LECTURE 2

reflex agent
  responds to input without thinking about consequences or making plan
  could be optimal, but typically not
planning agent
  decisions based on hypothesized actions

search problem
  state space
    list of possible states
  successor function
    returns actions and costs for a given state
  start state
  goal test
    may not be state dependent ("are all food pellets gone?")

world state
  includes all details about environment (anything changeable)
search state
  includes only details needed for planning
  e.g. k agents visiting N landmarks, minimizing the sum of the tour times for all agents
    states: locations of agents and (unordered) set of landmarks visited by any agent
    actions: one or more agents moving to a different location
    cost function: sum of time traveled by each agent during this action
    goal test: all landmarks visited?

state space graph
  nodes are abstracted world configurations
  transitions are actions
  goal test is a set of nodes (or a single node)

search tree
  rooted at start state
  children correspond to possible futures (successors)

each node in the search tree corresponds to a path through the state space graph
  to go from graph to tree, figure out all possible paths from S to G

complete search
  guaranteed to find solution if one exists
  may not return optimal solution

search tree with branching factor b and maximum depth m
  1 node at first level, b at next, b^2 at next, ... , b^m at last
  O(b^m) total nodes
  solutions at various depths

DFS = tree search with LIFO stack as fringe
  could expand entire tree, so takes O(b^m) time
  takes O(bm) space
    at each level, we can have b nodes expanded
    there are at most m levels
  not optimal; finds leftmost solution (given left-to-right search tree processing)

BFS = tree search with FIFO queue as fringe
  let s be level of shallowest solution
  search takes O(b^s) time
  takes O(b^s) space
    roughly the number of nodes in the last tier (s)
  optimal in terms of minimizing number of actions

iterative deepening
  get dfs space advantage with bfs time/shallow-solution advantage
  run dfs with depth 1, then depth 2, etc.
  small fringe, but repeat work

UCS = tree search with increasing equal-cost contours
  let C* be cost of solution, eps be minimum edge cost
  effective depth is C* / eps
  takes O(b^(C* / eps)) time
  takes O(b^(C* / eps)) space
  optimal in terms of cost

PROJECT 1

UCS queue priority = hypothesized total distance from start, not just distance from closest expanded node

building the optimal path during A* or UCS graph search
  create prev_map dict mapping states to their predecessors
  when expanding a state, store the predecessor in prev_map
    out of all expanded states, there can be multiple predecessors
    store whichever predecessor enqueued this particular copy of the state
  after goal is reached, trace back through the prev_map until the beginning is reached

heuristics
  remember that expanding a search tree node is different than just being at a location
    other state variables (like corners remaining) matter too!
  heuristic that calculates cost to closest goal + cost from that goal to next closest goal + ... can be inadmissible
    it may be better to pick a farther goal for step 1 and then loop back to an initially closer goal
  the heuristic should as closely approximate the (overall) cost as possible
    for collect-all-dots, max(manhattan distance to all goals)

sometimes greedy search is a easy-to-compute and almost-optimal alternative to A*

if UCS and A* ever return paths of different lengths, heuristic is inconsistent

HOMEWORK 1

to do A* by hand
  expand as in ucs, but don't forget to add heuristic values when calculating costs

LECTURE 3

conceptually, search algorithms all use priority queues
  practically, bfs and dfs can use queues and stacks to optimize (no log-time cost)

stop when goal is dequeued, not when enqueued
  intuition: when goal is dequeued, all other alternatives are worse

UCS is complete and optimal, but it explores in all directions, knowing nothing about the goal

heuristic function, h(x)
  estimate of distance from state to a goal
  high heuristic value means high estimated distance
  h(A) = 0 at a goal A

greedy search
  always expand to fringe node with lowest heuristic value
  complete but not optimal
  common case: end up at wrong goal in search tree (because cannot "plan" routes)
    remember that a search tree usually has many goal states, even if the search graph has just one
  worst case: like badly guided dfs

