Measuring Intelligence
- Thinks and acts like humans
    - Turing test
        - Basic: NLP, Knowledge Representation, Automated Reasoning, ML
        - Full: Vision, Motor Control, Other senses
        - Vision: CAPTCHA
- Thinks and acts rationally
    - Doing the right thing
    - More general, the goal is well-defined

Intelligent Agents
- Perceiving its environment through sensors and acting upon that environment through its effectors to maximize progress toward its goals

PAGE
- Percepts, Actions, Goals, Environment
- See, Think, Do
- Goals, Percepts, Sensors, Effectors, Actions, Environment
- Example: Lane Keeping Agent
    - Goals: Stay in the current lane
    - Percepts: Lane center, Lane boundaries
    - Sensors: Vision
    - Effectors: Steering wheel, Accelerator, Brakes
    - Actions: Steer, Speed up, Slow down
    - Environment: Freeway

Behavior and Performance of Intelligent Agents
- Perception Sequence to Action Mapping
    - What action should an agent take at any point in time
- Performance measure: Subjective measure to characterize how successful an agent is
- Autonomy: What extent an agent is able to make decisions and take actions on its own

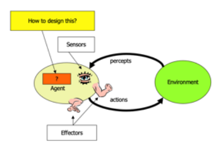
Agent vs Software
- Agents are autonomous, acting on behalf of the user
- Agents contain some level of intelligence
- Agents sometimes act proactively
- Agents have social ability, communicating with user, system, and other agents
- Agents could cooperate
- Agents may migrate from one system to another

Types of Environments
- Accessible (observable) vs Inaccessible (partially observable)
    - Accessible: Sensors give a complete state of an environment
- Deterministic vs nondeterministic
    - Deterministic: The next state can be determined based on the current state and action
- Episodic (History insensitive) vs. episodic (Sequential)
    - Episodic: History does not matter, history does not affect the future
- Hostile vs Friendly

- Static vs Dynamic
  - Dynamic: Environment changes during deliberation
- Discrete vs Continuous
  - Ex: Chess vs. Driving

Types of Agents
- Reflex Agents
  - Reactive: No memory
- Reflex Agents with Internal States
- Goal-based Agents
  - Goal information needed to make decision
- Utility-based agents
  - How well can a goal be achieved (degree of happiness)
  - What to do if conflicting goals
  - What goals should be selected if multiple are available
- Learning agents
  - How can I adapt to the environment
  - How can I learn from mistakes

Summary on Intelligent Agents



- **Intelligent Agents:**
  - Anything that can be *viewed as* **perceiving** its **environment** through **sensors** and **acting** upon that environment through its **effectors** to maximize progress towards its **goals**.
  - PAGE (Percepts, Actions, Goals, Environment)
  - Described as a Perception (sequence) to Action Mapping: $f: \mathcal{P}^* \rightarrow \mathcal{A}$
  - Using look-up-table, closed form, etc.
- **Agent Types:** Reflex, state-based, goal-based, utility-based, learning
- **Rational Action:** The action that maximizes the expected value of the performance measure given the percept sequence to date

Formulating a Problem
1. States Representation
2. Operators/Actions
3. Initial State
4. Goal State

Types of Problems
- Single State Problem
  - Deterministic, Accessible
  - The agent knows everything about the world and can calculator optimal action sequence to the goal state
- Multi-State Problem
  - Deterministic, Inaccessible
  - The agent does not know the exact state
  - Assumes states while working towards goal state

- Contingency Problem
    - Non Deterministic, Inaccessible
    - Must use sensors
- Exploration Problem
    - Unknown state space

Criteria for search algorithms

- Completeness: Does it always find a solution if exists
- Time complexity
- Space complexity
- Optimality

NP vs P problems

- Polynomial vs Non polynomial time problems
- Nondeterministic Polynomial (NP) has an algorithm that can guess a solution and then verify it in polynomial time

Summary

- This Week:
- Problem formulation usually requires abstracting away real-world details to define a state space that can be explored using computer algorithms.
- Once problem is formulated in abstract form, complexity analysis helps us picking out best algorithm to solve problem.

