Set 2: State-spaces and Uninformed Search

ICS 271 Fall 2015

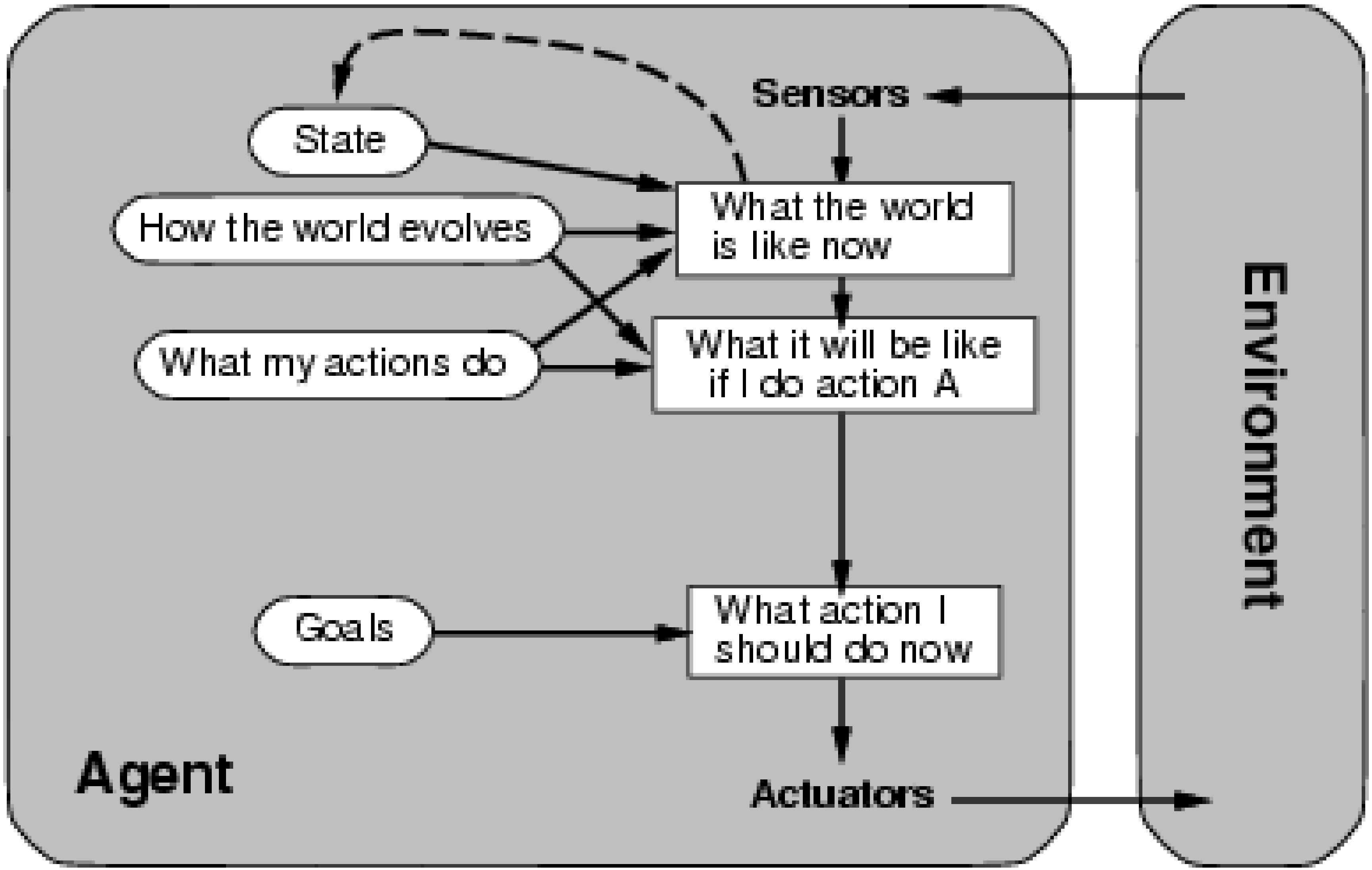
Kalev Kask

You need to know

- State-space based problem formulation
  - State space (graph)
- Search space
  - Nodes vs. states
  - Tree search vs graph search
- Search strategies
- Analysis of search algorithms
  - Completeness, optimality, complexity
  - b, d, m

Goal-based agents

Goals provide reason to prefer one action over the other.
We need to predict the future: we need to plan & search



Problem-Solving Agents

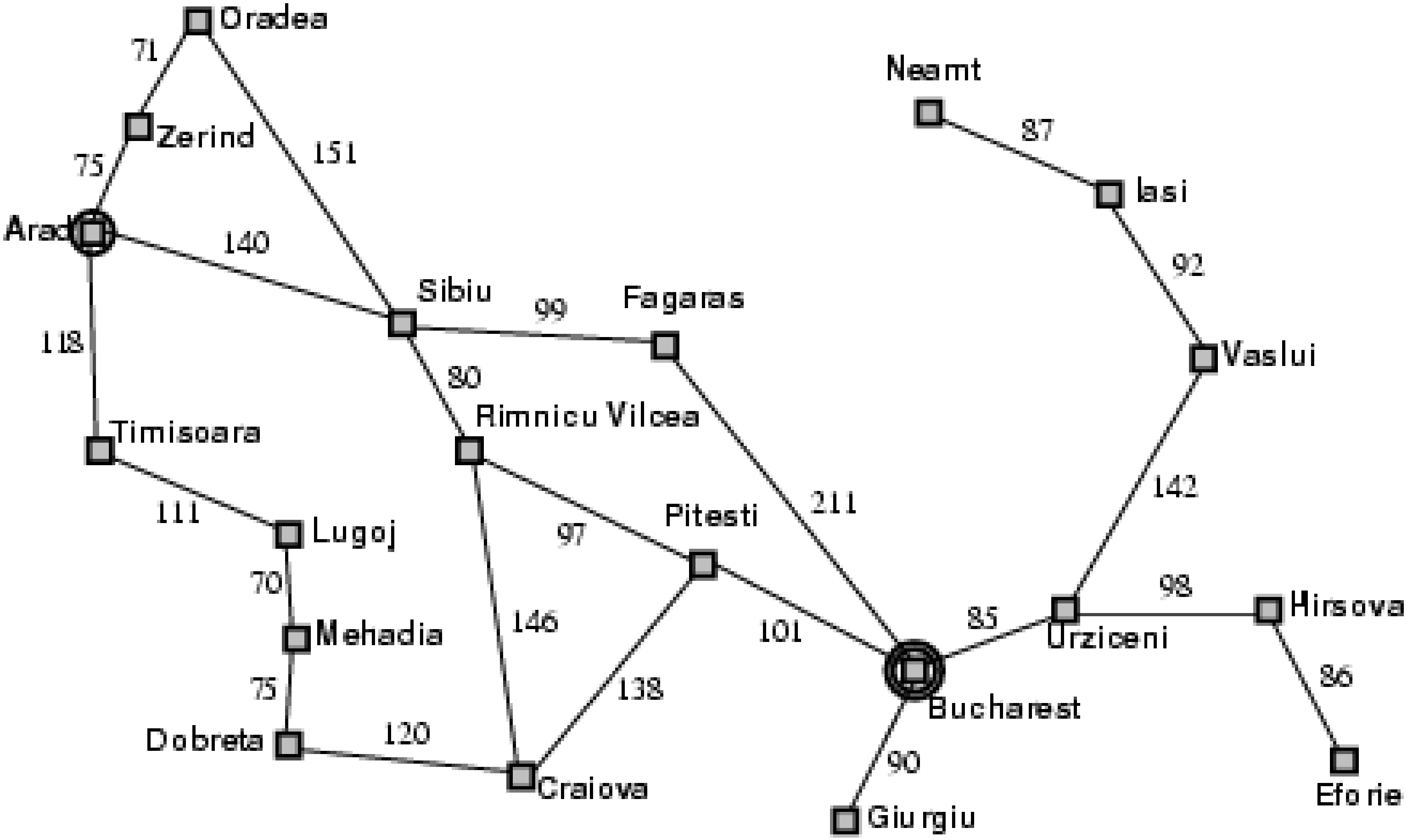
- Intelligent agents can solve problems by searching a state-space

- State-space Model
  - the agent's model of the world
  - usually a set of discrete states
  - e.g., in driving, the states in the model could be towns/cities

- Goal State(s)
  - a goal is defined as a desirable state for an agent
  - there may be many states which satisfy the goal
    - e.g., drive to a town with a ski-resort
  - or just one state which satisfies the goal
    - e.g., drive to Mammoth

- Operators(actions)
  - operators are legal actions which the agent can take to move from one state to another

Example: Romania



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 actions (cities), e.g., Arad, Sibiu, Fagaras, Bucharest

Problem Types

- Static / Dynamic

Previous problem was static: no attention to changes in environment

- Observable / Partially Observable / Unobservable

Previous problem was observable: it knew its initial state.

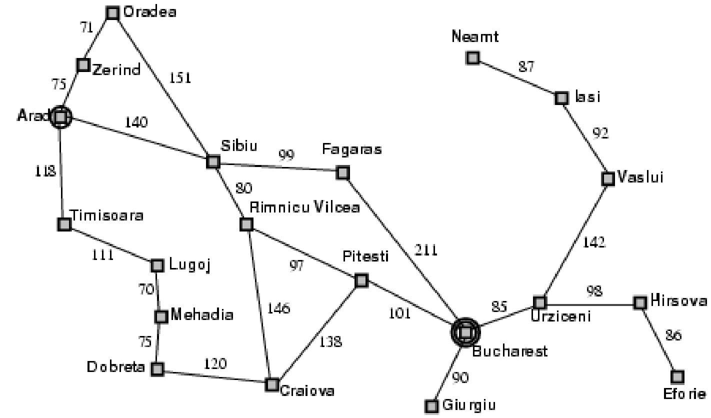
- Deterministic / Stochastic

Previous problem was deterministic: no new percepts were necessary, we can predict the future perfectly

- Discrete / continuous

Previous problem was discrete: we can enumerate all possibilities

State-Space Problem Formulation



A **problem** is defined by five items:

states e.g. cities

initial state e.g., "at Arad"

actions or successor function *S(x)* = set of action–state pairs

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

transition function - maps *action* & *state* → *state*

goal test, (or goal state)

e.g., *x* = "at Bucharest", *Checkmate(x)*

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

State-Space Problem Formulation

- **A statement of a Search problem has components**
  - 1. States
  - 2. A start state S
  - 3. A set of operators/actions which allow one to get from one state to another
  - 4. transition function
  - 5. A set of possible goal states G, or ways to test for goal states
  - 6. Cost path

- **A solution consists of**
  - a sequence of operators which transform S into a goal state G

- **Representing real problems in a State-Space search framework**
  - may be many ways to represent states and operators
  - key idea: represent only the relevant aspects of the problem (**abstraction**)

Abstraction/Modeling

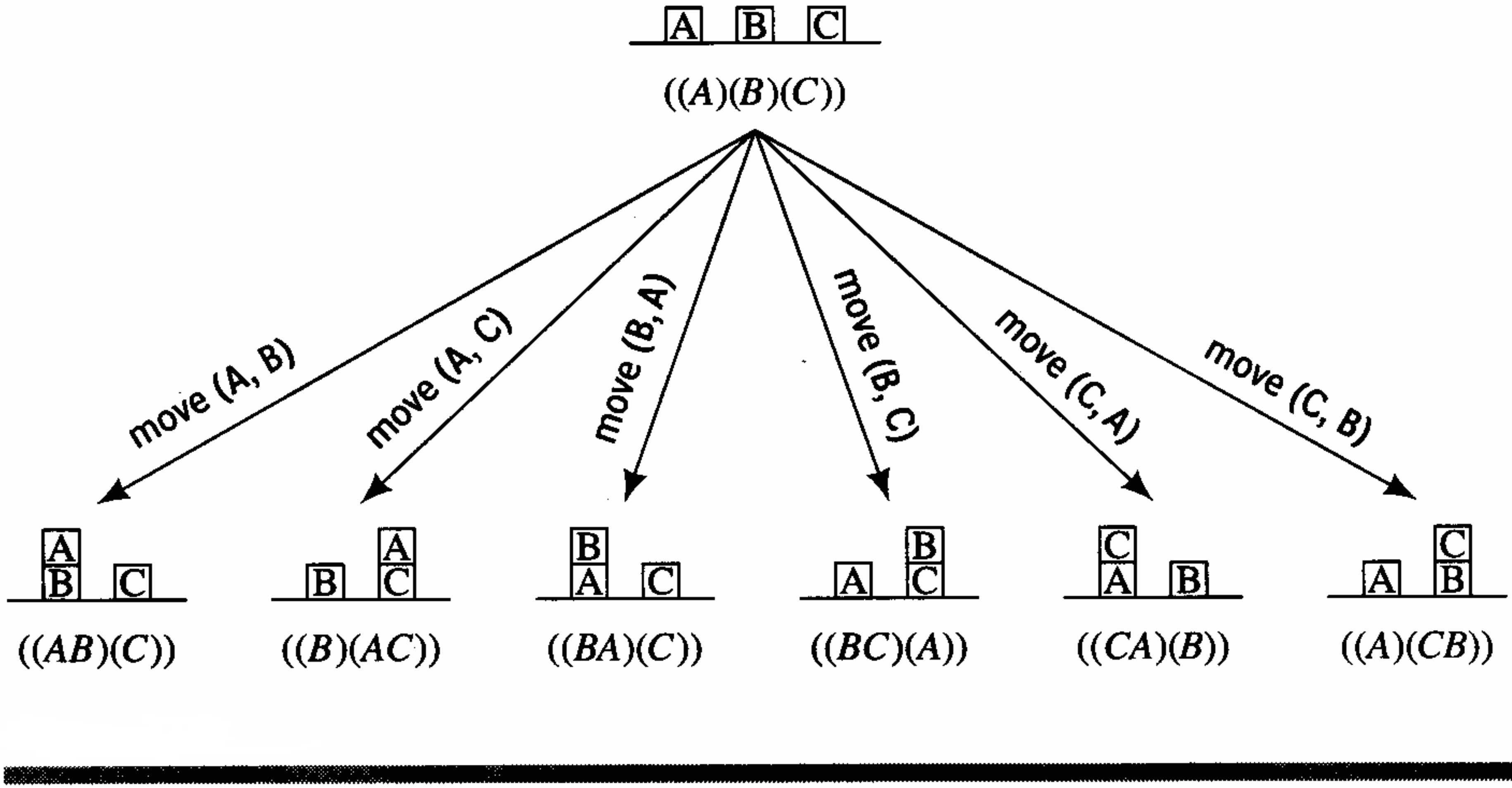
- **Definition of Abstraction (states/actions)**
  - **Process of removing irrelevant detail to create an abstract representation: ``high-level", ignores irrelevant details**
- **Navigation Example: how do we define states and operators?**
  - First step is to abstract "the big picture"
    - i.e., solve a map problem
    - nodes = cities, links = freeways/roads (a high-level description)
    - this description is an abstraction of the real problem
  - Can later worry about details like freeway onramps, refueling, etc

- **Abstraction is critical for automated problem solving**
  - must create an approximate, simplified, model of the world for the computer to deal with: real-world is too detailed to model exactly
  - good abstractions retain all important details
  - an abstraction should be easier to solve than the original problem

Robot block world

- Given a set of blocks in a certain configuration,
- Move the blocks into a goal configuration.
- Example :
  - ((A)(B)(C)) → (ACB)

Operator Description



Effects of Moving a Block

The State-Space Graph

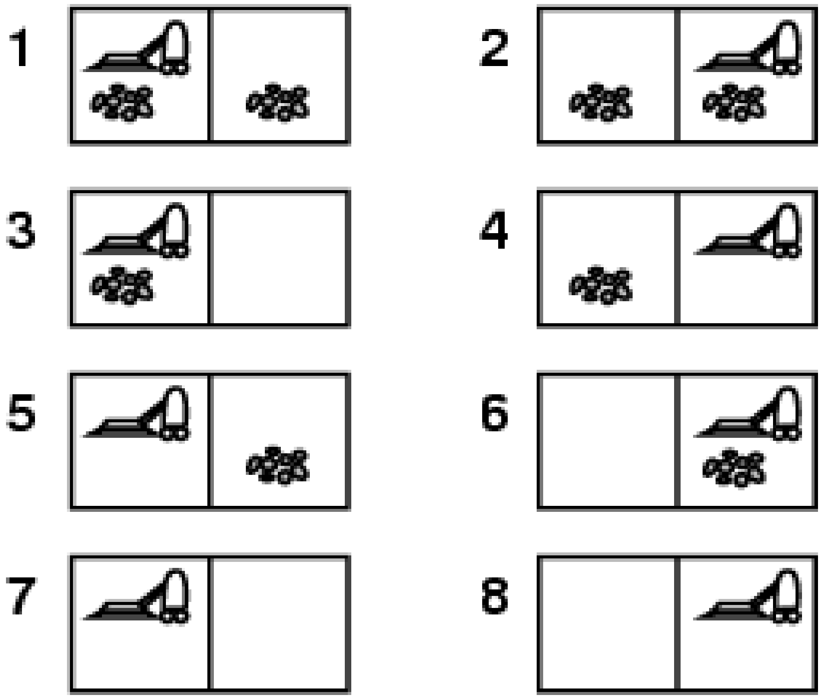
- **Problem formulation:**
  - Give an abstract description of states, operators, initial state and goal state.
- **Graphs:**
  - vertices, edges(arcs), directed arcs, paths
- **State-space graphs:**
  - States are vertices
  - operators are directed arcs
  - solution is a path from start to goal
- **Problem solving activity:**
  - Generate a part of the search space that contains a solution

State-space:

1. A set of states
2. A set of “operators”/transitions
3. A start state S
4. A set of possible goal states
5. Cost path

Example: vacuum world

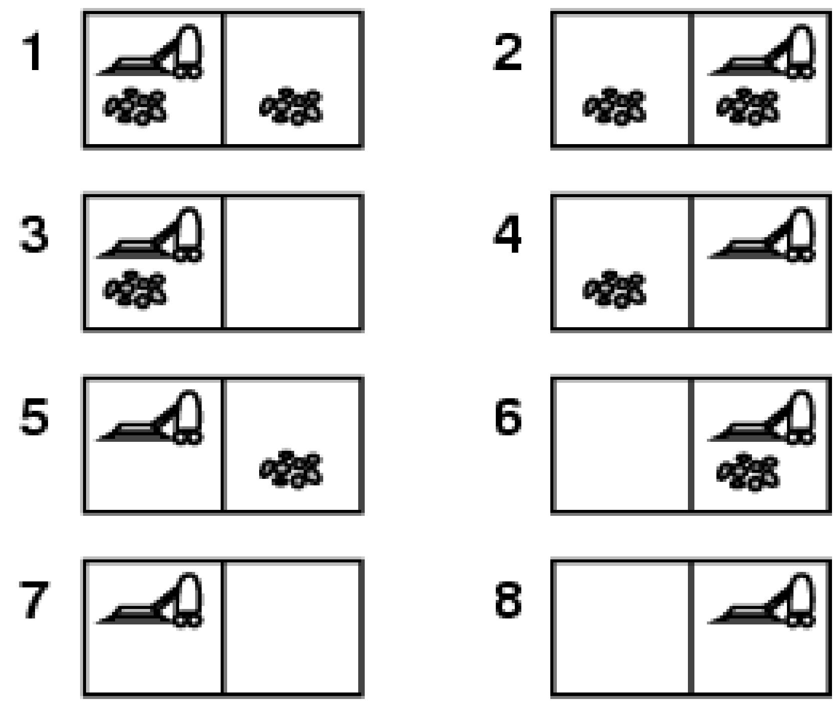
- Observable, start in #5.
  Solution?