A* search
  UCS orders by path cost, or backward cost g(n)
    total true distance between start node and n
  greedy search orders by goal proximity heuristic, or forward cost h(n)
    estimated cost between n and nearest goal
  A* expands uniformly by f-value, where f(n) = g(n) + h(n)

admissible heuristic
  h(x) is admissible if, for all nodes n, 0 <= h(n) <= true cost from n to nearest goal
  often are solutions to relaxed problems (more successor options)
    e.g.: pacman with no walls (manhattan distance)

A* tree search is optimal with an admissible heuristic
  https://youtu.be/gz9OEtQWDM0?t=40m18s

semi-lattice of heuristics
  dominance
    h1 dominates h2 if h1(n) >= h2(n) for all n
    max of admissible heuristics is admissible
  at top: exact heuristic (optimal cost from any node)
  at bottom: trivial heuristic (zero everywhere), which is dominated by all others
  more towards the top means fewer search states expanded

graph search
  never expand a state twice
  store a closed set of expanded states and only expand states not in it

consistent heuristic
  heuristic arc cost <= actual arc cost
    from A to C, h(A) - h(c) <= dist(A, C)
  consequence: f-cost along a path never decreases (monotonicity)

A* graph search is optimal with a consistent heuristic

LECTURE 4

constraint satisfaction problem (CSP) = special kind of search problem
  state representation = N variables with values from a domain D
  goal test = set of constraints specifying allowable assignments
  start state = empty assignment
  successor function = assign value to unassigned variable

example of CSP: map coloring
  n variables: states {MD, VA, DC, ...}
  d domains: {red, blue, green}
  constraints: adjacent regions must have different colors
    implicit (rule of some kind):
      MD != VA
    explicit (enumerates all possible valid assignments)
      (MD, VA) is an element of {(red, green), (blue, green), ...}

example of CSP: sudoku
  variables: open squares
  domains: {1,2,...,9}
  constraints: 9-way all-different for each column, row, and region

constraint varieties
  unary = single variable (e.g. DC != green)
  binary = two variables (e.g. DC != MD)

backtracking search = basic uninformed search for CSPs
  dfs with two modifications
  one variable at a time
    assign to a single variable at each step
  check constraints as you go (incremental goal test)

consider only values which do not conflict with previous assignments
if no possible valid moves forward, backtrack (return to prior level)

filtering (forward checking) for backtracking
keep track of domains for unassigned variables
cross off values that violate a constraint when adding to the existing assignment
doesn't provide early detection for all failures

arc consistency
arc X->Y is consistent if for every x in the tail there is some y in the head which is can be validly assigned
X and Y are variables
x and y are possible values (in the domains of X and Y)
how to check an arc for consistency
look at items at tail
remove any that, if chosen, would make the head impossible to satisfy
a CSP is arc consistent if all arcs are consistent
check all binary constraint edges in both directions

backtracking with arc consistency
after each assignment, enqueue all binary constraint arcs in both directions and enforce in order
if a domain changes, add all binary constraint arcs leading to that variable to the queue
after a run of arc consistency:
if each domain has multiple values left, continue searching this subtree
if each domain has exactly one value left, this is a solution
if any domain is empty, this branch has no solution, so backtrack
detects failures earlier than filtering, but detecting all possible futures is NP-hard

ordering = choosing which variable comes first (or which assignment to check)
minimum remaining values (MRV)
choose the variable with the fewest legal values left in the domain (most constrained variable)
least constraining value (LCV)
when multiple values can be assigned, choose value that rules out the fewest values in other variables

LECTURE 5

degrees of consistency
1-consistency (node consistency)
each node has a value that meets the node's unary constraints
2-consistency (arc consistency)
for each pair of nodes, any consistent assignment to one can be extended to another
k-consistency
for every k nodes, any consistent assignment to k-1 can be extended to the k-th
strong k-consistency = also k-1, k-2, etc. consistent
strong n-consistency means can solve without backtracking
choose any assignment to any variable
choose a new variable and by 2-consistency, there is an assignment consistent with the first
choose a new variable and by 3-consistency, there is an assignment consistent with the first two, etc.

