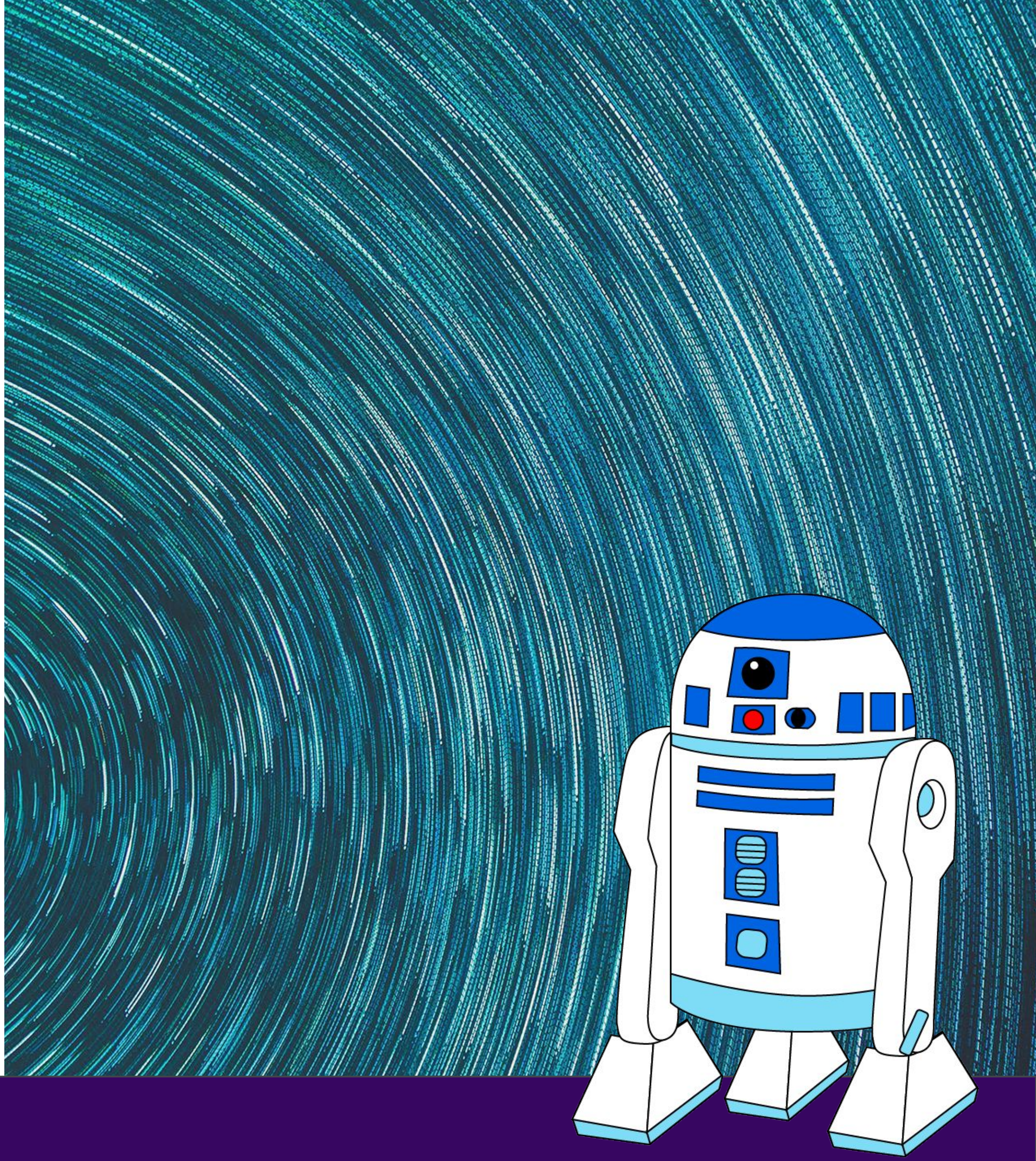
CIS 421/521:
ARTIFICIAL INTELLIGENCE

Search Problems

Professor Chris Callison-Burch



Penn Engineering
UNIVERSITY of PENNSYLVANIA

Problem Solving Agents & Problem Formulation

AIMA 3.1-3.3

Reflex Agents

A simple reflex agent is one that selects an action based only on the **current percept**.

It **ignores** the rest of the **percept history**.

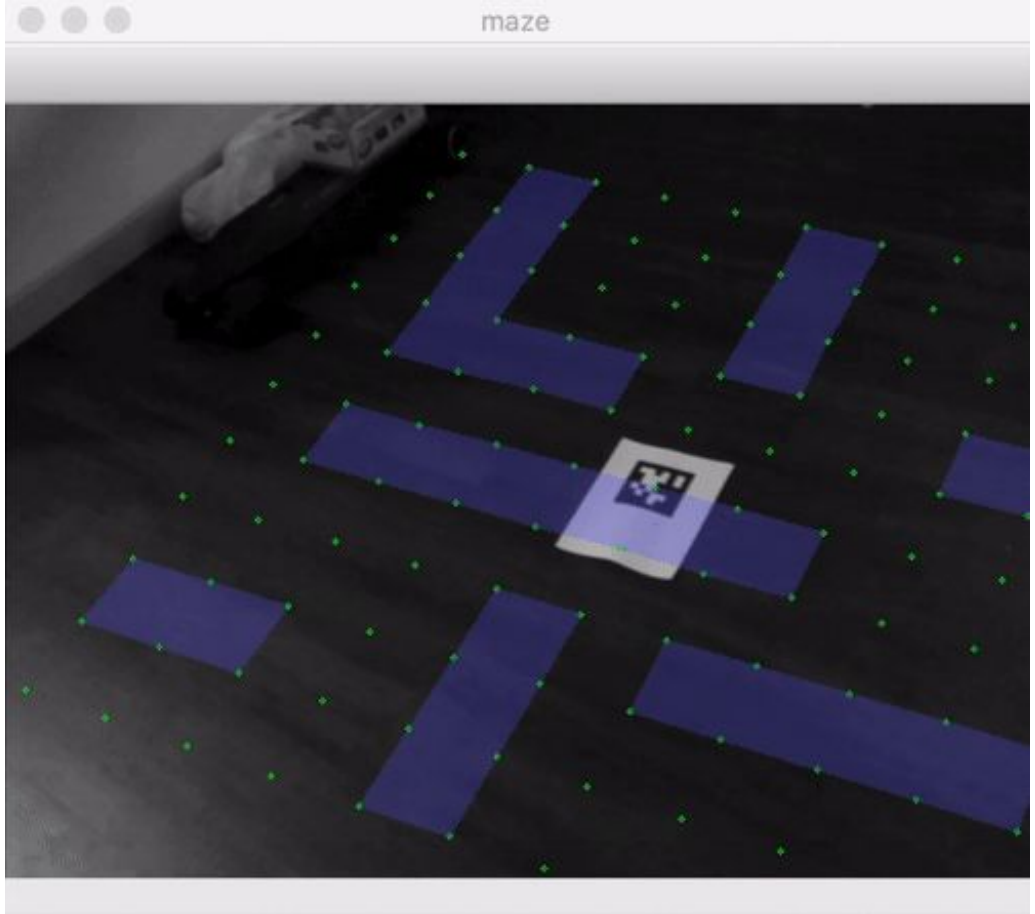
Problem-Solving Agent

A problem-solving agent must **plan ahead**.

