Tufts

Lecture 02: Intelligent Agents & the Use of Search

Artificial Intelligence (CS 131)

1

Turing Test: Intelligence = ***Acting Humanly***

- Alan Turing (1950) "Computing Machinery and Intelligence"
  - Proposed an imitation game
  - Predicted that by 2000, machines could fool average person for 5 minutes, 30% of the time
- One problem: not everyone agrees on the standard proposed by the test, and whether it is meaningful
- In any case, we still haven't got there yet…
  - Loebner prize for convincing bots would award up to $100,000 (and a gold medal) for a truly convincing interactive agent
  - No such agent has ever really been approached

2

What Should an Intelligent System Do?

- Following Turing, we take an operational approach:

> Intelligence is defined by some means of measuring performance in a set task.

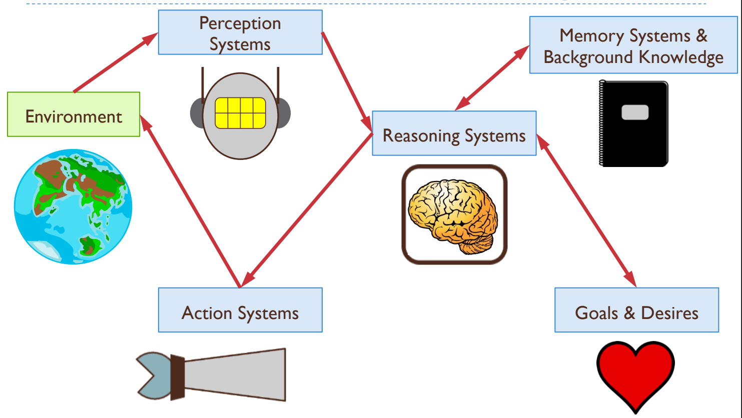
- An intelligent system is one that optimizes some measure
- How much it changes things so that it gets closer towards the goals that have been set for it
  - The word-count of error-free text translated
  - Customer satisfaction for automated dialogue systems
  - Hours of accident free, real-time driving
  - Amount of data collected by an autonomous space-vehicle
  - …

3

The Agent-Based Approach to AI

4

Possible Components of an AI Agent



Judging the Best Agent

- Suppose we have been given the common elements:
  1. A precise performance measure
  2. A sequence of world-information states (perceptions)
  3. A starting knowledge-base for the agent
  4. A fixed set of actions the agent can perform

> The best agent is then the one that: *maximizes* the performance measure (1), when *compared to* all agents that experience the *same world* (2), and have access to the *same knowledge* (3) and have the *same actions* available (4)

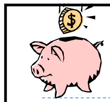
Using PEAS to Describe Agents

Performance	Environment	Actuators	Sensors

- **Example**: Autonomous automobile driving system
- **P**erformance: Hours of safe travel, obedience to laws, minimal time to destination.
- **E**nvironment: Road, traffic, pedestrians, passengers.
- **A**ctuators: Vehicle turning, acceleration, braking, signals.
- **S**ensors: Radar, video cameras, sonar, spoken-word interface for destinations, GPS.

Environments Vary

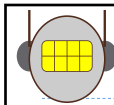
- Fully observable or partially observable?
  - Chess or poker?
- Deterministic or stochastic?
  - Pac-Man or Ms. Pac-Man?
- Episodic or sequential?
  - Assembly line robot or autonomous automobile?
- Static or dynamic?
  - Checkers or space exploration?
- Discrete or continuous?
  - Backgammon or robot soccer?
- Single agent or multiagent?
  - Mario game controller or NPC shooter team?