independent subproblems in CSP are connected components in graph

tree-structured CSPs (graphs have no loops)
solvable in O(nd^2) time, whereas general CSP has worst case O(d^n) time
algorithm
choose a variable as root, form directed tree, and linearize
make binary constraint arcs point only in the forward direction (right)
remove backward:
for each node in reverse linear order, enforce all arcs to that node
assign forward (no backtracking needed!):
assign in order from start of linearization to end

cutset conditioning
method of making nearly tree-structured CSPs into tree-structured CSPs
algorithm
choose a cutset
instantiate the cutset (all possible ways)
for each assignment, compute residual (remaining) CSP
solve each residual CSP, which is now tree-structured

min-conflicts algorithm
assign a value to each variable
while at least one constraint is violated:
randomly choose a variable that is violating a constraint
assign a value in its domain that minimizes total constraints violated

LECTURE 6

value/utility (of a state) = best achievable outcome from that state
generally the max of the values of the children states

deterministic zero-sum game
players alternate turns
one player minimizes result, other maximizes result

minimax
terminal states have values
minimax value = best achievable utility against an optimal adversary
recursively computed
dispatch function uses max-value function if maximizer's turn, else min-value
max-value function takes state and chooses successor of highest value
min-value function takes state and chooses successor of lowest value
same complexity as exhaustive DFS
O(b^m) time
O(bm) space

in real games, searching to leaves is impossible because of resource limits
solution: limit depth of searches
use evaluation functions to decide utilities of non-terminal states
ideal evaluation function is actual minimax value of position
in practice, usually weighted sum of features

alpha-beta pruning
alpha = maximizer's best option on path to root (max lower bound on possible solutions)
beta = minimizer's best option on path to root (min upper bound on possible solutions)
when a new subtree is provably never to be chosen, it is no longer explored
has no effect on minimax value of root, but values of intermediate nodes might be wrong

PROJECT 2

during depth-2 minimax, pacman might not pick up a pellet right next to it
it can pick it up at a later time without penalty
solution: factor the game time into the evaluation function (similar to discount factor)

LECTURE 8

markov decision process (MDP)
set of states S
set of actions A
transition function T(s, a, s')
probability that choosing action a at state s means next state is s'
reward function R(s, a, s')
start state
terminal state
T and R form the "model"
markov = at present, future not dependent on past

plan (for deterministic single-agent search) = sequence of actions from start to goal
policy = one action for each state
optimal policy (π^*) maximizes expected utility
explicit policy means reflex agent

discounting
rewards decay exponentially with time
$V^*(s)$ = optimal value (expected utility) of starting at s
$Q^*(s, a)$ = optimal value (expected utility) of starting at s and taking action a
$\pi^*(s)$ = optimal action from s
$V_k(s)$ = optimal value of s if game ends in k more time steps (same as depth-k expectimax value of s)

value iteration
start with $V_0(s) = 0$ for all s
find $V_{k+1}(s)$ from $V_k(s')$ for all s' reachable from s
maximize over all actions a corresponding to s:
sum over all s': $T(s, a, s') * (R(s, a, s') + (\text{discount factor})(V_k(s')))$
values guaranteed to converge, although often policies converge before values
converges to same V^* values regardless of how V is initialized

Q-value iteration = like value iteration, but with Q-values
start with $Q_0(s, a) = 0$
iterate
$Q_{k+1}(s, a) = \text{sum over s' of } T(s, a, s') * (R(s, a, s') + (\text{discount factor})(\max \text{ over a' of } Q_k(s', a')))$

LECTURE 9

fixed policy = always choose a certain action for each state
$V^\pi(s)$ = utility of state s under fixed policy π
value given by bellman equation (similar to value iteration)

policy iteration (2-step)
evaluation
find $V^\pi(s)$ values based on current policy
repeatedly iterate using value iteration equation for single action
improvement
get better policy $V^{\pi+1}(s)$ using policy extraction (pick action with max utility for each state)

HOMEWORK 4

discount happens after moving
e.g. if starting on an terminal state, you can exit without paying the discount

