IMPLEMENTATION OF THE CORPP ALGORITHM

ALEX SCARLATOS

1. Introduction

I have implemented the algorithm described in the paper "CORPP: Commonsense Reasoning and Probabilistic Planning, as Applied to Dialog with a Mobile Robot" and implemented the test case described there (Shopping Requests) as well. The CORPP algorithm allows for a program to reason with database knowledge, and generate an optimal policy graph in order to interact with a user to find a goal state. Though I wanted to invent my own test case for the CORPP algorithm, I found that implementing the described test case was more difficult than I intended. However, it could be done in the future as I tried to make the CORPP code as modular as possible.

The source code for my implementation can be found at https://github.com/alexscarlatos/CORPP

2. Implementation

The original paper describes a very simple workflow for the algorithm:

(1) Generate a set of possible worlds using a Logical Reasoner (LR)
(2) Associate each possible world with a probability using a Probabilistic Reasoner (PR)
(3) Generate an optimal policy to follow using a Probabilistic Planner (PP)

2.1. **LR/PR.**

The paper recommended using ASP for the LR and P-Log for the PR, but I simplified this by implementing both with a single program in Prolog. This was done by carefully stating rules that describe the probabilities of different outcomes, and collecting all possible worlds with their probabilities.

Here is the core of the logic portion for my Shopping Requests implementation:

```
getTasks(L) :-
    findall(task(I,R,P,Pr), task(I,R,P,Pr), L).

task(Item, Room, Person, FinalProb) :-
    validTask(Item, Room, Person),
    findall(Pr, task_p(Item, Room, Person, Pr), SubTasks),
    \+ SubTasks = [],
    product(SubTasks, FinalProb).

% probability associations for place
task_p(Item, Room, Person, Prob) :-
```

```
    place(Person, Room)
    ->
    Prob = .8
    ;
    Prob = .2.

% probability associations for item
task_p(Item, Room, Person, Prob) :-
    probAtTime(Item, Prob) % references currentTime
    ->
    true
    ;
    defaultItemProb(Prob).

% probability associations for person
task_p(Item, Room, Person, Prob) :-
    currentPerson(Person)
    ->
    Prob = .7
    ;
    Prob = .3.
```

getTasks is our main predicate, which will return a list of all possible worlds.

validTask is the predicate that does the logical reasoning (determining which states are valid ones). It calls other predicates that are described in the original paper.

task_p predicates are probability rules that determine the likelihood of a state. The product of all probability rules for a single state gives its total probability. When writing these rules, you have to be careful to make sure that each task_p returns exactly one result for each state tuple (i.e. no reliance on backtracking) because of our reliance on findall.

These probabilities are then normalized outside of Prolog to sum to 1.

2.2. **Probabilistic Planner.** The original paper recommends using a POMDP solver to generate an optimal policy. I used the one freely available from POMDP.org.

To make use of the POMDP solver, you need to supply it with a set of states, actions, observations, transition functions, observation probabilities, and rewards. The descriptions below are mostly in regard to the shopping requests example, but can be easily interpreted in the context of similar problems:

- States: All possible worlds plus a 'term' state.
- Actions (what the robot can do): One 'which' question for each variable in the state (ex: which item do you want?), one 'is' question for each variable/value pair (ex: is the requested item a coffee?) and one execution action for each state (ex: deliver_coffee_lab_alice).

- Observations (what the user can do): 'none' for a filler observation, 'yes' and 'no' for answering 'is' questions, and each possible value for answering 'which' questions (ex: coffee, bob, etc.).
- Transition functions: Every question action should have the 'identity' transition function, meaning that the state should not change when a question is asked. Every execution action should have a full chance of ending in the 'term' state, regardless of the starting state.
- Observation probabilities: This part was not discussed in the original paper, so I will go into full detail below in regards to the shopping requests example.
- Rewards: Question actions should be slightly disincentivized (small negative reward), and then every execution action should have a high reward for transitioning into a corresponding state (ex: deliver_coffee_lab_bob from state coffee_lab_bob to state term = 50), and should have a high negative reward for transitioning to any non-corresponding state (ex: deliver_coffee_lab_bob from state coffee_lab_alice to state term = -100). You can tweak these values to determine the robot's behavior. For example, with smaller incorrect execution penalties, the robot is likely to ask less questions and more likely to jump to conclusions. I will discuss this more in the Results section below.

You may also give the solver an initial belief state, which should be the probability associated with each world. This is function of the PR: rather than making the PP ask questions for everything, give it an initial setting so that it has something to work with, thus requiring it to ask less questions.

