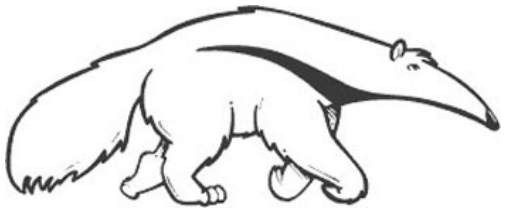
State Space Representations and Search Algorithms

CS171, Fall Quarter, 2019
Introduction to Artificial Intelligence
Prof. Richard Lathrop



Read Beforehand: R&N 3.1-3.4

Architectures for Intelligence

- Search?
  - Determine how to achieve some goal; “what to do”
- Logic & inference?
  - Reason about what to do
  - Encode knowledge / “expert” systems
  - Know what to do
- Learning?
  - Learn what to do
- Modern view: complex & multi-faceted

Search

- Formulate “what to do” as a search problem
  - Solution tells the agent what to do
- If no solution in the current search space?
  - Find space that does contain a solution (use search!)
  - Solve original problem in new search space
- Many powerful extensions to these ideas
  - Constraint satisfaction; planning; game playing; …
- Human problem-solving often looks like search

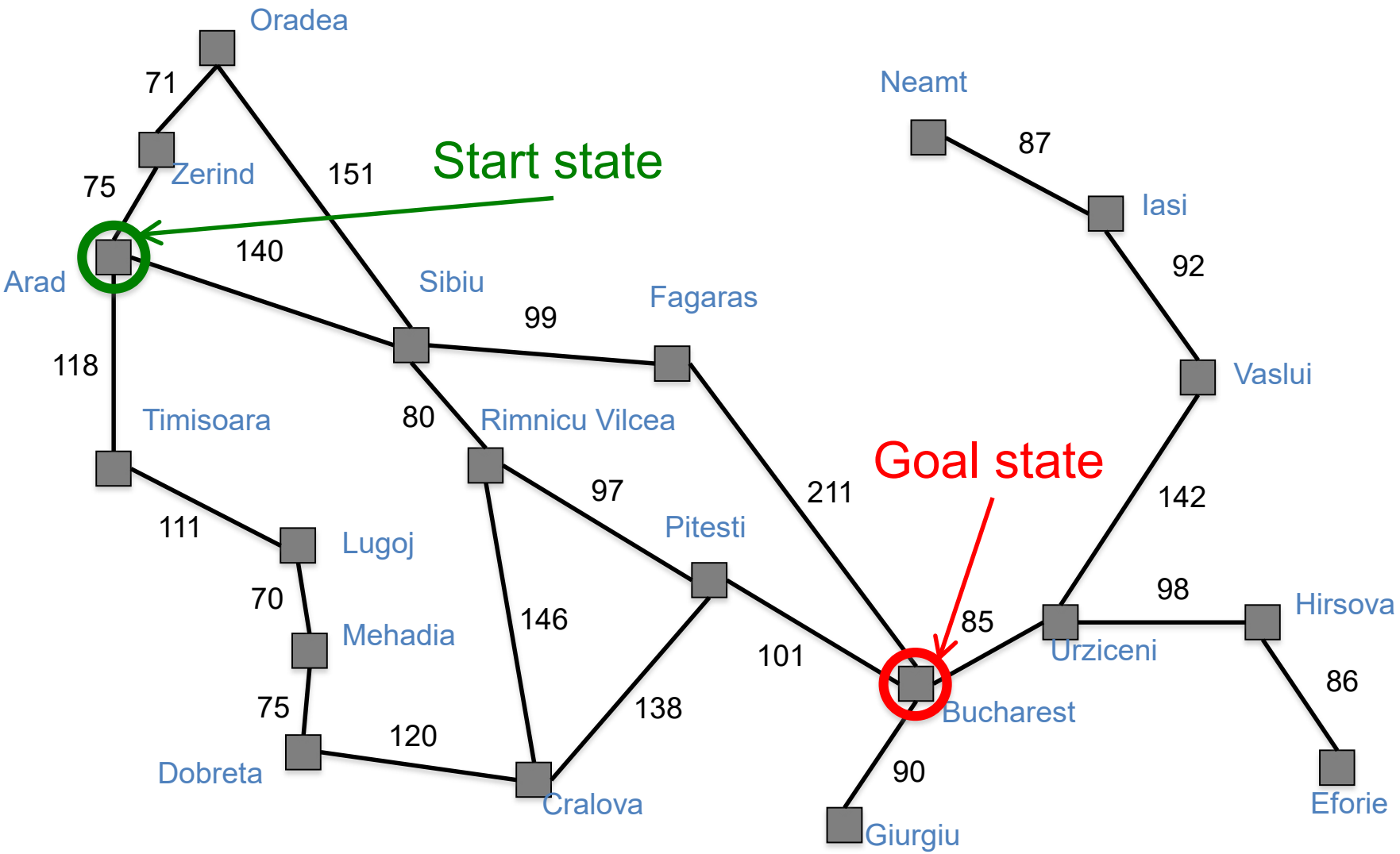
Why search?

- Engaged in bigger, important problem
  - Hit a search subproblem we need to solve
  - Search, solve it, get back to original problem
- Predict the results of our actions in the future
- Many sequences of actions, each with some utility
  - Maximize performance measure
- Want to achieve some goal by some (any?) means
- Or, find the best (optimal) way to achieve it

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 / choose next city
- Find a solution:
  - Sequence of cities, e.g.: Arad, Sibiu, Fagaras, Bucharest

Example: Romania



Environment types

Classifying the environment:

- Static / Dynamic

  Previous problem was static: no attention to changes in environment

- Deterministic / Stochastic

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

- Observable / Partially Observable / Unobservable

  Previous problem was observable: agent knew the initial state, etc.

- Discrete / Continuous

  Previous problem was discrete: we can enumerate all possibilities

Why not Dijkstra's Algorithm?

