Part II
Methods of AI

Chapter 2
Problem-solving

Chapter 2 – Problem-solving

2.1 Uninformed Search

2.2 Informed Search

2.3 Constraint Satisfaction Problems

SEARCH TECHNIQUES

- Note:

  Sometimes we have a course dedicated to search methods (4 hours per week).

  In that case we skip this part of the AI intro-lecture and ask you to attend this course.


2.1 Uninformed Search

AI Search Problems & Algorithms

→ AIMA: Chapter 3

Example: route-finding problem

On holiday in Romania; currently in Arad.

Flight leaves tomorrow from Bucharest

Formulate goal:

be in Bucharest

Formulate problem:

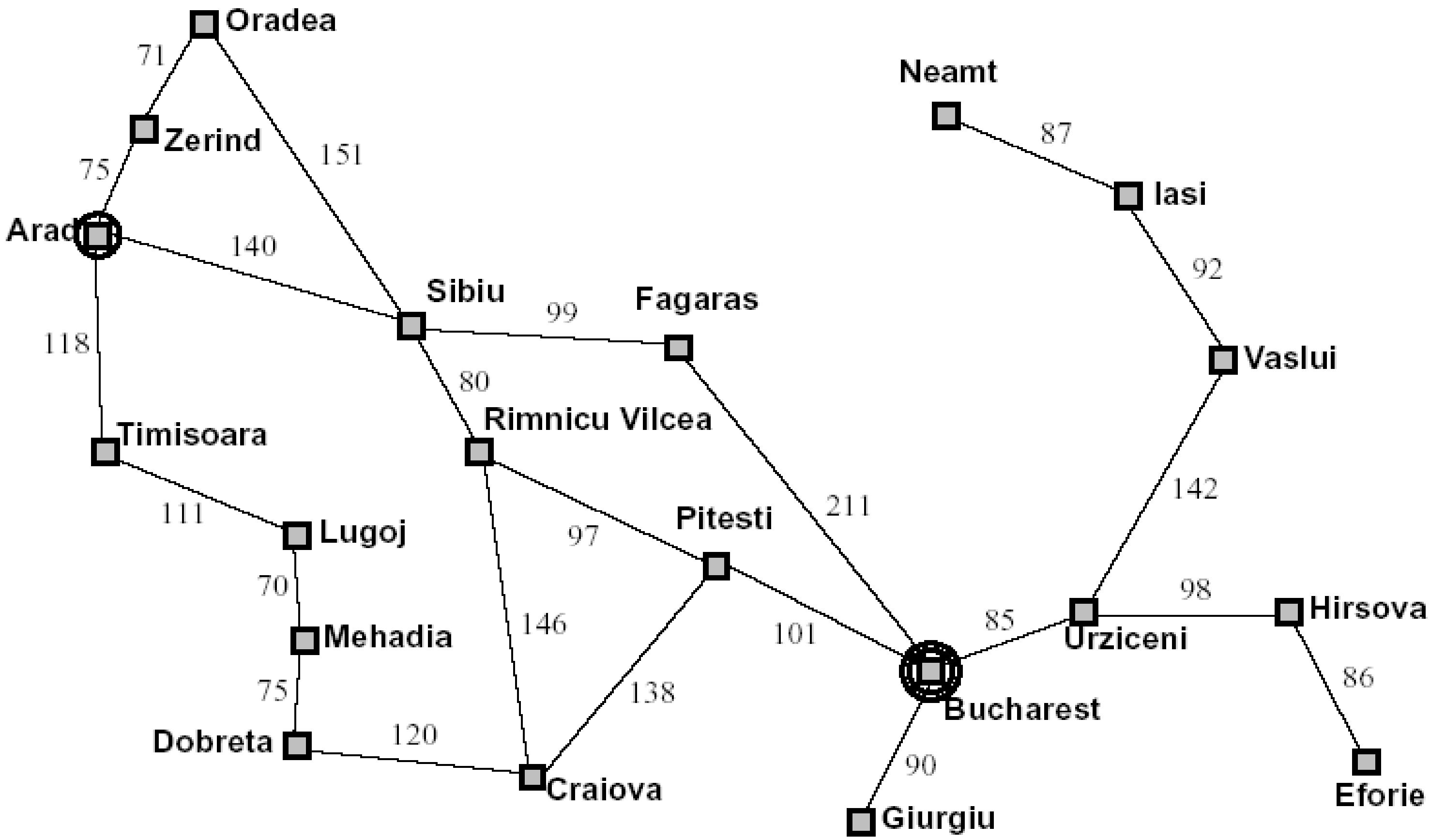
states: various cities

actions: drive between cities

Find solution:

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

Example: route-finding problem



Single-state problem formulation

A *problem* is defined by four items:

initial state e.g., “at Arad”

successor function *S(x)* = set of action-state pairs

e.g., *S(Arad)* = {<*Arad* → *Zerind, Zerind*>,…}

goal test, can be

explicit, e.g., x = “at Bucharest”

implicit, e.g., *No Dirt(x)*

path cost (additive)

