L01 – Introduction and State-Space Search

AIMA4e: Required: Ch 3.1–3.4; Optional: Ch 1, 2

What you should know after this lecture

- Basic logistics of 6.4110/16.420 Fall 2023
- The Rational-Agent view of AI
- Agent domains and where to find them
- State-space search
  - Minimizing additive path cost
  - Importance of avoiding redundant paths

6.4110/16.420 Course logistics

- Lecture: M, W 9:30 - 11
- Optional problem-solving sessions: Fridays at 11 (starting 9/15)
- Graduate extra meeting: Fridays at 10
- Staff:
  - Leslie Kaelbling (`lpk@mit.edu`)
  - Nick Roy (`nickroy@mit.edu`)
  - Laura Brandt (`lebrandt@mit.edu`)
  - Annie Feng (TA, `azf@mit.edu`)
  - Elizabeth Lee (TA, `jelizlee@mit.edu`)
  - Julian Yocum (TA, `juliany@mit.edu`)
- cat-soop: `https://airr.mit.edu`
- Piazza: Ask all questions here. Can be private to staff.
- Exams: Midterm October 30, in class; Final time TBD
- Homework: Weekly via cat-soop due Mondays 11PM
- Mini-projects: Added content for students in 16.420
- Collaboration: Homework and MPs are for learning; fine to study together but be sure to answer independently
- More info (grading, extensions, office hours) on cat-soop

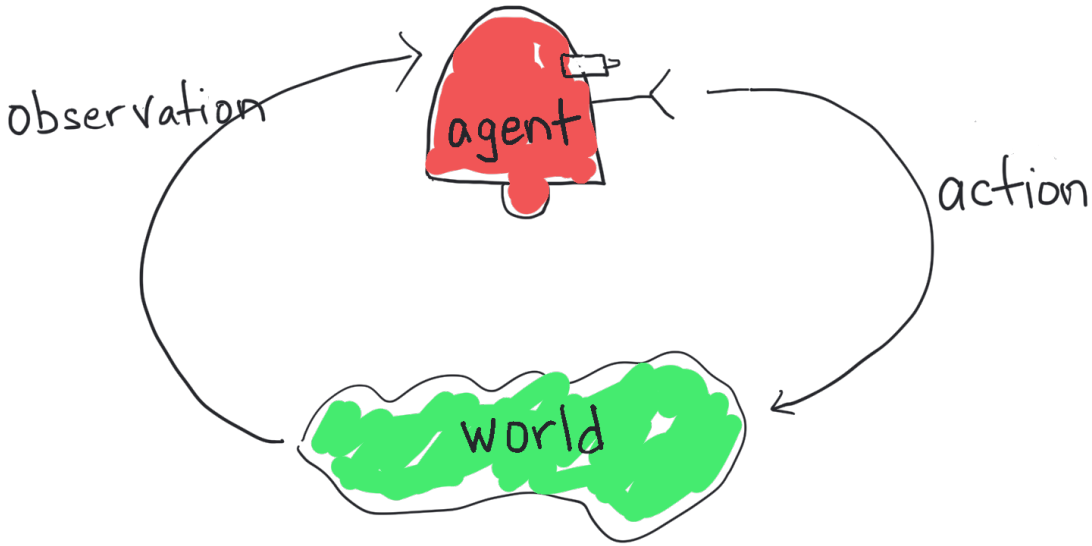
Course content

- Prereqs:
  - Probability (discrete and continuous)
  - Algorithms and Python (you can confidently go from pseudo-code to implementation and debug systematically)
- Focus on knowledge representation and reasoning methods of AI
- Complementary to machine learning

We like books!

- *Artificial Intelligence: A Modern Approach* (AIMA4e), Russell and Norvig `aima.cs.berkeley.edu`
- *Algorithms for Decision Making*, (KAlg), Kochenderfer `algorithmsbook.com`

Agents



- **agent**: program with an ongoing feedback connection to an external environment (e.g., robot, smart house, Siri, banking advisor)
- generally $\pi : (O, A)^* \to A$ is a program with "state" or "memory" that maps the history of actions and observations to the next action

Modeling the environment for an agent

This is the most general case—we'll consider some simplifications

- $\mathcal{S}$: set of world states
- $\mathcal{A}$: set of actions of the agent
- $\mathcal{O}$: set of possible observations the agent can make
- $P(S_{t+1} \mid S_t, A_t)$: transition model : distribution over the world state at time $t+1$ given the world state at time t and the action taken by the agent at time t
- $P(O_t \mid S_t)$: observation model : distribution over observations the agent might make at time t given the world state at time t

Policies for agents

How can we (engineers or nature) find a good policy for an agent? A policy is allowed to have memory—it could perform learning or search or solve differential equations or be made out of rubber bands. Anything (physically realizable) is fair as long as it maps histories of observations into the next action.

- Assume we have a specification (or a lot of examples) of a distribution of worlds where our agent is supposed to operate
- Assume also an objective function that measures how well the policy is performing
- Our job as engineers is to find the best policy, in expectation over possible environments, for this agent. But, how?

