

---

# **Intelligent Agents, Search Problem Formulation and Uninformed Search**

**AIMA, Chapters 2 and 3**



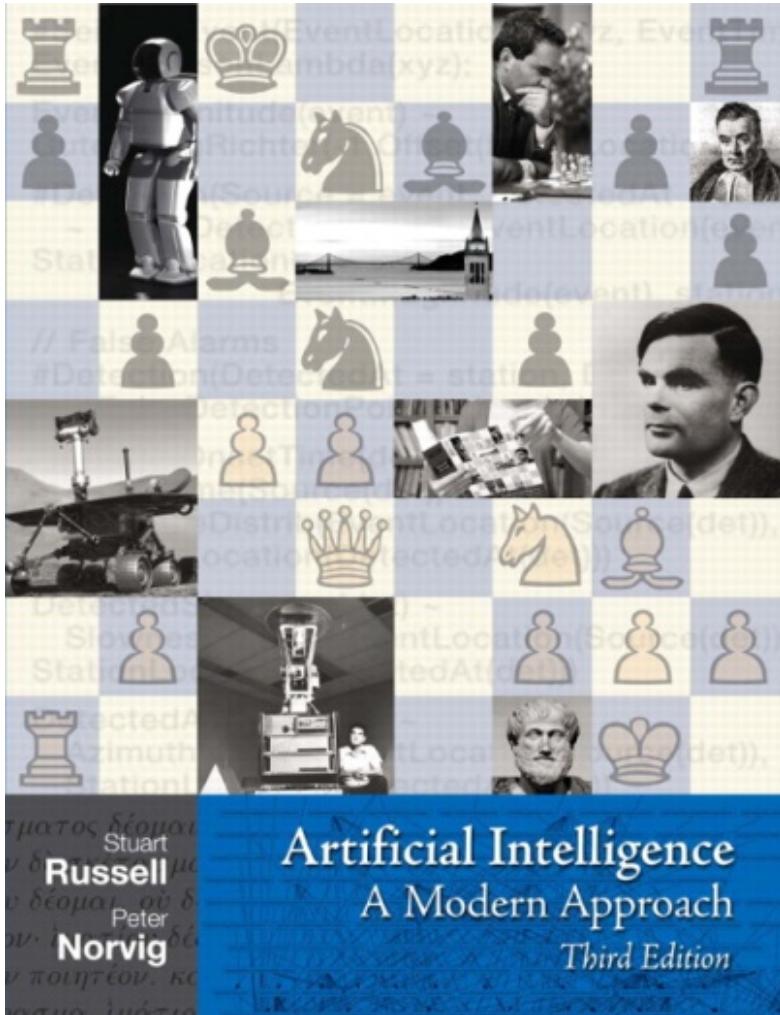
# Reminder – HW due Tuesday

---

- HW1 is due Tuesday night before 11:59pm.
  - Please submit early. 1 second late = 1 late day.
  - Verify by this afternoon that you have a Gradescope account for the class. Private post on Piazza if you don't.
- 
- My office hours are Tuesday/Thurs from 3pm-4pm. TAs office hours are on the course web site (still working out space for them).
  - No class on Tuesday (I'm traveling to DARPA).



# Outline for today's lecture



- ***Intelligent Agents***  
**(AIMA 2.1-2.3)**
- Task Environments
- Formulating Search Problems
- Uninformed Search  
**(AIMA 3.1-3.4)**

Thinking humanly	Thinking rationally
Acting humanly	Acting rationally

# Review: What is AI?

---

Views of AI fall into four categories:

Thinking humanly	Thinking rationally
Acting humanly	Acting rationally

***We will focus on "acting rationally"***



# Review: Acting rationally: rational agents

Thinking humanly	Thinking rationally
Acting humanly	Acting rationally

- **Rational behavior:** doing the right thing
- The right thing: that which is *expected to maximize goal achievement, given the available information*
- *Rational agent:* An agent is an entity that perceives and acts rationally

This course is about *effective programming techniques* for designing *rational agents*



# Agents



- An *agent* is anything that perceives its environment through *sensors* and can act on its environment through *actuators*
- A *percept* is the agent's perceptual inputs at any given instance



# What about your robot?

---

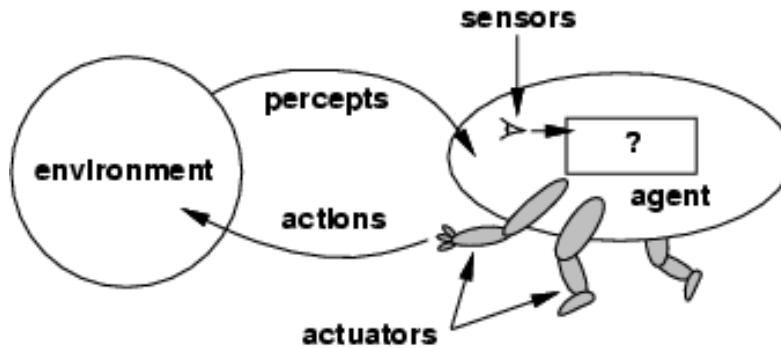


*What actuators does it have?*

*What sensors does it have?*

# Agents and environments

---



- An agent is specified by an **agent function**  $f:P \rightarrow A$  that maps a sequence of percept vectors  $P$  to an action  $a$  from a set  $A$ :

$$P = [p_0, p_1, \dots, p_t]$$

$$A = \{a_0, a_1, \dots, a_k\}$$

# Agents

---

- An **agent** is anything that can be viewed as
  - *perceiving* its *environment* through *sensors* and
  - *acting* upon that environment through *actuators*
- **Human agent:**
  - Sensors: eyes, ears, ...
  - Actuators: hands, legs, mouth, ...
- **Robotic agent:**
  - Sensors: cameras and infrared range finders
  - Actuators: various motors
- **Agents include humans, robots, softbots, *thermostats*, ...**



# Agent function & program

---

- The *agent program* runs on the physical *architecture* to produce  $f$ 
  - $\text{agent} = \text{architecture} + \text{program}$
- “Easy” solution: a giant table that maps every possible sequence  $P$  to an action  $a$ 
  - One small problem: exponential in length of  $P$

# Rational agents

---

- **Rational Agent:** For each possible percept sequence  $P$ , a rational agent selects an action  $a$  to *maximize its performance measure*
- **Performance measure:** An objective criterion for success of an agent's behavior, given the evidence provided by the percept sequence.

