Designing an AI to determine building codes

Joey Schutz ‘18 and Professor James Skon
Kenyon College Summer Science 2017

Abstract

The goal of our project was to develop an Artificial Intelligence agent that queries a user who is seeking to build a structure with relevant questions, and then uses the user’s answers to compile a list of all the building codes that their structure must adhere to. This system had two parts: an ASP program, that had a dataset of knowledge representing building codes; and a Java program, that would store a user’s responses, give the information to the ASP program, and decide what the next best question to ask was based on the ASP response.

We split the tasks into two parts: ASP and Java development, with Ben Weinberg handling the ASP, and me doing the Java side. In order to implement it, we had to create read and write capabilities to and from a knowledge base. With access to users’ answers, we could send that information to the ASP program in a way that ASP could understand it, and then parse ASP’s response in a meaningful way. This required thousands of queries in microseconds. To do this, we used a variation of the A* search algorithm, in order to get the widest search space possible in a such a short amount of time. Java then used that information to decipher the best question to ask next, found the file that contained the question we wanted, and sent it to the web client to present it to the user.

What came before

We already had a system in place for deciding questions. Essentially, every question gave the conditions (called “dependencies”) that would allow that question to “unlock,” making it possible to present that question to a user. This worked, but there were some problems with it.

(1) It was needlessly complex. If an unexpected question was asked, how do you know what caused it? There was no trace of logical decisions, just the end result.

(2) We had very limited control over user experience. We could define basic gates to keep certain questions from getting asked too early, but there was no real way to tailor questions to specific users.

(3) Every question was making decisions for itself...This led to extremely complicated and unpredictable behaviour, but it also led to maintainability problems. As you can see below, some of the logic could be really knotty, featuring many conditions chained together. The possibility of typos was very high, with chances of *finding* the typo dastardly difficult.

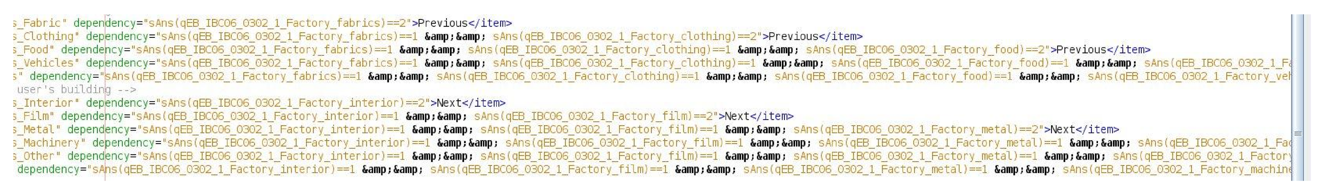


Figure 1: A particularly complicated example of the early “dependencies” system

New Approach

We needed our new AI agent to do a couple things:

1. Streamline the decision process
2. Be able to rank questions based on their relevance to a user
3. Be *extendable*. Building codes change frequently, so it was important to be able to add and subtract from our logic easily.

Our solution involved two parts: representing the codes in ASP (Answer Set Programming language), and a Java client, which could act as a go-between for the user’s answers so far, and where ASP suggested to go from there.

Java-ASP communication

Integrating Java-ASP communication was not too tricky, but required some performance-slashing workarounds. Because there is no way to have the two languages communicate directly with each other, Java had to execute commands in the command line to run the ASP, then read the output. While effective, opening up the command line takes a lot of time, so we had to place limits on how often we did it.

However, ASP had some quirks to it. For instance:

1. We couldn’t directly ask it questions. We had to load up facts, and then it would output all possible answer sets.
2. We needed something dynamic--something we could update as we gained more knowledge about our user’s building.
3. If we hadn’t well-defined anything in the ASP, it would flag it as an error. This made it hard to define rule sets for eventual scenarios (e.g. saying what a Nuclear Reactor would need before ASP knew if we had a Nuclear Reactor).

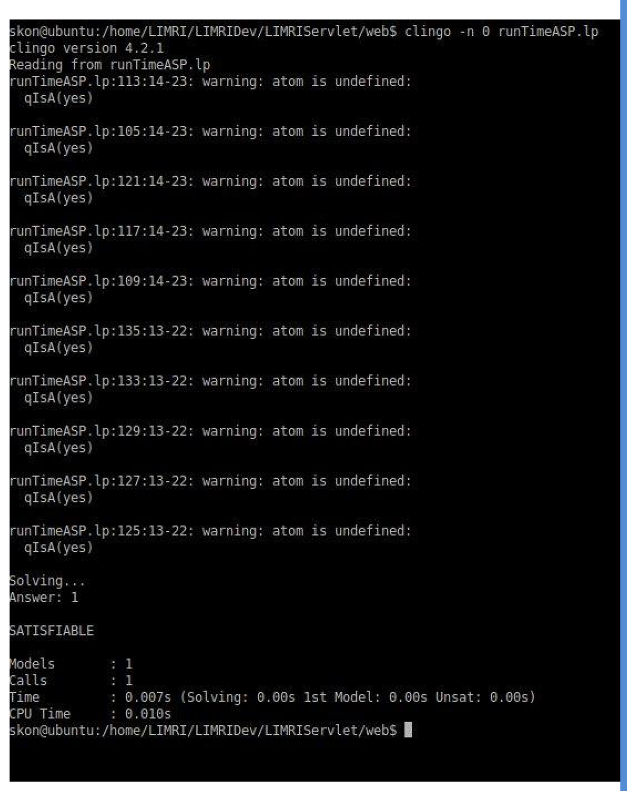


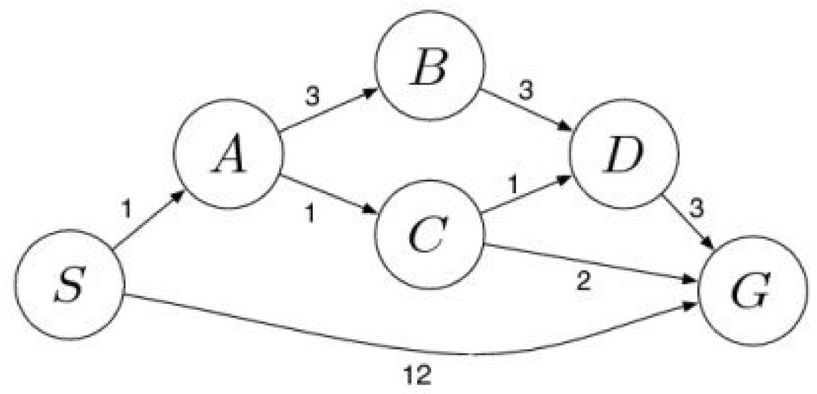
Figure 2: ASP Output (it doesn’t know what “qIsA” is, and so it’s flagging it as an error)

Initial Solution

Given the Java-ASP constraints, we decided to use ASP’s error-flagging to tell us what questions were relevant to ask. If ASP didn’t know what something was, it was because *it didn’t have enough information to know what it was*. We could then pinpoint what questions needed answering.

However, knowing what questions still needed answering was not enough: we needed to know the *best* question that still needed answering. To find that out, we went with an A* search algorithm.

Using A*



State	h1	h2
S	4	3
A	2	2
B	6	5
C	2	1
D	3	2
G	0	0

Figure 3: Search path possibilities

A* is a search algorithm that searches through potential paths, looking at the shortest estimated paths first. For example, take Figure 3. If we start at *S*, and want to end up at *G*, then A* would go point by point, examine the possible paths forward, and take the shortest *total* path available. So, looking at figure 3, our only path forward is S -> A. From A, we have 3 possible paths:

A -> C*, *A -> B, and ***A -> S***

Of course, we’ve already visited *S*, so *A -> S* gets taken off our list. This leaves the following set, with the following *total* distances (counting the distance we’ve already traveled from *S -> A*!):

A -> C (distance of 2), ***A -> B*** (distance of 4).

A -> C is shorter, so we go *S -> A -> C*, and continue from there. Our possible paths are:

S -> A -> B (distance of 4), ***S -> A -> C -> D*** (distance of 3), ***S -> A -> C -> G*** (distance of 4)

S -> A -> C -> D is shortest, so we go there, and so on. A* is highly optimized, which made it a perfect candidate for our Java program, which needed to run *many* possibilities in microseconds.

For our purposes, the “distance” was how many questions we would have to ask before reaching a desired amount of knowledge. To determine this, we ran many hypothetical situations, giving potential answers to questions and seeing what gave us the quickest route to the end. From there, we would ask the initial question that got us there, concluding that that would be the most relevant question to the user.

Ranking Questions

For choosing the most relevant question to ask a user, we used two metrics:

(1) How quickly does it get you to the end?

As demonstrated in our A* search algorithm, we decided that presenting users with the questions that would get them to the end quickest was ultimately the best decider of relevancy.

(2) What are the odds that this question will be relevant to any given user?

There are certain questions (e.g. “Is your Nuclear Reactor next to a brick building?”) that simply need never be posed to the vast majority of users. Even if those questions would get us to the end the quickest, we didn’t want to present an obviously silly question to a user unless we felt sure that we were in a situation where it made sense. To get around this, we added the option to add a “priority” attribute, which could be positive or negative, and which would weigh against “quickness.”

Optimizations

The biggest problem we ran into with A* was time. Our problem required massive search within just milliseconds (because we couldn’t afford large load times between questions), which simply wasn’t possible given how much time it takes to open the command line and run ASP (about 10 milliseconds each time--an eternity in computer time). The end result worked, but the practical constraints of the problem made it unusable.

Luckily, Ben discovered a new way to get ASP to output *every* answer set at once (see Figure 4). With this, we were able to evaluate each question’s “quickness” all at once, without having to open the command line more than once. This eliminated the command line’s sluggish load times from being a problem, but created a new dilemma: ASP’s huge answer sets took way too long (5, 10 seconds) to output to the console. While it was possible for Java to limit the number of answer sets that ASP put out, the answer sets given were in a random order, meaning we were cutting off potentially thousands of answer sets that might have pointed us towards a much better question to ask. Using only one hundred of, often, thousands of answer sets is effectively shooting in the dark--it was not a valid solution. We realized, though, that if we could order the answer sets by size, and *then* output them to the console, then we could feel significantly more confident in looking at only the first 100 answer sets. Unfortunately, the summer ended before we could figure out how to do that with ASP.

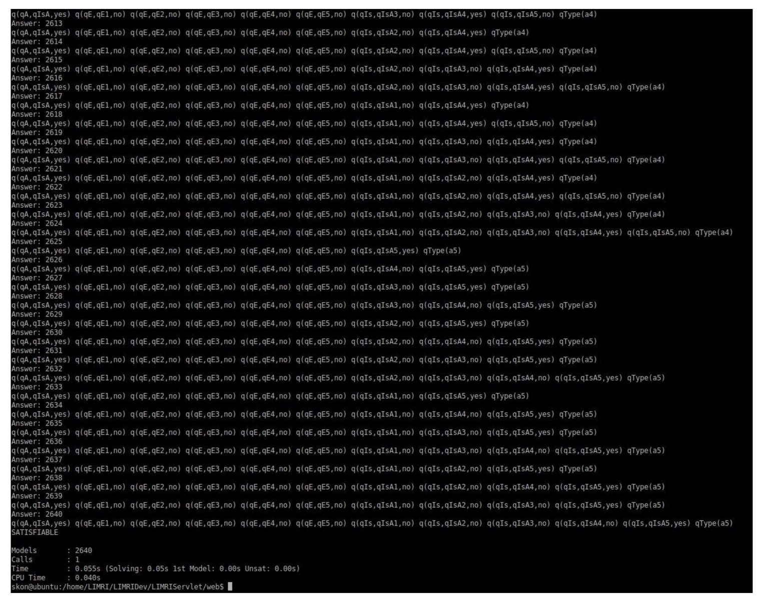


Figure 4: New ASP Output

Acknowledgments

I would like to thank Professor Skon, who gave a bottomless amount of guidance, support, and wisdom; Doug Karl, for his insightful feedback and experience; and the rest of the team that I worked with: Ben, Nicky, and Spalding.

This work was funded by the Kenyon College Summer Science Scholars program.