Traci Fairchild
tfairchild3
KBAI Final Exam
December 2, 2015

Synopsis
It is easy to imagine additional opportunities to use AI to help teach this AI course. Here are two. Your task in this final examination is to design AI agents that can exploit these two opportunities (both of which can lead to startups).

Q1. As you know the Piazza forum for this class is fairly busy. (As of now, the residential section with less than 100 students has more than 900 postings; the online section with less than 190 students has more than 6000 postings). Many of the postings are open-ended, novel and creative, but many fall under the category of "frequently asked questions": routine questions that, perhaps with small variations, are repeated semester after semester.

Design an AI agent that can automatically answer routine, repetitive, frequently asked questions on the Piazza forum of the KBAI class. Assume that the AI agent has access to all postings on the Piazza forums for previous sections of this class - some 40,000 in all. You may want to visit the Piazza forum to remind yourself of some of the routine questions and the answers to them. If such an AI agent was available for future sections of the class, it would be able to promptly answer many of the FAQ on Piazza.

My Conceptual Idea
In Piazza, we must filter each post as it is submitted. The first AI problem here is trying to determine whether an incoming post can be interpreted and answered using an agent, or if the post is unique and worthy of personal consideration by the piazza community.

For the purposes of illustrating this idea, I must introduce a hypothetical post:

> *T-square having problems?*
> *I am trying to upload my project 3 reflection and have noticed that T-Square has been taking a long time to upload. Is anyone else experiencing this? Other aspects of the site also seem really slow. I have tried in 2 different browsers and it takes forever or just says uploading....*
>
> *Thanks*

Where appropriate, this post will be referred to throughout this document as "incoming post". I chose this topic as an idea for a hypothetical post because it is a common issue at least once each semester and this issue has a standard, to the point answer. Note that this post would also benefit from the comments from other class members.

The logical response to a situation like this must be predefined. In this document it will be referred to as "the logical response". For this example the logical response would be the following:

Response:

In situations like you can always try the following:

1. Email your submission to <leadTA_email>
2. Send us a private piazza post with your attachment

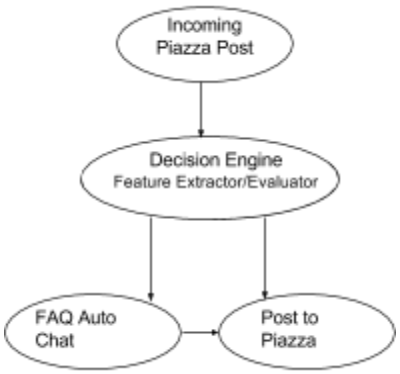
Note: please do within 15 mins after due time.

But note this is only a last option as this introduces a whole new manual cycle and your assignment would also miss Peer Review.

This document is designed using the topics discussed in class. Where appropriate, these topics are discussed one by one, using examples to illustrate the overall concept.

Semantic Nets

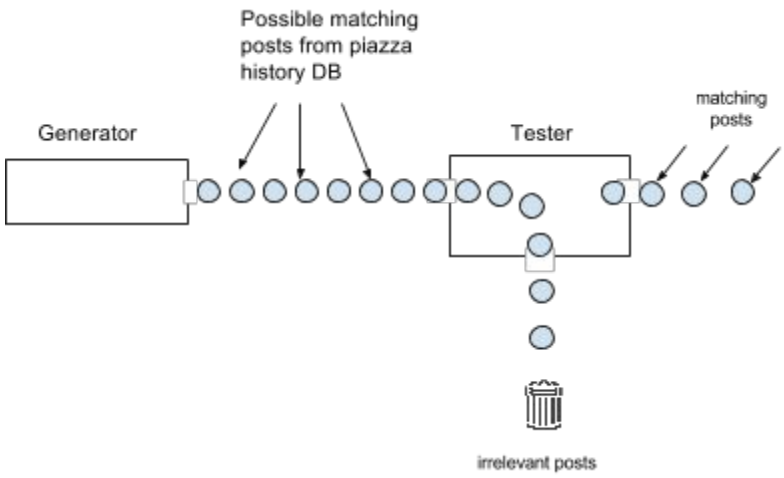
We have learned early on in the course that an AI problem with a good description and a good **representation** is almost solved. We can represent this problem using **Semantic Nets methodology. With semantic nets node-and-link representation** we can show each pathway that an incoming question could take, below:



The decision engine above will be the most important piece of the system. It will use many AI techniques in determining how to classify the incoming post. This can also be referred to as a **Generate-and-Test** system.

Generate and Test

The Generate-and-Test system, which is frequently used to solve identification problems, will be used to identify the type of incoming post it has received. It will search its database of past piazza posts and pull out the posts that have a match according to a predefined threshold. In this way, the system is generating possible **solutions** to the incoming post. A solution in this context is finding one or more previous posts in the database that has similar content and context to that of the incoming post. Once found the generate-and-test system can retrieve past solutions and test them for relevancy.



