Office hours: Mon. 1-2:30 TSRB 228
-Data structures
-Computability theory NP-hard
-40% 4 programming (Python 2.7) projects
-60% 2 tests
-Ungraded exercises

Solve NP-hard problems — algorithms

Turing test
-Judge connected to a separate person pretending to be a machine, and a machine
-Can ask questions of the two entities, and if you can't guess which is human/machine then the machine is intelligent

Strong AI - emulate human behavior (beat the Turing test), <u>broad AI</u> (does many tasks)
Weak/Narrow AI - solve specific problems humans wanted to solve
human-level, superintelligence

Machine learning - automated discovery of patterns in data + make decisions based on it
- big data/big iron

Neural networks - started improving recently

Cognitive Revolution - instead of taking input and producing <u>behavior</u>, computers might want to <u>think</u> about the inputs

Neural Net 1943 - mimic human brain

well-defined problems: know all possible moves (chess/checkers, proofs with certain symbols) —> easier to solve by brute force

Knowledge-based AI: brittle b/c can't work with info that it cant diagnose (medical diagnosis)

ImageNet: computers learning to recognize images

Neural nets acheived 85% success rate —> efficient b/c <u>can run in parallel</u>

Agent - anything that perceives its envt through sensors and acts on its envt through effectors
-interested in how to process perceptions into output actions

Agent function - mathematical description of agent's behavior in resp. to envt., mapping sensory perceptions to effector actions
Agent program - concrete implementation of agent fcn. (code)

~~Tabulation: list percept sequences, map to actions~~ <—— inefficient for complex agents (naive approach)

House with two rooms, can be in A or B
- Actions: move right, move left, kill person
- Sensors: which room, people
  - Evaluate actions according to goal of the agent (different resp. based on objective)

Objective function: defines what the goal is
-Maximize/minimize a value
-Take into account costs of and restrictions on actions
-exploration/exploitation tradeoff (look at more things vs. focus on few things)
-take into account unreliable sensors or effectors

Observability of agent's environment
-Fully observable if can sense everything in envt. without error
-Partially observable if limited sensors or randomness makes some info unsure

Determinism
-Deterministic envt only changes when agent acts, and exactly as desired
-Stochastic envt. can change at other times due to randomness, other effectors
  -partially observable worlds are sensor-stochastic
-Action determinism vs. sensor determinism

Static vs. Dynamic
-Static world doesn't change when agent is thinking
-Dynamic world changes while you think

Discreteness
-World broken up into discrete pieces vs. continuous/infinite gradation of values
-Discrete vs. continuous actions, perceptions, time

Episodic: history doesn't matter vs. **sequential**: history matters (take future into account)
-Past events don't affect current decision

Single-agent vs. **multi-agent** envt. (cooperative or competitive)

Goal-Based Agents (Search)
-Solving sequential problems (affect future decisions)
-<u>State</u>: unique config of relevant facts of the envt. from sensors
    -<u>Goal state</u> is desired state for agent —> want to <u>figure out how</u> to reach goal and then do it
-**Need representable state (init./goal), set of acceptable states, function that accepts states

Computer sees graph with costs on different paths —> doesn't have pathfinding intuition

Use search when you <u>don't have a better algorithm</u> (memory-inefficient, etc.)

Picking an algorithm: uninformed/informed search, adversarial search, constraint satisfaction, conformant, Markov decision processes

Search problem: find a <u>sequence of actions</u> that transforms envt. from init. state to a goal state

<u>state space</u> - collection of possible states, linked to each other

<u>Successor function</u>: Creates new states from old, given set of actions
    -Successors (single state s, set of actions a) —> s' (new states)
    -Different algorithms pick successors in different ways
——>
-Completeness: Always finds a solution if one exists **OR** can visit all states in state space
-Time complexity: Big-O, # of states created by successor fcn., worst case/average case
-Space complexity: # of states stored in memory at one timer
-Optimality: finds lowest cost/shortest solution

```
successor(state, actions)
        new_states = empty set
        foreach a in actions
                if a can be done to state
                        add a(s) to new_states
```

1. Pick state
2. Generate successors
3. Repeat until goal
4. Remember which actions led to which state —> execute

Issues: inefficient (extra states visited), infinite search, not optimal, incomplete
Fixes: remember where you've been, be systematic

Generic Search Algorithm
operators = [… list of actions …]
closed = nil (where you've been)
open = initial state (states we know about but haven't reached)
current = initial state

while (current isn't the goal and open != null)
    closed = closed + current
***** open = open - current union (successors(current, ops) - closed) add succ., remove closed
    current = first(open)

Don't repeat elements in open list!