# Performance measure - example

---

- A performance measure for a vacuum-cleaner agent might include e.g. some subset of:
  - +1 point for each clean square in time T
  - +1 point for clean square, -1 for each move
  - -1000 for more than  $k$  dirty squares



# Rationality is *not* omniscience

---

- Ideal agent: maximizes *actual* performance, but needs to be *omniscient*.
  - Usually impossible.....
    - But consider tic-tac-toe agent...
  - Rationality  $\neq$  Guaranteed Success
- Caveat: computational limitations make *complete rationality* unachievable
  - design best *program* for given machine resources
- In Economics:  
“Bounded Rationality” → “Behavioral Economics”



# Rational agents 2

---

- **Rational Agent:** For each possible percept sequence  $P$ , a rational agent selects an action  $a$  to *maximize its performance measure*
- **Performance measure:** An objective criterion for success of an agent's behavior, given the evidence provided by the percept sequence.

## Revised:

- **Rational Agent:** For the current percept sequence  $P$ , a rational agent selects an action  $a$  that *maximizes the expected value of its performance measure*

# Outline for today's lecture

---

- Intelligent Agents
- *Task Environments (AIMA 2.3)*
- Formulating Search Problems

# Task environments

---

- To design a rational agent we need to specify a *task environment*
  - a problem specification for which the agent is a solution
- **PEAS:** to specify a task environment
  - **P**erformance measure
  - **E**nvironment
  - **A**ctuators
  - **S**ensors



# **PEAS: Specifying an automated taxi driver**

---

**P**erformance measure:

- ?

**E**nvironment:

- ?

**A**ctuators:

- ?

**S**ensors:

- ?



# **PEAS: Specifying an automated taxi driver**

---

**P**erformance measure:

- safe, fast, legal, comfortable, maximize profits

**E**nvironment:

- roads, other traffic, pedestrians, customers

**A**ctuators:

- steering, accelerator, brake, signal, horn

**S**ensors:

- cameras, LiDAR, speedometer, GPS



# **PEAS: Medical diagnosis system**

---

- ***Performance measure:***

- ***Environment:***

- ***Actuators:***

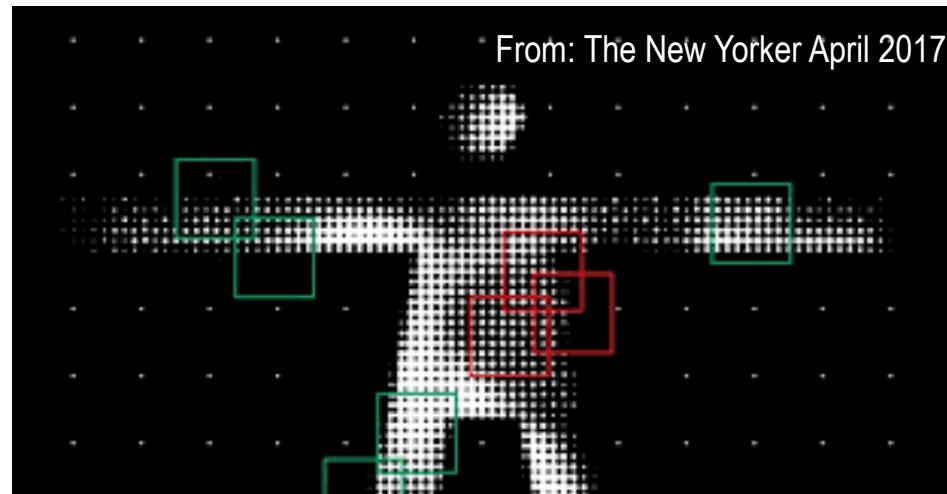
- ***Sensors:***



# **PEAS: Medical diagnosis system**

---

- **Performance measure:** Healthy patient, minimize costs, lawsuits
- **Environment:** Patient, hospital, staff
- **Actuators:** Screen display (form including: questions, tests, diagnoses, treatments, referrals)
- **Sensors:** Keyboard (entry of symptoms, findings, patient's answers)



# PEAS: Amazon Prime Air

---

**P**erformance measure:

- ?

**E**nvironment:

- ?

