1 Plan Graph Interpretation

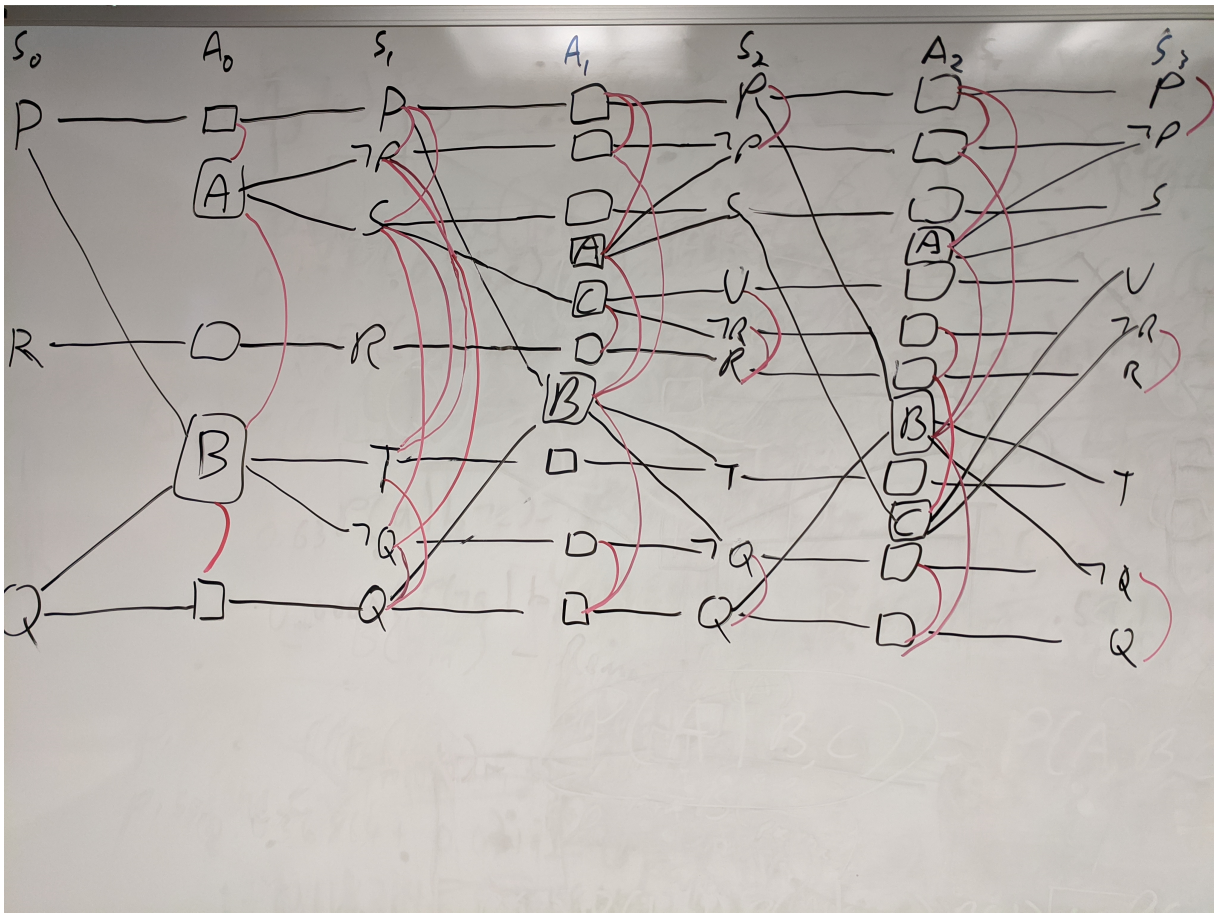

Figure 1: The Planning Graph from class.