At the end, the solver with generate a Policy Graph. This contains a list of nodes, each with an action to execute, followed by list indicating which node to jump to based on the observation made. The solver generates an alpha file, giving, for each node, a vector with weights for each start state. The node with the highest dot product with the initial belief state is the starting node. Starting there, simply navigate the policy graph, at each step executing an action, retrieving an observation from the user, and then going to the next state until we execute a terminating action.

2.3. **Observation Probabilities.**

In pomdp-solve, the notation is 'O: a : s : o n', which translates to "If you made action a and ended up in state s, there is an n/1 probability that you observed o."

In regard to shopping requests, I set up the observation probabilities like so:

- All delivery actions got a 'uniform' distribution, essentially meaning that it doesn't matter what is observed since a delivery action will get you to the term state no matter what. Everything following will regard question actions.
- Ending up in the term state always means you observed 'none', since it's impossible to end up there from asking a question.
- If you asked a 'which' question, for each state s, there is a 100% chance you observed the o that is contained in that state. For example, if you asked which_item, then for the state coffee_lab_alice, the observation 'coffee' has a 100% chance and every other observation has 0.

- If you asked an 'is' question, for each state s, there is a 100% change you observed 'yes' if that state is true for the question asked, and the observation is 'no' otherwise. For example, if you asked is_item_coffee, then for the state coffee_lab_alice, there is a 100% chance you answered 'yes' and every other observation has 0.

3. Results

The original paper seeks to validate the effectiveness of CORPP and its improvement over just using a POMDP. It does a good job of this, and makes sense as the LR and PR will only reduce the complexity of the POMDP problem. However, what is really in question is the usefulness and effectiveness of such an algorithm in general. Here, I will show some results testing the boundaries of my implementation.

3.1. Domain Size.

The major downfall of CORPP is that with a large enough domain to do something interesting, the problem becomes too complex for a POMDP solver. The number of states grows relative to the number of values for each state variable, and the complexity of the POMDP state is polynomially relative to the number of states (because there are NxN rewards for N states). It turns out that POMDP solvers begin to suffer severely with more complex init states.

I found that the solver did very well with N=4, and converged in .05 seconds. However, running with default settings on pomdp-solve, N=8 was taking about 10 seconds per epoch (it took 500 epochs to converge for N=4).

I found that by adjusting the 'epsilon' parameter, I could speed up execution, but at the cost of accuracy. Very high epsilons would complete after a few very short epochs, but would produce ridiculous results, like asking the same question many times and getting an incorrect delivery. So for different test cases I pushed epsilon as low as I could without getting crazy execution times. I realize that this is a huge downfall of any program that would be used in the real world, but I did it anyway for testing purposes.

3.2. Test Cases.

Below are some examples of the shopping requests test case. I tried to display the results of changing different variables and how they affect execution.

Rw = reward for 'which' questions, Ri = reward for 'is' questions, R+ = reward for correct deliveries, R- = reward for incorrect deliveries

```
Case 1:
    Relevant Defaults:
        currentTime(morning) (higher coffee probability)
        currentPerson(bob)

    Possible Worlds:
        ['sandwich', 'lab', 'bob']: 0.14
        ['sandwich', 'lab', 'alice']: 0.06
        ['coffee', 'lab', 'bob']: 0.56
        ['coffee', 'lab', 'alice']: 0.24
    epsilon = 1, pomdp-solve time = 0.04 seconds

    1. Rw = -1, Ri = -2, R+ = 50, R- = -100
      a: which_person
      o: bob
      a: which_item
      o: coffee
      a: deliver_coffee_lab_bob
    This is where CORPP seems redundant. Even though the result was correct,
    rather than generating a policy graph, we could have just asked the user those
    questions and gotten the same answer.

Case 2:
    Relevant Defaults:
        currentTime(night) (higher sandwich probability)
        currentPerson(alice)
        place(alice, office1)

    Possible Worlds:
        ['sandwich', 'office1', 'bob']: 0.059
        ['sandwich', 'office1', 'alice']: 0.546
        ['sandwich', 'lab', 'bob']: 0.059
        ['sandwich', 'lab', 'alice']: 0.137
        ['coffee', 'office1', 'bob']: 0.015
        ['coffee', 'office1', 'alice']: 0.137
        ['coffee', 'lab', 'bob']: 0.015
        ['coffee', 'lab', 'alice']: 0.034

    epsilon = 65, pomdp-solve time = 1.79 seconds

    1. Rw = -1, Ri = -2, R+ = 50, R- = -10
      a: deliver_sandwich_office1_alice
    Since there isn't much penalty for a bad delivery, the POMDP decided to
    immediately just go with the most likely delivery.

    2. Rw = -1, Ri = -2, R+ = 50, R- = -100
      a: which_room
      o: office1
      a: which_item
      o: coffee
      a: deliver_coffee_office1_alice
    CORPP was most successful here, since it didn't need to ask which_room
    given the evidence.
```

```
Case 3:
    Relevant Defaults:
        currentTime(noon) (similar coffee/sandwich probability)
        currentPerson(dan)
        place(alice, office1)
        place(bob, office2)

    Possible Worlds:
        ['sandwich', 'lab', 'dan']: 0.074
        ['sandwich', 'lab', 'bob']: 0.032
        ['sandwich', 'lab', 'alice']: 0.032
        ['sandwich', 'office2', 'dan']: 0.074
        ['sandwich', 'office2', 'bob']: 0.126
        ['sandwich', 'office2', 'alice']: 0.032
        ['sandwich', 'office1', 'dan']: 0.074
        ['sandwich', 'office1', 'bob']: 0.032
        ['sandwich', 'office1', 'alice']: 0.126
        ['coffee', 'lab', 'dan']: 0.049
        ['coffee', 'lab', 'bob']: 0.021
        ['coffee', 'lab', 'alice']: 0.021
        ['coffee', 'office2', 'dan']: 0.049
        ['coffee', 'office2', 'bob']: 0.084
        ['coffee', 'office2', 'alice']: 0.021
        ['coffee', 'office1', 'dan']: 0.049
        ['coffee', 'office1', 'bob']: 0.021
        ['coffee', 'office1', 'alice']: 0.084

    epsilon = 83 (any lower was taking too long), pomdp-solve time = 0.37 seconds

    1. Rw = -10, Ri = -1, R+ = 50, R- = -100
      a: which_room
      o: lab
      a: deliver_coffee_lab_bob
    This is where CORPP totally fails. Epsilon had to be very high to get this to
    run in any reasonable time (at 83 t<1 and at 82 I gave up after a minute),
    and thus the allowed error was very high. When we said we wanted to
    deliver to the lab, the solver jumped to the conclusion that we want to deliver
    a coffee to bob, even though there were 4 possible higher probability states
    (sandwich_lab_dan, sandwich_lab_bob, etc.).
```