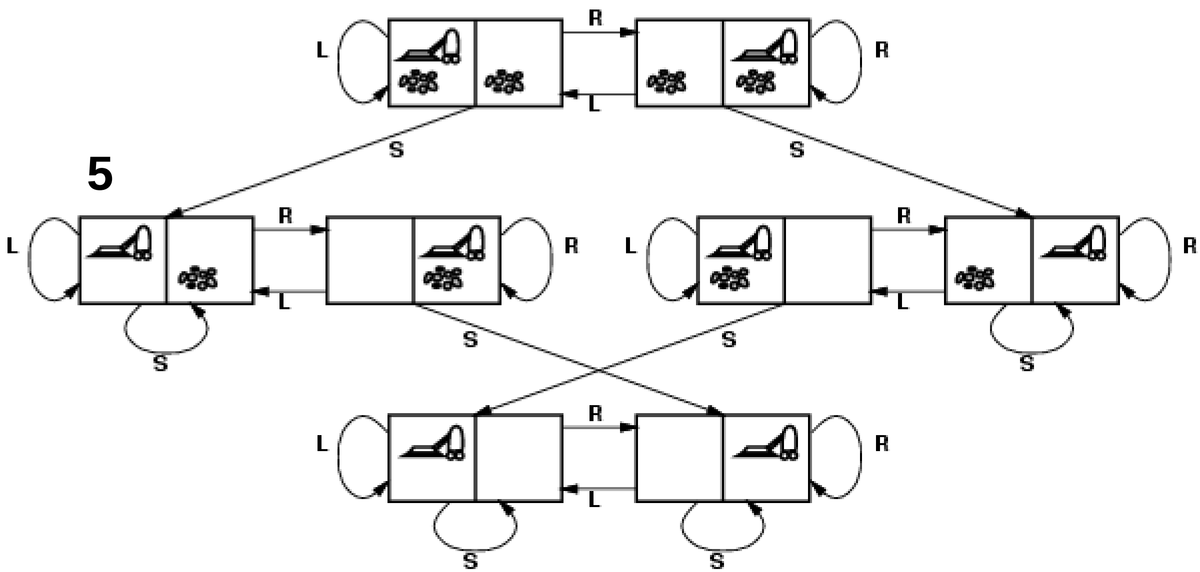
Example: vacuum world

- Observable, start in #5.
  Solution?

  [Right, Suck]

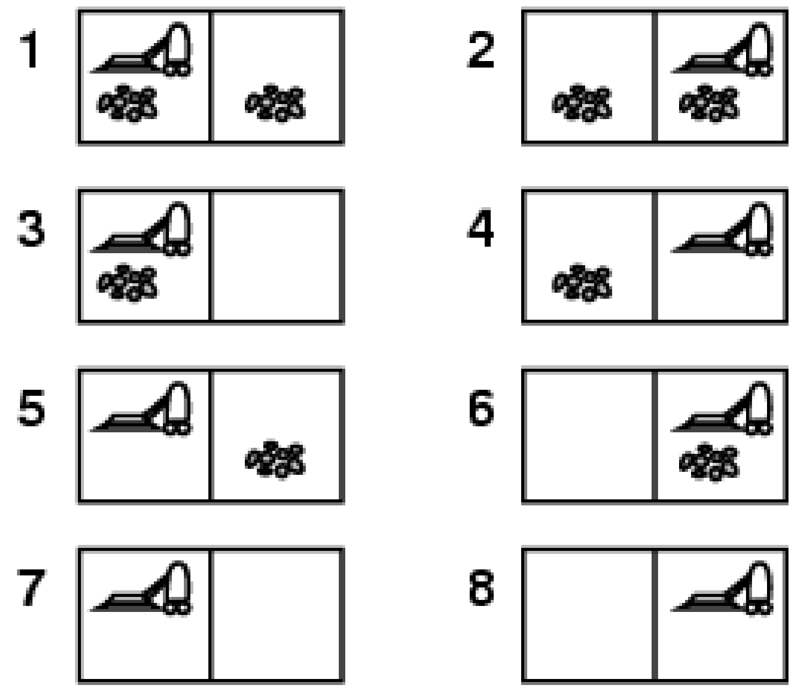


Vacuum world state space graph



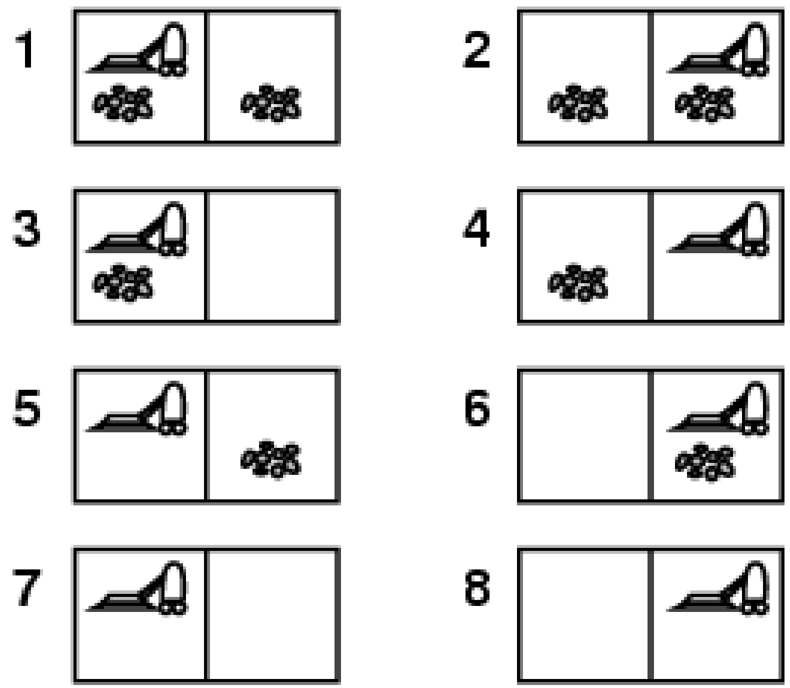
Example: vacuum world

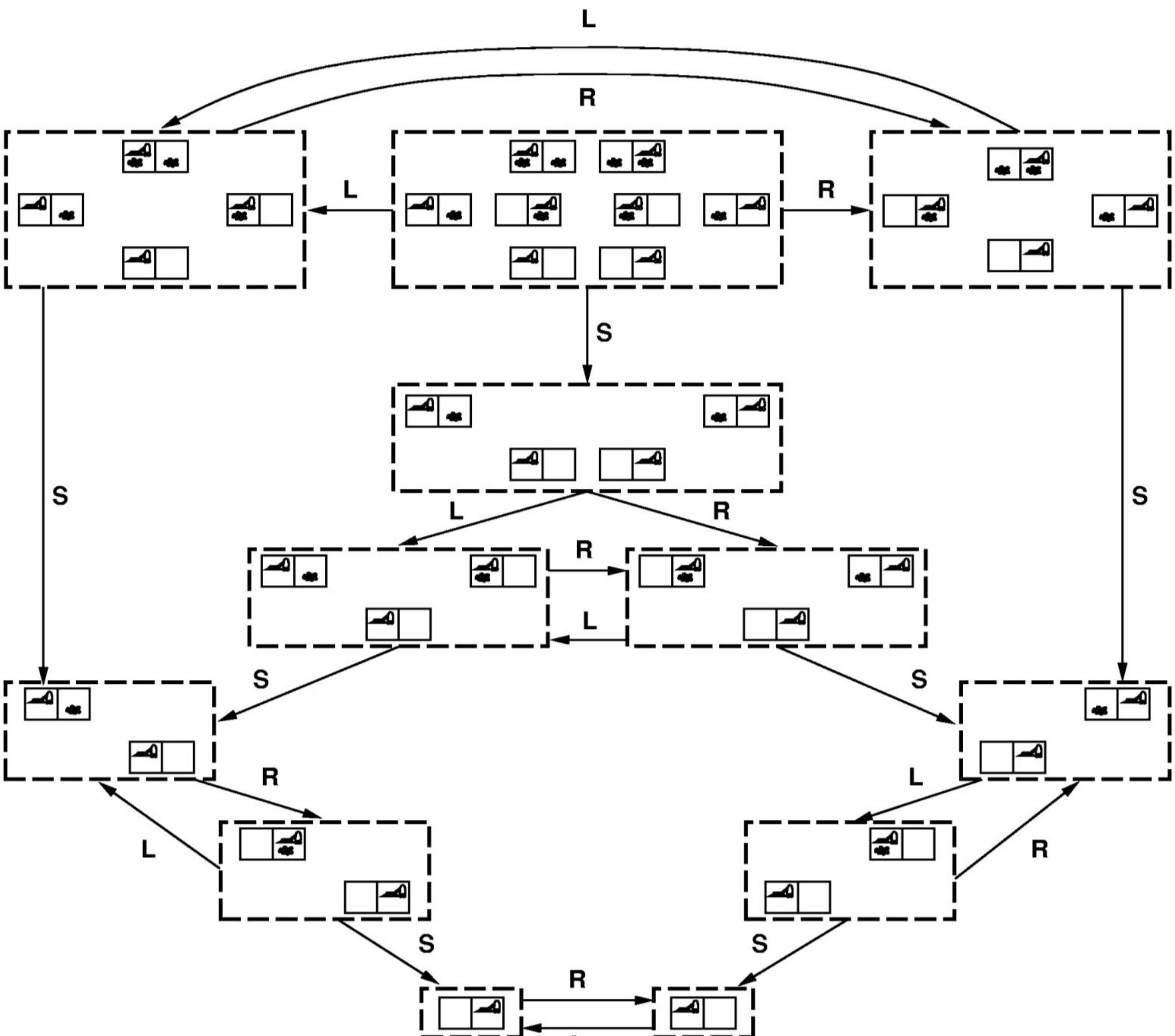
- Unobservable, start in {*1,2,3,4,5,6,7,8*} e.g., Solution?



Example: vacuum world

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

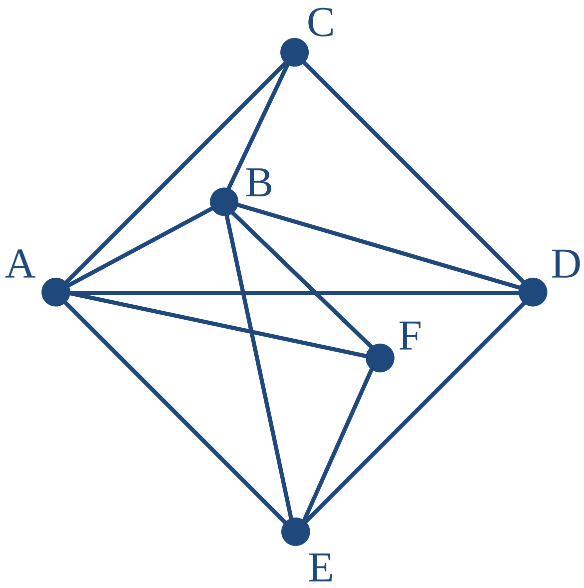




L
R
L
R
S
S
S
L
R
R
L
S
S
R
L
L
R
S
S
R
L

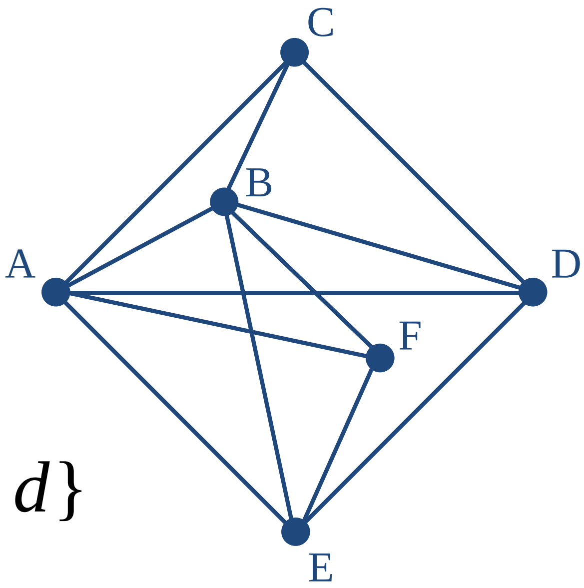
The Traveling Salesperson Problem

- Find the shortest tour that visits all cities without visiting any city twice and return to starting point.
- State:
  - sequence of cities visited
- $S_0 = A$



The Traveling Salesperson Problem

- Find the shortest tour that visits all cities without visiting any city twice and return to starting point.
- State: sequence of cities visited
- $S_0 = A$

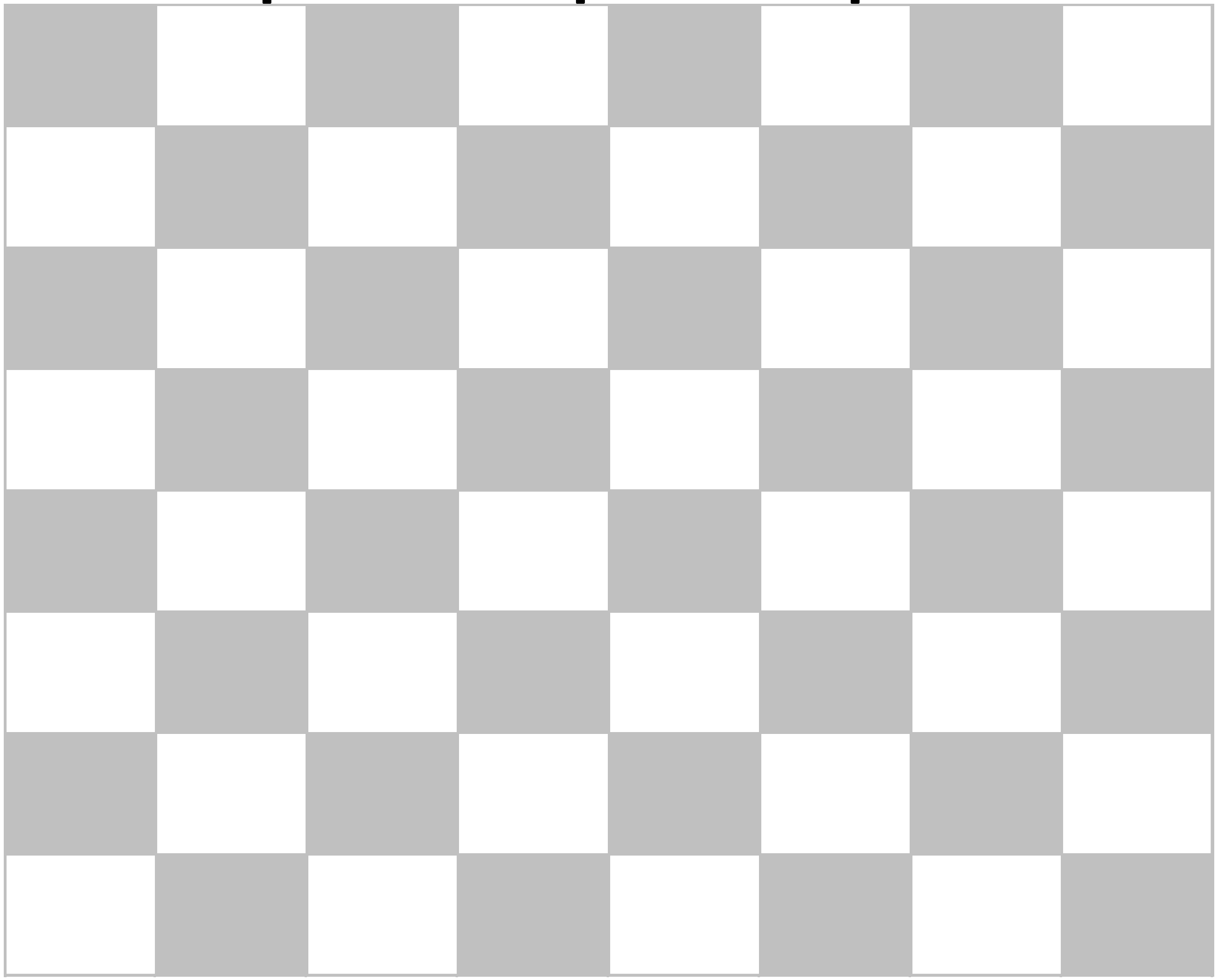


- Solution = a complete tour

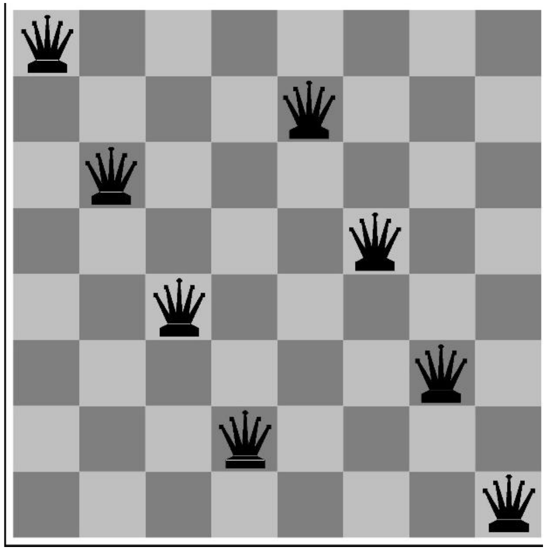
Transition model

$$\{a, c, d\} \quad \rightrightarrows \quad \{(a, c, d, x) \mid X \notin a, c, d\}$$