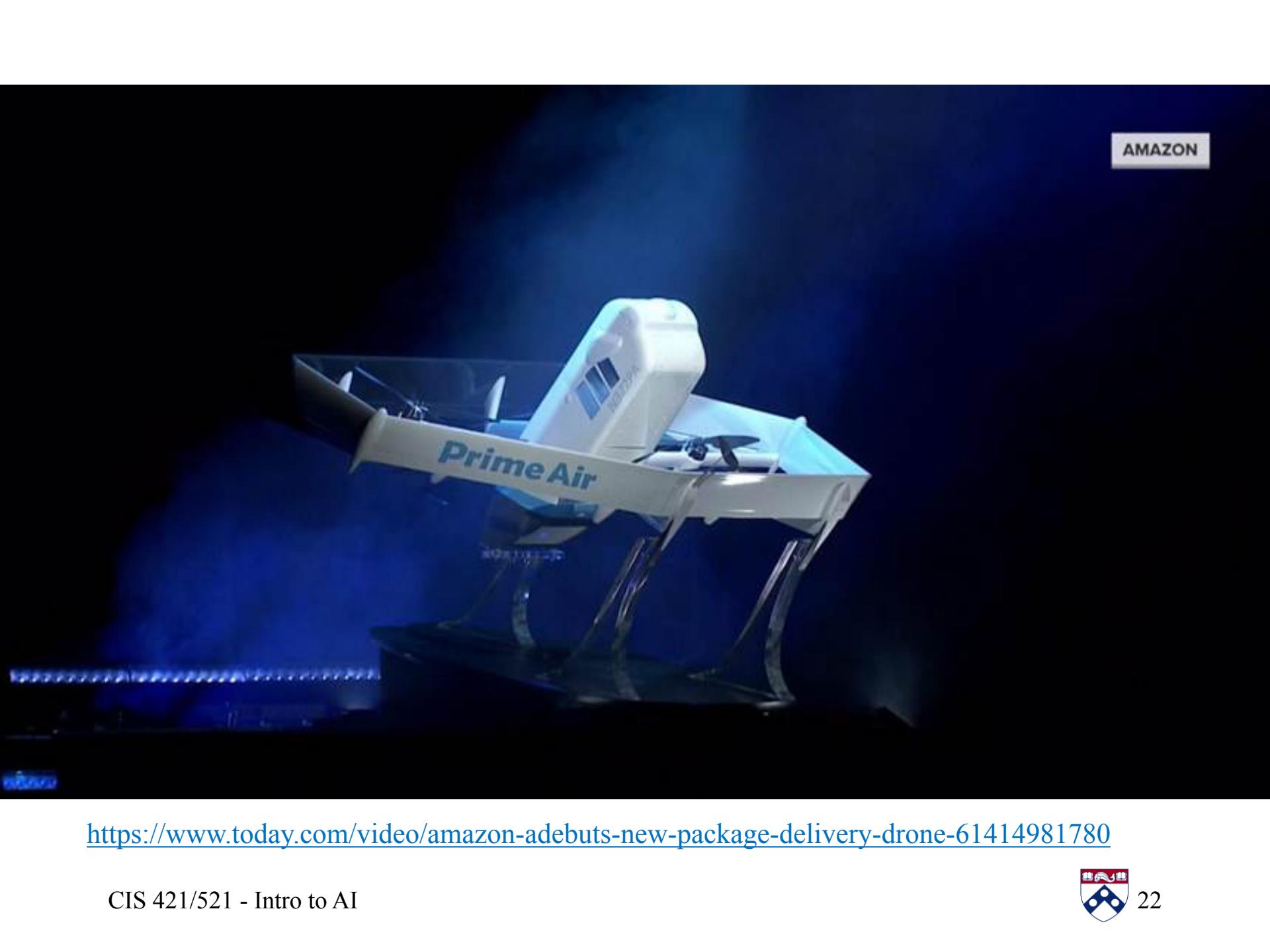
**A**ctuators:

- ?

**S**ensors:

- ?



A white Amazon Prime Air delivery drone is shown against a dark blue background. The drone has four propellers and a white body with "Prime Air" written on the side. A small "AMAZON" logo is in the top right corner.

AMAZON

<https://www.today.com/video/amazon-adebuts-new-package-delivery-drone-61414981780>

# PEAS: Specifying an Amazon delivery drone

---

## P<sub>erformance</sub> measure:

maximize profits - minimize time - obey laws governing  
airspace restrictions - deliver package to right location - keep  
package in good condition - avoid accidents - reduce noise -  
preserve battery life

## E<sub>nvironment</sub>:

- airspace - obstacles when airborne (other drones, birds, buildings, trees, utility poles)
- obstacles when landing (pets, patio furniture, lawnmowers, people, cars)
- weather - distances/route information between warehouse and destinations
- position of houses, and spaces that are safe for drop-off
- package weight

# PEAS: Specifying an Amazon delivery drone

---

## A<sup>c</sup>tuators:

- Propellers and flight control system- Payload actuators: E.g. Arm/basket/claw for picking up, dropping off packages- Lights or signals - Mechanism to announce/verify delivery- Device for delivering packages to customers

## S<sup>e</sup>nors:

- GPS - radar/Lidar- altitude sensor- weather sensors (barometer, etc). - gyroscope- accelerrometer- camera- rotor senors- weight sensor to recognize package

# The rational agent designer's goal

---

- Goal of AI practitioner who designs rational agents:  
given a *PEAS* task environment,
  1. Construct *agent function*  $f$  that maximizes the expected value of the performance measure,
  2. Design an *agent program* that implements  $f$  on a particular architecture

# Environment types: Definitions 1

---

- **Fully observable** (vs. partially observable): An agent's sensors give it access to the complete state of the environment at each point in time.
- **Deterministic** (vs. stochastic): The next state of the environment is completely determined by the current state and the action executed by the agent.
  - If the environment is deterministic except for the actions of other agents, then the environment is *strategic*.
- **Episodic** (vs. sequential): The agent's experience is divided into atomic "episodes" during which the agent perceives and then performs a single action, and the choice of action in each episode does not depend on any previous action. (example: classification task)

# Environment types: Definitions 2

---

- ***Static*** (vs. dynamic): The environment is unchanged while an agent is deliberating.
  - The environment is *semidynamic* if the environment itself does not change with the passage of time but the agent's performance score does.
- ***Discrete*** (vs. continuous): A limited number of distinct, clearly defined percepts and actions.
- ***Single agent*** (vs. multiagent): An agent operating by itself in an environment.

