Solving Problems by Searching

Reflex Agent

Lookup table

Direct mapping from states to actions

Cannot work well in some environments

- Mapping
  - Too large to store
  - Too long to learn

Goal-based agent is better

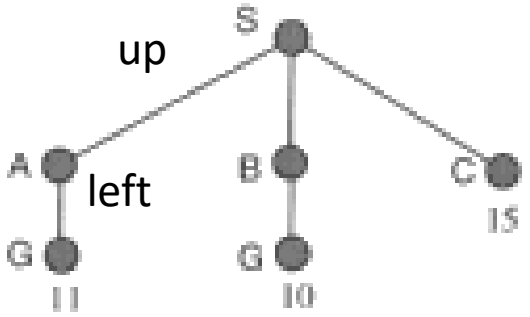
Percept sequence	Action
$[A, Clean]$	*Right*
$[A, Dirty]$	*Suck*
$[B, Clean]$	*Left*
$[B, Dirty]$	*Suck*
$[A, Clean], [A, Clean]$	*Right*
$[A, Clean], [A, Dirty]$	*Suck*
⋮	⋮
$[A, Clean], [A, Clean], [A, Clean]$	*Right*
$[A, Clean], [A, Clean], [A, Dirty]$	*Suck*
⋮	⋮

Problem-solving Agent

A kind of goal-based agent

Solves problem by finding

- Sequences of actions
- Lead to desirable states (goals)



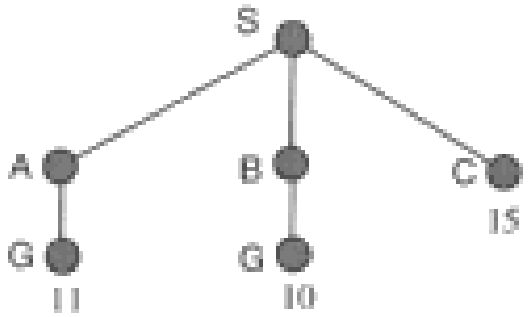
Problem-solving Agent – Goal Formulation

First step is **goal formulation**

- Based on current situation
- i.e. Define or describe a goal / objective

Goal is formulated

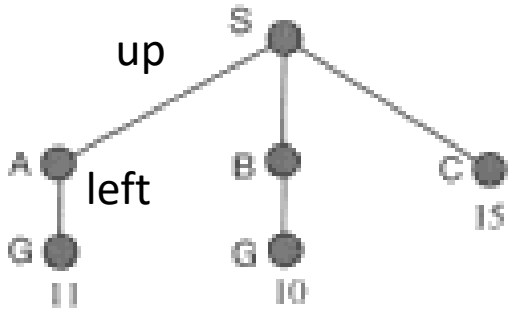
- As a set of world states
  - Goal is satisfied
- i.e. Goal states.



Problem-solving Agent – Problem Formulation

Problem = search tree

- Search Tree = {States, Actions}
- Consider related states & actions only
  - Actions: {up, left, down, right} – usually finite
  - States: {A, B, C, D, G} – usually infinite
    - May not be stated explicitly
- Small problems
  - All states can be listed out manually
- Large problems
  - States are generated during run-time
    - New states = action(old states)



Problem-solving Agent – Action

From initial state → goal state

- **Actions** (action sequence)
- E.g. {up, left}

Actions are operators

- Transitions between world states
- Simple / abstract
  - Not very detailed
- E.g. vacuum cleaner
  - Move left VS move left 50cm

Problem-solving Agent – Search

Problem = tree, solution = path

- Search path to find solution

Many paths to achieve same goal

- Multiple options at a node
  - Agent examine different possible sequences of actions
  - Choose the best
- Process of looking for the best sequence
  - **Search**
- Best sequence
  - A list of actions, called **solution**
  - i.e. = { down, right }

Search Algorithm

Given a search problem and search goal
- States + Actions + Goal
- Need a search algorithm
  - Take a *problem* (states + actions)
  - Return a *solution* (a path)
    - Satisfy the goal

Once a solution is found
- Agent follows the solution
- Carries out the list of actions
  - Execution phase

Design of a PS agent
- "Formulate, search, execute"