Uninformed Search

- BFS
    - FIFO
    - Shallowest unexpanded node
    - Complete, Time: O(b^d),Space: O(b^d), Optimal
- Uniform Cost
    - The least cost unexpanded node
    - Complete, Time: O(b^d),Space: O(b^d), Optimal
- DFS
    - LIFO
    - Deepest unexpanded node
    - Complete, Time: O(b^m),Space: O(bm), Not optimal
- Depth Limited
    - DFS with max depth set
- Iterative Deepening
    - Complete, Time: O(b^d),Space: O(bd), Optimal
    - Depth Limited Search inside a loop increasing depth until success

Bi-directional Search

- One search from the initial state, one search going up from the goal state, meet in the middle
- Complete, Time: O(b^(d/2)),Space: O(b^(d/2)), Optimal

- Bidirectional search issues
  - *Predecessors* of a node need to be generated
    - Difficult when operators are not reversible
  - What to do if there is no *explicit list of goal* states?
  - For each node: *check if it appeared in the other search*
    - Needs a hash table of O($b^{d/2}$)
  - What is the *best search strategy* for the two searches?

Criterion	Breadth-first	Uniform cost	Depth-first	Depth-limited	Iterative deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	b^(d/2)
Space	b^d	b^d	bm	bl	bd	b^(d/2)
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if l≥d	Yes	Yes

- *b* – max branching factor of the search tree
- *d* – depth of the least-cost solution
- *m* – max depth of the state-space (may be infinity)
- *l* – depth cutoff

Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can be explored using computer algorithms.
- Once problem is formulated in abstract form, complexity analysis helps us picking out best algorithm to solve problem.
- Variety of uninformed search strategies; difference lies in method used to pick node that will be further expanded.
- Iterative deepening search only uses linear space and not much more time than other uniformed search strategies.

Informed Search
- Best First Search
  - Estimate the “desirability” of a node and expand the most desirable node
- Greedy Search
  - h(n) is heuristic which is an estimation of the cost to the goal

        - Expands the first node that appears to be the closest to the goal (least future cost or h(n))
        - Complete with loop checking, Time: O(b^m),Space: O(b^m), Not Optimal
    - A* Search
        - Avoid expanding paths that are expensive
        - f(n) = g(n) + h(n)
        - Admissible heuristic - Never overestimates the goal

<u>Function Optimization</u>

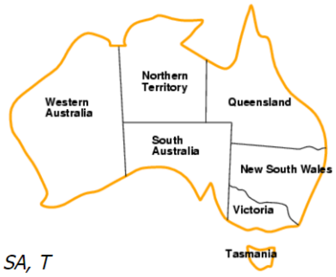
- Iterative Improvement
    - Keep a single current state and try to improve
- Hill Climbing
    - Maximizes the value of the current state by replacing it with a successor state that has the highest value as long as possible
- Simulated Annealing
    - From the current state, pick a random successor state
    - If the successor state is better, pick it, if it’s not, flip a coin and accept transition depending on a coin flip
    - A mix of greedy search and random search based on temperature
- Genetic Algorithm
    - How large is the population? How do you select the initial population? How will you cross-breed the population? How will you mutate the population?