- D's algorithm inputs the entire graph
  - Want to search in unknown spaces
  - Combine search with "exploration"
  - Ex: autonomous rover on Mars must search an unknown space
- D's algorithm takes connections as given
  - Want to search based on agent's actions, w/ unknown connections
  - Ex: web crawler may not know what connections are available on a URL before visiting it
  - Ex: agent may not know the result of an action before trying it
- D's algorithm won't work on infinite spaces
  - Many actions spaces are infinite or effectively infinite
  - Ex: logical reasoning space is infinite
  - Ex: real world is essentially infinite to a human-size agent

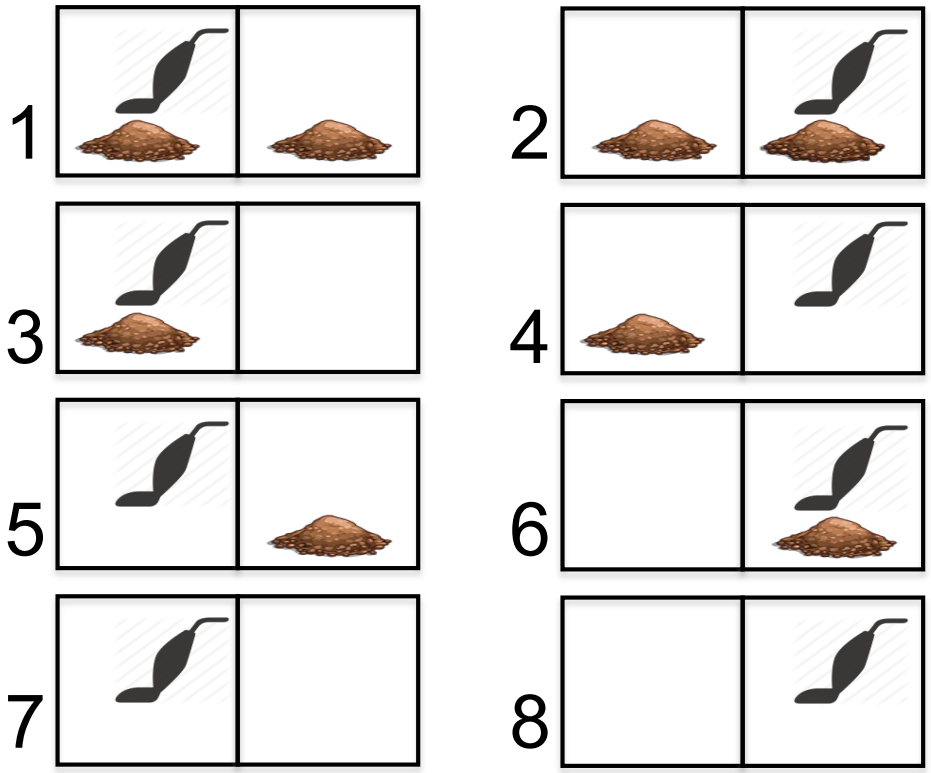
The State-Space Graph

- **Graphs:**
  - vertices (nodes), edges (arcs), directed arcs, paths
- **State-space graphs:**
  - States are vertices
    - Initial state (start state), possibly multiple goal states
  - Actions are directed arcs (carry state to state[s] that result from action)
- **Solution:**
  - A path from the start state to any goal state
  - May desire an optimal path (= lowest cost or highest value)
- **Problem solving activity:**
  - Generate a part of the search space that contains a solution
  - May desire an optimal path (= lowest cost or highest value)

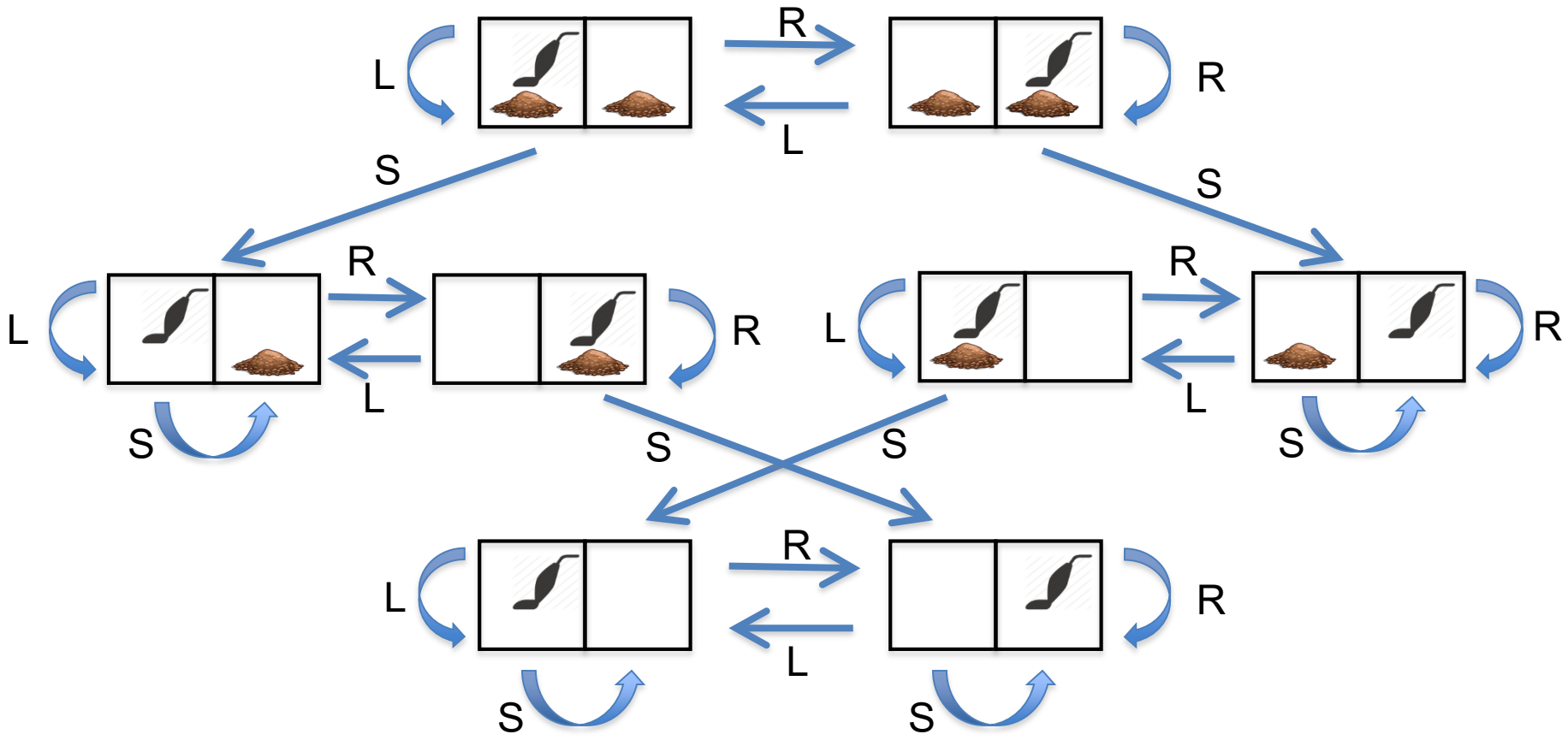
Example: Vacuum World

- Observable, start in #5.
- Solution?