What we discussed here was how to interpret a few bits of a Planning Graph. A planning graph is essentially a depiction of possible worlds along a timeline starting from some initial state. At level S_0, we have only that which is true in our initial state. At level A_0, we have only the actions possible given this initial state. Level S_1 contains anything that *could* be true, given that the agent could've performed any of the available actions from the initial state. A_1 gives all actions possible from any of the possibilities encoded in S_1. The graph continues in this way, alternating between A levels and S levels for some number of levels. A link from some literal l in an S level to some action a in the subsequent A level indicates that l is a precondition to performing a. Likewise, a link going from some action a to some literal l in the succeeding level indicates that l is an effect of a. The red lines between nodes on the same level are *mutual exclusion* links. A mutual exclusion link joining two literals essentially states that those two literals cannot both be true at that point in time. A mutual exclusion link between two actions could mean a few different things - one of the actions could negate an effect of the other, a precondition of one could be the opposite of a precondition of the other, or one action could negate a precondition of the other.

One piece of information you may want out of such a graph is to determine a lower bound on the number of moves that must be made to acheive some goal. For example, in the above graph, we can see that the lower bound for the goal $S \wedge T$ is 2. Even though both S and T show up in level S_1 of the graph, there is a red mutual exclusion link between them - illustrating that they cannot possibly both be true after just one action. Level S_2 is the first level in which they appear without such a link between them.

The other thing we examined was how to extract a plan from such a graph. If our goal state was $S \wedge T \wedge U$, we could see from looking at the graph that the earliest we could possibly acheive this goal was in two moves. However, if we look more closely, we'll see that we actually require three moves to acheive this goal. We can see that the only moves (besides the unlabeled *persistence* actions which just say "doing nothing to this variable doesn't change its value") which can make S, T, and U true are A, B, and C, respectively. As each of A, B, and C only make one of our goal predicates true, we know we need to perform all three of them. Further, by observing the links between each node labeled A and a corresponding node labeled $\neg P$, we can see that action A causes P to become false. Likewise, the links between each node labeled P and a corresponding B tell us that

P is a precondition of B. As such, we must perform action B before we can perform action A. Finally, since C has S as a precondition, we must perform A (which is the only way to make S become true) before we can perform C. Thus, the plan we can extract from this graph is to first perform B, then A, then finally C.

2 Markov Decision Processes

For this part of the lecture, we focused on the simple Wumpus World depcited in Figure 2:

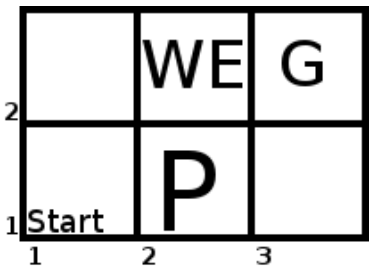

Figure 2: Simple Wumps World

We stipulated that the hero, Lady Eve, was standing in room $< 2, 2 >$ next to the dead Wumpus. In room $< 2, 3 >$ sits the gold. We further stated that a state in this simple version of the Wumpus World can be represented by a vector $(location, haveGold)$, where location is one of $< 1, 1 >, < 1, 2 >, < 1, 3 >, < 2, 1 >, < 2, 2 >, < 2, 3 >$ and $haveGold$ is either $true$ or $false$. The reward for escaping (that is, getting back to $< 1, 1 >$) with the gold is 1000. For falling in the pit (that is, going to location $< 1, 2 >$ the reward is -1000. For every other action (inclucing escaping *without* the gold), the reward is -1.

There are few things we did here. The first was define the transition function as a table. Given that Eve has four actions available (we're forgetting about arrows, and just taking it for granted she'll pick up the gold when she enters the room containing it): $Up, Down, Left, Right$, and there are 8 non-terminal states (getting to $< 1, 1 >$ with or without the gold, or getting to $< 1, 2 >$ with or without the gold are terminal states), we've got a 8×5 table:

s	Up	$Down$	$Left$	$Right$
$(< 1, 3 >, false)$	$(< 2, 3 >, true)$	-	$(< 1, 2 >, false)$	-
$(< 2, 1 >, false)$	-	$(< 1, 1 >, true)$	-	$(< 2, 2 >, false)$
$(< 2, 2 >, false)$	-	$(< 1, 2 >, false)$	$(< 2, 1 >, false)$	$(< 2, 3 >, true)$
$(< 2, 3 >, false)$	-	$(< 1, 3 >, false)$	$(< 2, 2 >, false)$	-
$(< 1, 3 >, true)$	$(< 2, 3 >, true)$	-	$(< 1, 2 >, true)$	-
$(< 2, 1 >, true)$	-	$(< 1, 1 >, true)$	-	$(< 2, 2 >, true)$
$(< 2, 2 >, true)$	-	$(< 1, 2 >, true)$	$(< 2, 1 >, true)$	$(< 2, 3 >, true)$
$(< 2, 3 >, true)$	-	$(< 1, 3 >, true)$	$(< 2, 2 >, true)$	-