Means-Ends Analysis

In the context of problem solving, states correspond to where you are or might be in the process of solving a problem.[1] We can consider the matching posts found from the Piazza history database as our **state space** and we can use each post as an **initial state** as we move towards our **goal state**. The goal state here is to find an answer from the matching post that will reasonably answer our incoming post. As we move forward, that is, as we **transition** from one state to the next, we analyze each of them - their content and context, to draw comparisons with the incoming post.

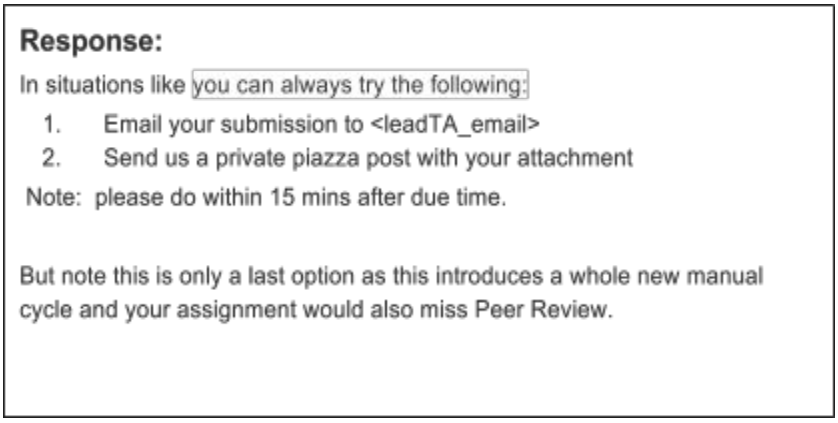
Production Systems

One of the major tasks of a cognitive system is to select actions.[2] And we now have the implementations of semantic nets and means-ends-analysis defined above which will provide the architecture for the next level of a cognitive agent, the **task or knowledge level**. This architecture is called SOAR (**S**tate, **O**perator **A**nd **R**esult). Within this level, the agent has access to two kinds of knowledge, oftentimes referred to as memory. The first is long-term memory which consists of **Procedural** knowledge (how to do things), **Semantic** knowledge (generalizations in the form of concepts and models of the world) and **Episodic** knowledge (specific instances of events). The second within the task or knowledge level architecture is **working memory**. The working memory in SOAR is a way to capture the precepts and goals of a situation in a feature: value format.

The agent will use detailed and specific knowledge from each post, past or present, to build the concepts of the **working memory**.

Recall our incoming post and desired response,

> *T-square having problems?*
>
> *I am trying to upload my project 3 reflection and have noticed that T-Square has been taking a long time to upload. Is anyone else experiencing this? Other aspects of the site also seem really slow. I have tried in 2 different browsers and it takes forever or just says uploading....*
>
> *Thanks*



with the data from the post we can build the concepts of the working memory as follows:

Source of Concern:	t-square
Description of Concern:	having problems, taking a long time, seem really slow
Action of Concern:	trying to upload
Attempted Solution:	tried in 2 different browsers
Subject Line is a Question:	yes
Num Questions in Post:	1
Keywords:	t-square, tsquare, upload, uploading, problems, time, slow
Goal:	upload

and the response could be predefined as

alternate_submission_procedure	In situations like you can always try the following: 1. Email your submission to <leadTA_email> 2. Send us a private piazza post with your attachment Note: please do within 15 mins after due time. But note this is only a last option as this introduces a whole new manual cycle and your assignment would also miss Peer Review.

The working memory defined above will help to identify and invoke the different types of productions stored in the procedural, semantic or episodic knowledge bases.

With the working memory established, we can move on to defining **production rules** which will reside in the **procedural** knowledge base and will help the agent make “thoughtful” decisions about the situation. A few

hypothetical production rules would looks like this, the first production rule listed below being the best rule for this paper's scenario:

Rule...	Then...
if goal is *upload*, and source of concern is *t-square* and description of concern like *slow* or *taking long*	suggest *alternate_submission_procedure*
if goal is *upload*, and source of concern is *t-square* and description of concern like *resubmissions allowed*	suggest *resubmission_policy*
if goal is *status* and source of concern is *t-square* and description of concern like *available* or *unavailable*	suggest *tsquare_maintenance_schedule*

When a course of action is decided, the agent can store this information into its episodic memory. The **episodic** memory can be useful when the AI agent reaches an impasse as to which rule to choose, in situations where more than one rule can apply. The agent will have the ability to save the production system it retrieved from the procedural knowledge base along with the solution to the episodic memory. The agent should have the ability to follow up on responses, searching for confirmation that the rule followed and posted was accurate, such as a reply of "Thanks", or "It's working now" or "Okay, I will try that" would give the necessary confirmation. The agent could store this

