Inferential Artificial Intelligence Methods

Kenji Mah

Ira A. Fulton Schools of Engineering, ASU Online
Arizona State University
Tempe, Arizona
kmmah@asu.edu

***Abstract*—In this report, we will be exploring the various ways of representing and solving different types of inferential problems. Factors such as the goal of the problem and the type of information available to us will influence how we design and model our solutions. We will specifically be looking at how to approach problems and formulate solutions using Bayesian Networks, Neural Networks, and AI Planning methods.**

***Index Terms*—Bayesian Networks, Neural Networks, AI Planning and Scheduling, Planning Domain Definition Language**

I. Introduction

The exciting and expanding field of AI has made significant progress towards solving modern problems such as autonomous vehicles, virtual assistants, and even overcoming a Go champion. This is possible by leveraging the data and information about a given problem and manipulating it in ways that become useful to us. The information can be presented in different forms such as direct samples from the problem's environment, known limitations of the problems environment, or even simple observations. The goal can also have a variety of forms such as inferring the probability of an event happening or devising a plan to achieve a certain state. These variables influence how we approach and model a problem in AI. We will be going over four different problems using different models to solve each of them.

II. Bayesian Networks

Bayesian Networks are probabilistic graphical models that we can algorithmically make inferences on. BN's compactly represent joint distributions of a set of random variables using directed acyclic graphs which we can then perform probabilistic queries about them. We will be using the pgmpy library in python to create our models and efficiently perform inferences on variables. In this section we will be looking at two different problems that use different variations of Bayesian Networks, a simple Bayesian Network, and a Dynamic Bayesian Network.

Data

For both networks, local conditional probabilities are required for each variable to define the models of the networks. For real world problems obtaining the dependency graphs require well tested experiments, previous historical data, or strong assumptions about the data. The conditional probability tables can be derived from samples of a dataset.

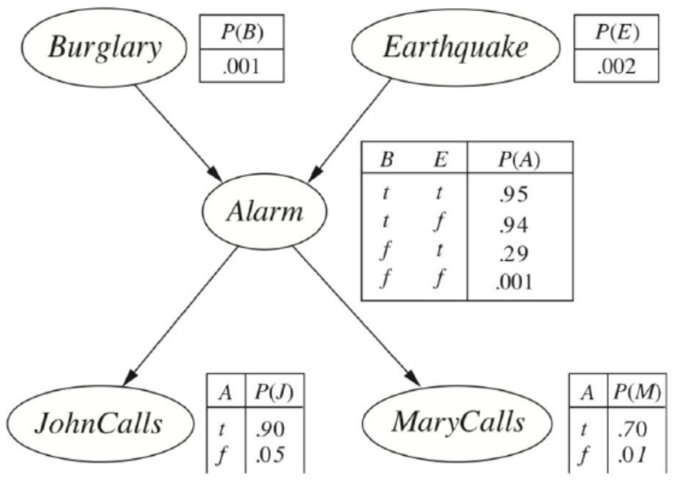


Fig. 1. Simple Bayesian Network model overview

Simple Bayesian Network Model

The first problem requires us to create a model that allows us to make inferences about data of an alarm system. The provided dependency graph with their probability tables are shown in Figure (1). From this model, we can make exact or approximate inferential probabilistic queries about the variables. Inferring the probability of the event that John calls given that there is an earthquake occurring is an example of such a query.

In order to create the model using the pgmpy library we need to only create a dependency graph and align them with their conditional probability tables with a specific format. The dependency graph will be represented as a set of tuples, each of which correspond to an edge in a graph (e.g. (A, B) where B is dependent on A). The local conditional probability table for a variable a is represented by a $m \times n$ matrix where m is the number of states that a can be in and n is the number of all possible combination of states between a's dependent variables. For independent variables, m will be the number of states a can be in and n will be one.