Performance Measures Vary

- Goal-directed performance: a single end-point that is either achieved, or not, no matter how it is done
  - Winning a game of robot soccer
  - Clearing all levels in Pac-Man
  - Vacuuming the living room
- Utility-directed performance: a numerical measure, which can be achieved in greater or lesser amounts
  - Driving a vehicle safely while arriving at finish in the least time
  - Achieving the highest level of customer satisfaction ratings
  - Exploring the greatest number of square kilometers of Mars while returning the most varied set of rock samples

9



Agents Vary

- Reflex agents: pre-programmed routines for dealing with any situation they perceive in their environment
  - Construction robots
  - Roomba vacuums
  - Video-game NPC controllers
- Memory agents: keep track of the world around them as they perceive it over time, and build up more complex knowledge & action possibilities
  - Autonomous vehicles
  - Spoken dialogue systems
- Reasoning agents: represent knowledge of world and do explicit problem solving
  - Chess playing programs
  - Space exploration systems
- Adaptive agents: can change their behavior over time
  - Learning agents for game play
  - Preference-learning automated assistants

10

Using Search to Solve AI Problems

11

Basic Search Techniques for AI

- Search is a common method for solving AI problems
  - Allows precise problem formulations
  - Solves a variety of problems directly
  - Provides a simple and direct algorithm
- We will first consider some uninformed search methods
  - No special information about the problem used
  - Automatic, simple ways of choosing how search will proceed
  - Technique relies heavily on proper problem formulation
  - A range of algorithms, with different performance profiles

12

Sample Problems for Search

- Simple puzzles, like 8-puzzle:

2	8	3
1	6	4
7		5

Start State

1	2	3
4	5	6
7	8	

End State

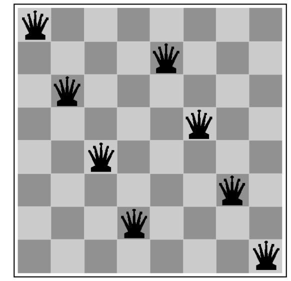
- Cryptarithmetic problems:

```
  SEND
+ MORE
 MONEY
```

```
O = 0   N = 6
M = 1   D = 7
Y = 2   R = 8
E = 5   S = 9
```


The n-Queens Problem

- Place n Queens on an $(n \times n)$ chessboard, so that no two attack (are in line with) one another
  - Popular algorithmic benchmark
  - Solved via search for $n \leq 500{,}000$
  - Can be solved mathematically using work of Hoffman, Loessi, & Moore (1969) for any values of $n > 3$



(Not a solution! Can you find one?)

Real Examples

- A large number of problems of real interest can be solved using search techniques:
  - Theorem proving in math and logic
  - Combinatorial optimization in chip design
  - Robot navigation and path planning
  - Resource scheduling in computing
  - Complex game play
- Solving such problems involves re-formulating them so search techniques can be applied

Formalizing a Search Problem

1. States: the set of all things to search through
2. Initial state: where we starts
3. Goal states/tests: how we know we've reached solution
4. Actions: what things we can do to change the state, moving along some path in our search-space
5. Transition model: what happens when we take some action a in some state s_1 (i.e., state s_2 we end up in)
6. Action cost function: what it costs (if anything) to take our actions, moving from state to state

Solving a Search Problem

- A solution to a search problem is a sequence of actions that generates a complete path from a starting state to a goal state
- An optimal solution is one that has minimal overall cost

- This leads to a couple of questions:

1. How do we *balance* the cost of a solution with the cost of doing the search itself?
2. How do we *measure* these costs?

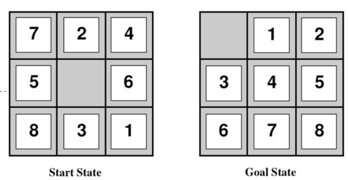
Important Assumption: Non-negative Costs

- The text (p. 65) notes that in all search problems considered, it is assumed that the cost of any search step is always some postive value $c > 0$, and total cost is just the ***sum***

- Why is this important?

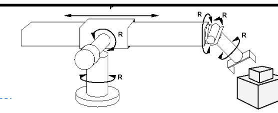
1. If the cost of actions can be *any* value, what does an "optimal" algorithm need to do?
   - Note: since a negative "cost" is actually a *reward*, this would mean that there is no upper limit on rewards we could get
2. What if negative costs *are* allowed, but there is a *lower bound*, so every cost is $c \geq -\varepsilon$, for some fixed value ε?
   - How does this affect search where infinite looping solutions *are not* allowed? What about when infinite loops *are* allowed?

