Implementing Agents that Perform Goal/Plan Recognition to Interact Better

Yu Chen
ggkevin.chen@mail.utoronto.ca
Undersupervision of Yves Lesperance
Department of Computer Science and Engineering,
York University,
lesperan@eecs.yorku.ca

Abstract

A cognitive robot is a robot that uses explicitly represented knowledge to make decisions about what to do. One possible cognitive agent is a plan recognition agent which recognize and react to intentions and plans of others. In recent years, people have been working on the implementation and formalization of plan and intention recognition methods. However, the framework for a plan recognition cognitive agent is not well established. In this project, I will produce a example of programming agents combining probabilistic plan recognition (Ramirez and Geffner, 2010) and belief-based control (Belle and Levesque, 2015), Using and possibly extending the programming cognitive robot language "Ergo".(J.Levesque, 2019)

1 Introduction

The programming language used in this projected is called "Ergo" and it is created by Hector Levesque. Before Ergo, there are other similar cognitive programming languages like GOLOG(Levesque and Scherl, 1997), CONGOLOG(Giuseppe De Giacomo, 2000) and INDIGOLOG(Reiter, 2001). These are the ancestors of the Ergo and many structures and ideas in Ergo can be found from those languages.

Some basic structure of Ergo will be introduced first and the methods for plan recognition and belief based programming will be talked in the methodology section.

Ergo is a language based on racket system and it is used to construct cognitive agents. The idea of basic action theory is used in the language.

Basic action theory contains the following three concepts: fluent, primitive action and state. A fluent is a property of the world that can be changed by an action. A primitive action is an event that changes some fluents. And a state is a setting of all the fluents in the world in a specific time.

The world starts in some initial state where each fluent has a certain values. Agents will perform some primitive action and as actions occur, the state of world changes and fluents come to have different values.

Cognitive robots in ergo usually has the following architecture(figure 1). The ergo program can receive exogenous actions or requests from user and determine the next endogenous action for the robot to execute. Then the action will be sent to the robot manager. In this project, robots(or agents) are running in a virtual environment and will be shown in some graphic interface.

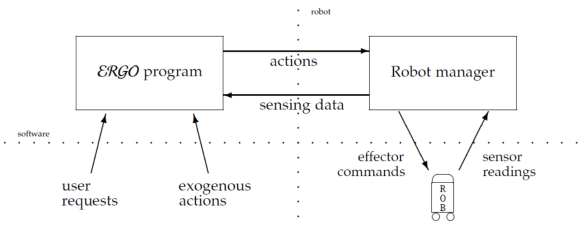

Figure 1: Architecture of a cognitive robotic system (J.Levesque, 2019)

2 methodology

There are two ways for Ergo to predict the next action for the robot. (ergo-simplan goal actions) and (ergo-do program).

Given a goal and a set of available actions, Ergo-simplan will find a shortest list of actions that solves the planning problem. The "goal" here is a function of no arguments that returns true in states where the goal is achieved. And "actions" is a list of actions that agent can perform.

Ergo-do function is a more complex function that searches and executes sequential programs. The

”program” argument is a nondeterministic program which can be executed successfully in more than one way. The ergo processor will find a sequence of actions that constitutes one of the successful execution of the program by backtracking search.

Ergo-do also have several different modes. The default one is the 'offline mode which will display a sequence of actions and then ask for confirmation from the user.

A more practical and frequently used mode is the 'online mode mentioned in figure 1. In online execution, an ergo program only computes the next endogenous action to be performed by the robot. The robot then executes the action and returns sensing data to the running ergo program. After receiving possible exogenous actions, the ergo program will update fluents and then computes the next action using the information acquired in previous steps, possibly looking ahead to future actions.

As mentioned in abstract, i am building examples of programming agents combining probabilistic plan recognition and degrees of belief(Belle, 2018). So there are two key implementations in this project, plan recognition and belief based control.

Plan recognition agent will try to analyze and predict other target's intention. Given a target agent and its actions, our agent needs to figure out what the target agent is trying to do. To achieve that, we have to pre-define appropriate plans for the target agent as a sequential program. For example, if the target is a customer in a Jewelry store, then it will have three possible plans, to browse, to buy or to steal an item.

We will create a non-deterministic sequential program that contains all the plans(which is called “exoprog” in the code). In the 'online architecture, target agent's primitive action will be received by ergo as exogenous actions. And each time there is only one incoming exogenous action. A method matching the incoming exogenous action with the pre-defined sequential program is used in the project. And it's an alternative mode of Ergo-do which is called the 'offlineStepMatch mode.

