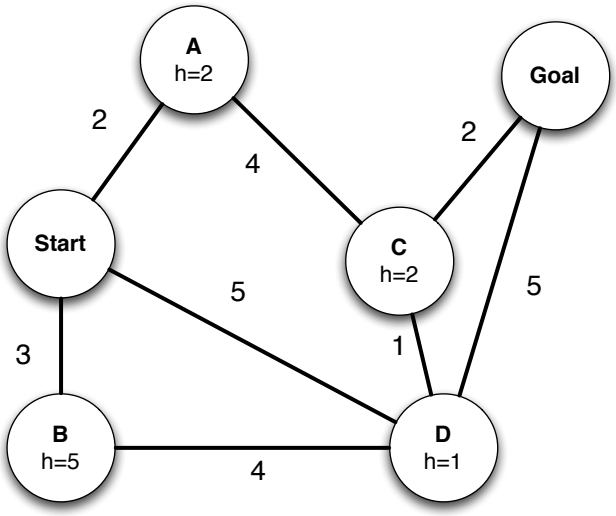
CS188 Spring 2014 Section 0: Search

1 Search algorithms in action



For each of the following graph search strategies, work out the order in which states are expanded, as well as the path returned by graph search. In all cases, assume ties resolve in such a way that states with earlier alphabetical order are expanded first. The start and goal state are S and G, respectively. Remember that in graph search, a state is expanded only once.

a) Depth-first search.
States Expanded: Start, A, C, D, B, Goal
Path Returned: Start-A-C-D-Goal

b) Breadth-first search.
States Expanded: Start, A, B, D, C, Goal
Path Returned: Start-D-Goal

c) Uniform cost search.
States Expanded: Start, A, B, D, C, Goal
Path Returned: Start-A-C-Goal

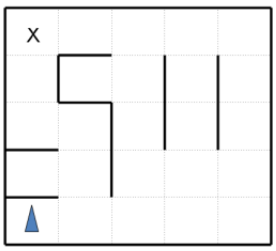
d) Greedy search with the heuristic h shown on the graph.
States Expanded: Start, D, Goal
Path Returned: Start-D-Goal

e) $A^\star$ search with the same heuristic.
States Expanded: Start, A, D,B, C, Goal
Path Returned: Start-A-C-Goal

CS188 Spring 2014 Section 1: Search

1 Search and Heuristics

Imagine a car-like agent wishes to exit a maze like the one shown below:



The agent is directional and at all times faces some direction $d \in (N, S, E, W)$. With a single action, the agent can *either* move forward at an adjustable velocity v *or* turn. The turning actions are *left* and *right*, which change the agent's direction by 90 degrees. Turning is only permitted when the velocity is zero (and leaves it at zero). The moving actions are *fast* and *slow*. *Fast* increments the velocity by 1 and *slow* decrements the velocity by 1; in both cases the agent then moves a number of squares equal to its NEW adjusted velocity. Any action that would result in a collision with a wall crashes the agent and is illegal. Any action that would reduce v below 0 or above a maximum speed V_{max} is also illegal. The agent's goal is to find a plan which parks it (stationary) on the exit square using as few actions (time steps) as possible.

As an example: if the agent shown were initially stationary, it might first turn to the east using (*right*), then move one square east using *fast*, then two more squares east using *fast* again. The agent will of course have to *slow* to turn.

1. If the grid is M by N, what is the size of the state space? Justify your answer. You should assume that all configurations are reachable from the start state.

   The size of the state space is $4MN(V_{max} + 1)$. The state representation is (direction facing, x, y, speed). Note that the speed can take any value in $\{0, ..., V_{max}\}$.

2. What is the maximum branching factor of this problem? You may assume that illegal actions are simply not returned by the successor function. Briefly justify your answer.

   The maximum branching factor is 3, and this happens when the agent is stationary. While stationary it can take the following 3 actions - $fast$, $left$, $right$.

3. Is the Manhattan distance from the agent's location to the exit's location admissible? Why or why not?

   No, Manhattan distance is not an admissible heuristic. The agent can move at an average speed of greater than 1 (by first speeding up to V_{max} and then slowing down to 0 as it reaches the goal), and so can reach the goal in less time steps than there are squares between it and the goal. A specific example: the target is 6 squares away, and the agent's velocity is already 4. By taking only 4 *slow* actions, it reaches the goal with a velocity of 0.

4. State and justify a non-trivial admissible heuristic for this problem which is not the Manhattan distance to the exit.

   There are many answers to this question. Here are a few, in order of weakest to strongest:

   (a) The number of turns required for the agent to face the goal.

