What I Learned in Heuristics Class

It's been over a year since I took Dennis Shasha's Heuristic Problem Solving class at NYU. Right when the course ended I wanted to write up for myself what I thought I had learned, but Dennis suggested that I wait a while, to separate what was fleeting from what would really soak in.

So here's what has stuck with me. I really did learn these things, in that I did not start out thinking these things but discovered their value in doing the puzzles and competitions. None of these ideas is original, and some were inspired by watching (i.e. copying) what my classmates did. I realize that some of this might sound preachy, so when I say "you" just imagine that I'm talking to my old self.

First solve it, then solve it well.

Is there a trivial solution to the problem? Code that first.

On at least two occasions the candy went unclaimed because no one's solution worked. And in both cases there was a trivial solution. By trivial I mean a solution that is easy to come up with, is clearly bad, but meets the problem criteria, for example dropping a muncher on every node in Nanomunchers.

For one of the early competitions I dove right in to implementing a grand and expansive algorithm I had devised. On competition day I wasn't even close to getting it working. Many students didn't have a working solution that day either and the competition was won by a simple greedy algorithm.

I realize writing trivial solutions isn't the point of the class. But a trivial solution is better than no solution, and is often a good starting point.

Model the problem.

Start by modeling the problem. Write code that will evaluate potential solutions with respect to the actual puzzle specification.

Normally this is the job of the architecture team, and by the time you start coding there may be a server already that does this work for you. It's tempting to save time and "reuse" code by using the server to evaluate your solutions. *Resist the temptation.*

Why? The following are all things that actually came up:

1. The server code may be wrong.
2. The server code may get "updated" and stop working, while you're coding.
3. The server may do annoying things like try to start a GUI when it's invoked.
4. You can learn a lot about the problem by having to model it yourself - especially things you thought you understood.
5. IPC/network communication is much slower than code within your thread, which can severely limit the number of solutions you can try out in a given amount of time.
6. Often there is additional information that is or can be computed while evaluating a solution, and that information can show you exactly how to improve the solution. If you write the evaluation code yourself it is easy to keep that information around and make use of it.

The first three are good enough reasons, but the last three really stick out in my mind.

If you can model the problem, you're halfway to a solution.

Write a solution class.

Often the form of the required solution to a puzzle is some fairly basic type, like a string or an array of integers. It's tempting in the name of performance, simplicity, and rapid coding to use these basic types internally. Don't give in to this form of Primitive Obsession. Encapsulate your solution in a class right away, even if it seems silly, and access your solution data only through that class.

The first reason is that it's common to discover after a while that you really want to keep around some additional information with your solution. For example it's frequently useful to store your evaluation of the solution along with the solution. It's trivial to change the code to do this if it's already a class. The same goes for wanting additional behavior out of your solution class, for example a custom toString method for logging or debugging.

The second reason is that it's common to want to change the solution data structure itself. int[] may have seemed like a natural choice at the time but now LinkedHashSet<Double> would work much better. This change is much easier if your class is already encapsulated.

This is really OO Programming 101, but that didn't keep me from learning this one the hard way. The first few times I used a primitive solution type I found myself furiously changing code (in class) all over the place when I realized I needed to handle something differently, and more than once broke the code in the process. After I rediscovered this design principle I found it easy to change my algorithms as I came up with new ideas, and even to try out several ideas in parallel.

Heuristic Problem Solving in Three Easy Steps

I'm joking of course. But I found the following steps are often enough to go from a trivial solution to a competitive solution.

1. Model the problem.
2. Write a solution class.
3. Run a stock search algorithm and return the best solution you find in the time you have.

You've modeled the problem so you have a way to evaluate solutions quickly. You've got an encapsulated solution class that you can store and pass around. Now it's time to search the solution space:

- The most naive thing you can do is generate random solutions, evaluate each one, and return the best. This actually performs decently in practice, especially when the problem is so hard that no one comes up with anything better.
- If the problem lends itself to some kind of greedy or hill-climbing approach, you can easily combine the two, performing greedy search from randomly chosen starting points. Several students did very well with basically this setup.
- This setup is the ideal framework for applying an evolutionary algorithm or some of the other optimization techniques discussed in class.

For some puzzles it is possible to just do these steps, throw on a random search, and be done. That's not why I'm presenting this though. Rather it's something I found worked very well and wish I had found sooner, because it actually makes a great starting point for getting to what I think is the real work of the class, which is coming up with novel ways to do the search.

Logging is ok. Visualization is better.

It's common to output information to a log file or the console as the program runs. Logging is usually (but not always) better than nothing.

Visualization is much better though. If you can produce output that gives a picture of what is going on inside the program, it can be invaluable for debugging and spotting ways to improve the algorithm.

For example, consider the following excerpts from a hypothetical log file of a program for an asset allocation game. First, normal logging:

```
Asset ID, Price, # Shares
...
131, 14.2, 361.25
132, 61.9, 109.33
```

```
133, 0.72, 24607.0
```

Now the same log file but with a little graph on each line visually depicting the size of the allocation. Each asterisk is 100 shares.

```
Asset ID, Price, # Shares
...
131, 14.2, 361.25 ***
132, 61.9, 109.33 *
133, 0.72, 24607.0 *************************************************
********************************************************************
********************************************************************
*************************************************************
```

The second file takes one more minute of coding. If I'm skimming through a log file of 10,000 lines and I'm trying to see whether my allocations are coming out balanced or unbalanced, which log file will make that easier to see?

We can process a huge amount of visual information very quickly. There were several occasions where my crude ASCII graphs helped me spot problems that would either have gone unnoticed or would have taken a long time in the debugger to find.

Speed doesn't matter. Algorithms do.

This was the biggest surprise for me in taking this class. I came into the class thinking we were going to become expert at all kinds of optimizations. I was ready to dust off my copy of Stroustrup and code C++ again, which if you know me is a big deal. The computer for running competitions was a dual processor machine so I was getting ready to write multithreaded code (in C++ no less). At the very least I thought I'd have to do some fancy stuff to keep track of elapsed time inside my programs so that I could squeeze the most out of the allotted two minutes.

None of it mattered.

As far as I can recall, speed was never an issue. No one did better by virtue of using highly optimized compiled code versus an interpreted script. After the first competition I don't think anyone even bothered using both processors. Some people did go over the time limit, but no one did well simply because they shaved an extra few seconds off of their program. In most cases the time limit wasn't even an issue: programs either finished well under two minutes, or had no chance of completing in anything like two minutes. Personally I found that most of my programs topped out in 30 seconds to a minute, and more time than that led to no significant improvement. Even in two-player adversarial search games, where it would seem that a slight speed advantage would make all the difference, no such advantage appeared.

What did matter was the choice of algorithms, data structures and heuristics. When people chose these wisely (or even just chose one of these wisely) they did well.

For the kinds of problems in this class, it seems that *how* you search the solution space matters much more than *how quickly* you search it.

Tips and Friends

Things I discovered or re-learned that don't have to do with heuristic problem solving directly.

Subversion is your friend.

From my programming career I knew the value of source control, but until fellow student Hunter Freyer showed me I didn't know how easy it is to set up a personal Subversion repository on the NYU system. The instructions are at http://cims.nyu.edu/systems/resources/subversion/index.html. I think every CS student should have one. A nice thing about Subversion is that, unlike some systems, it's pretty usable even when you're offline.

I use mine for just about everything I do now. It can be a lifesaver and it beats copying and stashing backup files with confusing version names all over the place.

A side benefit is that when you want to run your latest code on the server in class, you can just sync, build, and go.

Configuration files are your friend.

There are usually all kinds of extraneous parameters in your program to deal with, either related to the competition itself (e.g. port number of server) or to the algorithm you wrote (e.g. search cutoff depth). Sure you can hardcode them and then change the code each time (and recompile if your language requires it). But that's not so slick, especially in front of the class. And sure you can pass them as extra parameters, but do you really want to pass 5 parameters to your program each time (when you only need to change 1) and try to remember the correct order?

It's easy to write a text configuration file with one key-value pair per line. Now you can have all the named parameters you want and only fiddle with the ones that matter. This saved me some headaches during the class competitions, when apparently I was most prone to screw up my parameter order.

A nice thing about this is that you only have to write the code that reads configuration files once and then you can reuse it for every program.

NOTE: Some things really should be passed on the command line, like the puzzle parameters or input file name, because they will be unknown until the competition. I'm talking about those things that I am tempted to hardcode because I don't want to pass an additional argument, but that I might want to change during the competition.

toString is your friend.

Java-specific. It takes just a minute or two to write a little toString method that indicates the state of whatever class you've written. And then it shows up nicely and compactly in the debugger. I find that debugging my code is an order of magnitude faster when I can see the contents of my objects without having to drill into each field to piece together the state. Even better, I can have a Collection of my objects and the debugger will show me the contents of the array or whatever in one go.

java.util.BitSet is your friend.

Java-specific. BitSet is a neat little class for storing and passing around a vector of boolean bits. It takes up much less space than a boolean array and provides methods that would normally require writing a for loop.

I started putting BitSets everywhere once I discovered this class.

How to do two-player search.

Some of the competitions are two-player adversarial search with perfect information. I found that my head would hurt trying to reason about it ("If I know that you know that I know that this move will make me lose…") until I found a basic template that simplifies it, at least to my aching head. I was able to use this pattern a couple of times in the class:

```
boolean moveMakesMoverWin(move, currentState):
  if isCached(move, currentState) return cached value
  if isIllegal(move, currentState) return false
  tmpState = applyMove(move, currentState)
  for m in possibleMoves(tmpState):
    if moveMakesMoverWin(m, tmpState):
      // mover is now the other guy
      return false and update cache
  return true and update cache
```

If the search tree is too big to search in time you'll have to have a cutoff based on depth or some other criterion. Then moveMakesMoverWin has to return "true", "false", or "don't know", and then you have to guess which move to make in the case of "don't know". Better yet you could introduce a heuristic and return a number in [0, 1], with 0 = definitely false, 1 = definitely true, and values in between giving a heuristic guess as to the probability of true or false.

What I Didn't Learn

How to finish things early so that I'm not rushing at the last minute.

I managed to progress from finishing coding just before my turn in class to finishing coding just before class. That's about it.

All those cool programming languages I've always wanted to try.

I thought this class would be a great chance to try out new programming languages. I was planning to use a different language every week, starting with K.

No dice. The puzzles are hard enough without the challenge of doing them in an unfamiliar language. For me the best way to learn a new language is by doing things in it that I already know how to do.

To be fair, many people in the class did actually try out new languages, which I think is amazing.

Addendum

As I said at the top, I wanted to write this piece at the end of the class a year ago but Dennis asked me to hold off. Well as an experiment, a year ago I quickly jotted down the things I wanted to say at the time and stashed it away. I haven't looked at it again until now, after writing all of the above.

Looking over the list, I see that I had much *more* to say now than I did then. I'm not sure how that works. Maybe I've been reading too many academic papers. Anyway, there were just two things I had on my list then that I didn't have now. I can't recall what I meant by one of them, and the other one I will describe only briefly since it didn't stand the test of time.

Sometimes things don't matter.

Some of the puzzles have important seeming details: the squaring of the overlap in Wordsnakes, the order of the directions the Nanomunchers move, the special color light in MicroArray. I spent a lot of time thinking about these things and in the end they made no difference. Solutions that tried to take advantage of these features didn't fare any better than those that didn't.

page revision: 58, last edited: 30 Jan 2009, 09:26 EST (6 minutes ago)