarithmetic

```
  FORTY
+   TEN
+   TEN
-------
= SIXTY
```

29786	F=2, O=9, R=7, T=8, Y=6
+ 850	T=8, E=5, N=0
+ 850	T=8, E=5, N=0
= 31486	S=3, I=1, X=4, T=8, Y=6

States:	A cryptarithmetic puzzle with some letters replaced by digits
Actions:	Replace all occurrences of a letter with a digit not already appearing in the puzzle. (Unique substitution). ?other operators
Goal test:	Puzzle contains only digits, and represents a correct sum
Path cost:	Zero. All solution equally valid.

Example: Missionaries (M) and cannibals (c)

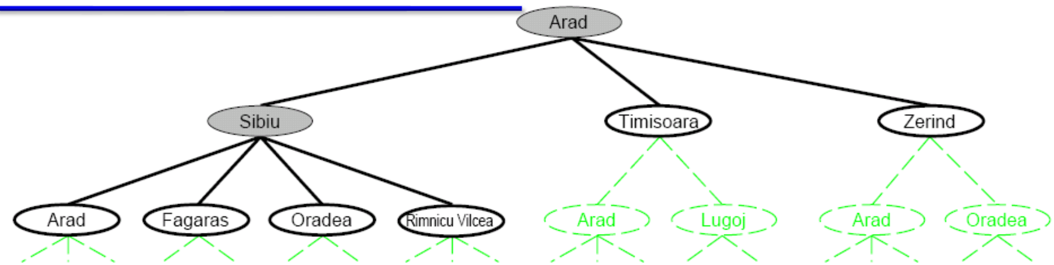
Demo->

- States: a state consists of an ordered sequence of 3 numbers representing the numbers of missionaries, cannibals, and boat on the bank of the river from which they started. Thus, the start state is (3,3,1).
- Constraints: (#M) ≥ (#c) in any place
- Actions: from each state the possible Actions are to take either 1 missionary, 1cannibal, 2 missionaries, 2 cannibals, or 1 of each across in the boat. Thus, there are at most 5 actions. Most states have fewer to avoid illegal state. If we were to distinguish between individual people then there would be 21 actions instead of just 5. (b=5)
- Goal test: reached state(0,0,0).
- Path cost: number of crossing

Example: Real-world problems

- Route finding
  - e.g. airline travel planning (complex path cost: money, quality of service, safety etc.)
- Touring and traveling salesperson problems (TSP).
  - NP-hard.
- VLSI layout: cell layout, channel routing
- Robot navigation
  - many dimension
- Assembly sequencing
- Protein design:
  - amino acid sequence folding into 3D protein with properties to cure disease; drug discovery
- Internet searching
- Big Data Analytics

TREE SEARCH ALGORITHMS



Basic idea:

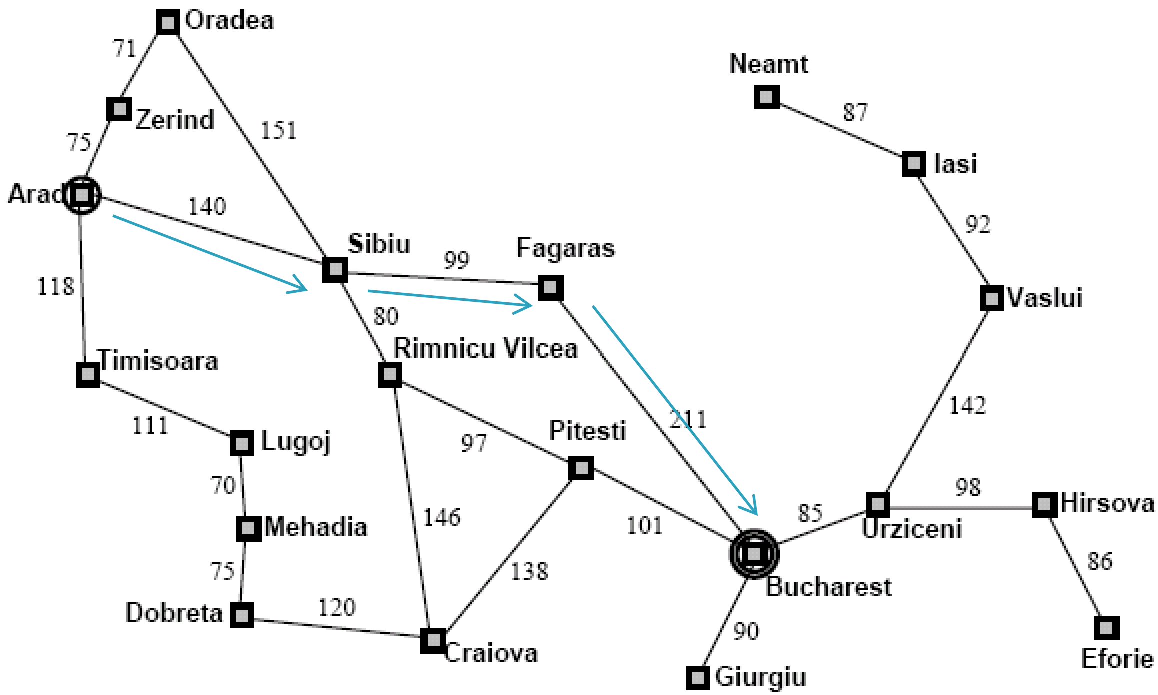
Current state → generate a new set of states (expanding the state) → choose one (expanding state) by a search strategy to expand

