Goal-Based Agents: Solving Problems by Searching

Chapter 3, Sections 1-4

Problem solving agents

Form of **goal-based agent** that formulates the problem of **reaching a goal** in its environment, searches for a **sequence of actions** solving the problem, and executes it.

Assumptions about the task environment:

- ♢ fully observable
- ♢ deterministic
- ♢ static
- ♢ single agent

offline (or open-loop) problem solving is suitable under those assumptions; the entire sequence solution can be **executed "eyes closed."**

Outline

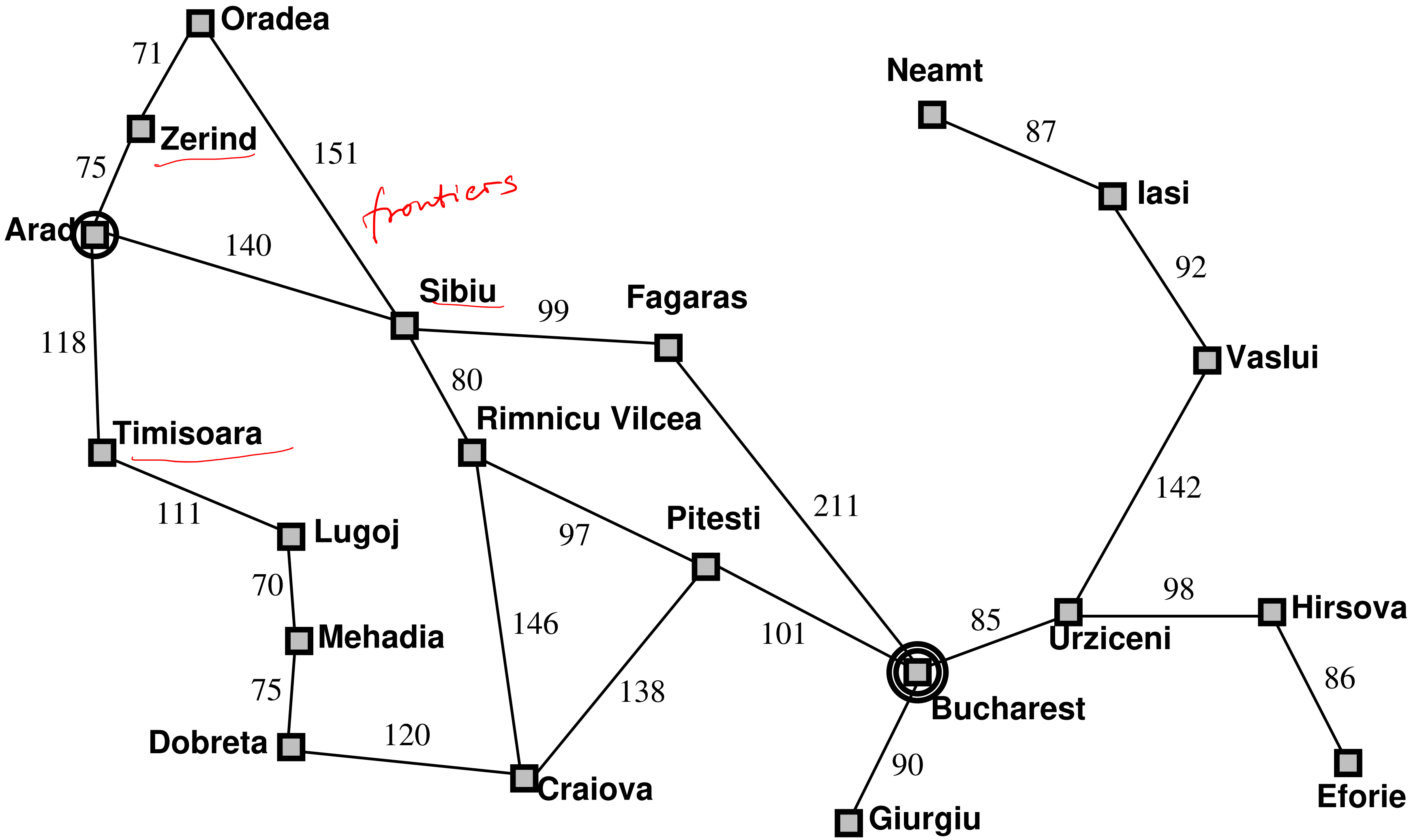
◊ Problem formulation

◊ Problem formulation examples

◊ Tree search algorithm

◊ Uninformed search strategies

Example: Romania



Example: Romania

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

Formulate problem:

- initial state: in Arad
- goal: be in Bucharest
- states: various cities
- actions: drive between cities

Search for a solution:

- sequence of drive actions or equivalently (in this case)
- sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

