15-780: Problem Set #3

March 31, 2014

1. **Partial-Order Planning [15 pts]** Consider a version of the milk//banana//drill shopping problem used in the second planning lecture (slides 17–26).

   (a) Modify the description of the *Buy* action (slide 18) so that the agent needs to have a credit card to by anything.

   (b) Write a *PickUp* operator that enables the agent to *Have* an object if it is portable and at the same location as the agent.

   (c) Assume that CC is a credit card that is at home but *Have*(CC) is initially false. Construct a partially ordered plan that achieves the goal state from the initial state (both of which are described on slide 17).

   (d) Explain in detail what will happen during the planning process when the agent explores a partial plan in which it leaves home without a credit card.

2. **Baysian Network [20 pts]** Suppose there is an alarm in a train engine that senses when the speedometer exceeds a given threshold. The speedometer measures the speed of the train. Consider the Boolean variables A (alarm sounds), FA (alarm is faulty), and FS (speedometer is faulty) and the multivalued nodes SR (speedometer reading) and S (actual speed of the train)).

   (a) Draw a Bayesian network for this domain, given that the speedometer is more likely to fail when the actual speed gets too high.

   (b) Is your network a polytree? Why or why not?

   (c) Suppose there are just two possible actual and measured speeds, normal and high; the probability that the speedometer gives the correct speed is x when it is working, but y when it is faulty. Give the conditional probability table associated with SR.

   (d) Suppose the alarm works correctly unless it is faulty, in which case it never sounds. Give the conditional probability table associated with A.

   (e) Suppose the alarm and speedometer are working and the alarm sounds. Calculate an expression for the probability that the actual speed is too high, in terms of the various conditional probabilities in the network.

3. **Solving an MDP using linear programming [15pts]** Recall from lecture that the optimal value function for an MDP is given by

$$V^\star(s) = R(s) + \gamma \max_a \sum_{s' \in S} T(s, a, , s') V^\star(s'). \tag{1}$$

In class we discussed two (model based) ways of finding this optimal value function: value iteration and policy iteration. Here we'll dicuss a third possibility: linear programming.

Consider the linear programming problem

$$\begin{aligned} &\underset{V \in \mathbb{R}^{|S|}}{\text{minimize}} \ \sum_{s \in S} V(s) \\ &\text{subject to } \ V(s) \geq R(s) + \gamma \sum_{s' \in S} T(s, a, s') V(s'), \ \forall a \in A \end{aligned} \tag{2}$$

where here we are optimizing over the elements in V. Show that the solution to this optimization problem is the optimal value function. Hint: first show that for the solution

$$V(s) \geq R(s) + \gamma \max_a \sum_{s' \in S} T(s, a, s') V(s') \tag{3}$$

and then show this inequality must be tight at the optimal solution.

4. **Value iteration for Markov decision processes (MDP) [20 pts]** For this programming assignment, you are to implement an MDP solver using the value-iteration algorithm. As you recall, an MDP solver, given a set of states, a transition model, a discount factor, and an error margin, will return an optimal policy, i.e., the optimal action to take for each state. Specifically, you are to implement the following function:

```
def compute_optimal_policy(num_states, transition_model, rewards,
                           discount, maximum_error)
```

where the arguments have the following forms:

num_states A positive integer. This integer represents the number of states, where the state names are the integers between 0 and num_states - 1.

transition_model - A dict of state transition entries, where the key is the state name and the value is an actions-outcomes table. An actions-outcomes table is a dict of action-outcome entries, where the key is the action and the val is a list of 2-element tuples representing the name of the resulting state and the probability of that state. States that are terminal have no transitions out and, therefore, do not need to be represented in the model. The following is an example transition model for two states.

```
{0 : { 'left'  : [(0, 0.9),
                  (4, 0.1)],
       'up'    : [(4, 0.8),
                  (0, 0.1),
                  (1, 0.1)],
       'right' : [(1, 0.8),
                  (4, 0.1),
                  (0, 0.1)],
       'down'  : [(0, 0.9),
                  (1, 0.1)]},
 4 : { 'left'  : [(0, 0.8),
                  (1, 0.2)],
       'up'    : [(1, 0.8),
                  (0, 0.1),
                  (2, 0.1)],
       'right' : [(2, 0.8),
                  (1, 0.2),
                  (0, 0.0)],
       'down'  : [(1, 0.8),
                  (2, 0.1),
                  (0, 0.1)]}}
```

rewards - A dict of rewards, where the key is the name of the state and the value is the reward.