  [Right, Suck]



Vacuum world state space graph



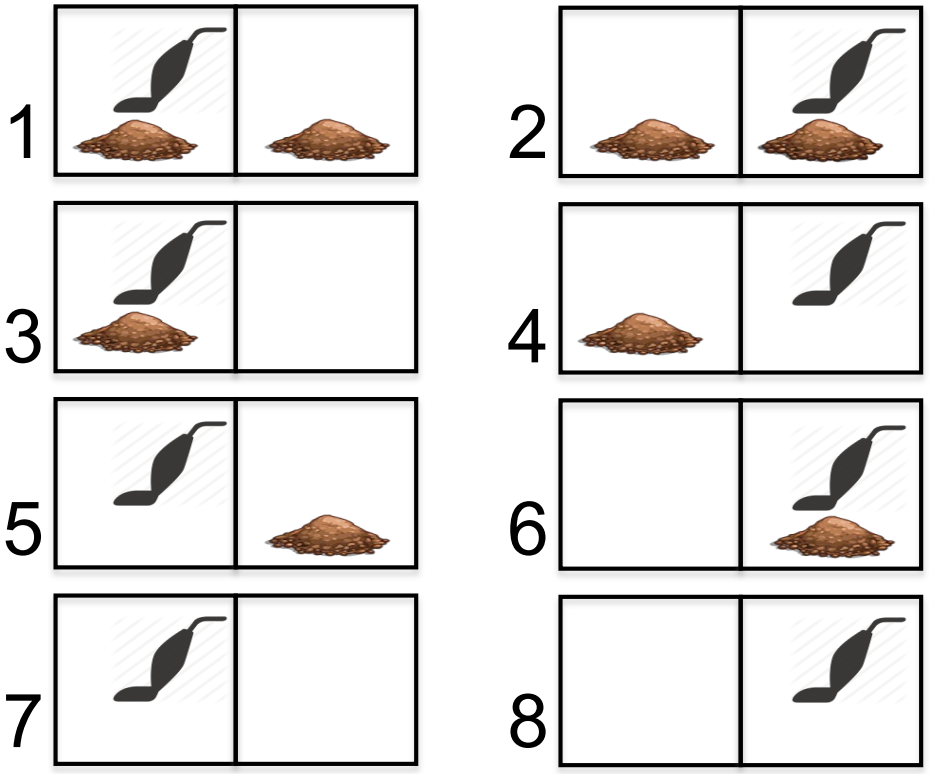
Example: Vacuum World (Unobservable)

- Unobservable

start in {1,2,3,4,5,6,7,8}

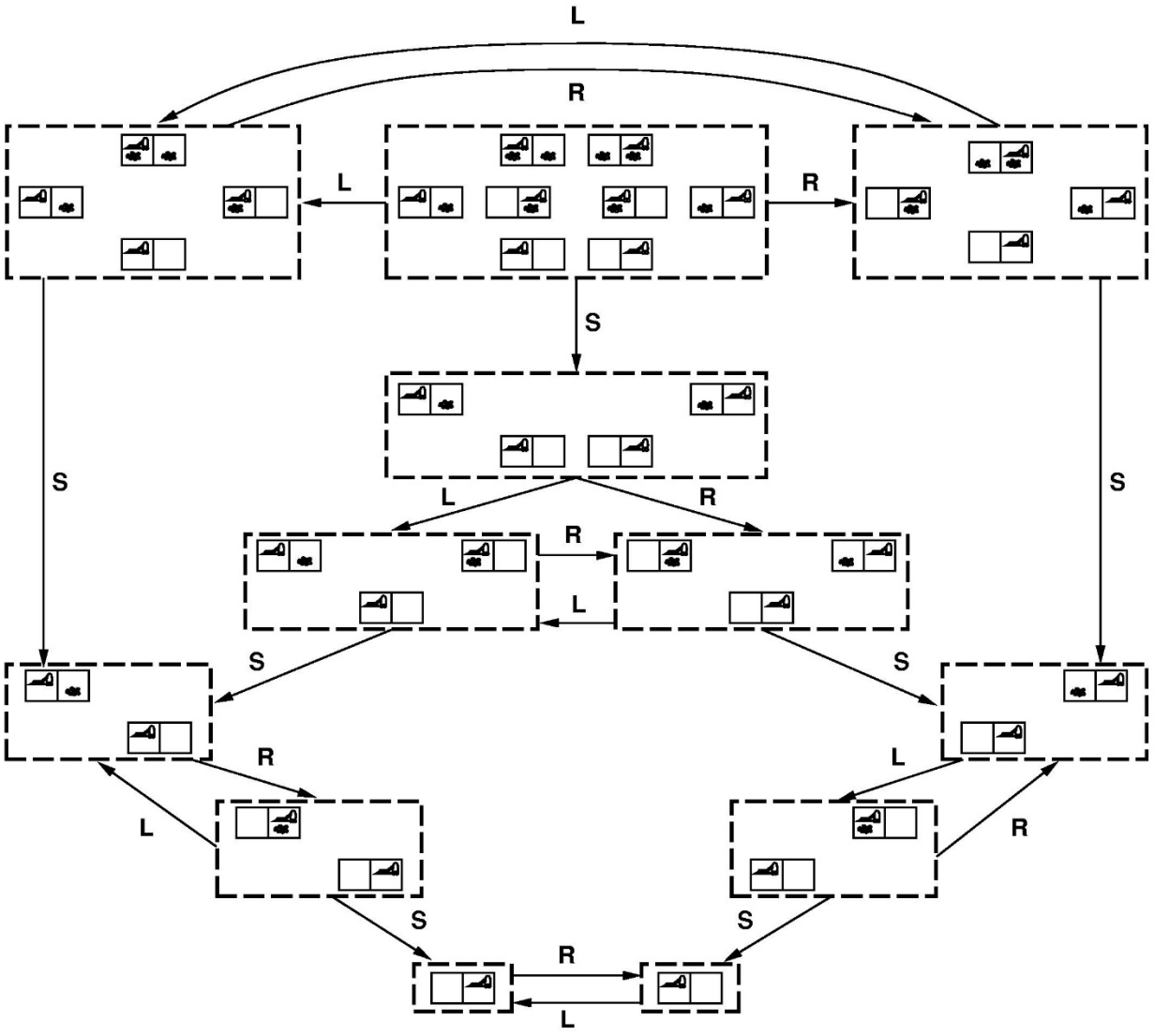
- Solution?