e.g., sum of distances, number of a 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

Selecting a state space

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 $\rightarrow$ 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!

Example: the 8-puzzle

7	2	4
5		6
8	3	1

Start State

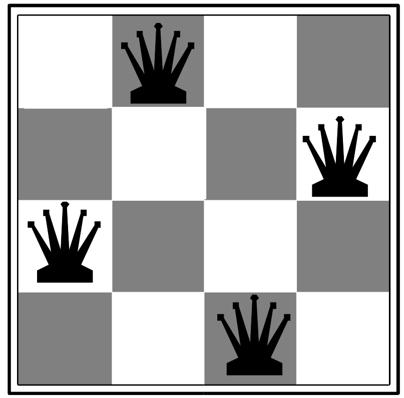
1	2	3
4	5	6
7	8	

Goal State

states?? integer locations of tiles (ignore intermediate positions)
actions?? move blank left, right, up, down (ignore unjamming etc.)
goal test?? = goal state (given)
path cost?? 1 per move

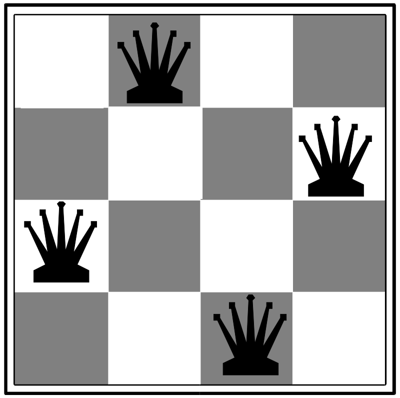
[Note: optimal solution of n-Puzzle family is NP-hard]

Example: n-queens problem (1)



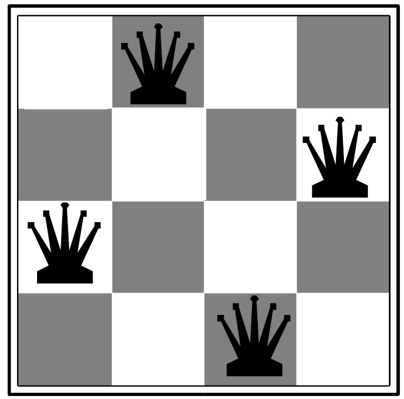
states??	arrangement of n queens on the board
actions??	move a queen left, right, up, down
goal test??	no queen attacks any other
path cost??	0, path cost is of no interest

Example: n-queens problem (2)



states??	arrangement of 0 to n queens on the board
actions??	add a queen to any empty square
goal test??	n queens on the board; no queen attacks any other
path cost??	0, path cost is of no interest

Example: n-queens problem (3)