Source of Concern:	t-square
Description of Concern:	having problems, taking a long time, seem really slow
Action of Concern:	trying to upload
Attempted Solution:	tried in 2 different browsers
Subject Line is a Question:	yes
Num Questions in Post:	1
Keywords:	t-square, tsquare, upload, uploading, problems, time, slow
Goal:	upload
Suggest Rule:	alternate_submission_procedure
Result:	satisfaction confirmed

Frames

This brings us to frames and how we can represent what we know about the posts and how we can represent it on a detailed, class oriented level. You can do a lot more with frames than you can with production systems, so, using **slots** and **fillers**, we can populate frames into individual knowledge structures using the data we know about the incoming post.

Problem

Subject:	user
Object:	trying
Attempted Action:	upload
System:	t-square
Time:	<timestamp>

User	
Name:	tfairchild3
Days Online:	27
Posts Viewed:	120
Contributions:	24
Piazza Post ID's:	KBAI1234@123, KBAI1234@345, KBAI1234@876
Piazza Course ID's:	KBAI1234
Last login date/time:	<timestamp>

FORUM	
Name:	Piazza
num users online now:	34
Num users online this week:	216
Average Response Time (min):	42
Course ID's:	KBAI1234, SDP1234, SPECROB1234
Course Max Activity:	KBAI1234@123
Last Maint:	<timestamp>
Next Maint:	<timestamp>

CLASS	
Name:	KBAI

ID:	KBAI123
Unread Posts:	245
Unanswered questions:	1
Unresolved follow ups:	4
Total posts:	670
Total Contributions:	7060
Instructors Responses:	392
Student Responses:	191

Frames also represent stereotypes. A slow system could be signalled by few users and a long response time average. Representing a slow system then could look like this:

SLOW	
num users online now:	< 10
Average Response Time (min):	> 1
Num of logoffs in last 60 seconds:	86

These frames are stored in the **Semantic** memory portion of production systems and get pulled out when the subject of the frame is encountered in the piazza post.

Learning by Recording Cases

As the AI agent receives a post, it may not always be obvious as to what the proper reply should be. In situations such as this, the agent should search through its stored list of previous posts looking for a post that *is the same* as the incoming post - not equal in the words of the post, per se, but equal to the frame that the agent has defined in the working memory of its production system. This is an example of **learning by recording cases.**

Case-Based Reasoning

But as the AI agent continues to process posts, there will come a time where the incoming post is vague or the wording is not quite recognizable by the agent. Perhaps a few key elements in the defined frames is not accurate or incomplete. Enter **case-based reasoning.** Rather than searching for an exact match as in Learning by Recording Cases above, the agent must now search through the history of posts looking for a match that *most closely resembles* the incoming post. From this, the agent can extract the solution for the recorded case and use it to fill in the missing information, hopefully being provided with a reasonable reply.

Incremental Concept Learning

In **incremental concept learning** we learned that responding to examples improves models. I will talk briefly about how my AI agent could respond to incoming posts to improve its definition of the concept of response time. Initially the agent can take a few baseline tests and determines that a normal system response time can be defined as below

System Response Measure:	Normal
Number of Users:	25-100
Avg user response time:	37 min
Time of Day:	19:00 EST

However, how would the agent know when response time started to slow down? One way it could understand the concept of *slow* is if the agent started getting a few posts from users complaining with key phrases such as “having a problem logging on”, “system just hangs”, or “with no response”. This could trigger the agent to do some further metric gathering to help in it’s definition of response times.

Classification

Classification in this system is ubiquitous. We must classify the incoming posts, we must classify the replies. The challenge here is that the number percepts and the number of replies is very large. The agent needs to be able to map the percepts to the actions/replies that have been used. The first thing the agent needs to recognize is how to classify the incoming topics into smaller sets. This is already being done in Piazza by the user posting a question and should be used by the agent:

This is one way of classifying the incoming posts. Of course, there are other more effective ways the agent can classify its own understanding of the data, in a way that helps in the storage and retrieval of past and future problems. One example is classifying by keywords, such as forum keywords, or time keywords:

forums: t-square, piazza

concepts of time or speed: slow, slower, slowness, forever, hangs, too long, no response, deadline

experiences: experiencing, finally able, frustrated

events: outage, maintenance, holiday

Logic

Of course **Logic** is elemental in the design of an AI agent. We’ve seen it above in our production rules. Moreover, we need the agent to have a good ability to interact with the sentences and language being posted on the forum so the agent *must* have a good **knowledge base** of the world. With this we can build an **inference engine** where the

agent applies these rules of knowledge that it has to interpret the posts. With the inference engine, we need rules of inference so the agent can prove to itself it has the valid conclusion. With this, the conclusion must be sound and complete. With these rules of inference and a complete knowledge base of the world, our agent can begin to use logic to parse out the information. Here are some brief examples using real posts of writing things in the language of logic:

Planning

Planning is used for action selection. We use this to setting up goals and states. This is a very simple example of planning.