Example: 8-queen problem



Example: 8-Queens



- states? -any arrangement of n<=8 queens

  -*or* arrangements of n<=8 queens, 1 per column, such that no queen attacks any other (BETTER)

  -*or* arrangements of n<=8 queens in leftmost n columns, 1 per column, such that no queen attacks any other (BEST)
- initial state? no queens on the board
- actions? -add queen to any empty column

  -*or* add queen to leftmost empty column such that it is not attacked by other queens.
- goal test? 8 queens on the board, none attacked.
- path cost? 1 per move

The Sliding Tile Problem

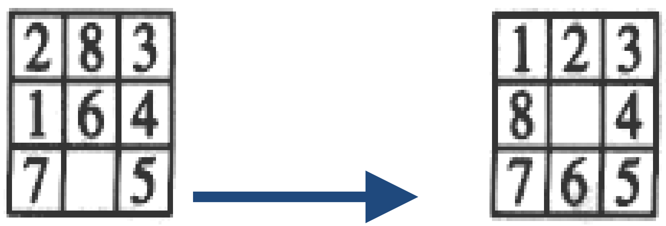


Figure 8.1

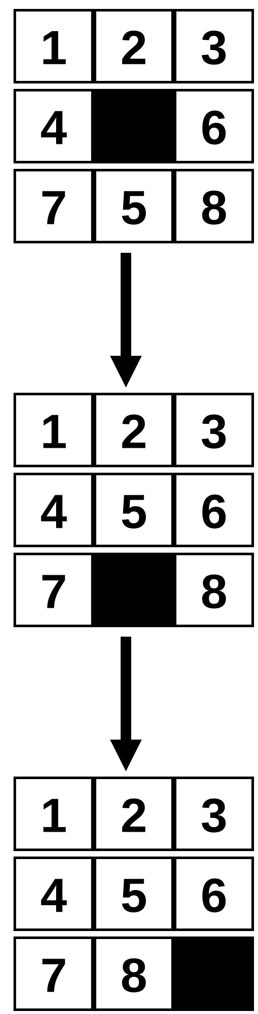
Start and Goal Configurations for the Eight-Puzzle

$$move(x, loc\, y, loc\, z)$$

Up
Down
Left
Right

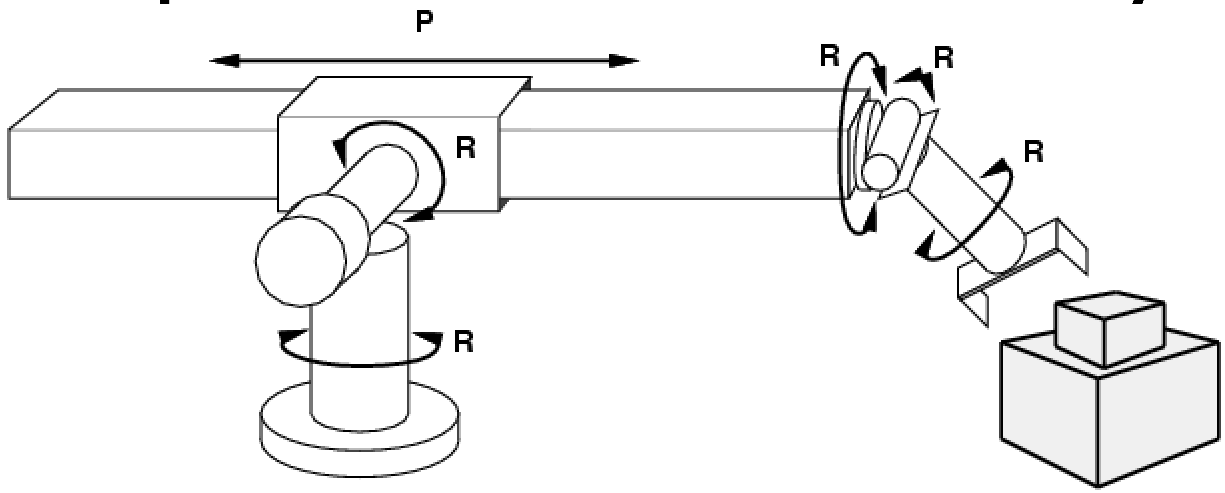
The “8-Puzzle” Problem

Start State



Goal State

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

new

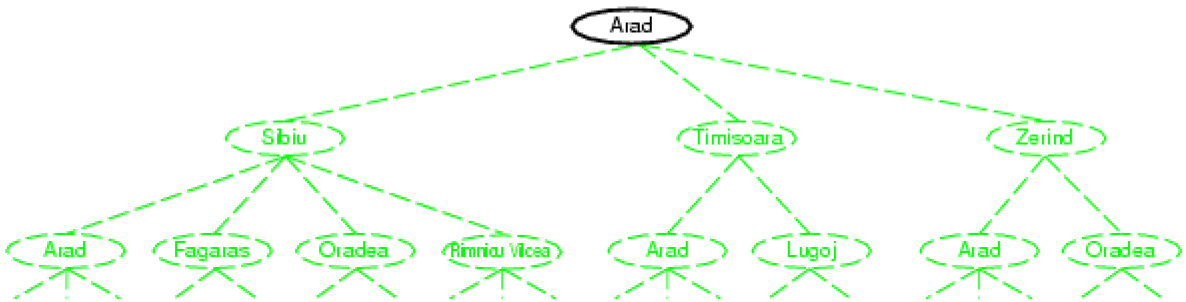
Formulating Problems; Another Angle

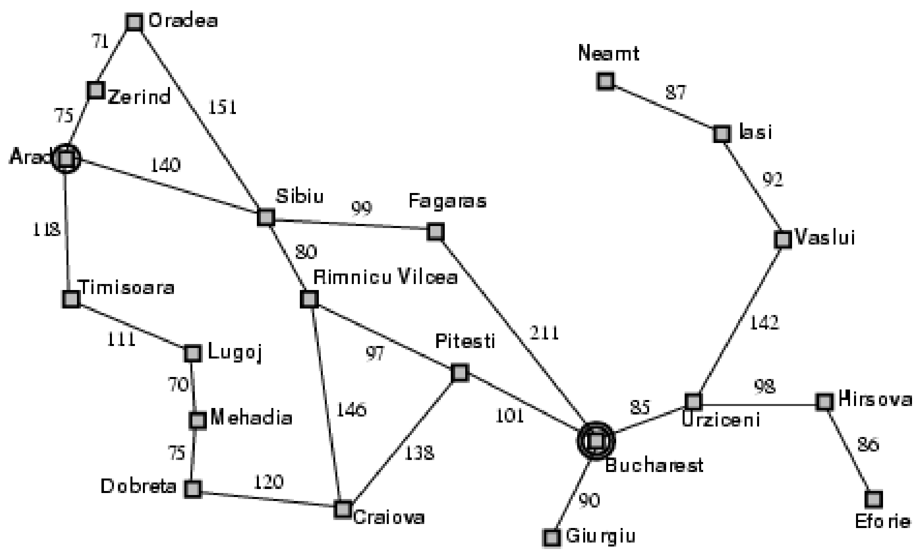
- **Problem types**
  - Satisfying: 8-queen
  - Optimizing: Traveling salesperson
    - For traveling salesperson satisfying easy, optimizing hard
- **Goal types**
  - board configuration
  - sequence of moves
  - A strategy (contingency plan)
- **Satisfying leads to optimizing since “small is quick”**
- For traveling salesperson
  - satisfying easy, optimizing hard
- **Semi-optimizing:**
  - Find a good solution
- **In Russel and Norvig:**
  - single-state, multiple states, contingency plans, exploration problems

Searching the State Space

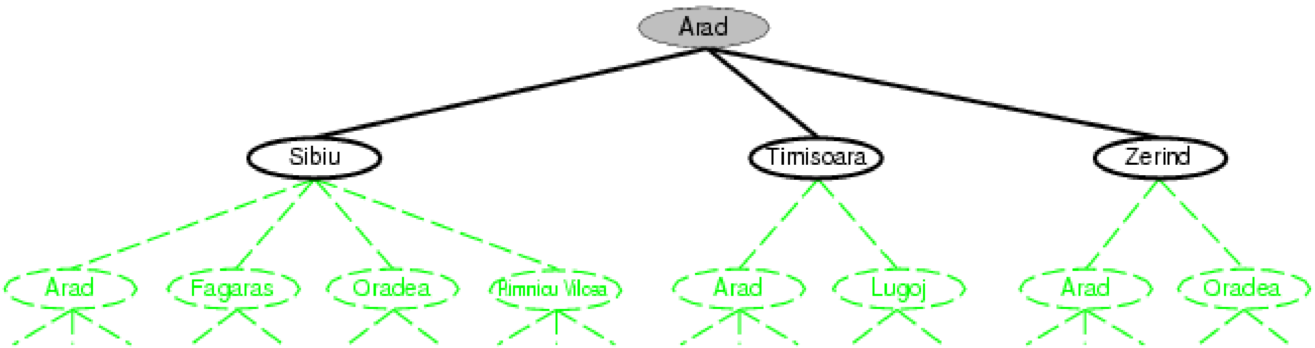
- Exploration of the state space
  - states, operators
  - by generating successors of already explored states (aka **expanding** states)
- The search space graph is implicit
- **Control strategy** generates a search tree.
- Systematic search
  - Do not leave any stone unturned
- Efficiency
  - Do not turn any stone more than once

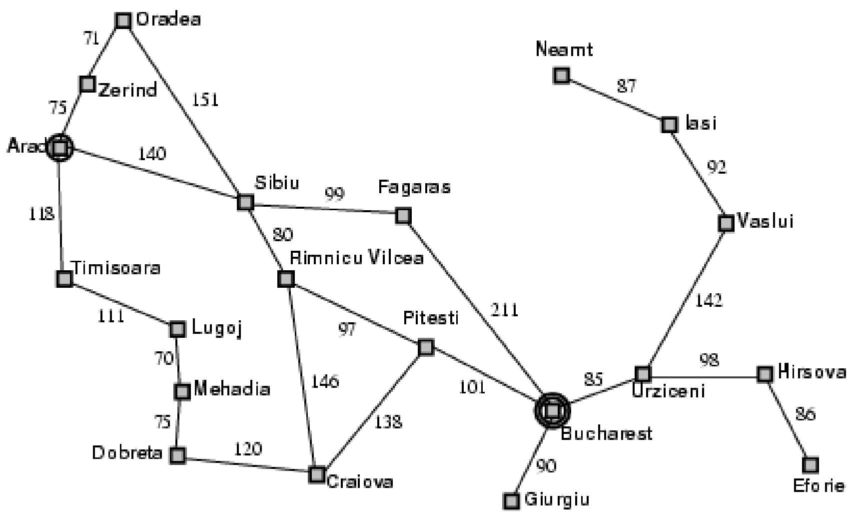
Tree search example



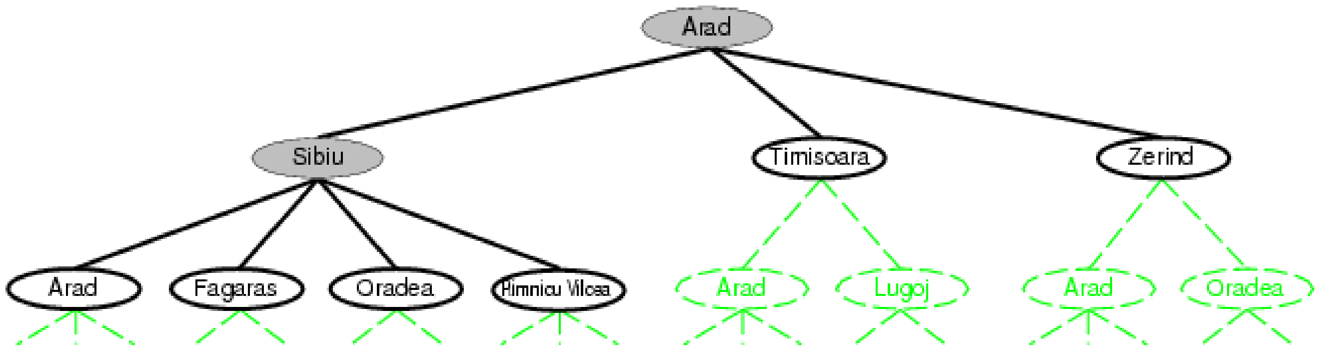


Tree search example





Tree search example



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

State-Space Graph of the 8 Puzzle Problem

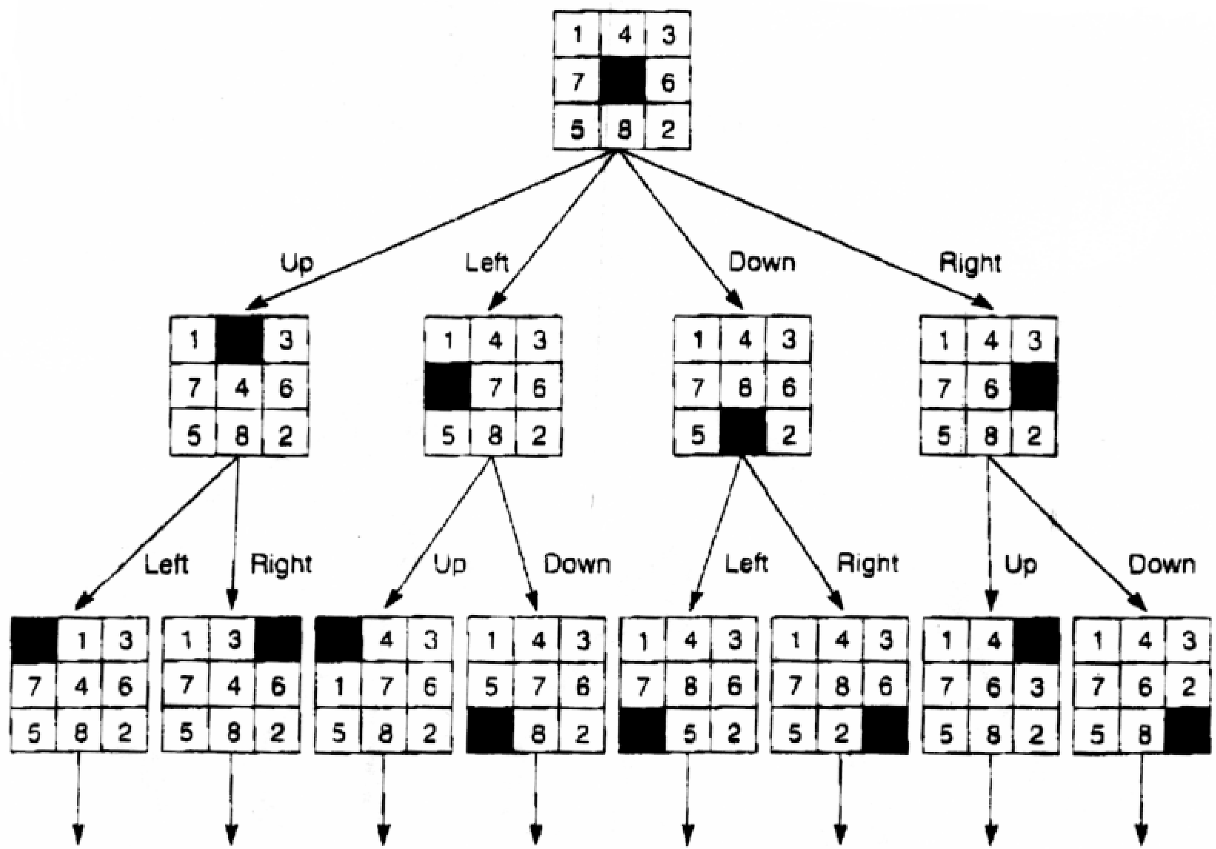
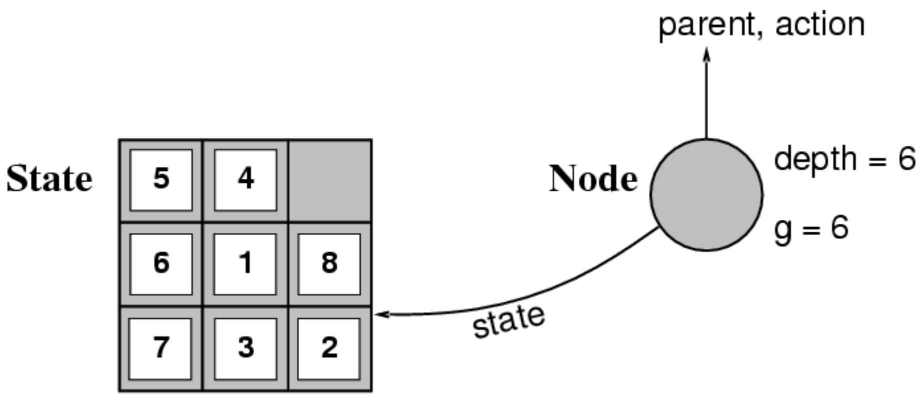


Figure 3.6 State space of the 8-puzzle generated by "move blank" operations.