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., $S(Arad) = \{\langle Arad \rightarrow Zerind, Zerind \rangle, \langle Arad \rightarrow Sibiu, Sibiu \rangle, \ldots\}$

goal test, can be

explicit, e.g., x = "at Bucharest"

implicit, e.g., $HasAirport(x)$

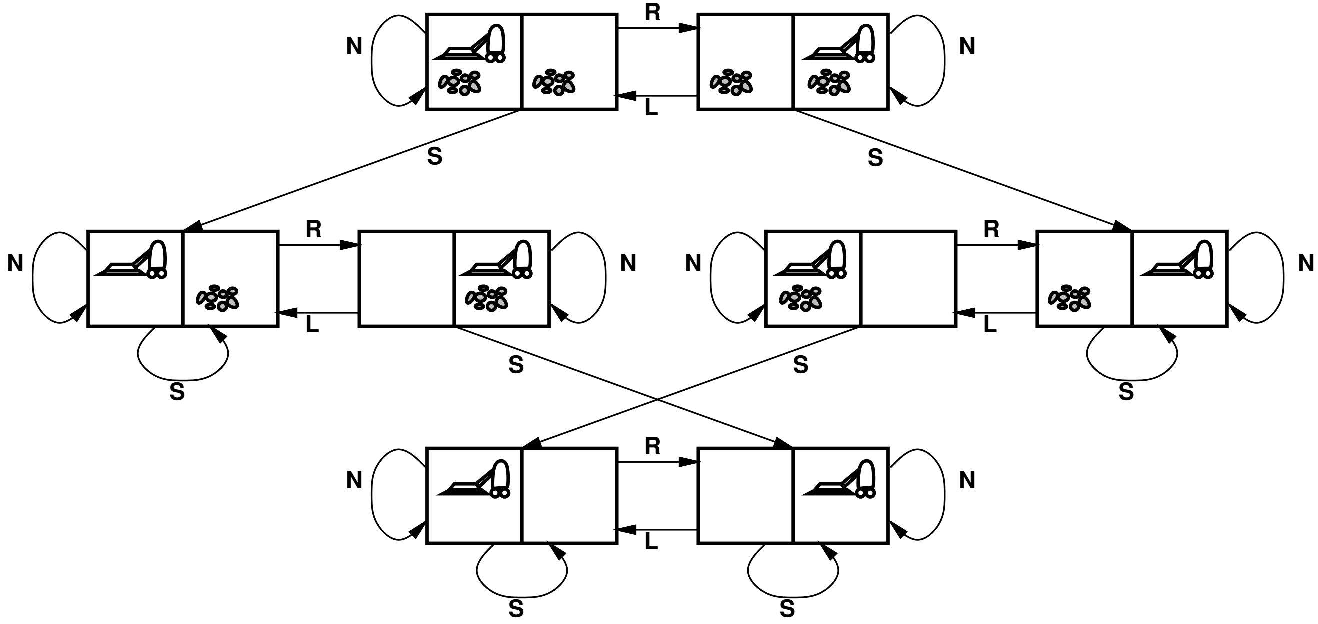
path cost (additive)

e.g., sum of distances, number of actions executed, etc.

$c(x, a, y) \geq 0$ is the step cost of going from state x to state y via action a

A solution is a sequence of actions
leading from the initial state to a goal state

Example: Vacuum world



states??: dirt presence in each room and robot location (ignore dirt amounts)
actions??: $Left$, $Right$, $Suck$, $NoOp$
goal test??: $=$ no dirt
path cost??: 1 per action (0 for $NoOp$)

Selecting a state space

Real world is absurdly complex
$\Rightarrow$ state space must be **abstracted** for problem solving

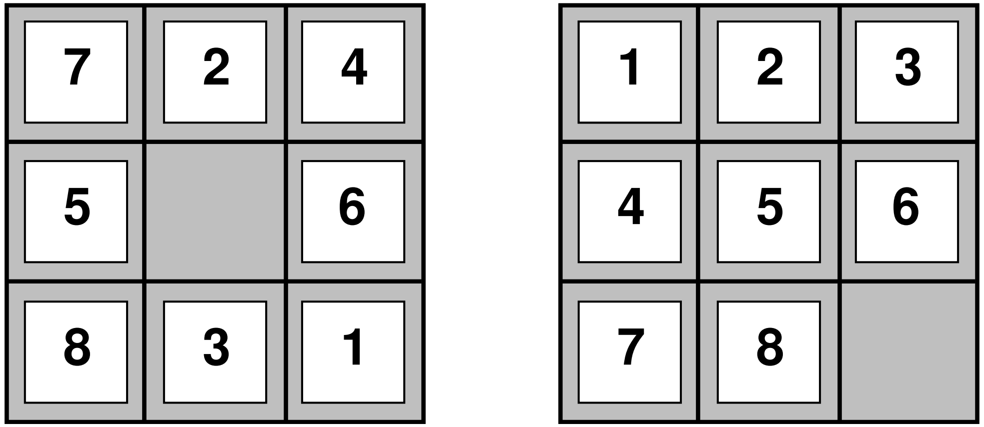
(Abstract) state = set of real states

(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

Abstraction should be "easier" than the original problem!