   (b) Consider a relaxation of the problem where there are no walls, the agent can turn and change speed arbitrarily. In this relaxed problem, the agent would move with V_{max}, and then suddenly stop at the goal, thus taking $d_{manhattan}/V_{max}$ time.

   (c) We can improve the above relaxation by accounting for the deceleration dynamics. In this case the agent will have to slow down to 0 when it is about to reach the goal. Note that this heuristic will always return a greater value than the previous one, but is still not an overestimate of the true cost to reach the goal. We can say that this heuristic *dominates* the previous one.

5. If we used an inadmissible heuristic in A* tree search, could it change the completeness of the search?

   No! If the heuristic function is bounded, then A* tree search would visit all the nodes eventually, and would find a path to the goal state if there exists one.

6. If we used an inadmissible heuristic in A* tree search, could it change the optimality of the search?

   Yes! It can make the good optimal goal look as though it is very far off, and take you to a suboptimal goal.

7. Give a general advantage that an inadmissible heuristic might have over an admissible one.

   The time to solve an A* tree search problem is a function of two factors: the number of nodes expanded, and the time spent per node.

   An inadmissible heuristic may be faster to compute, leading to a solution that is obtained faster due to less time spent per node. It can also be a closer estimate to the actual cost function (even though at times it will overestimate!), thus expanding less nodes.

   We lose the guarantee of optimality by using an inadmissible heuristic. But sometimes we may be okay with finding a suboptimal solution to a search problem.

2 Expanded Nodes

Consider tree search (i.e. no closed set) on an arbitrary search problem with max branching factor b. Each search node n has a backward (cumulative) cost of $g(n)$, an admissible heuristic of $h(n)$, and a depth of $d(n)$. Let c be a minimum-cost goal node, and let s be a shallowest goal node.

For each of the following, you will give an expression that characterizes the set of nodes that are expanded before the search terminates. For instance, if we asked for the set of nodes with positive heuristic value, you could say $h(n) \geq 0$. Don't worry about ties (so you won't need to worry about $>$ versus $\geq$). If there are no nodes for which the expression is true, you must write "none."

1. Give an expression (i.e. an inequality in terms of the above quantities) for which nodes n will be expanded in a breadth-first tree search.

   $d(n) \leq d(s)$: BFS expands all nodes which are shallower than the shallowest goal. Recall that our search performs the $goal-test$ after popping nodes from the fringe, so we typically expand some nodes at depth s, before we expand the optimal goal node.

2. Give an expression for which nodes n will be expanded in a uniform cost search.

   $g(n) \leq g(c)$: Uniform cost search expands all nodes that are closer than the closest goal node. Recall that our search performs the $goal-test$ after popping nodes from the fringe (this ensures optimality!), so we might expand some nodes of cost $g(c)$, before we expand the optimal goal node.

3. Give an expression for which nodes n will be expanded in an A* tree search with heuristic $h(n)$.

   $g(n) + h(n) \leq g(c)$: All nodes with a total cost of less than g(c), get expanded before the goal node is expanded. This can be proved by induction on the cost $g(n) + h(n)$. Consider a node n_1 which satisfies this property. Note that its parent n_0, will also satisfy this inequality, and by the induction hypothesis, n_0 will be expanded before the goal is expanded, which means that it will put n_1 on the fringe, which will get expanded before the goal node is expanded.

4. Let h_1 and h_2 be two admissible heuristics such that $\forall n, h_1(n) \geq h_2(n)$. Give an expression for the nodes which will be expanded in an A* tree search using h_1 but not when using h_2.

   Let S, be the set of all the nodes. Using the above part, set of nodes expanded by h_1 is $N_1 = \{n : g(n) + h_1(n) \leq g(c)\}$, and, set of nodes expanded by h_2 is $N_2 = \{n : g(n) + h_2(n) \leq g(c)\}$. The set of nodes expanded using h_1 but not using h_2, is $N_1 \cap (S - N_2)$. Since, $h_1 \geq h_2$, $N_1 \subseteq N_2$, hence $N_1 \cap (S - N_2) = \phi$.

5. Give an expression for the nodes which will be expanded in an A* tree search using h_2 but not when using h_1.

   As above, set of nodes expanded using h_2 but not using h_1, is $N_2 \cap (S - N_1) = \{n : g(n) + h_2(n) \leq g(c)$ and $g(c) \leq g(n) + h_1(n)\}$.