# Examples

---

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle Chess with a clock	Fully Fully	Single Multi	Deterministic Deterministic	Sequential Sequential	Static Semi	Discrete Discrete
Poker Backgammon	Partially Fully	Multi Multi	Stochastic Stochastic	Sequential Sequential	Static Static	Discrete Discrete
Taxi driving Medical diagnosis	Partially Partially	Multi Single	Stochastic Stochastic	Sequential Sequential	Dynamic Dynamic	Continuous Continuous
Image analysis Part-picking robot	Fully Partially	Single Single	Deterministic Stochastic	Episodic Episodic	Semi Dynamic	Continuous Continuous
Refinery controller Interactive English tutor	Partially Partially	Single Multi	Stochastic Stochastic	Sequential Sequential	Dynamic Dynamic	Continuous Discrete

# The Hardest Environment

---

- The hardest case is
  - *Continuous*
  - *Partially Observable*
  - *Stochastic*
  - *Continuous*
  - *Multiagent*
  - *Unknown Outcomes*

# Environment Restrictions for Now

---

- We will assume environment is
  - *Static*
  - *Fully Observable*
  - *Deterministic*
  - *Discrete*

---

# **Problem Solving Agents & Problem Formulation**

**AIMA 3.1-3.2**

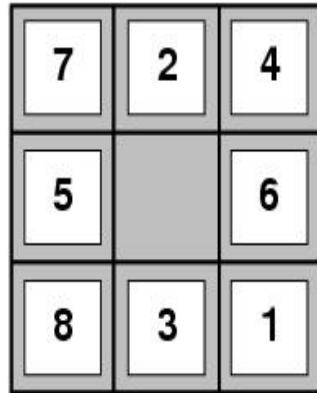




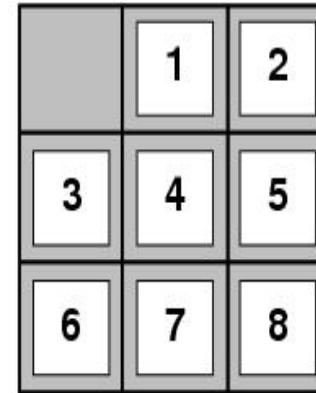
# Example search problem: 8-puzzle

- **Formulate *goal***

- Pieces to end up in order as shown...



Start State



Goal State

- **Formulate *search problem***

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

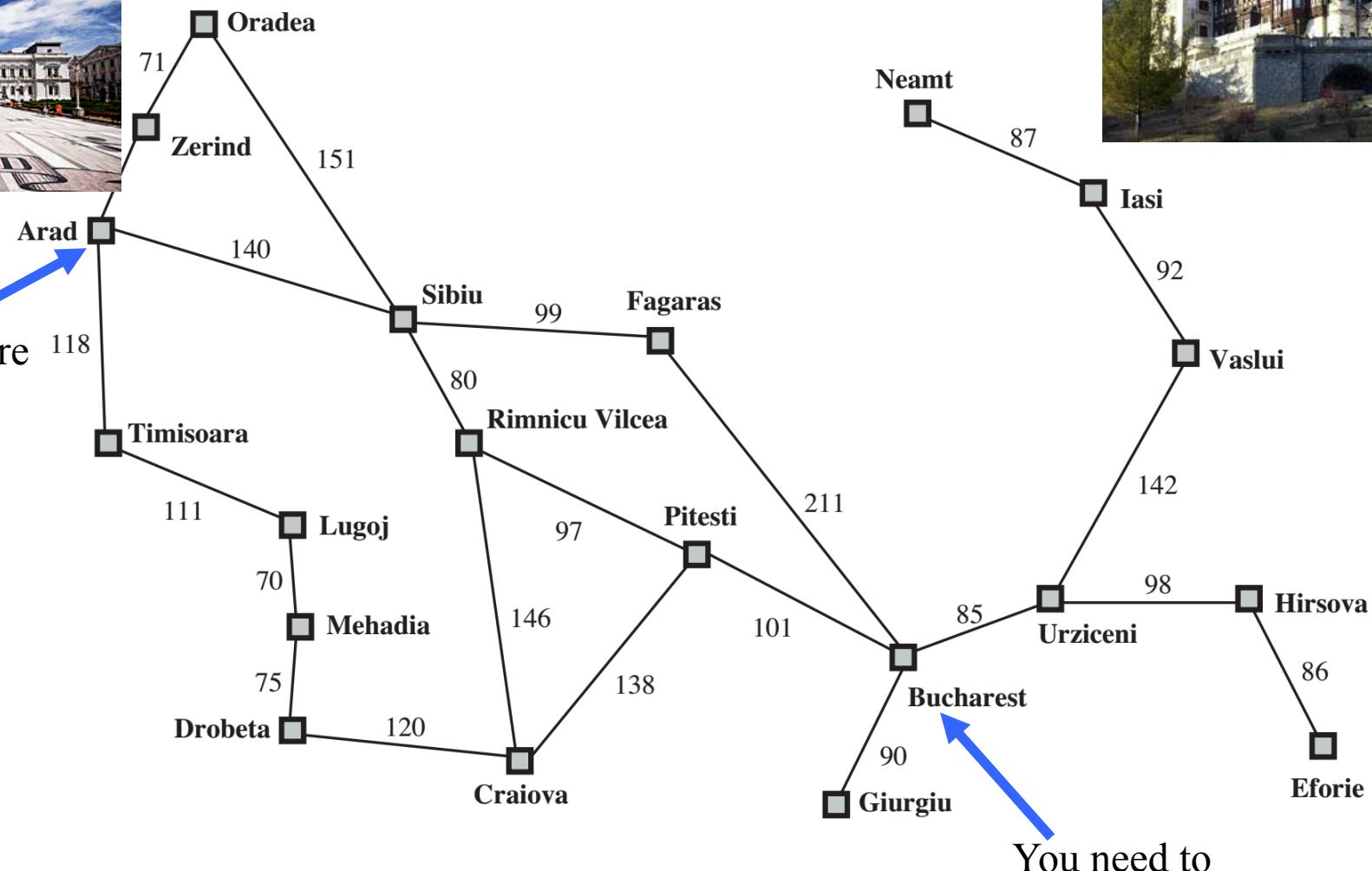
- **Find *solution***

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

# Example search problem: holiday in Romania



You are here



# 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  $A$   
 $\forall s \text{ } Actions(s) = \text{the set of actions that can be executed in } s,$   
that are *applicable* in  $s$ .
4. *Transition Model*:  $\forall s \forall a \in Actions(s) \text{ } Result(s, a) \rightarrow s_r$   
 $s_r$  is called a *successor* of  $s$   
 $\{s_i\} \cup Successors(s_i)^* = \text{state space}$
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  $\geq 0$   
– (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