Summary

- Best-first search = general search, where the minimum-cost nodes (according to some measure) are expanded first.
- Greedy search = best-first with the estimated cost to reach the goal as a heuristic measure.
  - Generally faster than uninformed search
  - not optimal
  - not complete.
- A* search = best-first with measure = path cost so far + estimated path cost to goal.
  - combines advantages of uniform-cost and greedy searches
  - complete, optimal and optimally efficient
  - space complexity still exponential
- Hill climbing and simulated annealing: iteratively improve on current state
  - lowest space complexity, just O(1)
  - risk of getting stuck in local extrema (unless following proper simulated annealing schedule)
- Genetic algorithms: parallelize the search problem

<u>CSP</u>

- Variables, Domains, Constraints



Variables: *WA, NT, Q, NSW, V, SA, T*

Domains: D_i = {red, green, blue} (one for each variable)

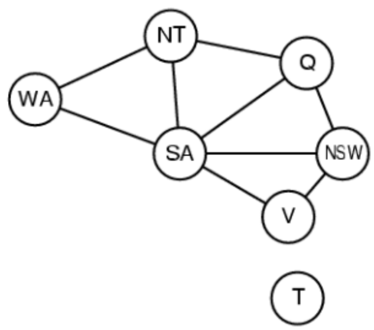
Constraints: C_i = <*scope*, *rel*> where *scope* is a tuple of variables and *rel* is the relation over the values of these variables

- E.g., here, adjacent regions must have different colors
  e.g., WA ≠ NT, or (WA,NT) in {(red,green), (red,blue), (green,red), (green,blue), (blue,red), (blue,green)}

- Consistent Assignment: Assigned values do not violate any constraints
- Complete Assignment: Every variable is assigned a value
- DFS in CSP with Unary Constraints is called Backtracking Search

<u>Constraint Graph</u>

- Each constraint is an arc
- Nodes are variables



<u>Constraint Types and Varieties</u>

- Unary Constraint
  - Single variable
- Binary Constraint
  - Pairs of Variables
- Higher Order Constraint (Global)
  - 3 or more variables

<u>Backtracking</u>

- Most Constrained Variable
  - Fewest legal value (Minimum remaining value heuristic)
- Most Constraining Varaible
  - Variable with the most constraints on remaining variables (Degree heuristic)
- Least Constraining Variable
  - One that rules out the fewest values in the remaining variables

<u>Forward Checking</u>

- Keep track of remaining legal values for unassigned variables
- Terminate search when any variable has no more legal values

<u>Node and Arc Consistency</u>

- A single variable is node constraint if the node is consistent if all values in the domain satisfy unary constraints
- A variable is arc-consistent if every value in its domain satisfies binary constraints
- A Network is arc-consistent if every variable is arc-consistent with each other
- Arc consistency algorithms

AC-3 Algorithm

- Start with a queue that contains all arcs
- Pop one arc and make the rest consistent
    - Check all arcs that are affected by that
- O(n^2d^3) n variables, d values

Summary

- CSPs are a special kind of search problem:
    - states defined by values of a fixed set of variables
    - goal test defined by constraints on variable values
- Backtracking = depth-first search with one variable assigned per node
- Variable ordering and value selection heuristics help significantly
- Forward checking prevents assignments that guarantee later failure
- Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies
- Iterative min-conflicts is usually effective in practice

A game as Search Problem

- Initial State, Operators, Terminal State, Utility Function

Minimax Algorithm

- Perfect play for deterministic environments with perfect information
- **Basic idea:** choose move with highest minimax value
  = best achievable payoff against best play
- **Algorithm:**
    1. Generate game tree completely
    2. Determine utility of each terminal state
    3. Propagate the utility values upward in the three by applying MIN and MAX operators on the nodes in the current level
    4. At the root node use minimax decision to select the move with the max (of the min) utility value
- Steps 2 and 3 in the algorithm assume that the opponent will play perfectly.