The computational process that it undertakes is called **search**.

It will consider a **sequence of actions** that form a **path** to a **goal state**.

Such a sequence is called a **solution.**



Impact of Task Environments

The properties of the task environments change the types of solutions that we need.

If an environment is:

- **Fully observable**
- **Deterministic**
- **Known environment**

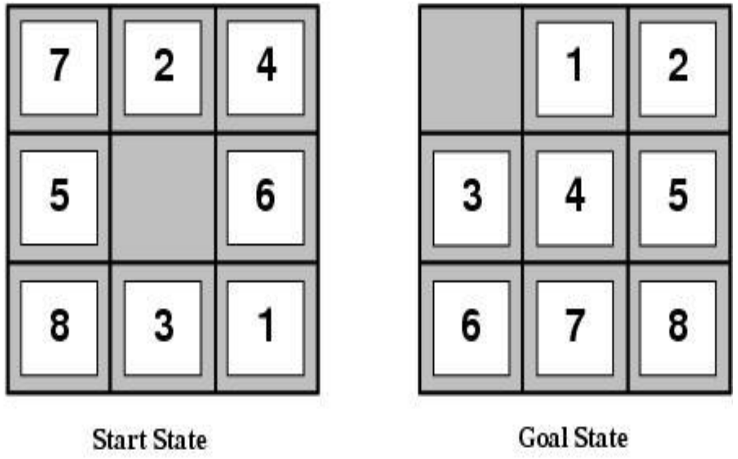
The solution to any problem in such an environment is a fixed sequence of actions.

In environments that are

- **Partially observable** or
- **Nondeterministic**

The solution must recommend different future actions depending on the what percepts it receives. This could be in the form of a *branching strategy.*

Example search problem: 8-puzzle



Formulate ***goal***

- Pieces to end up
  in order
  as shown…

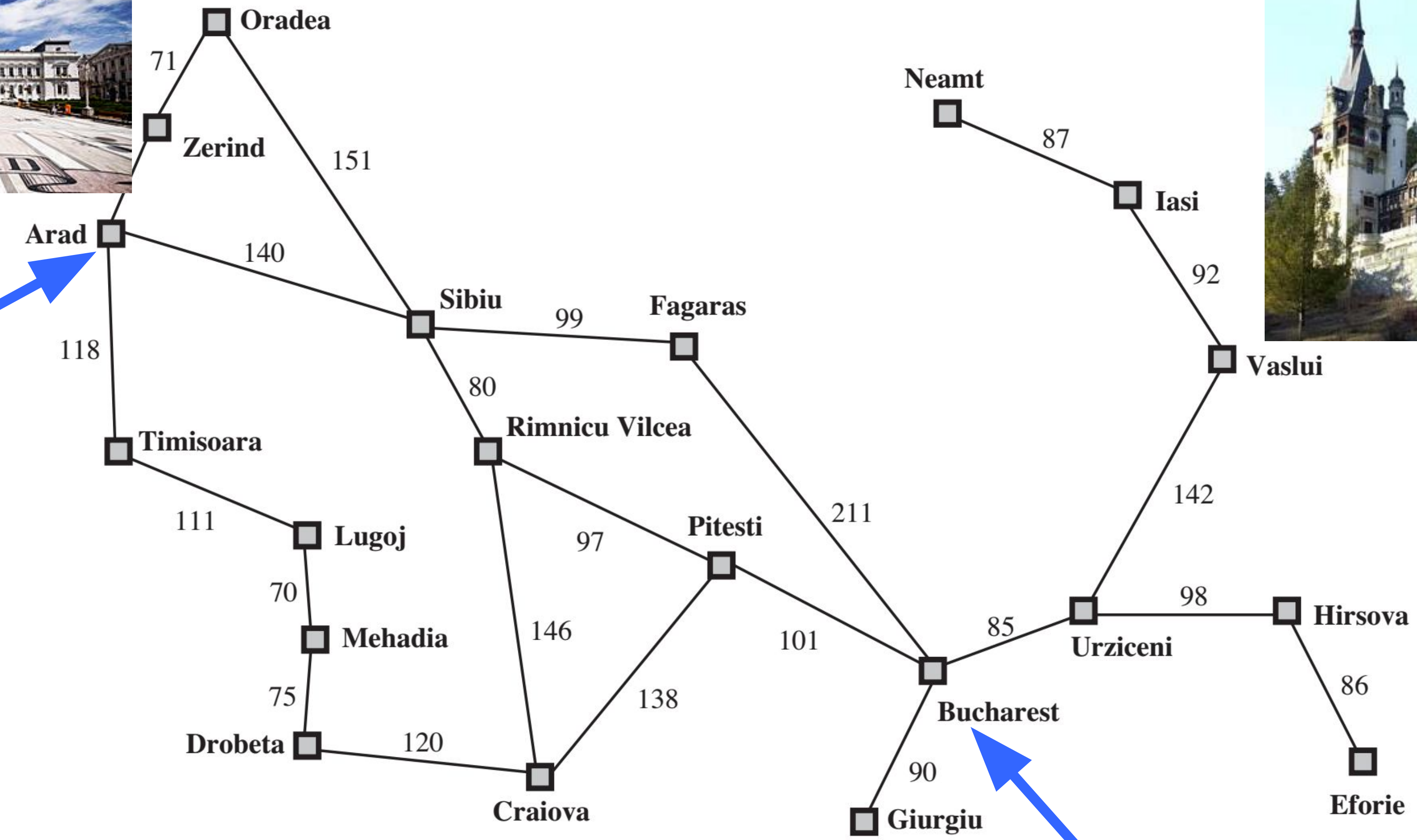
Formulate ***search problem***

- ***States:*** configurations of the puzzle (9! configurations)
- ***Actions***: Move one of the movable pieces (≤4 possible)
- ***Performance measure***: minimize total moves

Find ***solution***

- Sequence of pieces moved: 3,1,6,3,1,…

Example search problem: Holiday in Romania



You are here

You need to be here

Holiday in Romania

On holiday in Romania; currently in Arad

- Flight leaves tomorrow from Bucharest

Formulate ***goal***

- Be in Bucharest

Formulate ***search problem***

- States: various cities
- Actions: drive between cities
- Performance measure: minimize travel time / distance

Find ***solution***

- Sequence of cities; e.g. Arad, Sibiu, Fagaras, Bucharest, …

More formally, a problem is defined by:

1. **States:** a set S
2. *An **initial state*** $s_i \in S$
3. ***Actions:*** *a set* A

   $\forall\, s\ Actions(s)$ = *the set of actions that can be executed in* s, *that are applicable* in s.