# 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

---

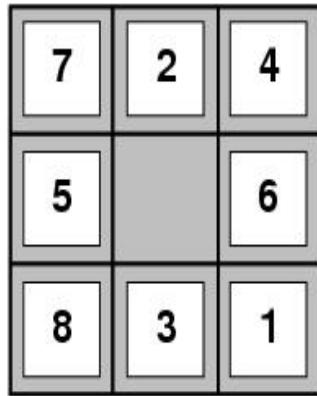
**Decide:**

- **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

---



Start State

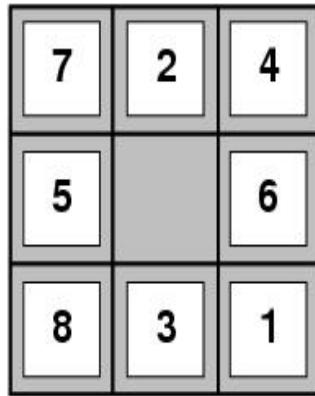


Goal State

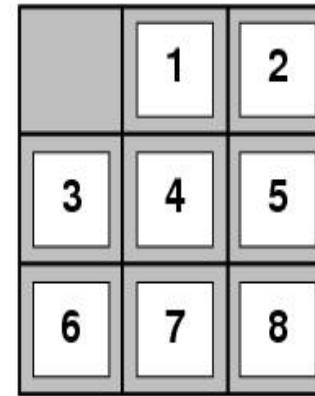
- **States??**
- **Initial state??**
- **Actions??**
- **Transition Model??**
- **Goal test??**
- **Path cost??**

# Example: 8-puzzle

---



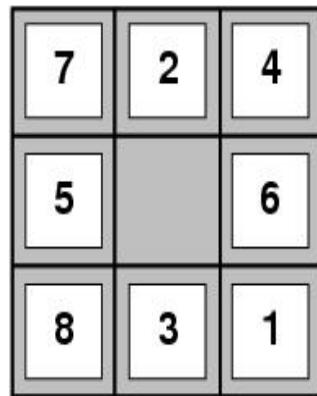
Start State



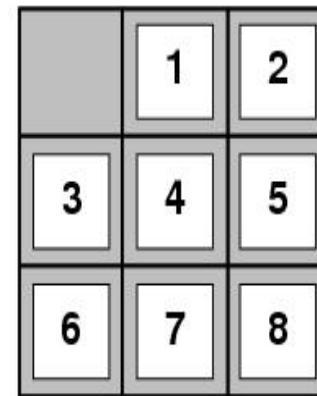
Goal State

- **States??**
- **Initial state??**
- **Actions??**
- **Transition Model??**
- **Goal test??**
- **Path cost??**

# Example: 8-puzzle



Start State



Goal State

- 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* → *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**

# Outline for today's lecture

---

- Intelligent Agents
- Task Environments
- Formulating Search Problems
- *Search Fundamentals (AIMA 3.3)*

# Useful Concepts

---

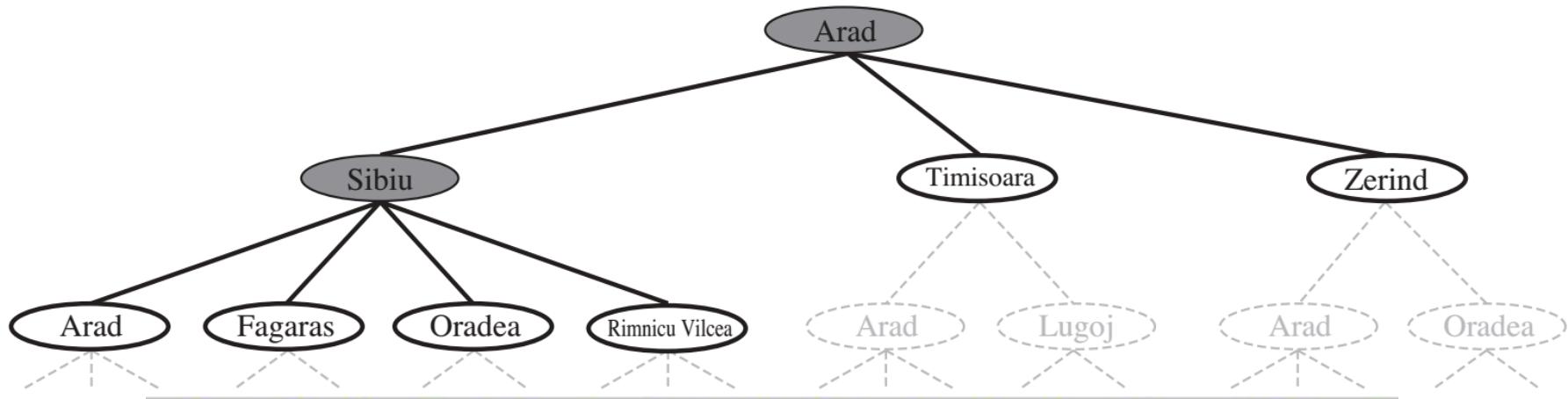
- **State space:** the set of all states reachable from the initial state by any sequence of actions
  - *When several operators can apply to each state, this gets large very quickly*
  - *Might be a proper subset of the set of configurations*
- **Path:** a sequence of actions leading from one state  $s_i$  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