Precondition:	schedule-maint(tsquare)
Postcondition:	reply(alternate_submission_procedure, tsquare_maint_schedule)

Understanding & Common Sense Reasoning

As the agent begins to make sense of the incoming posts, it will use **thematic role frames** to organize the content. The words in bold are indicative of prepositional constraints which my agent will use to draw inferences based on its semantic categories. A common definition of a semantic category is "a grouping of vocabulary within a language, organizing words which are interrelated and define each other in various ways.".[3] Using the prepositional constraints below, the AI agent will be able to break down the content of the post which will allow the agent to properly build a frame that represents the contents. The agent uses the key prepositions in the post (indicated by bold) to make sense out of the content. After the agent processes the post and builds on its understanding of the content, the frame might look like this:

T-square having problems?

*I am **trying to** upload my project 3 reflection and have noticed **that** T-Square has been taking a long time **to** upload. Is anyone else experiencing this? Other aspects **of the** site also seem really slow. I have tried **in** 2 different browsers and it takes forever or just says uploading....*

Prepositional Constraints

Preposition	Thematic Roles
by	agent, conveyance, location
for	beneficiary, duration
from	source
to	destination
with	coagent, instrument

Thematic Role

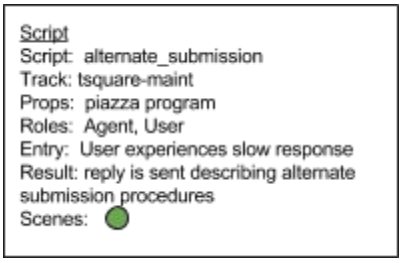
Verb: trying
Agent: Piazza user
Result: upload
Content: project 3 reflection
Instrument: t-square

Scripts

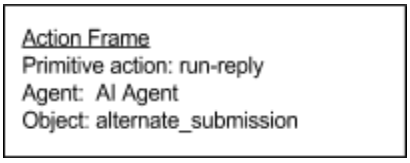
Now that the agent has reviewed and categorized the post, it is prepared to reply. The agent has a series of scripts available to ensure a coherent reply. As learned in class, a script is a causally coherent set of events, where each event sets off or causes the next event. The causal connections between events makes sense and the events are interpreted as scenes in the world. An event in this situation is the incoming message each time it is received in the forum.

One such script could be considered a "alternate_submission" script. An *entry condition* is a condition necessary to execute the script. An example of an entry condition for this script is the agent receiving a post regarding t-square slowness or unresponsiveness. This would cause the script to be executed.

The script would most likely contain *Tracks*. A track is a variation, or a subclass of the script. For example, one track might be a "tsquare-maint" track. Another track might be a "user-error" track. A third example might be a "resubmission_default" track.

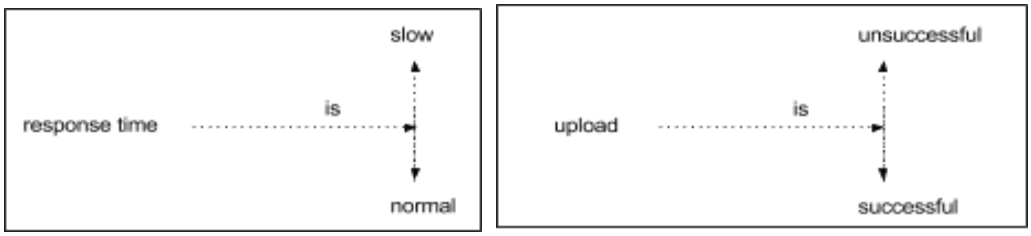


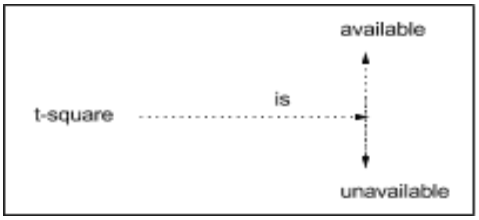
An *Action Frame* represents the actions needed in a scene of the script. In the track illustrated above, you can see they each have the attribute of Scene. The scenes for this track might look like this:



Explanation-Based Learning & Analogical Reasoning

How will the agent build an explanation of why this post is an instance of a concern of response time? The agent will use its prior knowledge to build this explanation in a process called **explanation-based learning.** Here the agent does not learn about new concepts with this method, rather it learns about the connection between existing concepts and the transfer of knowledge between an old situation and a new situation.





So, perhaps the agent in the past learned that when t-square is undergoing maintenance then the response time is typically slow. But in this situation t-square is not undergoing maintenance. So, the agent still concludes that if the response time is slow then it is possible that the upload functionality will be affected. Thus, the agent knows to respond in the same manner as previously - and that is to notify the user of alternate submission procedures.