A simple problem-solving agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT(p) returns an action
    inputs: p, a percept
    static: s, an action sequence, initially empty
            state, some description of the current world state
            g, a goal, initially null
            problem, a problem formulation
    state ← UPDATE-STATE(state, p)
    if s is empty then
        g ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, g)
        s ← SEARCH(problem)
    action ← RECOMMENDATION(s, state)
    s ← REMAINDER(s, state)
    return action
```

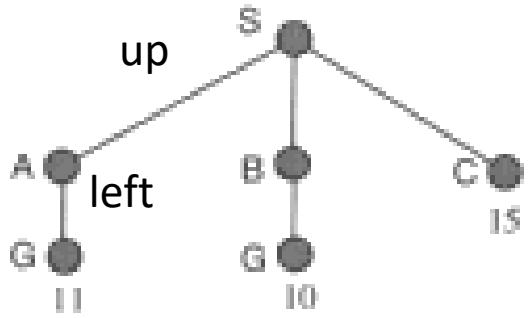
What states & actions are considered

Well-defined Problems and Solutions

Formal problem formulation

FOUR components

1. **Initial state**
   - Where agent starts
2. **Successor functions**
   - Set of possible actions



These two define the state space

- Set of all states reachable from initial state
- Whole search tree

Path in state space

- Any sequence of actions leading from one state to another

Well-defined Problems and Solutions

3. **Goal test**
   - Apply to current state
     - Check if the agent is in its goal
   - Not goal states, but test
     - Sometimes goal is described by properties
     - Not state explicitly the set of states
   - Example: Chess
     - Agent wins if captures KING of the opponent
     - No matter what opponent does

Well-defined Problems and Solutions

4. **Path cost function**
   - Check if it is the best solution
   - Assigns a numeric cost to each path
     - Performance measure
     - Denoted by *g*
   - Usually path cost is
     - Sum of step costs of individual actions (in action list)
     - i.e. Sum of costs for each action

Well-defined Problems and Solutions

Problem is defined by

1. Initial state
2. Successor functions
3. Goal test
4. Path cost function *g*

Solution of a problem

- Path from initial state to a state satisfying goal test

Optimal solution

- Solution with lowest path cost among all solutions

Abstraction

Simplification / Summarization

State representation should be concise

Take out irrelevant information

- Leave the most essential parts to the description of states
- i.e. The most important parts that contribute to searching

E.g. route finding from MACAU to USA

- Place, Method, Price, Time
- Not temperature, humidity, etc.

Example Problems

Toy problems

- Intended to illustrate or exercise various problem-solving methods