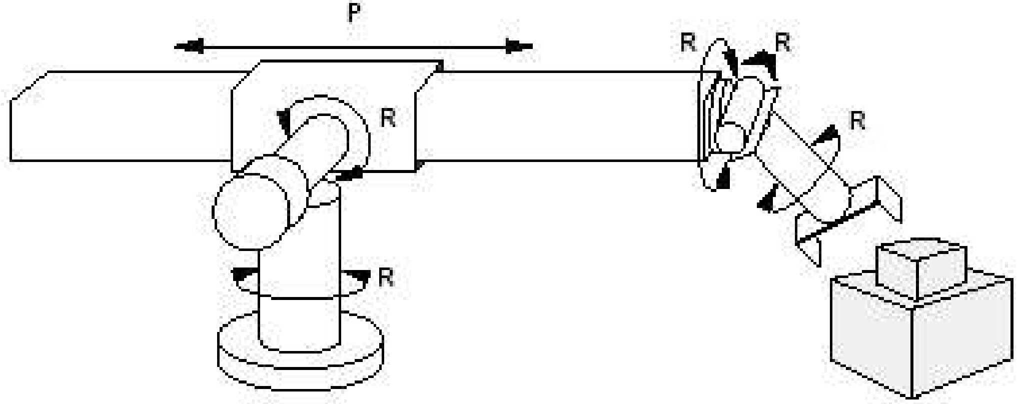
states?? arrangement of 0 to n queens on the board

actions?? add a queen to any empty square in the leftmost empty column, such that no queen attacks any other on the board

goal test?? n queens on the board; no queen attacks any other

path cost?? 0, path cost is of no interest

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 *with no robot included!*

path cost??: time to execute

Classification of search problems

Do we have a complete description of the search space?

No: ***Online Search Problem***

Yes: No additional information:

Uninformed Search Problem

Rough information on the topology of the search space:

Heuristic Search Problem

Structured information on the property of being a goal:

Constraint Satisfaction Problem

Problem-solving agents


Restricted form of general 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)
   if seq is empty, then
           goal ← FORMULATE-GOAL (state)
           problem ← FORMULATE-PROBLEM (state, goal)
           seq ← SEARCH (problem)
   action ← RECOMMENDATION (seq, state)
   seq ← Remainder (seq, state)
   return action
```

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

Classification of search algorithms

Path stored?

No: **Local *Search***

Yes: Memory space becomes a resource problem!

Are cycles detected?

No: ***Tree Search***

Yes: ***Graph Search***

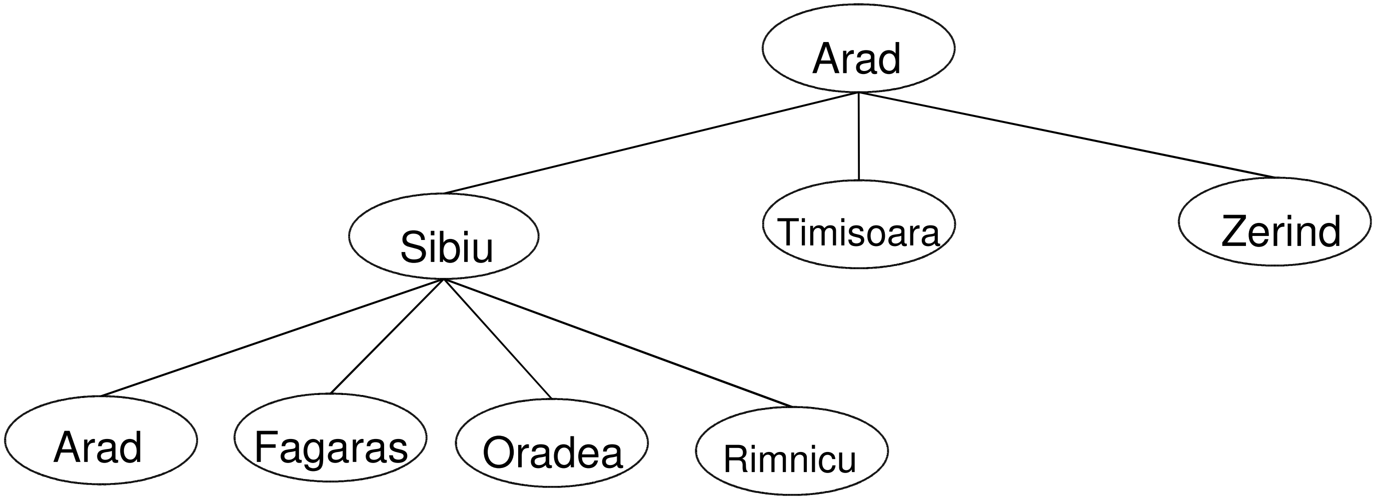
Tree search algorithms

Basic idea:

offline, simulated exploration of state space
by generating successors of already-explored states (*expanding* states)

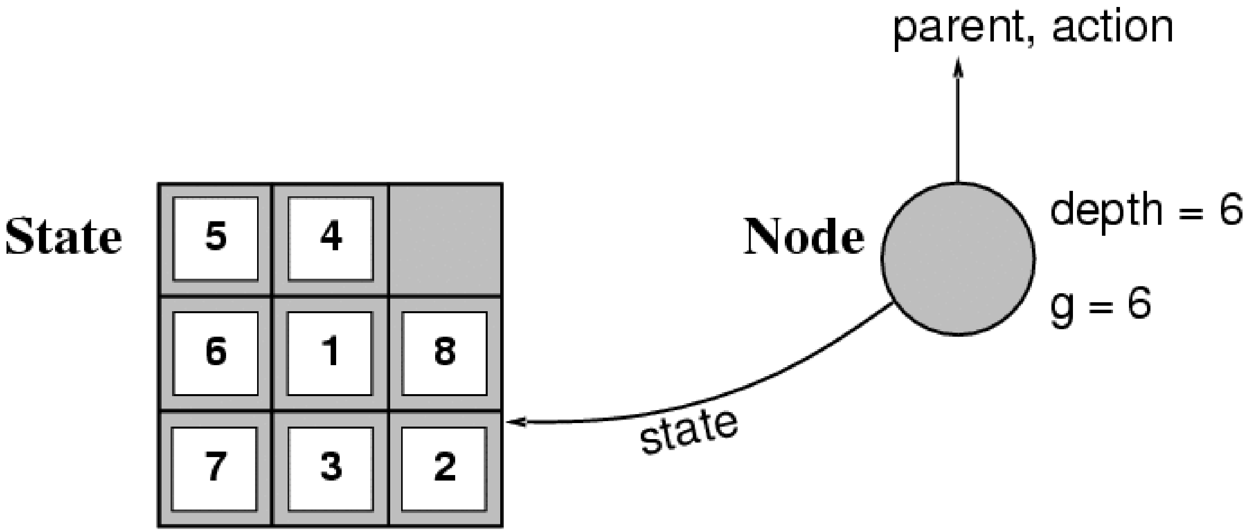
```
function TREE-SEARCH (problem, strategy)
        returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then
             return failure
        choose a leaf node for expansion according to strategy
        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
```

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 *parents*, *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 SUCCESSOR FN of the problem to create the corresponding states.


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 then
            return node
        fringe ← INSERTALL (EXPAND (node, problem), fringe)
```

```
function EXPAND (node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
      s ← a new NODE
      PARENT-NODE [s] ← node
      ACTION [s] ← action
      STATE [s] ← result
      PATH-COST [s] ← PATH-COST[node] + STEP-COST (node, action, s)
      DEPTH [s]  ← DEPTH [node] + 1
      add s to successors
    return successors
```

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 ∞)

Uninformed search strategies

Uninformed search strategies use only the information available in the problem definition