Ergo-do with offline step match mode has the following format: (ergo-do :mode 'offlineStepMatch :matchAct action exoProg). It takes two arguments, a match act that is a received exogenous action and exoProg which is the sequential program that contains all the plans. Then the ergo processor will search for a matching program that contains the given match action. It will remove the matched action from the program and return the residual program as the output of the ergo-do function. If no match is found, it will return the fail program. If the returned value is not a fail program, exoprog will be updated with the residual program. After receiving more and more actions, it will eventually converge to one of the pre-defined plan.

However, sometimes reactions need to be taken before it converges to a specific plan(100% sure about target's plan). So it's better to have a degree of belief for each plan so the controlled agent can take action corresponding to the plan with highest probability.

In order to determine the probability, a belief-based program is required. which is the second key word in the implementation. Belief-based program is a feature implemented in ergo to deal with uncertainty. In practice, there might be very little information that a cognitive robot actually knows for sure. Therefore, fluents in the world should be a degree of belief instead of an exact value.

This is how degree of belief achieved in Ergo. For example, if the initial position of the customer lies somewhere between 2 and 12 meters. Then instead of using one initial state, it will generate tons of initial states. (eg: one million states). And let h represent the position of the customer, ergo will sample h from a uniform distribution between 2 and 12. One million states are used here not because there are exactly that many possible values for h, but a larger number of states will model the distribution better.

```
(define-states ((i 1000000))
   h (UNIFORM-GEN 2.0 12.0))
```

Figure 2: Simple example of initialization of belief based program

Every state also has a weight fluent which is used when calculating the mean or variance of the sample. Weights are set to the same value for each state initially and can be changed due to future actions. A higher weight means this state will contribute more when calculating degree of believes and a zero weight means it won't be counted into the belief function.

Bringing this belief-based program back to the Jewelry store example, we can initialize the three plans each with probability 1/3. Because initially,

there isn't any information about the customer, so they should have same degree of beliefs. Each plan have different sequential program so the exoprog is assigned differently for each plan.

After receiving an action from the customer, the action will be passed into ergo-do under offline step match mode. If the action can be matched into exoprog, exoprog will be updated as the residual program. If not, the weight of this state will be set to zero and it won't contribute to the belief function any more. For example, if the customer perform an action called "Pay" which will pay for the selected item. Then the weight of states initialized with "Browse" or "Steal" plan will be set to zero since action "Pay" will never appear in these two plans. The probability of having a "Buy" plan will increase. This is the trick how the model filter out the plans that don't match the received exogenous action by setting the weight of the states to zero.

3 Examples

Two examples of agent combining plan recognition and belief based control are implemented in this project.

First one is the target shooting game example. Which is a simple example combines intention recognition and belief-based control. It is a modification from the example of "target shoot" written by professor Yves Lesperance. In the game, an agent is trying to aim and shoot a target and the system will observe and try to shield the aimed target. If the system shields the correct target, it will gain points. Otherwise the agent will gain points. The game can be played for many rounds until the agent calls "halt!". Each round consists of a sequence of actions defined as "oneRound". Whoever has the highest points wins the game at the end.

The belief-based control feature is used in this example. It is introduced since there are always noise and inaccuracy in real world situation. When the agent is aiming at a target of 30 degree, the agent won't always get the exactly same aiming degree. Therefore Gaussian distribution is usually used to mimic the uncertainty. in our example, if the agent calls "nSetAim!" actions, it's "aim" fluent will be sampled from a Gaussian distribution with mean equal to the exact degree (30 in this case) with a sigma of 5 degrees. And at the beginning of the program, 1 millions of states with same weight will be generated. This means we will have 1 millions of samples for the Gaussian which is very close the real Gaussian distribution.

In this example, the system receives exogenous actions from the agent. After each action, the belief of all the fluents will be updated and printed to the screen. The system will monitor the degree of belief and once the belief of aiming at certain target is higher than certain threshold or the observation ends, it will shield the corresponding target.

The second example is a more complex example which is called squirrel world. This example is based on a project from the book "Programming cognitive robots"(J.Levesque, 2019).

In the squirrel world (SW) setting, squirrels are represented as agents moving in a two-dimensional grid. Squirrels can sense the world to gather information and interact with objects inside the world. The SW world is represented as a 30 x 50 grid, acorns will be randomly generated at the beginning of each game. There are four controllable agents starting at each corner of the grid. The goal of each agent is to become the first squirrel to gather four acorns. There are several possible actions for each agent, like move, pick or drop an acorn, build a wall, eat an acorn etc...