Note that I used dashes (-) anywhere the action would cause Eve to remain in the same state to save on "writing". Also note that since $(< 2, 3 >, false)$ isn't possible due to the way we've defined the game, it doesn't matter whether *e.g.* moving Up in that state causes her to pick up the gold or not.

Then we similarly defined the Reward table as a function. This time we need a 12×2 table:

s	$R(s)$
$(< 1, 1 >, false)$	-1
$(< 1, 2 >, false)$	-1000
$(< 1, 3 >, false)$	-1
$(< 2, 1 >, false)$	-1
$(< 2, 2 >, false)$	-1
$(< 2, 3 >, false)$	-1
$(< 1, 1 >, true)$	1000
$(< 1, 2 >, true)$	-1000
$(< 1, 3 >, true)$	-1
$(< 2, 1 >, true)$	-1
$(< 2, 2 >, true)$	-1
$(< 2, 3 >, true)$	-1

Pretty much as defined in the English description of how the game works.

After that, we talked about value iteration. For this, we need to recall the *Bellman Equation*

$$\vec{v}_{i+1}(s) = Reward(s) + \beta \max_{a \in A(s)} \vec{v}_i(Result(s,a)) \quad (1)$$

Starting from a vector $v_0 = (-1, -1000, -1, -1, -1, -1, 1000, -1000, -1, -1, -1, -1)$ where we order the states first by $haveGold$ ($falses$ first then $trues$), and then by location in the order $< 1,1 >, < 1,2 >, < 1,3 >, < 2,1 >, < 2,2 >, < 2,3 >$, and with a discount factor of $\beta = 0.99$, we iterate on this until we reach some step k such that $\vec{v}_k = \vec{v}_{k-1}$. Of course, since this could take a lot of steps, we'll only really iterate over the first few steps. Also, since 12 is a lot of states to keep track of, I'll just keep track of the following 4, in this order: $(< 2,2 >, false), (< 2,1 >, true), (< 2,2 >, true)$ and $(< 2,3 >, true)$

After the first iteration, we end up with $(-1.99, 989, -1.99, -1.99)$. Note that $(< 2,1 >, true)$ jumps to 989 because it's next to $(< 1,1 >, true)$ which has value 1000 (and is better than the value of its other neighbor, $(< 2,2 >, true)$, which is -1). The values of all the neighbors of all the other states we're tracking is -1, so we end up with a value of 1.99 for them.

After the second iteration, we end up with $(-2.9701, 989, 978.11, -2.9701)$. Now, the big payoff from $(< 1,1 >, true)$ has propagated back to $(< 2,2 >, true)$ via $(< 2,1 > true)$. The payoffs of all the other states keep getting smaller, because remember - we're working off the vector we obtained on the previous iteration, where they all got a value of -1.99 (so all neighbors of all states which aren't $(< 2,2 >, true)$ or $(< 2,1 > true)$ have value 1.99 to multiply by 0.99 before being added to -1).

After the third iteration, we'll have $(-3.940399, 989, 978.11, 968.3289)$. At this point, we should see how eventually the large payoff will back up to all states from which the goal of $(< 1,1 >, true)$ is reachable.

Finally, I posed the question "what is the optimal policy for this situation". It turns out that you can pretty easily figure determine that it's $Right, Left, Left, Down$ just by looking at Figure 2.