Example: The 8-puzzle

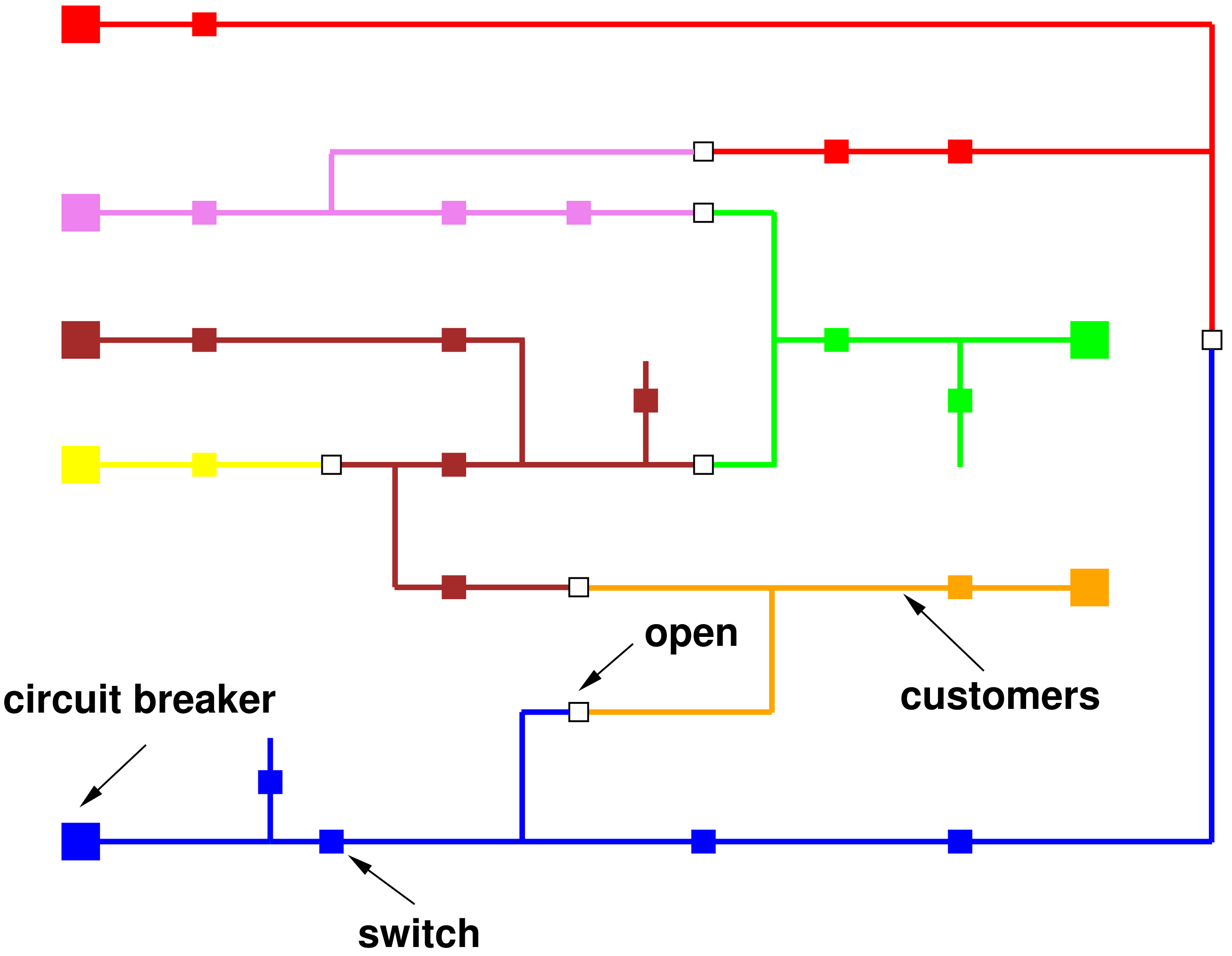


Start State **Goal State**

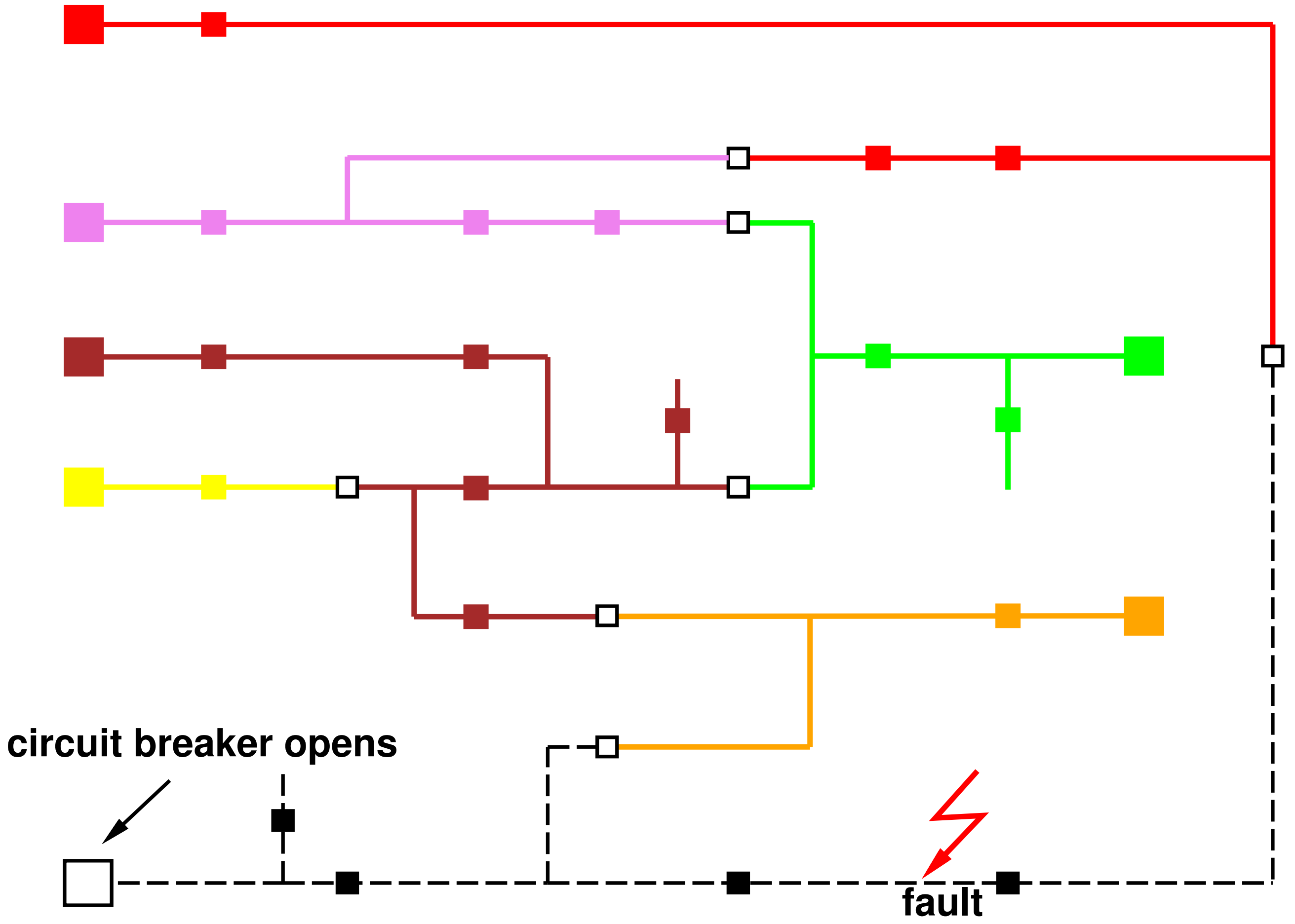
states??: integer locations of tiles (ignore intermediate positions)
actions??: move blank left, right, up, down (ignore unjamming etc.)
goal test??: = goal state (given)
path cost??: 1 per move