CS188 Spring 2014 Section 2: CSPs

1 Course Scheduling

You are in charge of scheduling for computer science classes that meet Mondays, Wednesdays and Fridays. There are 5 classes that meet on these days and 3 professors who will be teaching these classes. You are constrained by the fact that each professor can only teach one class at a time.

The classes are:

1. Class 1 - Intro to Programming: meets from 8:00-9:00am
2. Class 2 - Intro to Artificial Intelligence: meets from 8:30-9:30am
3. Class 3 - Natural Language Processing: meets from 9:00-10:00am
4. Class 4 - Computer Vision: meets from 9:00-10:00am
5. Class 5 - Machine Learning: meets from 10:30-11:30am

The professors are:

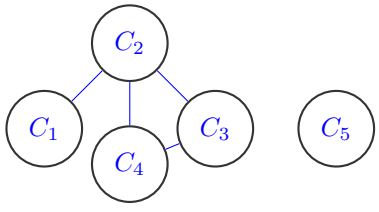
1. Professor A, who is qualified to teach Classes 1, 2, and 5.
2. Professor B, who is qualified to teach Classes 3, 4, and 5.
3. Professor C, who is qualified to teach Classes 1, 3, and 4.

1. Formulate this problem as a CSP problem in which there is one variable per class, stating the domains, and constraints. Constraints should be specified formally and precisely, but may be implicit rather than explicit.

Variables	Domains (or unary constraints)
C_1	{A, C}
C_2	{A}
C_3	{B, C}
C_4	{B, C}
C_5	{A, B}

   Binary Constraints
   $C_1 \neq C_2$
   $C_2 \neq C_3$
   $C_2 \neq C_4$
   $C_3 \neq C_4$

2. Draw the constraint graph associated with your CSP.



3. Your CSP should look nearly tree-structured. Briefly explain (one sentence or less) why we might prefer to solve tree-structured CSPs.

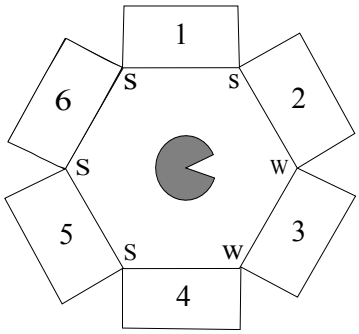
   Minimal answer: we can solve them in polynomial time. If a graph is tree structured (i.e. has no loops), then the CSP can be solved in $O(nd^2)$ time as compared to general CSPs, where worst-case time is $O(d^n)$. For tree-structured CSPs you can choose an ordering such that every node's parent precedes it in the ordering. Then after enforcing arc consistency you can greedily assign the nodes in order, starting from the root, and will find a consistent assignment without backtracking.

2 CSPs: Trapped Pacman

Pacman is trapped! He is surrounded by mysterious corridors, each of which leads to either a pit (P), a ghost (G), or an exit (E). In order to escape, he needs to figure out which corridors, if any, lead to an exit and freedom, rather than the certain doom of a pit or a ghost.

The one sign of what lies behind the corridors is the wind: a pit produces a strong breeze (S) and an exit produces a weak breeze (W), while a ghost doesn't produce any breeze at all. Unfortunately, Pacman cannot measure the strength of the breeze at a specific corridor. Instead, he can stand *between* two adjacent corridors and feel the max of the two breezes. For example, if he stands between a pit and an exit he will sense a strong (S) breeze, while if he stands between an exit and a ghost, he will sense a weak (W) breeze. The measurements for all intersections are shown in the figure below.

Also, while the total number of exits might be zero, one, or more, Pacman knows that two neighboring squares will *not* both be exits.



Pacman models this problem using variables X_i for each corridor i and domains P, G, and E.

1. State the binary and/or unary constraints for this CSP (either implicitly or explicitly).

   Binary:

   $X_1 = P$ or $X_2 = P$, $X_2 = E$ or $X_3 = E$,
   $X_3 = E$ or $X_4 = E$, $X_4 = P$ or $X_5 = P$,
   $X_5 = P$ or $X_6 = P$, $X_1 = P$ or $X_6 = P$,
   $\forall i, j$ s.t. Adj(i, j) $\neg(X_i = E$ and $X_j = E)$

   Unary:

   $X_2 \neq P$,
   $X_3 \neq P$,
   $X_4 \neq P$

2. Cross out the values from the domains of the variables that will be deleted in enforcing arc consistency.

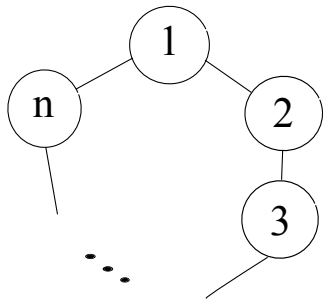
X_1	P		
X_2		G	E
X_3		G	E
X_4		G	E
X_5	P		
X_6	P	G	E

