Solving problems by searching

Uninformed search

Slides from Russell & Norvig book, revised by Andrea Roli

Outline

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

Prologue

- Is there a general strategy for solving problems such as ‘Wolf, goat and cabbage’, ‘Cryptoarithmetic’, ‘8-puzzle’, etc.?
- What are the entities that have to be formalized?
- Is it possible to design a machine that can solve these problems?
- What are the assumptions on the (real) world that we have to formulate?

Example: Romania

On holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest. Suppose we do not have a map, but we only know which cities we can reach from the city we are in.

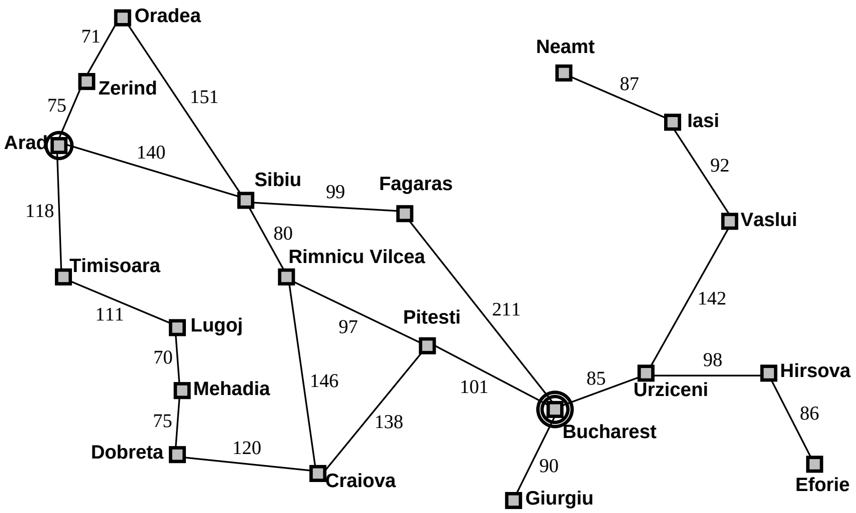
From Arad: Zerind, Sibiu, Timisoara

Example: Romania

On holiday in Romania; currently in Arad. Flight leaves tomorrow from Bucharest.

Formulate goal:
  be in Bucharest
Formulate problem:
  states: various cities
  actions: drive between cities
Find solution:
  sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



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.

Problem types

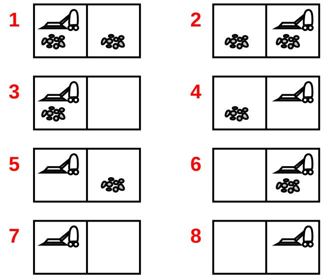
- Deterministic, fully observable ⟹ single-state problem
  Agent knows exactly which state it will be in; solution is a sequence
- Non-observable ⟹ sensorless problem
  Agent may have no idea where it is; solution (if any) is a sequence
- Nondeterministic and/or partially observable ⟹ contingency problem
  percepts provide **new** information about current state
  solution is a contingent plan or a policy
  often **interleave** search, execution
- Unknown state space ⟹ exploration problem (“online”)

Example: vacuum world

Single-state, start in #5. Solution??
$[Right, Suck]$

Sensorless, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$
e.g., $Right$ goes to $\{2, 4, 6, 8\}$. Solution??
$[Right, Suck, Left, Suck]$

Contingency, start in #5
Murphy's Law: $Suck$ can dirty a clean carpet
Local sensing: dirt, location only.
Solution??
$[Right,$ **loop**$\{$**if** $dirt$ **then** $Suck\}]$



Single-state problem formulation

A problem is defined by four items:

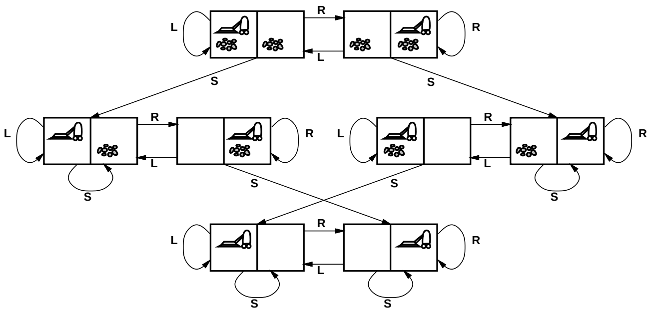
- initial state e.g., "at Arad"
- successor function $S(x)$ = set of action–state pairs
  e.g., $S(Arad) = \{\langle Arad \rightarrow Zerind, Zerind\rangle, \ldots\}$
- goal test, can be
  explicit, e.g., x = "at Bucharest"
  implicit, e.g., $NoDirt(x)$
- path cost (additive)
  e.g., sum of distances, 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

Selecting a state space

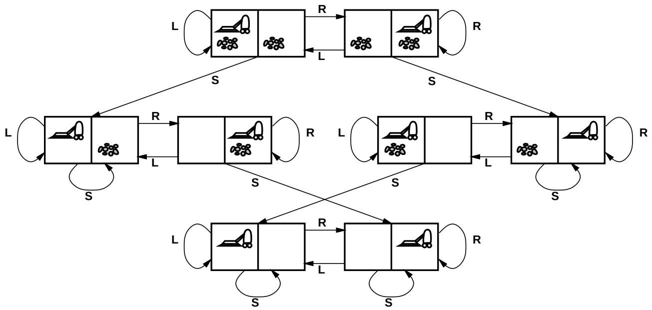
- Real world is absurdly complex $\Rightarrow$ state space must be **abstracted** for problem solving
- (Abstract) state = set of real states
- (Abstract) action = complex combination of real actions
  e.g., "Arad $\rightarrow$ Zerind" represents a complex set
  of possible routes, detours, rest stops, 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: vacuum world state space graph



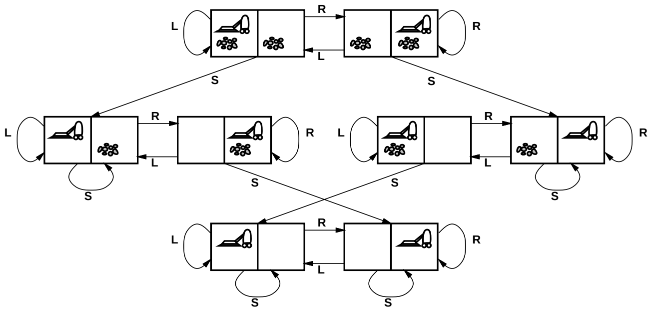
states??
actions??
goal test??
path cost??

Example: vacuum world state space graph



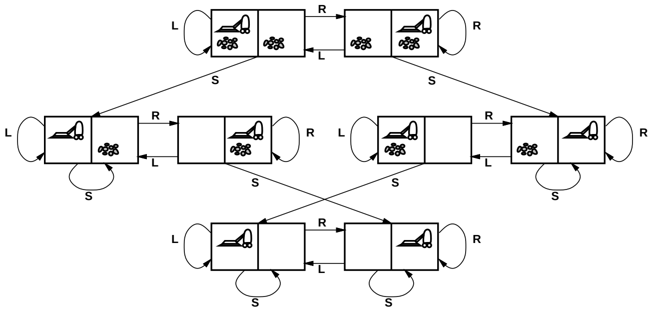
states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??
goal test??
path cost??

Example: vacuum world state space graph



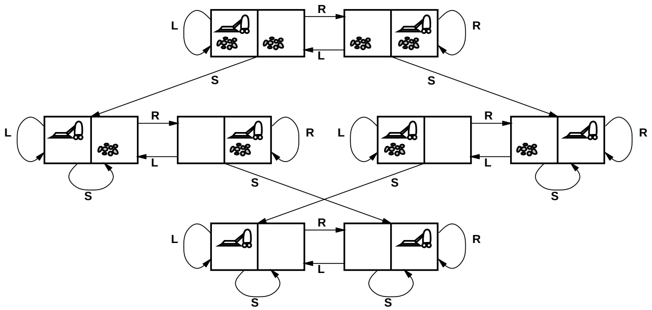
states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??: *Left*, *Right*, *Suck*, *NoOp*
goal test??
path cost??

