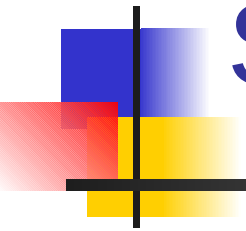


Problem Solving and Searching

CS 171/271

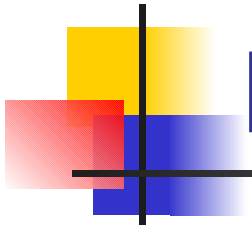
(Chapter 3)

Some text and images in these slides were drawn from Russel & Norvig's published material

Problem Solving Agent Function

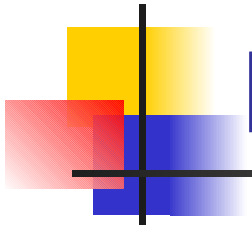
```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```



Problem Solving Agent

- Agent finds an action sequence to achieve a goal
- Requires problem formulation
  - Determine goal
  - Formulate problem based on goal
- Searches for an action sequence that solves the problem
- Actions are then carried out, ignoring percepts during that period



Problem

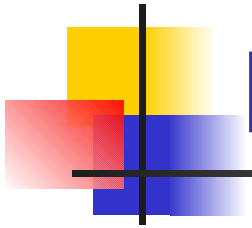
Initial state

Possible actions / Successor function

Goal test

Path cost function

* State space can be derived from the initial state and the successor function



Example: Vacuum World

Environment consists of two squares, A (left) and B (right)

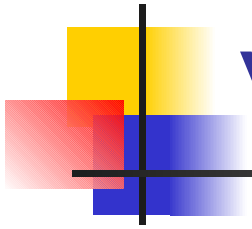
Each square may or may not be dirty

An agent may be in A or B

An agent can perceive whether a square is dirty or not

