Designing an AI to Determine Building Codes

Joey Schutz
Kenyon College

James Skon
Kenyon College

Follow this and additional works at: https://digital.kenyon.edu/summerscienceprogram

Recommended Citation
Schutz, Joey and Skon, James, "Designing an AI to Determine Building Codes" (2017). Kenyon Summer Science Scholars Program. Paper 76.
https://digital.kenyon.edu/summerscienceprogram/76

This Article is brought to you for free and open access by the Summer Student Research Scholarship at Digital Kenyon: Research, Scholarship, and Creative Exchange. It has been accepted for inclusion in Kenyon Summer Science Scholars Program by an authorized administrator of Digital Kenyon: Research, Scholarship, and Creative Exchange. For more information, please contact noltj@kenyon.edu.
# 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.

## 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).

## 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.

### 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.

### 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.

## Optimization

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.

## 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.