LECTURE 10

offline planning (e.g. solving MDPs)
determine all quantities through computation, without actually playing the game
must know all details about MDP beforehand

reinforcement learning
agent receives feedback in the form of rewards
must learn to maximize expected rewards
similar to MDP, but don't know model

the transition function T(s, a, s') and reward function R(s, a, s') are unknown
must try actions to learn them

model-based learning (2-step)
learn empirical MDP model
observe episodes (training)
count outcomes s' for each (s, a), then normalize to estimate T(s, a, s')
determine R(s, a, s') when experiencing (s, a, s')
solve the learned MDP
e.g., use value iteration or policy iteration to determine policy
model-free learning
deriving policy without understanding the model (T and R)

passive reinforcement learning
policy evaluation (learn state values for fixed policy)
execute the fixed policy and learn from experience
no choice of actions

direct evaluation
observe episodes (training)
for each episode passing through s, record what the sum of discounted rewards turned out to be
average those samples
easy to understand, model-free, and eventually computes correct average values
takes a long time and wastes information about state connections (each state is independent)

exponential moving average
$x_n = (1-\alpha)x_{n-1} + \alpha(x_n)$
recent samples more important

temporal distance (TD) learning
model-free policy evaluation of fixed policy π, just like direct evaluation
update V(s) every time we experience a transition (s, a, s', r)
sample = r + (discount factor)V(s')
$V(s) = (1-\alpha)V(s) + \alpha(\text{sample}) = V(s) + \alpha(\text{sample} - V(s))$
α is learning rate ($0 < \alpha < 1$)
with TD learning, cannot turn values into a new policy (because T values unknown)

active reinforcement learning
learner makes choices and learns optimal policies/values
tradeoff between exploration and exploitation

Q-learning = sample-based Q-value iteration
update Q(s, a) every time we experience a transition (s, a, s', r)
sample = r + (discount factor)(max over a' of Q(s', a')))
$Q(s, a) = (1-\alpha)Q(s, a) + \alpha(\text{sample})$
is example of off-policy learning
converges to optimal policy, even if acting suboptimally
α must decrease to 0 over time, and each state-action pair must be visited infinitely often
in the limit, it doesn't matter how you choose actions
caveats: must explore enough, can't decrease learning rate too fast

LECTURE 12

approximate Q-learning with features
Q(s, a) = sum over all i of w_i * f_i(s, a)
w_i is a weight (shared across all states)
f_i(s, a) is the i_th feature of (s, a)
for instance, distance to nearest ghost, distance to nearest pellet, etc.
update weights for every transition (s, a, s', r)
difference = (r + (discount factor)(max over a' of Q(s', a'))) - Q(s, a)
$w_i = w_i + \alpha * \text{difference} * f_i$

PROJECT 3

bridge-crossing MDP is hard with noise
due to noise, safer to just cling to starting point reward, even if farther reward is bigger

LECTURE 13

X is independent of Y means
for all x, y, P(x, y) = P(x) * P(y)
for all x, y, P(x|y) = P(x)

X is conditionally independent of Y given Z means
for all x, y, z, P(x, y|z) = P(x|z) * P(y|z)
for all x, y, z, P(x|y, z) = P(x|z)

bayesian network = directed graph where Parents(X) means the direct parents of X
each node denotes a random variable
each node has a conditional probability distribution (CPD) describing P(X|Parents(X))
the CPT for X contains one P(X|...) for each combination of values for Parents(X)
edges encode (local) conditional independence
node X is conditionally independent of its non-descendants, given its immediate parents
sometimes they happen to indicate direct causation/influence
joint distributions are implicitly encoded
P(X, Y, ..., Z) = the product P(X|Parents(X)) * P(Y|Parents(Y)) * ... * P(Z|Parents(Z))