4. ***Transition Model:*** $\forall\, s\ \forall\, a \in Actions(s)\ Result(s, a) \rightarrow s_r$

   s_r *is called a successor of* s

   $\{s_i\} \cup Successors(s_i)^*$ *= state space*

5. ***Path cost*** *(Performance Measure):* Must be additive, e.g. sum of distances, number of actions executed, …

   $c(x,a,y)$ is the step cost, assumed ≥ 0

   - (where action a goes from state x to state y)

6. ***Goal test:*** $Goal(s)$

   Can be implicit, e.g. $checkmate(s)$

   s is a *goal state* if $Goal(s)$ is *true*

Vacuum World

States: A state of the world says which objects are in which cells.

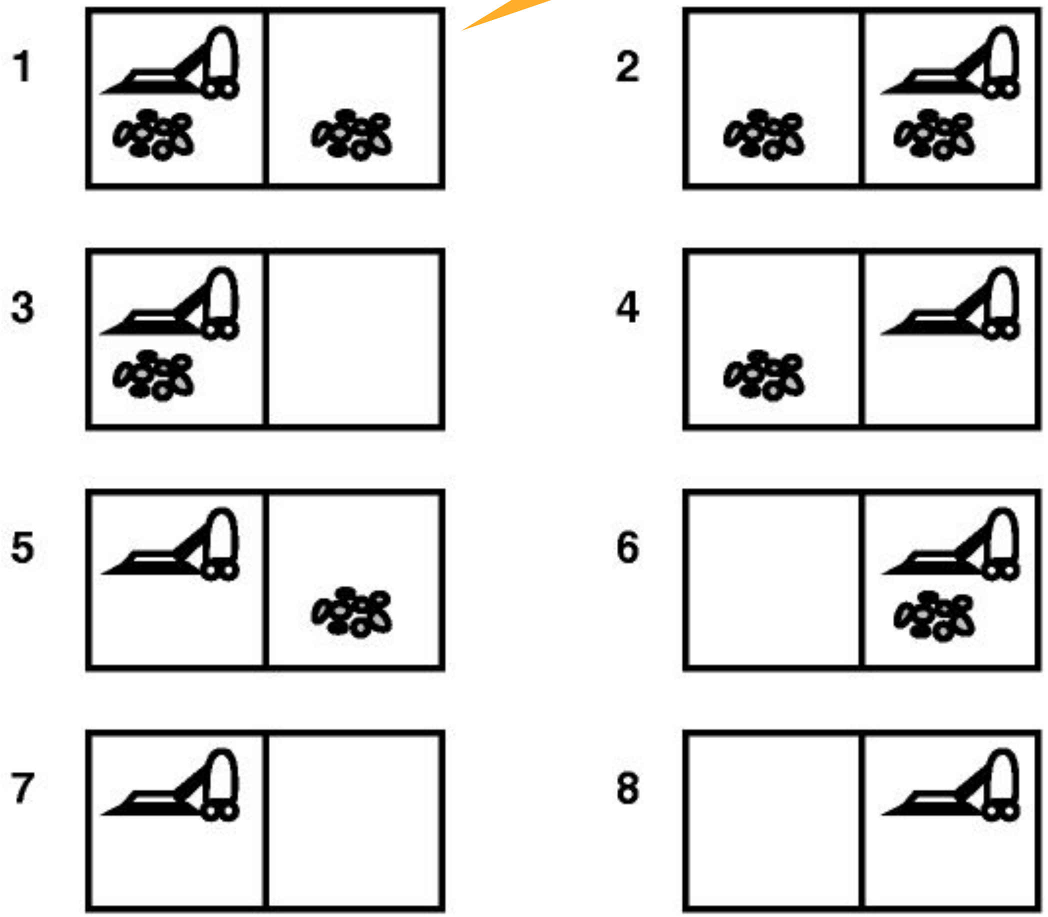
In a simple two cell version,

- the agent can be in either cell
- each cell can have dirt or not

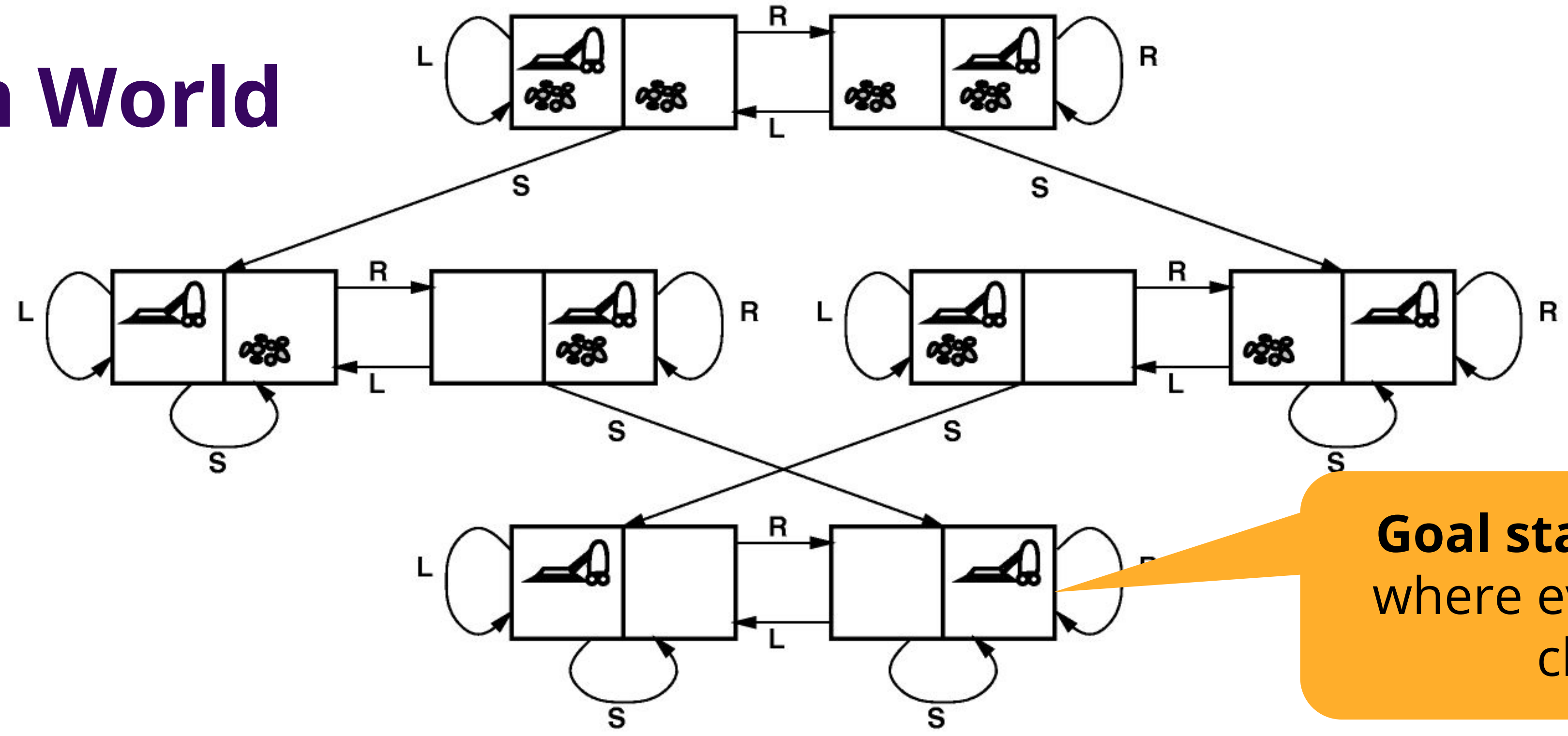
2 cells * 2 positions for agent * 2 possibilities for dirt = 8 states.