- Alpha Beta Pruning
    - Alpha is for max, Beta for min

Alpha Beta Pruning In Depth

- Alpha and Beta are Negative Infinity and Infinity respectively
- Find Max of the left-most branch and update v and alpha
    - Update the beta of the parent node
    - Repeat for every branch and prune as needed

- Go to the next branch and carry alpha and repeat

Non Determinist Games

- Like Backgammon
- Expectiminimax (Probabilities)
    - Expectimin: Add up both values
    - Expectimax: Average both values

State, Action, and Sensor Model

- Actions, Percepts, States, Transition, Appearance, Current Model
- Can be non deterministic

Little Prince's Model

$$\begin{aligned} A &\equiv \{\text{forward, backward, turn-around}\} \\ P &\equiv \{\text{rose, volcano, nothing}\} \\ Z &\equiv \{s_1, s_2, s_3, s_4\} \\ \phi &\equiv \phi(s_0, \text{forward}) = s_3, \phi(s_0, \text{backward}) = s_2, \ldots \\ \theta &\equiv \theta(s_1) = \{\text{volcano}\}, \theta(s_0) = \{\text{rose}\}, \theta(s_2) = \theta(s_3) = \{\} \end{aligned}$$

Hidden Markov Model

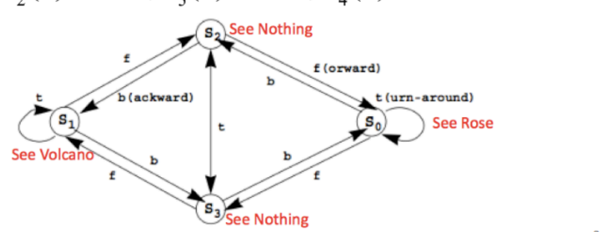
- Actions, Percepts, States, Appearance (State -> Observations), Transitions, Current State

The HMM for Little Prince's Planet

(with uncertain actions and sensors)

$$\begin{aligned} A &\equiv \{\text{forward, backward, turn-around}\} \quad \{f, b, t\} \\ Z &\equiv \{\text{rose, volcano, nothing}\} \\ S &\equiv \{s_1, s_2, s_3, s_4\} \\ \phi &\equiv \{P(s_3|s_0,f)=.51, P(s_2|s_1,b)=.32, P(s_4|s_3,t)=.89, \ldots\} \\ \theta &\equiv \{P(\text{rose}|s_0)=.76, P(\text{volcano}|s_1)=.83, P(\text{nothing}|s_3)=.42, \ldots\} \end{aligned}$$

$$\pi_1(0)=0.25, \pi_2(0)=0.25, \pi_3(0)=0.25, \pi_4(0)=0.25$$



- Lookup tables for Transition Probabilities, Appearance Probabilities, and Initial State Probabilities
- Markov Assumption: History doesn't matter

Rewards and Utilities

- 3 types of rewards
    - Being at a state
    - Doing an action on a state
    - Making a transition

- Every state may have a utility value

Partially Observable Markov Decision Process (POMDP) and Markov Decision Process (MDP)

- Partially Observable: The agent doesn't see the states, only percepts
- MDP: State and Actions, Initial State and Probability Distributions, Transition Model, Reward functions
- POMDP: State and Actions, Transition Model, Reward function, Sensor Model, Belief of current state

Maximum Expected Utility (MEU) and Rational Agents

$$EU(a|e) = \sum_{s'} P(result(a) = s' | a, e) U(s')$$

Overview

- Goals are given to the agent
- Rewards are given to the agent
- Utility values are computed by the agent based on rewards
- Policies are computed or learned by the agent and used by the agent to select actions
- Utility value: Sum of all future rewards
  – Add the rewards as they are
  $$U_h = ([s_0, s_1, s_2, \ldots]) = R(s_0) + R(s_1) + R(s_2) + \ldots$$
  – Discount the far-away rewards in the future
- $$U_h = ([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \ldots$$

State Utility Value Iteration

- Bellman Equations
  – For n states, there are n equations must be solved *simultaneously*
  $$U^*(s) = R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U^*(s') \quad (17.5)$$
  Bellman iteration:
  – Converge to $U^*(s)$ step by step
- $$U_{i+1}(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U_i(s') \quad (17.6)$$

  In POMDP, where sensors are uncertain, we must average over all possible evidences for the next state *s'* (see the last term below):
- POMDP:
  $$\alpha_p(s) = R(s) + \gamma \left(\sum_{s'} T(s, a, s') \sum_e P(e | s') \alpha_{p.e}(s') \right)$$