The 8-Puzzle Problem



- States: arrangements of tiles
  - Integer sequences like: <7, 2, 4, 5, 0, 6, 8, 3, 1>
  - Gives 9! = 9 x 8 x 7 x … x 2 x 1 = 362880 states
- Goal: sequence <0, 1, 2, 3, 4, 5, 6, 7, 8>
- Actions: move blank tile in one of 4 directions
  - Not all moves always available
- Transitions: deterministic transitions from state to state
- Path cost: 1 unit per move (why?)

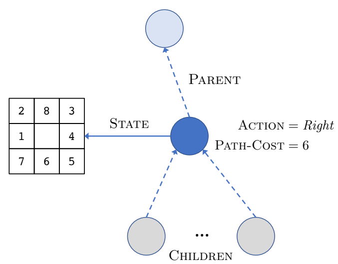
Robotic Assembly



- Robot arm tasked to build a specific object out of known parts
- States: combinations of positions for arm and object to build
  - Robotic joint angles
  - Location and orientation of each part
  - Is the space continuous? How do we handle this?
- Goal: Assembled object
  - How do we distinguish one of many?
- Actions: continuous arm movements, $\text{config}_1 \rightarrow \text{config}_2$
- Transitions: changes of the state of robot and parts, given actions
  - Deterministic or not?
- Path cost: looking for most efficient solution
  - Time to construct entire object?
  - Most reliable solution?

Search Trees

- A search tree represents a process using a graph
- Nodes in the graph represent search up to some point
- Nodes contain information:
  1. Current *state* of problem
  2. *Action* taken to get into state
  3. *Path-cost* to get to that state
  4. Link back to *parent*, i.e. the previous step in the search-path(for back-tracking purposes)



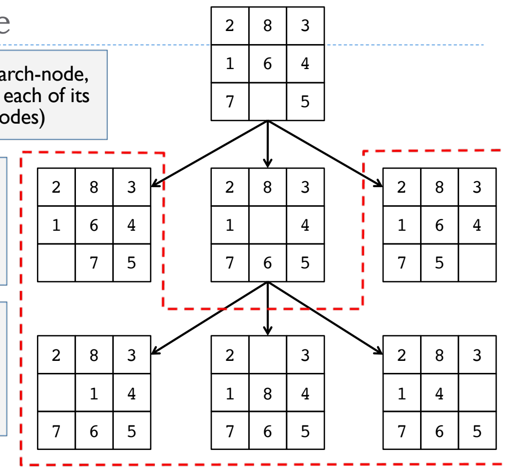
21

An Example

When we examine a search-node, we expand it, producing each of its successors (next nodes)

At each stage, we choose an unexpanded leaf node, typically *avoiding repeats*

The set of leaf nodes that have not been expanded yet is the search frontier



22

A General Search Technique

Image source: Russell & Norvig (2021)

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

- Every search starts from the specified initial state of the problem
- In order to track our progress, we keep track of two sets of nodes:
  1. Frontier: nodes we know about, but haven't expanded yet (our initial state goes into the frontier at the start of search)
  2. Reached: nodes we have already expanded

23

A General Search Technique

Image source: Russell & Norvig (2021)

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```


- At each step, so long as we still have something left in the frontier:
  1. We grab the next node in the frontier; these will be ordered by some priority function that defines their order
  2. We run our goal-test on the node: if we hit the goal, we are done!

24

A General Search Technique

Image source: Russell & Norvig (2021)

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

- If we haven't reached the goal, we expand our node to get the other nodes that we can reach based upon our various actions
- We add any new node to the frontier if:

1. It is a brand-new node, with a state we haven't seen, OR
2. We have seen the state before, but have a lower path-cost (the new version replaces any older version in the frontier)

25

A General Search Technique

Image source: Russell & Norvig (2021)

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

Some searches fail!

- No algorithm solves every problem
- If the frontier becomes empty, we have expanded every node we can reach in the search, without ever passing our goal-test

26