Brad Israel
08/01/08
Intro to AI
Project 1

Executive Summary

The artificial intelligence system that I chose to implement for this project is a recommendation system, that is designed to help thirsty people choose the style of beer that best suits their personal tastes. As a recommendation system, it is a type of expert system that is used in quite a few real-world applications, such as Amazon's online store recommendation system, which helps users find products that are related to products they have looked at or bought in the past. My system is much more simplistic than something like Amazon's, but it does demonstrate the power of using a rule and fact based system over the traditional conditional statements. After the initial overhead of setting up the rules, adding, modifying, or removing rules became very simple and flexible because the rules code was completely separated from the control and fact code.

In the end the flexibility of the system allowed me to expand it from the very simple examples I started with, to the final (at least for project 1) system, which seems to perform very well. It is able to return useful results to the user and show them why that beer style was chosen.

Requirements

The main requirement for this project was to choose an area of expertise that lends itself to be coded into some sort of expert system. Then we were to think about the design of the expert system as a whole and the relevant knowledge that would need to be included. After we had a fairly good grasp on the concepts, we chose the tool that we felt the most comfortable with and started to implement a simple proof-of-concept example that got the bugs worked out of the system. After the simple test worked, we were to expand the system to meet the specifications, mainly that there were about 12 rules and the system was able to tell you why it produced the output.

Specification

The main problem for the domain of recommending beer is trying to determine what the user/customer really wants based on their perception of what they like. Once the user answers a few questions, it is possible to start narrowing down the field of choices. That's why my system was designed to start at the most obvious questions, that most people would know, and then moves down into just a little bit of detail. Even with the basic amount of field narrowing that was employed in my system, it is quickly realized that the domain has a second problem. This problem is that not only are there thousands, maybe even tens of thousands, of different brands of beer, but there are also about one hundred commonly known styles of beer. To see more of them select the styles drop down box on the website: http://www.beersuggest.com/beer/page/1/. I started out this project thinking that I could maybe do a few different styles and then drop down even lower to start recommending actual beers in those categories. However, I soon realized that this would be a huge task and decided to concentrate just on recommending different styles of beer. I decided that I would just start with the 13 most commonly heard styles, namely Pilsners, Lights, Lagers, Oktoberfest, Bocks, Black Lagers, Hefeweizens, Pale Ales, IPAs, Amber Ales, Brown Ales, Porters, and Stouts. These styles covered most of the range of tastes, without having to create rules for Trappist or Braggots, and made my life and design much easier. What I wanted to achieve for the first part of the project was to get my system fully working and able to recommend one or more of those 13 styles, based on questions answered by the user. The user's

answers are then turned into the facts of the system, which are then run through the rules and inference engines to choose the style and show the user, if they want, the proof that the system's choice was correct. For the second part of the project, I would like to go back over my rules and make them a little more “fuzzy”, so that styles are able to match a wider range of patterns and so that the system is even more flexible.

The tool that I used to complete the first part of the project is called Pyke (http://pyke.sourceforge.net/) and it's 100% python based. There are a few reasons that I was a little bit apprehensive about using this tool for this project, first, it is a very young project that is still listed as being in “alpha” stage and the first release was only in November, 2007, so I was worried about it having bugs and other problems. Second, the documentation on the site is not the best. It's more like reading the Unix MAN pages than a tutorial with lots of examples and explanations. This meant that my project required a lot of trial-and-error testing to try and make everything work together properly. And finally, I have very little experience with using Python as a developer. I've used it before, but just to do some quick scripts never any projects. Luckily, I was pleasantly surprised by the maturity and stability of the alpha-status tool and once I carefully studied and played with the examples, I was able to implement my projects design with only a couple major glitches. I was even able to quickly become used to Python's syntax and now feel fairly comfortable with the language and will probably continue to use it for future projects. The reason I started looking into Pyke was because of the debate team's presentation on it. I thought it would be interesting to try and use a different tool than Jess, which most people in the class seem to be using. Hopefully, I will receive a Jess project of similar complexity as part of the project 2 requirements, so that I can compare how fast the two applications run and how well they work.