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 contains info such as: 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.
- Queue managing frontier :
  - FIFO
  - LIFO
  - priority

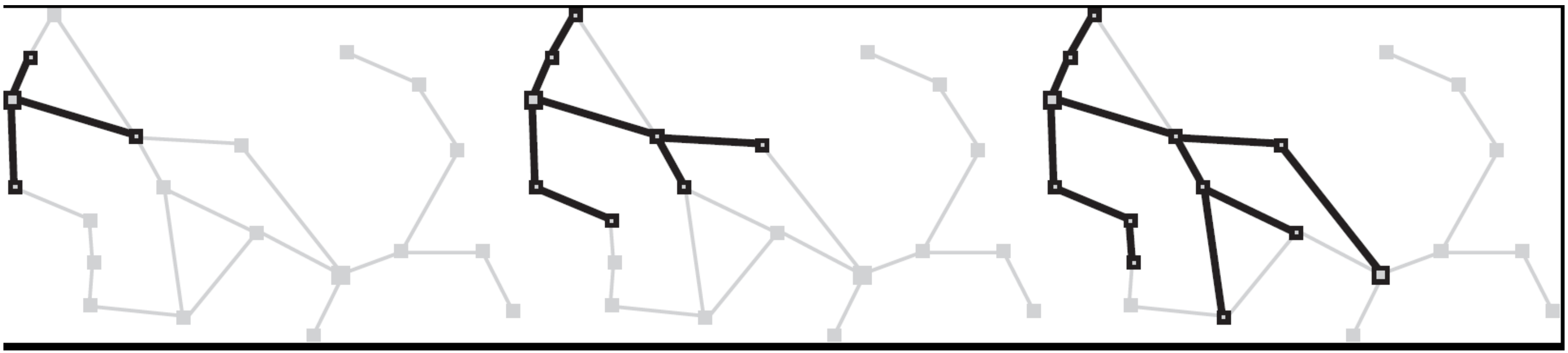
Tree-Search vs Graph-Search

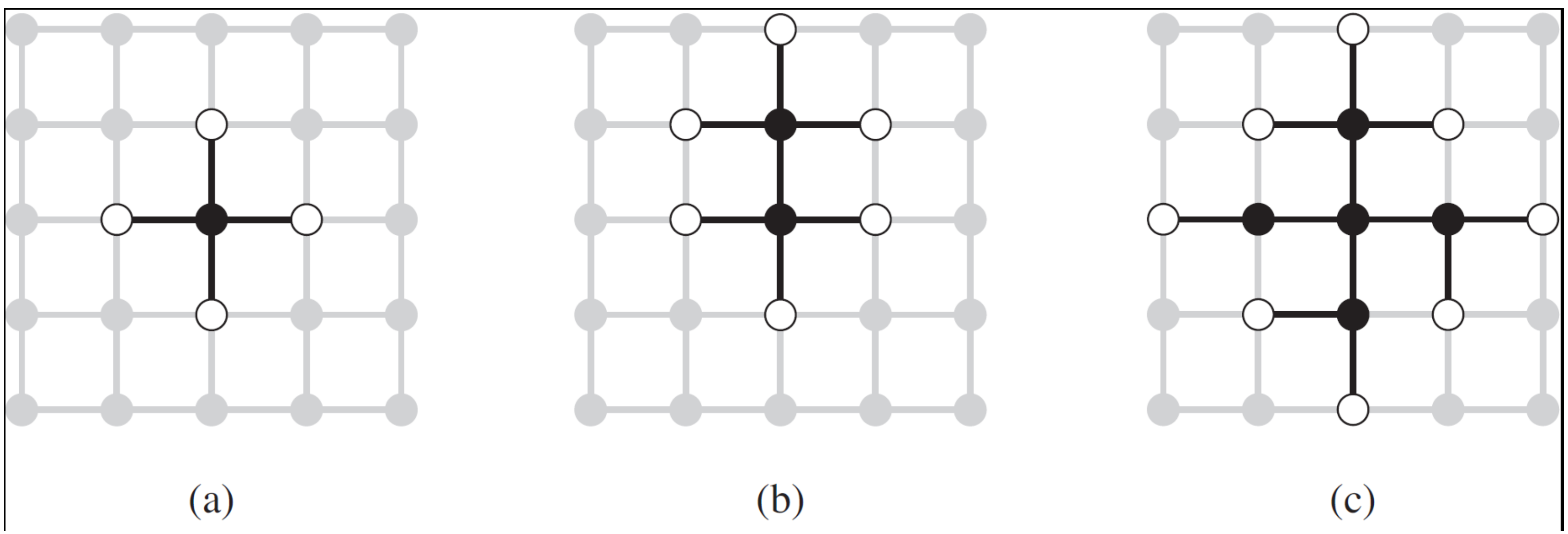
- **Tree-search(problem),** returns a solution or failure
- Frontier ← initial state
- Loop do
  - If frontier is empty return failure
  - Choose a leaf node and remove from frontier
  - If the node is a goal, return the corresponding solution
  - Expand the chosen node, adding its children to the frontier
  - ---
- **Graph-search(problem),** returns a solution or failure
- Frontier ← initial state, *explored* ←*empty*
- Loop do
  - If frontier is empty return failure
  - Choose a leaf node and remove from frontier
  - If the node is a goal, **return** the corresponding solution.
  - *Add the node to the explored*.
  - Expand the chosen node, adding its children to the frontier, *only if not in frontier or explored set*

Basic search scheme

- We have 3 kinds of states
  - explored (past) – only graph search
  - frontier (current)
  - unexplored (future) – implicitly given
- Initially frontier=start state
- Loop until found solution or exhausted state space
  - pick/remove first node from frontier using search strategy
    - priority queue – FIFO (BFS), LIFO (DFS), g (UCS), f (A*), etc.
  - add this node to explored,
  - expand this node, add children to frontier (graph search : only those children whose state is not in explored list)
  - Q: what if better path is found to a node already on explored list?

Graph-Search





Tree-Search vs. Graph-Search

- Example : Assemble 5 objects {**a, b, c, d, e**}
- A state is a bit-vector (length 5), 1=object in assembly
- 11010 = **a, b, d** in assembly, **c, e** not
- State space
  - number of states $2^5 = 32$
  - number of edges $(2^5) \cdot 5 \cdot ½ = 80$
- Tree-search space
  - number of nodes 5! = 120
- State can be reached in multiple ways
  - 11010 can be reached **a**+**b**+**d** or **a**+**d**+**b** etc.
- Graph-search :
  - three kinds of nodes : **unexplored**, **frontier**, **explored**
  - before adding a node, check if a state is in **frontier** or **explored** set

Why Search Can be Difficult

- **At the start of the search, the search algorithm does not know**
  - the size of the tree
  - the shape of the tree
  - the depth of the goal states

- **How big can a search tree be?**
  - say there is a constant branching factor b
  - and one goal exists at depth d
  - search tree which includes a goal can have
    b^d different branches in the tree (worst case)

- Examples:
  - b = 2, d = 10: $b^d = 2^{10} = 1024$
  - b = 10, d = 10: $b^d = 10^{10} = 10,000,000,000$

Searching the Search Space

- Uninformed (Blind) search : don’t know if a state is “good”
  - Breadth-first
  - Uniform-Cost first
  - Depth-first
  - Iterative deepening depth-first
  - Bidirectional
  - Depth-First Branch and Bound
- Informed Heuristic search : have evaluation fn for states
  - Greedy search, hill climbing, Heuristics
- Important concepts:
  - Completeness : does it always find a solution if one exists ?
  - Time complexity (b, d, m)
  - Space complexity (b, d, m)
  - Quality of solution : optimality = does it always find best solution?

Search strategies

- A search strategy is defined by picking the order of node expansion

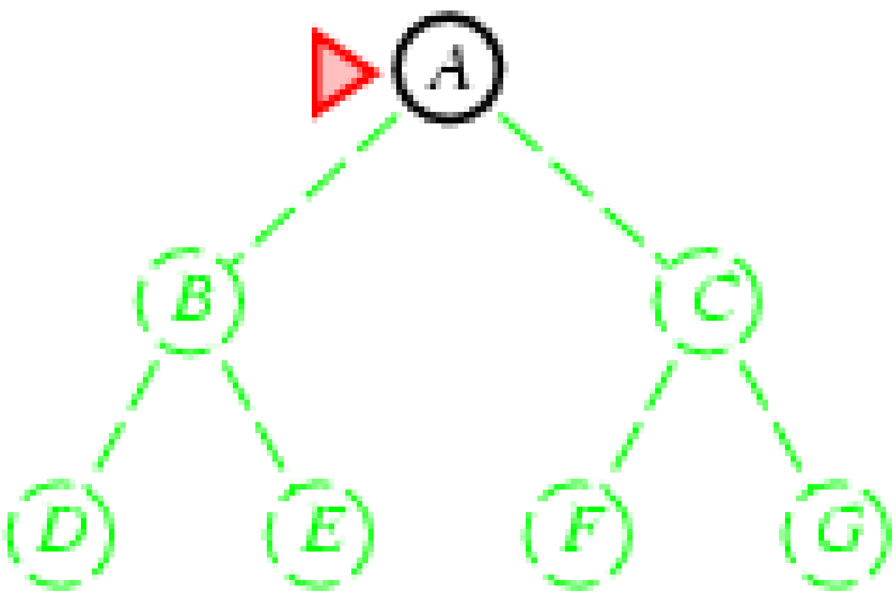
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?

- 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 ∞)

Breadth-First Search

- Expand shallowest unexpanded node
- Frontier: nodes waiting in a queue to be explored, also called OPEN
- Implementation:
  - *frontier* is a first-in-first-out (FIFO) queue, i.e., new successors go at end of the queue.

Is A a goal state?

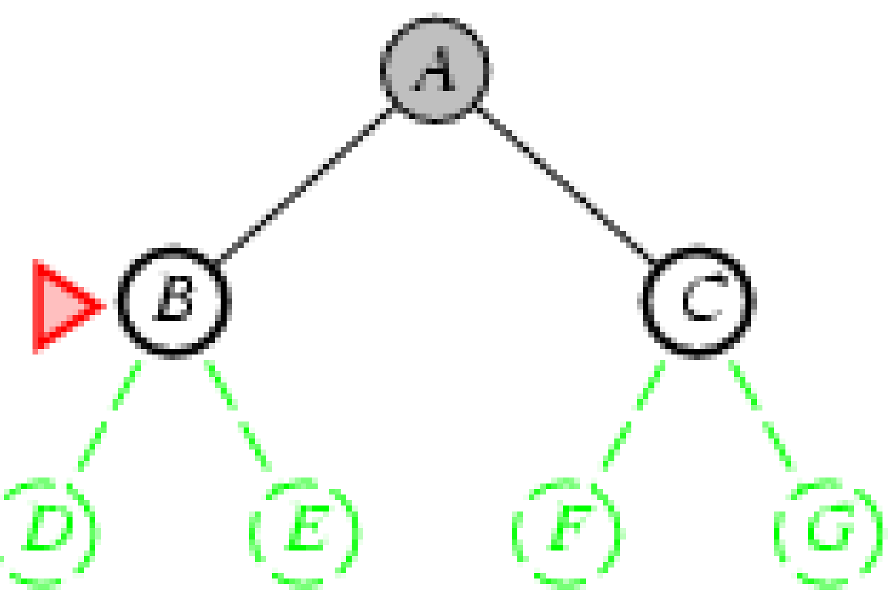


Breadth-First Search

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

Expand:
frontier = [B,C]

Is B a goal state?

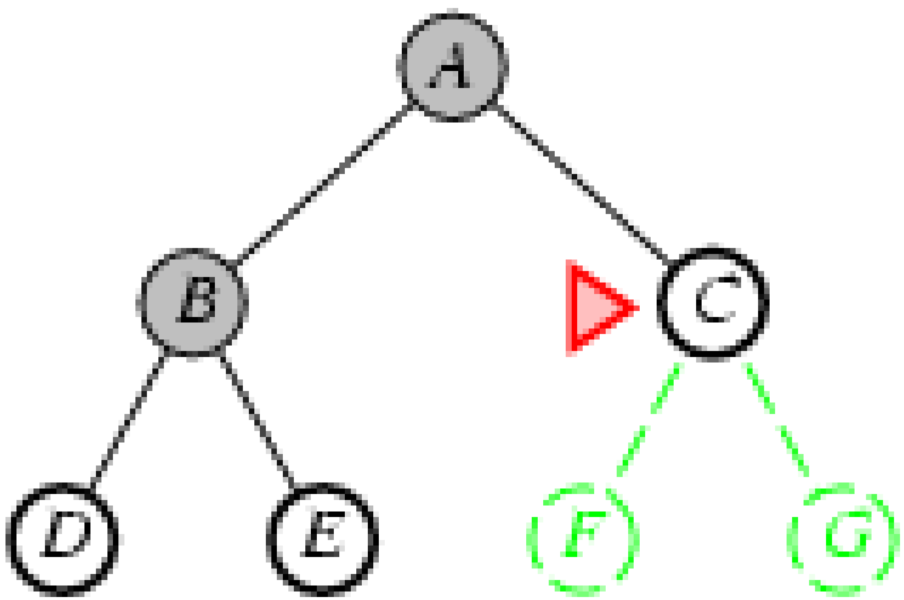


Breadth-First Search

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

Expand:
frontier=[C,D,E]

Is C a goal state?

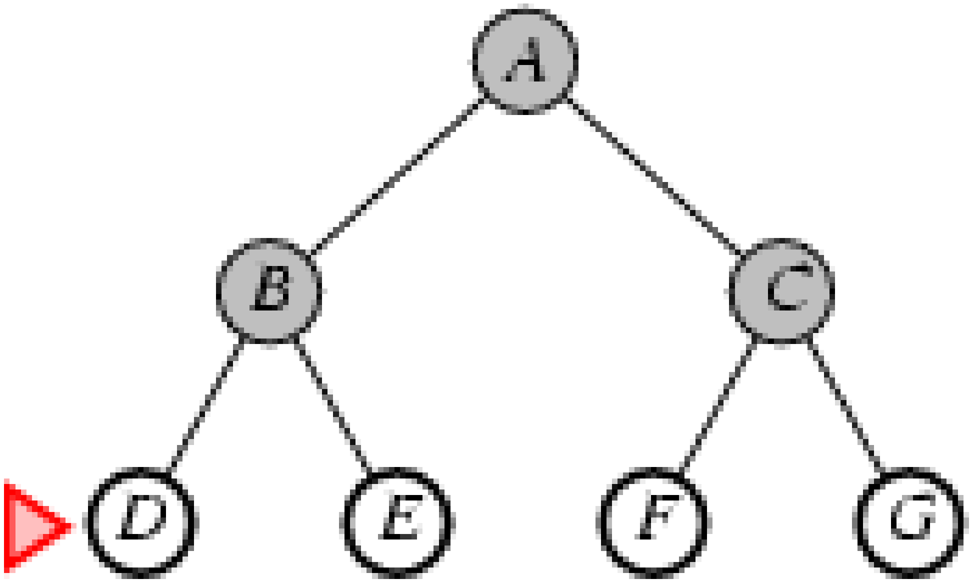


Breadth-First Search

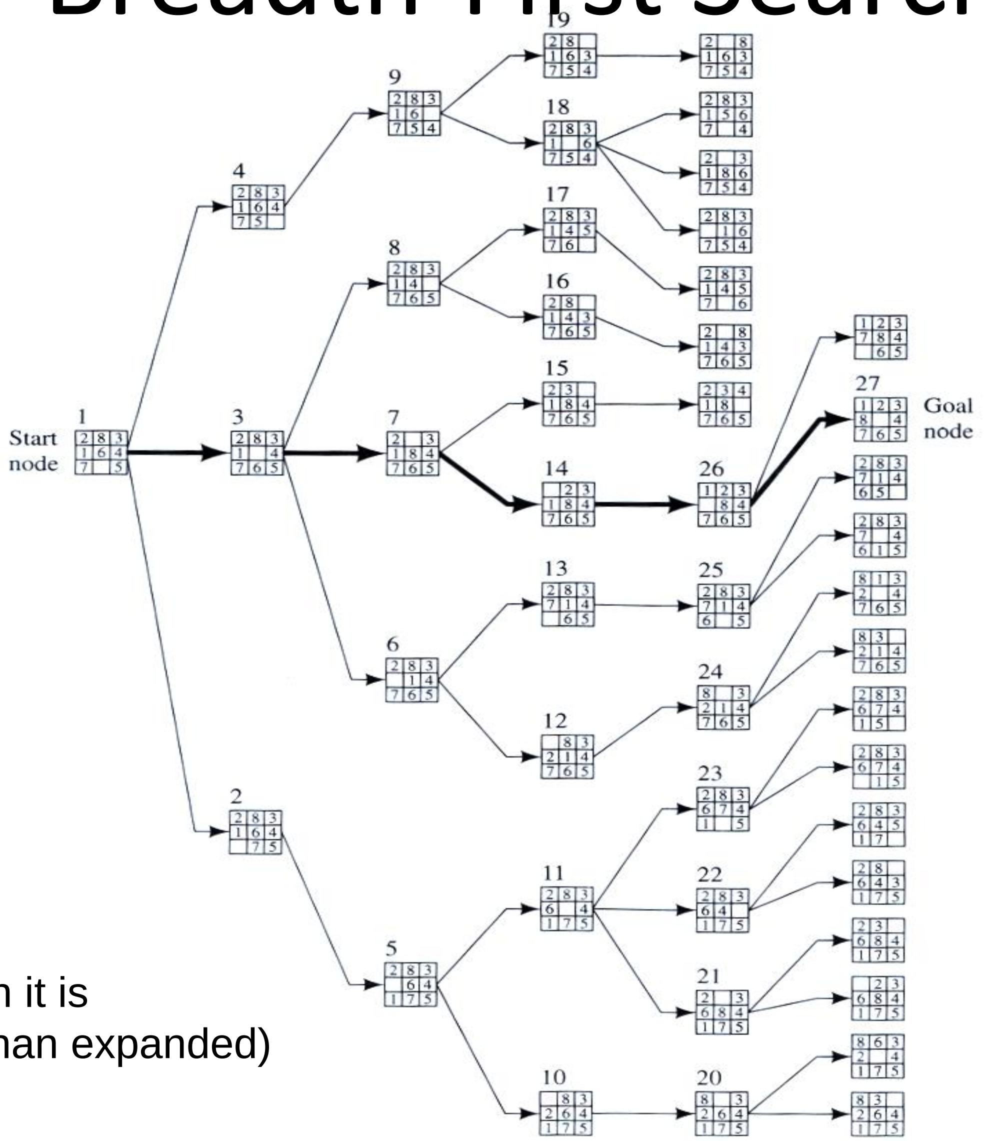
- Expand shallowest unexpanded node
- Implementation:
  - *frontier* is a FIFO queue, i.e., new successors go at end

Expand:
frontier=[D,E,F,G]

Is D a goal state?