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

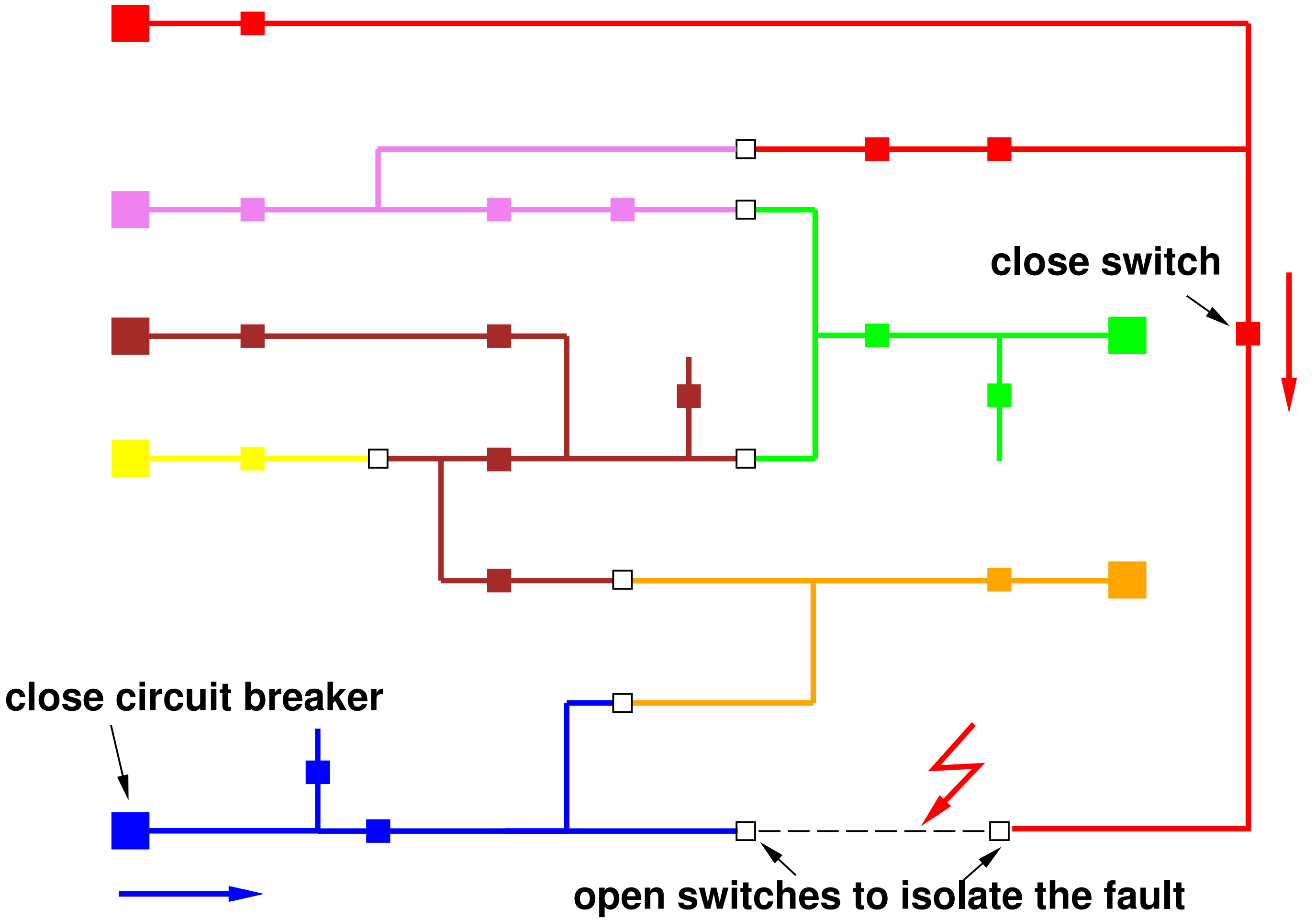
Example: power supply restoration



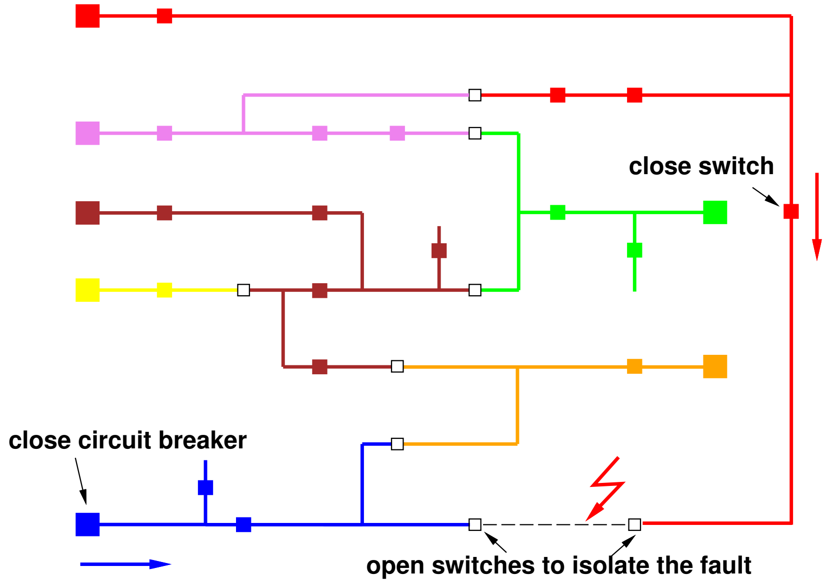
Example: power supply restoration



Example: power supply restoration



Example: power supply restoration



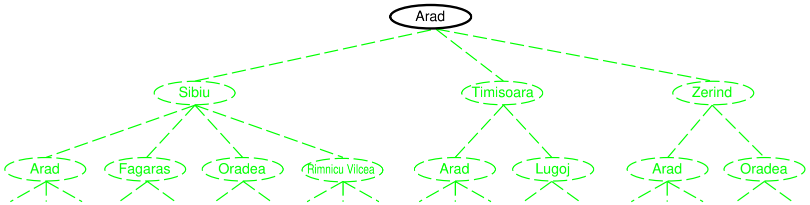
states??: connections, status faulty/non-faulty of the lines,
positions open/closed of the switches and circuit-breakers,
power consummed on each line, capacity of circuit-breakers and lines

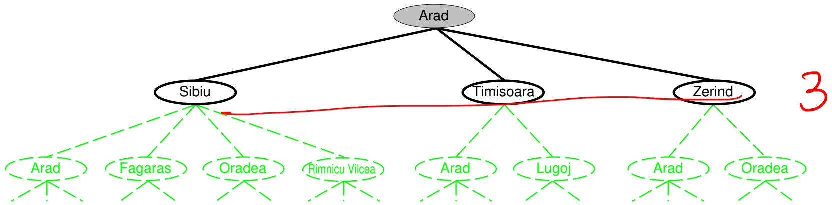
actions??: open/close a switch or a circuit-breaker without exceeding capacity

goal test??: resupply all non-faulty lines

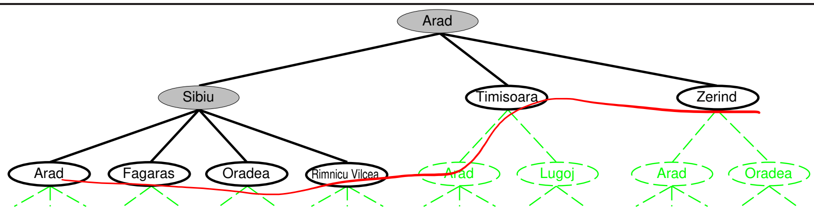
path cost??: number of actions, power margins

Tree search example





3 frontiers



6 frontiers.