With *n* cells, there are $n*2^n$ states.

One state is designated as the initial state



Vacuum World



Goal states: States where everything is clean.

Actions:

- *Suck*
- *Move Left*
- *Move Right*
- *(Move Up)*
- *(Move Down)*

Action Cost: Each action costs 1

Transition:

Suck – removes dirt

Move – moves in that direction, unless agent hits a wall, in which case it stays put.

Solutions & Optimal Solutions

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

Optimal Solution: **A solution is optimal if no solution has a lower path cost.**

Art: Formulating a Search Problem

Decide:

Which properties matter & how to represent

- *Initial State, Goal State, Possible Intermediate States*

Which actions are possible & how to represent

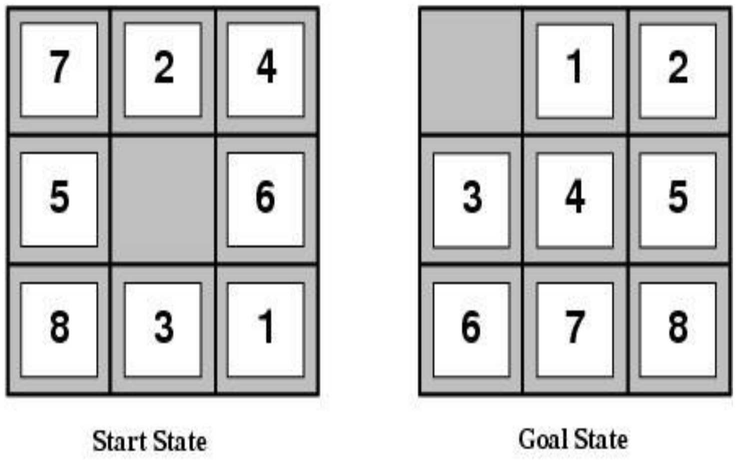
- *Operator Set: Actions and Transition Model*

Which action is next

- *Path Cost Function*

Formulation greatly affects combinatorics of search space and therefore speed of search

Example: 8-puzzle



States? List

Initial state?

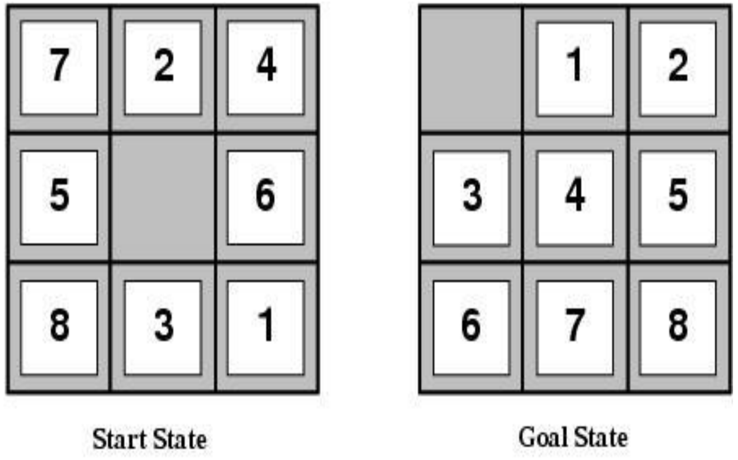
Actions? *{Left*

Transition Model?

Goal test? Che

Path cost? Nur

Example: 8-puzzle



States? List of 9 locations- e.g., [7,2,4,5,-,6,8,3,1]

Initial state? [7,2,4,5,-,6,8,3,1]

Actions? *{Left, Right, Up, Down}*

Transition Model? ...

Goal test? Check if goal configuration is reached

Path cost? Number of actions to reach goal

Hard subtask: Selecting a state space

Real world is absurdly complex

State space must be ***abstracted*** for problem solving

(abstract) ***State*** = set (equivalence class) of real-world states

(abstract) ***Action*** = equivalence class of combinations of real-world actions

- e.g. *Arad* $\rightarrow$ *Zerind* represents a complex set of possible routes, detours, rest stops, etc
- The abstraction is valid if the path between two states is reflected in the real world

Each abstract action should be "easier" than the real problem

Useful Concepts

State space: the set of all states reachable from the initial state by *any* sequence of actions

- *When several operators can apply to each state, this gets large very quickly*
- *Might be a proper subset of the set of configurations*

Path: a sequence of actions leading from one state s_j to another state s_k

Frontier: those states that are available for *expanding* (for applying legal actions to)

Solution: a path from the initial state s_i to a state s_f that satisfies the goal test

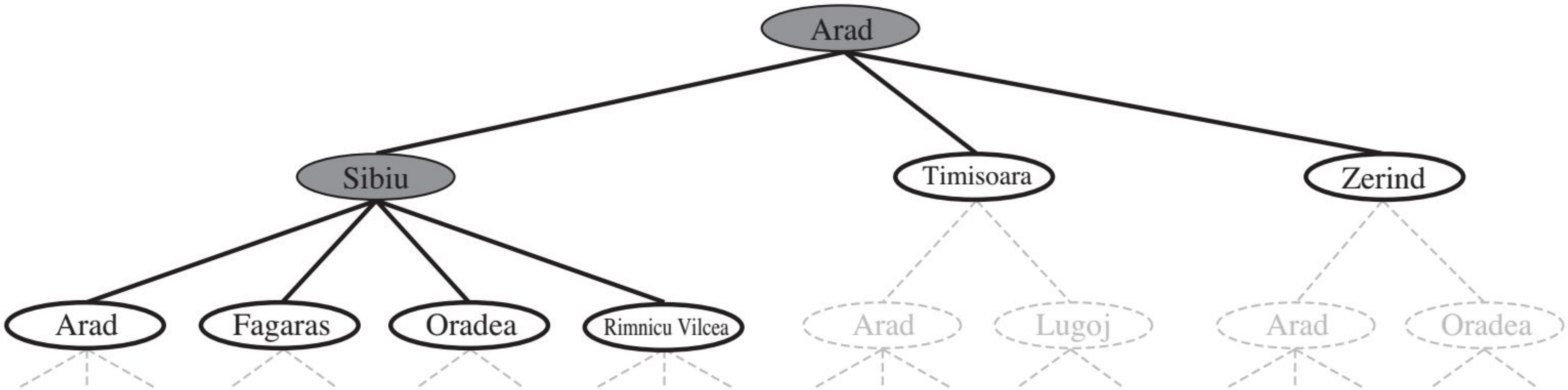
Basic search algorithms: *Tree Search*

Generalized algorithm to solve search problems

Enumerate in some order all possible paths from the initial state

- Here: search through ***explicit tree generation***
  - ROOT= initial state.
  - Nodes in search tree generated through ***transition model***
  - Tree search treats different paths to the same node as distinct