Breadth-First Search



Actually, in BFS we
can check if a node
is a goal node when it is
generated (rather than expanded)

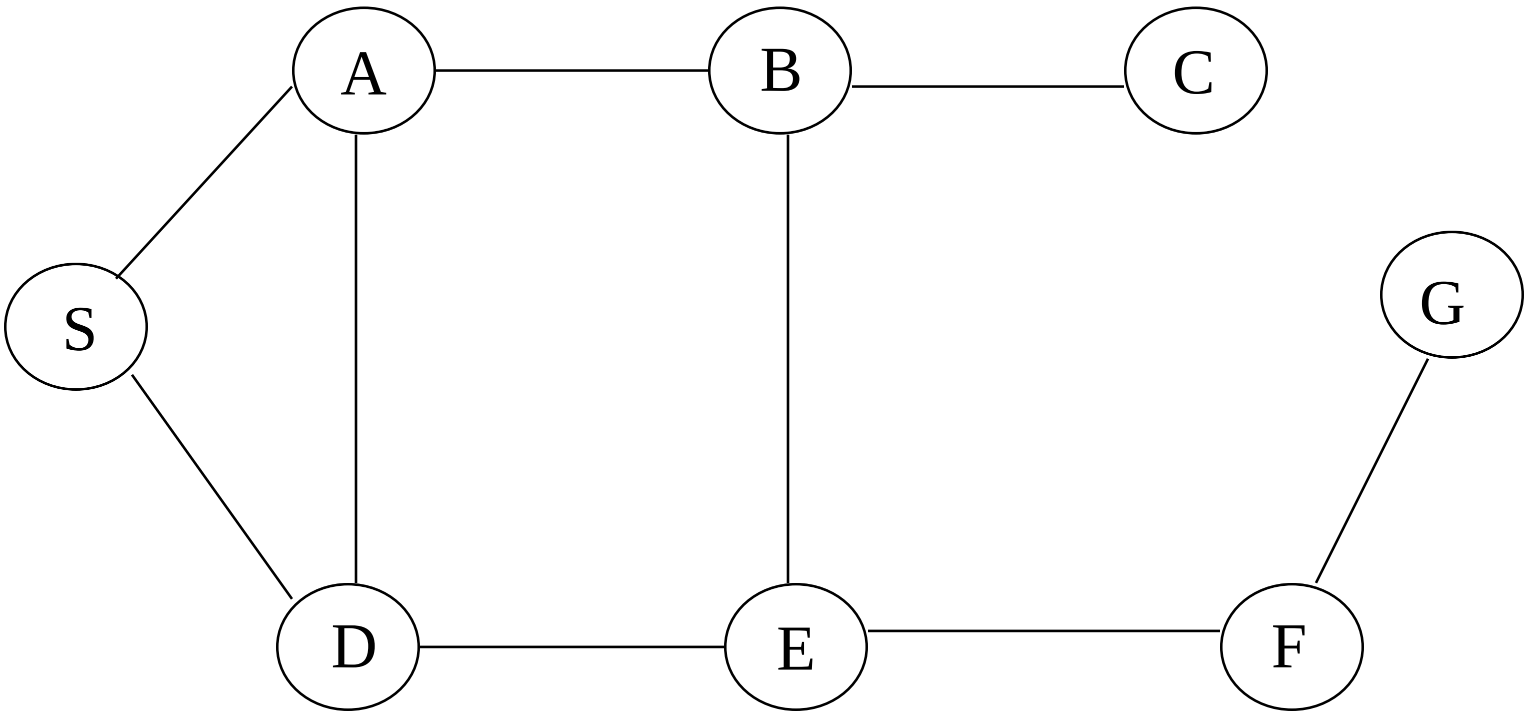
Breadth-First-Search (*)

OPEN = frontier, CLOSED = explored

- 1. Put the start node *s* on OPEN
- 2. If OPEN is empty exit with failure.
- 3. Remove the first node *n* from OPEN and place it on CLOSED.
- 4. Expand *n*, generating all its successors.
  - If child is not in CLOSED or OPEN, then
  - If child is not a goal, then put them at the end of OPEN in some order.
- 5. If *n* is a goal node, exit successfully with the solution obtained by tracing back pointers from *n* to *s*.
- Go to step 2.

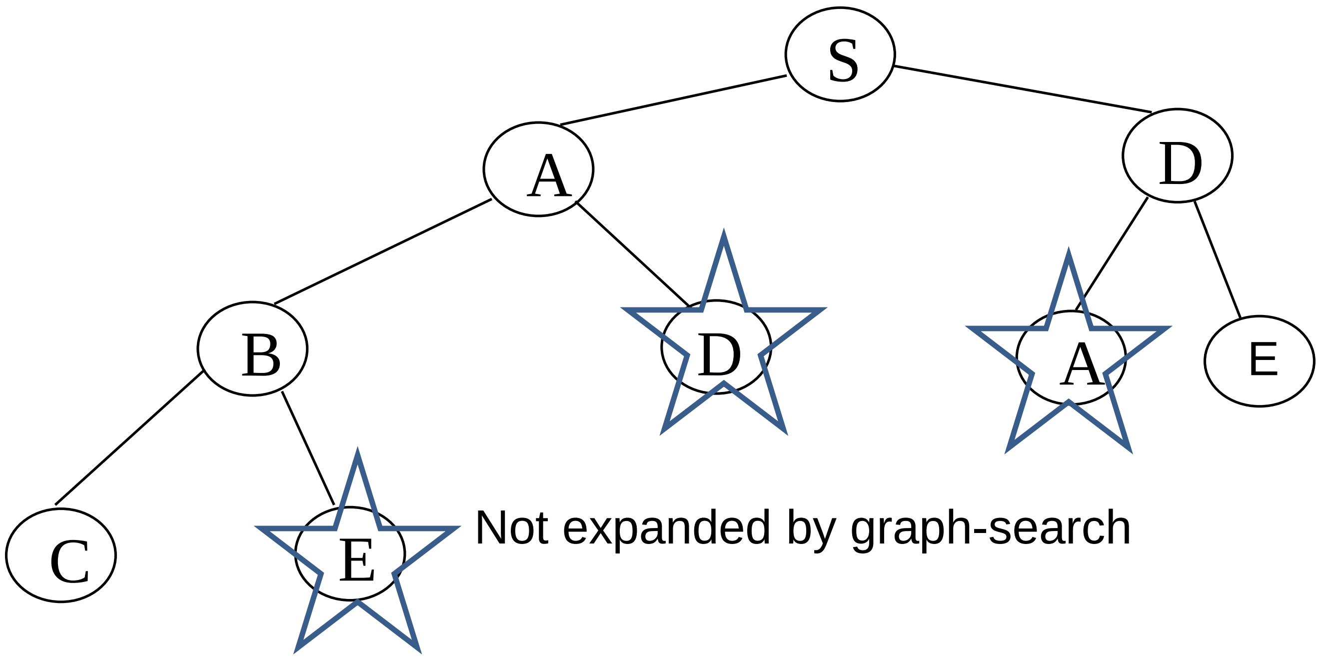
* This is graph-search

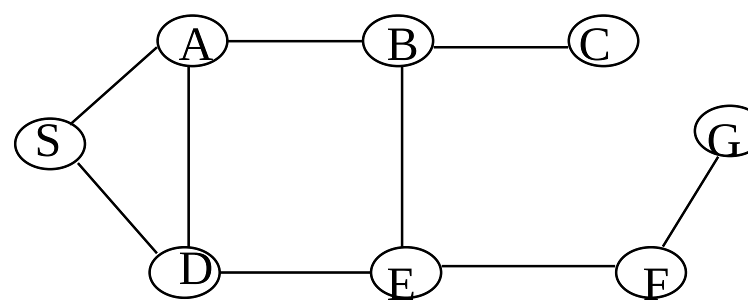
Example: Map Navigation



S = start,   G = goal, other nodes = intermediate states, links = legal transitions

Initial BFS Search Tree



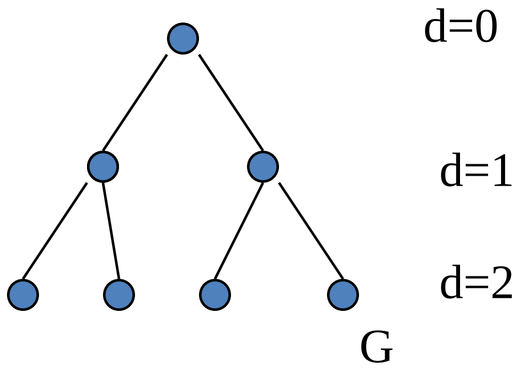


Note: this is the search tree at some particular point in in the search.

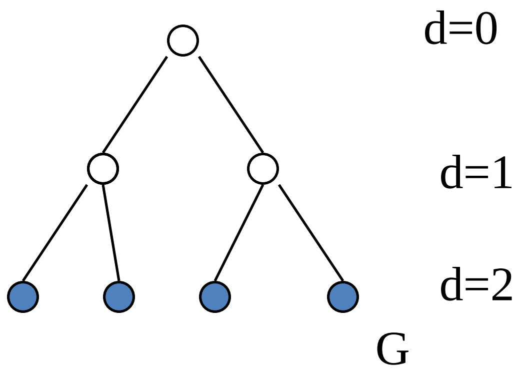
Complexity of Breadth-First Search

- **Time Complexity**
  - assume (worst case) that there is 1 goal leaf at the RHS
  - so BFS will expand all nodes

    $= 1 + b + b^2 + \ \ldots\ldots\ldots + b^d$
    $= \mathbf{O\ (b^d)}$



- **Space Complexity**
  - how many nodes can be in the queue (worst-case)?
  - at depth d there are b^d unexpanded nodes in the Q = $\mathbf{O\ (b^d)}$



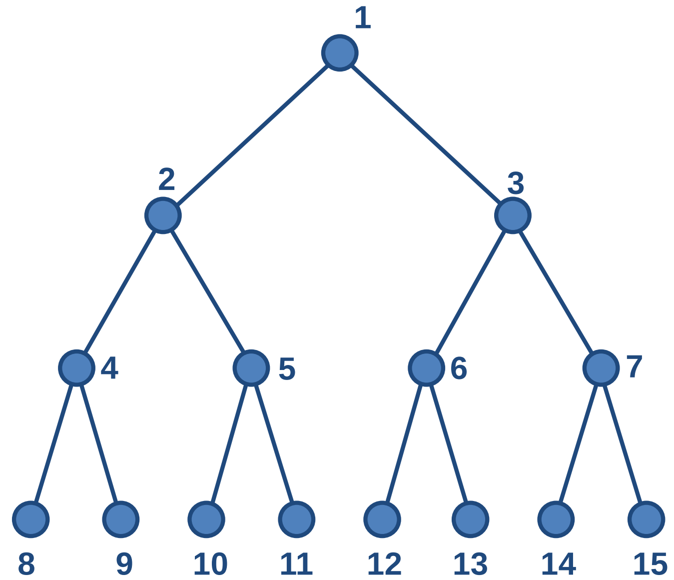
Examples of Time and Memory Requirements for Breadth-First Search

Depth of Solution	Nodes Expanded	Time	Memory
0	1	1 millisecond	100 bytes
2	111	0.1 seconds	11 kbytes
4	11,111	11 seconds	1 megabyte
8	10^8	31 hours	11 giabytes
12	10^{12}	35 years	111 terabytes

Assuming b=10, 1000 nodes/sec, 100 bytes/node

Breadth-First Search (BFS) Properties

- Solution Length: optimal
- Expand each node once (can check for duplicates, performs graph-search)
- Search Time: $O(b^d)$
- Memory Required: $O(b^d)$
- Drawback: requires exponential space



Uniform Cost Search

- Expand lowest-cost OPEN node ($g(n)$)
- In BFS $g(n) = depth(n)$

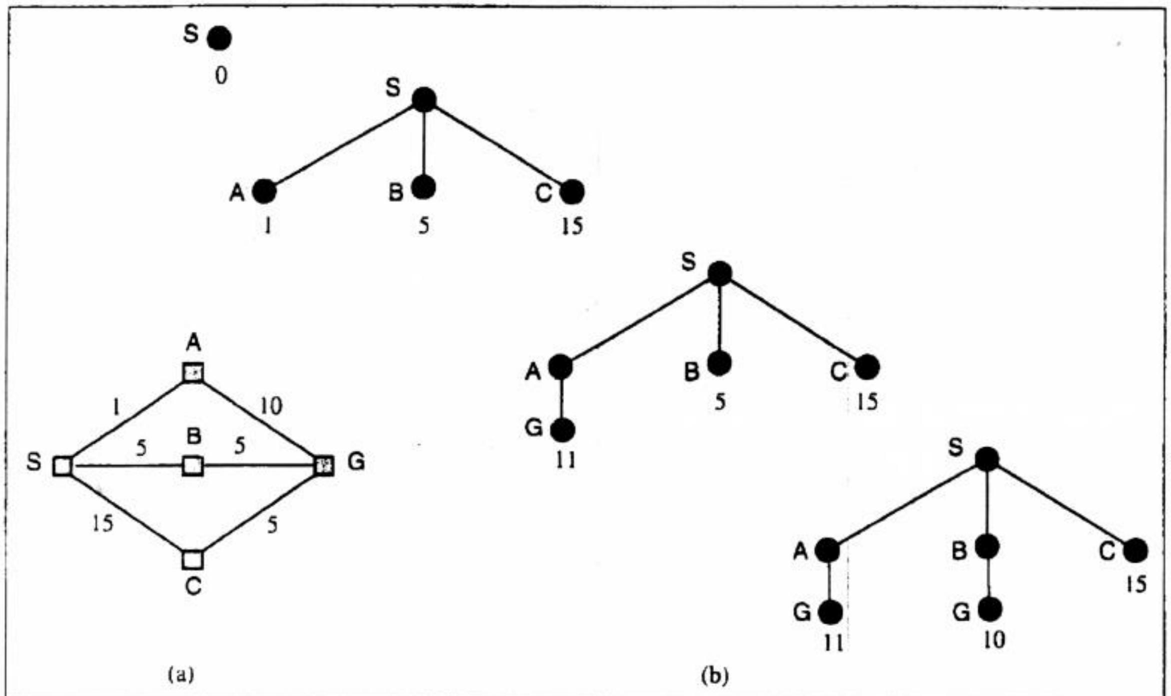


Figure 3.13 A route-finding problem. (a) The state space, showing the cost for each operator. (b) Progression of the search. Each node is labelled with $g(n)$. At the next step, the goal node with $g = 10$ will be selected.

- Requirement
  - $g(\text{successor})(n)) \geq g(n)$

Uniform cost search

1. Put the start node *s* on OPEN
2. If OPEN is empty exit with failure.
3. Remove the first node *n* from OPEN and place it on CLOSED.
4. If *n* is a goal node, exit successfully with the solution obtained by tracing back pointers from *n* to *s*.
5. Otherwise, expand *n*, generating all its successors attach to them pointers back to *n*, and put them in OPEN *in order of shortest cost*
6. Go to step 2.

DFS Branch and Bound

At step 4: compute the cost of the solution found and update the upper bound U.

at step 5: **expand *n*, generating all its successors attach to them pointers back to *n*, and put on top of OPEN*.***

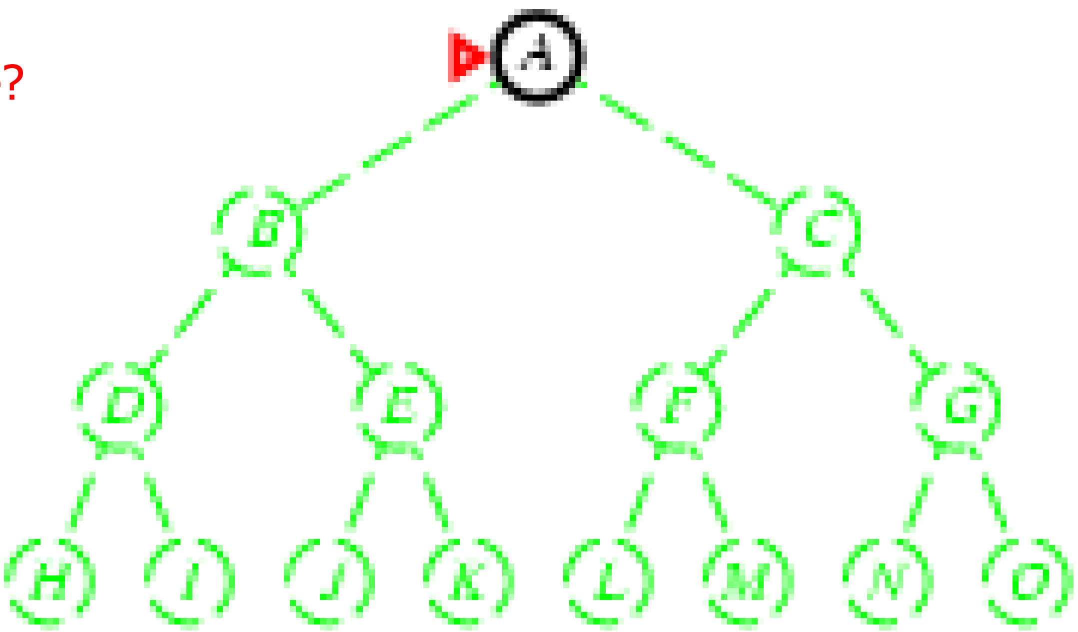
Compute cost of partial path to node and prune if larger than U.

.

Depth-First Search

- Expand *deepest* unexpanded node
- Implementation:
  - *frontier* = Last In First Out (LIFO) queue, i.e., put successors at front

Is A a goal state?