- E.g. puzzle, chess, etc.
  - Easy, fully-observable, static, deterministic

Real-world problems

- More difficult
- People care about the solutions
- E.g. Design, planning, etc.

Toy Problem – Vacuum World

Initial state: Any

Number of actions: 4

- Left, right, suck, noOp

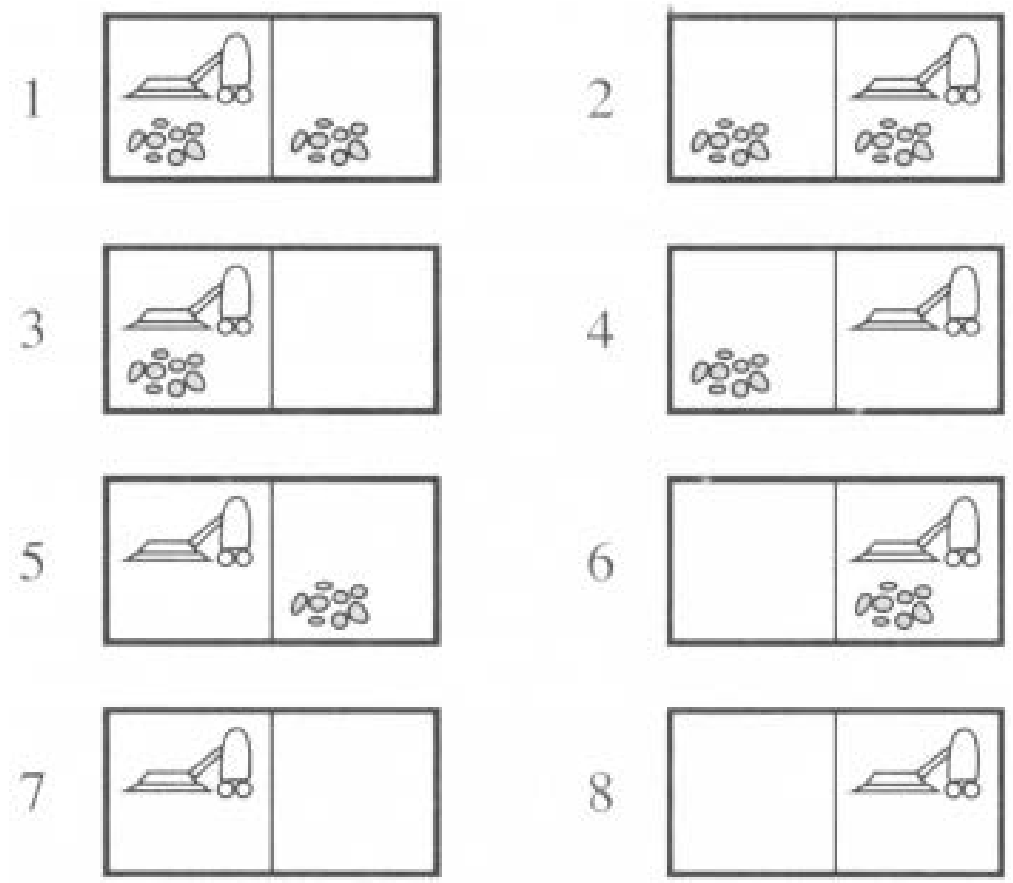
Goal: clean up all dirt

- Goal states: {7, 8}

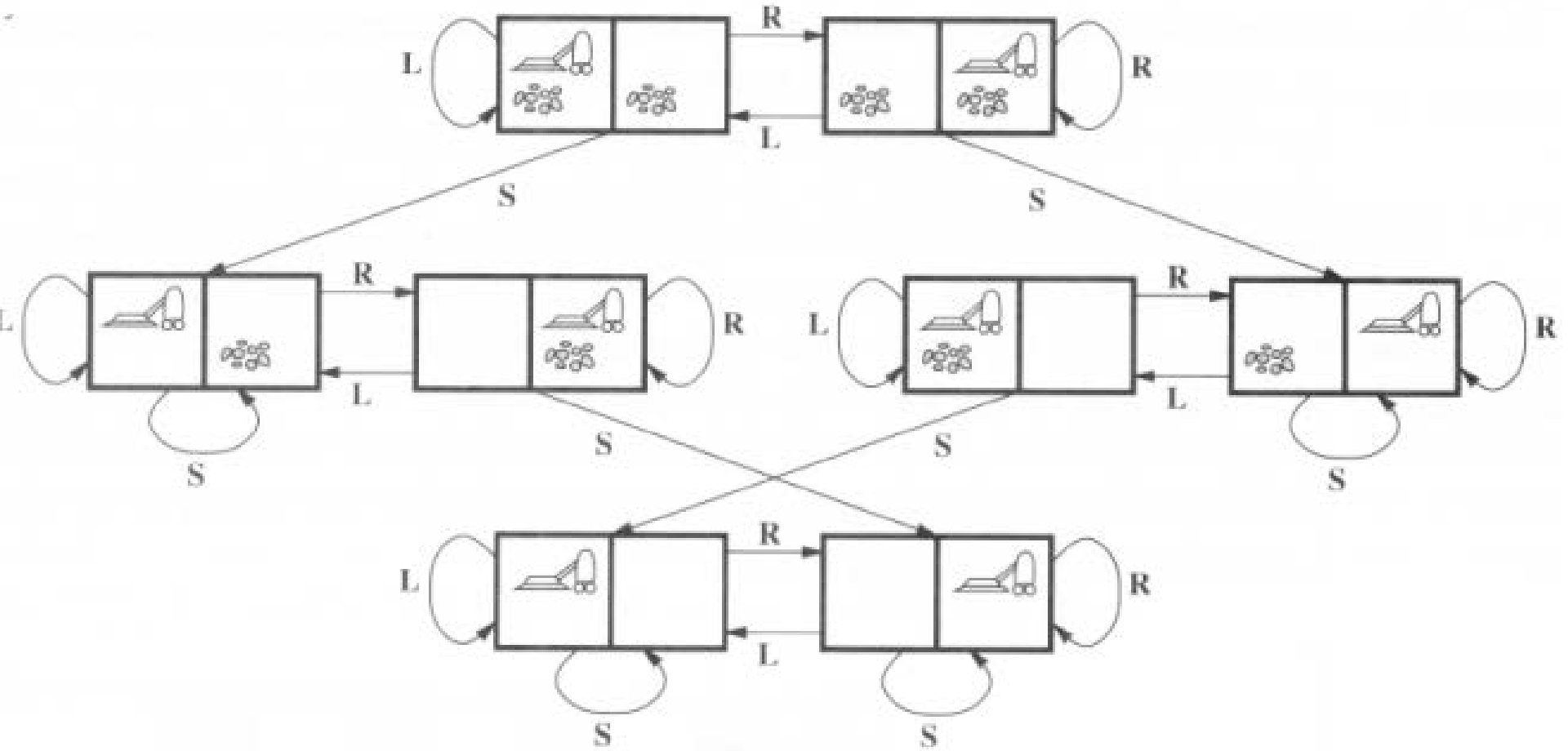
Path Cost:

- Each step costs 1

Number of states: 8



State Diagram for Vacuum Cleaner



Toy Problem – The 8-puzzle

States:

Specifies location of the eight tiles and blank in the nine squares

Initial State:

Any state in state space

Successor function:

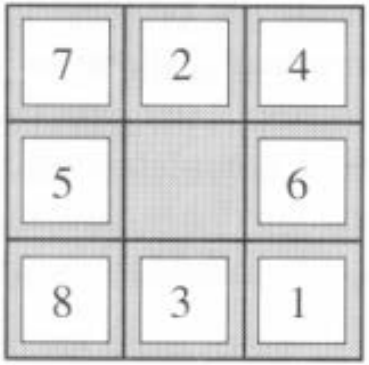
Blank moves: Left, Right, Up, or Down

Goal test:

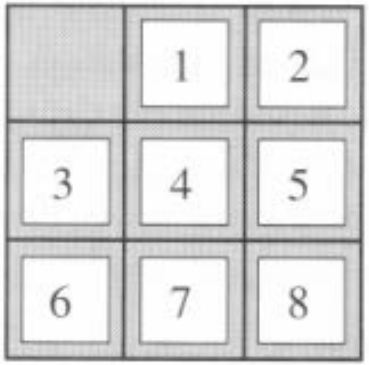
Current state matches the goal configuration

Path cost:

Each step costs 1, path cost is length of the path



Start State



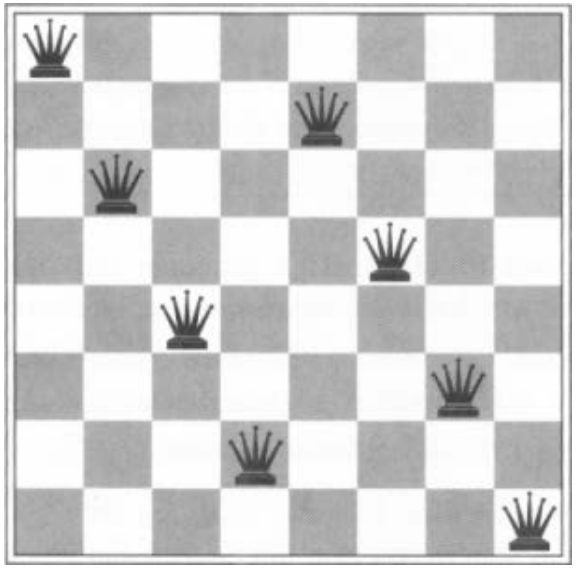
Goal State

Toy Problem – The 8-queens

Two ways to formulate the problem

Both have common properties

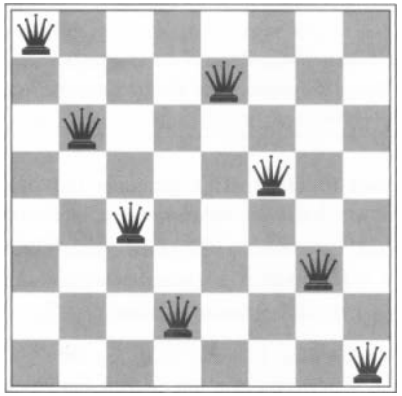
- Goal test: 8 queens on board, not attacking to each other
- Path cost: zero



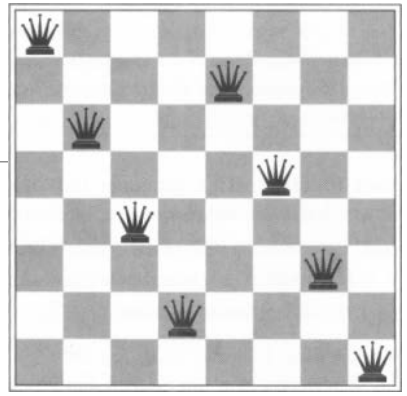
Toy Problem – The 8-queens

(1) Incremental formulation

- Involves operators that modify the state description
  - Start from an empty state
  - Each operator adds a queen to the state
- States:
  - Any arrangement of 0 to 8 queens on board
  - Initial state: an empty board (0 queens)
- Successor function:
  - Add a queen to any empty square
- State space:
  - 64*63*62*61*60*59*58*57 different states



Toy Problem – The 8-queens



(2) Complete-state formulation

- Starts with all 8 queens on board
- Move the queens individually around
- States:
  - Any arrangement of 8 queens, one per row in leftmost columns
- Operators:
  - Move an attacked queen to another column, not attacked by others
  - At most 7! different states

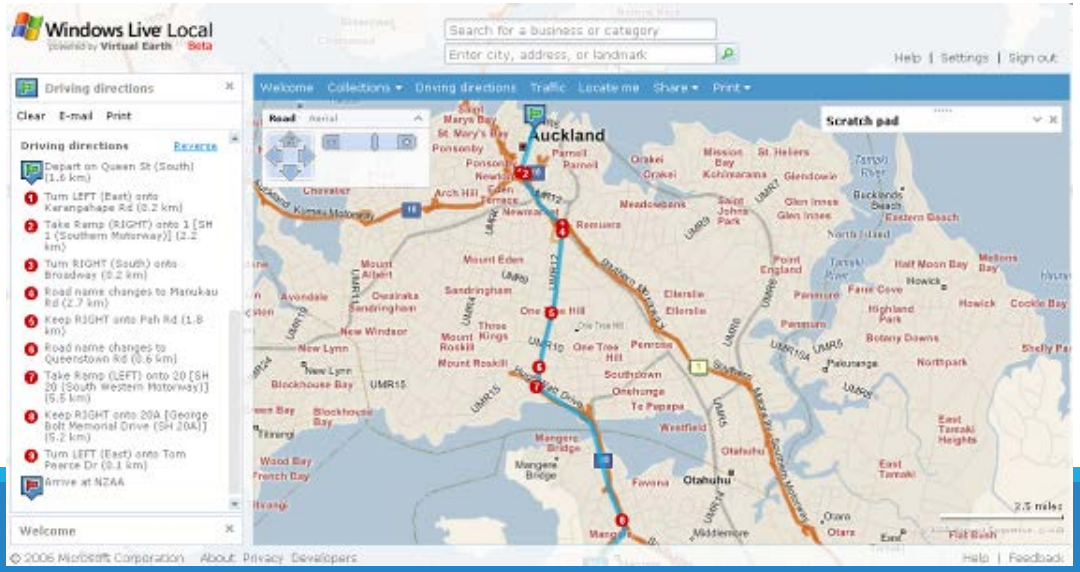
Conclusion

- Correct formulation makes a big difference to the size of search space

Real-world Problems

Route finding

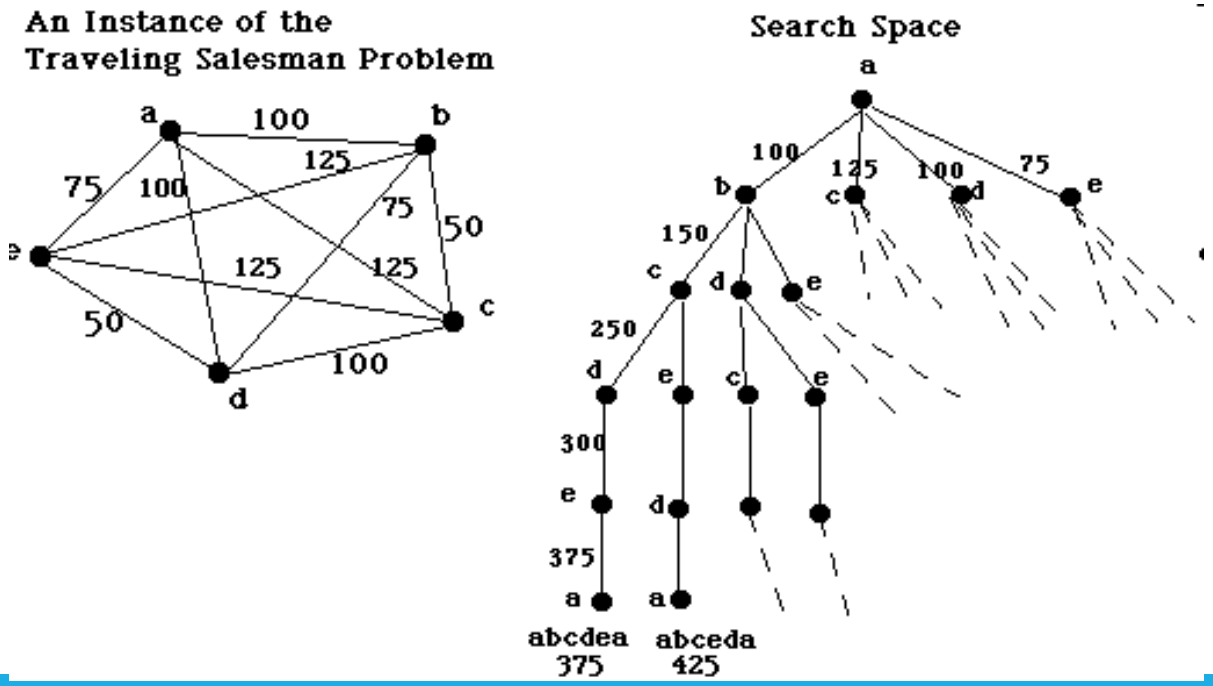
- Find out a path from source to destination
- For example
  - Route in computer networking
  - Military operations planning
  - Airline travelling with seat quality, time of day, type of airplane, etc



Real-world Problems

Touring and traveling salesperson problems (TSP)