Generalized tree search



function TREE-SEARCH(*problem*, *strategy*) return a solution or failure

Initialize frontier to the *initial state* of the *problem*

do

if the frontier is empty then return *failure*

choose leaf node for expansion according to strategy & remove from frontier

if node contains goal state then return *solution*

else expand the node and add resulting nodes to the frontier

The strategy determines search process!

8-Puzzle: States and Nodes

A ***state*** is a (representation of a) ***physical configuration***

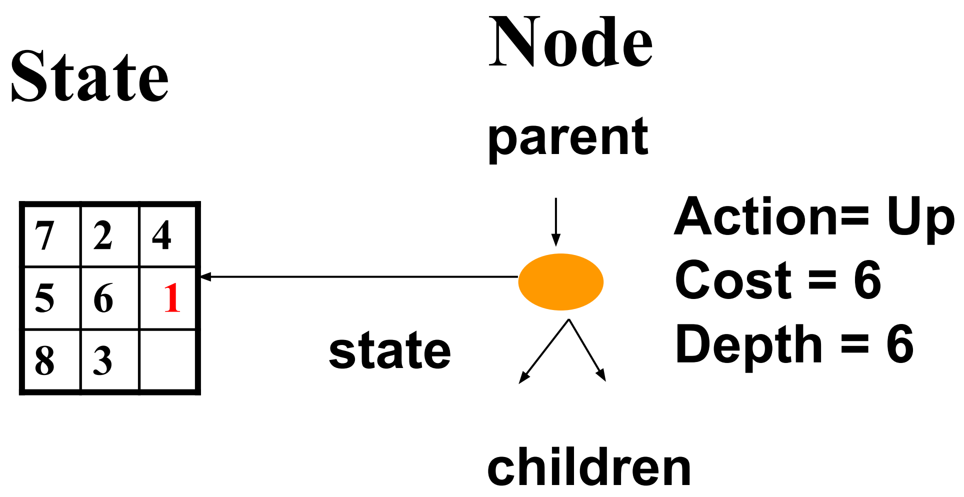
A ***node*** is a data structure constituting ***part of a search tree***

- Also includes *parent, children, depth*, *path cost g(x)*
- Here *node*= *<state, parent-node, children, action, path-cost, depth>*

States do not have parents, children, depth or path cost!

The EXPAND function

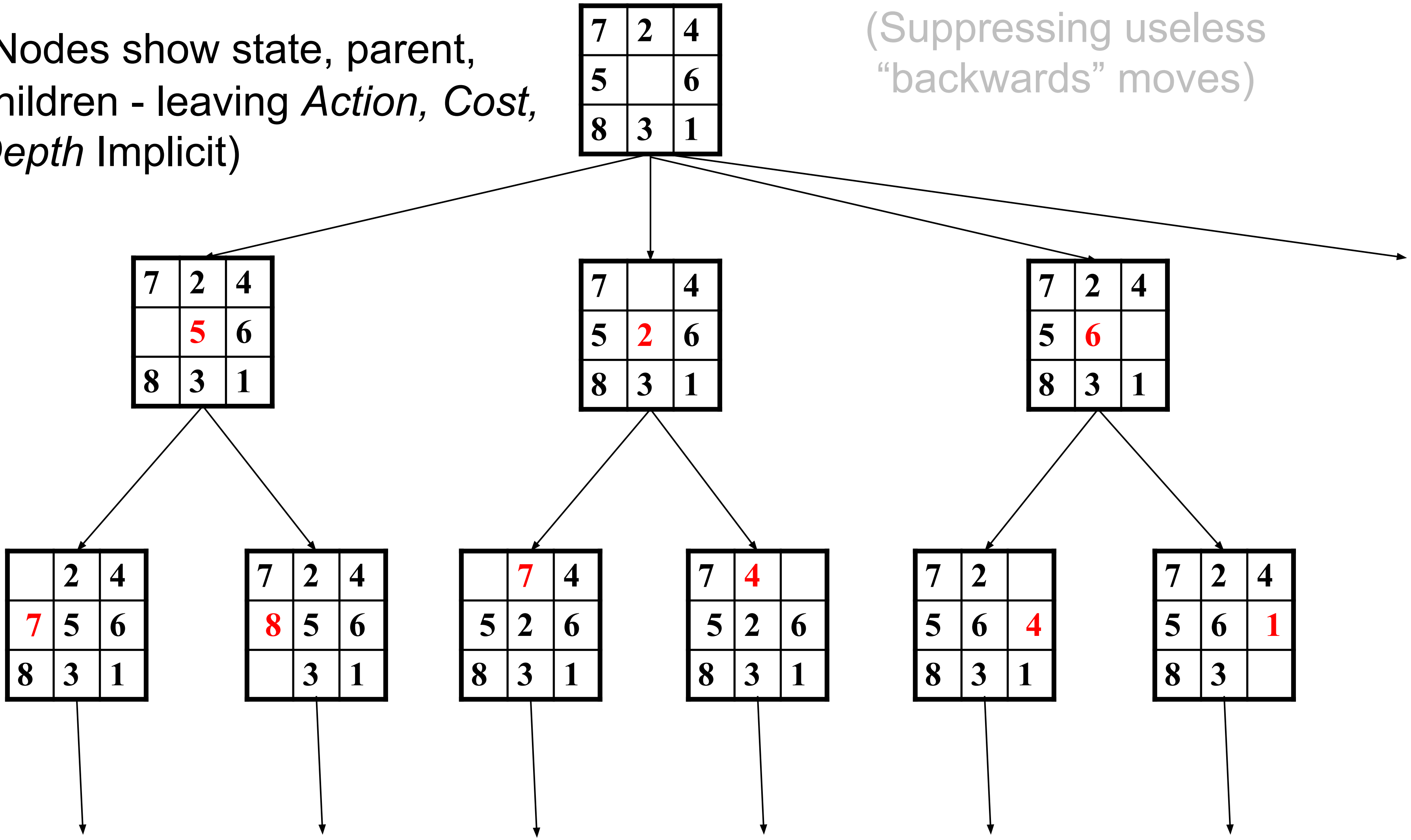
- uses the Actions and Transition Model to create the corresponding states
    - creates new nodes,
    - fills in the various fields