How to design policies for agents?

1. Be really smart, think hard, and write a program.
2. Systems-level "offline" rational design
    - Write down space of solutions
    - Write down objective function
    - Use search algorithm to find good solution
      Most applications of reinforcement learning fit here
3. Design of agent that is rational "online"
    - Be somewhat smart and construct a program for a rational agent that:
    - represents its knowledge about the environment and
    - reasons about what actions to take

Rational agents

Principle of rationality: Select actions that will maximize expected future utility

- **expected**: take expectations over uncertainty about current environment state and/or outcomes of actions
- **future**: will consider one-step, finite, and infinite future horizons
- **utility**: Assume the agent has a scalar measure of utility or desirability of states of the external environment

Rationality is not the same as:

- omniscience or clairvoyance
- success

Representation and reasoning

- Representation: information stored in a computer—specific to the problem at hand
  - **syntax**: a "language" for encoding information in a computer
  - **semantics**: an intended relationship between a piece of syntax inside the machine and the state of the world outside the machine
- Reasoning: (inference) semantics-preserving manipulation of a syntactic representation—general-purpose, domain-independent
  - Given observations and prior knowledge, make conclusions about the world state and, from there, about which action to select.
  - *Planning*, specifically, is reasoning about action sequences.

"All our knowledge begins with the senses, proceeds then to the understanding, and ends with reason. There is nothing higher than reason." –Kant

Compositional representation

A compositional representation has:

- syntax that is formed from a small set of primitive components and a set of rules for putting them together
- semantics: meaning of a the whole representation can be formed by combining meanings of the components

"The infinite use of finite means." –Alexander von Humboldt, on language

Representation and reasoning

Knowing some information about:

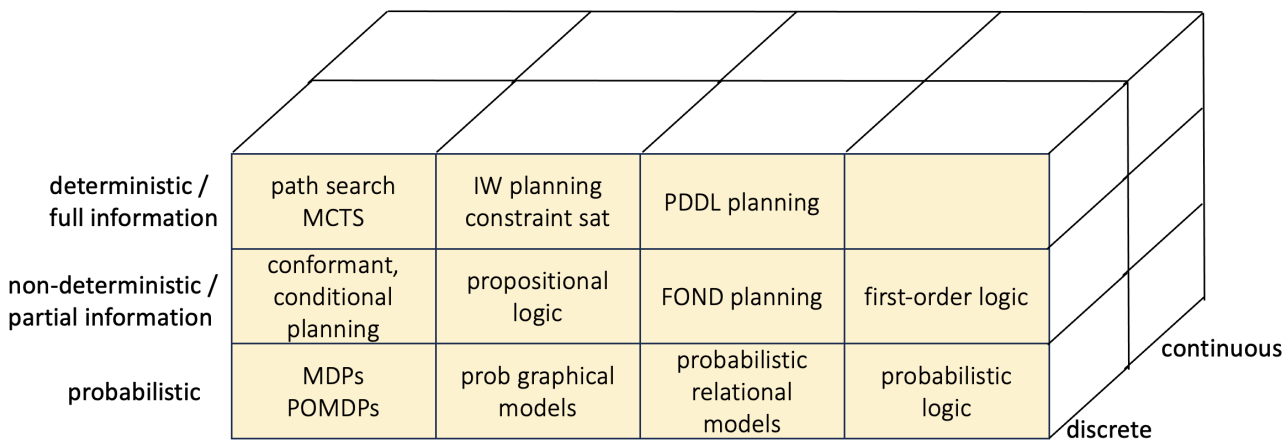
- Current world state
- Transition model
- Objective

Infer one or more of:

- Some aspect of world state
- What action to take next

Methods depend on

- How much information is known about current and future states
- Representational method



Representation styles

How do we describe a state?

- **atomic**: s_{44} (element of some enumeration)
- **factored**: $\langle 4, 5.2, \text{'hippo'} \rangle$ (vector of attributes)
- **relational**: $\{bigger(o_1, o_2), resting(o_2, o_2), size(o1) = 3.2\}$
  (set of properties and relations on "objects"; also, interpretable as a graph)
- **first-order**: $\underline{\text{all the bacteria in the jar are dead}}$
  $\forall x.bacterium(x) \wedge in(x, jar) \rightarrow dead(x)$

Different representations offer different opportunities for

- Expressing partial information
- Generalization

Connections to machine learning

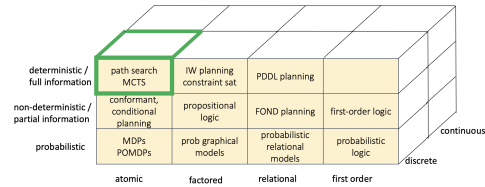
Machine learning is everywhere! Probably most of you have studied it in some form. Many connections to what we will be studying:

- Learning the models / "knowledge" that our search or inference algorithms operate on
- Learn heuristics or other search control information to speed up computation
- Learn problem-specific functions for quickly computing solutions to a space of common queries