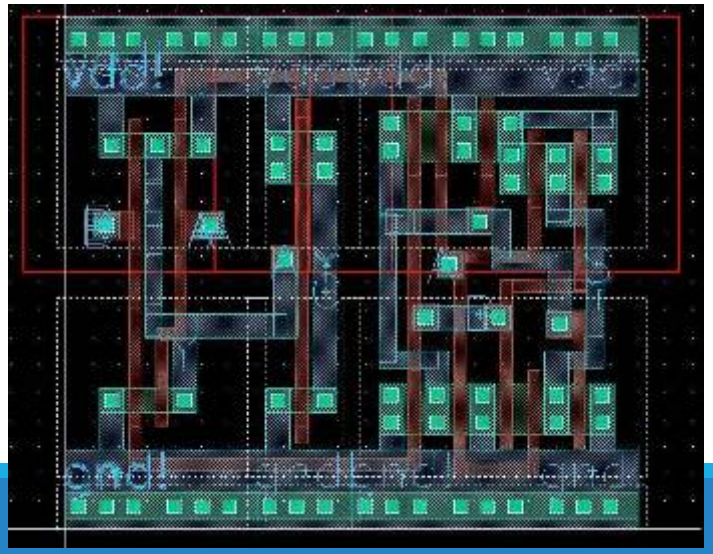
- Visit every city in a map at least (or even exactly) once
- Start and end at the same place



Real-world Problems

VLSI layout

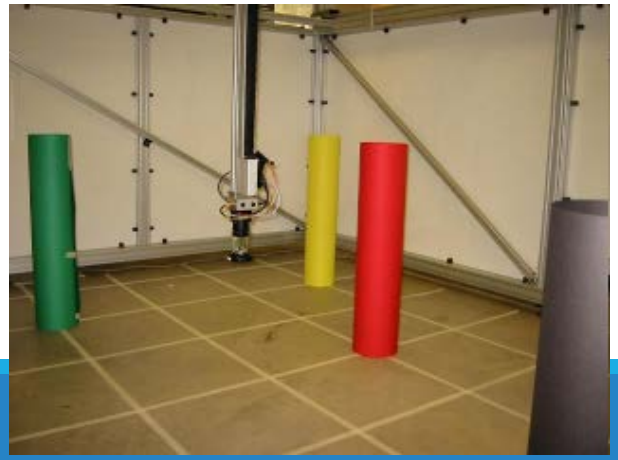
- Find out connections
  - Among gates on a chip
  - Do not overlap
  - Minimize area and connection lengths
- Speed is maximized and cost is minimized

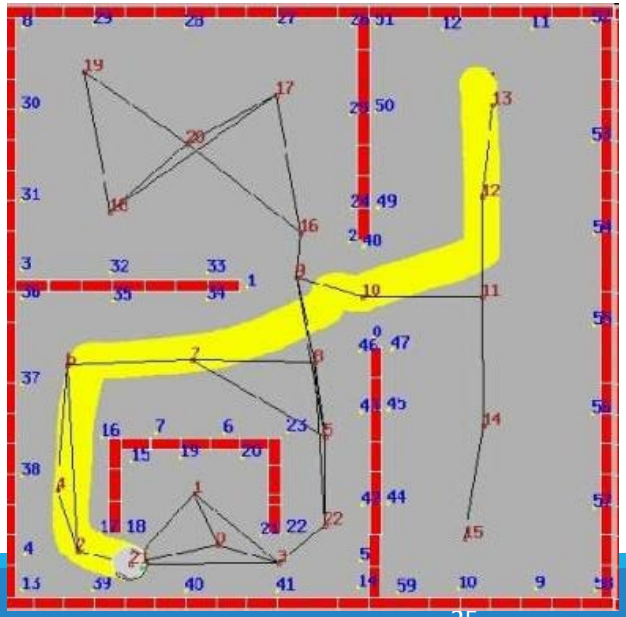


Real-world Problems

Robot navigation

- Generalization of route-finding problem
- Not a discrete set of routes
- In a continuous space
  - Infinite set of possible actions and states
- Search space
  - Multi-dimensional
  - Infinite

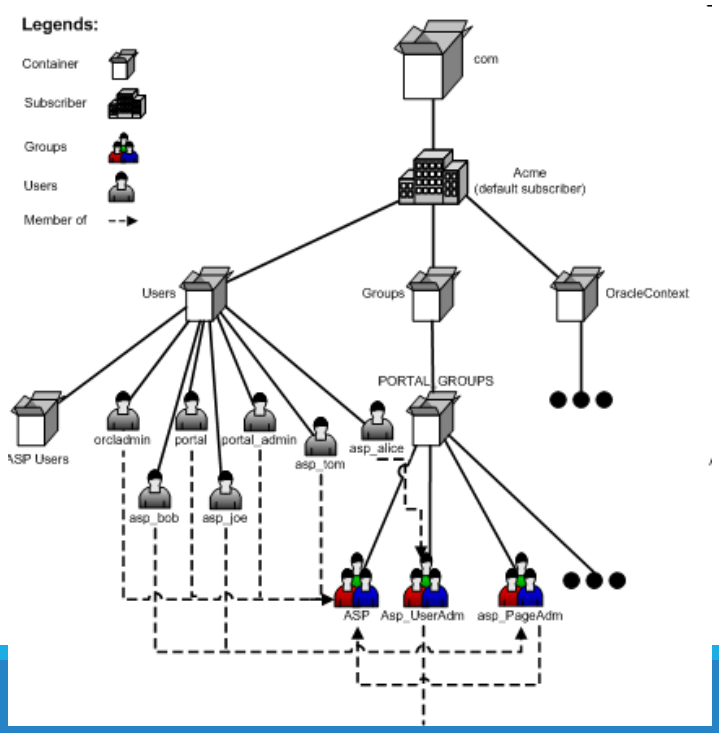




Real-world Problems

Internet searching

- Search engine using software robots
- Internet is viewed as a graph of nodes
  - Web pages → nodes
  - Hyperlinks → edges



Searching for Solutions

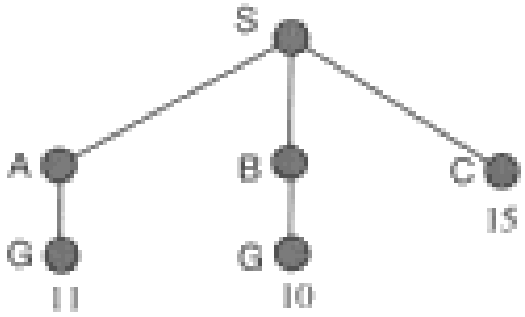
Solution is found by
- Searching through the state space

Problems are transformed as search tree
- Generated by initial state and successor function

Initial state
- Root of search tree, search node

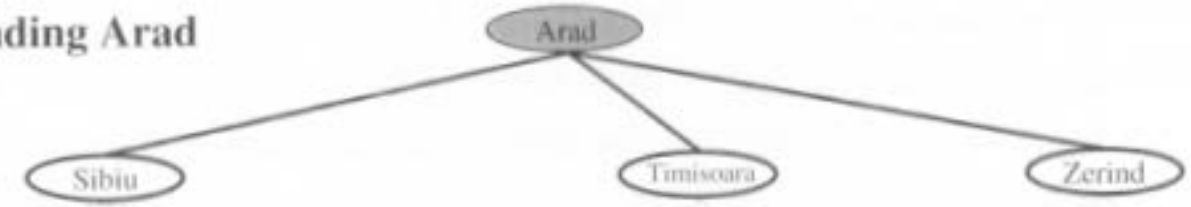
Expanding
- Apply successor function to current state
- Generate a new set of states

Leaf nodes (Terminals)
- States having no successors
- Have not yet been expanded (fringe)



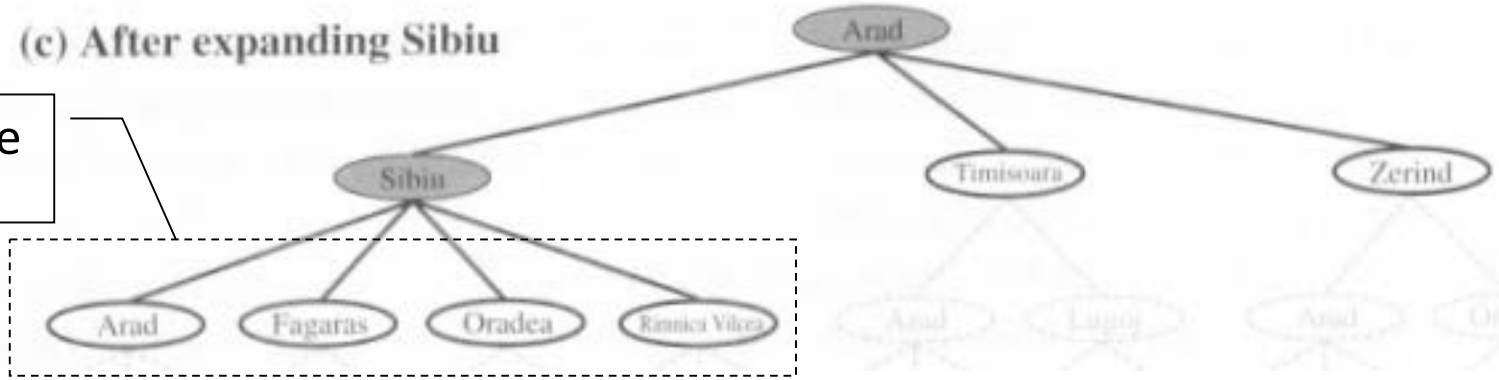
(a) The initial state

(b) After expanding Arad



(c) After expanding Sibiu

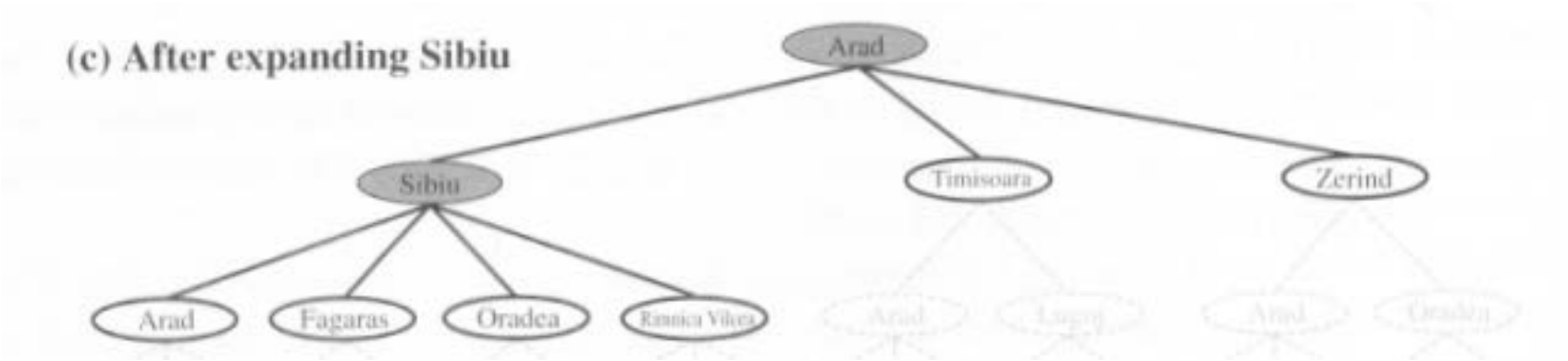
fringe



Search Strategy

Essence of searching

- In case first state is not correct
- Choosing one option
- Keep others for later inspection



Search Strategy

Determine the state to expand firstly

Good choice → fewer work → faster

Important:

- State space ≠ Search tree

State space

- Set of unique states {A, B}

Search tree

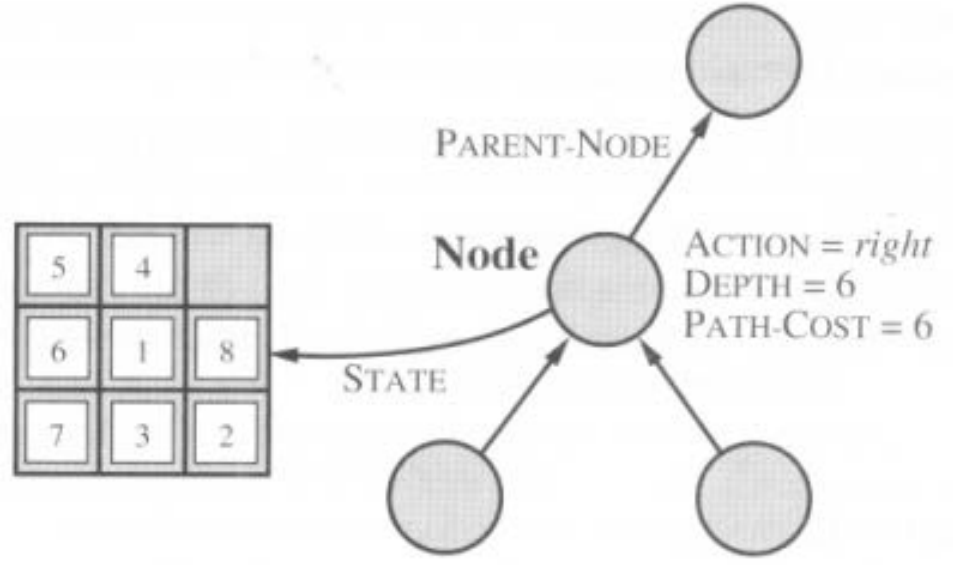
- May have cyclic paths: A-B-A-B-A-B- …

Good search strategy should avoid cyclic paths

Well-defined Node

A node should have five components

- STATE: Which state it is in state space
- PARENT-NODE: From which node it is generated
- ACTION: Which action applied to its parent-node to generate it
- PATH-COST: $\boldsymbol{g}(n)$, cost from initial state to the node n
- DEPTH: Number of steps along the path from initial state



Queue

Better organize the fringe

- Use queue to store them (FIFO)
- Many queue arrangements / operations
- Operations