Dynamic Bayesian Network Model

In a new problem, we are tasked with making inferences about an agent in a 2×2 grid world that can only move in a

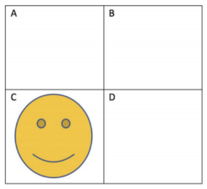
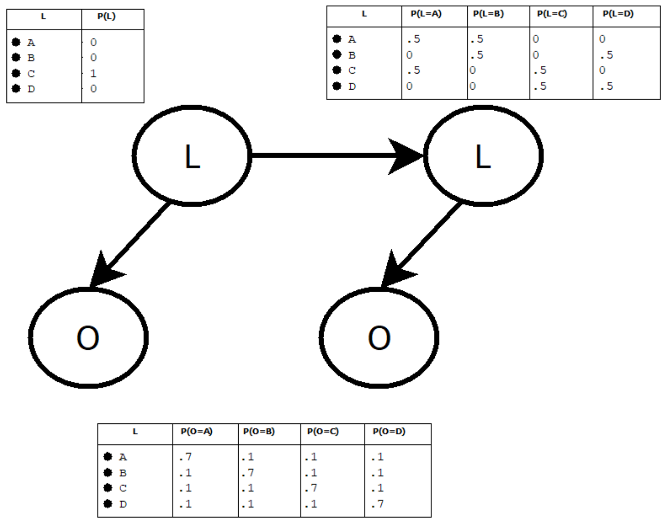


Fig. 2. Problem environment: An agent starts at C and can only move in a clockwise direction



L	P(L)
● A	0
● B	0
● C	1
● D	0

L	P(L=A)	P(L=B)	P(L=C)	P(L=D)
● A	.5	.5	0	0
● B	0	.5	0	.5
● C	.5	0	.5	0
● D	0	0	.5	.5

L	P(O=A)	P(O=B)	P(O=C)	P(O=D)
● A	.7	.1	.1	.1
● B	.1	.7	.1	.1
● C	.1	.1	.7	.1
● D	.1	.1	.1	.7

Fig. 3. Dynamic Bayesian Network model. $t + 1$ is dependent on t

clockwise direction. In this case the agent will start at square C as shown in Figure (2). The agent has a 0.5 chance to move to the next successive square and a 0.5 chance to stay in place. There is also a sensor that has a 0.7 chance to report the correct location of the robot, and a 0.1 chance to incorrectly report a reading to each of the other three states. This is a problem that differs from the previous problem because the state depends on a time variable. A Dynamic Bayesian Network, as shown in Figure (3), can be used in this case.

We created this model by first creating a dependency between the observation and location variables and formulating the conditional probability table with the proper pgmpy format. Because DBN's only rely on the previous beliefs of a network, we can include the temporal aspect of our model by creating a dependency between two copies of the network, one for time t and one for time $t+1$. In our case we will only need to create the dependency and conditional probability table between the location variables from t to $t + 1$. This is a special case of a DBN called a Hidden Markov Model because there is only one state variable (L) with one observation variable (O).

Results

After the models are created, we can input conditional probability queries to get a maximum a posterior probability of a variable as an output. The pgmpy model will efficiently calculate the queries by using a variable elimination algorithm to exclude any unnecessary calculations [1]. So, for example if we wanted to get the probability that Mary calls given evidence that a burglary occurred, and an earthquake occurred, we would need to input the query into our model following pgmpy's documentation [1] and output the probability distribution of that variable.

In the DBN model the pgmpy will handle the calculations regarding the filtering and monitoring steps for the DBN. So, we can create queries like the ones in the simple BN, however we also need to specify the time slices of the evidence and inference variables.

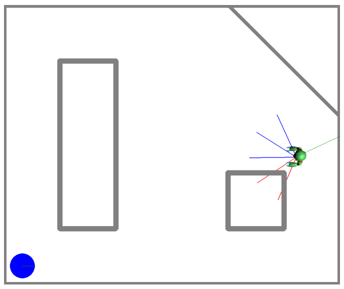


Fig. 4. Example of simulated environment. The blue dot represents the robot's goal.

III. Neural Networks

In another problem we are tasked with creating an application that helps a small robot navigate a simulated environment without any collisions. We will only be focusing on predicting if a collision will occur and not on other aspects of the problem such as the best path to take to reach the goal location, or how to avoid a collision if it is about to occur. The problem is situated in a controlled environment where if a collision would occur the robot is preprogrammed to use an action to turn in around in a direction away from the potential collision. The other action that the robot can perform is a moving forward action which can have various magnitudes to simulate speed. The robot is equipped with five sensors that are uniformly distributed in 30° increments in front of the robot to capture the distance it is from an obstacle. A visualization of the environment is shown in Figure (4) and was created using the following libraries: pymunk, pygame, and noise.