➔ search tree with search nodes

initial state = root

function Tree-Search(*problem*, *strategy*) **returns** a solution, or fail
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion **then return** fail
    choose a leaf node for expansion according to strategy
    if the node contains a goal state **then return** the corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end

Example: Romania



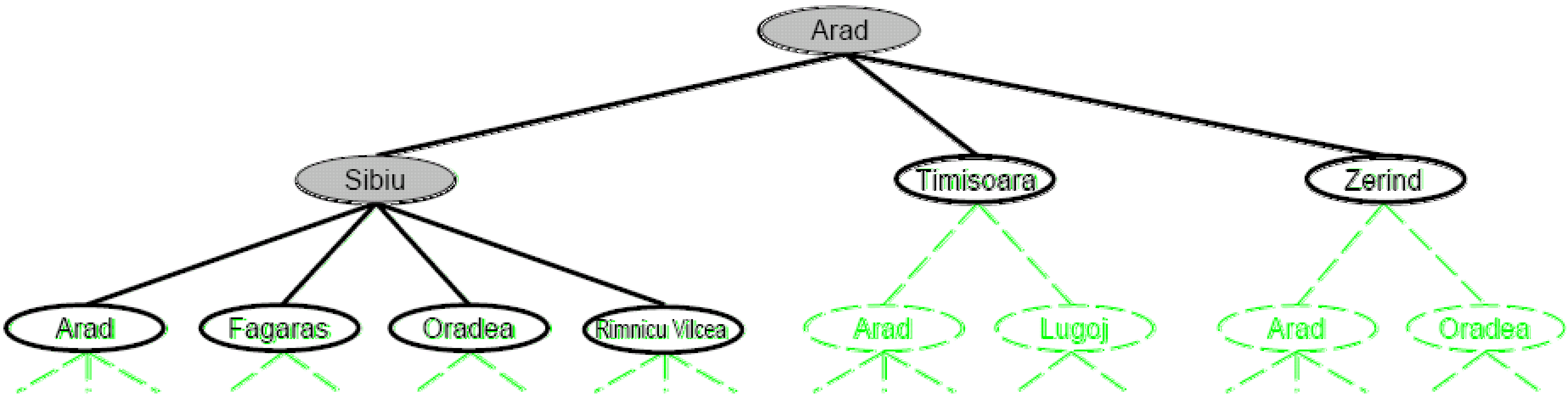
Sequences: Arad → Sibiu → Fagaras → Bucharest

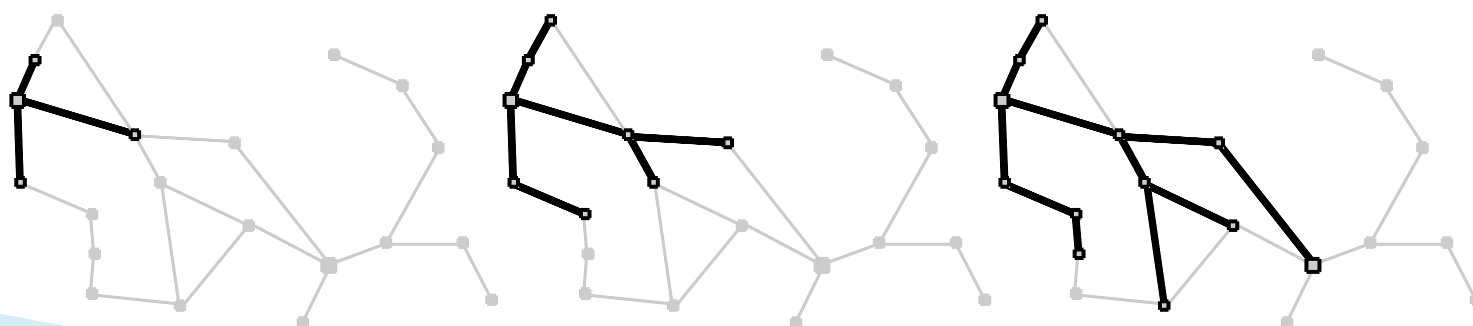
Slide 9

Oradea: o ‘ra ya; Neamt: ne ‘am; Timisoara: ti mi ‘shwa ra; Pitesti: pe ta ‘she te

Tree search example

fringe??