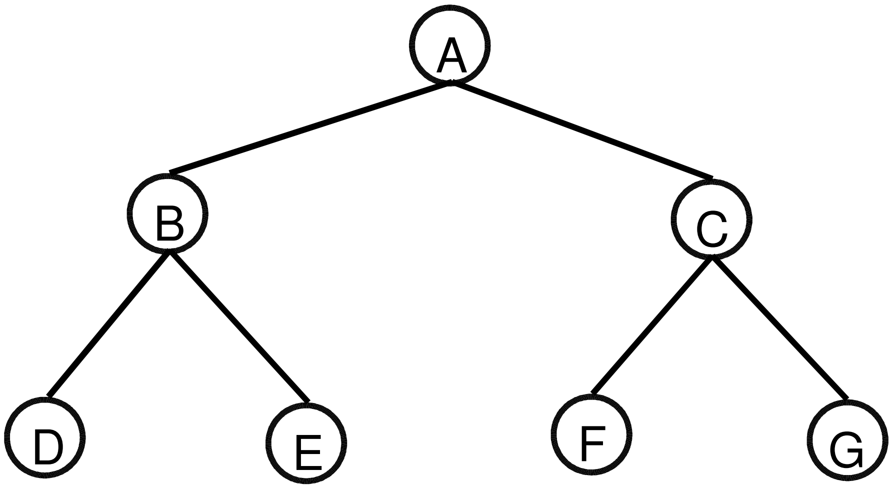
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

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



Properties of breadth-first search

Complete?? Yes (if *b* is finite)

Time?? $1+b+b^2+b^3+\ldots+b^d+b(b^d-1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? Yes (if cost = 1 per step); not optimal in general

Space is a big problem; can easily generate nodes at 10MB/sec
so 24hrs = 860 GB

Space → Paging → Time

Conclusion: If steps have different costs, use uniform cost search instead!

Uniform-cost search

Expand least-cost unexpanded node

Implementation:

fringe = queue ordered by path cost

Equivalent to breadth-first if step costs are all equal

Complete?? Yes, if step cost $\geq \varepsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{[C^*/\varepsilon]+1})$

where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{[C^*/\varepsilon]+1})$

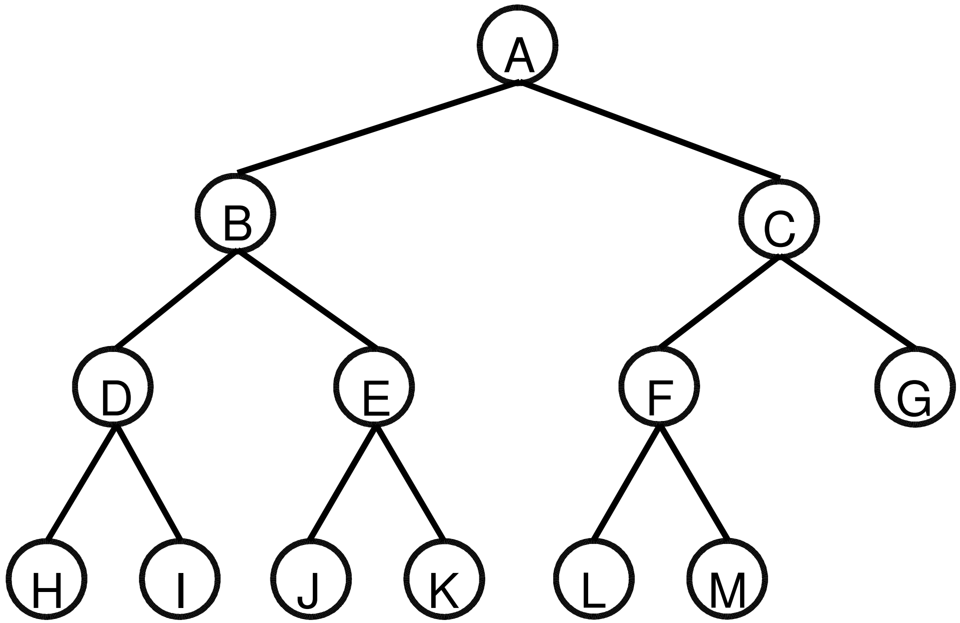
Optimal?? Yes – nodes expanded in increasing order of $g(n)$

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
$\Rightarrow$ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, can be much faster than breadth-first

Space?? $O(bm)$, i.e., linear space!