  - MAKE-QUEUE*(element, …)*
  - EMPTY*(queue)*
  - FIRST*(queue)*
  - REMOVE-FIRST*(queue)*
  - INSERT*(element, queue)*
  - INSERT-ALL*(elements, queue)*

Measuring Problem-solving Performance

Many search strategies

- Evaluate / compare search strategy
  - **Completeness**
    - Guarantee in finding solution
  - **Optimality**
    - Best among many solutions
  - **Time complexity**
    - Time it takes to find a solution
  - **Space complexity**
    - Memory required in performing the search

Time & Space Complexity

Complexity

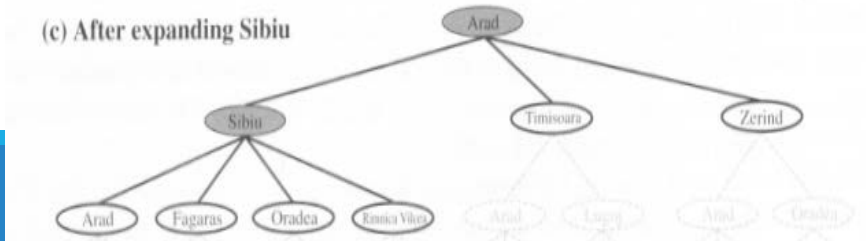
- Branching factor **b**
  - Maximum number of successors of any node
- Depth of the shallowest goal node **d**
- Maximum length of any path in state space **m**

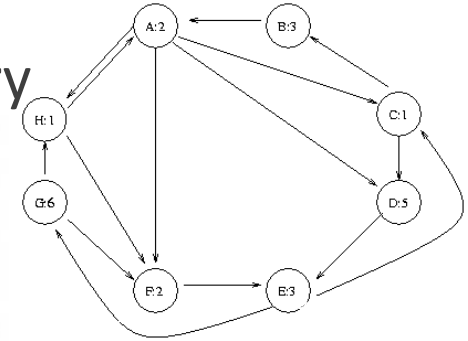
Time

- Number of nodes generated during the search

Space

- Maximum number of nodes stored in memory

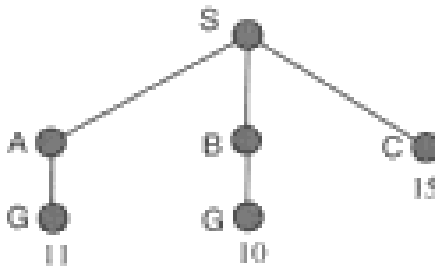




Total Cost of Algorithm

Design a search algorithm

- Consider total cost
- Total cost = path cost (*g*) + search cost
  - Search cost = time necessary to find the solution
- E.g. finding out {S, A, G} and {S, B, G}
  - Path cost = 11 VS 10
  - Time = t units VS 2t units



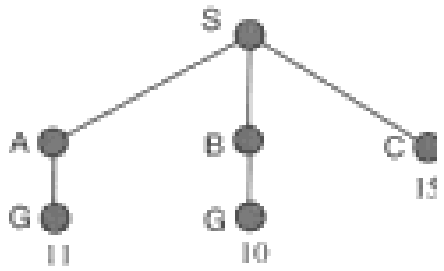
Trade -off

- Optimal solution with least *g* (path cost)
  - Requires much longer time (search cost)
- Solution with slightly lager path cost *g*
  - Requires much fewer time

Kinds of Search Strategies

Uninformed search

- Search state space *blindly*
  - Whole search is built and searched
- No information is given
  - Number of steps
  - Path cost
- No guidance towards the goal



Informed search (heuristic search)

- Cleverer strategy
  - Search towards the goal
  - Based on information from current state so far

Uninformed Search Strategies

Breadth-first search

- Variant: Uniform cost search

Depth-first search

- Variant: Depth-limited search
- Variant: Iterative deepening search

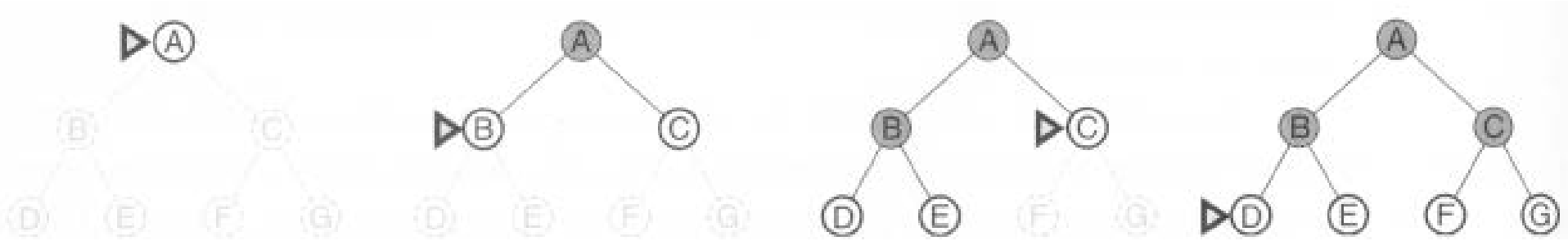
Bidirectional search

Breadth-First Search (BFS)

Root node is expanded firstly (FIFO)

All nodes generated by root node are then expanded

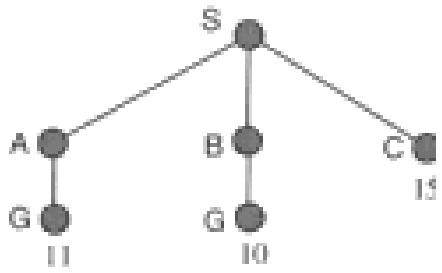
And then their successors and so on



{A} {B, C} {C, D, E} {D, E, F, G}

Evaluation of BFS

Breadth-first search



- Complete
  - Find the solution eventually
- Optimal
  - If the path cost is a non-decreasing function of depth of the node

Disadvantage