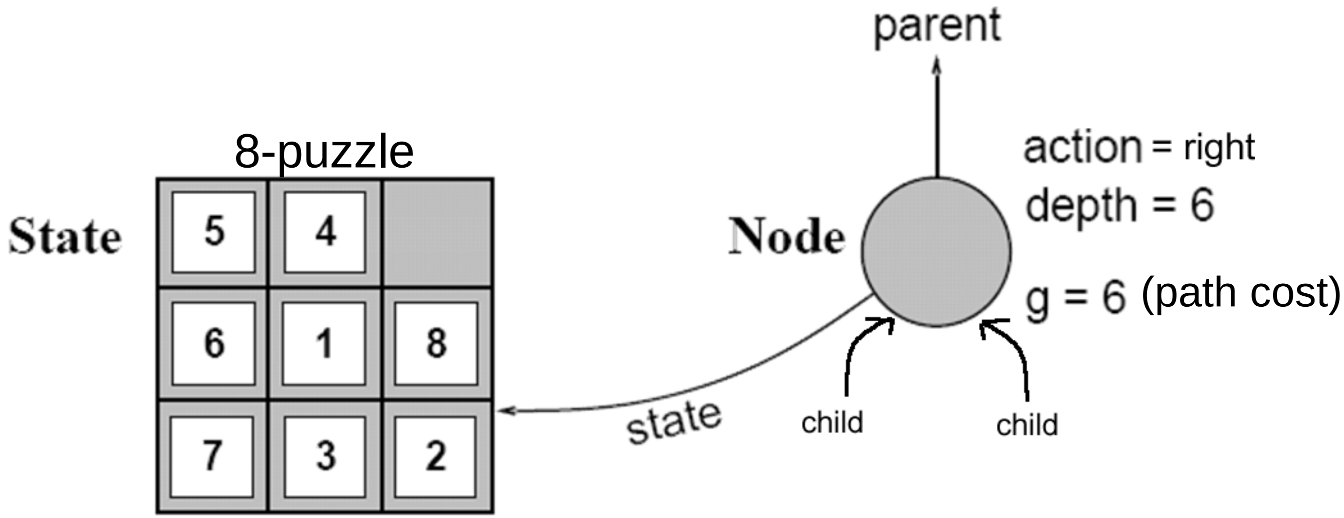


Implementation: states vs. nodes

A state is a (representation of) a physical configuration of real problem

A node is a data structure constituting part of a search tree includes parent, children, depth, path cost g(x)

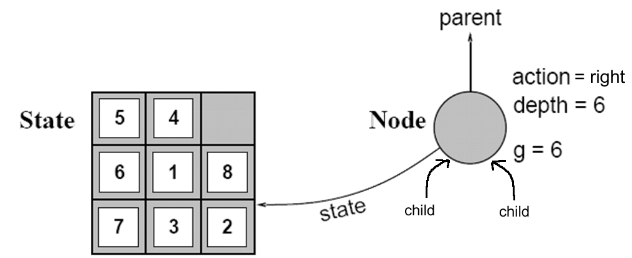
States do not have parents, children, depth, or path cost!



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

(action) ?

Implementation: general tree search




```
function Tree-Search(problem, fringe) returns a solution, or failure
  fringe ← Insert(Make-Node(Initial-State[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← Remove-Front(fringe)
    if Goal-Test[problem] applied to State(node) succeeds return node
    fringe ← InsertAll(Expand (node, problem), fringe) //according to strategy
  end
```

```
function Expand(node, problem) returns a set of node   //creates new nodes for each action
  successors ← the empty set
  for each action, result in Successor-Fn[problem](State[node]) do  //compute state
    s ← a new Node                                                  //e.g.L,R,U,D
    Parent-Node[s] ← node;  Action[s] ← action; State[s] ← result
    Path-Cost[s] ← Path-Cost[node] + Step-Cost[node, action, s]     //g
    Depth[s] ← Depth[node] + 1
    add s to successors                      //newly expanded nodes
  return successors
  end
```

Search strategies

A 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/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 the search tree