3. According to MRV, which variable or variables could the solver assign first?

   X_1 or X_5 (tie breaking)

4. Assume that Pacman knows that $X_6 = G$. List all the solutions of this CSP or write *none* if no solutions exist.

   (P,E,G,E,P,G)
   (P,G,E,G,P,G)



5. The CSP described above has a circular structure with 6 variables. Now consider a CSP forming a circular structure that has n variables ($n > 2$), as shown below. Also assume that the domain of each variable has cardinality d. Explain precisely how to solve this general class of circle-structured CSPs efficiently (i.e. in time linear in the number of variables), using methods covered in class. Your answer should be at most two sentences.

   We fix X_j for some j and assign it a value from its domain (i.e. use cutset conditioning on one variable). The rest of the CSP now forms a tree structure, which can be efficiently solved without backtracking by enforcing arc consistency. We try all possible values for our selected variable X_j until we find a solution.

6. If standard backtracking search were run on a circle-structured graph, enforcing arc consistency at every step, what, if anything, can be said about the worst-case backtracking behavior (e.g. number of times the search could backtrack)?

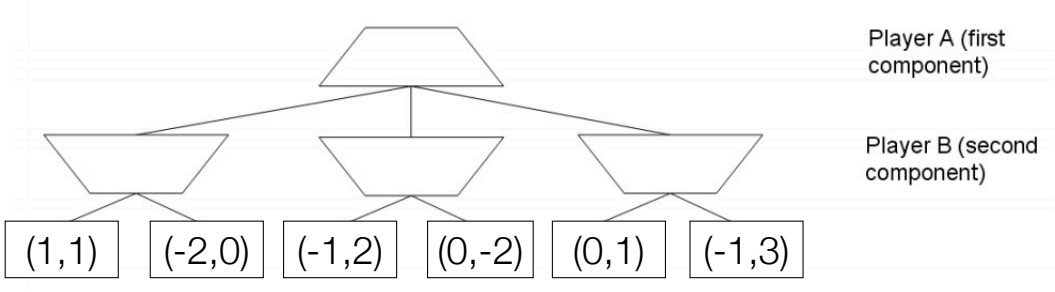
   A tree structured CSP can be solved without any backtracking. Thus, the above circle-structured CSP can be solved after backtracking at most d times, since we might have to try up to d values for X_j before finding a solution.

CS188 Spring 2014 Section 3: Games

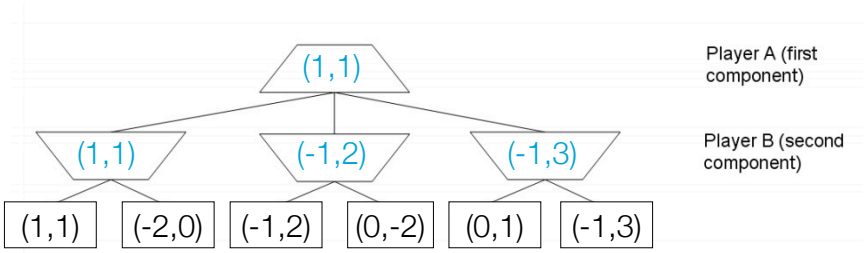
1 Nearly Zero Sum Games

The standard Minimax algorithm calculates worst-case values in a *zero-sum* two player game, i.e. a game in which for all terminal states s, the utilities for players A (MAX) and B (MIN) obey $U_A(s) + U_B(s) = 0$. In the zero sum case, we know that $U_A(s) = -U_B(s)$ and so we can think of player B as simply minimizing $U_A(s)$.

In this problem, you will consider the *non zero-sum* generalization in which the sum of the two players' utilities are not necessarily zero. Because player A's utility no longer determines player B's utility exactly, the leaf utilities are written as pairs $(U_A; U_B)$, with the first and second component indicating the utility of that leaf to A and B respectively. In this generalized setting, A seeks to maximize U_A, the first component, while B seeks to maximize U_B, the second component.



1. Propagate the terminal utility pairs up the tree using the appropriate generalization of the minimax algorithm on this game tree. Fill in the values (as pairs) at each of the internal node. Assume that each player maximizes their own utility.



2. Briefly explain why no alpha-beta style pruning is possible in the general non-zero sum case.
   Hint: think first about the case where $U_A(s) = U_B(s)$ for all nodes.