Optimal?? No

Conclusion: Not useful. Take Iterative Deepening Search instead.

Depth-limited Search

= depth-first search with depth limit

Recursive implementation:

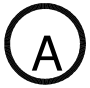
```
function DEPTH-LIMITED-SEARCH(problem; limit)
          returns soln/fail/cutoff
   RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)


function RECURSIVE-DLS(MAKE-NODE(node, problem; limit)
          returns soln/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
```

Iterative deepening search

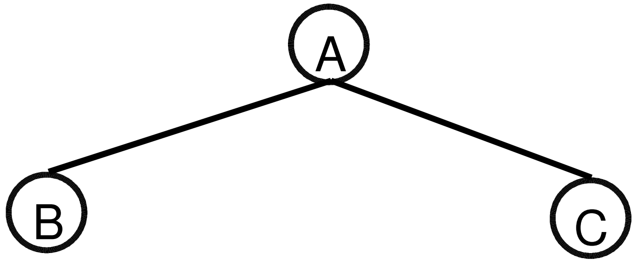
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEAPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
    end
```

Iterative deepening search (L= 0,1,2)

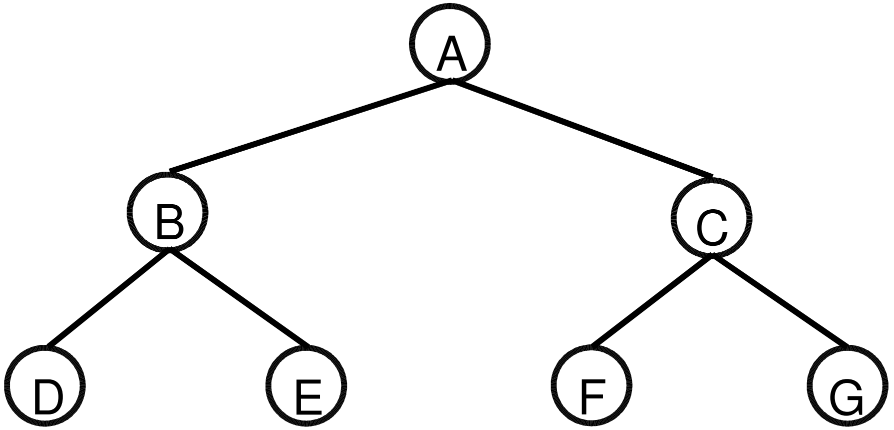
Limit = 0



Limit = 1

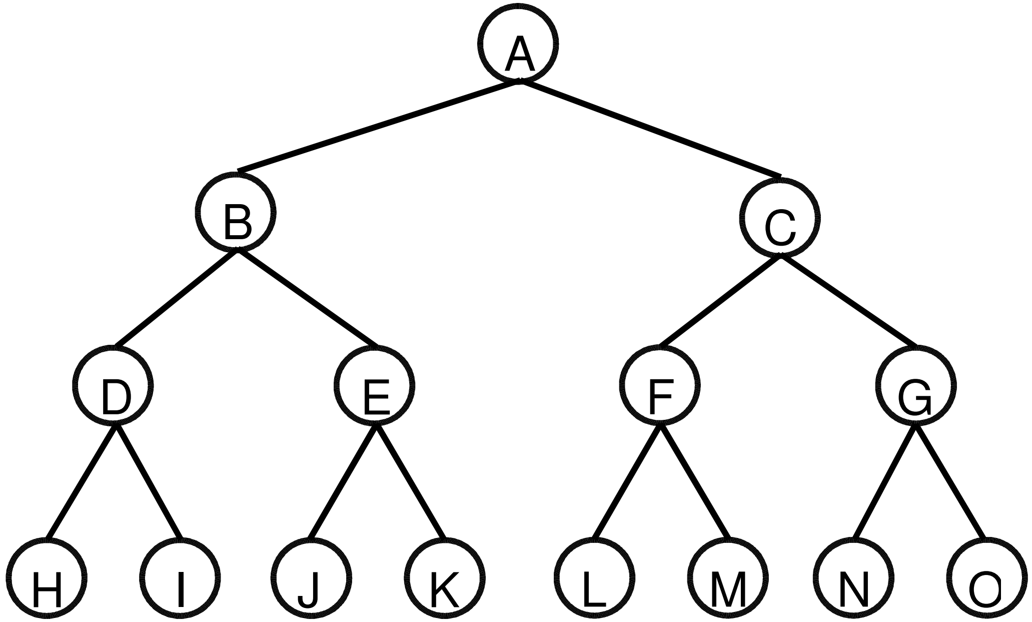


Limit = 2



Iterative deepening search (L= 3)

Limit = 3



Properties of iterative deepening search

Complete?? Yes

Time?? $(d+1)b^0+db^1+(d-1)b^2+\ldots+b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost = 1

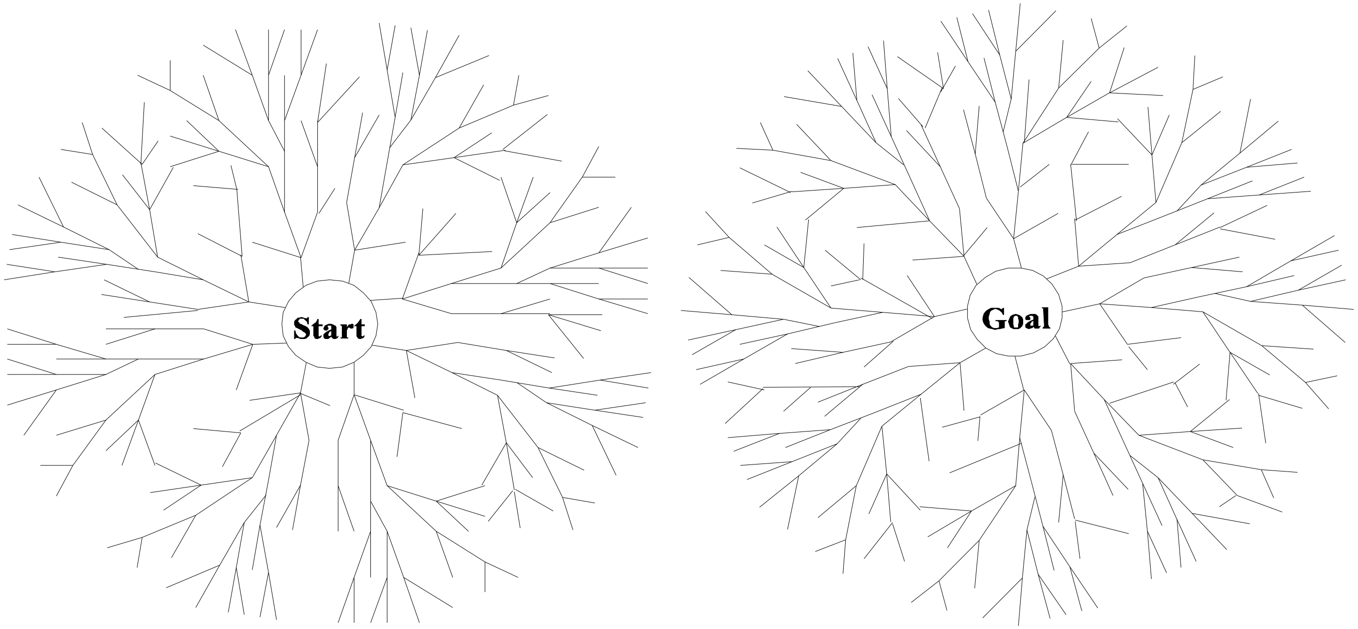
Note: can be modified analogously to uniform-cost tree

Numerical comparison for b = 10 and d = 5, solution at far right:
N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450
N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100


Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	$O(b^{d+1})$	$O(b^{[C^*/\varepsilon]+1})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{[C^*/\varepsilon]+1})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes*	Yes	No	No	Yes*

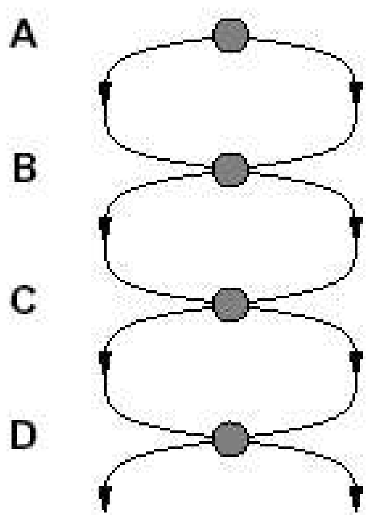
Bi-directional search

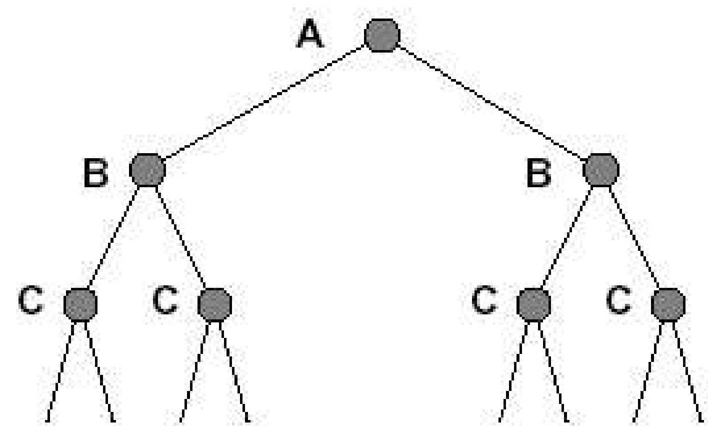


Schematic view of a bidirectional search is about to succeed, when a branch from the start node meets a branch from the goal node.
The motivation is that $b^{d/2} + b^{d/2}$ is much less than b^d, or in the figure, the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.

Repeated states

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





Graph search

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
    closed ← an empty set
    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](STATE[node]) then return node
        if STATE [node] is not in closed then return
            add STATE[node] to closed
            fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
    end
```