[Right, Suck, Left, Suck]



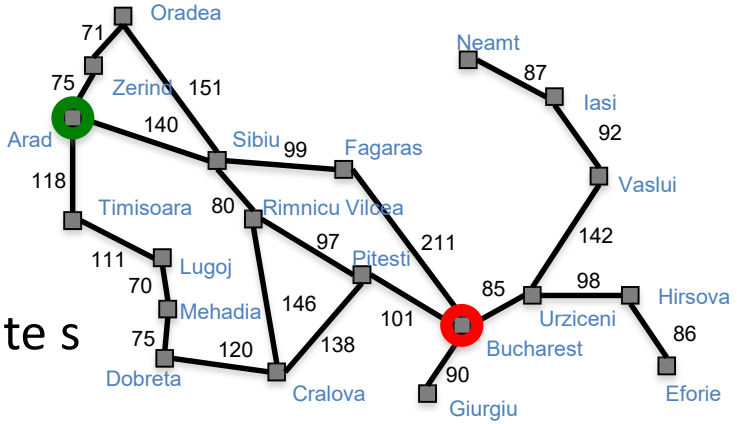
Vacuum world belief-state space

R&N Fig. 4.14



State-Space Problem Formulation

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



(1) initial state e.g., "at Arad"

(2) actions *Actions(s)* = set of actions avail. in state s

(3) transition model Results(s,a) = state that results from action a in state s

Alt: successor function *S(x)* = set of action–state pairs

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

(4) goal test, (or goal state)

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

(5) 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 (and often, assumed to be $\geq \varepsilon > 0$)

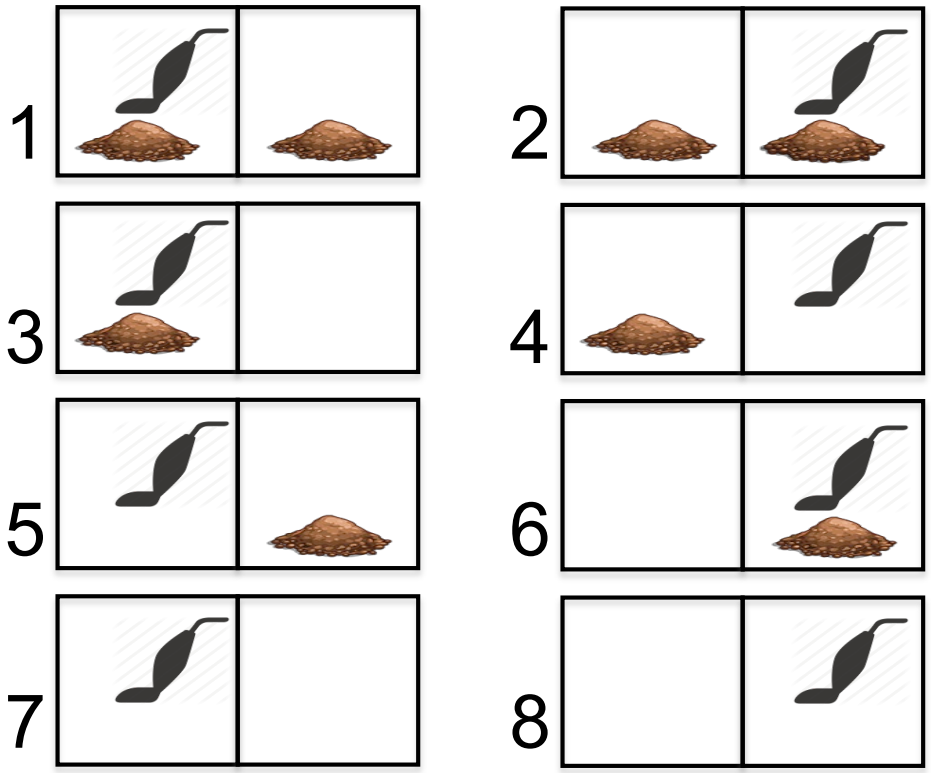
A **solution** is a sequence of actions leading from the initial state to a goal state

Selecting a state space

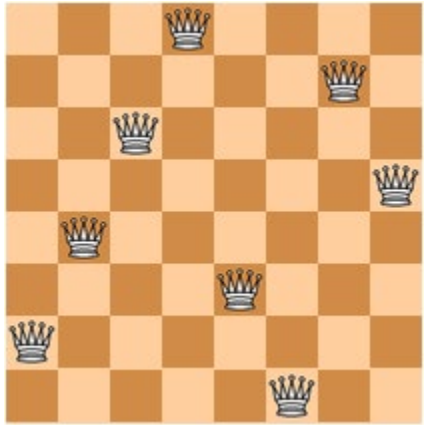
- Real world is absurdly complex
  → state space must be abstracted for problem solving
- (Abstract) state ← set of real states
- (Abstract) action ← complex combination of real actions
  - e.g., "Arad → Zerind" represents a complex set of possible routes, detours, rest stops, etc.
- For guaranteed realizability, 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

- States?
  - Discrete: dirt, location
- Initial state?
  - Any
- Actions?
  - Left, Right, Suck
- Goal test?
  - No dirt at all locations
- Path cost?
  - 1 per action



Example: 8-Queens



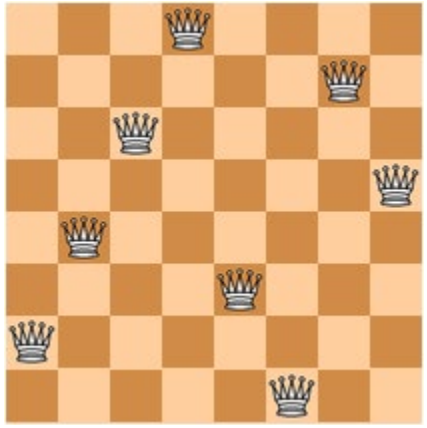
Place as many queens as possible
on the chess board without capture

- states?

- initial state?
- actions?

- goal test?
- path cost?

