NOAH'S ARC

Pieter Weemaes, Tomoya Hasegawa, Randall Raymond, Andrew Fai

https://www.youtube.com/watch?v=_A0rIlHa8Mo

05/01/2015

Introduction to Computer Science II [CS51]

Professor: Greg Morrisett

Teaching Fellow Advisor: Nicholas Larus-Stone

Overview

We implemented a Constraint Satisfaction Problem solver that can be applied to any type of CSP. We chose to implement a Sudoku solver and an N-Queens solver using three different algorithms: backtracking, AC-1, and AC-3. Our application defines the cells on a Sudoku board and chess board as variables, takes the constraints of the solutions for the given problem as well as the pre-set values of the given problem, and creates a simple output for the solution. The output also compares the runtime of these algorithms.

Planning

The central things we planned to implemented from the beginning were a CSP solver that used the AC-3 (Arc Consistency Algorithm #3) and the backtracking algorithm and a model for the Sudoku problem environment. Secondary goals included a graphical implementation in which we could show a computer solving a given problem in real-time, cooler problem applications such as a course planner that took into account workload and course requirements, and tests for other algorithms such as A* and AC-1. We successfully implemented all the 'central' things we planned for, but decided not to devote our time to a graphical interface. Instead, we decided that we should implement more algorithms and test our CSP on problems other than Sudoku. While implementing a course planner would be interesting, it would take too much time, and we decided to instead compare the runtimes of different algorithms. We therefore decided to add an N-Queens solver and implement the AC-1 algorithm.

Nicholas, our TF, advised us to explore A*. Since it was not part of our original specification, we devoted more of our time to perfecting backtracking, AC-3, and AC-1, as well as expanding our CSP to solve the N-Queens problem. However, we did partially implement the A* algorithm for the Sudoku solver. Our application only used the past path-cost function, because the future path-cost function (or the "heuristic" function) relied on changing the domains of all variables. This was a feature that we did not wish to incorporate into the infrastructure of our application since the other algorithms did not require this, and we did not want to change such an integral part of the infrastructure just for an additional feature. Regardless, we were able to see the power of the A* algorithm even with only its past-cost function, as we will show later in our runtime results.

Design & Implementation

Starting from scratch was really intimidating, as we didn't know what we would truly need in our modules. We knew that we would need Variables and Constraints but did not exactly how they would work. At first we made constraints operate on a whole list of Variables, and we were able to create a Sudoku problem with 27 different constraints. However, we realized that in order to create the AC-3 solver, we needed binary constraints. So we rewrote our constraints for Sudoku to operate on sets of only two variables at a time and slightly modified our constraint class. We were also very concerned with making our algorithms work with any type of CSP and not just Sudoku. This meant that while the code could have been optimized to run faster on Sudoku, we instead focused on making it robust and compatible with any CSP.

The difference between our algorithms was often marginal. Comparing pure running time, we were able to come up with the following chart of sample runtimes:

SUDOKU			
	Blank	**Easy**	**Hard**
Backtrack	0.076049961	3.561507742	0.178689830
AC-1	0.094282791	3.534189279	0.186945931
AC-3	0.186569434	2.282008923	0.146290037
A*	0.185516794	0.816992254	0.390732471

N-QUEENS			
	4	**8**	**15**
Backtrack	0.000051555	0.001340656	0.035425073
AC1	0.000100613	0.001251938	0.038010485
AC3	0.000097143	0.001413356	0.039480331

We saw some interesting things in our runtimes. Most notable was the difference between the algorithms on the Easy Sudoku board. For this board, normal backtracking was very slow. We believe this has to do with the naive way the algorithm selects a variable to assign. For this specific board, the algorithm seemed to just be unlucky. AC-3 and AC-1 both made slight improvements over simple backtracking, but unfortunately they too had to perform a lot of backtracking. A* in this case was by far the best algorithm. With this board, the algorithm was able to avoid some of the backtracking traps into which the other algorithms had been falling. This board shows the power of the variable selection process. The other interesting thing was that our arc consistency algorithms didn't improve the queens runtimes. This may have to do with the simple nature of the problem, which requires little backtracking. The arc consistency checks, while still reducing domain values, were time-costly and ended up hurting as much if not more than they helped.

Reflection

We all discussed and planned the design and outlined the basic infrastructure of the application. We didn't delegate any particular roles to a single member, but there were several notable contributions. Pieter played a pivotal role in writing the interface which laid the foundation for our code and implementing most of the AC-3 algorithm. Andrew wrote a large part of the backtracking algorithm. Tomoya and Randall wrote a large part of the specification, organized the video, and spearheaded the AC-1 and A* algorithms. At its core though, we all contributed as a group to the base of the work: research, implementation, design, testing, and debugging the algorithms.