Summary (I)

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

- Variety of uninformed search strategies


Summary (II)

- A Problem consists of four parts:
  1. Initial state
  2. Set of actions
  3. Goal test function
  4. Path cost function

- Search algorithms are judged on the basis of **completeness, optimality, time complexity** and **space complexity**. Complexity depends on *b*, the branching factor in the state space, and *d*, the depth of the shallowest solution

- Uniform cost search is similar to breadth-first search but expands the node with lowest path cost, *g(n)*. It is complete and optimal if the cost of each step exceeds some positive bound ε

Summary (III)


- ***Iterative deepening search*** calls depth-limited search with increasing limits until a goal is found. It is complete, optimal for unit step costs, and has time complexity of *$O(b^d)$* and space complexity *$O(bd)$*. I.e. it uses only linear space and not much more time than other uninformed algorithms.

- ***Bidirectional search*** can enormously reduce time complexity, but is not always applicable and may require too much space.

  Assumptions we made about the search space:
  1. observable
  2. static and
  3. completely known

Summary (IV)

- When the environment is partially observable, the agent can apply search algorithms in the space of **belief states**, or sets of possible states that the agent might be in. In some cases, a single solution sequence can be constructed; in other cases, the agent needs a **contingency plan** to handle unknown circumstances that may arise.

Chapter 2 - Problemsolving

2.1 Uninformed Search

2.2 Informed Search

2.3 Constraint Satisfaction Problems