Pyke is a fairly unique tool for developing expert systems. It is able to do both forward and backward chaining, even combinations of the two in the same application, they developed their own rule file syntax, which is automatically compiled back into straight python code that implements the Rete algorithm. After the design work on the project, the hardest part about implementing it in Pyke was to convert my design into their rule syntax, or KRB syntax. All of my rules were forward chaining because the goal was unknown and the system had to link the facts together with the rules to find the goal, which in my case was a style recommendation. Forward chaining also has the advantage that the rule priority is setup so that the rules are run in the order given in the file, but if new facts are asserted during the generation of “chained” facts, the engine knows to go back and see if some of the previous rules affect the new facts. This allowed me to setup my priority as the rules being run in order, highest priority method, and have more specific rules override less specific ones. For my application, this seems to work really well. After I got the rules working, I started on the facts, which unlike most applications were not going to be universal facts, but facts that were generated based on the user's answers to prompts. Universal facts, in Pyke, are facts that are put into the knowledge engine and cannot be removed or modified, but specific facts are facts that can be purged from the engine and replaced. I think this is the best way to do it because I think of universal facts as sort of global variables that would apply to every users rules if the program was run in a continuous loop, which is not what you want when every user's answers could be different. The last part was to implement the main control program that initializes the engine, rules, prompts users, converts the answers to new facts, tries to find and prove the recommendation, and then outputs the recommendation and statistics to the user. The main part of the Pyke that I would have liked to learn more about is exactly how the prove functions work. Unfortunately, the functions have very incomplete and hard to understand documentation that left me very confused. The main documentation on proving goals is at http://pyke.sourceforge.net/using_pyke.html#proving-goals and basically only somewhat explains backward chaining proofs and there is also one forum post here http://sourceforge.net/forum/forum.php?thread_id=1946837&forum_id=744446 that goes into a little more detail. I assume that, at least for forward chaining, it is fairly simple to go through the rules and

link them together because they're already fully generated when the prove function is called.

Feasibility Report

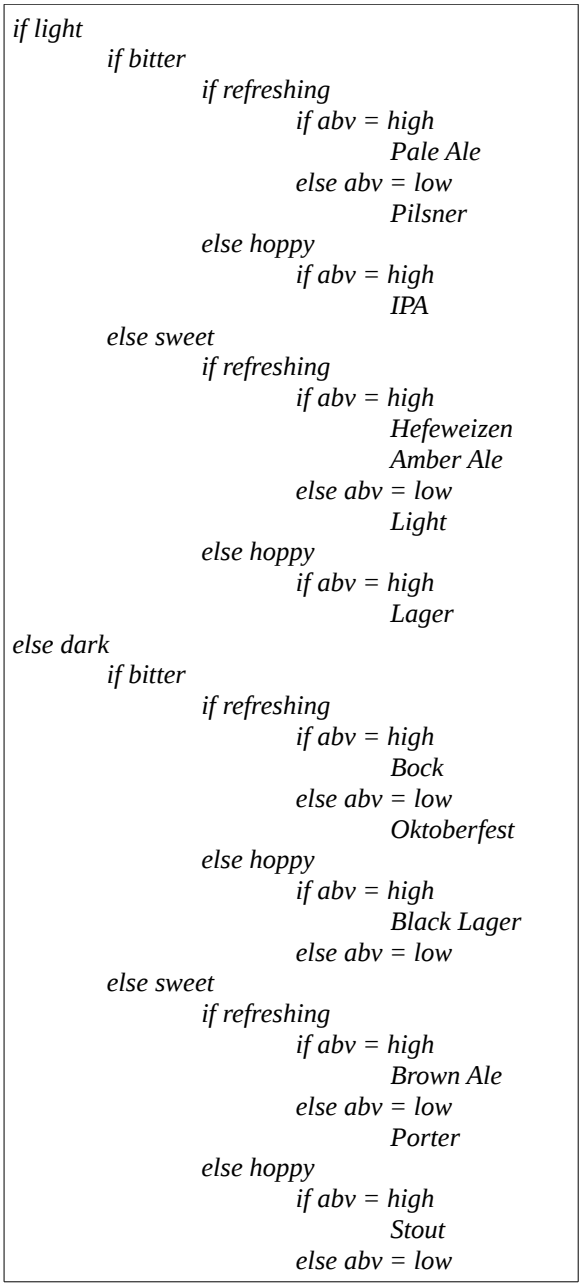
The obvious alternative to using Pyke for my project is to use Jess. Jess has many advantages, such as, it is a very well known and respected tool for developing expert systems. It is also available as a free download and integrates directly with Eclipse, which is the development environment that I am most familiar with. It is also based on Java, which is probably the language that I've had the most experience with. You would think that these factors would make it the best choice for my project and I should have used it instead of learning a new language as well as a new tool, but I am still very curious to see what the results will be when I compare my system to a similarly complex system written in Jess. Also, from what I've seen of Jess and heard other people in the class complaining about, it seems that the rule engine in Jess is much harder to work with than the KRB syntax in Pyke. I think the KRB syntax compiles it's simpler format into the more complex, LISP-like rules used by Jess and Pyke. It is also much easier to incorporate and use Python functions and libraries from within the KRB rules. When I was testing Pyke, I was able to type Python code into the rules just by prepending the command with 'python' and it even let me use python to convert variables back and forth between KRB and python easily. I never had to deal with the common problem in Jess of not being able to read or write to any non-global variable or having to setup and call Java functions from within the LISP syntax. Overall, I'm glad I stuck with Pyke instead of moving over to Jess, which I almost did when I got too stuck and confused when learning Pyke.