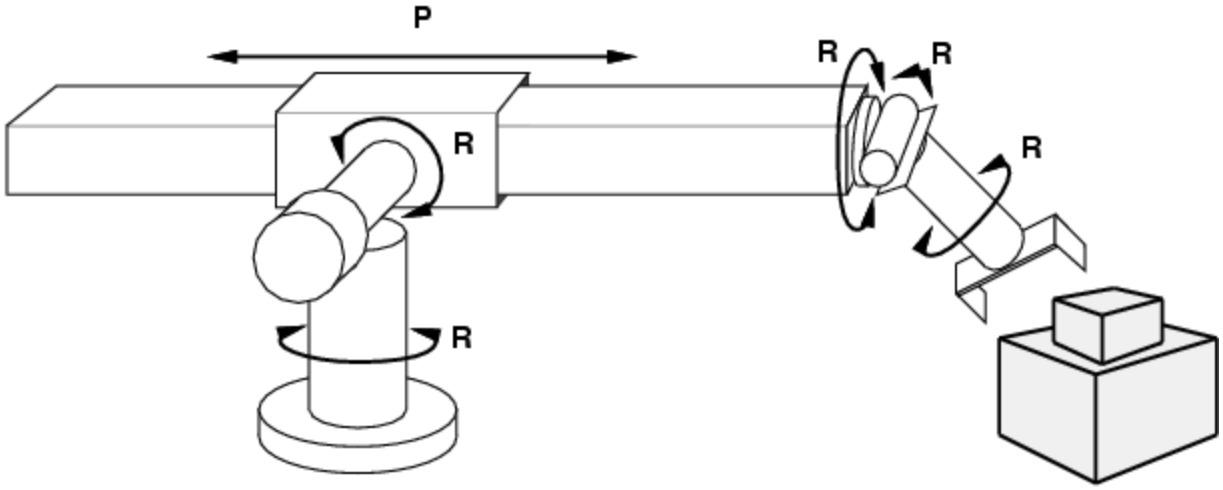
Example: 8-Queens



Place as many queens as possible
on the chess board without capture

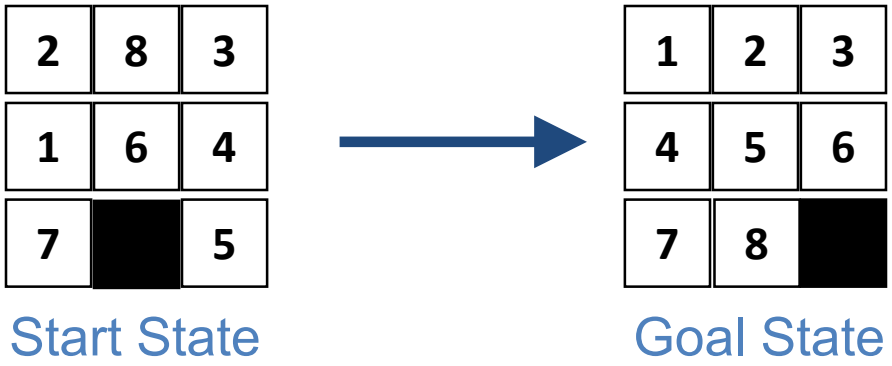
- states? - any arrangement of n<=8 queens
  - *or* arrangements of n<=8 queens in leftmost n columns, 1 per column, such that no queen attacks any other.
- initial state? no queens on the board
- actions? - add queen to any empty square
  - *or* add queen to leftmost empty square such that it is not attacked by other queens.
- goal test? 8 queens on the board, none attacked.
- path cost? 1 per move (not relevant…)

Example: Robotic assembly



- states? Real-valued coordinates of robot joint angles & parts of the object to be assembled
- initial state? Rest configuration
- actions? Continuous motions of robot joints
- goal test? Complete assembly
- path cost? Time to execute & energy used

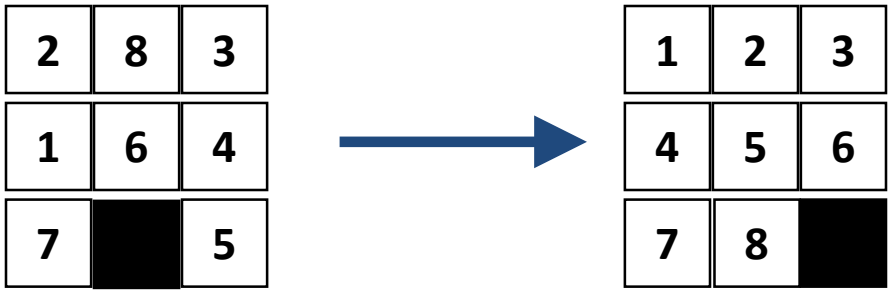
Example: Sliding tile puzzle



- states?
- initial state?
- actions?
- goal test?
- path cost?

Try it yourselves…

Example: Sliding tile puzzle



- states? Locations of tiles
- initial state? Given
- actions? Move blank square up / down / left / right
- goal test? Goal state (given)
- path cost? 1 per move

# of states: (n+1)! / 2
8-puzzle: 181,440 states
15-puzzle: 1.3 trillion
24-puzzle: 10^25

Optimal solution of
n-Puzzle family is NP-hard

Importance of representation

- Definition of “state” can be very important
- A good representation
  - Reveals important features
  - Hides irrelevant detail
  - Exposes useful constraints

  } **Most important**

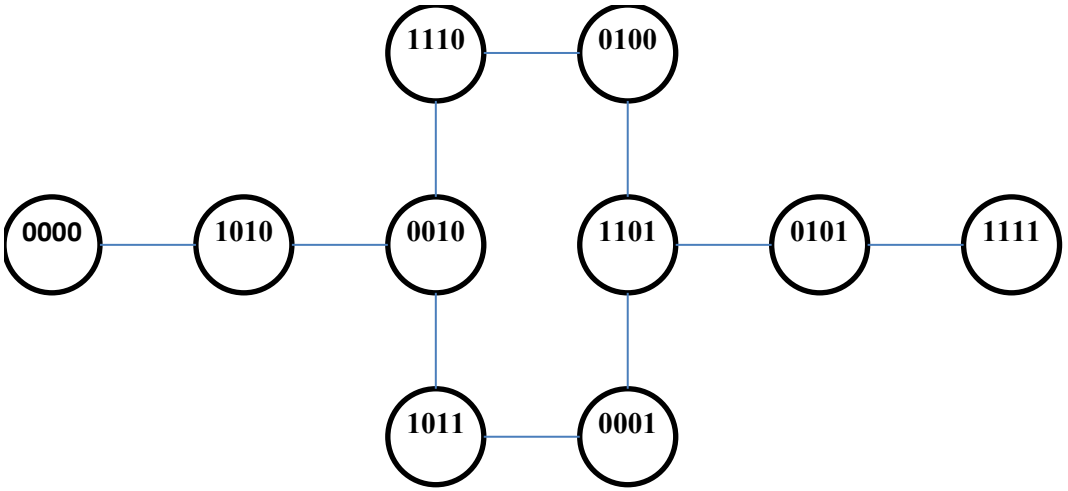
  - Makes frequent operations easy to do
  - Supports local inferences from local features
    - Called “soda straw” principle, or “locality” principle
    - Inference from features “through a soda straw”
  - Rapidly or efficiently computable
    - It’s nice to be fast

Reveals important features
Hides irrelevant detail

•**In search:** *A man is traveling to market with a fox, a goose, and a bag of oats. He comes to a river. The only way across the river is a boat that can hold the man and exactly one of the fox, goose or bag of oats. The fox will eat the goose if left alone with it, and the goose will eat the oats if left alone with it.*

How can the man get all his possessions safely across the river?

