Implementing Some Foundations for a Computer-Grounded AI

Thomas Lin

MIT
410 Memorial Drive
Cambridge, MA 02139
tlin@mit.edu

Abstract

This project implemented some basic ideas about having AI programs that are embedded in a computer file system environment. The programs learn, communicate, and take actions on the file system. They use the FF planning system between actions in order to generate new plans to decide which action to take next.

Introduction

This project implemented some foundations for a computer-grounded AI. A graphical control panel was created. Each program currently uses four methods: a knowledge base, a chat component, an emotions component, and a FF [Hoffman & Nebel, 2001] planning system. The system can currently run several demos.

In terms of 6.834 project objectives, this project looks at the FF planning method. In problem set two, students manually generated FF data files from a domain, and then use FF to generate a plan that solves the domain. In this project, FF is expanded to continuously generate plans (whose first actions get executed) in a dynamic environment.

Robots and Simulations

Much AI development has gone into developing autonomous robots in real life and robots in simulation. Much AI development has also gone into developing non-autonomous AI software that help users do things.

This project takes the stance that interesting AI can be written that is both autonomous and embodied only as software. In such a domain, I/O takes the form of computer I/O such as the keyboard and the network. Programs like this could read images and sound already encoded from the internet. They would access huge computer databases and libraries.

While programs developed in this environment would have trouble controlling robots, I believe they still have the potential to perform higher-order reasoning tasks. For instance, I don't see limits barring them from being able to conduct research alongside each other and alongside people.

Developing programs in such an environment has the advantage of creating robots in that the programs actually are in, and interact with, the real world. It has the advantage of creating robots in simulation in that everything is already in digital form, so more time can be spent working on reasoning techniques, as opposed to working on hard problems like robot motor control and filtering sensor input.

This project implemented some foundations for a such a computer-grounded AI architecture.

Development of Control Panel

A Java Swing control panel was created. This control panel lets the user start the program runs, watch the programs communicate, communicate with the programs, and see the internal states of the programs.

There is one code development directory (/Original-Mind). The user uses the control panel to specify that they want *n* copies of the program to be run simultaneously. Multiple copies can be run so that each copy can have other copies to interact with. The user presses "Go," and the Control Panel copies the original directory *n* times, into the /Mind1, /Mind2, ... /Mind*n* directories. The Control Panel then executes each copy of the program as a separate process.

Each program begins with the same state and knowledge, but they diverge quickly. The user can choose to say things to individual programs. The programs can explore the environment in random patterns. Programs begin execution at slightly differing times, so one might gain control of a limited resource before another.

Having the control panel also lets the developer see the internal states of the programs, and this makes it easier for him to understand the techniques being used and makes the programs easier to debug.

6.834 Final Project, Fall 2002

Each Program

Overview

The base Mind program currently uses four methods: a knowledge base, a chat component, an emotions component, and a FF planning system. These methods do not necessarily work together right now, but they would in the future. They were chosen because they all implement behavior or state that would be expected in a person.

During runtime, each Mind program repeats a basic loop. The main loop consists of: 1. Check for new messages. If any, add them to the knowledge graph. 2. Run FF and perform the first action in the resulting plan. 3. Update the emotions table. 4. Learn a random sentence.

The program is implemented as a mix of Java, Perl, C, and some precompiled code. This allowed for quick integration of any existing code. For instance, the main program is written in Java but the basic FF code I had was in C. So, I had Java do a command line call to run the FF code and retrieve the results.

Knowledge

Whenever a sentence is addressed to a program, it takes in the sentence, parses it, and adds it to a general knowledge graph. Basic inference can then by run over this graph.

The program currently has two methods of part-of-speech tagging the sentences before converting them into the patterns and concepts needed for the graph. The default is for the program to send the sentence to a copy of the Connexor parser running on a Media Lab machine. The backup is a locally implemented Brill Tagger [Brill, 1992] program that is part of each program's code directory.