Example: vacuum world state space graph



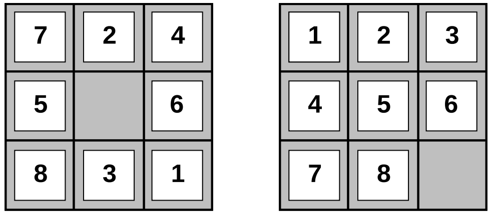
states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??: *Left*, *Right*, *Suck*, *NoOp*
goal test??: no dirt
path cost??

Example: vacuum world state space graph



states??: integer dirt and robot locations (ignore dirt amounts etc.)
actions??: *Left*, *Right*, *Suck*, *NoOp*
goal test??: no dirt
path cost??: 1 per action (0 for *NoOp*)

Example: The 8-puzzle

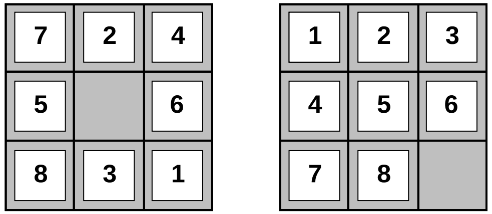


Start State

Goal State

states??
actions??
goal test??
path cost??

Example: The 8-puzzle

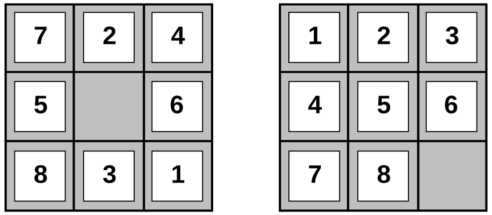


Start State

Goal State

states??: integer locations of tiles (ignore intermediate positions)
actions??
goal test??
path cost??

Example: The 8-puzzle

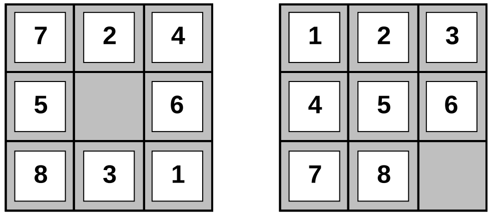


Start State

Goal State

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

Example: The 8-puzzle

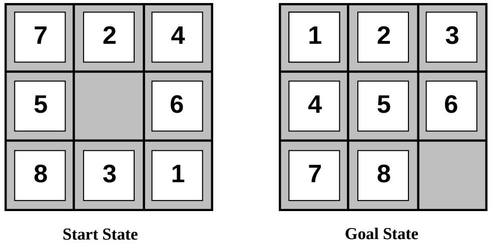


Start State

Goal State

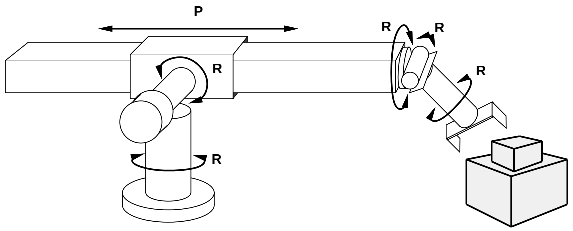
states??: integer locations of tiles (ignore intermediate positions)
actions??: move blank left, right, up, down (ignore unjamming etc.)
goal test??: = goal state (given)
path cost??

Example: The 8-puzzle



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: 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

Other famous problems

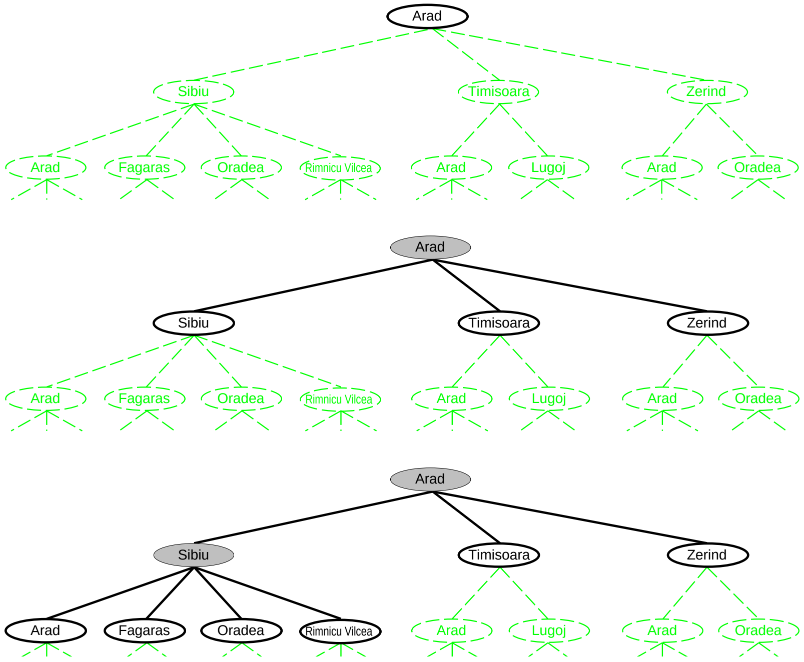
- Missionaires and cannibals problem
- Hanoi tower
- Monkey and banana problem
- Puzzles and logical games