•**A good representation makes this problem easy:**



MFGO

M = man
F = fox
G = goose
O = oats
0 = starting side
1 = ending side

Exposes useful constraints

- **In logic:**

If the unicorn is mythical, then it is immortal, but if it is not mythical, then it is a mortal mammal. If the unicorn is either immortal or a mammal, then it is horned. The unicorn is magical if it is horned.

⇒ ***Prove that the unicorn is both magical and horned.***

- **A good representation makes this problem easy:**

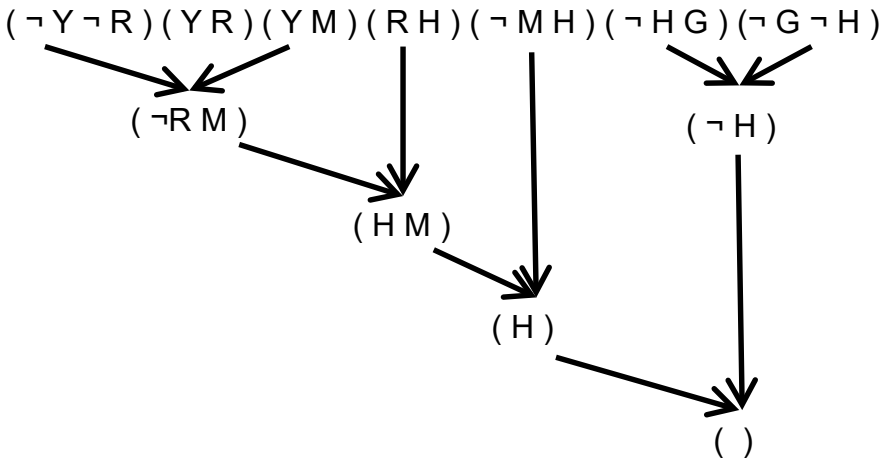
Y = unicorn is m<u>Y</u>thical

R = unicorn is mo<u>R</u>tal

M = unicorn is a ma<u>M</u>mal

H = unicorn is <u>H</u>orned

G = unicorn is ma<u>G</u>ical



Makes frequent operations easy-to-do

- **Roman numerals**
  - M=1000, D=500, C=100, L=50, X=10, V=5, I=1
  - 2000 = MM; 1776 = MDCCLXXVI; 16 = XVI; 111 = CXI

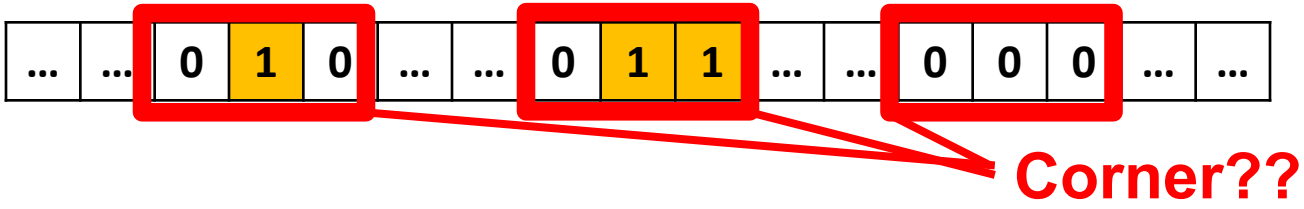
- Long division is **very tedious** (try MDCCLXXVI / XVI = CXI)
- Testing for N < 1000 is very easy (first letter is not “M”)

- **Arabic numerals**
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, “.”

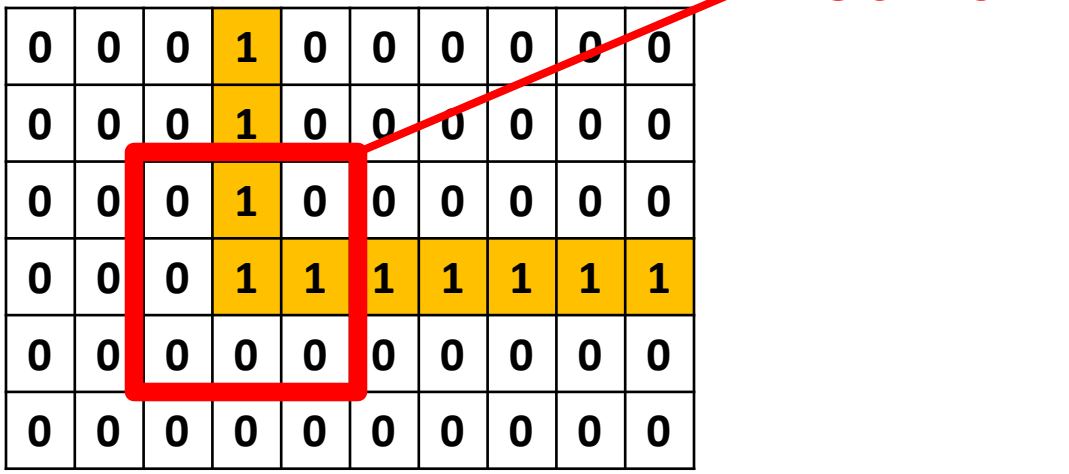
- Long division is **much easier** (try 1776 / 16 = 111)
- Testing for N < 1000 is slightly harder (have to scan the string to test for three or fewer digits)

Local inferences from local features

- Linear vector of pixels
  = highly non-local inference for vision



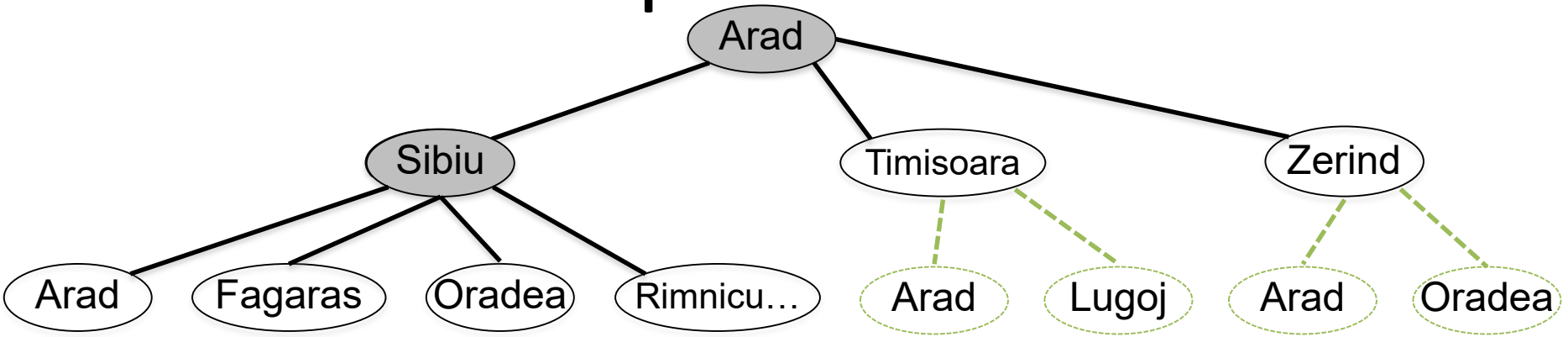
- Rectangular array of pixels
  = local inference for vision