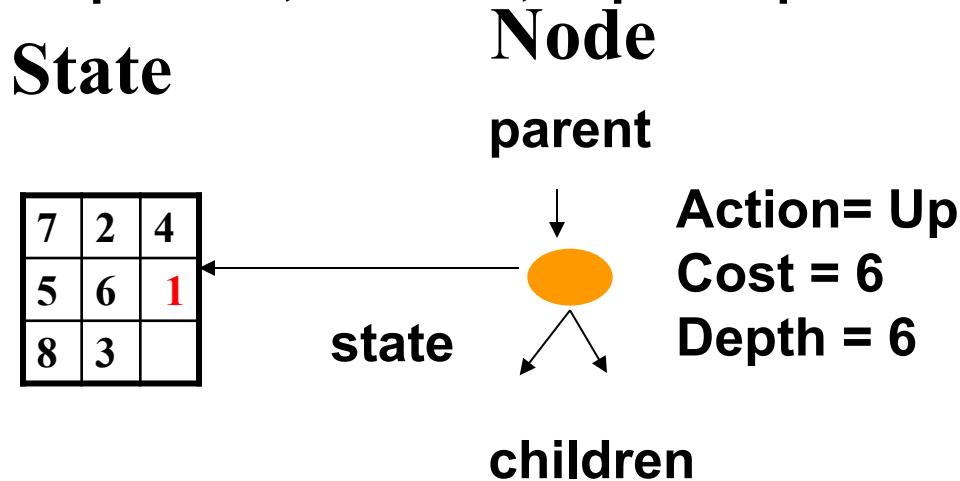
    if node contains goal state then return **solution**

    else expand the node and add resulting nodes to the frontier

Determines search process!!

# 8-Puzzle: States and Nodes

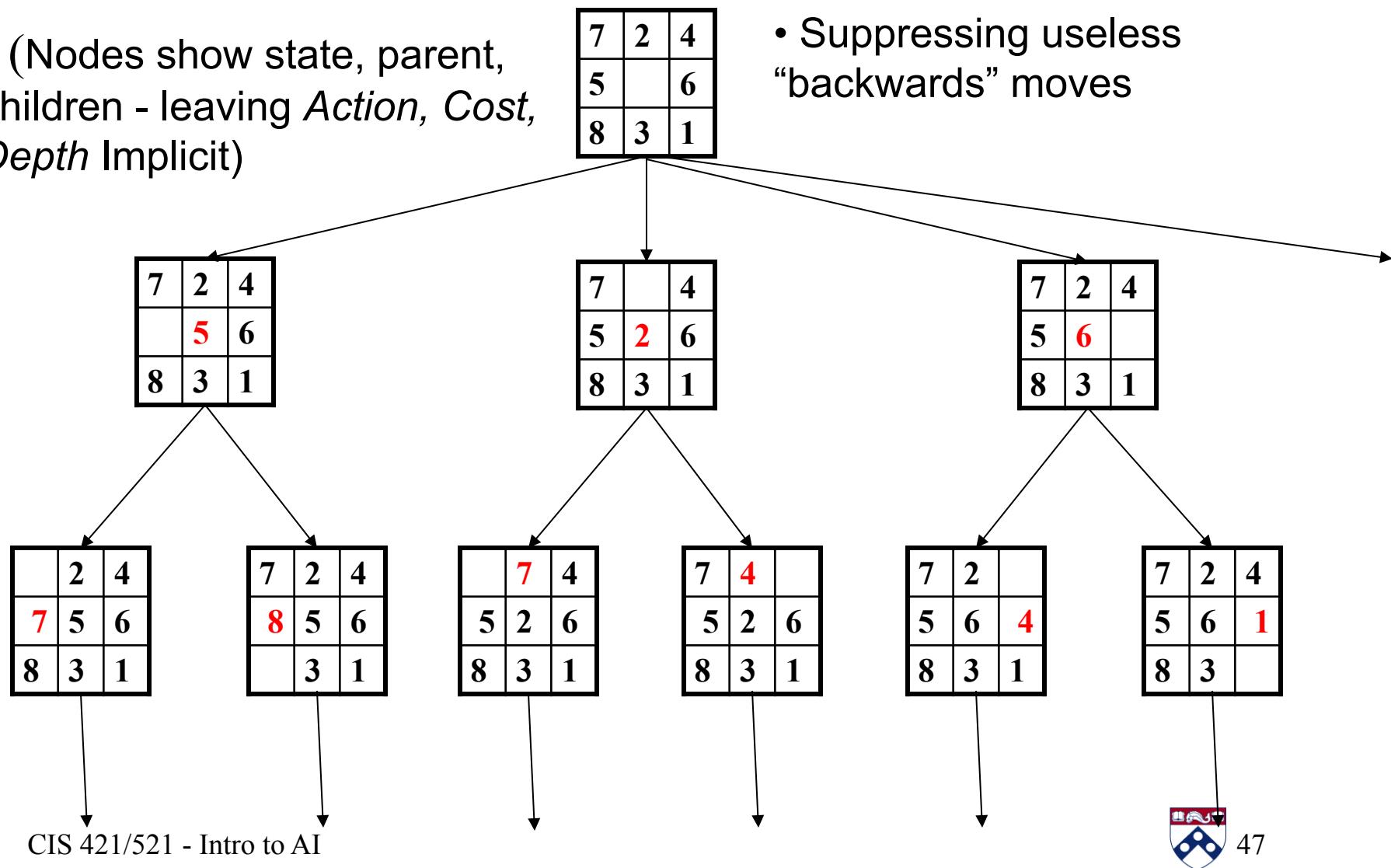
- A **state** is a (representation of a) **physical configuration**
- A **node** is a data structure constituting **part of a search tree**
  - Also includes *parent*, *children*, *depth*, *path cost*  $g(x)$
  - Here  $\text{node} = \langle \text{state}, \text{parent-node}, \text{children}, \text{action}, \text{path-cost}, \text{depth} \rangle$
- 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