One thing we learned while coding in Java was that we missed the powers of functional programming. We often wished to simply map over a list, and purge domains or find constraints that operate on a certain variable. We would then go to the internet for advice on how to do maps, and realized that the function doesn't really exist outside of functional programming languages.

Throughout the process we faced a few surprises. One nice surprise was how easy it was to implement AC-1 after the completion of AC-3. In fact, we did not originally plan to implement AC-1, but after researching it we realized it would be easy to implement using the backbone of AC-3. On the other hand, there were some unpleasant surprises. Most notable was the difficulty in building our foundation from scratch. The infrastructure of our program definitely changed shape from the beginning to the end with regards to our interfaces and classes. However, it ultimately ended up being for the best of the program.

We made some good decisions throughout the process, including rewriting the Constraint class to operate as binary constraints to allow for the implementation for AC-3. However, decisions that did not come about so well included planning for an architecture that didn't fully encompass A*. While A* was something we had thought about implementing, it definitely wasn't at the forefront of our planning, especially since it was a rather minor goal that didn't even exist in our original specification. However, when we actually decided to try implementing A* we realized the infrastructure of our code didn't allow for the full implementation of A*.

With that in mind, if we were to redo this project, we would definitely try to setup the code base differently so that it could allow for a full implementation of A*. However, if we simply had more time, we would try to find another algorithm that already worked well with our given infrastructure. Furthermore, we would definitely look into implementing a course planner.

Advice for Future Students

At the end of the day the most important thing that stuck with us was how critical it was to set goals - not just plan, but have really concrete items that we would finish no matter what. Planning is nice, but having those concrete goals that you strive to achieve is not only what gives you the greatest drive to push on mid-way through the project but also what gives you something tangible by which you can to measure your success. So to those students who will do this project in the future, make sure you identify two to three goals that you have to complete by the end of this project and I guarantee that you'll complete them - and feel great about doing so.

Draft Specification:

Brief Overview

Our team hopes to solve Constraint Satisfaction Problems (CSPs). CSPs can be defined as sets of objects (i.e. a set of numbers) whose overall state must satisfy a set of constraints. Fun examples of problems that can be modeled as CSPs include Sudoku (where an example constraint would be a column of nine numbers must each hold distinct values), the eight queens puzzle (where an example constraint is that no queen can 'fight' another), and more. What we are concerned with however, is finding an algorithm that can efficiently and quickly solve a given CSP. We have decided to explore two such algorithms: the backtracking and AC-3 algorithms. Thus, our major goals for this project are to implement both algorithms and compare them amongst one another in order to prove AC-3's superiority and optimize it in the long run. We want our focus to lie in the implementation of our algorithms, hence the actual problem we will be applying to our algorithms with will be a simple, but rewarding one: Sudoku. Finally, with all this in play, we are striving to implement a graphical interface to show our computer solving a Sudoku puzzle with our optimized algorithm of Noah's Arc.

Prioritized List of Features

- **Core Features**
  - We will implement the AC-3 (Arc Consistency Algorithm #3) as described at (http://en.wikipedia.org/wiki/AC-3_algorithm)
  - We will implement the backtracking algorithm as described at (http://en.wikipedia.org/wiki/Backtracking)
  - We will model "Sudoku" described at (http://en.wikipedia.org/wiki/Sudoku) as a Constraint Satisfaction Problem and use it to run our implemented algorithms.

- **Cool Extensions**
  - We will establish a graphical interface through which to 'show' our optimized algorithm solving a given CSP (in this case, Sudoku)
    - We were unable to get to this extension
  - We will try out other CSPs on our algorithm, starting with a course-planner that takes into account course prerequisites and an estimate for how many hours a student is willing to put into schoolwork for a given semester.
    - We weren't able to use this on course planning but we were able to show our algorithms working on n-Queens.
  - We will try implementing AC-1, which is supposed to be a slower algorithm. Ideally we would compare run-time and test the efficiency of the algorithms.

Tech Specs

The overall design of our project will be easily broken down into separate blocks. That is the main reason we chose to use a language so well known for its object oriented qualities. In order to achieve an end goal of solving CSPs, we will of course need some module that contains the solving algorithm. This is the largest and most complicated piece of our larger puzzle, but we will also be careful to design it in such a way that we can implement the entire program with a different algorithm without changing much code. But the algorithm is only part of the whole system, we also need some way to design different types of CSPs under one interface such that all types of problems can be solved with the same code. Basically, we need an interface that defines a CSP, which contains variables, the domains of the variables, and the constraints that they must hold to. The CSP interface can then be used to create classes such as a "sudoku" class or an "8 queens" class. These classes will have specific methods that give the individual problems a way to communicate with the outside world (hopefully eventually graphically). We will then pass these problems into our algorithm, which will inherit from another generic "algorithm" interface. Whatever algorithm we choose to use at that moment will then execute and spit out the solved problem. Ideally, the algorithms will simply modify and mutate the CSP objects so that whatever methods we declare to display the current state of a CSP object can be used at any point during execution (possibly even live?).

CSP interface
- variables
  - domains
- constraints
- printf

Sudoku CSP
- implements CSP interface
- variables a1-i9
  - domains 1-9
- constraints
  - each row/column/box 1-9
- initialize board
- printf

Algorithm interface
- must be able to take in any CSP

AC3 Algorithm
- implements Algorithm interface

This was actually very similar to our final structure. We ended up having other classes for constraints and variables, but on the whole this is what our final code looks like.

In order to eventually add a graphical way to display results, we must allow whatever print function defined in a CSP object to be accessible outside of the object. This way, we can simply call the printf function whenever we update with our algorithm and hopefully attain a near-live graphical representation of how the algorithm performs.

Next Steps

We will implement Noah's Arc in Java, and develop the application on BlueJ.

We chose to code in Java for several reasons. To begin with, it is an object-oriented language, which seemed suited for implementing AC3. Additionally, three of our members already have significant experience coding in Java, and we figured that we could save time and resources due to a better learning curve. Java is also well-documented, and we think that there are sufficient and relevant libraries at our disposal. Since the Java platform is also supported ubiquitously, it would be easy to run our application in other environments, should we decide that we would like our application to be used widely.

We chose to use BlueJ because we wanted an IDE to unify our development environments. BlueJ is useful because it visualizes objects and classes visually, and it is easy to test and debug code. Several members of our team also have prior experience with BlueJ, and it seemed logical to adopt the IDE that some of us were already using.

Now that we have a general documentation prepared, we will begin by studying arc consistency, and the logic behind the AC3 algorithm. Then, we will decide on more specific designs, and then begin development.

Final Specification:

Meeting with TF

We met with our TF, Nicholas Larus-Stone on 4/15 at Winthrop D-hall.

Nicholas was generally happy with our initial draft specification. He advised us to focus on implementing the algorithm, rather than get caught up in its applications, such as the sudoku solver that we proposed. He recommended that we use a text output for the sudoku solver, and not a graphical interface.

Additionally, Nicholas advised us to implement the A* algorithm as well.

Signatures/Interface

In order to keep our code safe and secure, while still being capable of solving any type of CSP, we must be careful in our interface for creating the problems. Looking at an API interface for a CSP solver (CSP4J) we were able to use some of the same ideas to create our solver. The idea is still simple enough, we will create different problems which are extensions of ProblemGenerator. These problems all contain variables and constraints. A variable object contains a name and an integer array. The constraint object will need to take in variables (right now it only takes in two, but we can implement complex constraints that are subclasses of constraint. The Constraint class also will carry a method for checking if the constraint is fulfilled, and a method for returning the value of a given variable within the constraint at any time. There is also an interface for a solver, and an AbstractSolver class which will hold our algorithm. We will be able to create new instances of Solver, ones which contain each of the algorithms we wish to implement.

What we have right now only contains the core functionality. Essentially, we are creating an API that we can then utilize to create a more interesting program, such as one which compares the speed of two different algorithms. We have also have created a sampleproblem class which contains a main method. This is based off of the Queens class shown in the CSP4J main page, but simply shows how we might actually generate a real CSP and then solve it using all of our previously listed Classes. As of now. We are restricting Variables to only contain an integer domain, but we can expand or interpret the answers given by our program to implement non integer based CSP's such as a meeting or course scheduler.

This program is also highly modularized, in that there are many different facets that can be worked upon at the same time. We can for instance allow some members of the group to work on creating more useful subclasses of Constraint, while other members implement the actual AC3 solver. Now that we have a solid interface, and set of rules that each Class much prescribe to, we can already work on different algorithms that can solve any type of CSP.

Please find our github repository at the clone URL:
https://github.com/pweemaes/Noahs-Arc.git or
https://github.com/pweemaes/Noahs-Arc
and take a look at our interfaces for our interface classes and abstract implentations of Solver and ProblemGenerator (Problem)

Timeline

4/10 - Deadline: Submit first draft documentation

- Setup version control (GitHub)
- Setup development environment (BlueJ)
- Write modules/interface
- Familiarize ourselves with the A* algorithm

4/17 - Deadline: Submit final specification

- Familiarize with and learn tools to be used (i.e. Java programming language)
- Finish implementing the backtracking algorithm
- Test functionality of backtracking algorithm
- Finish implementing AC-3 algorithm

4/24 - Deadline: Functionality Checkpoint

- Finish implementing AC1 algorithm
- Finish implementing A* algorithm
- Implement Sudoku Solver
- Film Demo Video

5/1 - Deadline: Submit project

Progress Report

- Implemented skeleton code (interfaces, modules)
- Setup source version control with GitHub (using desktop client)
- Setup development environment (BlueJ)