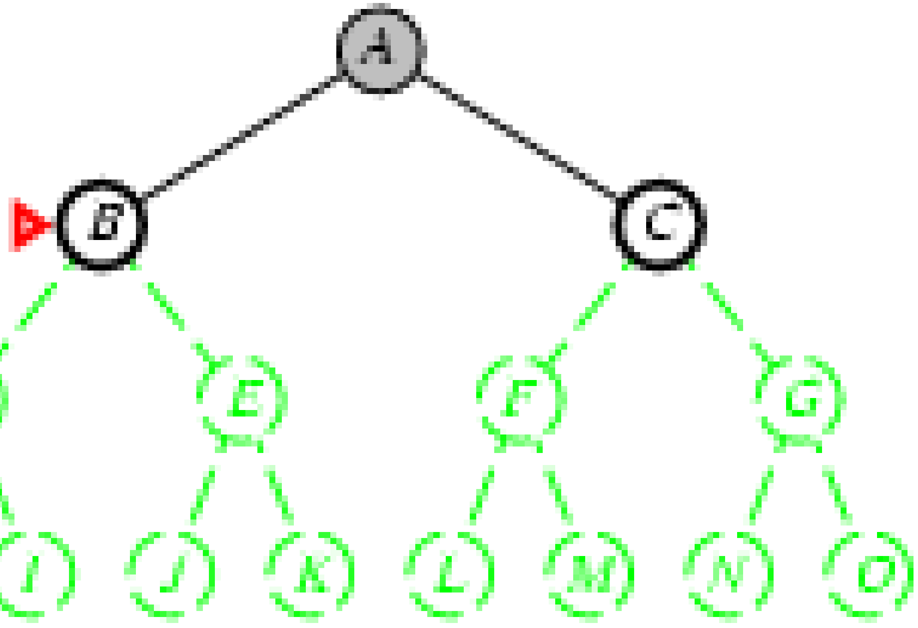


Depth-first search

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

queue=[B,C]

Is B a goal state?

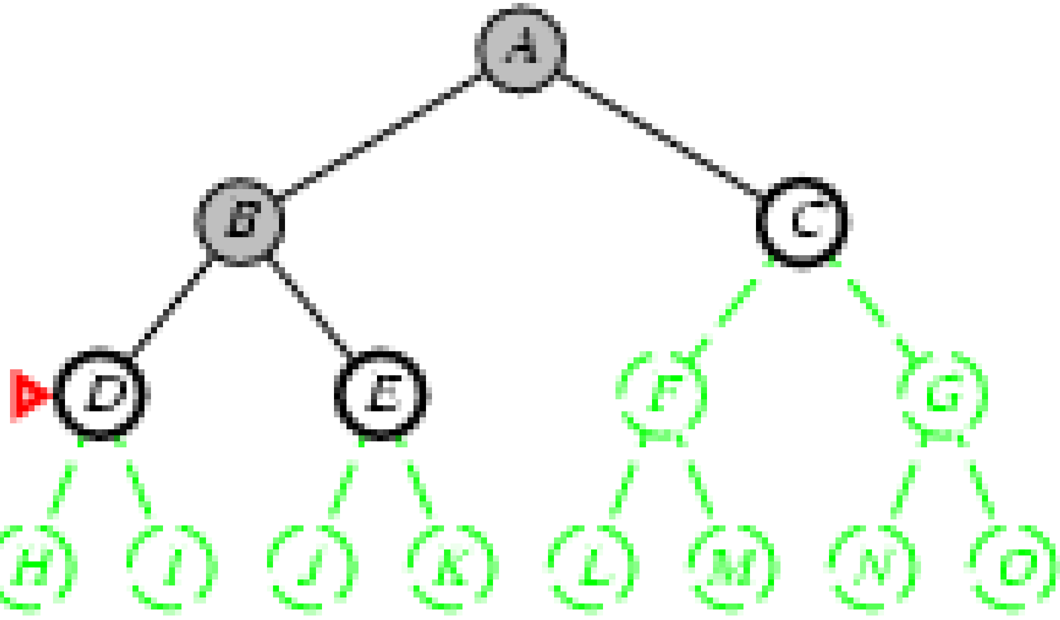


Depth-first search

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

queue=[D,E,C]

Is D = goal state?

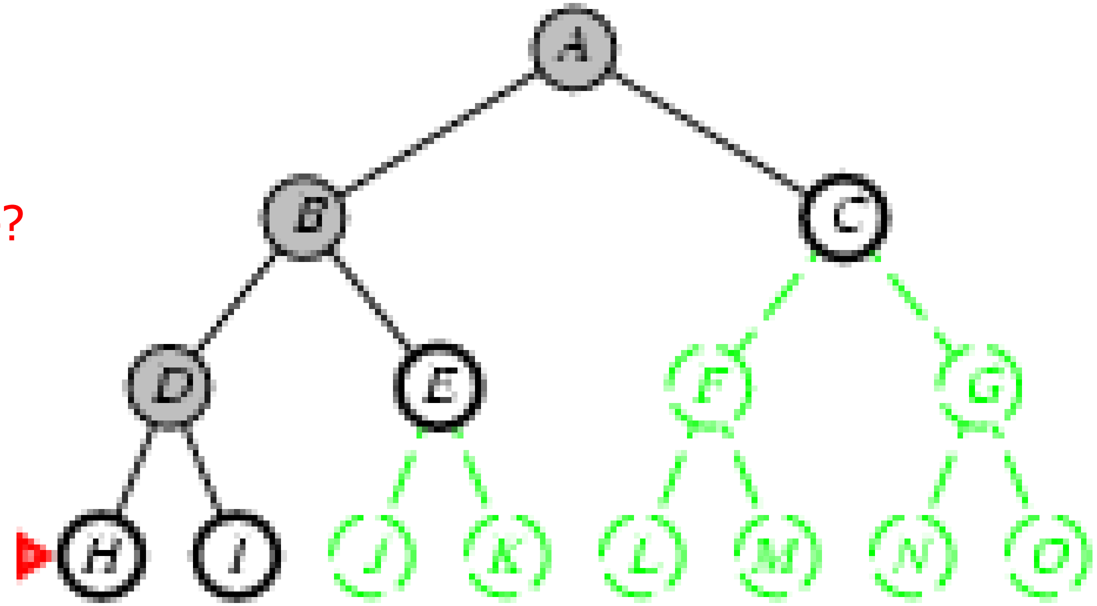


Depth-first search

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

queue=[H,I,E,C]

Is H = goal state?

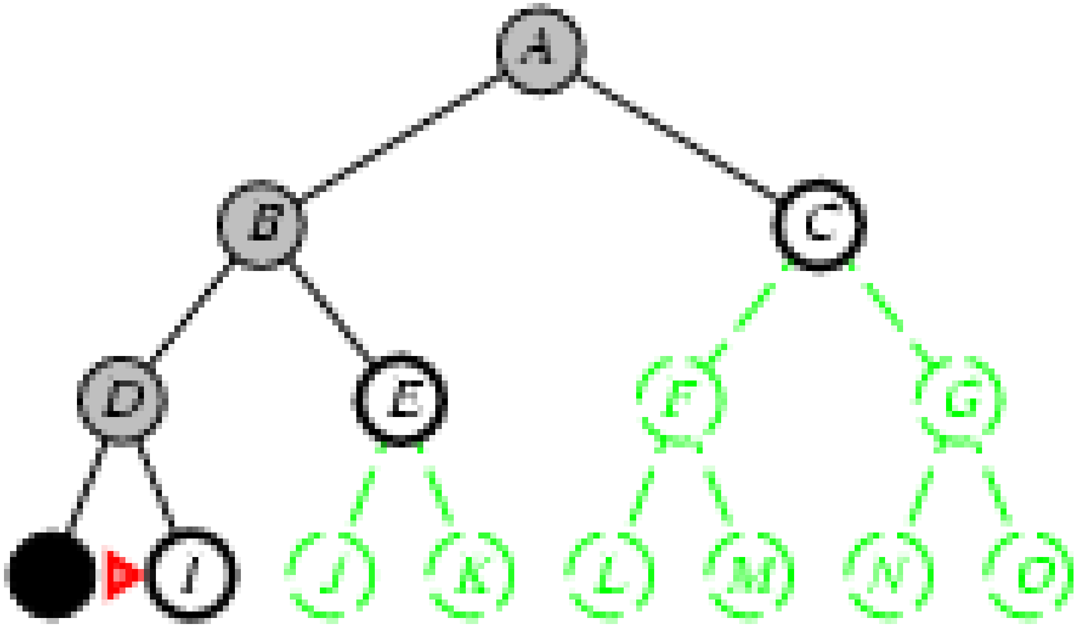


Depth-first search

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

queue=[I,E,C]

Is I = goal state?

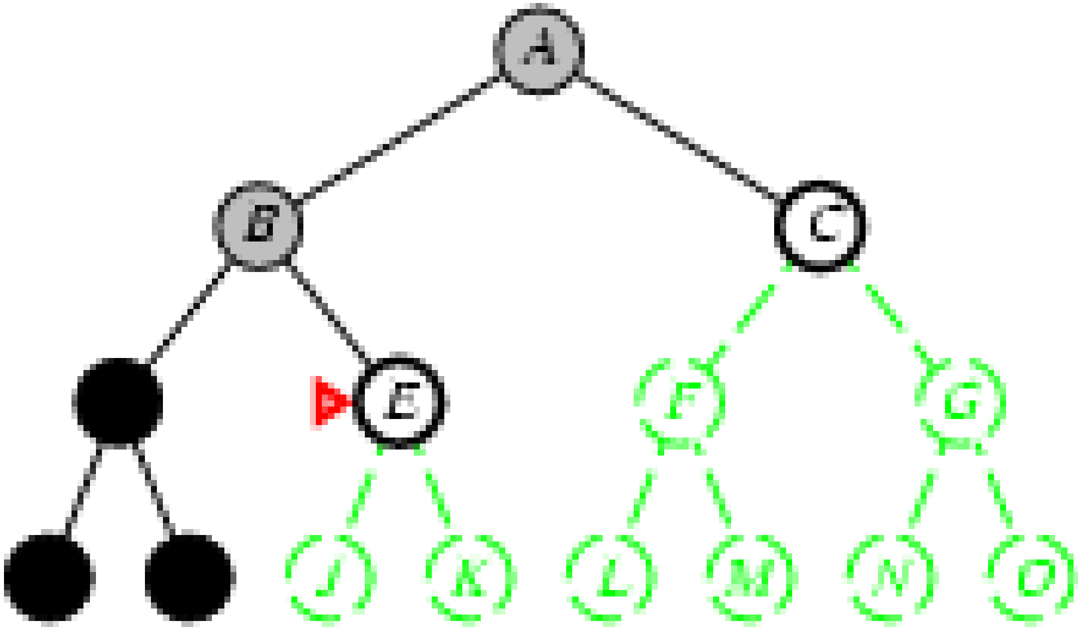


Depth-first search

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

queue=[E,C]

Is E = goal state?

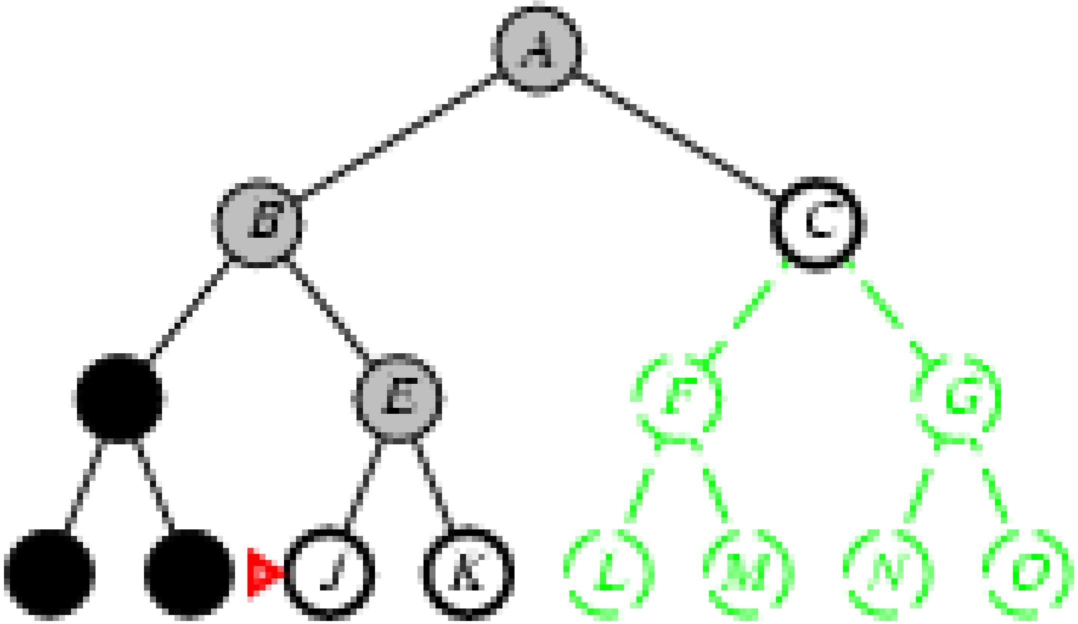


Depth-first search

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

queue=[J,K,C]

Is J = goal state?

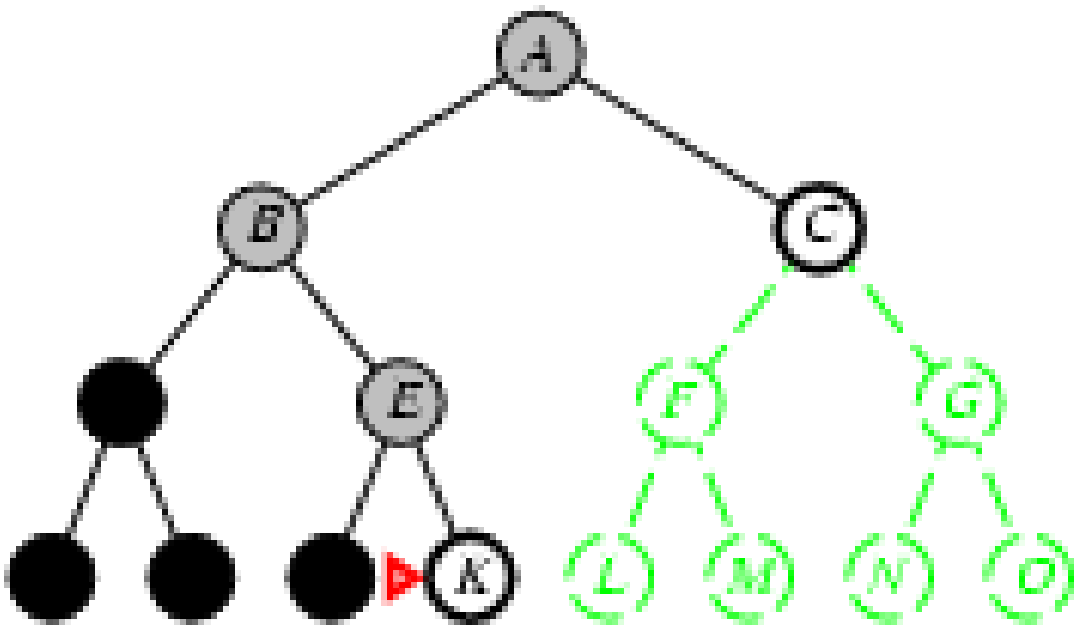


Depth-first search

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

queue=[K,C]

Is K = goal state?

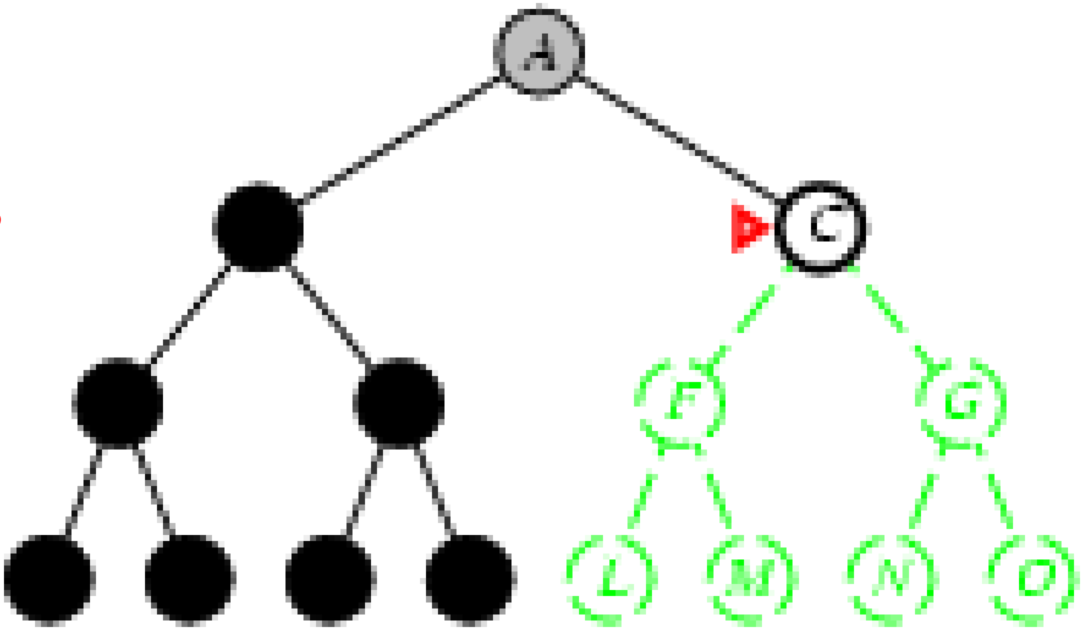


Depth-first search

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

queue=[C]

Is C = goal state?

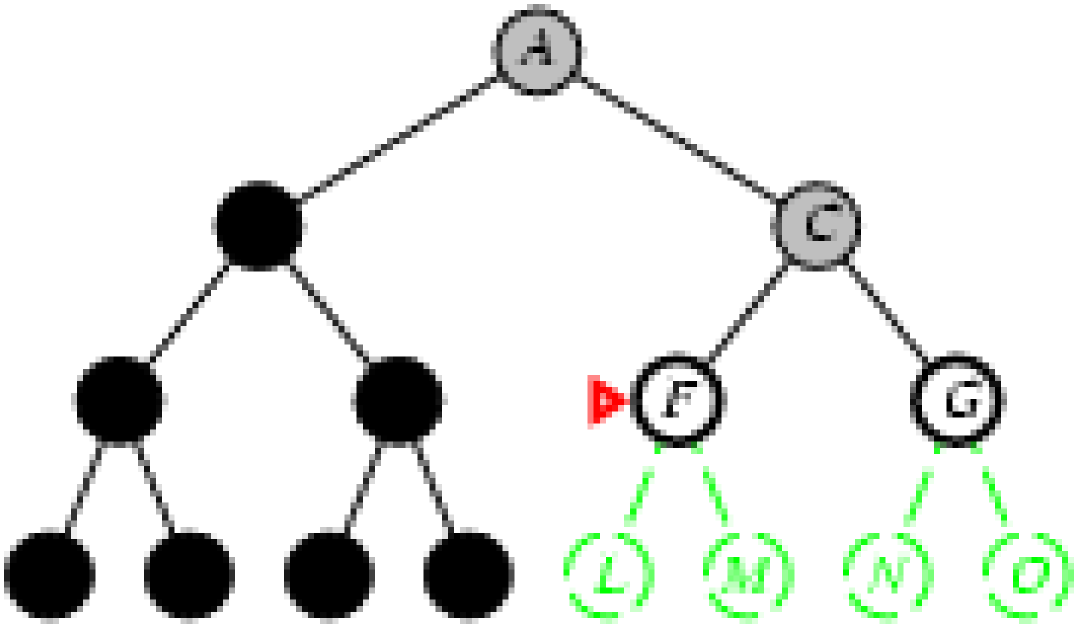


Depth-first search

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

queue=[F,G]

Is F = goal state?