d – depth of the least-cost solution

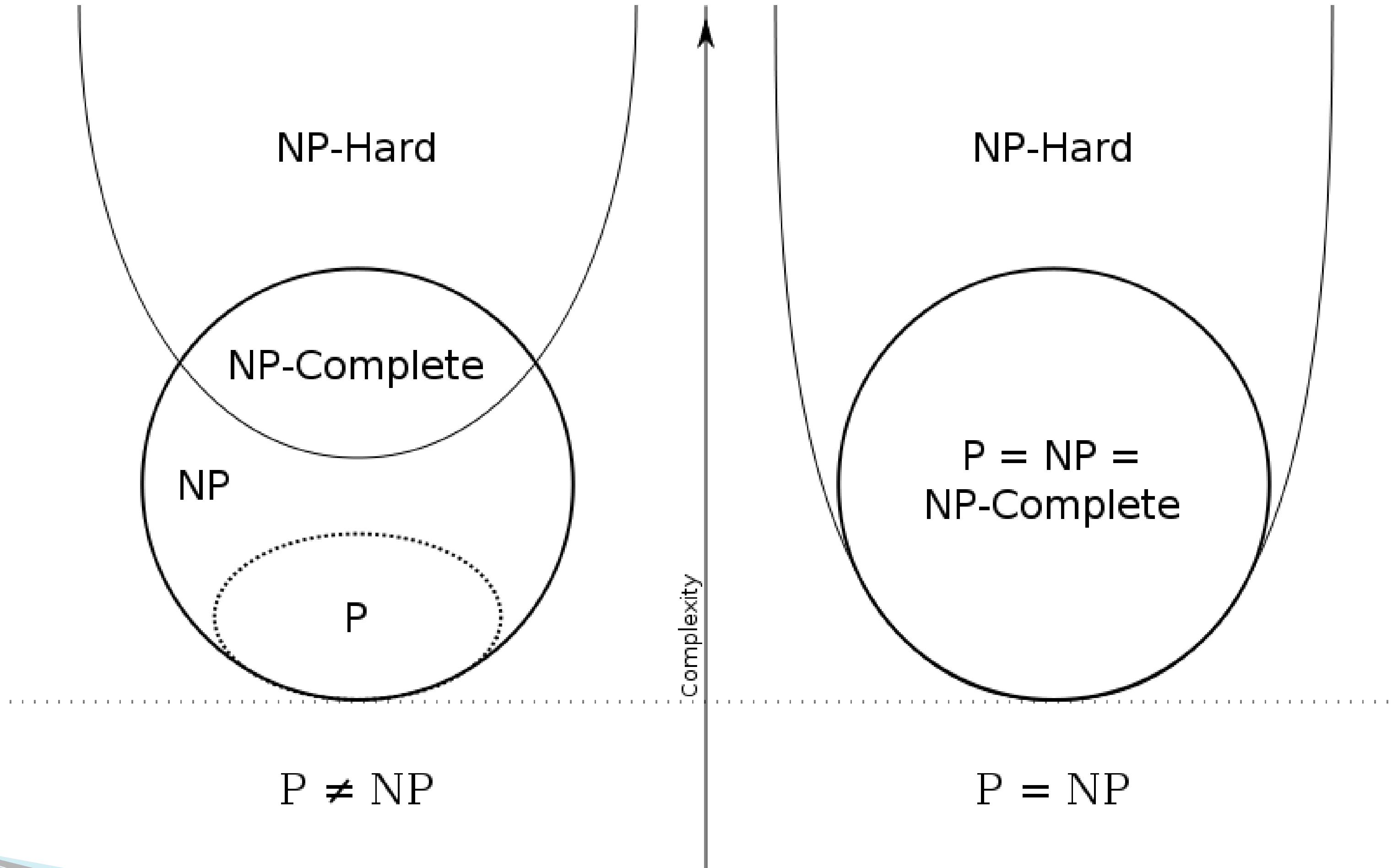
m – maximum depth of the state space (may be ∞)??

Search strategies

Complexity Analysis (of problems)

- P: the class of Polynomial problems – consider "easy"
  - E.g. O(log n), O(n) but $O(n^{1000})$.
- NP: the class of Nondeterministic Polynomial problems – inherently hard
  - E.g. $O(2^n)$; $\geq$ P
- NP-complete: most complex NP problems, e.g. Satisfiability, TSP, bin packing problems; $\geq$ NP
- NP-hard: more complex (problems harder) than NP problems
  - E.g. the optimization versions of the above NP-complete problems; $\geq$ NP-complete
- PSPACE-hard problems (problems requiring polynomial amount of space)
  - Worse than NP-complete problems ??