3 Decision Tree Learning

First, some function definitions:

- $B(q) = -(q\ log_2(q) + (1-q) log_2(1-q))$
- $Remainder(A) = \sum_{k=1}^{d} \frac{p_k + n_k}{p+n} B(\frac{p_k}{p_k + n_k})$
- $Gain(A) = B(\frac{p}{p+n}) - Remainder(A)$

We'll use these to learn a decision tree using the training data detailed in the following table:

	A_1	A_2	A_3	Output
x_1	1	1	0	0
x_2	1	0	1	1
x_3	1	1	1	1
x_4	0	0	1	0

What we have here is four examples (x_1, x_2, x_3, x_4), and they've got three binary attributes (A_1, A_2, A_3) as well as the output column. Running the decision tree learning algorithm on this dataset, for the first iteration:

First we'll calculate the entropy over the entire set. We've got four examples, split half and half between positive and negative examples: $B(\frac{2}{2+2}) = -(0.5 log_2(0.5) + (1-0.5) log_2(1-0.5)) = 1$

And then we'll get the remainder for our first attribute:

$Remainder(A_1) = (\frac{1}{4}) * B(\frac{0}{1}) + \frac{3}{4} * B(\frac{2}{3}) = 0.6887219$

The above being the fraction of the entire set where A_1 is 0, multiplied by the entropy of that subset (in this case, 0 out of 1 example is positive), plus the fraction of the entire set where A_1 is 1, multiplied by the entropy of that set (2 out of the 3 are positive examples).

Now we repeat for the other two:

$Remainder(A_2) = \frac{2}{4} * B(\frac{1}{2}) + \frac{2}{4} * B(\frac{1}{2}) = 1$

$Remainder(A_3) = \frac{1}{4}) * B(\frac{0}{1}) + \frac{3}{4} * B(\frac{2}{3}) = 0.6887219$

In this case, the numbers were the same for A_1 and A_3. Now to get the gain for each attribute, we subtract its remainder from the entropy of the entire set:

$Gain(A_1) = 1 - 0.6887219 = 0.3112781$

$Gain(A_2) = 1 - 1 = 0$

$Gain(A_3) = 1 - 0.6887219 = 0.3112781$

And then pick the attribute with the highest gain. Since A_1 and A_3 are tied, we can pick one arbitrarily. If we pick A_1, then we have to recursively do what we just did on the following two sets:

	A_1	A_2	A_3	Output
x_1	1	1	0	0
x_2	1	0	1	1
x_3	1	1	1	1

and

	A_1	A_2	A_3	Output
x_4	0	0	1	0

Two things to note here: that second "set" of one example is perfectly classified, so we don't have to do anything. We make a leaf node that outputs a 0. For the first set, splitting on A_1 would be redundant and useless, so we only need to consider A_2 and A_3:

	A_2	A_3	Output
x_1	1	0	0
x_2	0	1	1
x_3	1	1	1

Doing the calculations again, we get that this set has total entropy of $B(\frac{2}{3}) = 0.91829586$, and the Gain for each of the attributes:
$Gain(A_2) = 0.91829586 - (\frac{1}{3} * B(\frac{1}{1}) + \frac{2}{3} * B(\frac{1}{2})) = 0.91829586 - 0.6121973 = 0.30609858$
$Gain(A_3) = 0.91829586 - (\frac{1}{3} * B(\frac{0}{1}) + \frac{2}{3} * B(\frac{2}{2})) = 0.91829586 - 0 = 0.91829586$

So this time we split on A_3. Note that we had perfect gain (that is, the gain was exactly the entropy of the original set). This means we've perfectly classified the data and can stop.

4 Neural Networks

In the lecture, I mentioned you should know that a neuron passes the weighted sum $\sum_{i=0}^{n} w_i x_i$ of its inputs to its activation function g to determine its output (that is, the neron outputs $g(\sum_{i=0}^{n} w_i x_i)$). I also mentioned that you should have a high level understanding of how back propagation works. You don't need to memorize any of the mathematical formulae for it, but have an idea that we calculate an error at the output neurons, and that each neuron in a hidden layer will incorporate the error for all the neurons it outputs to when calculating its own error. This should be very easy if you've done the extra credit, but even if not I wouldn't stress too much over it.