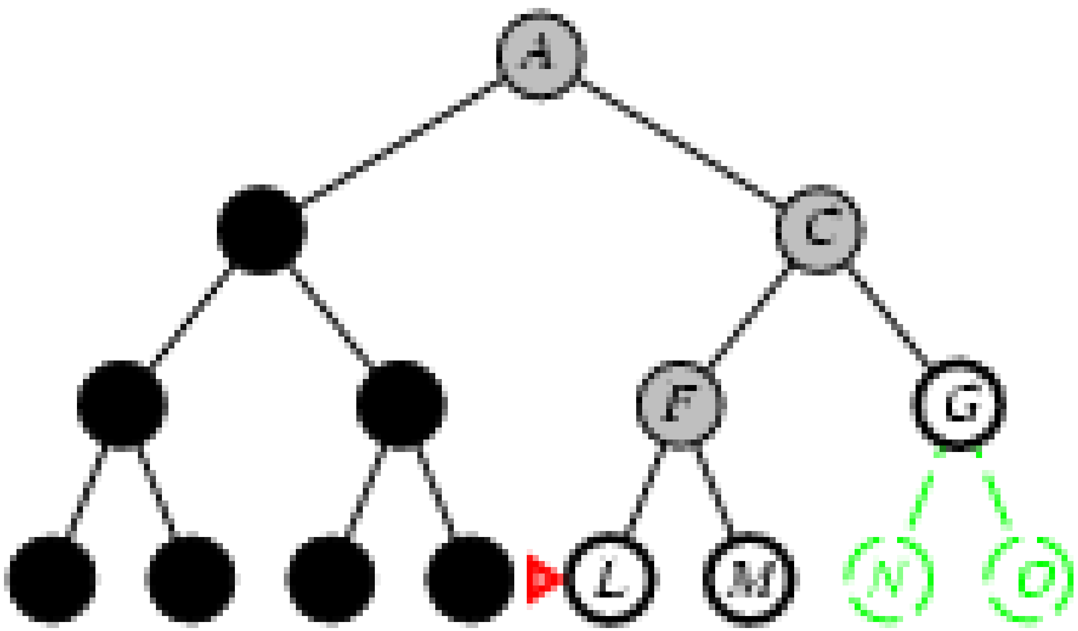


Depth-first search

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

queue=[L,M,G]

Is L = goal state?

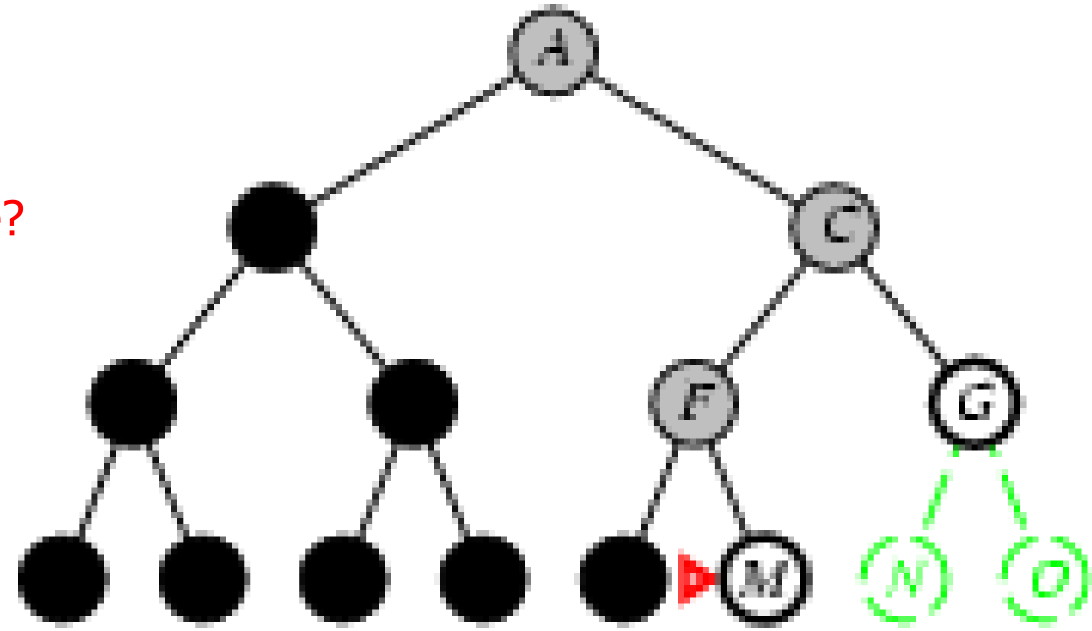


Depth-first search

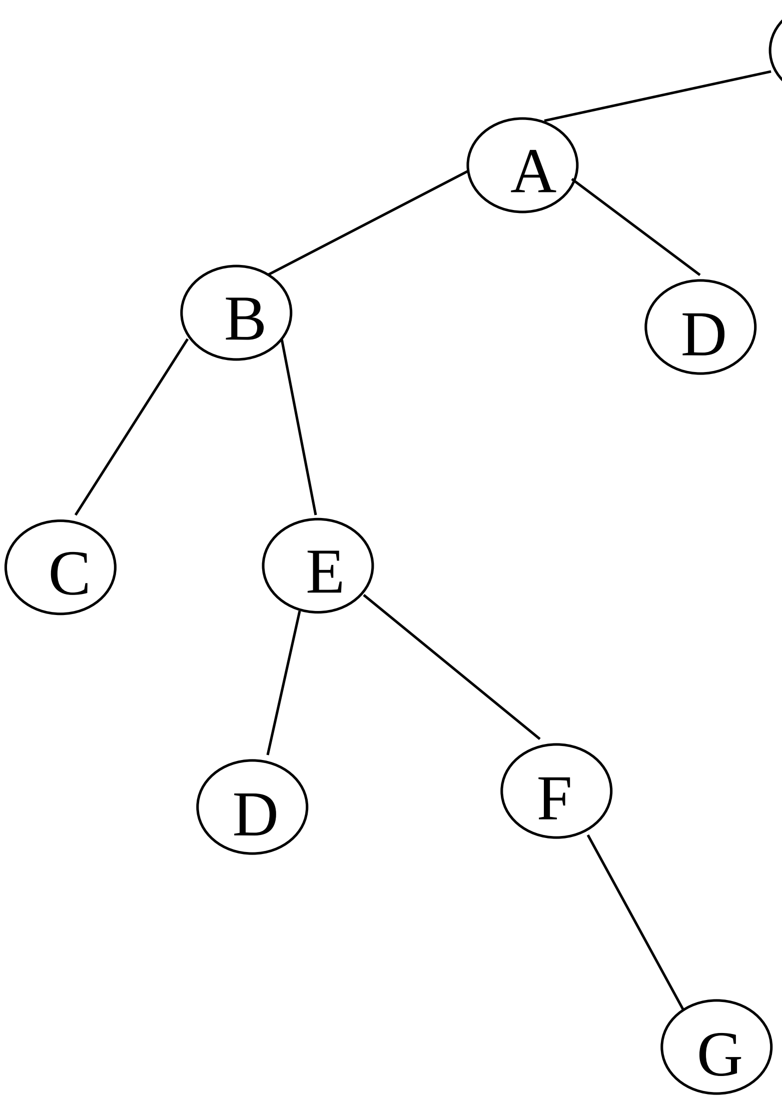
- Expand deepest unexpanded node
- Implementation:
  - *frontier* = LIFO queue, i.e., put successors at front

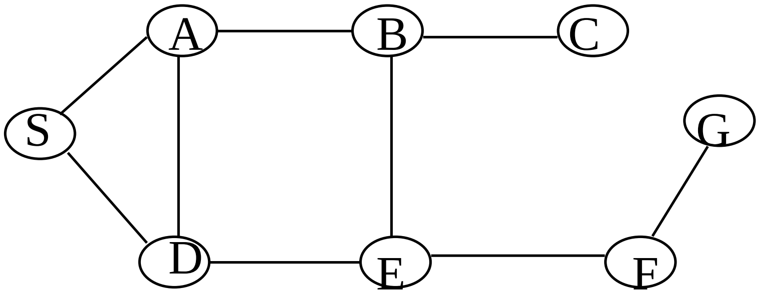
queue=[M,G]

Is M = goal state?



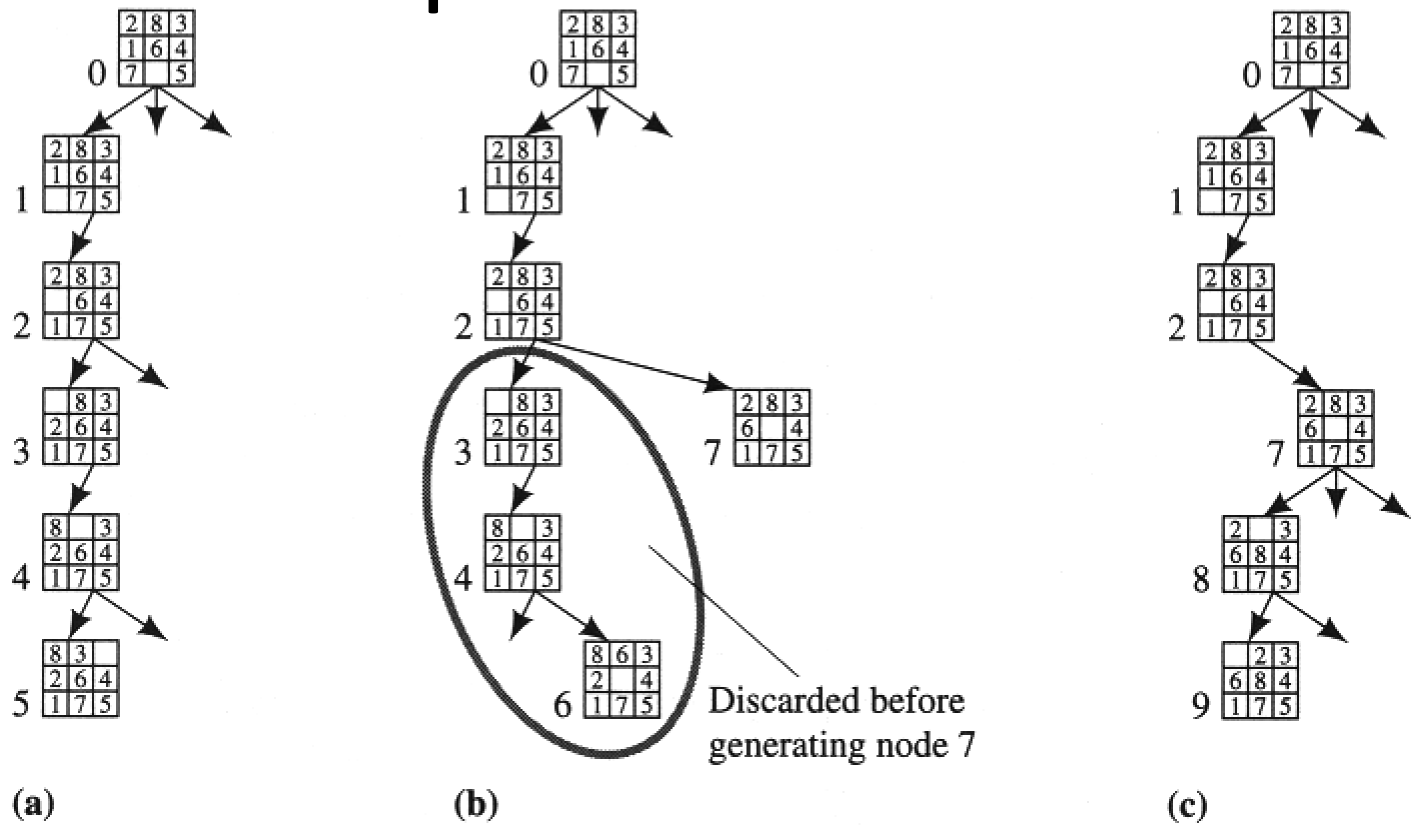
Depth-First Search (DFS)



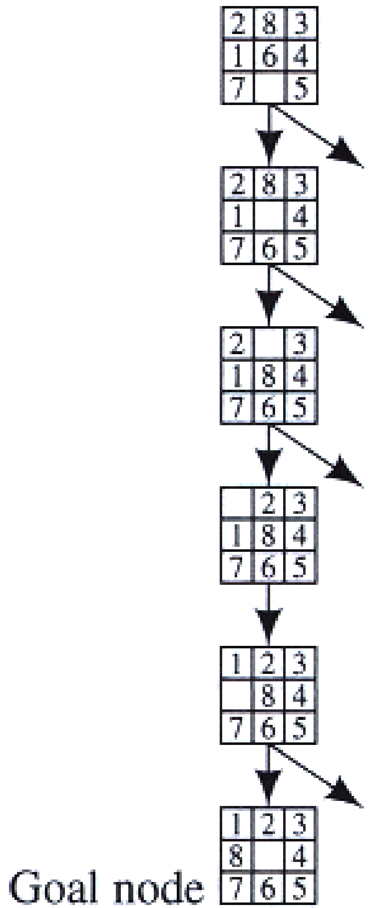


Here, (if tree-search) then to avoid infinite depth (in case of finite state-space graph) assume we don't expand any child node which appears already in the path from the root S to the parent. (Again, one could use other strategies)

Depth-First Search



Generation of the First Few Nodes in a Depth–First Search



The Graph When the Goal Is Reached in Depth–First Search

Depth-First-Search (*)

1. Put the start node *s* on OPEN
2. If OPEN is empty exit with failure.
3. Remove the first node *n* from OPEN.
4. If *n* is a goal node, exit successfully with the solution obtained by tracing back pointers from *n* to *s*.
5. Otherwise, <u>expand *n*</u>, generating all its successors (check for self-loops)attach to them pointers back to *n*, and put them at the top of OPEN *in some order*.
6. Go to step 2.

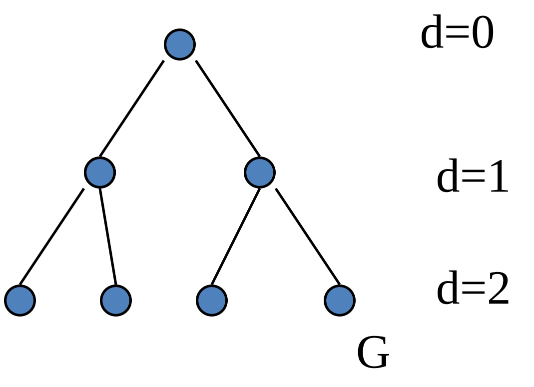
*search the tree search-space (but avoid self-loops)
** the default assumption is that DFS searches the underlying search-tree

Complexity of Depth-First Search?