After part-of-speech tagging each incoming sentence, the program separates the sentences into concepts and patterns. All adjacent adjectives and nouns in the sentence are turned into concepts. The pattern is the remaining part of the sentence after the concepts are removed. So for instance, "the fast cat ate the mouse" becomes concepts "fast cat" and "mouse" connected by the pattern "the ? ate the ?". A graph is then organized with each unique concept being a node, and the patterns being the edges.

The control panel allows users to see all of the concepts, patterns, and links between concepts and patterns that each running copy of the program has picked up.

Once a graph is in place, some basic inference can be done [Lin, 2002]. The program can now relate any two concepts by finding and returning the shortest path in the graph between those two concepts. The program can generate inference rules (e.g. "?x likes ?y" + "?y is often in ?z" -> "?x is often in ?z") by finding cycles within the graph and turning the concepts into variables.

To model how people do different independent research in the real world, the programs have access to a database of around 400,000 English sentences, and read random sentences from it. These random sentences are then tagged and added to the program's knowledge graph. This leads to quick divergence in what the different programs know.

Chat

Each program is able to chat with other programs and chat with the user. Each program's chat component is currently implemented as the Eliza program [Weizenbaum, 1966], but in future development this will change so that the chat becomes more correlated with the program's knowledge, goals, and choice of actions.

The current demo to show that the chat infrastructure works is to set up several programs and have them chat with each other. Because the demo is set up with Eliza right now, the conversation goes back and forth with no real topic. However, the demo is also a demo for showing knowledge graph construction. All knowledge passed to a program is added to that program's graph, so incoming chat is tagged, broken into patterns and concepts, then added to the knowledge graph. Eliza and part of speech tagging both run fast, so the user can see the graph being built very quickly.

Actual chat is currently implemented as having the programs write to and read from a log file. When the user has something to say, the control panel takes the user's message and appends it to the log file so that the programs are able to read what the user said. The programs currently open this log file in Java as a RandomAccessFile and each seek to the position they last read from. This way, if the log file gets large, the time it takes to read from the file does not explode.

Future directions for this include allowing the programs to communicate using email, over internet messaging protocols, and by posting on internet message boards.

Emotions

Emotions is implemented using one of Kismet's [Breazeal, 2001] emotion charts. Two internal numbers are stored: one for happiness/unhappiness and one for excitement/sleepiness. A space is formed with these two numbers as the axes. Different emotions come from having different values along these two axes. For example, a high value for happiness and a low value for excitement leads to the "content" emotion. A low value for happiness and a high level for excitement leads to the "afraid" emotion.

The control panel displays an emotion chart for each program being run. It uses a blue dot to update the current emotion values for each program.

The happiness/unhappiness and excitement/sleepiness values will eventually come from the chat, the goals/actions, and the knowledge. Positive or negative comments from the chat would have a small, temporary effect. Being able to achieve goals and successfully complete actions would lead to a medium range effect on emotions. Positive or negative outlook on the world from the knowledge graph would lead to a long-term effect. This setup has not yet been implemented yet though, and the emotions at time *t* for any program are currently set to the emotions at time *t*-1 plus a small random change.

Implementing FF

What is FF

In the Planning Problem, programs are given a set of possible states of the world, and possible actions to take in the world. Each action has a precondition and an effect. The program is then given a start state and a goal state, and asked to find the sequence of actions that can be taken to go from the start state to the goal state.

The main planning techniques that we learned in class were Partial Order Planning, Graph Plan, and FF. FF stands for "Fast Forward." FF can generate plans with both the STRIPS and ADL representations.

The FF planning system does fast plan generation through heuristic search. It runs by combining a relaxed Graph Plan with enforced hill-climbing. Because of its ability to prune the search space, FF is the fastest of the three methods. FF was the most successful automatic planner at the AIPS-2000 planning competition.

Applying FF to this domain

FF is used in the program to handle goals and actions. This is an extension of what was done on problem set two with FF. In that problem, we generated FF data files by hand and used it to generate one plan one time.