8-Puzzle *Search Tree*

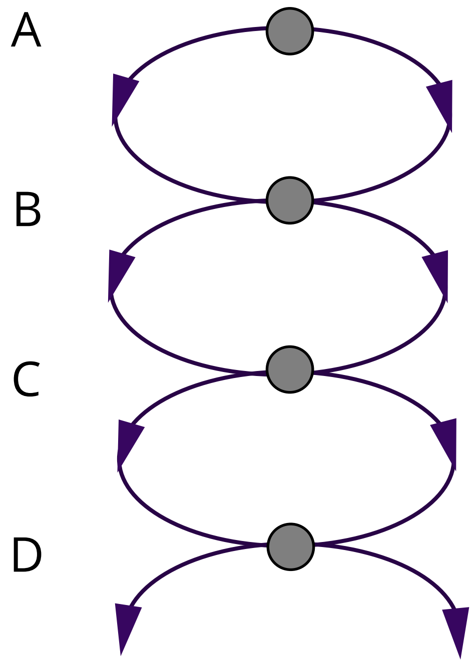
(Nodes show state, parent, children - leaving *Action, Cost, Depth* Implicit)

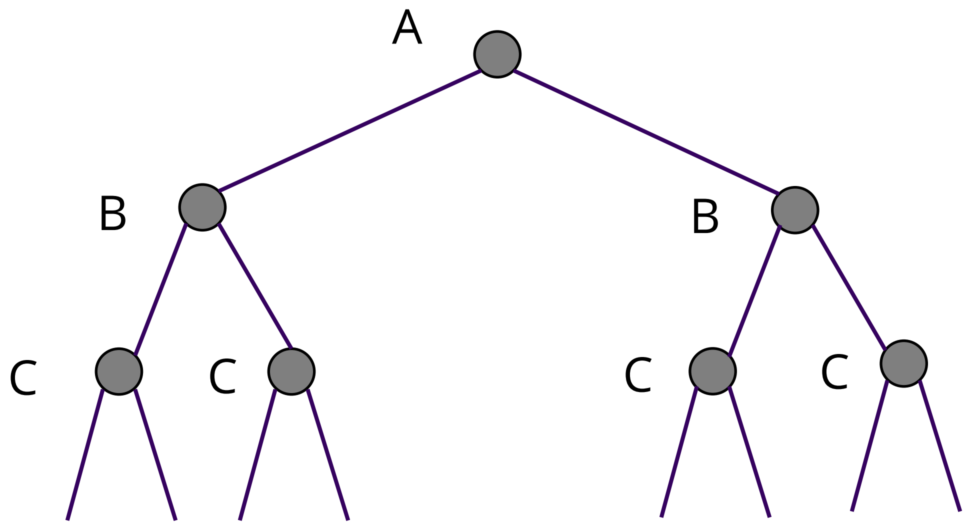
(Suppressing useless "backwards" moves)



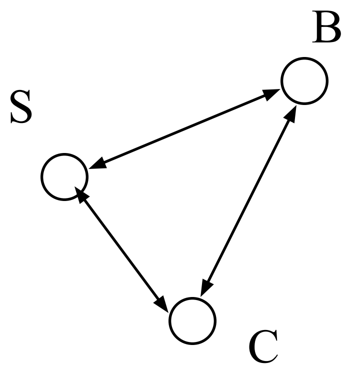
Problem: Repeated states

Failure to detect ***repeated states*** can turn a linear problem into an ***exponential*** one!

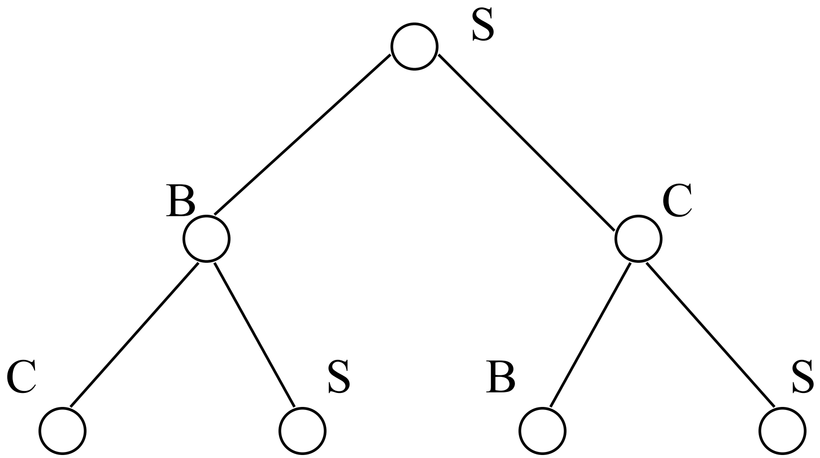




Solution: Graph Search!



State Space



Search Tree

Graph search

- Simple Mod from tree search: *Check to see if a node has been visited before adding to search queue*
  - must keep track of all possible states (can use a lot of memory)
  - e.g., 8-puzzle problem, we have 9!/2 ≈182K states

Graph Search vs Tree Search

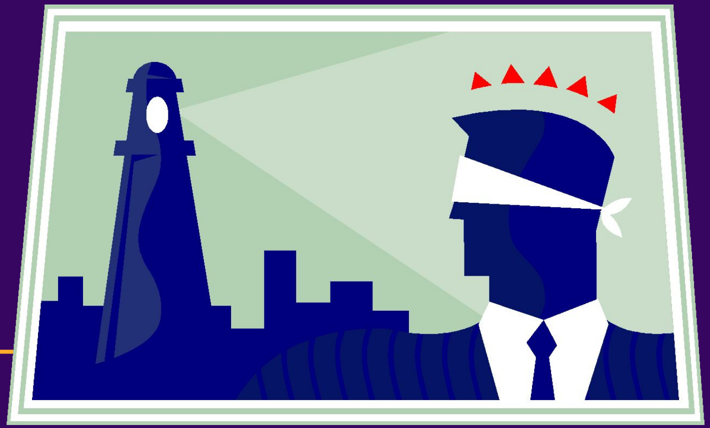
function TREE-SEARCH(*problem*) **returns** a solution, or failure
    initialize the frontier using the initial state of *problem*
    loop do
        if the frontier is empty **then return** failure
        choose a leaf nose and remove it from the frontier
        if the node contains a goal state **then return** the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier

function GRAPH-SEARCH(*problem*) returns a solution, or failure
    initialize the frontier using the initial state of *problem*
    initialize the explored set to be empty
    loop do
        if the frontier is empty **then return** failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state **then return** the corresponding solution
        add node to the explored set
        expand the chosen node, adding the resulting nodes to the frontier
            only if not in the frontier of explored set

An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italics are the additions needed to handle repeat states.