   The values that the first and second player are trying to maximize are independent, so we no longer have situations where we know that one player will never let the other player down a particular branch of the game tree.

   For instance, in the case where $U_A = U_B$, the problem reduces to searching for the max-valued leaf, which could appear anywhere in the tree.

3. For minimax, we know that the value v computed at the root (say for player A = MAX) is a worst-case value. This means that if the opponent MIN doesn't act optimally, the actual outcome v' for MAX can only be better, never worse than v.

   In the general non-zero sum setup, can we say that the value U_A computed at the root for player A is also a worst-case value in this sense, or can A's outcome be worse than the computed U_A if B plays sub-optimally? Briefly justify.

   A's outcome can be worse than the computed v_A. For instance, in the example game, if B chooses $(-2, 0)$ over $(1, 1)$, then A's outcome will decrease from 1 to 0.

4. Now consider the nearly zero sum case, in which $|U_A(s) + U_B(s)| \leq \epsilon$ at all terminal nodes s for some ϵ which is known in advance. For example, the previous game tree is nearly zero sum for $\epsilon = 2$.

   In the nearly zero sum case, pruning is possible. Draw an X in each node in this game tree which could be pruned with the appropriate generalization of alpha-beta pruning. Assume that the exploration is being done in the standard left to right depth-first order and the value of ϵ is known to be 2. Make sure you make use of ϵ in your reasoning.

   We can prune the node $(0, -2)$ and if we allow pruning on equality then we can also prune $(-1, 3)$. See answers to the next two problems for the reasoning.

5. Give a general condition under which a child n of a B node (MIN node) b can be pruned. Your condition should generalize α-pruning and should be stated in terms of quantities such as the utilities $U_A(s)$ and/or $U_B(s)$ of relevant nodes s in the game tree, the bound ϵ, and so on. Do not worry about ties.

   The pruning condition is $\boxed{U_B > \epsilon - \alpha}$.

   Consider the standard minimax algorithm (zero-sum game) written in this more general 2 agent framework. The maximizer agent tries to maximize its utility, U_A, while the second agent (B) tries to minimize player A's value. This is equivalent to saying that player B wants to maximize $-U_A$. Therefore we say that the utility of player B is $U_B = -U_A$ in the standard minimax situation.

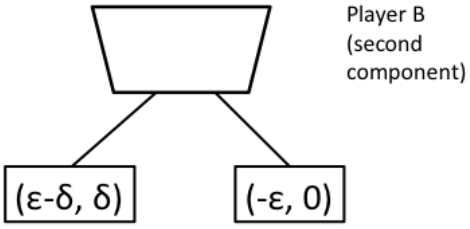
   Recall from lecture that in standard $\alpha - \beta$ pruning we allow a pruning action to occur under a minimizer (player B) node if $v < \alpha$. Under our more general 2 agent framework this condition is equivalent to saying you can prune under player B if $U_A = -U_B < \alpha \Rightarrow U_B > -\alpha$.

   For this question we have an ϵ-sum game so we need to add an additional requirement of ϵ on U_B before pruning can occur. In particular, we know that $|U_A + U_B| \leq \epsilon \Rightarrow U_A \leq \epsilon - U_B$. We want to prune if this upper bound is less than α because then we guarantee that max has an better alternative elsewhere in the tree. Therefore, in order to prune we must satisfy $\epsilon - U_B < \alpha \Rightarrow U_B > \epsilon - \alpha$.

6. In the nearly zero sum case with bound ϵ, what guarantee, if any, can we make for the actual outcome u' for player A (in terms of the value U_A of the root) in the case where player B acts sub-optimally?

$\boxed{u' \geq U_A - 2\epsilon}$

To get intuition about this problem we will first think about the worst case scenario that can occur for player A. Consider the small game tree below for $\delta > 0$:



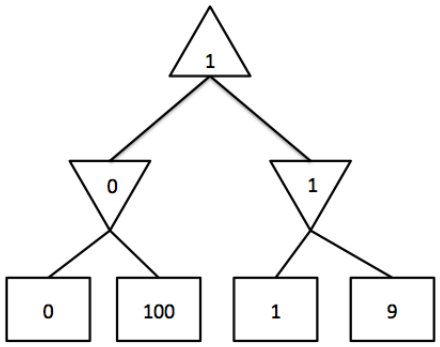
The optimal action for player B to take would be $(\epsilon - \delta, \delta)$. If player B acts optimally then player A will end up with a value of $U_A = \epsilon - \delta$. Now, consider what would happen if player B acted suboptimally, namely if player B chose $(-\epsilon, 0)$. Then player A would receive an actual outcome of $u' = -\epsilon$. So, we see that $u' \geq U_A - 2\epsilon + \delta$. Now let δ be arbitrarily small and you converge to the bound boxed above.