In its application here, the FF data files need to be generated by the program.

Also, FF is run multiple times here. The programs repeatedly sense the state of the world, generate the FF plan that helps them achieve their goals, then take the first action of that plan. Running FF multiple times lets the program plan in a dynamic environment.

If it wants to do 3 things, then the FF plan might be: 1. do thing 1, 2. do thing 2, 3. do thing 3. In a dynamic environment however, thing 1 might become undone while the program is doing thing 2. In this case, after doing thing 2, we'd want the program to generate the new plan: 1. do thing 1, 2. do thing 3, instead of just doing thing 3 and assuming that thing 1 is still done.

Another difference is that the actions used in FF here actually need to correspond with some code for executing the action.

How was it implemented

The programs use the FF executable from problem set 2. However, there is also an extensive amount of code "around" the FF executable that senses the state of the world, keeps track of the programs' goals, and generates the right data files.

As mentioned earlier in the Overview section, step 2 of the main loop is to "Run FF." Run FF consists of the following steps: 1. Update the state of the world, 2. Generate the goal state. 3. Execute FF. 4. Repeat step 2 until step 3 produces a plan or can't produce a plan that moves toward the goal state.

The first step is to update the state of the world in order to create a "Start State" for FF to work with. Some states of the world are internal, while others need to be sensed. For example, one state that would need to be sensed is whether a certain file exists on the file system. This would need to be checked again each time step 1 is run because an external force could have deleted the file.

Each program maintains a list of goals that it wants to achieve. Each goal also has a priority associated with it. Some goals are pre-programmed into the system. New goals can be added to the program during runtime.

The system uses a binary count to determine which goal state it wants to pass to the FF executable. For instance, let's assume that there are four actions: A, B, C, and D (ordered in terms of highest priority to lowest priority). The four digit binary numbers go 0000, 0001, 0010, 0011, ..., 1111.

0000 corresponds to the state that contains all four goal states. So first, the system calls FF with the start state and 0000 as the goal state. If a plan is generated, then the program moves on and executes the first action in the plan. If FF says that the goal is already satisfied, then FF moves on and does nothing, because it has already reached it's goal state.

If no plan is generated, then the system calls FF with 0001. 0001 corresponds to the state that has the first 3 goals but not the last one. A zero in position *n* means that goal *n* is in the goal state passed to FF, and a one in position *n* means that goal *n* is not in the goal state passed to FF. If 0001 returns a plan or returns true, then the program moves on. If it returns false, then the program tries FF with the next binary number.

If no combination of goals returns a plan or returns true, then the program does nothing because no action it can take will help it attain any of its goal states.

Executing Actions

I currently have Actions as a Java class. Each Action contains a unique name, a list of parameters, a list of preconditions, a list of effects, and also an ExecutionPattern. ExecutionPattern currently has an execution method that needs to be overwritten when defining the Action.

The ExecutionPattern specifies how the Action can be executed. So, if the execution method was overwritten with a method that prints "Hello World," then executing that Action would result in printing "Hello World." Actions are executed when they are returned as the first step of a FF plan.

The program currently does not allow the creation of new Actions during runtime. However, this functionality could be implemented as follows: The user tells the program the code for a new Action. The program then saves this code into a new file, compiles the file, and is able to start using it.

The ability to add Actions during runtime (either by having a user specify them, or eventually by having it learn new Actions on its own) is important for the program to be able to learn to do new things.

Control Panel Display of FF

The Control Panel allows the user to see what's going on with each of the techniques during the run. For FF, the user has five tabs to choose from. They can view a table of the Goals, which lists the Goals, their priorities, and whether they are satisfied. They can view a table of Actions, which lists all the Actions the program knows of, their preconditions, and their effects. They can view the "Plan," which describes the current state, the current goal state, the current generated plan, and a description of the current Action. They can view the "FF pddl" file, which is the list of Actions that the program passes to FF. Lastly, they can view the "FF facts" file, which is the list of states and objects that the program passes to FF.