Uninformed Search Strategies

AIMA 3.3-3.4



Uninformed search strategies:

AKA “Blind search”

Uses only information available in problem definition

Informally:

Uninformed search: All non-goal nodes in frontier look equally good

Informed search: Some non-goal nodes can be ranked above others.

Search Strategies

Review: ***Strategy*** = order of tree expansion

- Implemented by different queue structures (LIFO, FIFO, priority)

Dimensions for evaluation

- ***Completeness***- always find the solution?
- ***Optimality*** - finds a least cost solution (lowest path cost) first?
- ***Time complexity*** - # of nodes generated *(worst case)*
- ***Space complexity*** **- # of nodes simultaneously in memory** *(worst case)*

Time/space complexity variables

- *b, maximum branching factor* of search tree
- *d, depth* of the shallowest goal node
- *m*, maximum length of any path in the state space (potentially ∞)

Introduction to *space* complexity

You know about:

- "Big O" notation
- *Time complexity*

Space complexity is analogous to time complexity

Units of space are arbitrary

- Doesn't matter because Big O notation ignores constant multiplicative factors
- Plausible Space units:
  - One Memory word
  - Size of any fixed size data structure
    - For example, size of fixed size node in search tree

Review: Breadth-first search

Idea:

- Expand *shallowest* unexpanded node

<u>Implementation:</u>

- *frontier* is FIFO (First-In-First-Out) Queue:
  - Put successors at the *end* of *frontier* successor list.

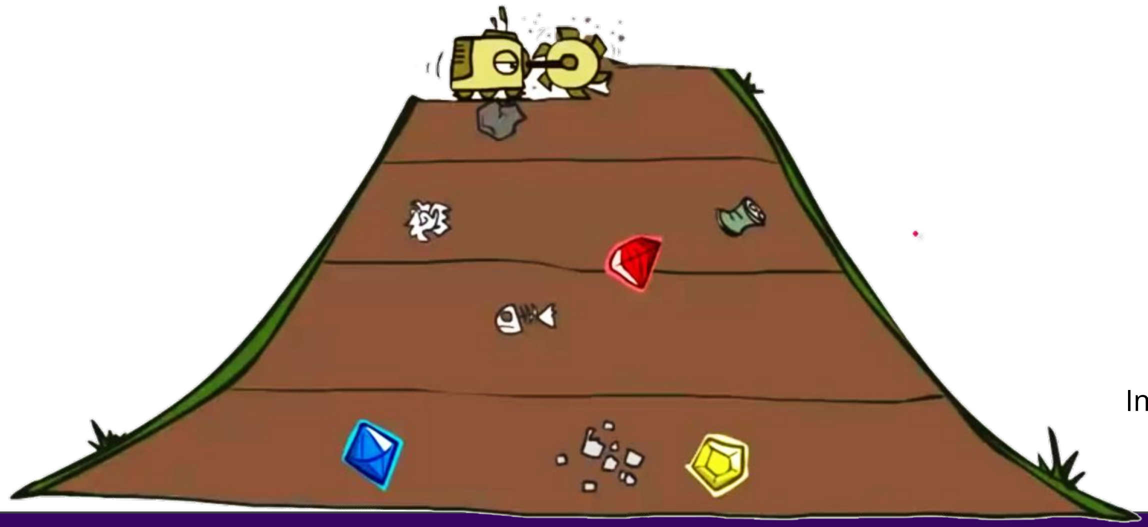


Image credit: Dan Klein and Pieter Abbeel http://ai.berkeley.edu

Breadth-first search (simplified)

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTIO
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)   /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier the
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Position within queue of new items determines search strategy

Subtle: *Node inserted into queue only after testing to see if it is a goal state*

Breadth-first search (simplified)

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
   node <- a node with STATE = problem.INITIAL-STATE, PATH-COST=0
   if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
   frontier <- a FIFO queue with node as the only element
   loop do
      if EMPTY?(frontier) then return failure
      node <- POP(frontier) // chooses the shallowest node in fronti
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
         child <- CHILD-NODE(problem, node, action)
         if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
         frontier <- INSERT(child, frontier)
```

Position within queue of new items determines search strategy

Subtle: *Node inserted into queue only after testing to see if it is a goal state*

From Figure 3.11 Breadth-first search (ignores loops, repeated nodes)

Properties of breadth-first search

Complete? Yes (if *b* is finite)

Time Complexity? $1+b+b^2+b^3+\ldots+b^d = O(b^d)$

Space Complexity? $O(b^d)$ (keeps every node in memory)

Optimal? Yes, if cost = 1 per step
(not optimal in general)

b: maximum branching factor of search tree
d: depth of the least cost solution
m: maximum depth of the state space (∞)

Exponential Space (and time) Not Good...

- Exponential complexity uninformed search problems *cannot* be solved for any but the smallest instances.
- *(Memory* requirements are a bigger problem than *execution* time.)

DEPTH	NODES	TIME	MEMORY
2	**110**	**0.11 milliseconds**	**107 kilobytes**
4	**11110**	**11 milliseconds**	**10.6 megabytes**
6	**10^6**	**1.1 seconds**	**1 gigabytes**
8	**10^8**	**2 minutes**	**103 gigabytes**
10	**10^{10}**	**3 hours**	**10 terabytes**
12	**10^{12}**	**13 days**	**1 petabytes**
14	**10^{14}**	**3.5 years**	**99 petabytles**

Assumes b=10, 1M nodes/sec, 1000 bytes/node

Review: Depth-first search

Idea:

- Expand *deepest* unexpanded node

<u>Implementation:</u>

- *frontier* is LIFO (Last-In-First-Out) Queue:
  - Put successors at the *front* of *frontier* successor list.

Image credit: Dan Klein and Pieter Abbeel http://ai.berkeley.edu

Properties of depth-first search

Complete? No: fails in infinite-depth spaces, spaces with loops

- Modify to avoid repeated states along path

 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? $O(b*m)$, i.e., linear space!

Optimal? No

b: maximum branching factor of search tree
d: depth of the least cost solution
m: maximum depth of the state space (∞)

Depth-first vs Breadth-first

Use depth-first if

- *Space is restricted*
- There are many possible solutions with long paths and wrong paths are usually terminated quickly
- Search can be fine-tuned quickly

Use breadth-first if

- *Possible infinite paths*
- Some solutions have short paths
- Can quickly discard unlikely paths

Outline for today's lecture

Formulating Search Problems – An Example

Search Fundamentals

Introduction to Uninformed Search

- Review of Breadth first and Depth-first search

Iterative deepening search (AIMA 3.4.4-5)

- *Strange Subroutine: Depth-limited search*
- *Depth-limited search + iteration = WIN!!*

Search Conundrum

Breadth-first

- Complete,
- Optimal
- *but* uses $O(b^d)$ space

Depth-first

- Not complete *unless m is bounded*
- Not optimal
- Uses $O(b^m)$ time; terrible if m >> d
- *but* only uses *O(**b*m) space***

How can we get the best of both?

Depth-limited search: A building block

Depth-First search *but with depth limit ℓ.*

- i.e. nodes at depth ℓ *have no successors.*
- No infinite-path problem!

If $\ell = d$ (by luck!), then optimal

- But:
  - **If $\ell < d$ then incomplete** ☹️
  - **If $\ell > d$ then not optimal** ☹️

Time complexity: $O(b^l)$

Space complexity: $O(bl)$ ☺️

Iterative deepening search

A general strategy to find best depth limit l.

- **Key idea: use *Depth-limited search* as subroutine, with increasing l.**