Data

For this problem we used simulations of robot collisions to gather our data. The process we used for simulating the crashes follow the pseudocode in Algorithm (1).

For our implementation we decided to run the simulation to obtain 20000 samples. However, we decided that it was best to down sample the majority class of no collision samples because it made up about 80% of our dataset. This was decided in hopes to reduce the number of false negatives in the testing set. To down sample our dataset we randomly selected n non collision samples, where n is the number of collision samples, and then shuffled them in with the rest of the collision samples. Then we normalized the data because the numerical values between the sensors and actions greatly differed and it is a

Algorithm 1: pseudocode

```
Data: []
randomly place robot in environment;
for number of samples do
    sensor_readings ← sensors;
    action ← choose random action;
    perform action;
    if action resulted in collision then
        collision = 1;
        randomly place robot in environment;
    else
        collision = 0;
    Data.append(sensor_readings, action, collision)
```

good practice to do in general to prevent a single feature to overwhelm the network. We then did an 80/20 train test split on the data. Lastly, we prepared our data as tensors because we will be implementing our neural network using the pytorch library [2].

Model

The reason we cannot simply set a threshold for the distance a sensor reads is because we also need to consider the speed of the robot. We also do not know the exact physics of the environment, so we will attempt to use a neural network to capture that information in a black box. The model we implemented is very simple because we are dealing with few features, so a relatively small neural network is enough. The network takes in 6 features which it then propagates through a single hidden linear layer with 6 nodes to output a number. The ReLu function is then applied to that output to create a prediction (1 for a collision, 0 for no collision).

Training and Results

We trained the network using the Stochastic Gradient Decent optimizer with a 0.01 learning rate and evaluated loss with the Means Square Error loss function. This was repeated over 5 epochs and the resulting test loss for each epoch is shown in Figure (5). After the first epoch we can see a significant drop in loss which was expected because of the simplicity of the problem. Our model contained 8/1000 false negative and 2/1000 false positives after being evaluated on the holdout set.

IV. Automated Planning and Scheduling

Like the problem where we implemented a Dynamic Bayesian Network, we have another problem where we have a closed environment and a definitive number of states that the system can be in. The objective in this new problem is for the agent to eat all the food items and then end at the goal location. We will call this the Pacman problem because of its similarity to the game. AI planning can be applied to this problem because it has the aspects of a classical planning environment in that it is fully observable, deterministic, static environments with single agents [2].

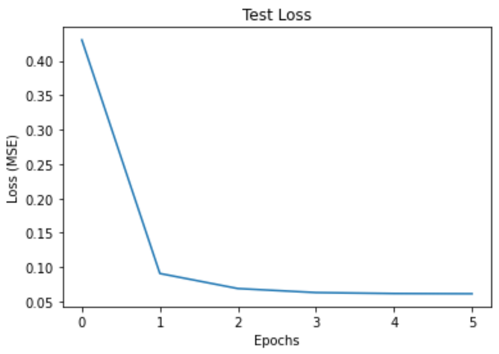


Fig. 5. Graph of test loss vs number of training epochs

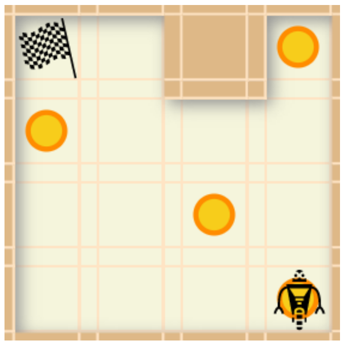


Fig. 6. Pacman environment. Circles indicate food the agent needs to eat before it reaches goal. The coordinate system starts with [1,1] at the top left corner

In order to represent this model for planning solutions, we need to express our world as a factored representation where the state of world can be represented as a collection of variables that are true or false. We will be using the language Planning Domain Definition Language (PDDL) to formulate our world and then use the Fast-Downward planner to solve and create a sequence of actions for our agent to execute and achieve its goal. When using PDDL, variables within the world are represented using relational representations.