- In case of large branching factor **b**
  - Space and time complexity are enormous
    - Even small instances (e.g. chess)

Evaluation of BFS

Assume 1 million nodes can be processed per second

Each node with 1KB of storage

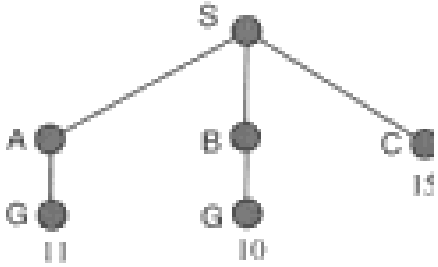
Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

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

Uniform Cost Search

Breadth-first search

- Find the shallowest goal state
- Not necessarily be the least-cost solution
- Only if all step costs are equal



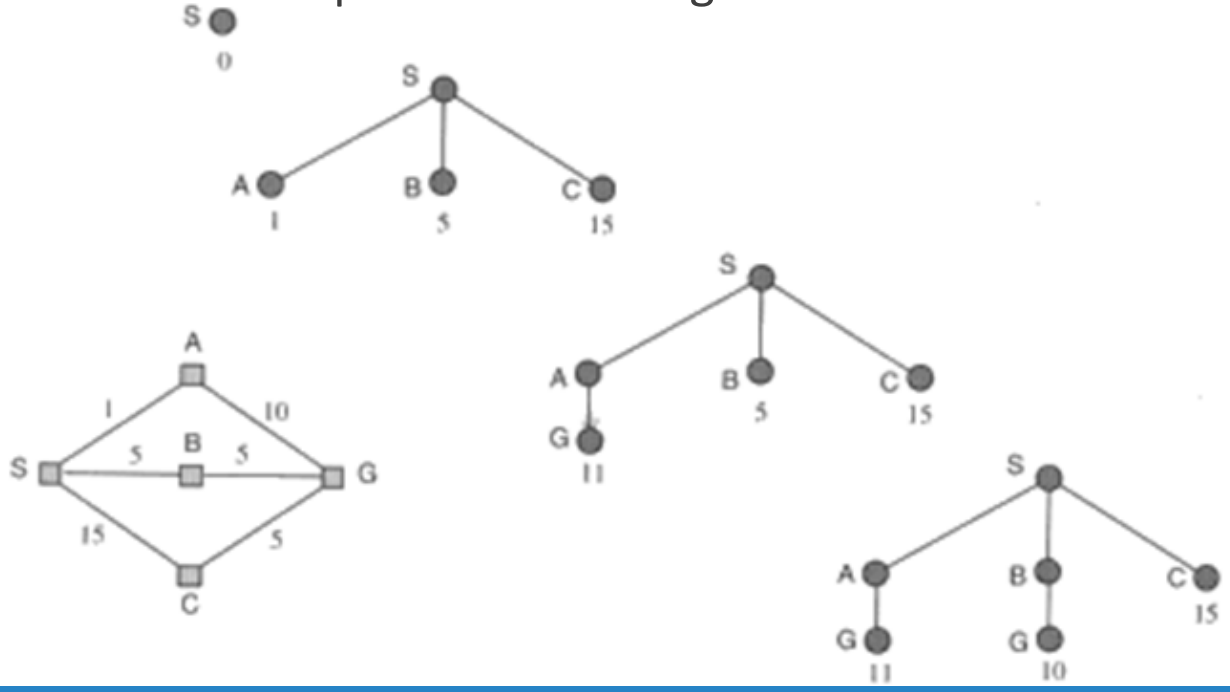
Uniform cost search

- Modify breadth-first strategy
  - Always expand the lowest-cost node
- Lowest-cost node is measured by path cost *g(n)*

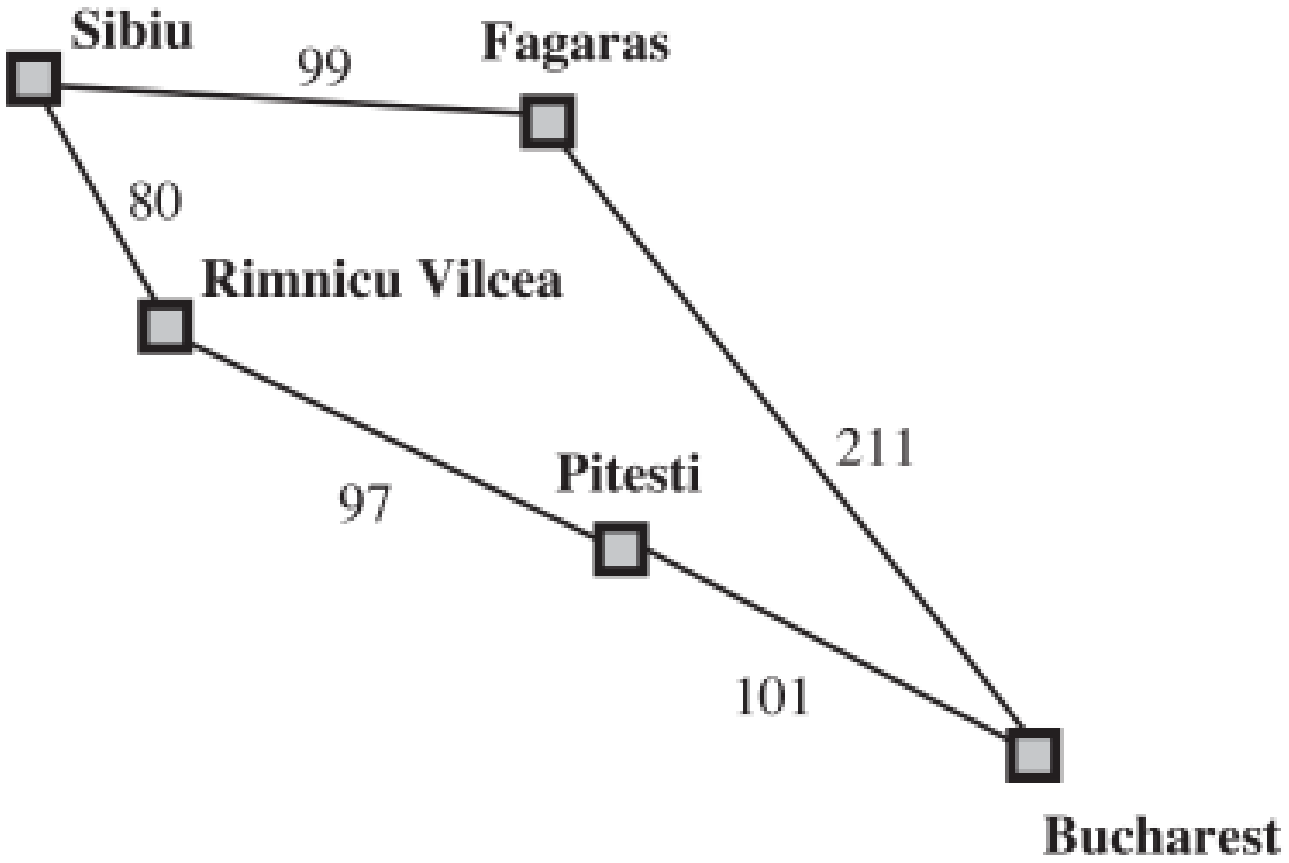
Uniform Cost Search

Guarantee the first found solution

- Cheapest
- Least in depth (the same as BFS)
- Restricted to *non-decreasing* path cost
- Unsuitable for operators with negative cost



Uniform Cost Search



Depth-First Search (DFS)

Always expand one of the nodes at the *deepest* level of the tree (LIFO)

Only when search hits a dead end

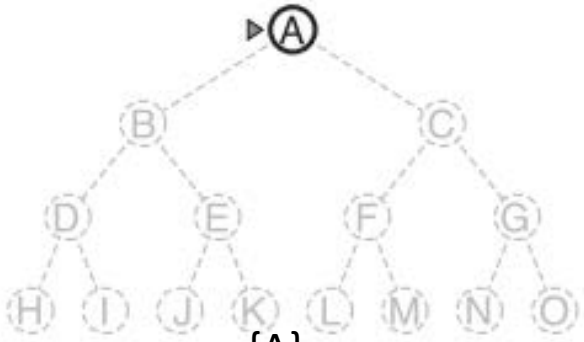
- Go back and expand nodes at shallower level
- Dead end: leaf node but not the goal

Backtracking search

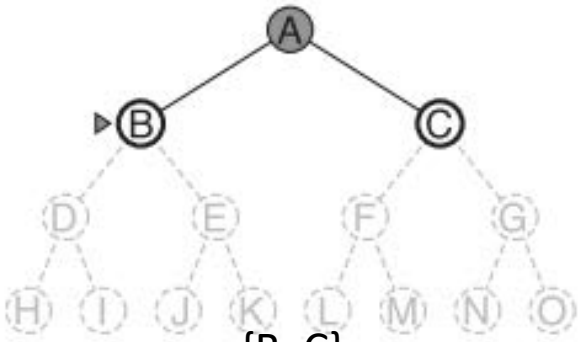
- Only one successor is expanded
  - Rather than all successors
- Fewer memory

Depth-First Search (DFS)

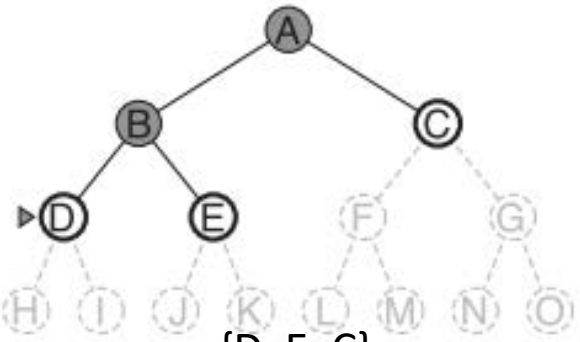
Works like a stack



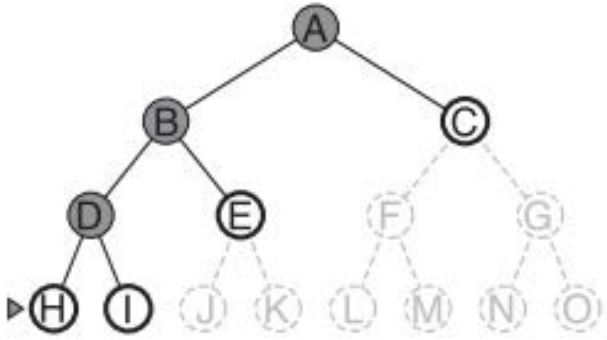
{A}



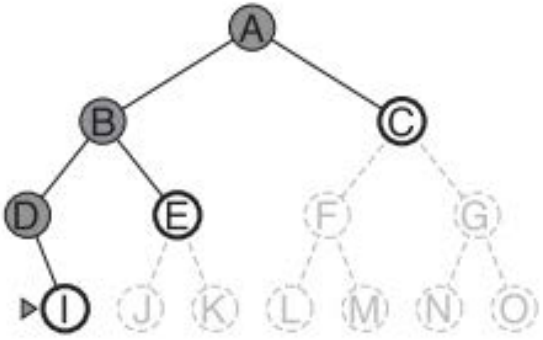
{B, C}



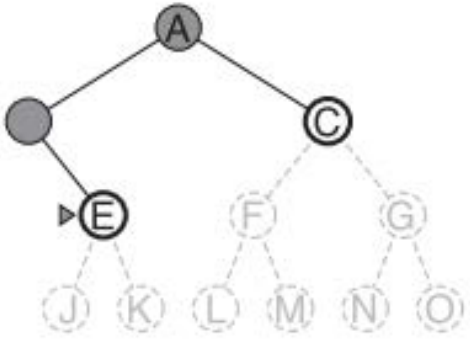
{D, E, C}



{H, I, E, C}

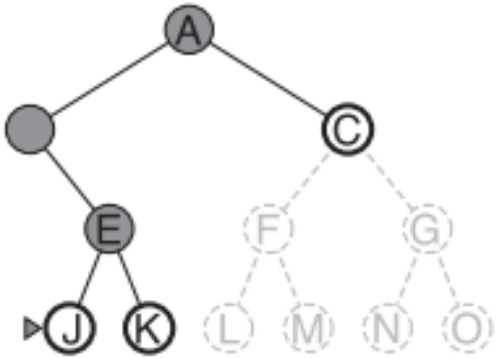


{I, E, C}

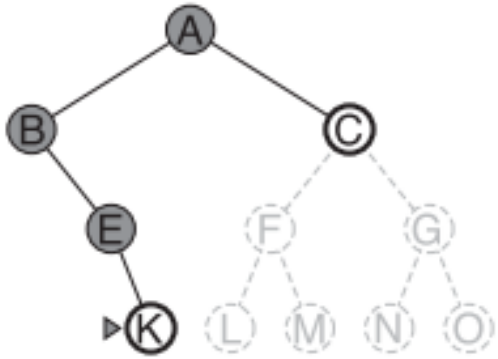


{E, C}

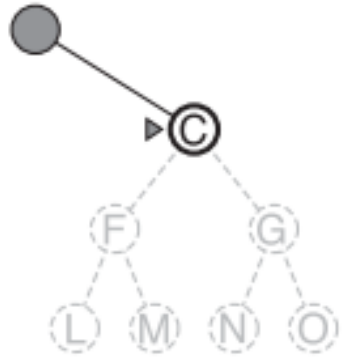
Depth-First Search (DFS)



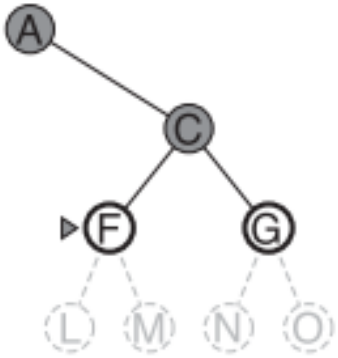
{J, K, C}



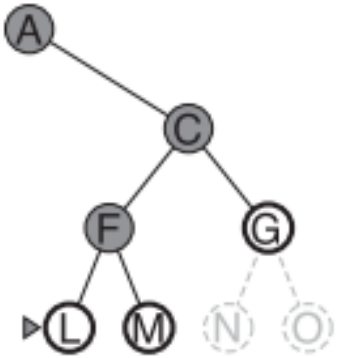
{K, C}



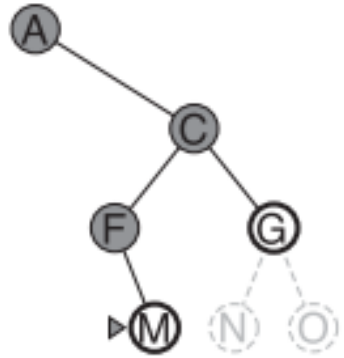
{C}



{F, G}



{L, M, G}



{M, G}

Evaluation of DFS

Not complete

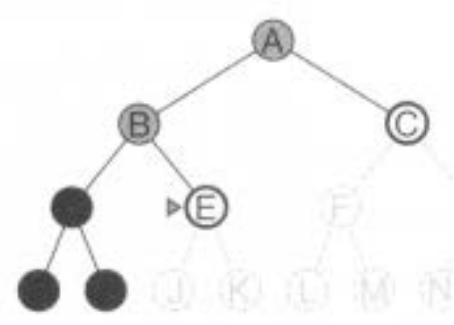
- Path may be infinite or looping
  - Never fail and cannot go back try another option

Not optimal

- Not guarantee the best solution

Overcome reasonably

- Time and space complexities



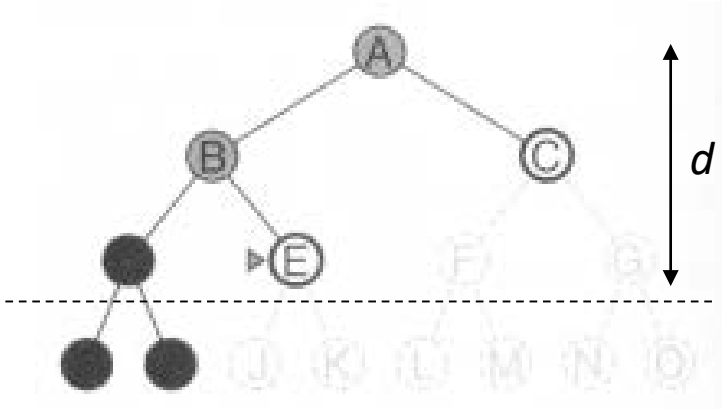
Depth-Limited Search (DLS)

Depth-first search with a **predefined** maximum depth

- Not easy to define the suitable maximum depth
