CS8803
Knowledge Based AI
Project4

Puyan Lotfi

Implementation:

I used the same code from my project 3. For project 4 there were few design changes.

My project 4 implementation is actually 2 separate projects.

Implementation #1 of Meta Reasoner:

Originally I was under the impression that the meta reasoner had to ascribe intentions or meaning to every instance where aicop prints a message. I ended up implementing something very much like preconditions and postconditions (since I never implemented them to begin with, see my reports for project 2 and 3 for why), but I also added a field for "reason." The idea was that I would have a file written in the grammar from project 3 that would have statements like:

```
(META (call police) (running-from-the-police-to-escape-capture (run-away)))

ie:

(META (predecessor tags) (reason (successor tags)))
```

And that whenever aicop reports a frame that is part of a causal chain it would check to see if the previously printed event matches the predecessor tags and if the next frame to be printed belongs to the successor tags. This was the first implementation and I kept it. Since I did a second implementation I kept this one, but I don't call it the meta-reasoner I call it "The Oracle" so when you see code that prints something like:

```
Another Chain: rob-store-plan
Sub Script:
        (     ( go-to, ( robert-e-ford,store,,,, )))
Intention of robert-e-ford:
                go-to-store
Sub Script:
```

```
ORACLE SAYS: robert-e-ford wants-to-buy-some-stuff
        (     ( give, ( robert-e-ford,fake-check,clerk,,, )))

ORACLE SAYS: robert-e-ford wants-to-commit-fraud
```

The first implementation will read the “meta reasons” from the file meta.txt... they are quite humorous. The meta reasons are all store into a very embedded data structure called the MetaRegistry:

```
extern map<Frame*, map<Frame*, list<MetaFrame*> > > MetaRegistry;

struct MetaFrame
{
    list<Frame*> preds;
    list<Frame*> succs;
    string reason;
};
```

When each input in meta.txt is read, the predecessor tags and the successor tags are used to filter out a list of frames which are put into a MetaFrame instance. This instance is also given the reason string read from the input.

This implementation of the meta reasoner is only enabled if you give meta.txt as a command line argument to aicop.

Implementation #2 of Meta Reasoner:

I later looked at the example output and realized that a lot of what was expected was very much like debug output. I began implementing debug output with verbose explanations of the logic in my code: things like why one frame was used instead of another when there were multiple matches, or why intentions were ascribed to one actor and not another, or what the frequency of an agent or co-agent were in a causal chain which lead to a certain outcome.

I wrote this output much more like they were explanations than typical debug output, but the example output very much seemed like it was more centered around waiting until the end of a causal chain and telling why the intentions came out a certain way. So if the program is run with a command like argument of “enablemetaqueing.txt” then all the outputs of what and why things ended up as they did for a specific chain are printed out at the end of the causal chains. It ends up looking like this:

```
Another Chain: mayoral-race-shenanigans-plan

(     ( voted-against, ( john-torch,liqour-license,,,, )))
(     ( opposed, ( john-torch,gambling,,,, )))
(     ( raised, ( john-torch,30000,,,, )))
(     ( runs-for, ( john-torch,alderwood-mayor,,,, )))
(     ( were-taken, ( steamy-photos,starbucks,,,, )))
(     ( print, ( steamy-photos,starbucks,,,, )))
(     ( requested, ( pete-sartin,resignation,john-torch,,, )))
(     ( refused, ( john-torch,resignation,,,, )))
(     ( decreased, ( john-torch,effort,after-starbucks-scandal,,, )))
(     ( drops-out, ( john-torch,mayoral-race,,,, )))
Intention of john-torch:
        wanted-to-be-mayor-but-was-thwarted-by
political-opponents


META REASONER SAYS:

         - Most Frequent Agent: john-torch occurs 7 times in this chain.
         - Second most Frequent Agent: steamy-photos occurs 2 times in this chain.
         - Most Frequent Object:  occurs 10 times in this chain.
         - john-torch is the most frequent agent, so it must be central to intentions of
           actors in this chain.
         - Script Intention has no specific information about which actor the intention
           belongs too.
```

Core Components, Design and Methodologies.

Everything relating to these areas is mostly the same as what is noted in Project2_Report.pdf and Project3_Report.pdf. But here is a list of changes in each areas:

Core Components: The only struct added was MetaFrame, which is used in implementation #1.

Design and Methodologies: The overall design were unaffected. I still heavily rely on maps to get things done (especially with the MetaRegistry, where I use maps of maps). I also still use the grammar that I wrote in project 2, and it keeps coming in handy. The entire meta.txt is written in that grammar.

Self Critique and Evaluation:

Unlike what I claimed in project 3 (that I had no time to fix project 2 bugs), I actually did fix several bugs at the very last moment. For this project everything was strictly based on using the existing behavior of aicop to describe why it had reached the conclusion it did. I felt that I understood what aicop was supposed to do, and the foundations for adding functionality were not as shaky so overall I am happy with the results. I also ended up implementing two different meta-reasoners, so there is a lot of information that gets presented and I think that is good.

Overall I still think my code is very difficult to understand, and I still think if there is one thing that I could change about the design, it would be to have a more structured set of classes to help in calculating statistics on the frequency of actors used in a given causal chain. I cleaned up some of the indentation and variable names of that code in project 3, but I still have the same rant.

I still rely on the input files being in a correct format. Syntax and semantic type errors in the entries of a script a lot of times show them selves as segfaults. This happens less often than before, and a lot of times the lexer or the parser will catch a syntax error before it hits code that that would segfault on malformed input.

Knowledge Representation and Output:

The only place where I had to represent knowledge in the project was for implementation #1 discussed in the implementation section. I used the same yacc grammar that I wrote in project 2, but I just added a new keyword.

For implementation 2 I strictly just inserted code at points where it looked like some decisions were being made. I generally went about doing this by looking for structured control constructs like if else statements, for loops, and while loops. If the condition for taking a specific branch was non-trivial (which was very subjective) I added the statement “printMeta(“message”)” which is store until the chain is completed.

I don't see a need for example output here as I did before, since I had to cover that in the implementation section when discussing the two different implementations.