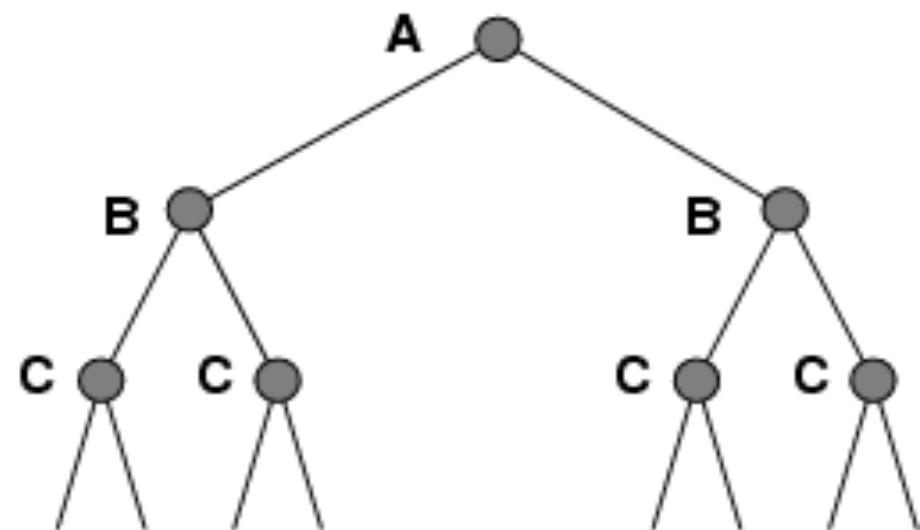
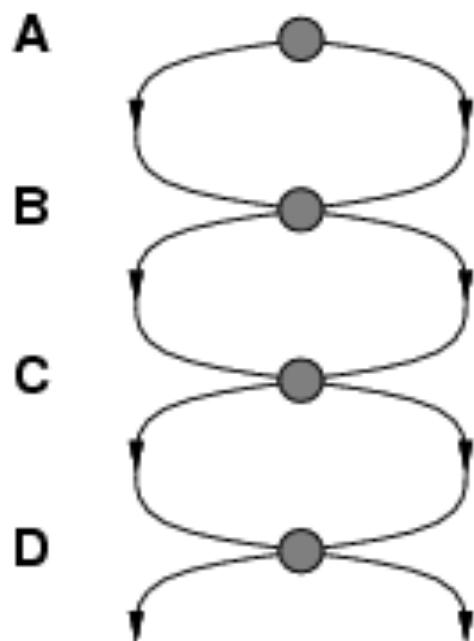
# 8-Puzzle Search Tree

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

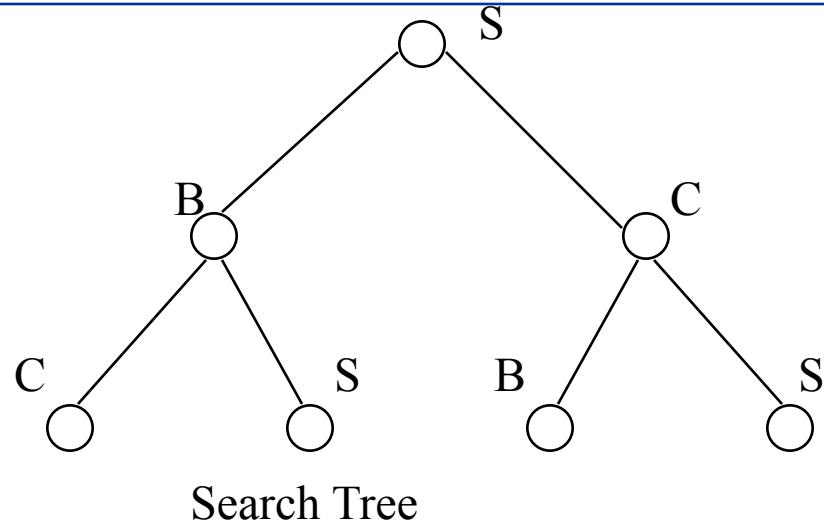
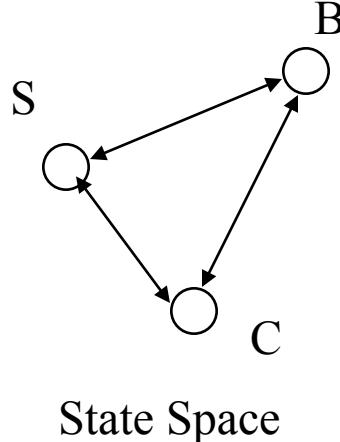


# Problem: Repeated states

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



# Solution: Graph Search!



- **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  $9!/2 \approx 182K$  states

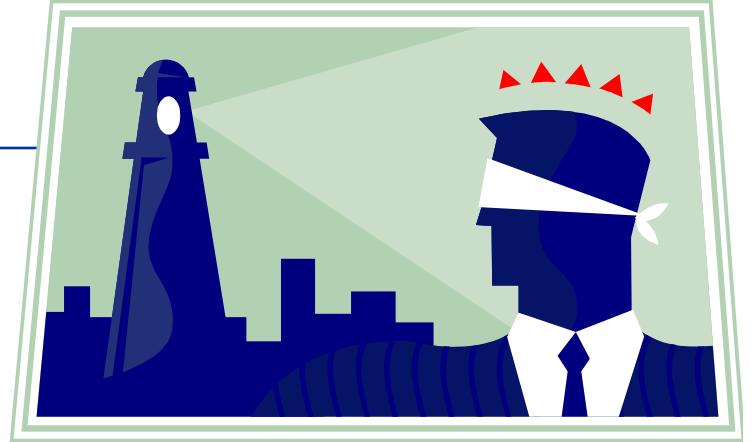
# Graph Search vs Tree Search

```
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 node 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 the node to the explored set
        expand the chosen node, adding the resulting nodes to the frontier
        only if not in the frontier or explored set
```