Thus far we have just shown (by example) that we cannot hope for a better guarantee than $u' \geq U_A - 2\epsilon$ (if someone claimed a better guarantee, the above would be a counter-example to that (faulty) claim). We are left with showing that this bound actually holds true. To do so, consider what happens when Player B plays suboptimally. By definition of suboptimality, that means the outcome of the game for player B is not the optimal U_B but some lower value $U'_B = U_B x$ with $x > 0$. This will have the maximum effect on player A's pay-off when for the optimal outcome we had $U_A + U_B = \epsilon$, but for the suboptimal outcome we have $U'_A + U'_B = U'_A + U_B x = \epsilon$. From the first equation we have $U_B = \epsilon U_A$, substituting into the second equation gives us: $U'_A = \epsilon\epsilon + U_A = U_A 2\epsilon$ as the worst-case outcome for player A.

2 Minimax and Expectimax

In this problem, you will investigate the relationship between expectimax trees and minimax trees for zero-sum two player games. Imagine you have a game which alternates between player 1 (max) and player 2. The game begins in state s_0, with player 1 to move. Player 1 can either choose a move using minimax search, or expectimax search, where player 2's nodes are chance rather than min nodes.

1. Draw a (small) game tree in which the root node has a larger value if expectimax search is used than if minimax is used, or argue why it is not possible.



We can see here that the above game tree has a root value of 1 for the minimax strategy. If we instead switch to expectimax and replace the min nodes with chance nodes, the root of the tree takes on a value of 50 and the optimal action changes for MAX.

2. Draw a (small) game tree in which the root node has a larger value if minimax search is used than if expectimax is used, or argue why it is not possible.

Optimal play for MIN, by definition, means the best moves for MIN to obtain the lowest value possible. Random play includes moves that are not optimal. Assuming there are no ties (no two leaves have the same value), expectimax will always average in suboptimal moves. Averaging a suboptimal move (for MIN) against an optimal move (for MIN) will always increase the expected outcome.

With this in mind, we can see how there is no game tree where the value of the root for expectimax is lower than the value of the root for minimax. One is optimal play – the other is suboptimal play averaged with optimal play, which by definiton leads to a higher value for MIN.

3. Under what assumptions about player 2 should player 1 use minimax search rather than expectimax search to select a move?

   Player 1 should use minimax search if he/she expects player 2 to move optimally.

4. Under what assumptions about player 2 should player 1 use expectimax search rather than minimax search?

   If player 1 expects player 2 to move randomly, he/she should use expectimax search. This will optimize for the maximum expected value.

5. Imagine that player 1 wishes to act optimally (rationally), and player 1 knows that player 2 also intends to act optimally. However, player 1 also knows that player 2 (mistakenly) believes that player 1 is moving uniformly at random rather than optimally. Explain how player 1 should use this knowledge to select a move. Your answer should be a precise algorithm involving a game tree search, and should include a sketch of an appropriate game tree with player 1's move at the root. Be clear what type of nodes are at each ply and whose turn each ply represents.

   Use two games trees:

   Game tree 1: max is replaced by a chance node. Solve this tree to find the policy of MIN.

   Game tree 2: the original tree, but MIN doesn't have any choices now, instead is constrained to follow the policy found from Game Tree 1.

CS188 Spring 2014 Section 4: MDPs

1 MDPs: Micro-Blackjack

In micro-blackjack, you repeatedly draw a card (with replacement) that is equally likely to be a 2, 3, or 4. You can either Draw or Stop if the total score of the cards you have drawn is less than 6. Otherwise, you must Stop. When you Stop, your utility is equal to your total score (up to 5), or zero if you get a total of 6 or higher. When you Draw, you receive no utility. There is no discount ($\gamma = 1$).

1. What are the states and the actions for this MDP?

   The state is the current sum of your cards, plus a terminal state:

   $$0, 2, 3, 4, 5, Done$$

   (answers which include 1,6,7,8,9, a *Bust* state, or just "the current sum of your cards plus a terminal state" are acceptable.)

   The actions are $\{Draw, Stop\}$.