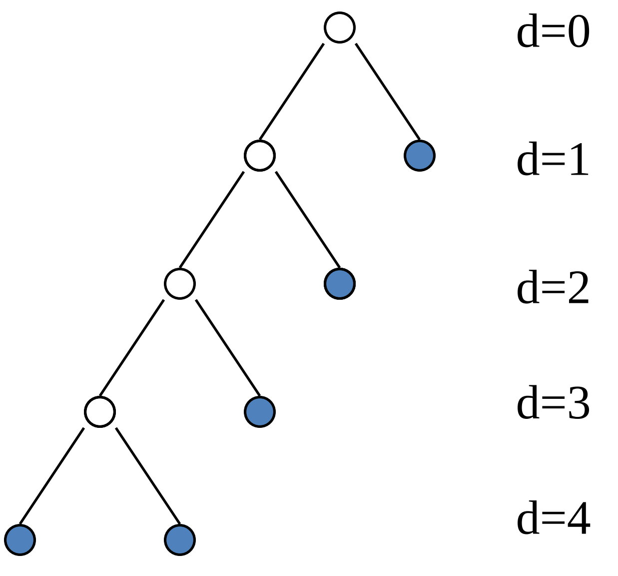
- Time Complexity
  - assume d is deepest path in the search space
  - assume (worst case) that there is 1 goal leaf at the RHS
  - so DFS will expand all nodes

  $=1 + b + b^2 + \quad .... + b^d$

  $= O\ (b^d)$

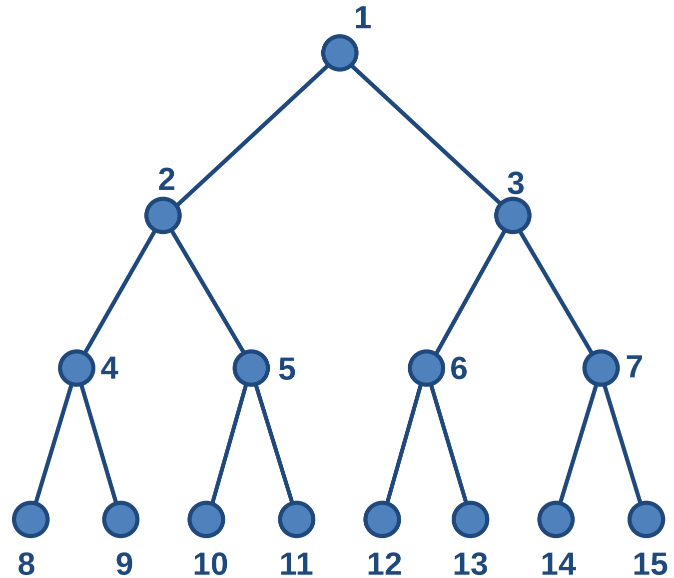


- Space Complexity (for tree-search)
  - how many nodes can be in the queue (worst-case)?
  - **O(bd) if deepest node at depth d**

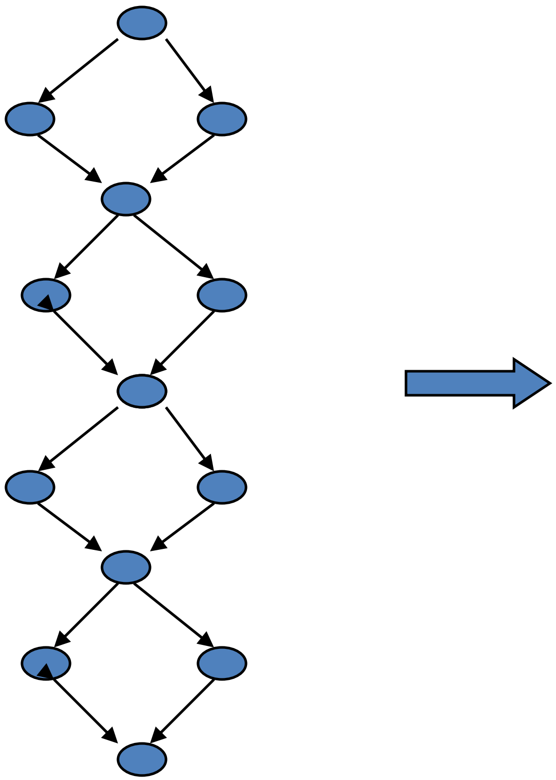


Breadth-First Search (BFS) Properties

- Solution Length: optimal
- Expand each node once (can check for duplicates, performs graph-search)
- Search Time: $O(b^d)$
- Memory Required: $O(b^d)$
- Drawback: requires exponential space



Example, Diamond Networks graph-search vs tree-search (BFS vs DFS)



- Graph-search & BFS
- Tree-search & DFS

Depth-First tree-search Properties

- Non-optimal solution path
- Incomplete unless there is a depth bound
- (we will assume depth-limited DF-search)
- Re-expansion of nodes (when the state-space is a graph)
- Exponential time
- Linear space (for tree-search)

Comparing DFS and BFS

- BFS optimal, DFS is not
- Time Complexity worse-case is the same, but
  - In the worst-case BFS is always better than DFS
  - Sometime, on the average DFS is better if:
    - many goals, no loops and no infinite paths
- BFS is much worse memory-wise
    - DFS can be linear space
    - BFS may store the whole search space.
- In general
    - BFS is better if goal is not deep, if long paths, if many loops, if small search space
    - DFS is better if many goals, not many loops
    - DFS is much better in terms of memory

Iterative-Deepening Search (DFS)

- Every iteration is a DFS with a depth cutoff.

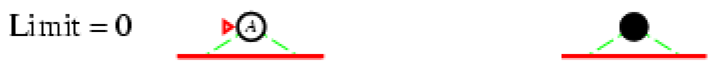
Iterative deepening (ID)

1. i = 1
2. While no solution, do
3. DFS from initial state S_0 with cutoff i
4. If found goal, stop and return solution, else, increment cutoff

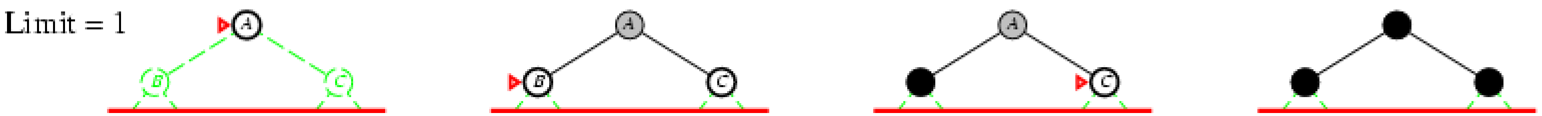
Comments:

- IDS implements BFS with DFS
- Only one path in memory
- BFS at step i may need to keep 2^{i} nodes in OPEN

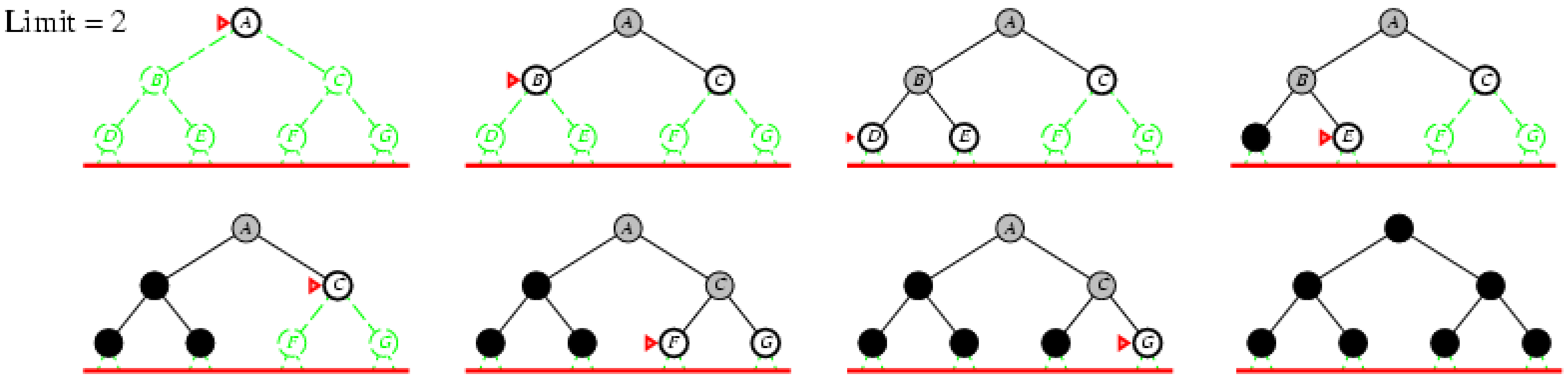
Iterative deepening search L=0



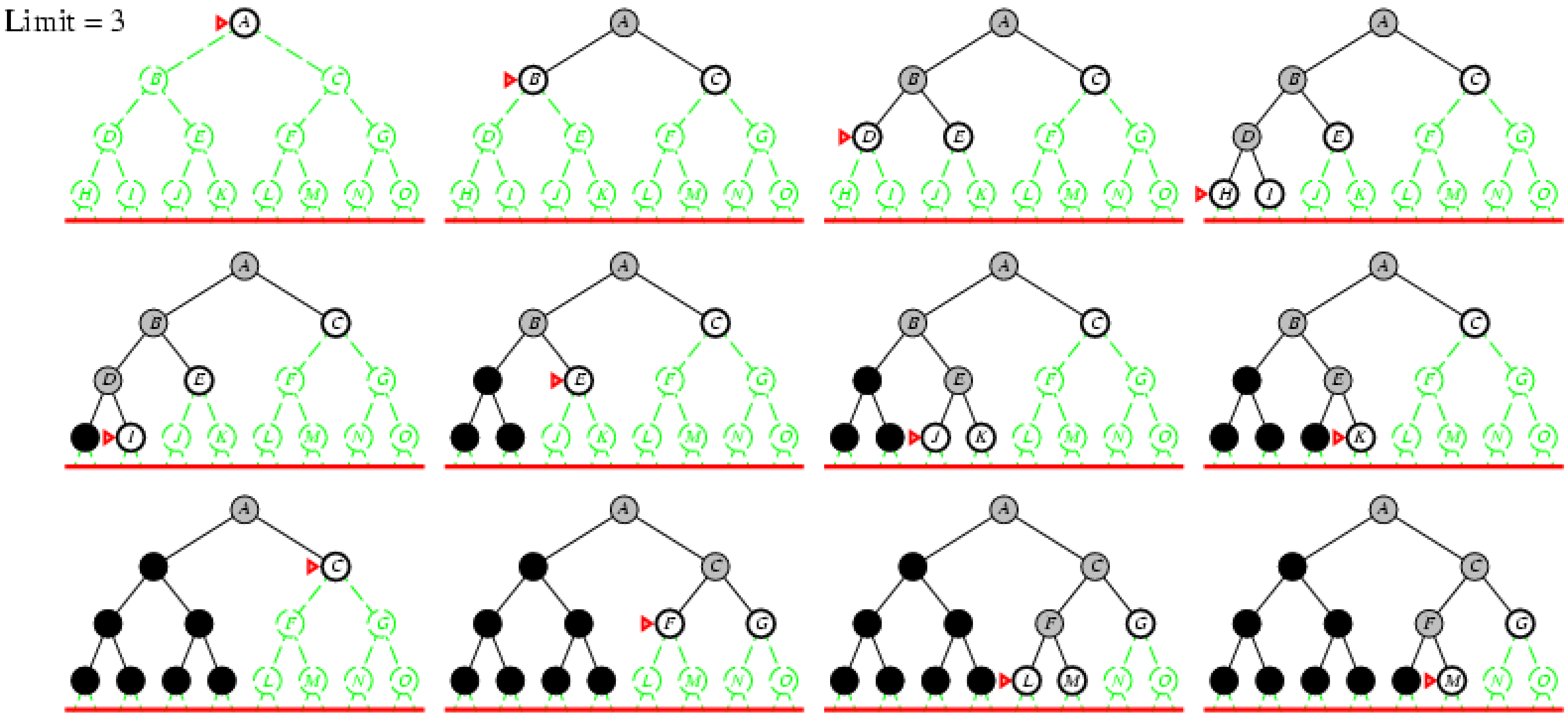
Iterative deepening search L=1



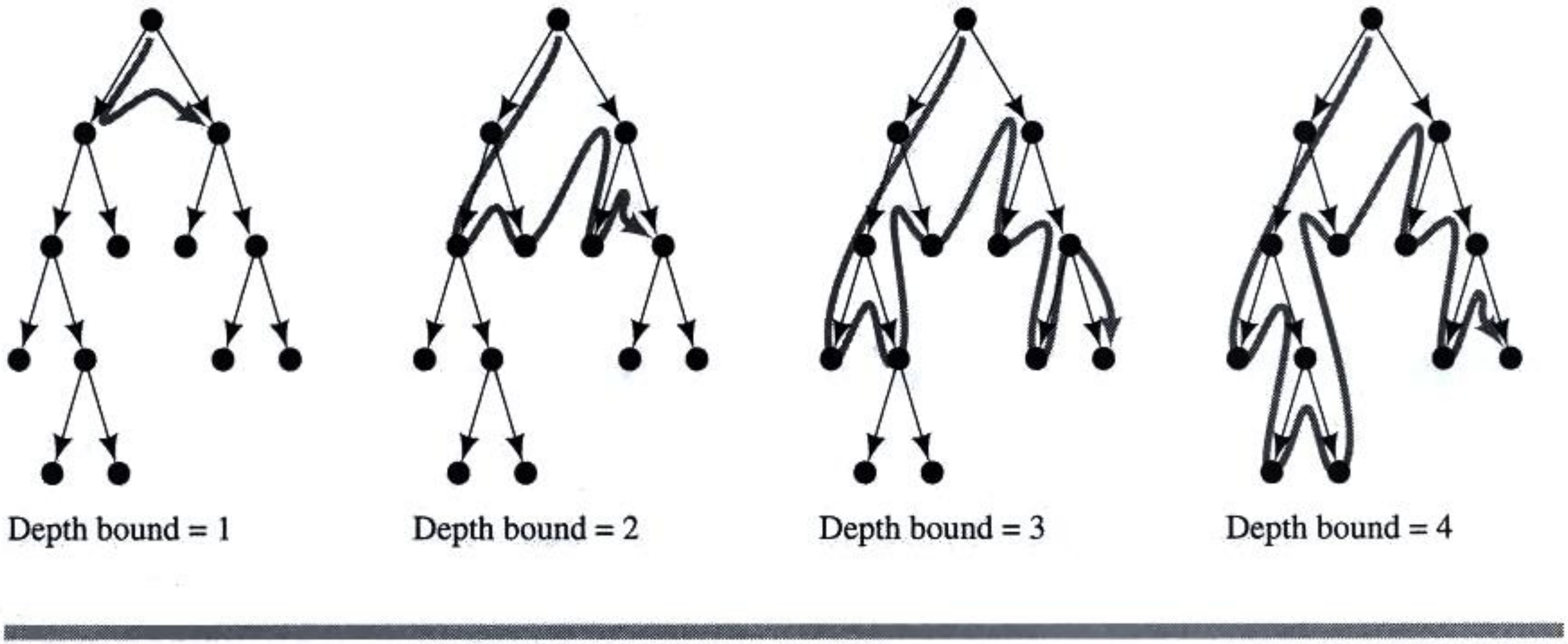
Iterative deepening search L=2



Iterative Deepening Search L=3



Iterative deepening search



Stages in Iterative–Deepening Search

Iterative Deepening (DFS)

- Time: $$T(n)=\sum_{j=1}^{n}\frac{b^{j+1}-1}{b-1}=\frac{b^{n+2}}{(b-1)^2}=O(b^n)$$

  - BFS time is $O(b^n)$, b is the branching degree
  - IDS is asymptotically like BFS,
  - For b=10 d=5 d=cut-off
  - DFS = 1+10+100,…,=111,111
  - IDS = 123,456
  - Ratio is $\frac{b}{b-1}$

Summary on IDS

- A useful practical method
  - combines
    - guarantee of finding an optimal solution if one exists (as in BFS)
    - space efficiency, O(bd) of DFS
    - But still has problems with loops like DFS

Bidirectional Search

- Idea
  - simultaneously search forward from S and backwards from G
  - stop when both “meet in the middle”
  - need to keep track of the intersection of 2 open sets of nodes
- What does searching backwards from G mean
  - need a way to specify the predecessors of G
    - this can be difficult,
    - e.g., predecessors of checkmate in chess?
  - what if there are multiple goal states?
  - what if there is only a goal test, no explicit list?
- Complexity
  - time complexity is best: $O(2\ b^{(d/2)}) = O(b^{(d/2)})$
  - memory complexity is the same

Bi-Directional Search

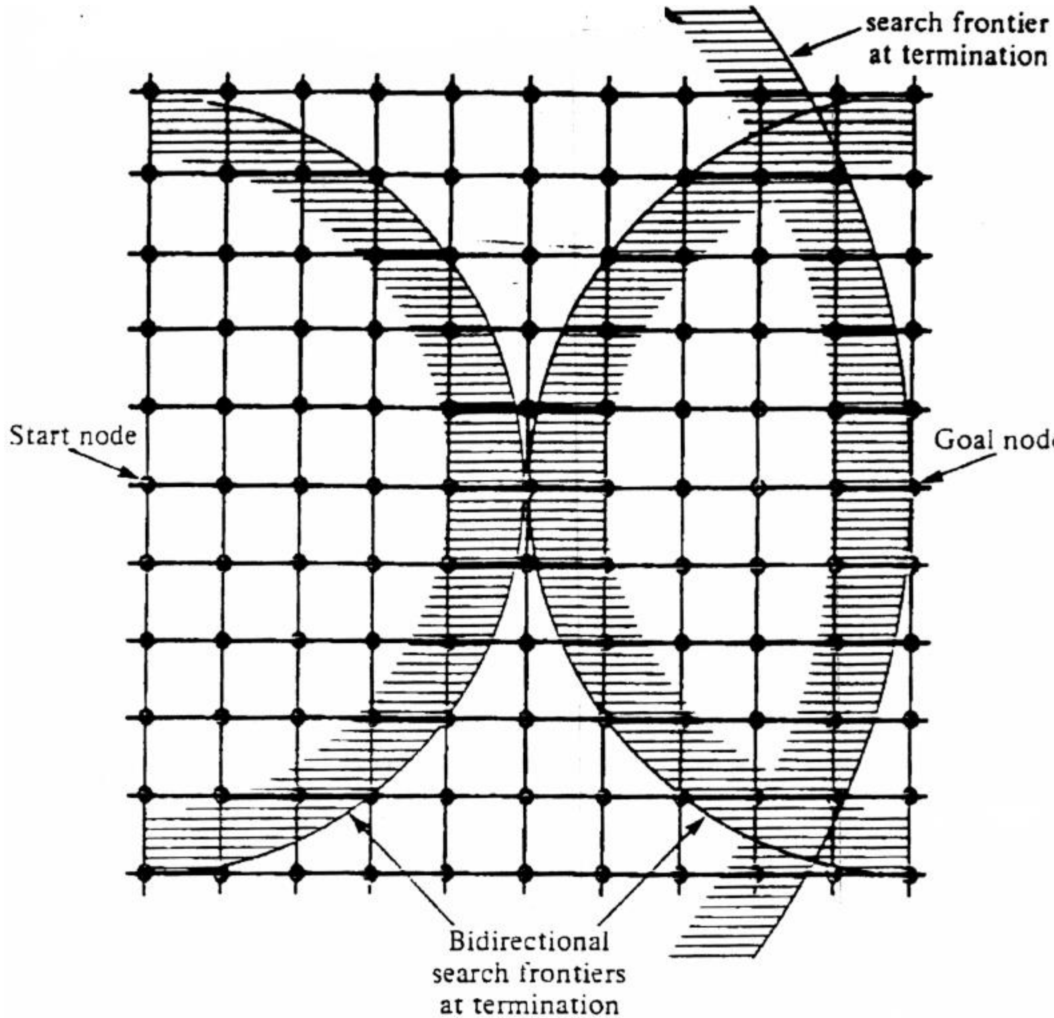


Fig. 2.10 Bidirectional and unidirectional breadth-first searches.

Comparison of Algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

Figure 3.18 Evaluation of search strategies. b is the branching factor; d is the depth of solution; m is the maximum depth of the search tree; l is the depth limit.

Summary

- A review of search
  - a search space consists of nodes and operators: it is a tree/graph

- There are various strategies for “uninformed search”
  - breadth-first
  - depth-first
  - iterative deepening
  - bidirectional search
  - Uniform cost search
  - Depth-first branch and bound

- Repeated states can lead to infinitely large search trees
  - we looked at methods for detecting repeated states

- All of the search techniques so far are “blind” in that they do not look at how far away the goal may be: next we will look at informed or heuristic search, which directly tries to minimize the distance to the goal.