**Figure 3.7** An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated 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  $\infty$ )

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



# Breadth-first search (simplified)

**function** BREADTH-FIRST-SEARCH(*problem*) **returns** a solution, or failure

*node*  $\leftarrow$  a node with STATE = *problem*.INITIAL-STATE, PATH-COST = 0

**if** *problem*.GOAL-TEST(*node*.STATE) **then return** SOLUTION(*node*)

*frontier*  $\leftarrow$  a FIFO queue with *node* as the only element

*explored*  $\leftarrow$  an empty set

**loop do**

**if** EMPTY?(*frontier*) **then return** failure

*node*  $\leftarrow$  POP(*frontier*) /\* chooses the shallowest node in *frontier* \*/

add *node*.STATE to *explored*

**for each** *action* **in** *problem*.ACTIONS(*node*.STATE) **do**

*child*  $\leftarrow$  CHILD-NODE(*problem*, *node*, *action*)

**if** *child*.STATE is not in *explored* or *frontier* **then**

**if** *problem*.GOAL-TEST(*child*.STATE) **then return** SOLUTION(*child*)

*frontier*  $\leftarrow$  INSERT(*child*, *frontier*)

Position within queue of new items determines search strategy

Subtle: Node inserted queue only after test 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 frontier
```

```
    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)

CIS 421/521 - Intro to AI

# Properties of breadth-first search

---

- Complete? Yes (if  $b$  is finite)
- Time Complexity?  $1+b+b^2+b^3+\dots+b^d = O(b^d)$
- 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 ( $\infty$ )*

# 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*.)

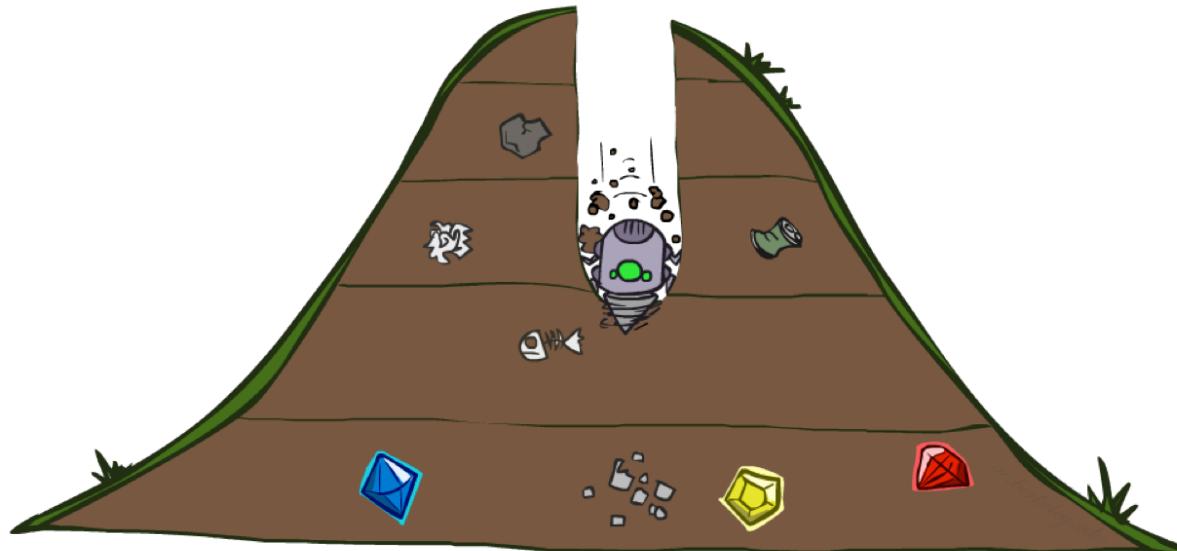
DEPTH	NODES	TIME	MEMORY
2	110	0.11 milliseconds	107 kilobytes
4	11110	11 milliseconds	10.6 megabytes
6	$10^6$	1.1 seconds	1 gigabytes
8	$10^8$	2 minutes	103 gigabytes
10	$10^{10}$	3 hours	10 terabytes
12	$10^{12}$	13 days	1 petabytes
14	$10^{14}$	3.5 years	99 petabytes

Fig 3.13 Assumes b=10, 1M nodes/sec, 1000 bytes/node

# Review: Depth-first search

---

- **Idea:**
  - Expand *deepest* unexpanded node
- **Implementation:**
  - *frontier* is LIFO (Last-In-First-Out) Queue:
    - Put successors at the *front* of *frontier* successor list.



# 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 ( $\infty$ )*

# 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  $m \gg d$
- 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  $\ell$* 
  - i.e. nodes at depth  $\ell$  *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:  $O(b^\ell)$
- 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

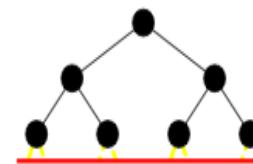
- Depth 0: 0 (Given the node, doesn't *construct* it.)



- 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<sup>2</sup> nodes + 1\*b<sup>3</sup> nodes**

# ID search, Evaluation II: Time Complexity

---

- More generally, the time complexity is
  - $(d)b + (d-1)b^2 + \dots + (1)b^d = O(b^d)$
- As efficient in terms of  $O(\dots)$  as Breadth First Search:
  - $b + b^2 + \dots + b^d = O(b^d)$

# ID search, Evaluation III

---

- 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?	<b>YES</b>	<b>NO</b>	<b>NO</b>	<b>YES</b>
Time	$b^d$	$b^m$	$b^l$	$b^d$
Space	$b^d$	$bm$	$bl$	$bd$
Optimal?	<b>YES</b>	<b>NO</b>	<b>NO</b>	<b>YES</b>