```
For l = 0 to ∞ do
      depth-limited-search to level l
      if it succeeds
         then return solution
```

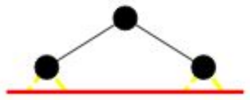
- ***Complete & optimal*: Goal is always found at depth *d*, the depth of the shallowest goal-node.**

Could this possibly be efficient?

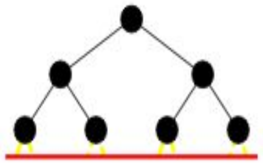
Nodes constructed at each deepening

Depth 0: 0 (Given the node, doesn't *construct* it.)

Depth 1: b^1 nodes



Depth 2: b nodes + b^2 nodes



Depth 3: b nodes + b^2 nodes + b^3 nodes

…

Total nodes constructed:

Depth 0: 0 (Given the node, doesn't *construct* it.)

Depth 1: b^1 = b nodes

Depth 2: b nodes + b^2 nodes

- Depth 3: b nodes + b^2 nodes + b^3 nodes
- …

Suppose the first solution is the last node at depth 3:

Total nodes constructed:

*3**b nodes + *2**b^2 nodes + *1**b^3 nodes

ID search, Evaluation: Time Complexity

- More generally, the time complexity is
  - $(d)b + (d-1)b^2 + \ldots + (1)b^d =$ *$O(b^d)$*

As efficient in terms of O(…) as Breadth First Search:

- $b + b^2 + \ldots + b^d =$ *$O(b^d)$*

ID search, Evaluation

Complete: YES (no infinite paths) ☺️

Time complexity: $O(b^d)$

Space complexity: $O(bd)$ ☺️

Optimal: YES if step cost is 1. ☺️

Summary of algorithms

Criterion	Breadth-First	Depth-First	Depth-limited	Iterative deepening
Complete?	YES	NO	NO	YES
Time	b^d	b^m	b^l	b^d
Space	b^d	bm	bl	bd
Optimal?	YES	NO	NO	YES

Next Up: Informed Search

AIMA 3.5-3.6

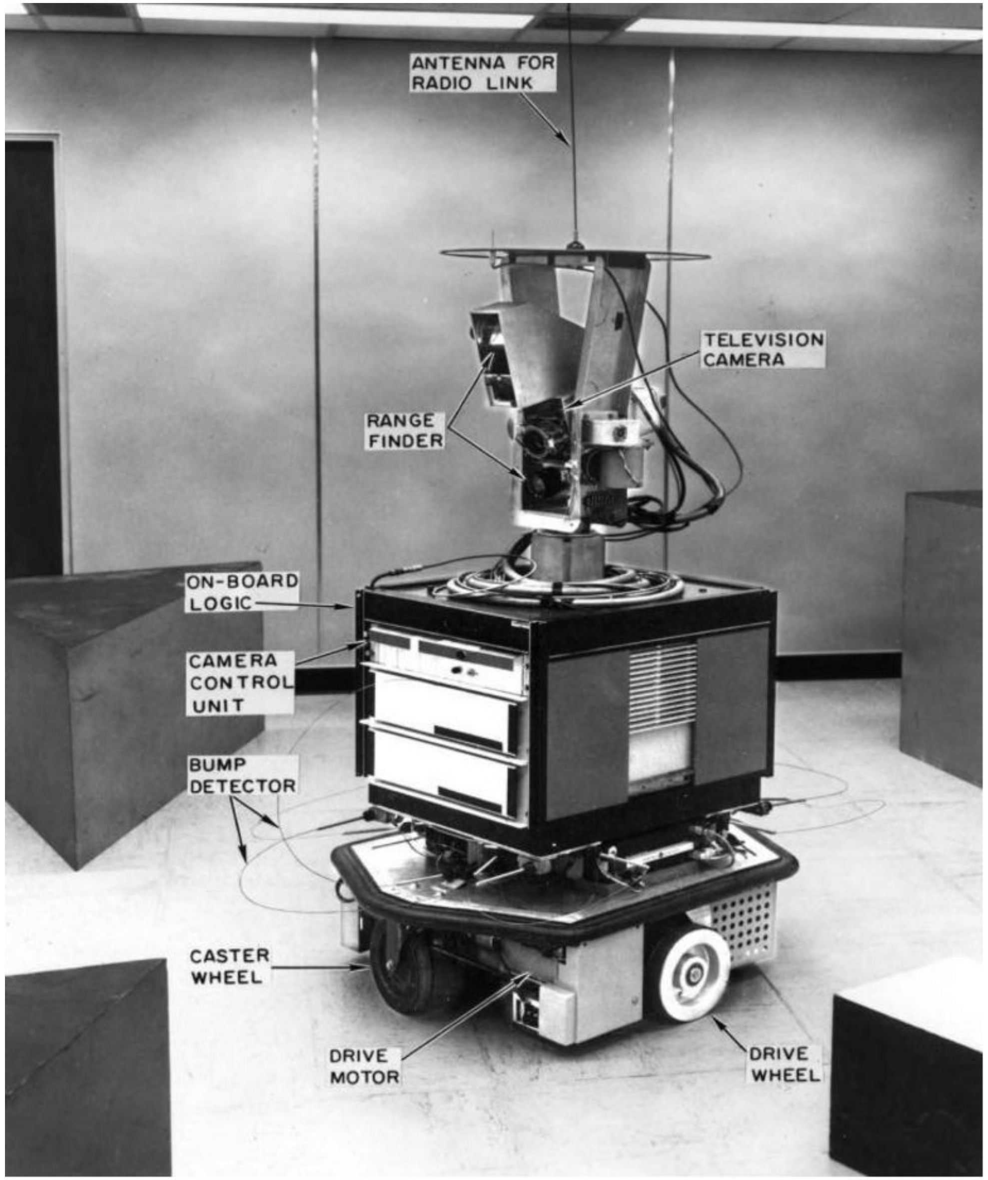
Informed Search

An **informed search** strategy uses **domain-specific information** about the location of the goals in order to find a solution **more efficiently** than uninformed search.

Hints will come as part of a **heuristic function** denoted h(n).

One of the most famous informed search algorithms is **A*** which was developed for **robot navigation**.

Shakey the robot was developed at the Stanford Research Institute from 1966 to 1972.



https://www.youtube.com/watch?v=7bsEN8mwUB8