Trade-offs between model-based and model-free methods

- Model-based often generalizes much more effectively
- Model-free method not hampered by poor choices of model

First problem setting: fully observable, deterministic



Atomic, discrete

- Agent knows:
  - State set: $\mathcal{S}$
  - Initial state: s_0
  - Action set: $\mathcal{A}$
  - Transition model: $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$
  - Goal set: $G \subset \mathcal{S}$
  - Cost function $C : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$
- We need to find next action to take
- Find plan $a_1, \ldots, a_m$ from s_0 to some state in G such that $T(s_0, a_1) = s_1, \ldots, T(s_{m-1}, a_m) = s_m$ and $s_m \in G$
- Usually we try to minimize

$$\sum_i C(s_i, a_i, s_{i+1})$$

If path costs are not additive, then many algorithmic tricks don't apply and problem is harder.

Measuring problem-solving performance

- Completeness: If there is a solution to your problem, is the algorithm guaranteed to find it?
- Cost optimality: If there is a solution, is the algorithm guaranteed to find the solution with the lowest cost?
- Computational complexity: As the size of the problem grows, how do the computation and time and space requirements grow? The answer to this depends on how we encode the input!
  - In CS algorithms tradition, problems are described as *graph search* problems, and complexity is characterized in terms of the number of vertices (states) and edges in the graph; usually nearly linear in the size of the input.
  - In our applications, we will often have a huge or even infinite S but it is not input to the algorithm. Instead, we provide s_0 and T, and incrementally expose the graph as we search. Characterize complexity in terms of branching factor $|A|$ and depth (also called "horizon" or "plan-length.") Usually exponential in the horizon.

Best-first search framework

- Critical to make a distinction between state (element of $\mathcal{S}$) and node of the search tree, which represents a path from s_0 to some state s. (Every search node has an associated state. It is possible to have multiple nodes with the same state (representing different paths to reach that state.)
- This framework takes a priority function f. Different values of f will yield different search algorithms.

Best-first search framework

BEST-FIRST-SEARCH$(\mathcal{S}, \mathcal{A}, s_0, T, G, C, f)$

```
 1  n = NODE(s_0)
 2  frontier = PRIORITYQUEUE(f)
 3  frontier.ADD(n)
 4  reached = {s_0 : n}
 5  while not frontier.EMPTY():
 6      n = frontier.POP()                          // Get node with lowest f value
 7      s = n.s
 8      if s ∈ G: return n
 9      for a ∈ A:                                  // Expand s
10          s' = T(s, a)
11          path_cost = n.path_cost + C(s, a, s')
12          if not s' ∈ reached or path_cost < reached[s'].path_cost:
13              n' = NODE(s', n, a, path_cost)
14              reached[s'] = n'                    // visit s'
15              frontier.ADD(n')
```

Redundant Paths

- Stupidest possible algorithm (SPA): enumerate all legal paths and pick the first one that reaches a goal state.
- There can be exponentially more paths than states!
- The *reached* data structure (sometimes called a visited list) and test in line 12 ensures that we never consider a path to a state that is higher in cost than the best one we've already found.
- In most of the searches we'll consider, in fact, we can prove that the first time we pop some path to a state off the frontier, we will have done so via a least-cost path, and so we never expand (consider the successor of) a state more than once.

Breadth-First Search

- Best-first with $f(n)$ = number of steps in path n
- First path found to s has fewest steps
- Can actually move the goal test earlier (from s in line 8 to s' just after line 10)
- Complete and optimal (in number of steps)
- Worst case time (and space) complexity $O(|\mathcal{A}|^d)$ where d is the length of the shortest path to the goal. This happens when there are no redundant paths.
- Note independence of $|\mathcal{S}|$
- But, complexity is also bounded by $O(|\mathcal{S}||\mathcal{A}|)$ which in some problems is smaller than $|\mathcal{A}|^d$.

Uniform Cost Search

- Best-first search with $f(n) = n.\mathit{path\_cost}$
- Assume costs are all positive.
- Like breadth-first, but pushes out frontier in equal-path-cost contours
- First path to s expanded has least cost.
- First path to s visited **does not necessarily have** least cost. See text for example of this, and why we cannot move the goal test earlier.
- Worst case time (and space) complexity

$$O(|\mathcal{A}|^{1+\lfloor C^*/\epsilon \rfloor})$$

  where C^* is the cost of the least-cost path and ϵ is the cost of the least-cost action.
- Sometimes called Dijkstra's Algorithm (although Dijkstra's is often used to compute shortest paths to all vertices in a given finite graph.)
- As with BFS, complexity is also bounded by $O(|\mathcal{S}||\mathcal{A}| \log |\mathcal{S}|)$.

Reading and next time

- Read about depth-first search (AIMA 3.4.3) and iterative deepening (AIMA 3.4.4)
- Next time!
  - Informed search methods
  - Monte-Carlo tree search
  - Problems described in terms of rewards instead of costs and goal