- Too small → no solution can be found
- Too large → suffer the same problem

Evaluation of DLS

- Size of the tree is limited
  - Not guarantee completeness
- Still not optimal

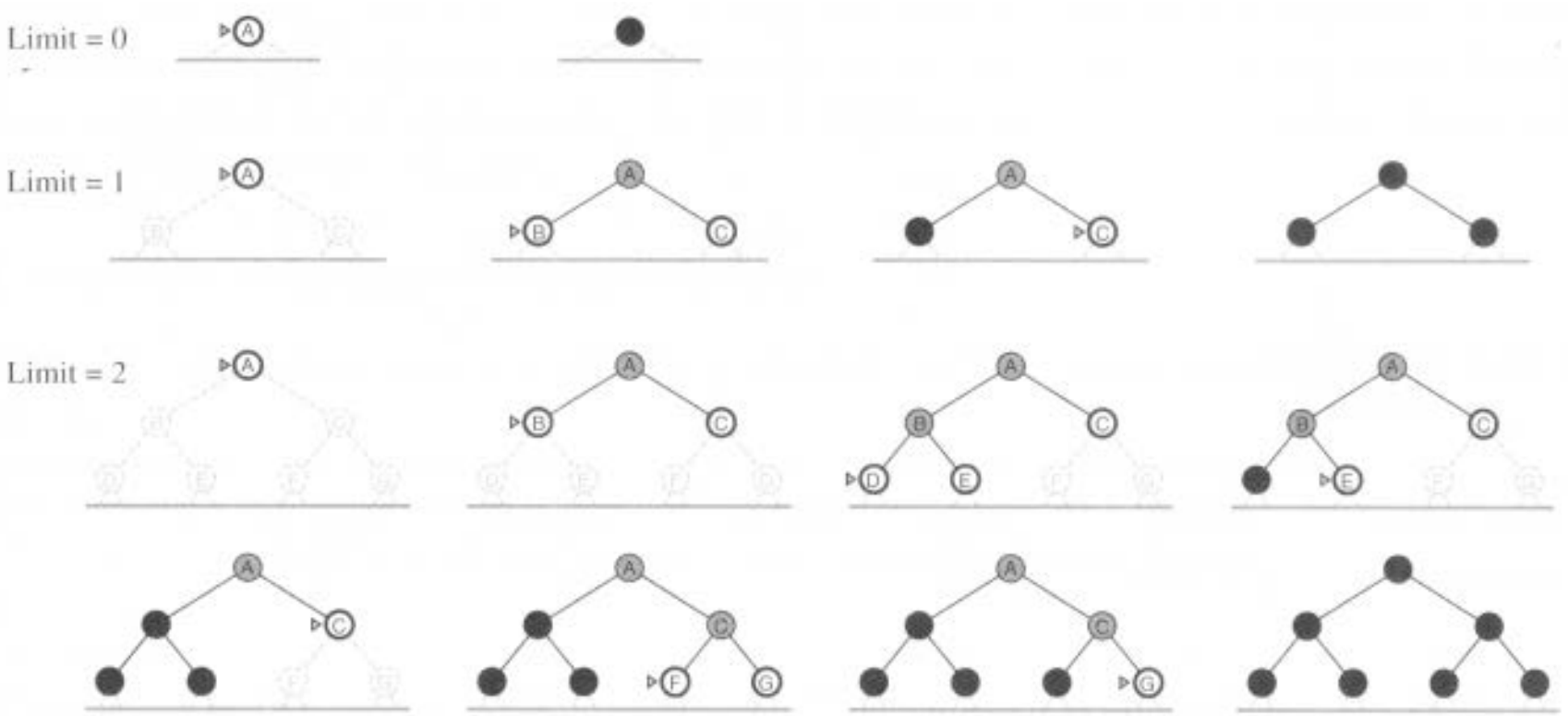


Iterative Deepening Search (IDS)

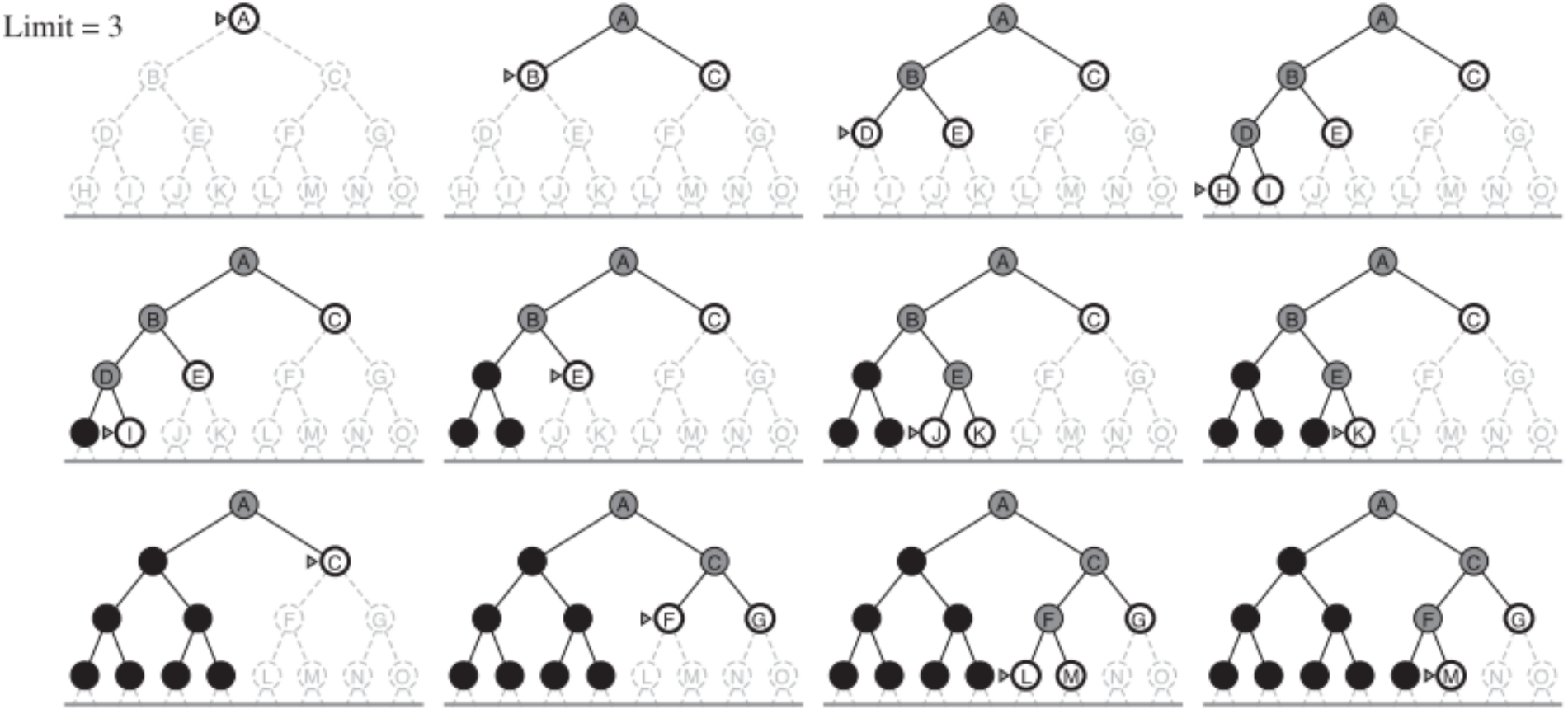
Depth limit d is not predefined (fixed)

Try all possible depth limits

- Firstly d = 0, then 1, 2, and so on



Iterative Deepening Search (IDS)



Evaluation of IDS

Optimal

Complete

Time and space complexities

- Reasonable

Suitable for the problem

- Having large search space
- Depth d of the solution is not known

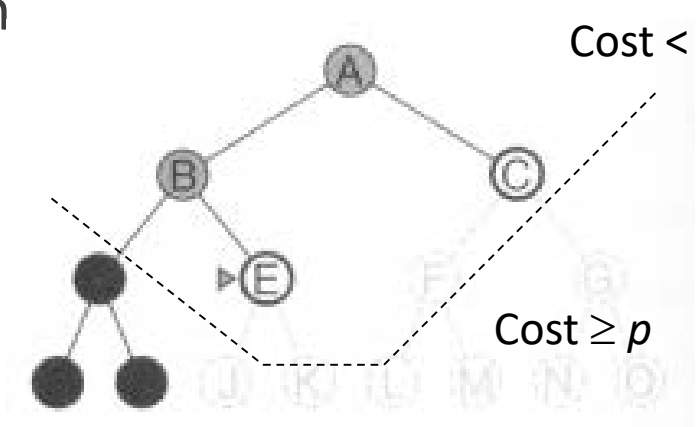
Iterative Lengthening Search (ILS)

IDS

- Depth d as limit

ILS

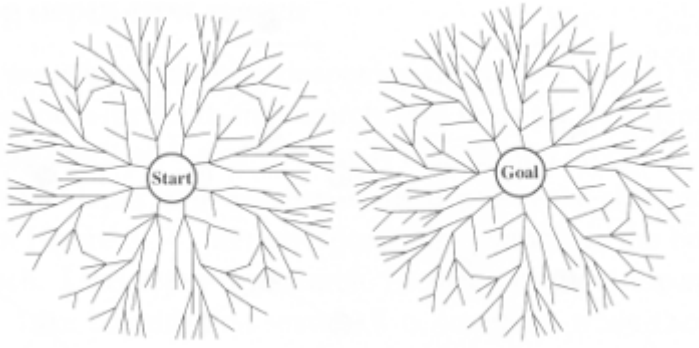
- Path cost p as limit
- Iterative version for uniform cost search
- Advantages of uniform cost search
  - Avoiding its memory requirements
- Incurs substantial overhead
  - Compared to uniform cost search



Bidirectional Search

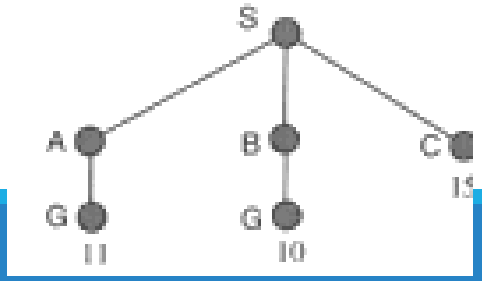
Run two simultaneous searches

- One forward from initial state
- Another backward from the goal
- Stop when two searches meet



Computing backward is difficult

- Huge amount of goal states
- At goal state, which actions are used to compute it?
- Can actions be reversible to compute its predecessors?
  - G = f(B), f^{-1} exists?



Comparing Search Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

Avoiding Repeated States

Repeated states

- States already encountered or expanded before
- On some other path

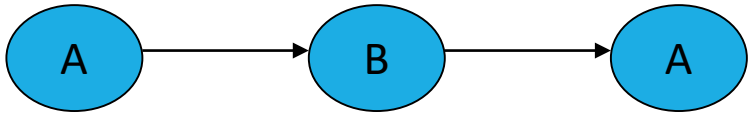
All search strategies

- Possible to suffer from repeated states
  - May cause the path to be infinite loop
  - Even finite loop, the search becomes much inefficient

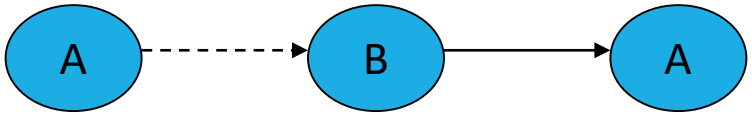
Avoiding Repeated States

Three ways

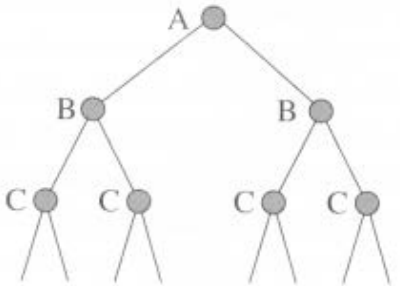
- Do not return to state it just came from
  - Refuse generation of successor same as its **parent state**



- Do not create paths with cycles
  - Refuse generation of successor same as its **ancestor states**



- Do not generate any generated state
  - Not only ancestor states
  - All other expanded states have to be checked against



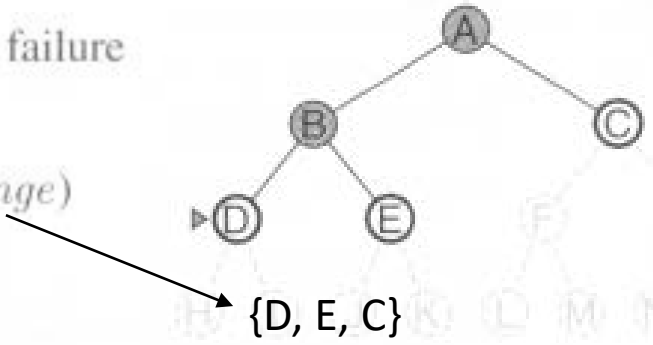
Increased Difficulty

Avoiding Repeated States

Closed list

- Data structure defined
- A **set** storing every expanded node
- Discard current node that matches a node in closed list