Tree search algorithm

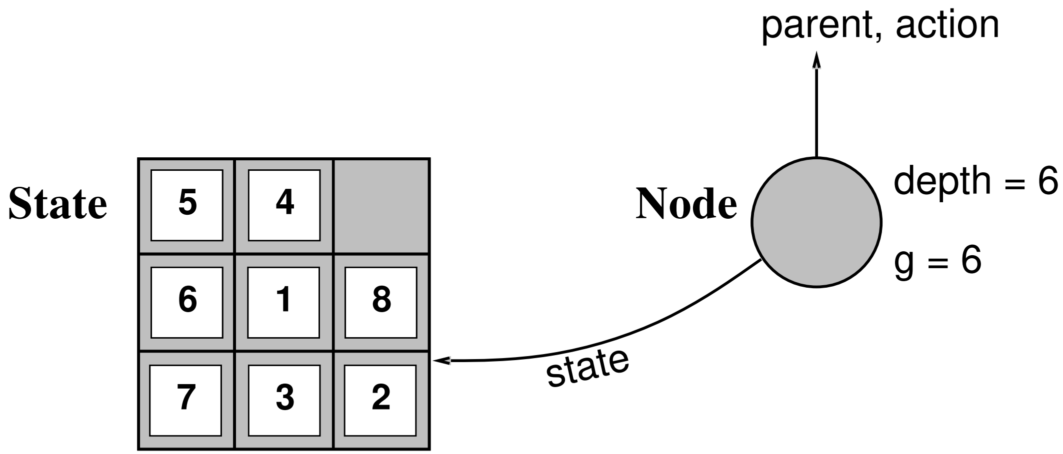
Basic idea:

offline, simulated exploration of state space
by generating successors of already-explored nodes
(a.k.a. expanding nodes)

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 on the frontier **then return** failure
        choose a frontier 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 frontier of the tree
    end

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 state, parent, children, depth, path cost $g(n)$
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.

Frontier implemented as priority queue of nodes ordered according to strategy

the most priority
to the least

Implementation: general tree search

```
function TREE-SEARCH( problem, frontier) returns a solution, or failure
   frontier ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), frontier)
   loop do
       if frontier is empty then return failure
       node ← REMOVE-FRONT(frontier)
       if GOAL-TEST(problem, STATE(node)) then return node
       frontier ← INSERTALL(EXPAND(node, problem), frontier)
```

```
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
```

Uninformed search strategies

A strategy is defined by picking the **order of node expansion**

This is the order used for the **priority queue** implementing the frontier

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

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

Iterative deepening search

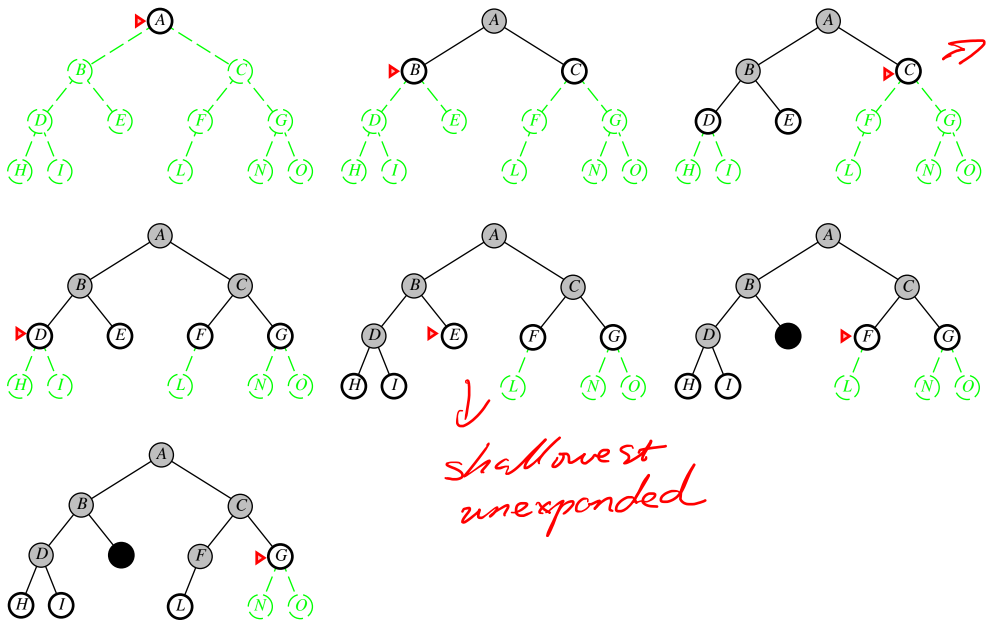
Breadth-first search

Expand shallowest unexpanded node

Implementation: First in First out.

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

shallowest unexpanded.



shallowest unexpanded

Search strategies

A strategy is defined by picking the **order of node expansion** what to expand next?

Strategies are evaluated along the following dimensions:

solvable? completeness—does it always find a solution if one exists?
optimal? solution optimality—does it always find a least-cost solution?
fast? time complexity—number of nodes generated (or expanded)
space-saving? space complexity—maximum number of nodes in memory

Time and space complexity are measured in terms of

m. branching b—maximum branching factor of the search tree
s. depth d—depth of the shallowest solution
m. depth. m—maximum depth of the state space (may be ∞)

Properties of breadth-first search

Complete?? Yes (if b and d are finite)

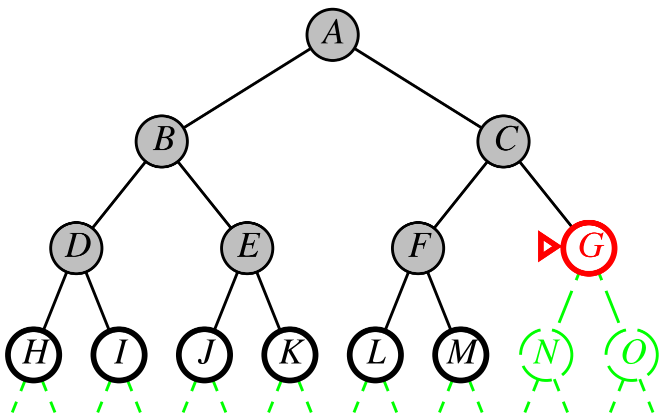
Time?? $1 + b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$

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

$b = 10$, 1 million node/sec, 1Kb/node, d=12 would take 13 days and 1 petabyte of memory.