An agent may move left, move right, suck dirt (or do nothing)

Question: is this a complete PEAS description?

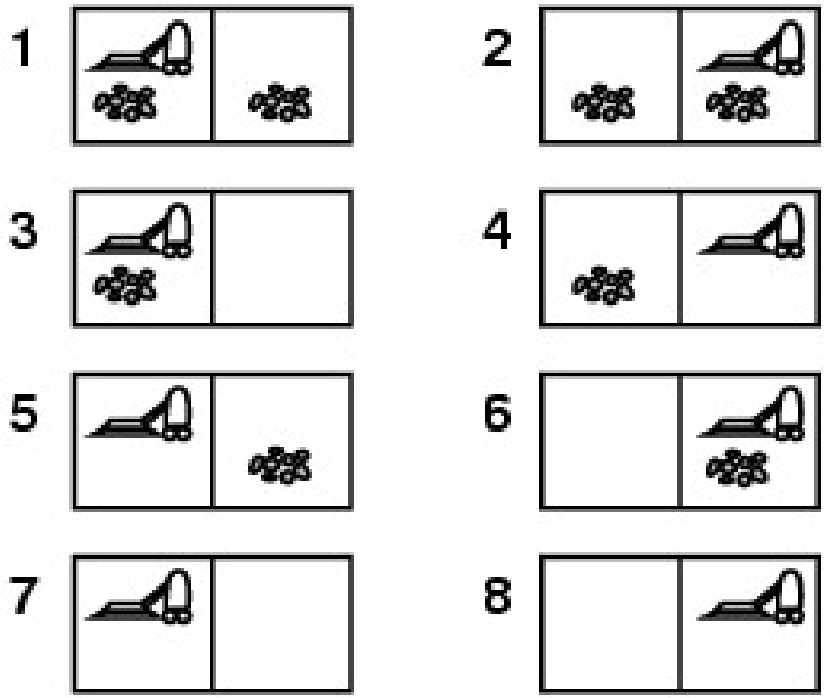


Vacuum World Problem

- Initial state: configuration describing
  - location of agent
  - dirt status of A and B
- Successor function
  - R, L, or S, causes a different configuration
- Goal test
  - Check whether A and B are both not dirty
- Path cost
  - Number of actions

State Space

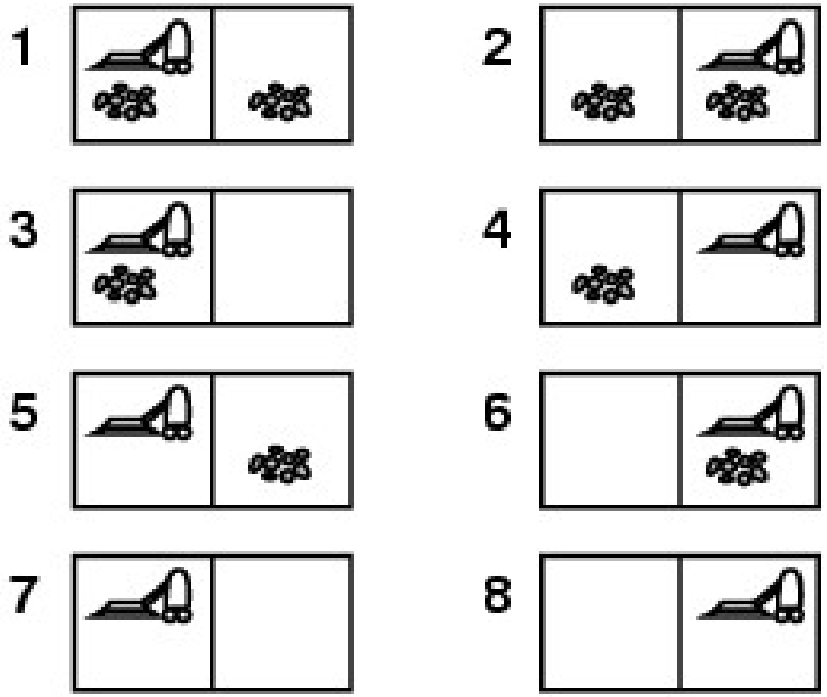
2 possible locations
x
2 x 2 combinations
(A is clean/ dirty,
B is clean/ dirty)
=
8 states



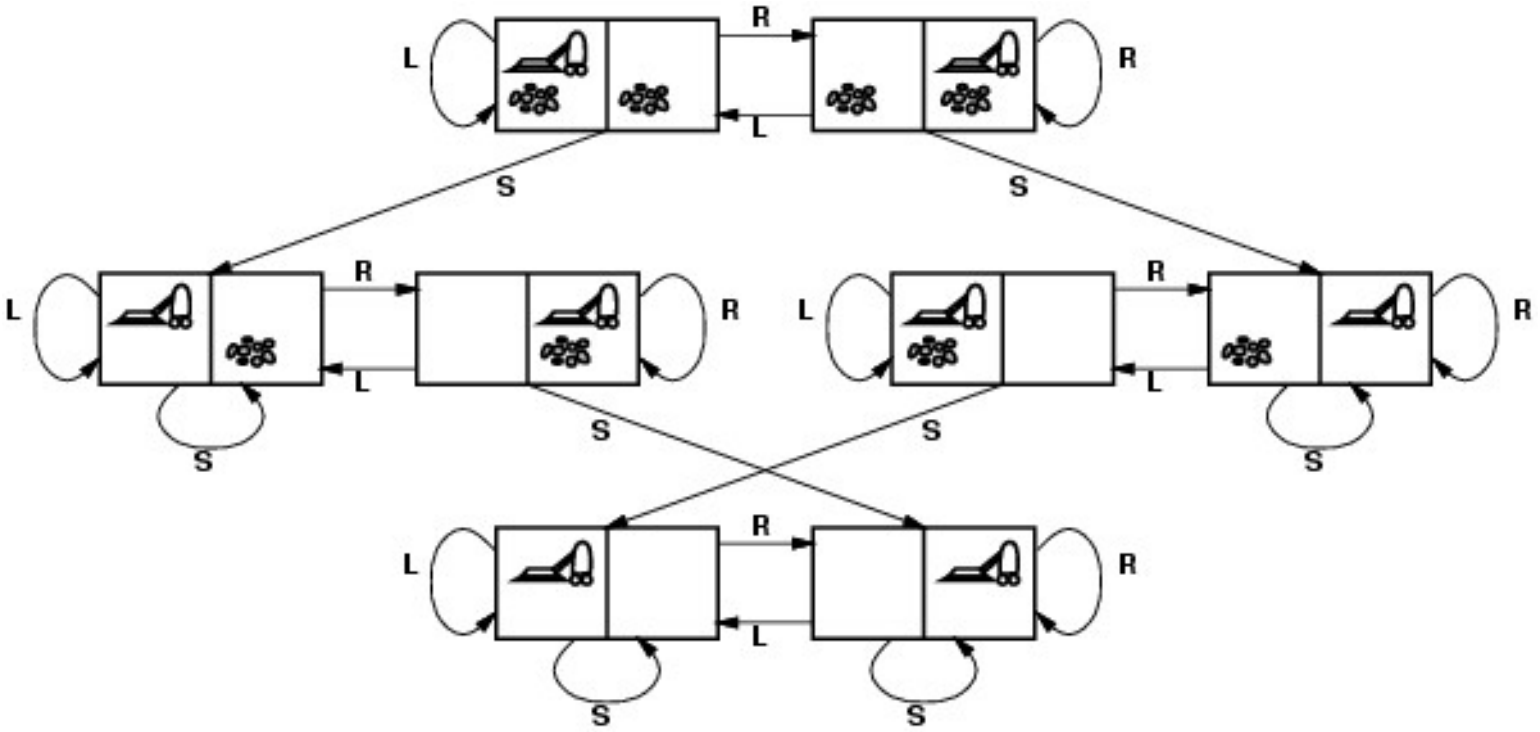
Sample Problem and Solution

Initial State: 2

Action Sequence: Suck, Left, Suck (brings us to which state?)



States and Successors



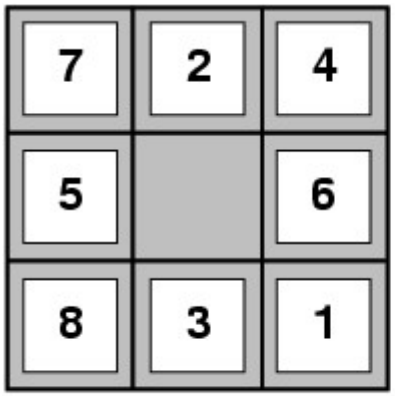
Example: 8-Puzzle

Initial state:
as shown

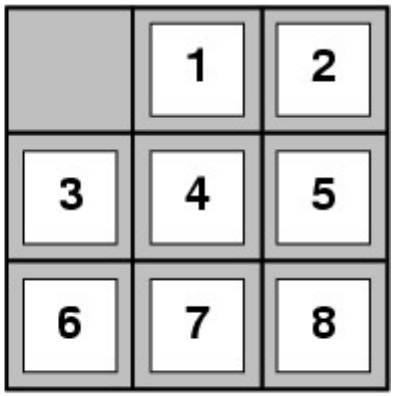
Actions?
successor
function?

Goal test?

Path cost?

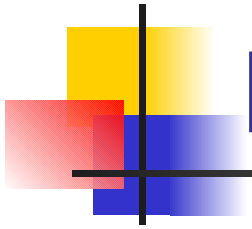


Start State



Goal State

Example: 8-Queens Problem



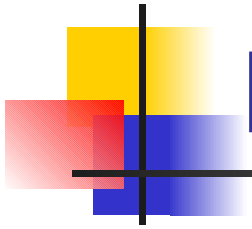