LECTURE 14

bayes net inference = solve some query P(...) using bayes net and given evidence (known variables)

enumeration (naive, exponential-time inference)
write equation for joint probability of all variables, filling in evidence
sum over all possible permutations of the unknown variables of the joint probability

variable elimination (worst-case exponential time but often better)
enumeration with memoization (marginalize each variable immediately after joining over it)
to eliminate variable X
pick all factors with X and cross them off (permanently)
the new factor is the sum over x of the product of the selected factors
the factor name is f_i(all items on left of conditionals | all items on right of conditionals)
items that fall on both sides appear only on the right
the first factor created by this process is called f_1
factor size = total number of variables - number of evidence variables
if query is conditional, break it into two joints and solve top and bottom using variable elimination

polytree
digraph where underlying undirected graph is a tree
has good variable elimination order (reverse topological sort)

MIDTERM REVIEW

admissible heuristic must have h(n) = 0 for goals

A* tree search
has no closed set
optimal when heuristic is admissible
A* graph search
has a closed set (nodes whose true minimum f-values are known)
optimal when heuristic is consistent (which implies admissibility)

UCS edge weight transformation must preserve relative order of any linear combination of weights
e.g. adding 1 to every weight is not okay, but multiplying by a positive constant is

utility of lottery with outcomes x_i and outcome probabilities p_i = sum over i of p_i * utility(x_i)
e.g. if utility(g) = sqrt(g) and lottery is 50-50 between 1 and 4, utility of lottery is 1.5

epsilon-greedy technique for choosing Q-learning action
epsilon chance to choose random action (exploration)
1 - epsilon chance to choose best known action (exploitation)

marginalization
sum over x of P(... x ...|...) removes x completely from the probability
result is 1 if nothing left on left side, e.g., sum of a of P(a|B, C) = 1
cannot directly marginalize out variables on right of conditional, but can often use product rule
e.g., sum of b of P(a|b)P(b) is P(a)

LECTURE 15

inference by variable elimination runs in exponential time
sampling = approximated inference

prior sampling
use the probabilities encoded in the bayes net to sample from joint probability, ignoring evidence
to infer, tally the appropriate counts, then normalize
problem: does not work if there is evidence

rejection sampling
use prior sampling, but ignore (reject) samples where assignments are not consistent with the evidence
problem: if evidence is unlikely, rejects many samples

likelihood weighting
fix evidence variables and sample every other variable X_i from P(X_i|Parents(X_i))
each sample is assigned a weight, which is the product over all evidence variables e_i of P(e_i|Parents(E_i))
problem: evidence influences downstream variables, but not upstream ones

gibbs sampling
fix evidence, initialize other variables randomly, then repeat the following:
choose a non-evidence variable X
resample X from P(X|all other variables)
is an example of a MCMC (markov chain monte carlo) algorithm

LECTURE 16

decision network
bayes net that allows us to find the action corresponding to the MEU
MEU = maximum expected utility, given the evidence
node types:
chance nodes (as in bayes nets)
action nodes (cannot have parents, and act as observed evidence)
utility node (depends on action and chance nodes)
action selection
instantiate all evidence
set action nodes each possible way
calculate posterior for all parents of utility node, given the evidence
calculate expected utility (EU) for each action
choose maximizing action

VPI (value of perfect information) of a node
gain in MEU from knowing the exact value of that node
VPI(E'|e) = MEU(e, E') - MEU(e) where we have evidence E=e but E' is initially unknown

nonnegative, nonadditive, and order-independent

partially observable markov decision process (POMDP)
like an MDP, but with observations O and an observation function P(o|s)

LECTURE 17

markov model
like a MDP with no choice of action
P_t(X) is the distribution describing the probability of being in state x at time t
stationary distribution
P(X) is stationary if P_t(X) = P_{t+1}(X) for t = infinity

hidden markov model (HMM)
observations are generated at each time step based on underlying (hidden) markov chain states
current observation is independent of all else given current state

filtering = tracking the belief state over time
belief state B_t(X) = P(X_t|e_1, e_2, ..., e_t) = P(X_t|e_{1:t})
where e_i is the evidence at time t
example: Kalman filter (used in Apollo program for trajectory estimation)