if current is the goal, success, else failure

Append to the end —> open list is a queue, breadth-first search
Append to front —> open list is a stack, depth-first search
Open list is priority queue —> A*, uniform cost search

Breadth-first: Successor fcn. gives state in alphanumeric order
- -Solutions close to initial state
- -Goal found —> move back up decision tree for path (remember parents)

Depth-first:
- -Follows a path as long as possible without looping
- -not optimal b/c doesn't account for multiple paths to goal

Smaller closed list —> more efficient process

	BFS	DFS
Complete	Complete	Yes if finite, no if infinite
Time Complexity	Number of nodes expanded (successive function calls) —> branching factor ^ depth (branching factor = avg. # successors)	branching factor ^ m, m is maximum depth search will go to (generally m > depth)
Space	Keep all nodes	If state space is tree shaped, keep only ancestors (otherwise keep everything)
Optimality	yes (shortest # moves)	no

Tradeoff: Optimality vs. space/time complexity

Action Cost
Uniform cost search (UCS) - instead of least actions, lowest total cost —> **open list is a PQ**

-g(a) - cost of moving from initial state to current state using shortest known path (minimize this)

Explore all g values starting with first element in PQ, then moves onto later ones

Note: override a node's value if it has a lower one as a child of a different parent

This is **DIJKSTRA'S ALGORITHM** —> **Optimal** and **Complete**

Informed Search
Don't want to spend time looking at nodes close to init that are not intuitively efficient (ex. moving left first to get to a place on the right)

Heuristic function h() - gives an estimate of how far S is away from the goal
- -Should operate in O(1), O(n), O(n^k)
- - Ex: Euclidean distance on map

Greedy Best First Search
Sort open list (PQ) using h(s) (smaller h(s) is better) instead of g(s)
- -Estimate instead of actual costs

no admissibility required

Remove item from open list, add successors with h(s) values to priority queue

Complete: yes
Time/space: same as before, but good h() will make it better
Optimality: not optimal

******Best First Search******
Merge UCS and greedy best first

Sort open list PQ on f(s) = g(s) + h(s) instead of just h(s)

Complete: yes
Time/space: exponential, but good h() helps
Optimal: depends on h()

Shortcut!
-Resort open list + revisit all descendants
- -Shortcut created by "bad" heuristic

A*

-Best first search w/heuristic guaranteed never to create shortcuts
-Prove that h() is <u>admissible</u> (never overestimates distance to goal)

f(s) = g(s) + h(s) but now h() must be admissible

***Not guaranteed to find optimal solution! (h values might be too big)

Let h*(s) be the **true cost** of state s to the goal
Also f*(s) = g*(s) + h*(s) (perfect heuristic)

Heuristic is **admissible** if h(s) <= h*(s) for all s (heuristic never overestimates)
h(s) = 0 is admissible, but doesn't help at all —> UCS

A* is optimal (given h is admissible) —> prove admissibility

Want admissible heuristic that is close in approximation

```
add start to openSet
while openSet isn't empty
	current = pop from openset
	if current == goal
		return reconstruct_path(current)
	closedSet.add(current)
	for every neighbor of current
		if neighbor is in closedSet, continue for loop
		gScore = current.gScore + heuristic(current, neighbor)
		if neighbor not in openset
			openSet.add(neighbor)
		else if gScore < openSet.get(neighbor).gScore)
			openSet.replace(openSet.get(neigbbor), neighbor)
```

Informedness

-More informed: gives value closer to h* than other heuristics
size of total search space / avg. # states explored with heuristic h (want high informedness)

ha dominates hb when hb(n) <= ha(n) <= h*(n) (hb underestimates more)

Consistency: always increasing or decreasing relative to neighbors
for all s1, s2
h(s1) - h(s2) <= k(s1, s2)   diff between heuristics less than actual cost between states

Creating a heuristic: relax problem conditions, then re-add after finding a heuristic

<u>Search</u> (Randomized optimization)
heuristic, but no goal —> how to find best final state?
    <u>don't care about path</u>
Each state is a representation, that can be searched

<u>Naive approach</u>: generate representation at random and test how well it does
    - Issues: large hypothesis space, doesn't <u>learn</u>

Otrhr approaches: Hill-climbing/greedy search, simulated annealing, genetic algos

<u>Hill-Climbing</u>
y-axis is heuristic, x-axis is representations

Choose random selection from space
Initial score = heuristic(selection)
prevScore = 0
while score > prevScore
    prevScore = score
    Go through neighbors, choose highest (best) neighbor until you reach the top of the hill

If neighbors are equal, depends on neighbor function

local maxima - point with only worse neighbors
global maxima - point better than all points