```
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 EMPTY?(fringe) then return failure
    node ← REMOVE-FIRST(fringe)
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```



Searching with Partial Information

Previous problems are fully observable

- Known initial state
- Deterministic effect of actions

Incomplete knowledge of states or actions

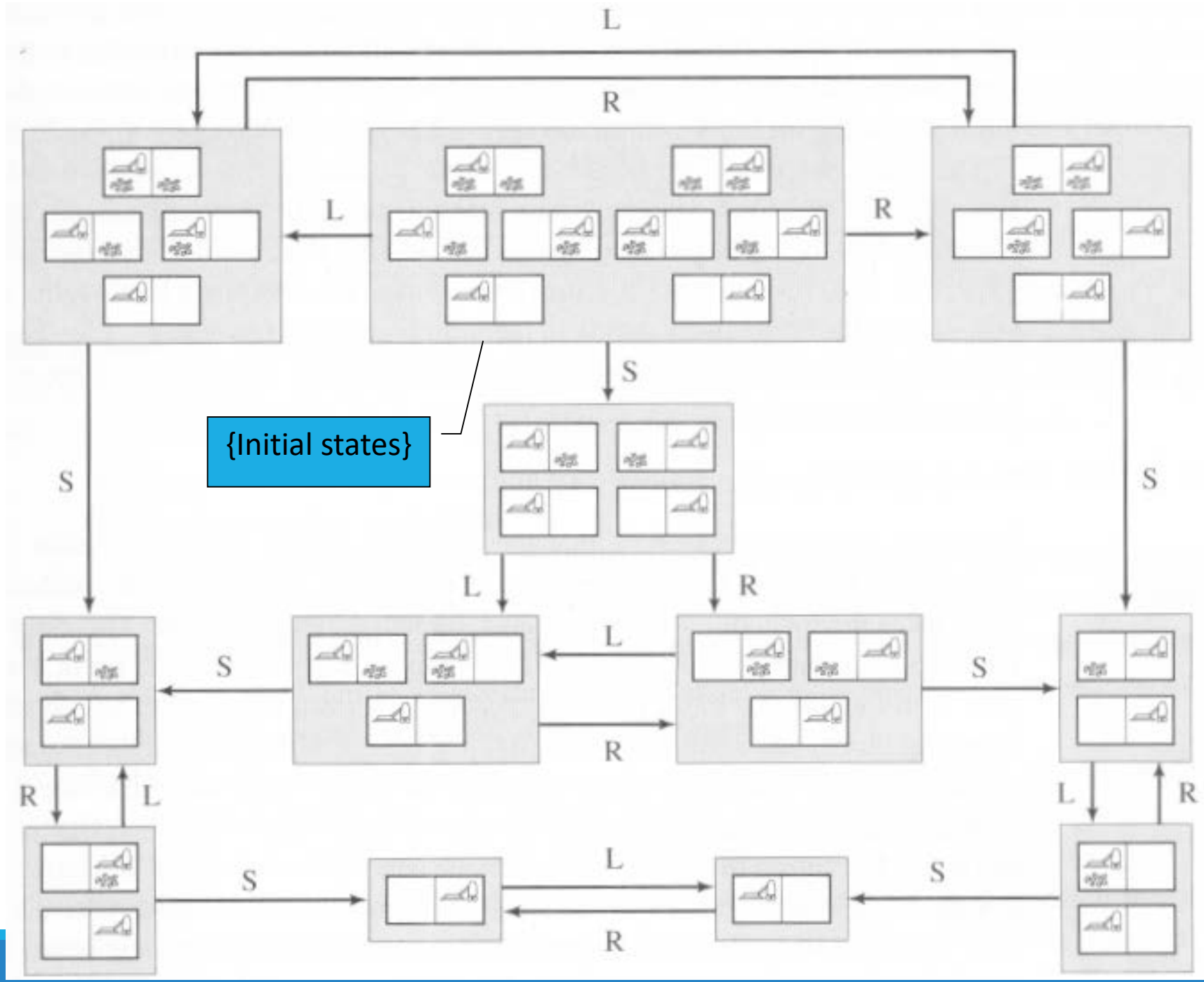
- Three distinct problem types

1. Sensorless or conformant problems
2. Contingency (Nondeterministic) problems
3. Exploration problems

Sensorless Problems

Agent has no sensors

- Do not know initial state
- Search tree = Initial state + Actions
  - No initial state, no search tree can be built
- Many possible initial states
  - Start with a set of possible initial states
- Each action: {Initial states} → {Possible successor states}
- Possible successor state
  - Belief states
  - States we believe agent is located


L
R
L
R
S
{Initial states}
S
S
L
R
S
L
S
R
L
R
L
R
S
L
S
R

Contingency Problems

Effect of actions are uncertain

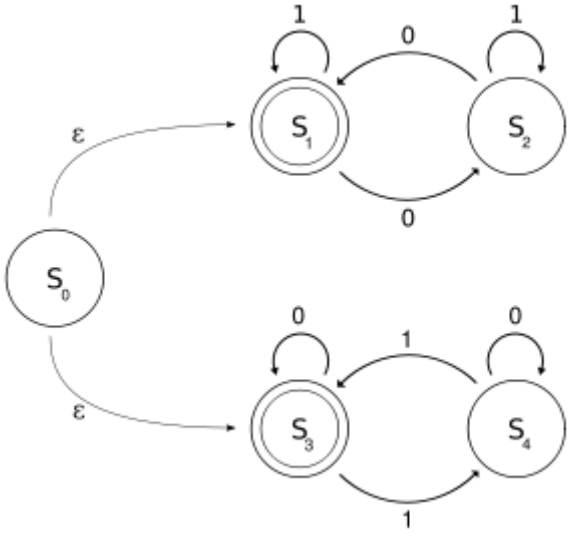
- Nondeterministic

Build a tree to represent

- Different possible effects of an action
- Contingencies

Agent is interleaving

- Search, execute, search, execute, …
- Instead of single “search, execute”



Exploration Problems

Both states and actions about environment are unknown

- Agent has to explore / do experiment
  - Perform some actions and see the results
  - 1) State 2) Effect of the actions
- Involve significant danger in some states
  - Some actions may damage agent

Learn knowledge (if survive)

- Environment and effects of actions
- Solve subsequent problems