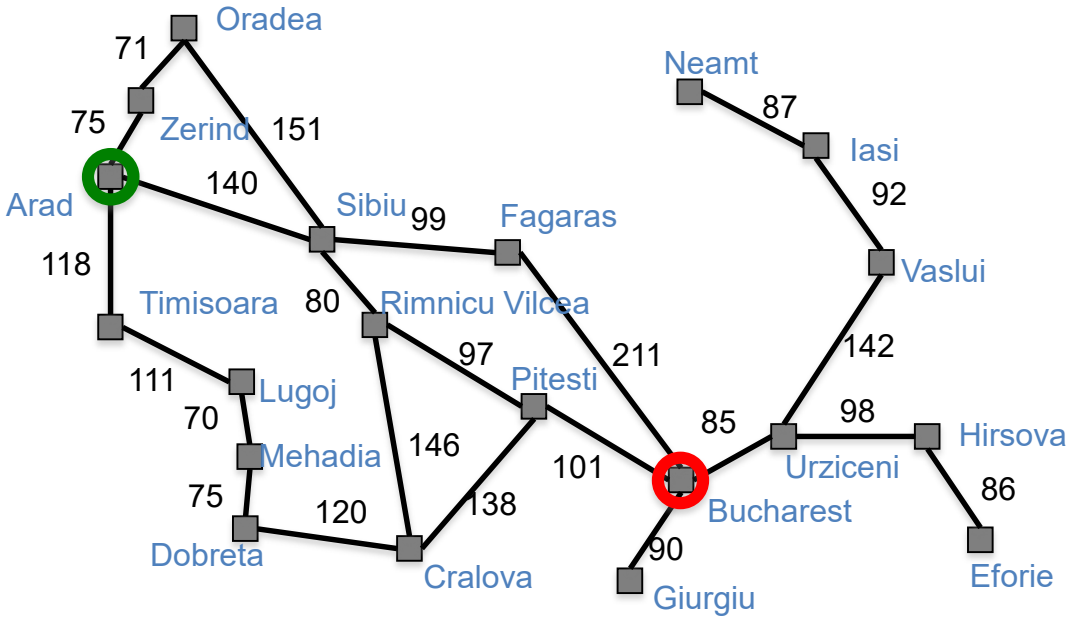


Tree search algorithms

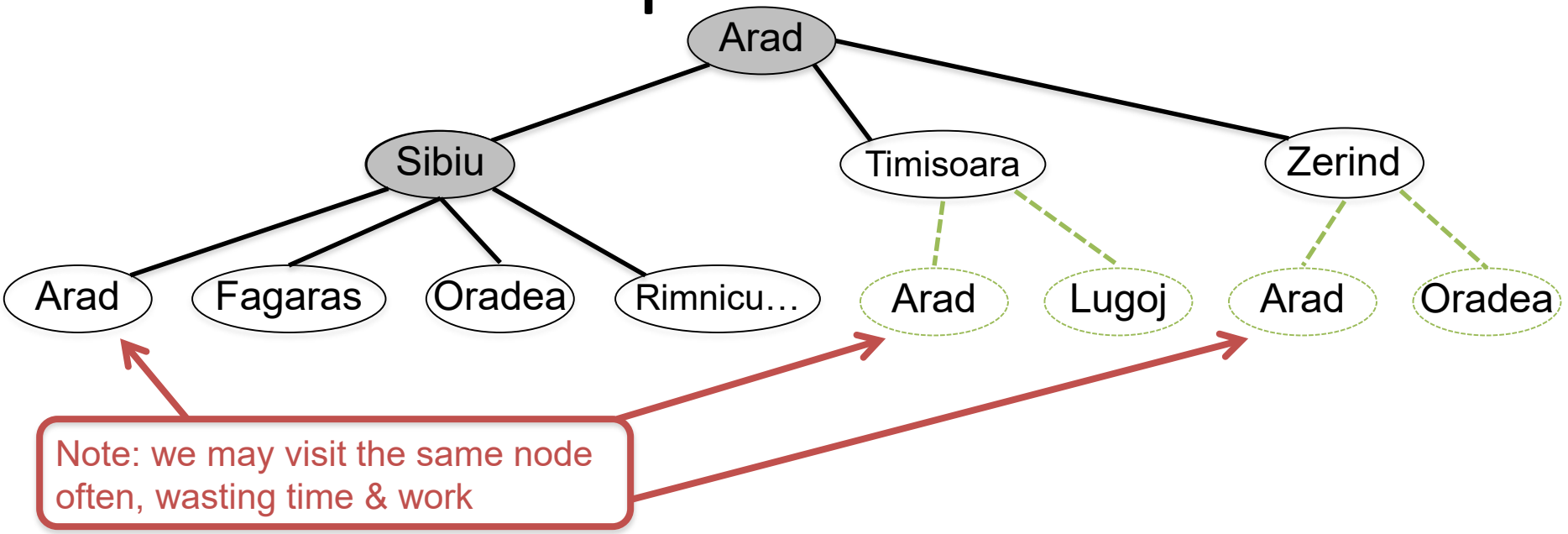
- Basic idea
  - Explore space by generating successors of already-explored states ("expanding" states)
  - Evaluate every generated state: *is it a goal state*?