Ex:
Representation: 6 bit strings
Heuristic: Shared bits with 101010
Neighbor function 1: flip pairs of adjacent bits
Neighbor function 2: flip any two bits

Use neighbor functions to find maxima with 010101, 000100, 110000

<u>Finding global max</u>
Hill climb with random restarts (not guaranteed/efficient)
    Works poorly in hilly or skewed sets

<u>Simulated Annealing</u>
Sometimes take random steps at beginning of search, less as you continue
(**See slides for pseudocode**)

Temperature = 0 —> hillwalk, infinity —> random walk
    -Choose a **good schedule** to make algorithm work well

<u>Genetic Algorithms</u>
Instead of walking from one pt., go from many points across search space —> make better jumps

Initialization —> random mutation —> informed jumps to find global max
Mutation —> reproduce (crossover) —> evolve

mutate - replace member with random neighbor
crossover - mix two members based on heuristic value
    population >= 2n
reduce - take the n best items after crossover

Grid World example
population size: 10 (arbitrary)
heuristic: Manhattan distance from goal - cost
mutation: change an action in the sequence
crossover: first half of one plan, another half of second plan
reduce: remove worst plans

<u>TEST: print notes</u>
- short answer <u>based on</u> definitions
- worked problems (search, other algorithms)

<u>optimization search</u>
need to know end (maxima) + don't care where we start or how we get there

evaluation fcn. - how good a state is (in absolute terms)

Random hill climbing - pick nearby state that is higher up until you reach a peak
Random hill climbing w/restarts - take max of multiple runs

<u>Simulated Annealing</u>
always go up, unless a successor less than you appears —> go down with prob. proportional to how much less it is
-temperature - how willing you are to make a risky move (move to a lower state)
    get less jumpy as algorithm continues
    - slow temperature decrease —> **guaranteed to find goal**

Algorithm: generate successor from neighborhood
    if e(succ) > e(current) —> current becomes successor
    if e(succ) < e(current) then maybe current = successor
    Reduce temperature
    Repeat

Genetic Algorithms
parallel hill climbing with sometimes information sharing

mutation - generate local successors
crossover - swapping information and make big jumps

Search problems w/Action stochasticity (**uncertainty** about outcomes of actions)
Episodic: solve with utility theory

Utility theory: given action a with results result; (a) for i = 1 to N

Expected utility(a) = sum from i = 1 to N of P(resulti(a) * U(resulti(a))

multiply probability * utility (how likely result is * how good it is)

*** **Utility of state is not clear in sequential problems**

Imperfect actions (ex. gridworld):
    80% chance of doing what you want, 10% of drifting left, 10% of drifting right

If we fail: replan (but this is expensive!)

Policy: P: s —> a is a function that maps all possible states to best action from that state

Markov Decision Process:
1-Markov assumption: to compute best option, only need 1 piece of historical info

Creating a policy:
    S: set of states - sink states are those you enter + can't leave (including goals)
    s0: initial state

Transition fcn. T(s, a, s'): probability of going from s to s' if action a is taken
    —> create transition table for each action

Reward fcn. R(s) - produces number from state (how good that state is)
    -Sparse (don't get rewarded often)
    -Defines optimality of behavior

Maximize reward by trying to move to state with higher reward if possible
    Give some states high reward, and make them sink states —> goes to those states

Optimal policy - gets the agent highest cumulative reward

Pi*(s) = argmax(sum of all states T(s, a s') * U(s'))
-find argument (action) that gives maximum sum of transition * utility of successor
-don't know utility, but do know reward

Utility: places might have the same score (0), but being closer to goal is obviously better
Want proximity to future reward

sink states - R(s) = U(s)

Additive utility - U(s0) = U([s0, s1, s2… sn]) = R(s0) + R(s1) + … R(sn)
n states in the future
for all possible future sequences.

Discounted utility - U(s0) = U([s0, s1 …]) = R(s0) + gamma * R(s1) + gamma^2 * R(s2) …
-gamma is discount factor (0 < y <= 1)
-Trust nearby states, discount future states

Bellman Equation - use discounted utility
U(s) = R(s) + gamma * max of A(sum of T(s, a, s') * U(s'))
-current reward + single-step discount * max(all possible actions: likelihood to get there * utility)
-**recursive!!**
-utility of successors is based on their reward + utility of their successors

Solving recursion: use value iteration
-Intuition: start with random utilities and incrementally update until they reach right answer

Updating Bellman: U_{i+1} = R(s) + gamma * max of A (sum of T(s, a, s') * U(s'))
-where Ui is guess of utility for state s after i iterations
-keep repeating function until right U is converged on