The display is continuously updated. So for instance, if the state changes, you can see it updated right away in the "FF facts" file display. If the program had reached a TRUE state (where it satisfied some goal, and couldn't do better), but then the state changed so that a plan that could lead to a better goal becomes possible, then the "Plan" shown on the Control Panel is immediately updated to show the new plan.

Sample Domain

The sample domain which FF was being used on here is the "File Moving problem" that I came up with. In this problem, there is a common directory that all programs can look at. Each program wants to find a file with the word "Flag" inside of it, and rename that file to become the name of itself (the program's name). After it senses the goal state, the program will carry out the "(say-it)" action which makes it say "I win."

This problem fits the Computer-Grounded system, because moving files is one thing that programs based on a computer can do easily.

The goals in this domain are "(alive me)" and "(win-game)." Actions include "(move ?x - file)," "(set-state-to-win ?x)," and "(say-it)". The objects in the world are entities (e.g. "me"), file names ("filename"), filenames with the word "Flag" inside of them ("flag-files"), and possible files ("potential-files").

Each time FF is called, it updates the objects in the world. So, if the user is changing the contents of the common directory, the user can see the changes reflected in the objects list of the FF facts file immediately. The program also updates whether each "file" object satisfies the "(named-as-me ?x)" predicate. Then, it tries to generate plans.

If the program finds a file named "flag" with the word "FLAG" inside of it, then a sample plan might be to: 1. rename the file, 2. declare a win. Step 2 then has the effect of adding the goal "(say-win)," so this new goal is added while step 2 is being executed. "(say-win)" is only satisfied by the action "(say-it)," which also removes "(say-win)" from the list of goals. This ensures that each program only says "I win" once each time it wins.

When this problem is run with one program and one flag-file, the program renames the flag-file. When it is run with two programs and one flag-file, the programs go back and forth each renaming the file, declaring a win, then losing the file, and repeating. The user is able to see this happening in terms of the continually changing plans shown in the control panel. When it is run with two programs and two flag-files, it settles into a state where each program claims and renames one of the two flag-files.

In this domain, the user is also able to join in the game by going to that directory, and renaming files. For instance, if there are two programs running and one flag-file, then they will be fighting over the same flag-file. However, I can put another flag file in that directory. A directory listing ten seconds later usually reveals that my new file has been renamed, and now each program has it's own flag-file, so they are no longer fighting over the original file.

In the future, this will be expanded so that FF handles general actions across many more domains. The "File Moving problem" was mainly to show how FF could work at a basic level.

Future Work

The immediate future steps are to have FF add new actions, and also to combine together some of the existing techniques. Emotions needs to be combined with the other techniques to give meaningful output. Combining FF with the Knowledge Graph will allow Actions that involve using the knowledge gained.

Longer term future work includes adding more techniques (automated programming is especially relevant to the Computer-Grounded domain) and combining existing techniques more. Also, once the programs are advanced enough, maybe they could have a society structure where each gets assigned a different role, and some have authority over others.

Contributions

The two main contributions of this project are that it:

1. Explored ideas relating to creating a Computer-grounded AI system, including how to set up multiple agents that can each learn, converse, plan, and take actions at a basic level.
2. Implemented these ideas.

With respect to 6.834 project goals, this project took an existing method, the FF planning system, and expanded on it beyond what was done for class work.

References

Breazeal, C., (2001). Affective Interaction between Humans and Robots. ECAL 2001: 582-591.

Brill, E., (1992). A Simple Rule-Based Part of Speech Tagger. Proceedings of ANLP-92.

Hoffman, J., Nebel, B. (2001). The FF Planning System: Fast Plan Generation Through Heuristic Search. Journal of Artificial Intelligence Research 14, 253-302.

Lin, T. (2002). Analogical Inference over a Common Sense Database. Proceedings of AAAI-02, 955-956.

Weizenbaum, J. (1966). ELIZA - A computer program for the study of natural language communication between man and machine. CACM 9, 36-45.