Example: Romania





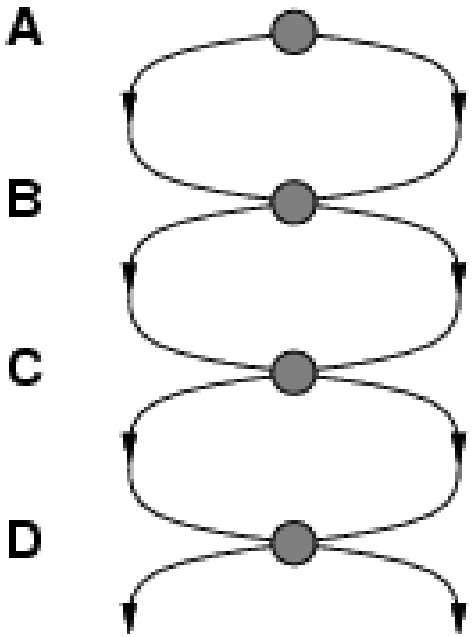
Example: Romania

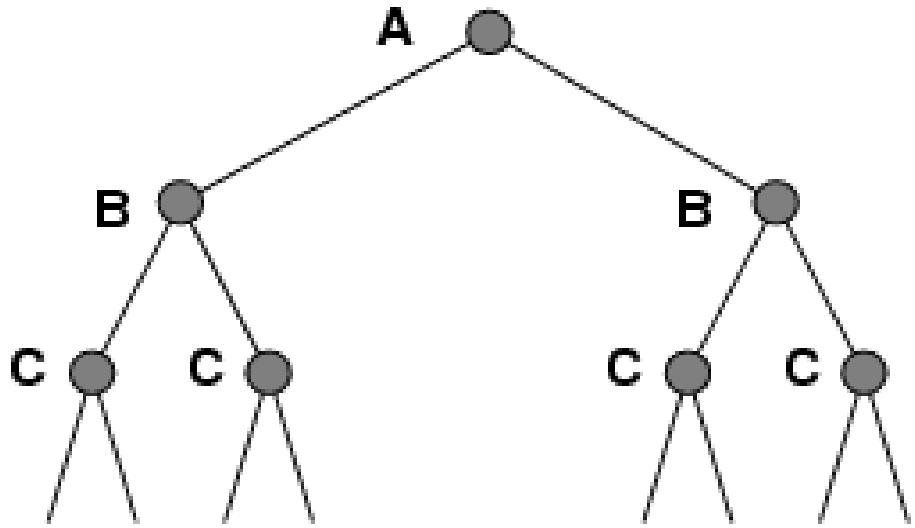


function TREE-SEARCH (*problem*, *strategy*) : returns a solution or failure
  initialize the search tree using the initial state of *problem*
  while (true):
    if no candidates for expansion: return failure
    choose a leaf node for expansion according to *strategy*
    if the node contains a goal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

Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!
- Test is often implemented as a hash table.

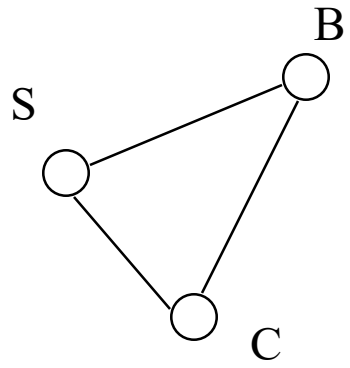




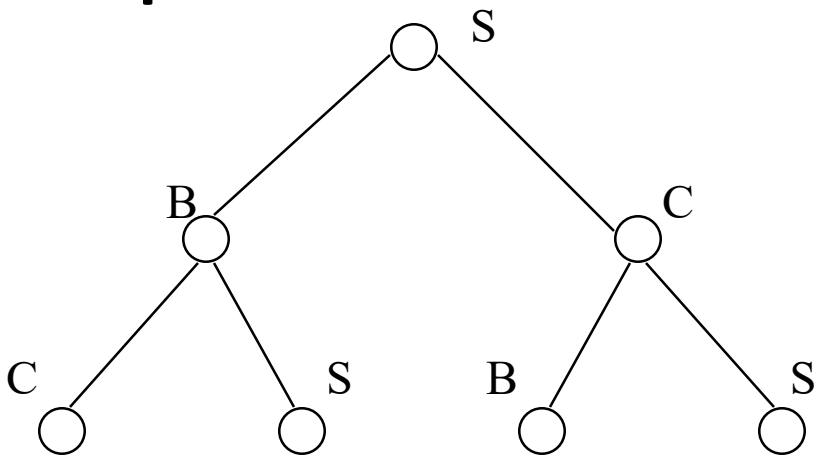
Tree search vs. Graph search
Review Fig. 3.7, p. 77

- What R&N call Tree Search vs. Graph Search
  - (And we follow R&N **exactly** in this class)
  - Has **NOTHING** to do with searching trees vs. graphs
- **Tree Search** = do **NOT** remember visited nodes
  - Exponentially slower search, but memory efficient
- **Graph Search** = **DO** remember visited nodes
  - Exponentially faster search, but memory blow-up
- **CLASSIC Comp Sci TIME-SPACE TRADE-OFF**

Solutions to Repeated States



State Space



Example of a Search Tree

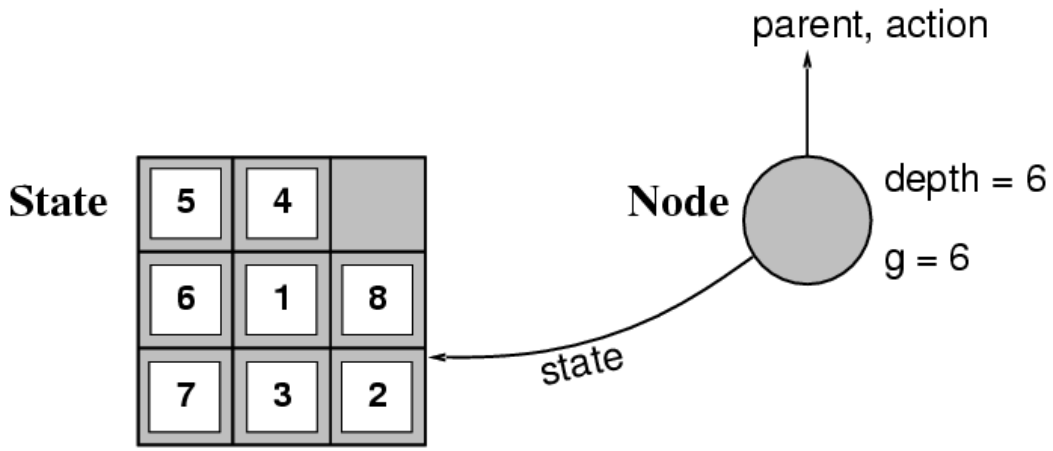
- Graph search ← faster, but memory inefficient
  - never generate a state generated before
    - must keep track of all possible states (uses a lot of memory)
    - e.g., 8-puzzle problem, we have 9!/2 = 181,440 states
    - approximation for DFS/DLS: only avoid states in its (limited) memory: avoid infinite loops by checking path back to root.
  - "visited?" test usually implemented as a hash table

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: $b^d = 10^{10} = 10{,}000{,}000{,}000$

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.

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 ∞)

Summary

- Generate the search space by applying actions to the initial state and all further resulting states.
- Problem: initial state, actions, transition model, goal test, step/path cost
- Solution: sequence of actions to goal
- Tree-search (don't remember visited nodes) vs. Graph-search (do remember them)
- Search strategy evaluation: b, d, m
  - Complete? Time? Space? Optimal?