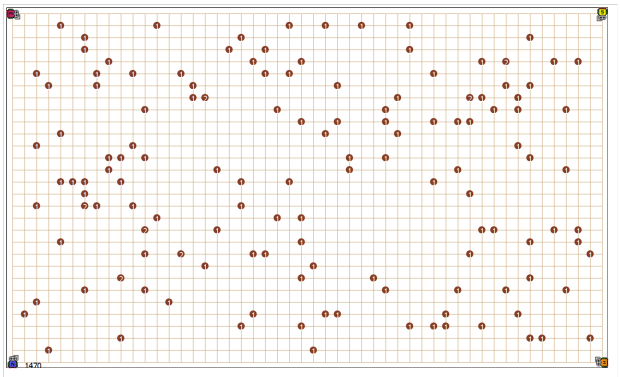

Figure 3: The graphic interface of squirrel world example

In the current version of SW, only two squirrels in the map will be activated. One is the controlled plan recognition squirrel and another one is the observed target squirrel. Compare to the first example, squirrel world has two agents that both need to send actions to the squirrel server which is the server that illustrate the graphic interface. Therefore controlled agent will have two input channel to receiving exogenous actions from both server and the observed agent. On the other hand, the observed agent will have two output channel to sending its exogenous action to both server and the

controlled agent.

There are 3 types of squirrel for the observed agent.

Squirrel 1 is a simple squirrel that just moves forward and turn around once it hits a wall. It records the number of steps to reach the wall and it will move between the origin and the wall forever.

Squirrel 2 will perform a systematic search for acorns near its home and stash acorns in the nest. The systematic search function is implemented in the book, the squirrel will search acorns row by row until it reaches the maximum capacity which is 2 in default. Then it will store acorns in the nest which is the starting point in default and it will go back to the block where it picked the last corn. The searching will continue until the end of game.

Squirrel 3 will also do a systematic search as squirrel 2 does. However, instead of picking up acorns immediately, it will memorize all the locations of the smelled acorns. And it will go back to pick up all the acorns once it found 4. Which is the minimum requirement to win the game. When the carrying capacity is 2, the shortest path for suqirrel 3 to win the game is to pick up 4th and 3rd acorns, drop them in the 2nd acorn's location. Then pick the 1st acorn and drop it to 2nd acorns' location to win.

The controlled agent will try to recognize the plan of the observed agent and take reactions to win the game against different squirrel.

If the observed agent is recognized as squirrel 1, controlled agent's reaction is to leave the target agent, and try to win the game by performing a systematic search. Because squirrel 1 is a simple agent that only running around without picking any acorns.

If the observed agent is recognized as squirrel 2, controlled agent's reaction is to prevent target agent from wining. Since squirrel 2 will store the acorns in the nest, one way to prevent it from wining is to steal the acorns from the nest so the observed agent will never reach a stack of 4. Therefore, the controlled agent will just waiting at the nest and steal the acorns once squirrel 2 drops them into the nest. Eventually the controlled agent can stack the stolen acorns to win the game.

If the observed agent is recognized as squirrel 3, controlled agent's reaction is also preventing target agent from wining. Since squirrel 3 will drop the 3rd and 4th acrons to the 2nd acorn's location. The controlled agent will follow the target agent and steal the acorn once target agent drops. Eventually the controlled agent can also stack the stolen acorns to win the game.

4 Challenging part

In this section, i want to show some challenging and tricky part in this project.

As mentioned before, for each plan, we need to make a non-deterministic sequential program which is name "exoprog". Therefore, given an ergo function of target agent, we need to transfer the main function into a non-deterministic program.

There is no constraint when writing a function. However, there are constrains when a function is transferred to the sequential program.

The program has to be a situational determined program. Which means given a matched exogenous action, the remainder of the program is unique. Figure 4 is an example of non-situational determined program.

```
(define (:starDFS prog)
  (:choose :nil (:begin prog (:starDFS prog)))
  )

(define (program)
  (:begin
   (:starDFS (:begin (:act forward) (:act smell) ))
   (:act forward)
   (:act halt)
   )
  )
```

Figure 4: example of program with non-unique remainder

The starDFS is just a star function that will perform the argument "prog" for arbitrary times.

The program will first do arbitrary number of pairs of "forward" and "smell". Then it will do another "forward" and "halt".

Then the following simple program should but won't match with the program defined in figure 4. Let simple_program = ((act :forward) (act :smell) (act:forward) (act:smell) (act:forward) (act:halt))

Because Ergo is using backtracking and only one exogenous action is received and matched each time. So it will always pick the "forward" action inside the starDFS function. The "forward" action outside the starDFS will never be reached. The ergo system will always expect a "smell" action after forward action, the "halt" action in simple_program will fail the matching.

Therefore, when implementing the sequential program for target's plan. One need to make sure

the action following the star function is not the same as first action inside the star function.

5 Results and Conclusion

Results for the target shooting examples are stored in the file "TargetShootGameV4". The text file "targetShootObs1Run1.txt" stores the observation for the first run. Due to the length of the observations, it won't be included in the report. The controlled agent(system) did a good job in the first run.