Position 8 queens on a chessboard so that no queen attacks any other queen

Initial state?

Successor function?

Goal test?

Path cost?



Example: Route-finding

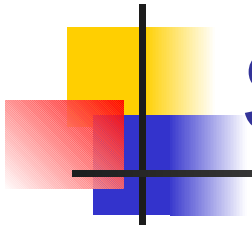
Given a set of locations, links (with values) between locations, an initial location and a destination, find the best route

Initial state?

Successor function?

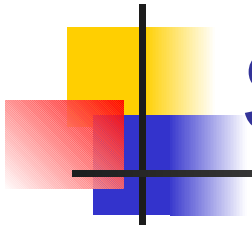
Goal test?

Path cost?



Some Considerations

- Environment ought to be be static, deterministic, and observable
  - Why?
- If some of the above properties are relaxed, what happens?
- Toy problems versus real-world problems



Searching for Solutions

Searching through the state space

Search tree rooted at initial state

A node in the tree is expanded by applying successor function for each valid action

- Children nodes are generated with a different path cost and depth

Return solution once node with goal state is reached

Tree- Search Algorithm

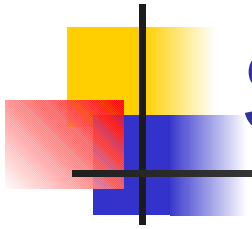
```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the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