Analogical reasoning involves understanding new problems in terms of familiar problems. Like explanation based learning, it involves the transfer of knowledge between an old situation and a new one.

Version Spaces

Version spaces technique of learning concepts incrementally we start with a specific model and a general model. As each new example comes along we decide if it is an example of a specific or general model. Suppose we have 5 posts that come into the system. Each one will be assumed to be a post regarding response time slowness, when in fact, they are not. The table below will illustrate this:

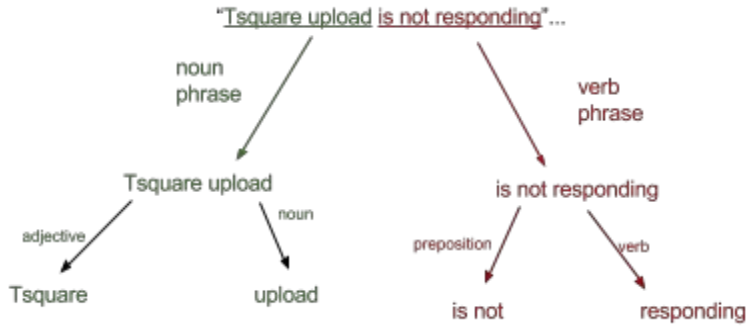
Piazza Post # - brief example	Keywords	Avg Response time in last hour	t-square available?	Reply with alternate_submission?
1. “tsquare upload is not responding”...	tsquare, upload, not responding	112 min	no	yes
2. “here is an article you may find interesting”...	article, interesting, you, url	47 min	yes	no
3. “Confused about question 1”...	question 1, confused	38 min	yes	no
4. “is anyone experiencing slowness?”...	slowness, anyone, experiencing	82 min	yes	yes
5. “can’t upload my profile pic on piazza”	upload, profile, piazza	33	yes	no

As you can see, the agent can make sense of the posts using the keywords in combination with the avg response time which will go up as the systems degrades, and some key metrics about tsquare such as is the system knowingly available or not (as in, a maintenance window is occurring). As the agent processes each post, it considers all of them applicable to suggesting an alternate submission response. But as it looks at the keywords and various measures it can asses what is and what is not a positive example of a “slowness” post.

Constraint Propagation

This is a way the agent make inferences and assign values to satisfy constraints. In language processing the sentences must be grammatically correct and it is assumed that the users of the piazza system are entering

sentences that are grammatically correct. So we can parse the incoming posts into a pass tree so that from there we can perform further analysis. With one of the examples above, the pass tree could look like this:



Configuration

Configuration is a routine design task which is very common in AI. It is a problem solving activity that assigns values to variables to satisfy constraints. So,
if the inference is that the user should receive a reply of alternate submission procedures, then the configuration would contain several sub tasks, such as

A. place the incoming post in the FAQ section of piazza
B. reply to the post informing the user of the alternate procedures
C. reply to the post informing the user that this was an automated message

Diagnosis

Diagnosis is the process of identifying faults and problems within a malfunctioning system. The agent should have a built in process of examining mistakes. For example, once a post is placed into the FAQ category and responded to, the system should make a button available to the user which will allow them to report that the process followed was not the desired effect. When this button is pressed, the agent should be able to analyze its mistake, store the post for future learning, and then post the question to the open forum for public review.

Learning by Correcting Mistakes

Finally, how can the agent adjust for, or compensate for its mistakes? It would first need to know and understand that a mistake has been made. This would be directly indicated by the user pressing the button to take the post out of the FAQ section. At this point the agent would know that a mistake has been made.

So, the agent receives notification that the interpretation of the post was incorrect. How can the agent isolate this error in it's former model? The agent could relook at the constraints and verify it's mapping. How can the agent explain the problem that led to the error?
This learning is incremental and explanation based. Before a re-reply is attempted, the agent can revisit past failures to ensure they do not happen again, if avoidable.
So, now armed with the reason for an inaccurate reply, the agent can repair the model to prevent the error from reoccurring. --- The agent could take in more knowledge of the world and try to understand what led to the initial failure and reattempt the configuration.

Meta-Reasoning

This is thinking about thinking. The agent is reasoning about itself and its own knowledge. How does the agent know what it does not know? This is the important question in meta reasoning.

Conclusion

As we have seen over and over again, making sense of the world around us is crucial in designing an AI agent that can replace humans in certain situations. The agent must be able to use everything it can to understand a situation and react or respond appropriately. This scenario seems simple at it's core, but trying to teach the agent to learn from its incoming data is paramount, because the designer simply cannot account for all possible conversations or words that the agent might be presented with. A good design would include most topics we have discussed in class.