Here are some of the key points in the observation. First, the belief for aiming at target 0,1 or 2 are set to one-third after the action "chooseTarget!". Then after the exogenous action "(obsAim! 90)", the system observed that the agent is aiming at degree 90 which is closer to target 2. Then the belief of aiming at target 2 increased to 98%. After the "endObsAim!" action, since the belief of aiming at target 2 has the highest probability, the system will perform an endogenous action to shield target 2. And it will win the game if the agent choose to shoot.

Since the system has all information about the agent, it will always win in the normal situation. However, agent can still win in some weird cases like shoot without aiming, aiming at target 1 but the bullet hit target 2 etc... Some of these observations are stored in text files "targetShootObs2" to "targetShootObs5".

In the SW example, the controlled plan recognition agent has high performance especially against squirrel 1 and squirrel 2. If one runs the example code from the github link. One will see that the controlled agent will always win against squirrel 1 and squirrel 2. There are high chance that the controlled agent will win against squirrel 3. However, if the acorns are too close together or there are two acorns located in the 2nd acorn's location, the controlled agent won't have time to steal the acorns.

In conclusion, the ergo system has robust functions on matching sequential program and calculating degree of believes. Together with the offline step match mode and exoprog structure, simple examples of two agents can be easily constructed following the methodology mentioned above.

The only challenging part in the methods is to transfer observed agent's behaviors into a situational determined sequential program.

However, there are still weak points for the ergo system. The racket system itself is not as friendly as other programming languages(such as python, java etc..). An expert knowledge of scheme language is required in order to do modification on the ergo-do function. In the recent year, scheme language is less likely to be taught in the university and it's is used more as a teaching language for beginning programmers. It's also my first time working with scheme language and even after 4 month of studying and working, i still find it's difficult to write or modify a sophisticated scheme code like ergo-do.

6 Further improvement

There are several possible improvement for the squirrel world example. First, current version of squirrel world example only supports game with 2 agents. It will be more interesting and complex if this can be extended to a multi agent program.

The current version uses another set of naming for fluents and actions for observed agents. This structure need to be improved when more agents are involved in the examples. For every fluent, we should have a list of fluents instead of a single value. And the size of the list should be equal to the number of observed agents excluding the controlled plan recognition agent.

A even more complex structure of nested belief can be implemented in a multi agent example. If the observed agents are also plan recognition agent, controlled agent can have degree of belief of observed agent's believes. (eg. probability of target agent belief that controlled agent's plan is steal.)

The other point need to be improved for the squirrel world example is the probabilistic belief system. Currently there isn't much actions can interact with the belief system. The power of belief based programming is not fully expressed in the example. One possible extension is to modify squirrel 2 so that the nest area is assigned randomly. And then extend the nest area to a 2x2 square instead of a single block. All the acorns stacked in this area can be counted to the wining constraint. The observed agent can drop acorns randomly in the nest area and the controlled agent need to steal all the corns in the nest area. The controlled agent will assign target's nest area randomly in the initial state. Once the observed agent drops acorns, the weight of every state will be updated by multiplying with a 2D Gaussian distribution located at the dropping location. So the states with nest area closer to the dropping block will have higher weights after the

multiplication.

The last point is the way we construct the sequential program "exoprog". For this project, given a target agent's ergo code, I have to test and summarize all the possible actions that an agent can perform in different setting. And i have to make sure it's situational determined which takes lots of time. It will be very useful if we can create a function that can automatically map the main function of target agent to a sequential program.

References

andLevesque H Belle, V., editor. 2018. *Reasoning about discrete and continuous noisy sensors and effectors in dynamical systems*. Artificial Intelligence 262:189–221.

Vaishak Belle and Hector J. Levesque. 2015. *ALLEGRO: Belief-Based Programming in Stochastic Dynamical Domains*. IJCAI.

Hector J. Levesque Giuseppe De Giacomo, Yves Lespérance, editor. 2000. *ConGolog, a concurrent programming language based on the situation calculus*. Artificial Intelligence, Pages 109-169.

Hector J.Levesque, editor. 2019. *Programming cognitive robots*. Unpublished Manuscript.

R.; Lespérance Y.; Lin F.; Levesque, H. J.; Reiter and R. B. Scherl, editors. 1997. *Golog: A logic programming language for dynamic domains*. The Journal of Logic Programing 31(1-3):59–83.

Miquel Ramirez and Hector Geffner. 2010. *Probabilistic Plan Recognition using off-the-shelf Classical Planners*. AAAI.

Raymond Reiter, editor. 2001. *Golog: A logic programming language for dynamic domains*. The MIT Press.

A Appendices

All the codes and progress of this project are stored in the following Github repository: `https://github.com/NevermoreCY/ergo-project`
Please follow the instruction in README.md to install ErgoExt and run the examples.