2. What is the transition function and the reward function for this MDP?

   The transition function is

   $$T(s, Stop, Done) = 1$$

   $$T(s, Draw, s') = \begin{cases} 1/3 \text{ if } s' - s \in \{2, 3, 4\} \\ 1/3 \text{ if } s = 2 \text{ and } s' = Done \\ 2/3 \text{ if } s = 3 \text{ and } s' = Done \\ 1 \text{ if } s \in \{4, 5\} \text{ and } s' = Done \\ 0 \text{ otherwise} \end{cases}$$

   The reward function is

   $$R(s, Stop, s') = s, s \leq 5$$
   $$R(s, a, s') = 0 \text{ otherwise}$$

3. Give the optimal policy for this MDP.

   In general, for finding the optimal policy for an MDP, we would use some method like value iteration followed by policy extraction. However, in this particular case, it is simple to work out that the opimal policy would be **Draw if $s \leq 2$, Stop otherwise.**

   For completeness, we give below the value iteration steps based on the states and transition functions described above. The optimal policy is given by taking the *argmax* instead of *max*, in the final iteration of value iteration.

V	0	2	3	4	5	Done
V_0	0	0	0	0	0	0
V_1	0	2	3	4	5	0
V_2	3	3	3	4	5	0
V_3	10/3	3	3	4	5	0
Policy Extraction	$10/3_{Draw}$	3_{Draw}	3_{Stop}	4_{Stop}	5_{Stop}	0_{Stop}

4. What is the smallest number of rounds (k) of value iteration for which this MDP will have its exact values (if value iteration will never converge exactly, state so).

   3

2 Pursuit Evasion

Pacman is trapped in the following 2 by 2 maze with a hungry ghost (the horror)! When it is his turn to move, Pacman must move one step horizontally or vertically to a neighboring square. When it is the ghost's turn, he must also move one step horizontally or vertically. The ghost and Pacman alternate moves. After every move (by either the ghost or Pacman) if Pacman and the ghost occupy the same square, Pacman is eaten and receives utility -100. Otherwise, he receives a utility of 1. The ghost attempts to minimize the utility that Pacman receives. **Assume the ghost makes the first move**.

For example, with a discount factor of $\gamma = 1.0$, if the ghost moves down, then Pacman moves left, Pacman earns a reward of 1 after the ghost's move and -100 after his move for a total utility of -99.

Note that this game is not guaranteed to terminate.

1. Assume a discount factor $\gamma = 0.5$, where the discount factor is applied once every time either Pacman or the ghost moves. What is the minimax value of the truncated game after 2 ghost moves and 2 Pacman moves? (Hint: you should not need to build the minimax tree)

   $$1 + 0.5 + 0.25 + 0.125 = 1.875$$

2. Assume a discount factor $\gamma = 0.5$. What is the minimax value of the complete (infinite) game? (Hint: you should not need to build the minimax tree)

   2

3. Why is value iteration superior to minimax for solving this game? Value iteration takes advantage of repeated states to efficiently solve for the optimal policy. Even a truncated minimax tree increases in size exponentially as you increase the search depth.

4. This game is similar to an MDP because rewards are earned at every timestep. However, it is also an adversarial game involving decisions by two agents.

   Let s be the state (e.g. the position of Pacman and the ghost), and let $A_P(s)$ be the space of actions available to Pacman in state s (and similarly let $A_G(s)$ be the space of actions available to the ghost). Let $N(s, a) = s'$ denote the successor function (given a starting state s, this function returns the state s' which results after taking action a). Finally, let $R(s)$ denote the utility received after moving to state s.

   Write down an expression for $P^*(s)$, the value of the game to Pacman as a function of the current state s (analogous to the Bellman equations). Use a discount factor of $\gamma = 1.0$. Hint: your answer should include $P^*(s)$ on the right hand side.

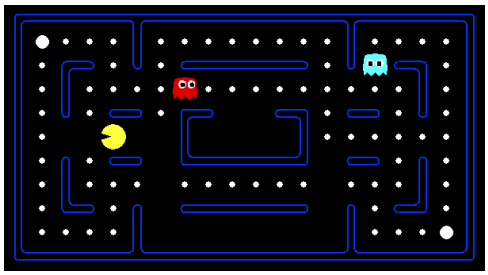
   $P^*(s) =$

   $$P^*(s) = \max_{a \in A_P(s)} R(N(s,a)) + \min_{a' \in A_G(N(s,a))} R(N(N(s,a),a')) + P^*(N(N(s,a),a'))$$

CS188 Spring 2014 Section 5: Reinforcement Learning

1 Learning with Feature-based Representations

We would like to use a Q-learning agent for Pacman, but the state size for a large grid is too massive to hold in memory (just like at the end of Project 3). To solve this, we will switch to feature-based representation of Pacman's state. Here's a Pacman board to refresh your memory:



1. What features would you extract from a Pacman board to judge the expected outcome of the game?
   The usual ones, for example as in Project 2.