References

[1]Winston, Patrick Henry. *Artificial Intelligence*. Reading, Mass.: Addison-Wesley Pub. Co., 1992. Print.

[2]Udacity.com,. 'Classroom - Udacity'. N.p., 2015. Web. 27 Nov. 2015.

[3]Bibliography: What is a semantic category? (1998, July 30). Retrieved 1 November 2015, from http://www-01.sil.org/lingualinks/LANGUAGELEARNING/OtherResources/GlssryOfLnggLrnngTrms/WhatIsASemanticCategory.htm In-line Citation: ('What is a semantic category?', 1998)

Q2. Again as you know, each term the students in the KBAI class produce a few hundred AI agents for addressing problems similar to those on the Raven's test of intelligence. Let us call them Student-Agents. By now we have several hundred Student-Agents. Some of these agents are quite good: they likely would give fairly decent performance on the Raven's test. However, none of the Student-Agents is perfect.

Further, just like we can think of different kinds of problems on the Raven's test, we can think of different types of Student-Agents: some types of Student- Agents may be better suited to some kinds of problems, but not necessarily others. You may want to look at your own agents, the agents you have reviewed, as well as the exemplary agents in this class to remind yourself of some of the different kinds of Student-Agents.

Design a meta-AI agent that can combine the capabilities of the hundreds of different types of Student-Agents such that the performance of the meta-AI agent on the Raven's test is superior to that of any of the individual Student-Agents. This meta-agent may invoke the right Student-Agent (or Student- Agents) for each different kind of problem. If such a meta-AI agent was available, students in future sections of the class will be able to observe its decisions and behaviors, and learn from it.

My Conceptual Idea

The Meta-Agent that I am designing will be utilizing and managing hundreds of different types of student-agents. The goal with a meta-agent is to maximize performance and results across these student agents to produce high accuracy with any set of RPM's.

To begin, the Meta-Agent will take as input a set of RPM problems and a set of student agents with which the Meta-Agent will be able to execute. In many multiagent applications, an agent needs to be able to reason about other agents[2] so the first thing the Meta-Agent will do is to look at a state table that is being maintained for each student-agent. And because beliefs change with time, the Meta-Agent will need to update and examine this information after every problem is attempted. The start of a new RPM problem is considered the beginning of a cycle.

Initially the Meta-Agent will send the first few problems to all of the student agents. As the student-agents begin to return from their execution, the Meta-Agent will be capturing the chosen answer and recording metrics to update the state table.

As for selecting the proper answer, the Meta-Agent will keep a tally of each answer coming back from the student-agents and when the count of one answer far exceeds the count of other answers, then the Meta-Agent will assume that this is the correct answer and return it to the calling module. All other completions will be ignored except for the data retrieved with regard to its performance. Before the agent moves on to the next cycle, the agent will update the state tables of each student agent using this data.

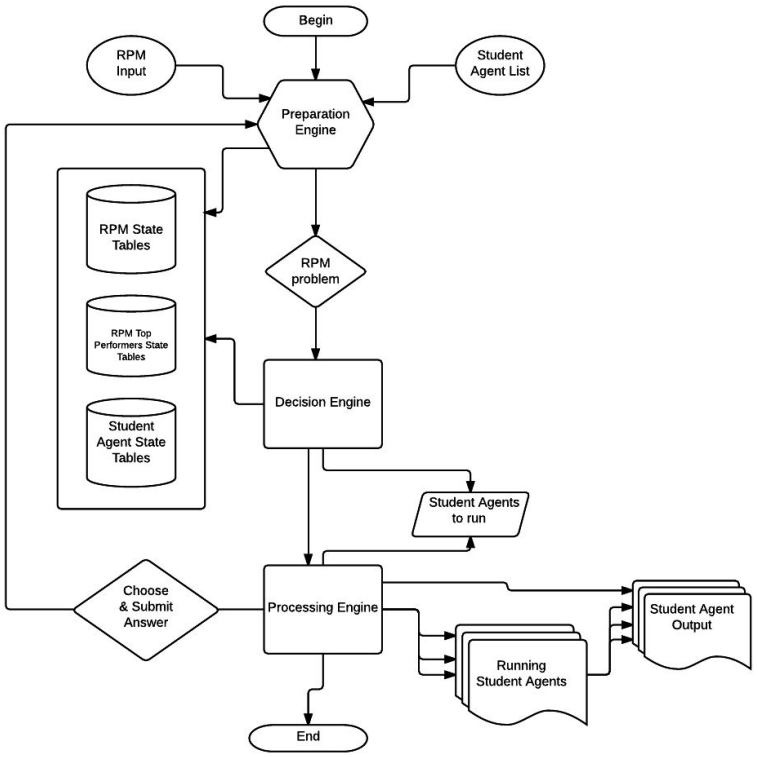
Assumptions

1. A list of student-agents, and their locations will be provided. The (student-)agents should be autonomous and should behave according to a clearly articulated set of operating principles.[1] We saw these principles defined in the requirement documents for each project and therefore they are expected of each agent.
2. The RPM problems contain the same level of information and complexity as the individual projects provided, both 2x2 and 3x3, both with verbal representations and without.
3. The student-agents will return only one piece of information, and that is the selected answer. All other output is ignored.
4. The Meta-agent will parse and handle the incoming sets of problems in the same way that RavensProject.py does.