The other tool that I could have used to develop my system is CLIPS, which is a C-based tool that has been around since the 1980's. It is very stable, fast, and has much better documentation than Jess or Pyke. It has been continually updated over the years and is pretty much the defacto standard for certain subtypes of expert systems. From what I've seen, it uses a fairly similar style of rules as Jess because Jess was based on it, which I still see as a downside. I like tools that allow me to focus more on the design and make it easier to implement the design when it's ready, which is what Pyke does well. CLIPS is also C-based, which I know pretty well, but can sometimes be a pain to work with, especially under deadlines. In the end, I probably would have gone to Jess over CLIPS even though I know the language and the documentation is better because of the hassle that C can cause and I've gotten used to using nice IDEs like Eclipse that can make development easier. There is a Python library that links into CLIPS, which could have been another possibility because the library would hide some of the pedantic C things behind nice API calls. If application speed was very important, I would probably look into using the Python library or going directly to CLIPS, but my current system is probably not complex enough to make much of a difference.

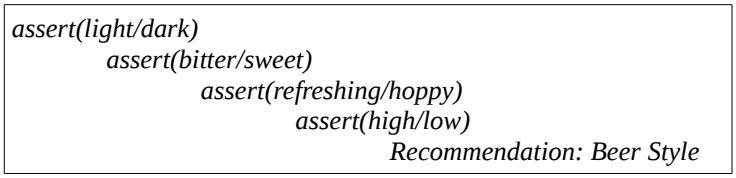
One design change that I would like to make would be to allow the users more open-ended control over their answers to the prompts. This would mean “fuzzifying” the rules a lot more to try and catch unexpected responses, while still trying to maintain some integrity over the final result. For example, if the user wanted a dark, sweet beer, the system shouldn't recommend anything in the light category, especially an IPA. It would also be interesting to see if my system could handle having the user enter in their own facts in a very open-ended prompt. It would basically be like letting the user decide what's important and trying to generate the facts themselves. I think parts of my current system are setup to allow that change easily, but the final style rules would probably need to be redesigned.

Implementation

After deciding what my narrowed domain focus was for this project, I started the rules design by creating a simple if/else tree structure that would later be converted into KRB syntax rules for Pyke.



This type of structure works well as a visualization for me and I found it very helpful to refer back to when trying to create the final rule set for the knowledge base. The data base is created by the main application 'pr1_bdi8241.py' when the user answer the prompts. The user specific facts that are created have the following structure:



Where the four main facts are asserted, then the recommendation is proved by applying the rules to the new facts.

There are two main code files that are need to run the recommendation system:

- pr1_bdi8241.py – Main python application file.
- beerrules.krb – KRB rules files. Contains the uncompiled rule set that follows the above tree structure in KRB format. Rules are automatically compiled when running the main program.

The project was implemented and tested on an Intel Core 2 Duo 64bit machine, but Python is cross-platform and interpreted, so there shouldn't be a problem running it on any other hardware that supports Python. The software requirements for compiling and running my project are Python 2.5, 2.4 should still work as well, Ply 2.5, and Pyke 0.3-1. More details on the software links and installation can be found in the User Manual section of this paper or the README file located in the code directory. The Pyke website and downloads section has a few example applications that I was able to draw influence from, especially the 'family_relations' example.