Optimal Policy

- $$\pi_s^* = \underset{a}{\operatorname{argmax}} \sum T(s,a,s')U^*(s')$$
- Pick the highest utility values
- Calculate optimal policy after completely calculating all utility values or compute policy incrementally every iteration where the utility values are updated

Summary for Uncertain Environments

- POMDP is very (the most) general model
  - Deal with uncertainty by
    - action models and sensor models
- Incorporate goals -> rewards -> utilities -> policy
  - Utilities and policies can be computed from rewards
    - Systematically (Bellman equations)
    - Iteratively (Bellman's iteration algorithm)
  - Solve a problem by following a good policy
    - One policy for one problem/goal
    - Different policies are needed for different goals

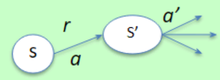
<u>Reinforcement Learning</u>
- Try different actions in states to discover an optimal policy and eventually tell the agent which way to go
- 2 General Classes
  - Model-Based RL
    - Learn the transition model, utility values, then policy
    - Learn approximate model based on random actions
  - Model Free RL
    - Learn policy without learning explicit transition model but from samples from the environment

<u>Monte Carlo</u>
- Generate a fixed policy based on the utility function
- In each iteration, the learner follows policy starting at a random state
- Generate policy with policy iteration
- Fixed Utility values

<u>Temporal Difference</u>
- Improved on Monte Carlo
- Updates Utility values with each sample

<u>Q Learning</u>
- Use Q value to represent the value of taking action a in state s

- Initially, let $Q(s, a)$=0, given α and γ
- Sample a transition and reward: (s,a,s',r)
  - Sample = r + γ $max_{a'}$ Q(s',a')
  - Q(s,a) = (1-α)Q(s,a) + α*Sample



$$Q_{t+1}(s,a) = (1-\alpha)Q_t(s,a) + \alpha[R(s,a,s') + \gamma \max_{a'} Q_t(s',a')]$$

-

Q Learning Summary

Modify Bellman equation to learn Q values of actions

- $U(s) = R(s) + \gamma \max_a \sum_{s1} (P(s_1 | s,a) U(s_1))$
  - We don't know R or P
  - But when we perform a' in s, we move to s' and receive $R(s)$
  - We want $Q(s,a)$ = Expected utility of performing a in state s

Update Q after each step

- If we get a good reward now, then increase Q
- If we later get to a state with high Q, then increase Q here too
- $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(R(s) + \gamma \max_{a'} Q(s',a'))$
  - α is the learning rate, γ is the discount factor

Converges to correct values if α decays over time

- Similar to the "temperature" in simulated annealing

<u>State Action Reward State Action (SARSA)</u>

Modify Q learning to use chosen action, a', not max

- $Q_{t+1}(s,a) \leftarrow (1-\alpha)Q_t(s,a) + \alpha\ [R(s,a,s') + \gamma\ \cancel{\max_{a'} Q_t(s',a')}]$
- $Q_{t+1}(s,a) \leftarrow (1-\alpha)Q_t(s,a) + \alpha\ [R(s,a,s') + \gamma\ Q_t(s',a')]$

On-policy, instead of off-policy

- SARSA learns based on real behavior, not optimal behavior
  - Good if real world provides obstacles to optimal behavior
- Q learning learns optimal behavior, beyond real behavior
  - Good if training phase has obstacles that won't persist

Q Learning & SARSA

Converge to correct values

- Assuming agent tries all actions, in all states, many times
  - Otherwise, there won't be enough experience to learn *Q*
- And if α decays over time
  - Similar to simulated annealing

Avoids the complexity of solving an MDP

- MDP requires solving for the optimal policy before starting
- An RL agent can start right away and then learn as it goes
  - Although this might be wasteful if you already know *P* and *R*

<u>Exploitation vs Exploration</u>

- Similar to simulated annealing, explore sometimes and exploit the policy other times