2. Say our two minimal features are the number of ghosts within 1 step of Pacman (F_g) and the number of food pellets within 1 step of Pacman (F_p). For this pacman board:

   Extract the two features (calculate their values).
   $f_g = 2, f_p = 1$

3. With Q Learning, we train off of a few episodes, so our weights begin to take on values. Right now $w_g = 100$ and $w_p = -10$. Calculate the Q value for the state above.
   First of all, the Q value will not depend on what action is taken, because the features we extract do not depend on the action, only the state.

$$Q(s,a) = w_g * f_g + w_p * f_p = 100 * 2 + -10 * 1 = 190$$

4. We receive an episode, so now we need to update our values. An episode consists of a start state s, an action a, an end state s', and a reward $R(s, a, s')$. The start state of the episode is the state above (where you already calculated the feature values and the expected Q value). The next state has feature values $F_g = 0$ and $F_p = 2$ and the reward is 50. Assuming a discount of 0.5, calculate the new estimate of the Q value for s based on this episode.

$$\begin{aligned} Q_{new}(s, a) &= R(s, a, s') + \gamma * \max_{a'} Q(s', a') \\ &= 50 + 0.5 * (100 * 0 + -10 * 2) \\ &= 40 \end{aligned}$$

5. With this new estimate and a learning rate (α) of 0.5, update the weights for each feature.

$$w_g = w_g + \alpha * (Q_{new}(s, a) - Q(s, a)) * f_g(s, a) = 100 + 0.5 * (40 - 190) * 2 = -50$$
$$w_p = w_p + \alpha * (Q_{new}(s, a) - Q(s, a)) * f_p(s, a) = -10 + 0.5 * (40 - 190) * 1 = -85$$

   Note that now the weight on ghosts is negative, which makes sense (ghosts should indeed be avoided). Although the weight on food pellets is now also negative, the difference between the two weights is now much lower.

6. Good job on updating the weights. Now let's think about this entire process one step back. What values do we learn in this process (assuming features are defined)? When we have completed learning, how do we tell if Pacman does a good job?
   The values we learn are the feature weights. Once Pacman completes its learning, we will evaluate its performance by running the game and looking at the win/lose outcomes and the reward accrued from the start state.

7. In some sense, we can think about this entire process, on a meta level, as an input we control that produces an output that we would like to maximize. If you have a magical function ($F(input)$) that maps an input to an output you would like to maximize, what techniques (from math, CS, etc) can we use to search for the best inputs? Keep in mind that the magical function is a black box.
   Of course, you can use random search (changing values indiscriminately), or try some evenly distributed values in the possible range (which is sometimes called beam search).

   Also, remember that earlier in the course we talked about local search, which includes techniques such as hill climbing (going in the direction of maximum positive change), simulated annealing (where the rate of random exploration is lowered as time goes on), and genetic algorithms. These are all possible solutions.

8. Now say we can calculate the derivative of the magical function, $F'(input)$, giving us a gradient or slope. What techniques can we use now?
   Gradient descent, or many techniques from calculus and optimization developed for such problems. Although these techniques will be very useful to you as an artificial intelligence researcher, don't worry: we will not expect you to know how to use any of them on the exams!

2 Odds and Ends

1. When using features to represent the Q-function is it guaranteed that the feature-based Q-learning finds the same optimal $Q*$ as would be found when using a tabular representation for the Q-function?
   No, if the optimal Q-function $Q*$ cannot be represented as a weighted combination of features, then the feature-based representation would not have the expressive power to find it.

2. Why is temporal difference (TD) learning of Q-values (Q-learning) superior to TD learning of values?
   Because if you use temporal difference learning on the values, it is hard to extract a policy from the learned values. Specifically, you would need to know the transition model T. For TD learning of Q-values, the policy can be extracted directly by taking $\pi(s) = \arg\max_a Q(s, a)$.

3. Can all MDPs be solved using expectimax search? Justify your answer.
   No, MDPs with self loops lead to infinite expectimax trees. Unlike search problems, this issue cannot be addressed with a graph-search variant.

4. When learning with ϵ-greedy action selection, is it a good idea to decrease ϵ to 0 with time? Why or why not?
   Yes, especially when using on-policy learning methods. The reason is that as the agent learns the actual optimal policy for the world, it should switch from a mix of exploration and exploitation to mostly exploitation (unless the world is changing, in which case it should always keep exploring).