Complexity Analysis diagram from Wikipedia



Quantum
computer

Uninformed search strategies

Uninformed 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

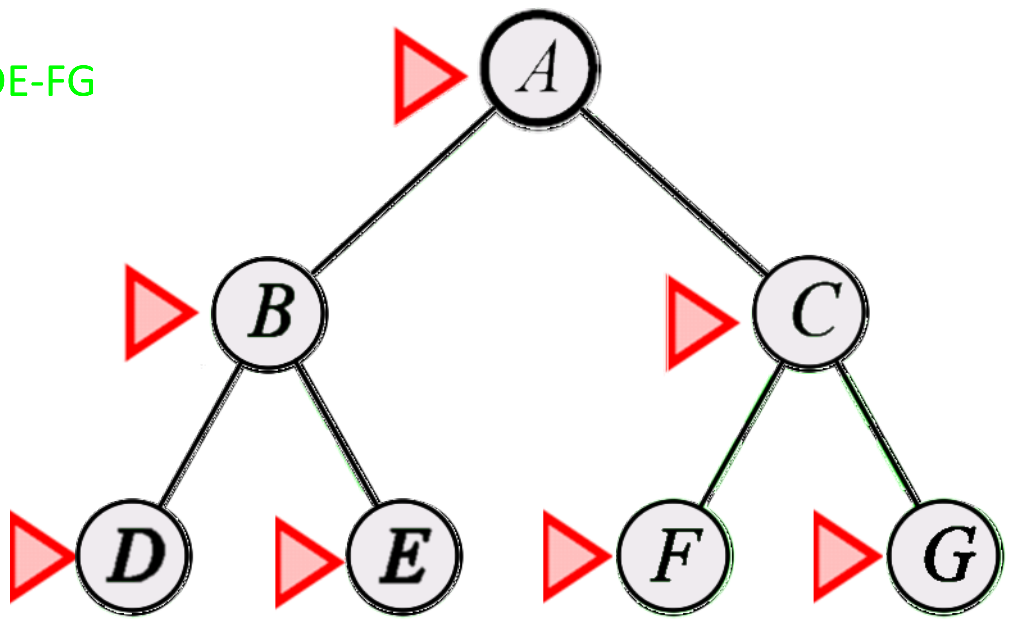
breadth-first search

- Expand shallowest unexpanded node

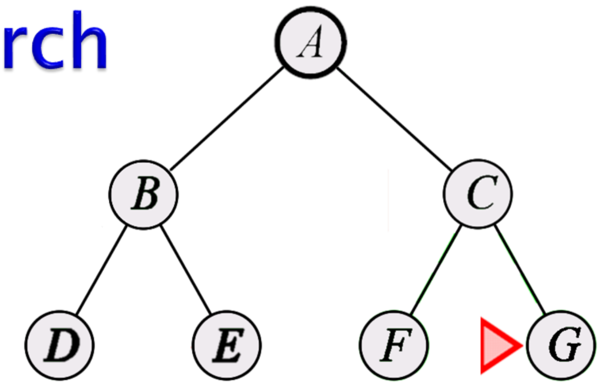
Implementation:

fringe is a FIFO queue, i.e. new successors go at end

Queue: A-BC-DE-FG



Properties of breadth-first search



Complete	Yes (if b is finite)
Time	$1+b+b^2+b^3+\ldots+b(b^d-1) = O(b^{d+1})$, i.e. exp. in d
Space	$O(b^{d+1})$ (keep every node in memory) (?)
Optimal	Yes(if cost = 1 (same) per step); not optimal in general

b: branching factor

d: depth of opt soln

Properties of breadth-first search

The memory (space) requirements are a bigger problem for breadth-first search than the execution time.

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breath-first search. The numbers shown assume branching factor b = 10; 1 million nodes/second; 1000 bytes/node

Our supercomputer: 6TB; exabytes: 10^{18}

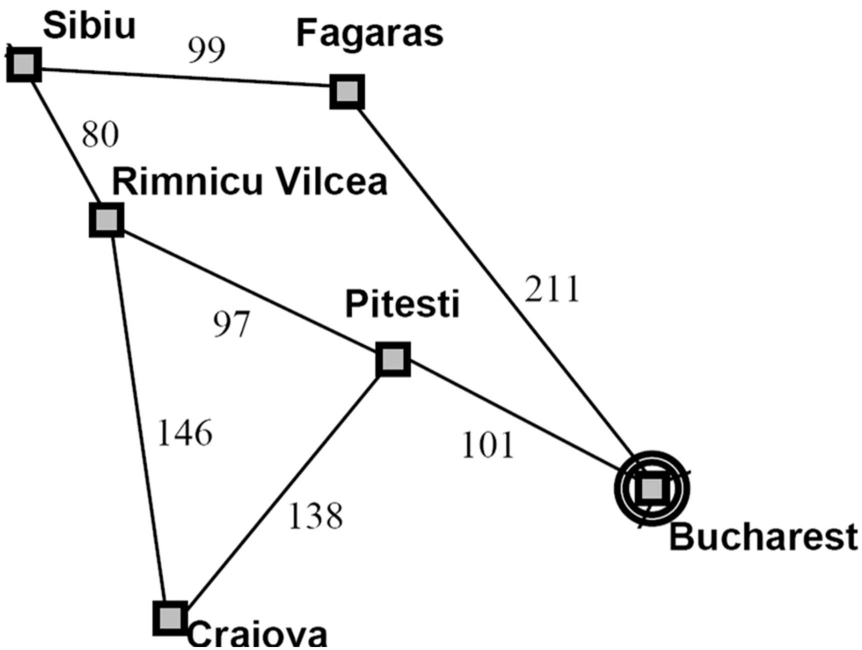
Uniform-cost search 1

- Breadth-first search finds the shallowest goal state, but this may not always be the least-cost solution for a general path cost function. (equal step cost only)
- Uniform cost search modifies the breadth-first strategy by always expanding the lowest-cost node on the fringe (as measured by the path cost g(n)), rather than the lowest-depth node. ⇒ best first
- Uniform cost search finds the cheapest solution provided: the cost of a path must never decrease as we go along the path (i.e. no negative step costs)

  $g(\text{SUCCESSOR}(n)) \geq g(n)$ (?)

  ⇒ optimal solution without exhaustive search possible ?

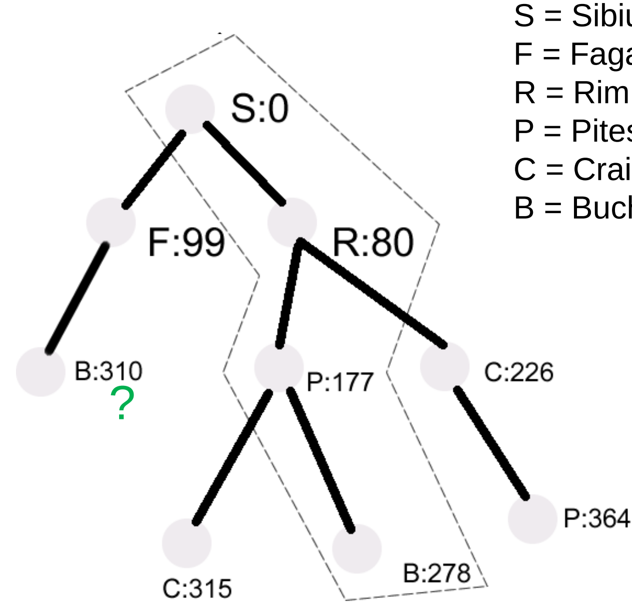
Uniform-cost search 2



Example of traveling from <u>Sibiu to Bucharest</u>

Always find a lowest path to expand until no other path can reach the destination with the lowest cost