passage of time
assume we have B(X_t) = P(X_t|e_{1:t})
then P(X_{t+1}|e_{1:t}) = sum over all x_t of P(X_{t+1}|x_t)P(x_t|e_{1:t})
compactly, B'(X_{t+1}) = sum over all x_t of P(X'|x_t)B(x_t)
as time passes, uncertainty "accumulates"

observation
assume we have B'(X_{t+1}) = P(X_{t+1}|e_{1:t}) before new evidence e_{t+1} comes in
then P(X_{t+1}|e_{1:t+1}) is proportional to P(e_{t+1}|X_{t+1})P(X_{t+1}|e_{1:t})
compactly, B(X_{t+1}) is proportional to P(e_{t+1}|X_{t+1})B'(X_{t+1})

forward algorithm
passage of time and observation combined
calculates P(x_t|e_{1:t}) is proportional to the observation * sum over all x_{t-1} of (recursion * transition)
observation = P(e_t|x_t)
recursion = p(x_{t-1}|e_{1:t-1})
transition = p(x_t|x_{t-1})
can normalize over all x_t to get P(X_t|e_{1:t})

LECTURE 19

particle filtering
approximate inference approach for HMMs
store N samples (particles), each of which belongs in one state at a particular time
particles start uniformly distributed, typically, each with weight 1

elapse time
move each particle by sampling the next position from the transition model
x' = sample(P(X'|x))
observe
give each particle at state x a weight w(x) = P(e|x)
resample
sample N times from the weighted sample distribution
this means all particles can have weight 1 again

dynamic bayes net (DBN)
generalization of HMM where variables at time t can condition on those from time t-1
inference procedure: unroll network for T time steps, then eliminate variables to find P(X_t|e_{1:T})

HMM state trellis
shows states and transitions over time (left to right)
each arc is a transition (x_{t-1} -> x_t) with weight P(x_t|x_{t-1})P(e_t|x_t)
each path is a sequence of states

Viterbi algorithm
finds most likely sequence of states that explains the produced output
i.e., it finds the highest probability path through a HMM state trellis
m_t[x_t] = P(e_t|x_t) * max over x_{t-1} of (P(x_t|x_{t-1}) * m_{t-1}[x_{t-1}])

LECTURE 21

perceptron properties
training data must be linearly separable if there exist parameters that produce zero training loss
training will converge (in binary case) if training data is linearly separable
if not separable, can bound error based on how un-separable the training data is

perceptron problems
if training data is not separable, weights may thrash
solution: average weight vectors over time (averaged perceptron)
finds barely separating solution (mediocre generalization)
overfitting (as shown by test loss dropping, then increasing)

MIRA (margin infused relaxed algorithm)
change weight vectors by minimum amount necessary to fix current mistake
helps to generalize better

SVMs (support vector machines)
like MIRA, but optimize for maximum margin over all training data points (examples) at once

FINAL REVIEW

initializing a NN
can't all be zero or else gradients will all be the same
close to zero is better because that's where the activation functions have the highest gradient
solution: draw from distribution closely centered around zero

SGD (stochastic gradient descent)
forward and backpropagate one example at a time
order in which examples are chosen matters
does not guarantee the training loss will decrease (even with a small enough learning rate)
learning rate must be lowered as the objective stops decreasing so that the weights can "settle" into a local optimum
computationally faster than BGD, and can be distributed
mini-batch training
use N examples during each mini-batch (iteration)
number of iterations per epochs = total number of examples / N
BGD (batch gradient descent)
forward all examples and backpropagate the average of the per-training-example gradients
uses the exact gradient of the training loss function in optimization
guarantees that the training loss will decrease (with a small enough learning rate)
good for debugging
momentum can help with convergence for both SGD and BGD
neither SGD nor BGD guarantee decrease in test loss

underfitting
typically means validation error dropping and training error not increasing
can get "free" improvement by increasing model complexity (adding more layers/nodes)
overfitting
typically means validation error increasing
can use dropout to prevent complex co-adaptations and improve generalizability
essentially averages a bunch of NNs
no learning
decrease learning rate (e.g. by a factor of 1/10) until learning happens

feature mean-variance normalization
turns elliptical contours into spherical contours, speeding up learning