Once Python and Pyke are installed and working correctly, it is very easy to compile and run the project with the command 'python pr1_bdi8241.py'. I included a lots of different examples of the recommendation system in action in a file called 'IO_Examples' and in the Appendix of this document. The user interface is basically a user prompt system, where as the user answers questions, new facts are created in the background. After all of the questions have been answered, the rules are run against them to not only find a recommendation, but also show to the user how the system came to that recommendation. A nice feature of the output is that it first gives the proof in plain English form and then asks if you want a more detailed proof.

```
###############################################################################################
###                  Beer Style Recommendation System                                       ###
###                                                                                         ###
###              Author: Brad Israel (bdi8241@cs.rit.edu)                                   ###
###                 Introduction to A.I. - Project 1                                        ###
###############################################################################################
Please follow the prompts to select a beer style
Available input is inside the brackets, separated by ','
Code currently doesn't error check input, so it will probably error
 or not return anything if you mistype something.

Do you like darker or lighter beer? [dark,light]  dark
Do you like more bitter or more sweet beer? [bitter,sweet]  bitter
Do you like beer that's refreshing (crisp) or hoppy (dry)? [refreshing,hoppy]  hoppy
Do you like beer with a higher or lower alcohol content? [high,low]  high

You have decided on a darker beer style that is both bitter and hoppy with a higher alcoholic content.

I would look for a good BLACK LAGER

Do you want to see the full proof output and statistics? [y,n]  y
Proof:
dark -> bitter for options [ltdk] and [bs]
bitter -> hoppy for options [bs] and [rh]
hoppy -> high for options [rh] and [abv]
high -> black lager for options [abv] and [recommendation]


Statistics:
Time to generate rules: 0.00432 secs, 2780 asserts/sec
Time for proof: 0.00034 secs
beer: 3 fact names, 0 universal facts, 12 case_specific facts
beerrules: 15 fc_rules, 8 triggered, 0 rerun
```

beerrules: 0 bc_rules, 0 goals, 0 rules matched
0 successes, 0 failures

User Manual

Dependencies:

Any OS that runs the latest version of Python, I use Linux, but it shouldn't matter.

Python - I used 2.5.2-5 Linux 64bit, but all 2.4 and 2.5 versions shoud work
- Downloads and Installation instructions on: http://python.org/

Ply - 2.5-1 Cross-platform, Required by Pyke and is a tool for lex yacc parsing
- Downloads and Installation instructions on: http://www.dabeaz.com/ply/

Pyke - 0.3-1 Cross-platform, Main libraries for my project.
- Downloads and Installation instructions on:
http://pyke.sourceforge.net/installing_pyke.html
- I recommend downloading the examples from the same site and making sure that
- the family_relations code works. Type 'python' in that examples folder, then in the
- interpreter type the following lines:
'import test'
'test.fc_test()'
- You should see something that looks like a family tree with no errors.

Files:

- pr1_bdi8241.py – Main application and control code
- beerrules.krb – Rules file in KRB syntax
- README – User manual to help install and run my project
- IO_Examples – Some examples of user input and the corresponding output

Running my code:

- Extract the pr1_bdi8241.zip file
- There are only four files (pr1_bdi8241.py, beerrules.krb, README, IO_Examples)
- In a terminal/console change to where ever the extracted files are.
- Run 'python pr1_bdi8241.py'
- This will compile the python code and krb file and run the application
- Just follow the prompts to test it out. Enjoy!
- If there are any questions, please feel free to email me.

Appendix – Abridged Input/Output Examples

IPA Example

Do you like darker or lighter beer? [dark,light] light
Do you like more bitter or more sweet beer? [bitter,sweet] bitter
Do you like beer that's refreshing (crisp) or hoppy (dry)? [refreshing,hoppy] hoppy
Do you like beer with a higher or lower alcohol content? [high,low] high

You have decided on a lighter beer style that is both bitter and hoppy with a higher alcoholic content.