Tree search algorithms

Basic idea:
offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. expanding states)

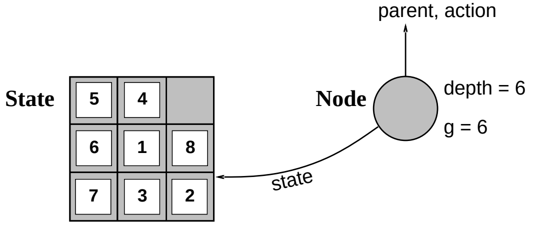
```
function Tree-Search(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node 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 parent, children, depth, path cost $g(x)$
States do not have parents, children, depth, or path cost!



The EXPAND function creates new nodes, filling in the various fields and using the SUCCESSORFN of the problem to create the corresponding states.

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, STATE(node)) 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(STATE[node], action, result)
        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

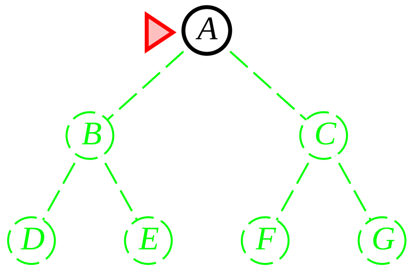
Uninformed strategies use only the information available in the problem definition

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

Breadth-first search

Expand shallowest unexpanded node

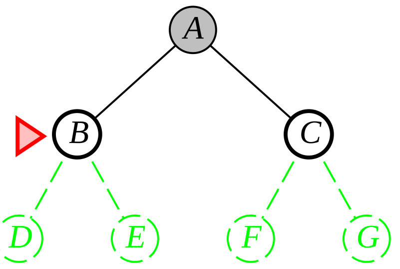
Implementation:
fringe is a FIFO queue, i.e., new successors go at end



Breadth-first search

Expand shallowest unexpanded node

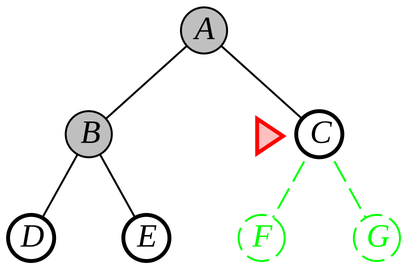
Implementation:
fringe is a FIFO queue, i.e., new successors go at end



Breadth-first search

Expand shallowest unexpanded node

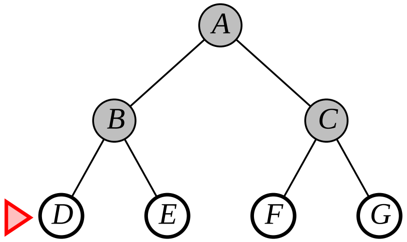
Implementation:
fringe is a FIFO queue, i.e., new successors go at end



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 is a nondecreasing function of node depth)

Space is the big problem; can easily generate nodes at 100MB/sec, so 24hrs = 8640GB.

Uniform-cost search

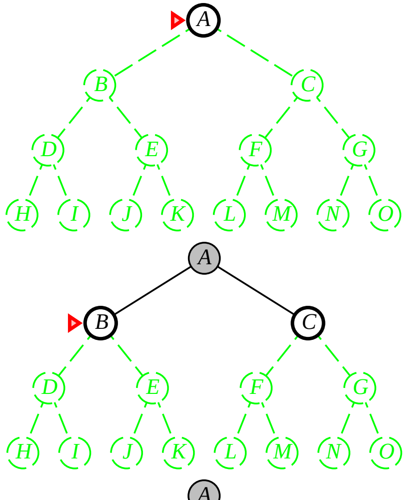
Expand least-cost unexpanded node
Implementation:
fringe = queue ordered by path cost, lowest first

- Complete and optimal
- Equivalent to breadth-first if step costs all equal

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, may be much faster than breadth-first
- **Space**: $O(bm)$, i.e., linear space!
- **Optimal**: No

Chronological backtracking

- Variant of DFS
- Successors generated one at a time
- Reduced space complexity wrt DFS: $O(b)$

Depth-limited search

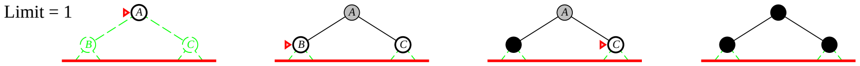
- depth-first search with depth limit l, i.e., nodes at depth l have no successors
- Not complete if $l <$ d.

Iterative deepening search

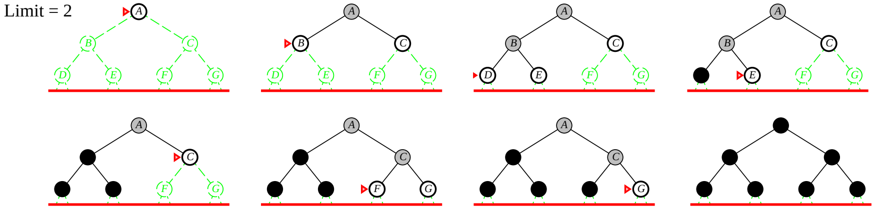
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
    inputs: problem, a problem

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

Iterative deepening search $l = 0$

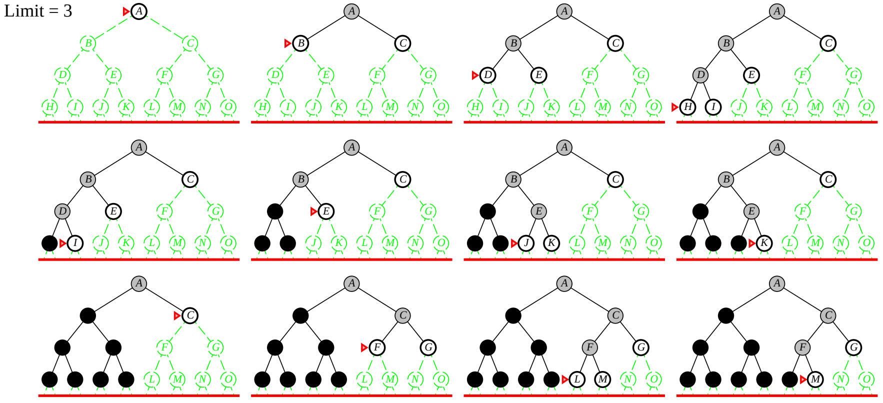
Iterative deepening search $l = 1$



Iterative deepening search $l = 2$



Iterative deepening search $l = 3$



Properties of iterative deepening search

- **Complete**: Yes
- **Time**: $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- **Space**: $O(bd)$
- **Optimal**: Yes, if step cost is a nondecreasing function of node depth.

► IDS preferred uninformed strategy when search space is large and solution depth not known.

Bidirectional search

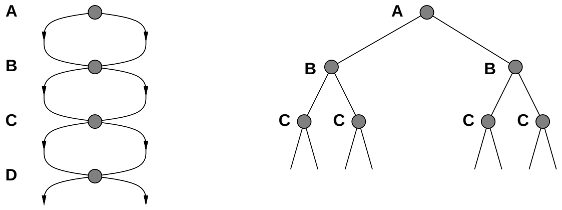
- Run two simultaneous searches
- one *forward* from the initial state
- and the other *backward* from the goal
- stop when they meet

Problems:

- How to compute predecessors?
- Sometimes the goal state is only implicitly defined

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
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms
- Graph search can be exponentially more efficient than tree search