Derek Reimanis

Collaborator: Rachael Luhr

AI Homework 2

1.)

Show how a single ternary constraint can be turned into three binary constraints with the use of an auxiliary variable.

Idea: map domains to domains, and treat as binary processes

To make this constraint into a binary constraint, this (single) auxiliary variable must encompass all possible relations for the ternary constraint.

We will represent the auxiliary variable as D.

We start by showing that:

D = (A + B = C), or domain(D) = (domain(A) + domain(B) = domain(C))

-This means that at any point in time, the domain of C is gathered from the domains of A and B, which are all represented in D.

Now, because we have finite domains, assume the following:

Domain(A) = {0, 1}

Domain(B) = {0, 1}

Domain(C) = {0, 1, 2} //remember, A + B = C. The method is shown immediately below.

Now, we have 4 different possible legal domains for D:

1. When domain(A) = 0, domain(B) = 0, so domain(C) = 0
2. When domain(A) = 0, domain(B) = 1, so domain(C) = 1
3. When domain(A) = 1, domain(B) = 0, so domain(C) = 1
4. When domain(A) = 1, domain(B) = 1, so domain(C)= 2

There are no more possible domains for D, because C is dependent on A and B. For example, the following cannot be true:

Domain(A) = 0, domain(B) = 0, domain(C) = 1

We can now show the domain(D) as the following:

Domain(D) = {1, 2, 3, 4} //each number corresponds to the numbered possibilities above.

Now, we can create three binary constraints by mapping each A, B, and C to D, where:

{(A, D)} = {(0, 1), (0, 2), (1, 3), (1, 4)}

{(B, D)} = {(0, 1), (0, 3), (1, 2), (1, 4)}

{(C, D)} = {(0, 1), (1, 2), (1, 3), (2, 4)}

These are three binary constraints, all of which include the domain from the original ternary constraint. We have included a constraint for the D variable, being a combined legal domain set.

Show how constraints with more than three variables may be treated similarly:

Idea: Use induction with the steps shown above.

Assumption: We can transform a ternary (or larger) constraint into three binary constraints by using an auxiliary variable.

Hypothesis: If we assume we can always represent domains inside other domains, then certainly any constraint size can be represented as three binary constraints.

Proof:

If we have the quaternary constraint A + B = C + D, where each variable has a unique domain, we can represent their cumulative domains as an auxiliary variable (Very similar to part 1).

Following this, we arrive with:

Domain(D) + domain(C) = domain(A) + domain(B)

Certainly, we can illustrate E as the unioned domains of D and C:

E = (Domain(D) ∪ Domain(C))

Then, we have:

Domain(A) + domain(B) = E

Now, we have the same ternary relation as is present in part 1. We can just run the same methods to achieve three binary constraints. Note that the variable E represents a unioned set of domains. This is the key point. Given any higher order constraint, we can always union variable domains together to get down to a ternary constraint.

QED.

Basically, this process involves combing domains until three variables (auxiliary or otherwise) are reached. From a ternary constraint, we can always turn it into a binary constraint.

Eliminating unary constraints through variable domain alteration:

Idea: A unary variable is constrained if its domain does exist in a listing of legal domains for that variable, but pre-condition show that that domain isn't valid.

For example:

Given unary constraint A with domain f and domain set D, such that f ∈ D, A is a constraining unary variable if conditions show that f cannot be the domain of A.

If we altered f, removing it from D, and chose a new domain variable d such that d ∈ D, the domain of A becomes valid.

Thus, we can eliminate the unary constraint A, because the domain is now valid.

2.)

Backtracking with conflict directed backjumping.