S = Sibiu
F = Fagaras
R = Rimnicu Vilcea
P = Pitesti
C = Craiova
B = Bucharest



Therefore the shortest path will be:
Sibiu → Rimnicu Vilcea → Pitesti → Bucharest

Uniform-cost search 3

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 (epsilon, small positive real no)
Time	# of nodes with g ≤ cost of optimal solution, $O(b^{\lceil C^*/\varepsilon \rceil})$ where C* is the cost of the optimal solution
Space	# of nodes with g ≤ cost of optimal solution, $O(b^{\lceil C^*/\varepsilon \rceil})$
Optimal	Yes-nodes expanded in increasing order of g(n)

g: path cost
ε: smallest possible step cost
d – worst case depth of optimal solution = $\lceil C^*/\varepsilon \rceil$

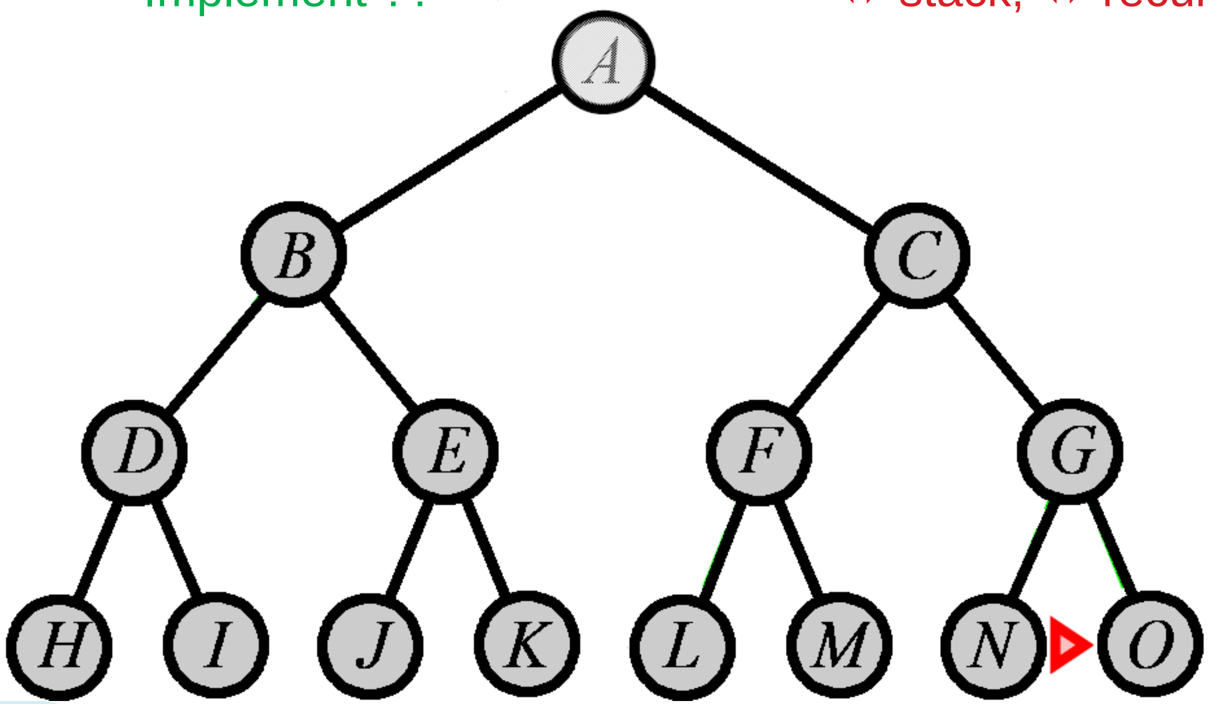
Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

Implement ??   ⇔ stack; ⇔ recursive calls



Properties of depth-first search

Complete	No, fails in infinite-depth spaces, spaces with loops Modify to avoid repeated states along path $\Rightarrow$ 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	$O(bm)$, i.e., linear space! (?)
Optimal	No

b: branching factor

m: max depth of search tree

d: depth of opt soln

Depth-limited search

= depth-first search with depth limit ℓ
i.e. nodes at depth ℓ have no successors

Recursive implementation: equivalent to a LIFO implemented by a stack

function DEPTH-LIMITED-SEARCH (*problem*, *limit*) **returns** solution/fail/cutoff
  return RECURSIVE-DLS (MAKE-NODE (Initial-State[*problem*]), *problem*, *limit*)

function RECURSIVE-DLS (*node*, *problem*, *limit*) **returns** solution/fail/cutoff
  cutoff-occurred? ← false
  if Goal-Test[*problem*](State[*node*]) **then return** node
  else if Depth[*node*] = *limit* **then return** *cutoff*
  else for each *successor* in Expand (*node*, *problem*) do
    result ← RECURSIVE-DLS (*successor*, *problem*, *limit*)
    if *result* = *cutoff* **then** *cutoff-occurred?* ← true
    else if *result* ≠ failure **then return** *result*
  if *cutoff-occurred?* **then return** *cutoff* **else return** failure

Depth-limited search

- Difficult for depth-limited search to pick a good limit.
- The diameter (max length from any node to any other nodes) of state space, gives us a better depth limit for a more efficient depth-limited search
- However, for most problems, not know until the problem solved.