Too slow



depth	*#nodes*
0	b^0
1	b^1
2=d	$b^2 = b^d$
3=d+1	$b^3 - b = b^{d+1} - b$

Uniform-cost search = BFS if cost = 1 per step.

Expand least-cost unexpanded node

Implementation:

$frontier$ = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

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

Time?? # of nodes with $g \leq C^*$, $O(b^{1+\lceil C^*/\epsilon \rceil})$
where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq C^*$, $O(b^{1+\lceil C^*/\epsilon \rceil})$

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

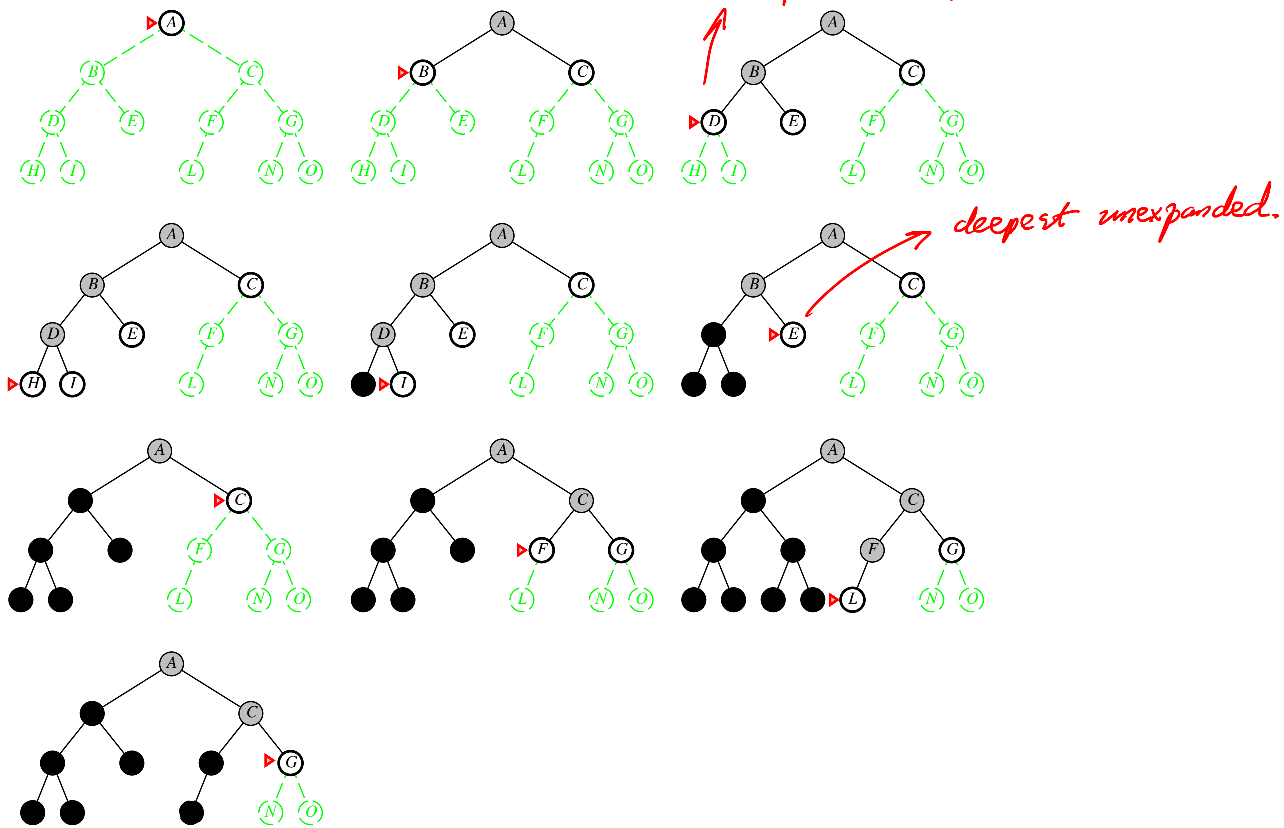
Depth-first search

Expand deepest unexpanded node

Implementation:

frontier = LIFO queue, i.e., put successors at front

deepest unexpanded



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

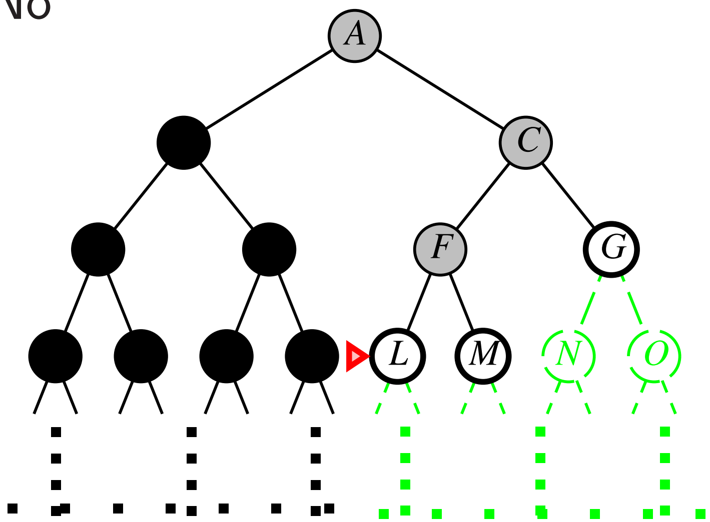
say if we going to a wrong way at first place.

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! (deepest node+ancestors+their siblings)

Optimal?? No



depth	#nodes
0	b^0
1	b^1
2	b^2
3	b^3
m	b^m

Breadth-first versus depth-first search

Use **breadth-first** search when there exists **short** solutions.

Use **depth-first** search when there exists **many** solutions.

Eternity II Puzzle
2 million dollar prize! Few deep solutions

Iterative deepening search = BFS + DFS

Combines advantages of breadth-first and depth-first search:

- completeness
- returns shallowest solution
- use linear amount of memory

Performs a **series of depth limited depth-first searches**

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** solution/failure
    inputs: *problem*, a problem

    for *depth* ← 0 **to** ∞ **do**
        result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
        if *result* $\neq$ cutoff **then return** *result*
    end

Depth-limited search

= depth-first search with depth limit l,
i.e., nodes at depth l have no successors

Recursive implementation:

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

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST(problem, STATE[node]) then return node
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

cutoff: no solution within the depth limit, failure: the problem has no solution

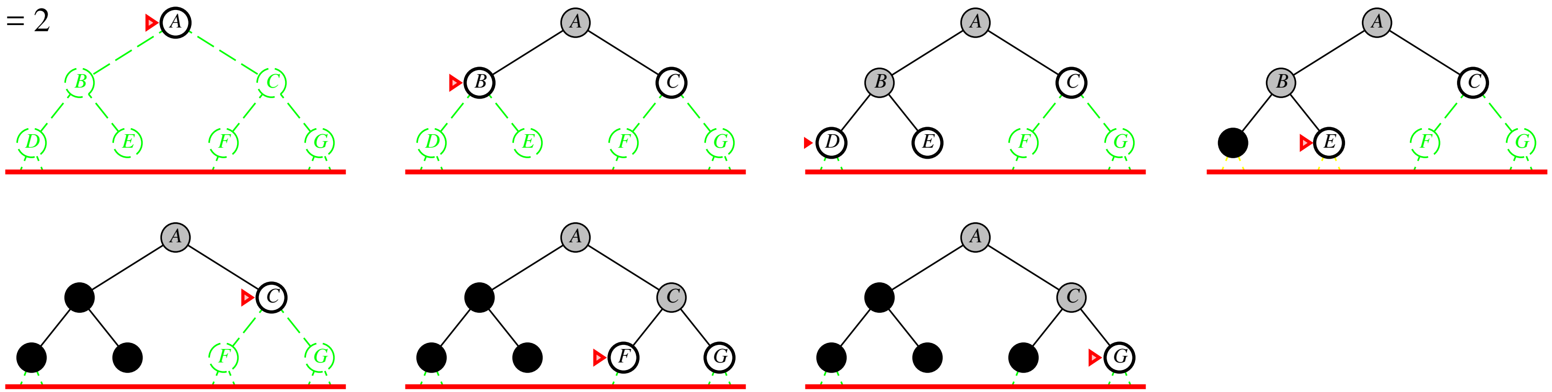
might have / not have a solution,
but just beyond our reach or costs too much.

Iterative deepening search

Limit = 0

Limit = 1

Limit = 2



Properties of iterative deepening search

Complete?? Yes (if b and d are finite)

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

Space?? $O(bd)$

Optimal?? Yes, if step cost $= 1$
Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

Time: IDS doesn't do much worse than BFS

$$\begin{aligned} N(\text{IDS}) &= 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456 \\ N(\text{BFS}) &= 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111 \end{aligned}$$

Assumes BFS applies the goal test when a node is **generated**

Space: IDS does much better N(IDS)$=$ 50, N(BFS) $\simeq$ 1,000,000

Summary of algorithms

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

Summary

◇ **Problem solving agents**
formulate a problem, search off-line for a solution, execute it

◇ **Problem formulation**
initial state, successor function, goal test, path cost

◇ **Example problems**
traveling around romania, 8-puzzle, power supply restoration

◇ **Tree search algorithms**
build and explore a tree, strategy picks up the order of node expansion

◇ **Implementation**
select the first node on the frontier, test for goal, expand

◇ **(Uninformed) strategies** (breadth first, uniform cost, ...)
characterised by their completeness, optimality, complexity

◇ **Iterative deepening** complete, finds the shallowest solution
uses only linear space and no more time than uninformed strategies