fringe: initially-empty container

```
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](STATE[node]) then return SOLUTION(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
```

What is returned?

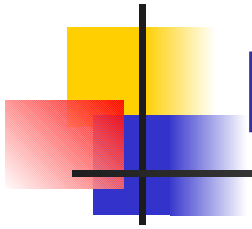


Search Strategy

Strategy: specifies the order of node expansion

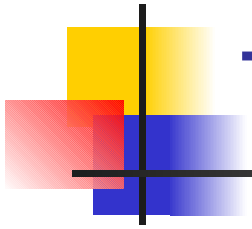
Uninformed search strategies: no additional information beyond states and successors

Informed or heuristic search: expands “more promising” states



Evaluating Strategies

- 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

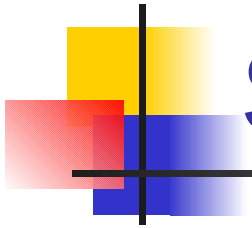
Expressed in terms of:

b: branching factor

- depends on possible actions
- max number of successors of a node

d: depth of shallowest goal node

m: maximum path-length in state space



Uninformed Search Strategies

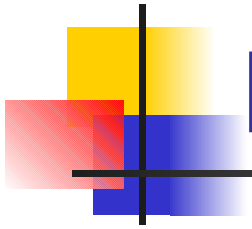
Breadth-First Search

Uniform-Cost Search

Depth-First Search

Depth-Limited Search

Iterative Deepening Search



Breadth-First Search

fringe is a regular first-in-first-out queue

Start with initial state; then process the successors of initial state, followed by their successors, and so on…

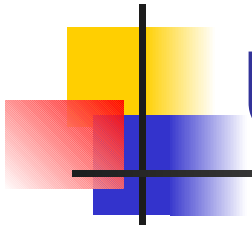
- Shallow nodes first before deeper nodes

Complete

Optimal (if path-cost = node depth)

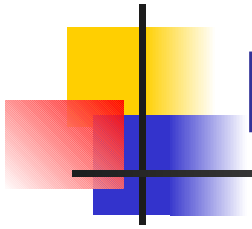
Time Complexity: $O(b + b^2 + b^3 + \ldots + \mathbf{b^{d+1}})$

Space Complexity: same



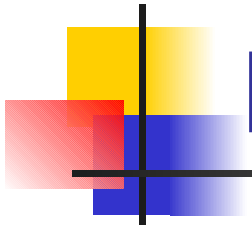
Uniform-Cost Search

- Prioritize nodes that have least path-cost (*fringe* is a priority queue)
- If path-cost = number of steps, this degenerates to BFS
- Complete and optimal
  - As long as zero step costs are handled properly
- The route-finding problem, for example, have varying step costs
  - Dijkstra's shortest-path algorithm <-> UCS
- Time and space complexity?



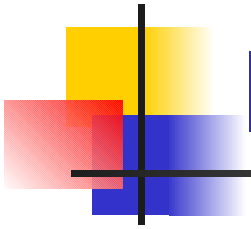
Depth-First Search

- *fringe* is a stack (last-in-first-out container)
- Go as deep as possible and then backtrack
- Often implemented using recursion
- Not complete and might not terminate
- Time Complexity: $O(b^m)$
- Space complexity: O(bm)



Depth-Limited Search

- DFS with a pre-determined depth-limit ℓ
  - Guaranteed to terminate
- Still incomplete
  - Worse, we might choose $\ell < m$ (shallowest goal node)
- Depth-limit ℓ can be based on problem definition
  - e.g., graph diameter in route-finding problem
- Time and space complexity depend on ℓ

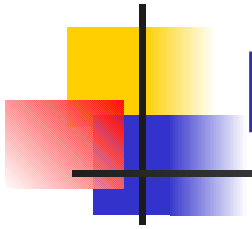


Iterative Deepening Search

- Depth-Limited Search for $\ell = 0, 1, 2, \ldots$
- Stops when goal is found (when ℓ becomes d)
- Complete and optimal (if path-cost = node-depth)
- Time and space complexity?

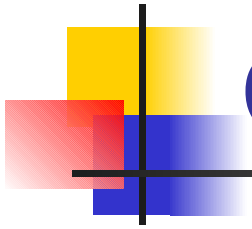
Comparing Search Strategies

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



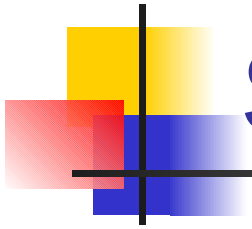
Bi-directional Search

- Run two simultaneous searches
  - One forward from initial state
  - One backward from goal state
- Stops when node in one search is in fringe of the other search
- Rationale: two “half-searches” quicker than a full search
- Caveat: not always easy to search backwards



Caution: Repeated States

- Can occur specially in environments where actions are reversible
- Introduces the possibility of infinite search trees
  - Time complexity blows up even with fixed depth limits
- Solution: detect repeated states by storing node history



Summary

A problem is defined by its initial state, a successor function, a goal test, and a path cost function

- Problem's environment <-> state space

Different strategies drive different tree-search algorithms that return a solution (action sequence) to the problem

Coming up: **informed** search strategies