4. Shortcomings and Possible Extensions

Other than the inability to deal with large sets of possible worlds, there are a few shortcomings in the theoretical base of the algorithm. Below are several that are not addressed in the original paper, along with some solutions that I think would improve the usefulness of CORPP greatly.

4.1. **Probability Learning Extension.** One extension that could be useful for the algorithm would be to adjust probabilities after successful executions. In its current state, the PR uses static probability values, which may be wrong or may change. The robot could instead be allowed to change these values based on its experiences. For example, if it finds that people are ordering coffee in the morning at a 90% rate rather than 80%, the probability will shift in that direction. Or if Alice is frequently making orders for Bob's office, the likelihood of the room being Bob's office when Alice is ordering will increase over executions.

4.2. **Dynamic Domain Extension.** A practical setback for CORPP is that it operates on a fixed domain. That is, each variable is given a set of possible values in the beginning, and the set of possible worlds only includes those fixed values. Ideally, we would want to be able to learn new values through user interactions (ex: maybe Alice wants a donut without having to modify the bot's database). Here is my proposal on how to do so:

(1) While running through the policy graph, we receive an observation that is not predefined.
(2) Remember all the user's answers so far as restrictions on the possible final state.
(3) Exit policy graph navigation.
(4) Add the user's most recent answer in the permanent set of possible values.
(5) Rerun the LR/PR, but with the restriction of the user's current set of answers (ex: if user answered 'lab' to 'which_room', then add Prolog assertion 'currentRoom(lab)'). The validateTask predicate will have to be modified to allow for these kinds of constraints.
(6) Rerun the POMDP solver on this narrowed set of possible worlds with the newest variable now included in the set of possible states.

Although running the POMDP solver multiple times would theoretically be a big slow down, every time we run it we are working with a smaller set of possible states, so it will run faster each time. However, the added domain values will undoubtedly make the POMDP solver run slower for new queries (without previous answers restricting the possible states). But this is a separate problem on its own.

4.3. **More Impressive Test Case.** While I like the idea of the algorithm, it is, in my opinion, major overkill for the test case used in the paper. The Shopping Requests test case requires the robot to discover a 3-tuple by interacting with a user. The robot could simply ask 3 questions, rather than generate possible worlds and calculate an optimal policy.

A better use for this algorithm would be in a situation where the state you are trying to discover is not directly linked to the user's answers, like if we expect the user doesn't know something or is withholding information. An example of this kind of system is WebMD, where a user inputs symptoms, and the program tries to find a reasonable diagnosis given all of the user's inputs. In the future, it would be interesting to see this kind of implementation, along with some performance optimizations, to test CORPP's true potential.

5. Resources

S. Zhang and P. Stone. "CORPP: Commonsense Reasoning and Probabilistic Planning, as Applied to Dialog with a Mobile Robot." *AAAI*, 2015.