Variable order: {A1, H, A4, F1, A2, F2, A3, T}

Color Order: {r, g, b}

We take in order, so:

A1 := r

H := r (conflict)

H := g

A4 := r

F1 := r

A2 := r (conflict)

A2 := g (conflict)

A2 := b

F2 := r

A3 := r (conflict)

A3 := g (conflict)

A3 := b (conflict)

A3 cannot be assigned, so we need to go back to where the conflict started (at A4 with the r assignment)

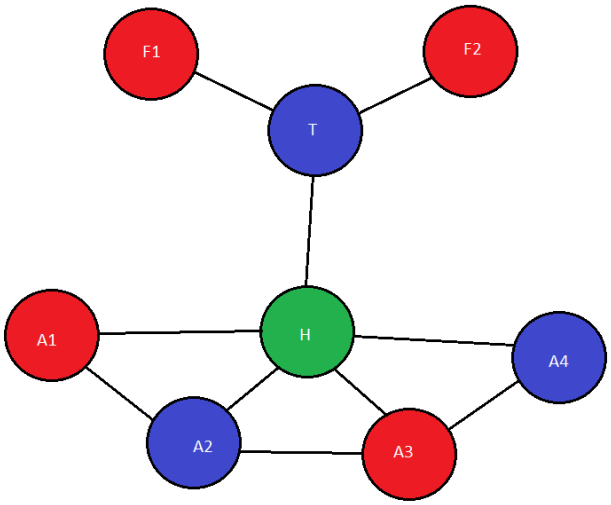
A4 := b

A3 := r (works now!)

T := r (conflict)

T := b

Graph complete!



Note the smiley face. :-D

3.)

Idea: It is difficult to distinguish an all-seeing oracle from a game where any number of possible moves may occur. Just because the oracle can predict any opponent move from any given state does not mean you have an advantage. One can still lose if the opponent is playing their best. (Tick-tak-toe with you being circles will at MOST cat the game when the opponent is playing their best game).

The game doesn't need to map out all of the possible moves. This allows for a larger search tree for finding moves.

The search problem would entail constructing a search tree, and, given your best move, calculating your opponents move. If the opponent's move is worth more than yours long term, then back up to a point where either of your moves was worth more that the opponent's.

This is very akin to a backtracking with conflict-directed backjumping method.

I would define the optimal move as one in which the sum of the move you take and the opponents following move gives you the largest heuristic value, calculated from the search tree. Of course, your knowledge of your opponents move will influence your move. One could almost define the entirety of game states from the initial position, creating a one-child, one-parent tree, but the question asks for an optimal move, not an optimal game. Unless an optimal game is defined as an optimal set of moves. But that may or may not be true, depending on heuristic value of that move at that point in the game.

So, because I can't envision why an optimal game would not be desired, I will define a game as the optimal set of moves, of which at any point in time corresponds to the best outcome for the game.

It may be worth noting that an optimal game is a stronger relation than an optimal move.

Algorithm:

Build a search tree with each node corresponding to a state and each edge corresponding to a move.

Once the search tree has reached a depth limit (provided by heuristics), use a heuristic on the resulting leaves.

For the leaf with the highest outcome to you, work back to the root.

For each node that corresponds to an opponent's state, feed that state into the oracle to see what the opponent's move will be.

If the opponent's move deviates (removing any chance of reaching that leaf's high outcome), select the next highest leaf and repeat the above step.

This way, the optimal moves for the optimal game favoring you will be selected.

4.)

Idea: show that if p2 plays sub optimally, p1 will always win with a higher utility.

Inductive proof:

Define the values:

n := root of Minimax tree from current play

c := child of node n

$v_{optimal}(n)$:= values of tree from root when playing optimally

$v_{sub\text{-}optimal}(n)$:= values of tree from root when opponent is playing sub-optimally

Assume that:

$v_{optimal}(n) <= v_{sub\text{-}optimal}(n)$ for all n in tree

Base case:

Certainly, if the root is the tree (1 node), then:

$v_{optimal}(n) <= v_{sub\text{-}optimal}(n)$

Inductive Steps:

As we traverse through the tree, n will alternate from min and max nodes.