Overall Design Summary

The design of the Meta-Agent is fairly simple. The agent has a set of processes it must repeatedly perform:

1. Agent initiates *preparation engine*
    a. *preparation engine* receives a list of RPM problems as input
    b. *preparation engine* receives a list of student-agents to load
    c. *preparation engine* initializes state variables
    d. *preparation engine* sets up the next problem to process
2. Agent initiates *decision engine*
    a. *decision engine* analyzes a single RPM problem and builds information about it
    b. *decision engine* reviews preconditions: state of the other agent's, state of RPM
    c. *decision engine* selects and builds a list of student-agents that it wants to execute for the current problem
3. Agent initiates *processing engine*
    a. *processing engine* initiates each student-agent that the decision agent has selected
    b. *processing engine* reviews postconditions, such as answers and processing status
    c. *processing engine* agent reviews answers returned from selected student-agents, chooses best one and submits it
    d. *processing engine* checks input for correct answer and updates state tables
4. agent repeats process



Major Components

Input Data Summary

The main data used by the Meta-Agent will include the following:

1. RPM Input
    a. a set of RPM problems as provided by projects 1 - 3
2. RPM State Table
    a. State table for each problem (ie, problem type, hasVerbal, list of student-agents who have attempted to process this question, and the processing time)
3. RPM Top Performers State Table
    a. State table regarding problem categories, such as *hasVisual* and *type*
4. Student-Agent Input List
    a. List of student-agents and their locations
5. Student-Agent State Tables
    a. State Table for each student-agent. These state tables will contain individual belief structures for each student-agent (ie, problem attempted, speed, correct, skipped, wrong, efficiency, availability)

Preparation Engine

The Preparation Engine will read and initialize the problem sets and the Student-Agents when the Meta-Agent begins. The 4 main input elements are set up as follows:

A. RPM Input

The preparation engine will first read the contents of ~/Problems/ProblemSetList.txt into memory and then empty the file. Then, for each problem set found in ProblemSetList.txt ("Basic Problems E" for example), the preparation engine will open and read the contents of the corresponding problem file into memory (~/Problems/Basic Problems E/ProblemList.txt) and empty that file as well.

The idea is, one by one, the Preparation Engine will write one problem set at a time back to problemSetList.txt and one problem at a time back to ProblemList.txt so that is able to manage the processing of the student-agents..

Assume the ProblemSetList.txt file data is slurped into memory and stored in an array called prob_set_list. Also assume the problems are also slurped into memory and stored in an array called prob_list, everytime a new problem is needed, it will be written back to original file:

Algorithm
For Pset in prob_set_list: write Pset to original ~/Problems/ProblemSetList.txt file For NewProb in prob_list: write NewProb to original file ~/Problems/Basic Problems X/ProblemList.txt

This will allow the the student-agents to only process one problem at a time and still allow them to run as designed. More importantly, this will allow the Meta-Agent to measure the performance of each student-agent once it completes. Once the Meta-Agent is satisfied with knowing how each Student-Agent

performs, it can start to process more and more problems at once by simply writing any number of them back to their original file.

This also gives the Meta-Agent some flexibility in that it can control the type of problems it wants to send to any chosen student-agents. For example, the agent may discover that student-agents # 22, 43, 87, 12, and 19 are particularly good at processing "2x2 hasVerbal" problems. Therefore, the agent can write only the "2x2 hasVerbal" problems to the ProblemList.txt files and then choose to initiate the student-agents #22, 43, 87, 12, and 19.

B. RPM State Table

The Preparation Engine will create a state table for each RPM in the list. The RPM State Table will be a hash structure holding information about each RPM problem.

RPM State Table Example

```
rpm['Basic Problem E-01']['name'] = 'Basic Problem E-01'
rpm['Basic Problem E-01']['set'] = "Basic Problems E"
rpm['Basic Problem E-01']['type'] = "2x2"
rpm['Basic Problem E-01']['answer'] = 3
rpm['Basic Problem E-01']['hasVisual'] = 1
rpm['Basic Problem E-01']['hasVerbal'] = 1
#average run time for this problem
rpm['Basic Problem E-01']['avg_run_time'] = 0
#the student-agent id's for the top 10 performers for this problem
rpm['Basic Problem E-01']['top_agents'] = ''
```

Note: `avg_run_time` and `top_agents` will be updated after each problem is considered complete.

C. RPM Top Performers State Table

It is important to save processing information about each type of RPM problem so the agent can learn from, enhance its overall process. In this design the Meta-Agent will update a state table used for keeping track of this type of information. The state table will look like this:

RPM Top Performers State Table

```
rpm-top['2x2']['avg_run_time'] = 0
rpm-top['2x2']['top_agents'] = ''
rpm-top['3x3']['avg_run_time'] = 0
rpm-top['3x3']['top_agents'] = ''
rpm-top['hasVerbalTrue']['avg_run_time'] = 0
rpm-top['hasVerbalFalse']['top_agents'] = ''
```

D. Student-Agent Input List

This input will be a .txt file storing a list of each Student-Agent along with their locations. The student-agent ID will be the GT userid. The format of the file should be GTID, location. for example,

Student-Agent Input List
tfairchild3, ~/Agents/tfairchild3/

E. Student-Agent State Tables

This is generated data that will be repeatedly updated after each problem pass. The state table will be stored as a hash in the same way that the RPM state table is stored. The information will be as follows:

Student-Agent State Tables
student-agent[‘tfairchild3’]['location'] = ‘~/Agents/tfairchild3/’ student-agent[‘tfairchild3’]['executable'] = ‘python’ #could be java too student-agent[‘tfairchild3’]['correct'] = 0 student-agent[‘tfairchild3’][‘wrong’] = 0 student-agent[‘tfairchild3’][‘skipped’] = 0 student-agent[‘tfairchild3’][‘avg_run_time’] = 0 student-agent[‘tfairchild3’][‘auto_skip’] = 0 #always skip this agent if true student-agent[‘tfairchild3’][‘auto_include’] = 1 #always run this agent if true

Decision Engine

Once the input data is read and initialized, control is passed to the decision engine which is responsible for the following:

1. Analyze the state table of the next RPM problem
2. Analyze the state tables for each existing student-agent
3. Decide, based on the state of each student-agent, whether or not to employ a particular agent for the current problem. When the Meta-Agent decides an agent should be included it will add the agent to a list. This list will be used by the processing engine to initiate the student-agent.

Algorithm

for agent in student-agent:
#based on state tables, decide if this agent would be good for processing this RPM
agents_to_run.append(agent)

Processing Engine

The processing engine is responsible for the following:

1. Initiating the desired student agents for the given RPM problem. This can be done one at a time, as described by this algorithm.

```
#for each agent in agents_to_run array
for agent in agents_to_run:
        #instantiate the StudentAgent class
        SA = New StudentAgent(agent)
```

Within the StudentAgent class we must check if the student agent is written in java or python to formulate the proper command:

```
if student-agent[agent][‘executable’] == ‘python’:
  this.cmd = “py “ . student-agent[agent]['location']
else:
  this.cmd = “java” . student-agent[agent]['location']
```

Execute the student agent while keeping track of the start and end times:

```
            SA.start()
            rc, stdout, stderr = SA.go()
            SA.end()
```

2. Consulting student-agent output. This is necessary to get the answer and other valuable metrics. This information can be parsed out from from stdout:

```
run_time[agent][answer] = SA.parse_answer(stdout)
      run_time[agent][run_time] = SA.get_run_time()
```

3. Choosing and submitting the correct answer can be done by counting all of the instances of a particular answer. To do this, we want the Meta-Agent to tally up all the different values of the answers returned:

```
for agent in run_time:
    answer[agent[answer]][count] = answer[agent[answer][count] + 1
```

The Meta-Agent will select the answer with the highest count. There are many ways to do it. One such way in python is accomplished as follows:

```
max(answer.keys(), key=(lambda k: answer[k]))
```

Once selected, the meta-agent will return it's choice of the correct answer. After this, the processing returns to the Preparation Engine where the state tables can be updated with the new metrics and the next problem selected. The Meta-Agent proceeds in this way until all the input problems are processed.

4. Monitoring the system resources and taking action where appropriate can be accomplished using a system library that will allow the Meta-Agent to monitor running processes and system utilization (CPU, memory, disks, network). It should be useful for system monitoring, profiling and limiting process resources and management of running processes. Examples are *Resource*[3] for Python.

Conclusion

This was an interesting question and I would have liked to have written this Meta-Agent as a class project. Maybe for future classes it should be considered. Perhaps projects 1 & 2 could be offered, and then project 3 could be a Meta-Agent. In any case, I think that my design is simple and straightforward.

Bibliography

[1] [2] Ac.els-cdn.com,. 'Meta-Agent Programs'. N.p., 2015. Web. 29 Nov. 2015.

[3] "36.13. Resource — Resource Usage Information¶." *36.13. resource*. Web. 30 Nov. 2015.

https://docs.python.org/2/library/resource.html>