- What if we do not know ℓ ??????

Iterative deepening search

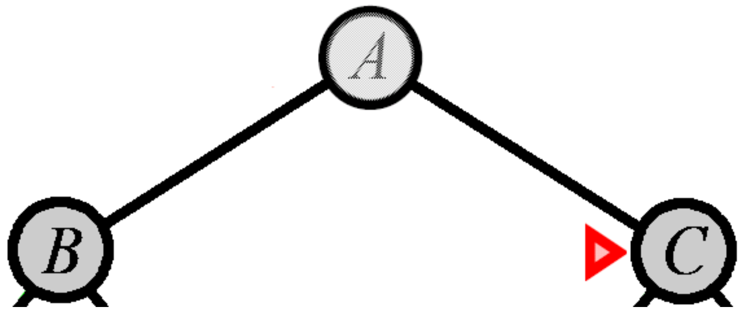
```
function Iterative-Deepening-Search (problem) returns a solution
   inputs: problem: a problem
   for depth ← 0 to ∞ do
      result ← Depth-Limited-Search (problem, depth)
      if result ≠ cutoff then return result
   end
```

?repeated visits of nodes?

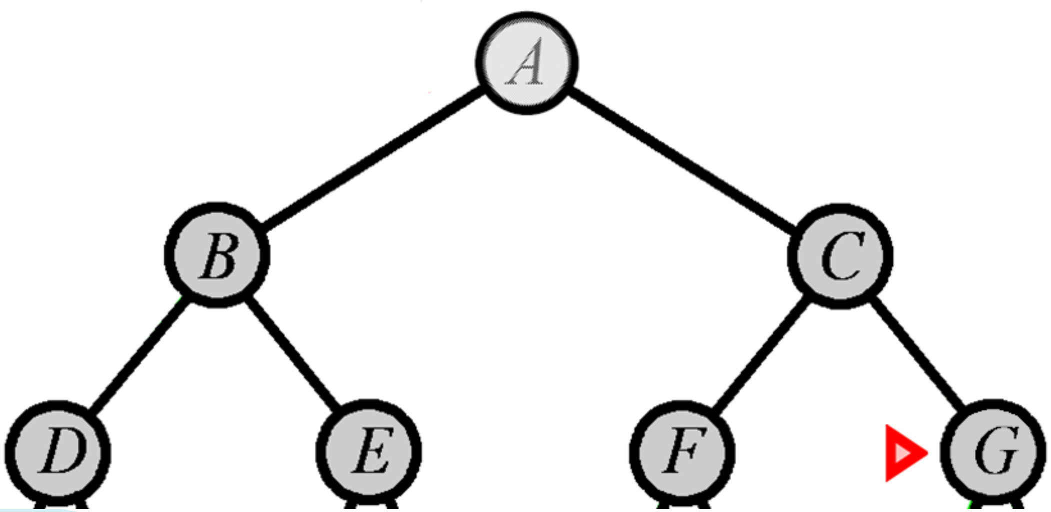
Iterative deepening search

($\ell = 0$)

($\ell = 1$)