discount - A real number between 0 and 1 and represents γ in the value-iteration algorithm

maximum_error - A real number between 0 and 1 and represents ϵ in the value-iteration algorithm.

compute_optimal_policy should return the optimal policy for the given states. The optimal policy is the list of actions representing the optimal action for each state, where each action corresponds to the state whose name equals the position of the action in the list. States that are terminal have "terminal" for their actions. The following is an example optimal policy:

```
['up', 'left', 'left', 'left', 'terminal"]
```

In this example policy, the optimal action for state 0 is up, for state 1 is left, for state 2 is left, for state 3 is left, and state 4 in terminal.

Your program will be tested with various inputs and validated by comparing against correct optimal policies for the inputs.

5. **Variable elimination in factor graphs [30pts]** For this problem you will implement the sum product variable elimination algorithm to perform inference in a factor graph. In particular, you will implement the following function, in the included file `factor_graph.py`:

```
def marginal_inference(factors, variables, elim_order=None):
```

This function takes as input a list of factors, describd via the `Factor` class in the `factor_graph.py` (more details on this shortly), a list of variable names for which we want to produce the marginal distribution, and (optionally) a variable ordering which consists of the order which to eliminate all the remaining variables.

The basis for the factor graph representation we will use is the `Factor` class included in the `factor_graph.py` file. You won't have to edit this class at all, or even necessarily understand all the code in this class, but you will need to understand how to use the class. Each `Factor` object contains 1) a Python dictionary containing all the variables for that Factor, along with their possible values, and 2) a set of factor values for all possible assignments to these variables. Let's look at a simple example: if we initialize `Factor` by the following:

```
f = Factor({"x1":[0,1], "x2":[0,1], "x3":[0,1,2]})
```

this will create a factor of three variables, "x1", "x2", and "x3," where the first two can take on values 0 or 1, and the third can take on values 0, 1, or 2 (note that in this problem, unlike the lecture notes, variables can take on more than two values, but this really introduces no added complexity). Note that there is no requirement that the list of possible values for each factor be numbers: indeed, for the real-world Bayes net example, these will typically be lists of strings, but this should not change your code at all. If you want to access this list of variables at any point, use the class member `f.variables`.

To access (either get or set) the particular factor value corresponding to some assignment of its variables, we can use the call

```
f[{"x1":0, "x2":1, "x3":1}]        # gets the corresponding value
f[{"x1":0, "x2":1, "x3":1}] = 0.1  # sets the corresponding value
```

(this will be the factor value when x1 equals 0, x2 equals 1 and x3 equals 1). For convenience (if you use this properly correctly, it can make your factor operations a bit more compact), if you use the get or set notation above, but with any additional variables that are not in the factor, the function just ignores these extra variables

```
# same as f[{"x1":0, "x2":1, "x3":1}]
f[{"x1":0, "x2":1, "x3":1, "x4":1}]
```

This does not work if you try to get or set the factor with fewer than all the variables specified (this will return an error).

If you want to get a list of all possible assignments to the variables, use `f.inputs()`; for example, in the above setting

```
for e in f.inputs():
    print str(e) + " = " + str(f[e])
```

will output:

```
{'x2': 0, 'x3': 0, 'x1': 0} = 0
{'x2': 1, 'x3': 2, 'x1': 0} = 0
{'x2': 0, 'x3': 1, 'x1': 1} = 0
{'x2': 0, 'x3': 2, 'x1': 0} = 0
{'x2': 1, 'x3': 0, 'x1': 0} = 0
{'x2': 1, 'x3': 1, 'x1': 1} = 0
{'x2': 1, 'x3': 1, 'x1': 0} = 0
{'x2': 1, 'x3': 2, 'x1': 1} = 0
{'x2': 0, 'x3': 1, 'x1': 0} = 0
{'x2': 1, 'x3': 0, 'x1': 1} = 0
{'x2': 0, 'x3': 2, 'x1': 1} = 0
{'x2': 0, 'x3': 0, 'x1': 1} = 0
```