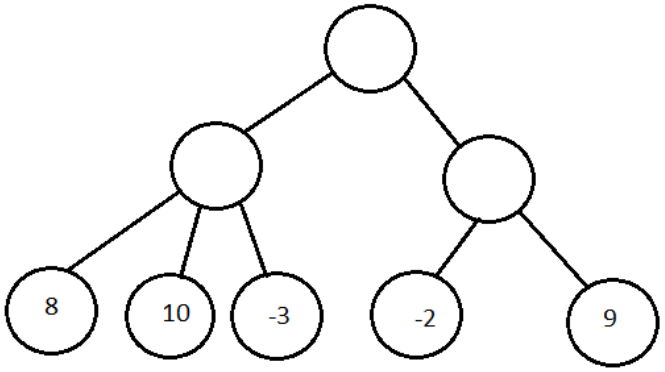
In the case where n is a max node, obviously $v_{optimal}(c) <= v_{sub\text{-}optimal}(c)$ will hold true.

This simply means that the optimal solution of your opponent is worth less to you than the sub-optimal solution. Certainly, the opponent cannot choose anything greater to them than the optimal solution.

In the case where n is a min node, we can either have $v_{optimal}(c) <= v_{sub\text{-}optimal}(c)$, in which c is the optimal choice of the parent node, or $v_{optimal}(d) <= v_{sub\text{-}optimal}(d)$ in which d is the sub-optimal choice of the parent node. In either case, $v_{optimal}(c) <= v_{sub\text{-}optimal}(c) <= v_{sub\text{-}optimal}(d)$ holds true from the assumption.

In all cases (max and min), $v_{optimal}(n) <= v_{sub\text{-}optimal}(n)$ holds true. Thus, the utility of a sub-optimal solution of a minimax decision will always be greater than the utility of an optimal one.

Here is a game tree that shows a sub-optimal max strategy against a sub-optimal min strategy:



Notice that if the min stage and max stage were both playing optimally, the next move would be a 9. However, if the min played sub-optimally, and presented 8 and 9, and the max played sub-optimally and chose 8, a 10 could be reached.

The min could also play 9 and 10, in which case the 10 could still be reached.

5.)

Here is the payoff matrix. Note that this is a zero-sum game, because either player either wins or loses.

Wins are represented as a 1, losses as a -1, and draws as a 0.

To represent the matrix from the other player's perspective, one only needs to invert the values. This includes the combined values.

The combined column shows the probability of winning with a corresponding row, with the following variable assignments:

r := rock

p := paper

s := scissors

f := fire

w := water = 1-(p+r+s+f)

	Rock	Paper	Scissors	Fire	Water	Combined
Rock	0	-1	1	-1	1	-2p -2f -r +1
Paper	1	0	-1	-1	1	-2s -p -2f +1
Scissors	-1	1	0	-1	1	-2r -s -2f +1
Fire	1	1	1	0	-1	2r +2p +2s +f -1
Water	-1	-1	-1	1	0	-r -p -s +f

Note that these combined values are simplified.

After developing this table, we can solve the system of linear equations to find the values of each variable.

I didn't do the hard-wrought math to solve this, I used a calculator. If you need me to show the math, I will certainly be willing and able to.

So, building a matrix from that we can build the equation:

inverse function

$$\begin{pmatrix} 1 & 2 & 0 & 2 \\ 0 & 1 & 2 & 2 \\ 2 & 0 & 1 & 2 \\ 2 & 2 & 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} r \\ p \\ s \\ f \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Solving, we find that:

r = 1/9 = 11.1%
p = 1/9 = 11.1%
s = 1/9 = 11.1%
f = 1/3 = 33.3%
w = 1 - (1/9 + 1/9 + 1/9 + 1/3) = 1/3 = 33.3%

This solution to the game is correct. Notice that each rock, paper, and scissors has an equal chance of winning. That is because each can beat two other plays, and each can lose to two other plays. Now, an interesting thing occurs when looking at fire and water. While fire can beat three other plays and only lose to one, water can beat only one other play and lose to three. Now, the high percentage of wins related to choosing fire directly correspond to the likelihood that fire will win more often than lose. However, because of this high likelihood, any moves that can best fire must hold great value. This explains why water, although has the most plays that any play can lose to, still has a high likelihood of winning.