Model

In PDDL there are two files that need to be defined, the domain file and the problem file. In the domain file, specify the world's predicates and actions. The predicates define the relational representations for the objects in our world. For our problem this is where we defined where Pacman is, the directional representations between locations (north, south, east, west), where food is, and whether the food is eaten at a location. The actions specify the functions to change the state of a world, which in our case will be moving actions and eating actions.

In the problem file is where we defined the objects, initializations, and the goals of our world. Our objects will only be Pacman and location. By implementation choice, food is not an object because we could just represent them as predicates for simplification of the model definition. If for some reason we would need to change this, such as different food properties, we could incorporate food as an object which would require us to change how we define the eating actions. The initializations define where Pacman starts, where food is, and a coordinate system for the locations. The goals define what variables need to be true for a plan to be considered successful. It is important to note that it does not matter what state the other variables need to be in for a plan to be successful.

Results

In order to use Fast-Downward we must pass it the domain and problem files as well as a heuristic to output a plan [3]. In this case we used the landmark count heuristic to find a plan. For the environment given in Figure (6), the planner outputs the following plan:

```
(eat-at pac1 loc_x4y4)
(move-north pac1 loc_x4y4 loc_x3y4)
(move-west pac1 loc_x3y4 loc_x3y3)
(eat-at pac1 loc_x3y3)
(move-north pac1 loc_x3y3 loc_x2y3)
(move-east pac1 loc_x2y3 loc_x2y4)
(move-north pac1 loc_x2y4 loc_x1y4)
(eat-at pac1 loc_x1y4)
(move-south pac1 loc_x1y4 loc_x2y4)
(move-west pac1 loc_x2y4 loc_x2y3)
(move-west pac1 loc_x2y3 loc_x2y2)
(move-west pac1 loc_x2y2 loc_x2y1)
(eat-at pac1 loc_x2y1)
(move-north pac1 loc_x2y1 loc_x1y1)
```

The outcome can possible depend on the heuristic used because there are many solutions to this problem. For example, the following is also another valid plan, but uses a different heuristic:

```
(eat-at pac1 loc_x4y4)
(move-north pac1 loc_x4y4 loc_x3y4)
(move-north pac1 loc_x3y4 loc_x2y4)
(move-north pac1 loc_x2y4 loc_x1y4)
(eat-at pac1 loc_x1y4)
(move-south pac1 loc_x1y4 loc_x2y4)
(move-south pac1 loc_x2y4 loc_x3y4)
(move-west pac1 loc_x3y4 loc_x3y3)
(eat-at pac1 loc_x3y3)
(move-north pac1 loc_x3y3 loc_x2y3)
(move-west pac1 loc_x2y3 loc_x2y2)
(move-west pac1 loc_x2y2 loc_x2y1)
(eat-at pac1 loc_x2y1)
(move-north pac1 loc_x2y1 loc_x1y1)
```

V. Conclusion

In conclusion, there are many different methodologies to represent and solve problems using AI. The use of these different types of models will allow a wide variety of problems to be solvable. The infinite variations of a problem make it difficult to determine the best methods and tools needed to solve them. In our case, we learned how to use specific tools such as pymunk and pygame to simulate problems, libraries such as pgmpy and torch to build models, and PDDL with Fast-Downward so create plans for our solutions. Even though each of these tools require specific assumptions such as the data of the problem or the resulting goal of the problem, we can use all of them in conjunction to formulate more robust intelligent agents.

References

[1] A. Ankan, Joris, D. Zhang, and D. Kasinathan, “pgmpy/pgmpy_notebook,” GitHub, 08-Oct-2020. [Online]. Available: https://github.com/pgmpy/pgmpy_notebook/blob/master/notebooks/2. Bayesian Networks.ipynb. [Accessed: 13-Dec-2020].

[2] “torch.nn,” torch.nn - PyTorch 1.7.0 documentation. [Online]. Available: https://pytorch.org/docs/stable/nn.html. [Accessed: 13-Dec-2020].

[3] S. J. Russell and P. Norvig, Artificial intelligence: a modern approach. Hoboken, NJ: Pearson, 2021

[4] J. Seipp, “Obtaining and running Fast Downward,” ObtainingAndRunningFastDownward - Fast Downward Homepage, 08-Jul-2020. [Online]. Available: http://www.fast-downward.org/ObtainingAndRunningFastDownward. [Accessed: 13-Dec-2020].