since factors are initialized to have all zero values by default (or you can specify the default value as a second input to the `Factor` initialization). Don't worry about the ordering of the assignments here, a side effect of the dictionary representation is that the factors aren't stored in any particular order. You can also access a list of all the values themselves (in the same order as `f.inputs()`) using `f.values()`.

Using this representation, a factor graph (and hence a probability distribution over all the variables) can be represented using just a list of factors. For example,

```
f1 = Factor({"x1":[0,1], "x2":[0,1]})
f2 = Factor({"x2":[0,1], "x3":[0,1,2]})
f3 = Factor({"x3":[0,1,2], "x4":[1,2]})
fg = [f1,f2,f3]
```

Then `fg` impliclitly represnts a factor graph. This is possible (i.e., defining this distribution without ever explicitly defining a graph structure or the correspond edges) because each factor itself contains a list of all its variables, so we could immediately construct the graph given any such list of factors.

Using this representation, there are two parts to this problem:

(a) Implement the aforementioned

`def marginal_inference(factors, variables, elim_order=None):`

assuming that it will be called with a valid elimination ordering provided in ***`elim_order`***. Using the terminology described so far, we can define the inputs and outputs of this function more concretely:

- `factors` will contain a list of `Factor` objects, representing the factor graph for the distribution. For example, this would be the `fg` variable from the code above.

- `variables` contains a list of variables that we want to find the marginal distribution of. For example, this could be `["x1","x4"]`, indicating that we want to use inference to compute the marginal distribution $p(x_1, x_4)$ (i.e., summing out x2 and x3).
- `elim_order` contains a list of variables equal to the set differenced of all the variables in `factors` minus all the variables in `variables`, ordered by the order we want to eliminate them in using the sum product algorithm. For example, given the two settings above, `elim_order` could be equal to `["x2","x3"]`, indicating that we want to eliminate x2 first and then x3
- As its return value, this function should return a *single* factor, representing the marginal probability over the variables in `variables`. For example, for all the cases above it should return a factor of the form `Factor("x1":[0,1],"x4":[1,2])` where the elements of this factor correspond to the probabilities for each assignment of these variables (since they are probabilities, the sum of all the factors must be one).

To do this, you will want to implement the sum product algorithm described on page 18 and 20 of the probabilistic inference notes. Your implementation will be a lot easier if you also implement the functions

```
def factor_product(f1,f2):
```

and

```
def factor_sum(f1,v):
```

given in the code skeleton. The functions respectively would form the product of two factors (which you can use to then compute the product of several factors) and would sum (marginalize) out a variable from a particular factor.

Some test cases for this code, including a test case on a Bayesian network used for patient alarm systems in a hospital, are included with the problem code.

(b) Implement the

```
def marginal_inference(factors, variables, elim_order=None):
```

using all the same notation as before, but now in the case where `None` is based as `elim_order`. In this case, you will need to determine the ordering yourself, which will consist of some order for eliminating all the variables in the factor graph except for those contained in `variables`. However, since computing the optimal variable elimination ordering in a factor graph is NP hard, you'll need to use a heuristic.

You are welcome to try whatever heuristics for ordering that you want, but a method that will work for all the test cases we will give you is the "minimum neighbors" heurstic. Two variables x and y are defined to be neighbors if there is some factor in the factor graph that contains them both. Note that is not quite the same as the typical notion of neighbors in a graph, since all the direct neighbors of a variable in the factor graph would be factors; this is actually equivalen to a "two

step" neighbors in the factor graph itself. At each step of variable elimination, choose to eliminate the variable (not in `variables`) that has the fewest number of neighbors. If you do this incorrectly, variable elimination should still eventually give you the right answer, but it will most likely take too long for the test cases we give.