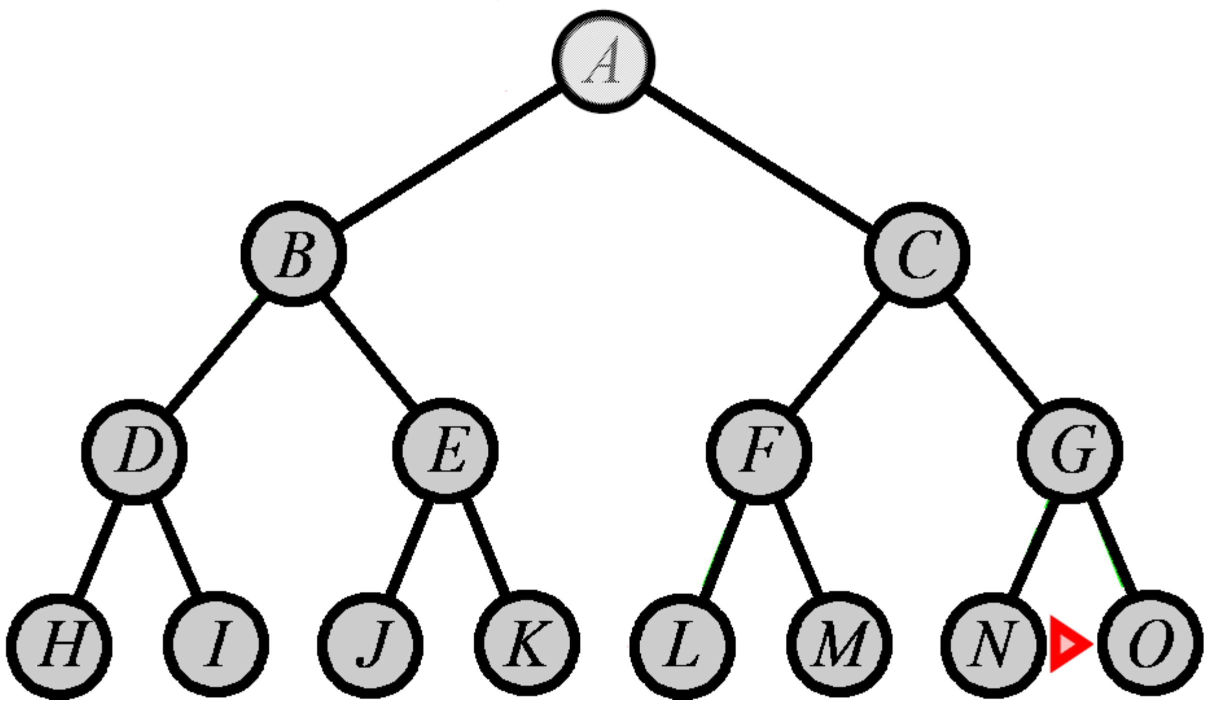


($\ell = 2$)



Iterative deepening search

(l = 3)



Properties of iterative deepening search

Complete	Yes
Time	$(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$ (?cf B First)
Space	$O(bd)$ (?cf D First)
Optimal	Yes, if step cost = 1. Can be modified to explore uniform-cost tree or C* ?

Numerical comparison for b = 10 and d = 5, solution at far right:

N(IDS) = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456

N(BFS) = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,101

- In general, preferred for a large search space (?) and unknown depth of the solution.
- Combines the benefits of depth-first and breadth-first search. Optimal and complete, like breadth-first search, but with modest memory requirements of depth-first search. Some states are expanded multiple times. ?

Summary of algorithms

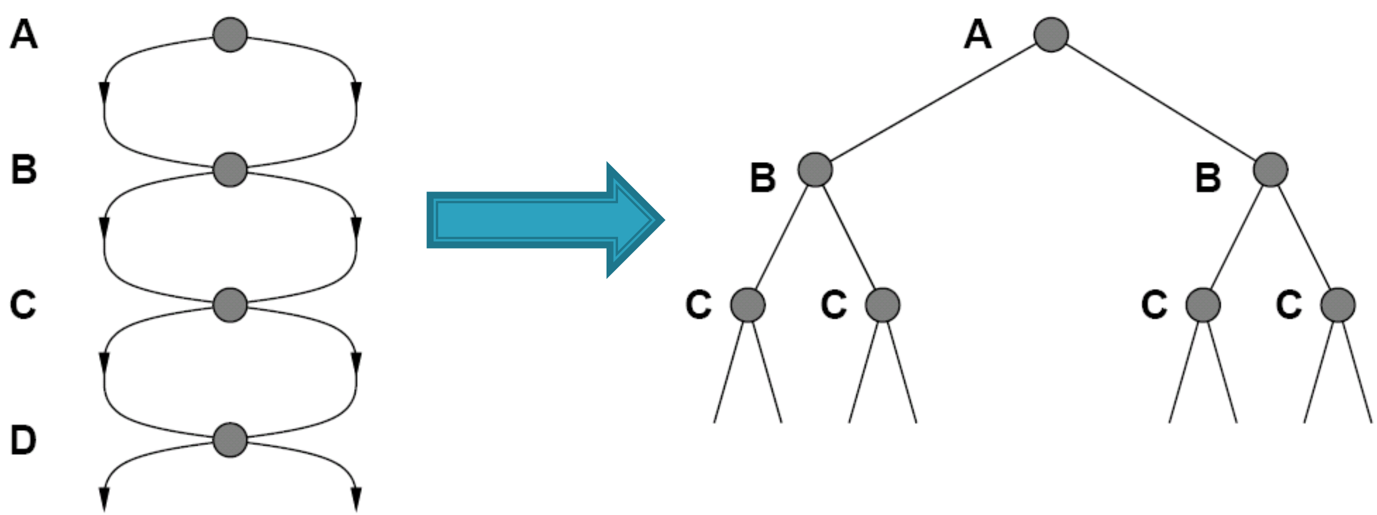
Criterion	Breadth-first	Uniform-cost	Depth-first	Depth-limited	Iterative Deepening
Complete?					
Time					
Space					
Optimal?					

b(branching factor), m(max depth), ℓ(depth limit), d(soln depth)
C* is the cost of the optimal solution

* : true for some conditions only:
1 b is finite
2 ε is the smallest +ve step size
3 BF and ID optimal if step costs are all equal

Repeated states

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



Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search engine
- Uniform cost is optimal
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms