

How do we know when we're winning?

Jake Beal
Genesis Workshop '03

What's this about? Well, one of the hardest things in AI is to figure out what you're trying to do. There are so many good projects out there that prove interesting and difficult points that have little to do with building things that are smart like people. The stuff I want to say here is about more than just AI, but I think it applies particularly here.

Four Heuristics

- What is the problem of which this is a subproblem?
- How will you know when you have succeeded?
- Characterize your failures.
- Build only what you can engineer with.

Let me start by giving you the end up front. The big message I want you to take out of this talk are these four heuristics. I'll talk a lot about what they mean later. I've learned about the first two bloodily at the hands of Hal and Gerry. The latter two are things I'm never satisfied unless a system fulfills.

I'm also going to talk about some desiderata more specific to the Human Intelligence Enterprise and our work here in the Genesis Group too. But these are the big four.

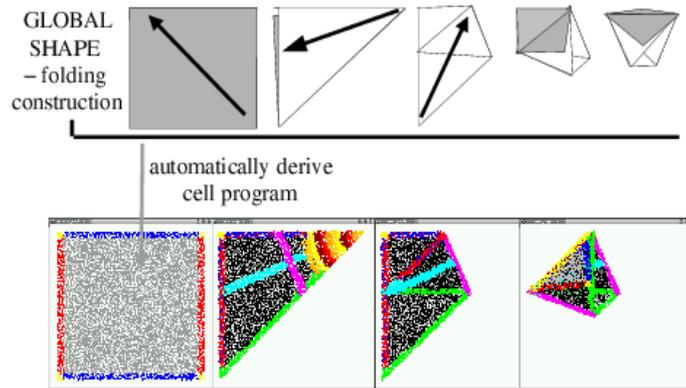
Disclaimer

- Take wisdom gleaned from Winston, Sussman, Abelson, Minsky, Moses, Knight, Brooks, etc...
- ... place in Bad Idea Sausage-Maker and crank

RANT WARNING!

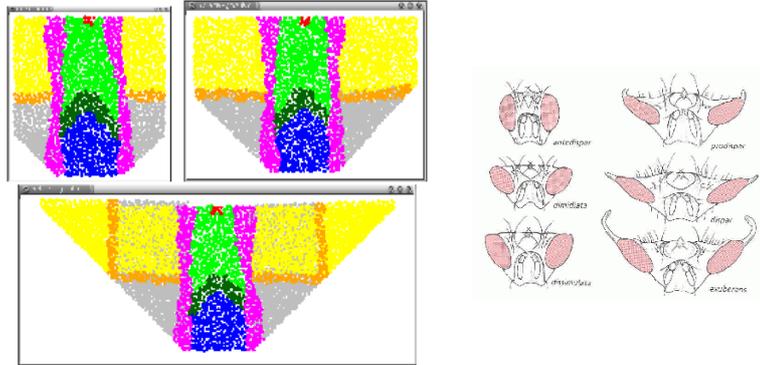
This is, frankly, a screed I've been wanting to give for a while now. It encompasses a bunch of ideas which I try to apply to my own projects, and which I inflict on my students in 6.034 Intensive. I think I've learned this stuff sitting at the feet of elders during my apprenticeship, but they might well disagree with much of what I have to say. It also bears mentioning that most of what will follow is highly biased and prejudiced. But I want to say it to you anyway.

Nagpal's Origami Language



Now I'll introduce a favorite example of mine --- Radhika Nagpal's amorphous computing work on self-organizing origami designs. In brief, her work takes an origami-language design and compiles it into locally executable code that forms a shape on the global network.

Sweet Origami Engineering



Check out the robustness on that thing!

It's also a really slick piece of engineering. Look what she got for free: when you change the boundary conditions, the program doesn't break --- it distorts. And it turns out to have some very interesting biological analogies. There are two parameters setting the boundary conditions of the three origami designs and two genes varying between the six fruitfly heads. Powerful stuff! No wonder Harvard snapped her right up.

Why is this cool?

“What is the problem of which this is a subproblem?”

Shape Formation on an Amorphous Computer

- Robust execution on unreliable parts
- Flexible code
- Principles of morphogenesis
- Biological Computing
- Self-Organizing Systems

Why is this cool? Or to put it another way, “What is the problem of which this is a subproblem?”

One way to answer that is with what it bears on in the larger world. So here we've got something that takes origami designs and self-organizes parts of a non-existent machine to draw them. What relevance can that possibly have?

A lot, actually... it stands at the crossroads of many disciplines. So does Genesis.

The Power of Languages

- What did Nagpal actually build?
 - A program that can fold a cup?
Cup is just a demo
 - A program that can do origami?
Origami is just the representation
 - A compiler for global→local→global shape language!

“If you give a system a program, it'll compute for a day. If you give a system a language, it'll compute until it's obsolete.”

But there's something more here. Not only is it relevant to many possible fields, Nagpal's system does a lot of different things itself. That's the other half of “What is the problem of which this is a subproblem?”

It's not a program to fold a paper cup. Nor is it a program to do origami. It's a language that does a global to local to global transformation on shapes. And that's powerful. That's sweet engineering.

Engineering you can Engineer with

- Nice systems provide you with new primitives
- More than modularity: **Abstraction**
- A well-engineered system is like an onion
 - Beauty is a uniform interface protocol

There are some systems which just feel lovely, powerful to me. This is one of them. I'm not sure exactly what to describe it as. It's something akin to Marr's Type I vs Type II categorization.

What I think today is that the systems I really like are those systems that are easy to engineer with. In other words, systems you can use as primitives in a higher level of abstraction.

Original TTL

Inside the stoplight control...

... is an MPU

Inside the MPU...

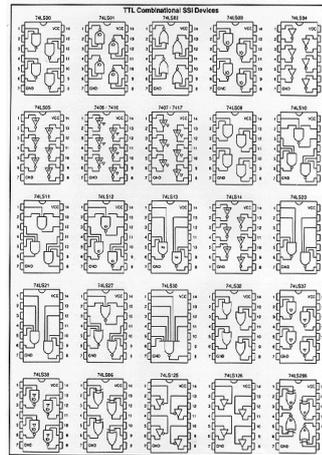
... is a counter

Inside the counter...

... is an adder

Inside the adder...

... are AND, OR, NOT



It's turtles all the way down!

Another favorite: the old TTL logic family. I can engineer at any level without thinking about the other levels. And better yet: I can explain exactly why the system does whatever it does at every level.

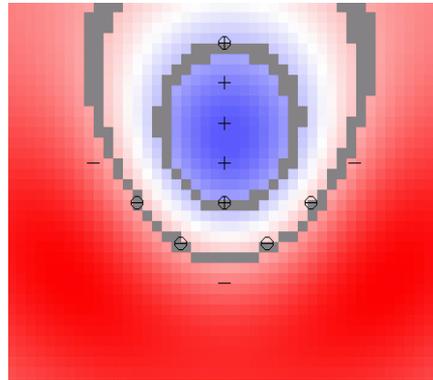
Poor Engineering creates Free Will

“Free Will is what what we call anything that cannot be explained by either randomness or determinism”
-Marvin Minsky

- Human intelligence is probably a design mess
- **Genesis Axiom:** Human-Like Intelligence is susceptible to clean engineering

The title of this slide is strictly tongue in cheek. What I mean is that we AI researchers have a tendency to create systems which appear cooler than they are because they are unanalyzeable. It's the exact inverse of the calculus system where you take the hood off and lose the marvel.

Unclean Engineering: SVMs



Can you characterize its composability?

I'm going to be unfair to SVMs here. They're actually pretty cool, and do a good job at binary decision making. The point, though, is that SVMs, by themselves, don't do us any good. We need to know how they can be composed with themselves and other primitives that we might use them with. Will they amplify or damp errors? What happens when we hook up three in a row? Why is the “road” widest here, and what does that mean?

Maybe we can answer those questions for SVMs and maybe we can't. The point is we have to answer them in order to use SVMs as primitives that can be abstracted away out of our calculations at a higher level.

Build only what you can engineer with

“You can't learn anything you don't almost already know” -PHW

“You can't build a system to learn something you don't know how to build.” -GJS

- AI is hard and complex
 - Scientist debugging cycles
 - Eventually, self-debugging!

This can be a GUI problem!

That brings me to my second heuristic: build only what you can engineer with. The two quotes here are about why that's important. We're attacking one of the hardest problems in the world here --- we'd better plan for a long effort involving a lot of debugging. Moreover, human-like intelligence is self-programming.

Do **you** understand what you've built?

- Talk to non-technical people about your work
 - Is your work interesting?
 - Do you understand it?
- Edit out the equations
- Simple questions
- Admit you don't know things

It's often hardest to understand what we're close to. If you can't explain what you're doing in coherent English sentences, you don't understand it. Teaching somebody else forces you to understand the material much better --- and will often bring you surprising perspectives on it.

You might discover what you're doing isn't as cool as you thought it was. Don't just throw it away: ask, "Why isn't this interesting?" Then you can throw it away and build an interesting replacement.

Not knowing something is a good start for a project. Another is what pisses you off (GJS).

The Scientific Method

- 1) Observation and description of a phenomenon or group of phenomena.
- 2) Formulation of an hypothesis to explain the phenomena. In physics, the hypothesis often takes the form of a causal mechanism or a mathematical relation.
- 3) Use of the hypothesis to predict the existence of other phenomena, or to predict quantitatively the results of new observations.
- 4) Performance of experimental tests of the predictions by several independent experimenters and properly performed experiments.

Remember the scientific method you were taught in school? It's applied surprisingly rarely here in artificial intelligence. We get so caught up in our mechanisms and our high ideals, we don't actually bother making testable hypotheses. Instead, consider my rather cynical version of the typical AI scientific method.

The Scientific Method: AI Version

- 1) Observation of a computational technique
- 2) Hypothesize that applying the technique to a problem, tweaking the technique, or combining two techniques will “produce interesting results”
- 3) Predict that “performance will be improved significantly”
- 4) Nobody will ever duplicate your experiment.

First, you pick something to do that hasn't been done before. Popular choices are randomly crossbreeding two existing techniques --- remember the Dilbert cartoon where he makes a “Skunkapotamus”? Now all you need to do is run it and discover that your numbers have improved.

No independent verification of your result will ever be done because your result is highly dependent on your particular code, and there's not much of a hypothesis there to verify.

It's kinda like alchemy.

Make a testable hypothesis!

“How will you know when your system is working?”

- Predict exact responses
 - Good place for demos
- Difficult to predict = Bad Engineerability
- Verify your prediction
 - Formal proof
 - Rigorous experiment

I've probably harped on this enough by now. But if you've got a problem of which this is a subproblem, then you ought to be able to do this. This is also a good place to get your demo tie-ins, either as the example you work out your prediction on becomes your demo or vice-versa.

If predicting is hard, go back a step!

Double points for formal proofs, but don't get physics envy.

Characterize Your Failures

- Under what conditions will your system crash?
- Under what conditions will there be wrong answers?
- What classes of failures are there?
- Why exactly does it fail?
“Because 0.3 is less than 0.5” isn't an answer
- When does your wrong answer matter?

We make students reason about the ragged edge of failure of systems in order to teach them how they work. Why should research be any different? You are grossly irresponsible if you report that your system has an 83% success rate but don't say what the failures are.

Computer Science likes perfection and allows functions to have “undefined” behavior when conditions aren't met. That's a bad idea for an intelligent architecture.

Vive La Revolution!

- Elegance is a priority
- Avoid “Skunkapotamus” incrementalism
- Wrestling with the exponential

Now, we've also got some extra responsibilities here in the Genesis Group, 'cuz we're in AI and we're the kool kidz. We're working on 20 year plans (that will produce theses in 6 months).

Contributions

- Four heuristics
 - What is the problem of which this is a subproblem?
 - How will you know when you have succeeded?
 - Characterize your failures.
 - Build only what you can engineer with.
- Lotsa hot air

Thank you to many mentors

That's about it. Ignore the ranting and raving if you want, but keep the heuristics and, most importantly, always ask the questions of each other.