I would look for a good IPA

Do you want to see the full proof output and statistics? [y,n] y
Proof:
light -> bitter for options [ltdk] and [bs]
bitter -> hoppy for options [bs] and [rh]
hoppy -> high for options [rh] and [abv]
high -> ipa for options [abv] and [recommendation]

Statistics:
Time to generate rules: 0.00443 secs, 2707 asserts/sec
Time for proof: 0.00035 secs
beer: 3 fact names, 0 universal facts, 12 case_specific facts
beerrules: 15 fc_rules, 8 triggered, 0 rerun
beerrules: 0 bc_rules, 0 goals, 0 rules matched
0 successes, 0 failures

Hefeweizen and Amber Ale Example – Shows what happens when two styles can be proven.

Do you like darker or lighter beer? [dark,light] light
Do you like more bitter or more sweet beer? [bitter,sweet] sweet
Do you like beer that's refreshing (crisp) or hoppy (dry)? [refreshing,hoppy] refreshing
Do you like beer with a higher or lower alcohol content? [high,low] high

You have decided on a lighter beer style that is both sweet and refreshing with a higher alcoholic content.

I would look for a good HEFEWIEZEN or AMBER ALE

Do you want to see the full proof output and statistics? [y,n] y
Proof:
light -> sweet for options [ltdk] and [bs]
sweet -> refreshing for options [bs] and [rh]
refreshing -> high for options [rh] and [abv]
high -> hefewiezen for options [abv] and [recommendation]
high -> amber ale for options [abv] and [recommendation]

Statistics:
Time to generate rules: 0.00456 secs, 2850 asserts/sec
Time for proof: 0.00041 secs
beer: 3 fact names, 0 universal facts, 13 case_specific facts
beerrules: 15 fc_rules, 9 triggered, 0 rerun
beerrules: 0 bc_rules, 0 goals, 0 rules matched
0 successes, 0 failures

No Detailed Proof Example

Do you like darker or lighter beer? [dark,light] dark
Do you like more bitter or more sweet beer? [bitter,sweet] sweet
Do you like beer that's refreshing (crisp) or hoppy (dry)? [refreshing,hoppy] refreshing
Do you like beer with a higher or lower alcohol content? [high,low] high

You have decided on a darker beer style that is both sweet and refreshing with a higher alcoholic content.

I would look for a good BROWN LAGER

Do you want to see the full proof output and statistics? [y,n] n

Mistyped User Input Example – Automatically suggests a Lager by default.

Do you like darker or lighter beer? [dark,light] dark
Do you like more bitter or more sweet beer? [bitter,sweet] bisdfsfsdf
Do you like beer that's refreshing (crisp) or hoppy (dry)? [refreshing,hoppy] hoppy
Do you like beer with a higher or lower alcohol content? [high,low] low

Sorry, your choices didn't return a beer style ... if all else fails try a good LAGER!

Do you want to see the full proof output and statistics? [y,n] y
Proof:
dark -> bisdfsfsdf for options [ltdk] and [bs]
bisdfsfsdf -> hoppy for options [bs] and [rh]
hoppy -> low for options [rh] and [abv]

Statistics:
Time to generate rules: 0.00409 secs, 2688 asserts/sec
Time for proof: 0.00031 secs
beer: 3 fact names, 0 universal facts, 11 case_specific facts
beerrules: 15 fc_rules, 7 triggered, 0 rerun
beerrules: 0 bc_rules, 0 goals, 0 rules matched
0 successes, 0 failures

No Input Example – What happens when the user doesn't enter anything, what does the proof look like?

Do you like darker or lighter beer? [dark,light]
Do you like more bitter or more sweet beer? [bitter,sweet]
Do you like beer that's refreshing (crisp) or hoppy (dry)? [refreshing,hoppy]
Do you like beer with a higher or lower alcohol content? [high,low]

Sorry, your choices didn't return a beer style ... if all else fails try a good LAGER!

Do you want to see the full proof output and statistics? [y,n] y
Proof:
-> for options [ltdk] and [bs]
-> for options [bs] and [rh]
-> for options [rh] and [abv]

Statistics:
Time to generate rules: 0.00375 secs, 2936 asserts/sec
Time for proof: 0.00028 secs
beer: 3 fact names, 0 universal facts, 11 case_specific facts

```
beerrules: 15 fc_rules, 7 triggered, 0 rerun
beerrules: 0 bc_rules, 0 goals, 0 rules matched
        0 successes, 0 failures
```