GP As If You Meant It: Real and Imaginary User Experience

William A. Tozier

Abstract (to be written last)

Key words: keywords to your chapter, these words should also be indexed

1 Why

More than a decade ago, Rick Riolo, Bill Worzel and I were working on a consulting project together that involved evolutionary algorithms and genetic programming. As we were chatting one day, Bill asked Rick what he'd most like to see as part of the research program of GP "moving forward". Rick's answer informs this contribution, as well as much of my professional work with GP in the years since.

As I recall, Rick answered that he'd like to better understand the "symptoms" we often see when we run an evolutionary search process: premature convergence, failure to improve, catastrophic lack of diversity, and the sense when we look at results or ongoing runs that some particular choices we've made setting the many parameters *just aren't quite right*. There are numerous well-written papers suggesting ways around local minima, proposing reasonable and exotic search operators, and running "horse races" between variant algorithms on benchmark problems. But years later, I take Rick's challenge not only to mean that it might be useful to have a benchmarking catalog which lists conditions in which search operator X acting under contingency Y tends to produce outcome Z, but also that *the things we identify as symptoms themselves* are poorly understood, to the point that we think of them as "subjective".

Somewhere during that same project, I remember a Project Manager telling the story that when domain expert customers were shown a collection of Pareto-optimal solutions to the problem being explored, they were confused. "We don't want a choice, we want *the best*." This came after months of analysis, interviews, and a

collective agreement that the problem was fundamentally multi-objective. But when the decision-makers were faced with the unquestionably successful results they had collectively specified, *those results weren't right enough*.

In this contribution we must face the fact that such a response is inevitable: not just from silly lay "customers" in an application project, but from ourselves; whether the project is "theoretical" or "practical"; whether it is "small" or "large". Projects resist our initial plans.

The "field" of GP has grown quite a bit in the intervening 15 years since these two anecdotes. All of us who follow it closely have played a role in the expanding front of new methodologies and techniques—the extraordinary ones we describe in our little workshop, and also the newsworthy and admirable public successes.

But this acceleration of practical successes has been accompanied by a dilution of the scope of the theoretical warrants we use to justify them: Most examples of "practical success" involve unique domain-specific quirks, each implementation makes numerous contingent (and sometimes arbitrary) design and architectural choices, and even the ontological foundations of "individual", "fitness", "behavior" and "population" start to get sloppy around the edges whenever we actually *look* at What People Actually Do to Solve Problems With GP.

While this can be disappointing for those of us who follow the progress of theoretical work closely, it has more serious consequences. The lack of interest in GP among statisticians, planners, designers and mathematical programming aficionados is, I argue, not just a matter of disciplinary border wars. I suspect rather that skepticism arises from our own inability to say what's *happening* in any particular GP run—*even when we look*. Nor for that matter can we be very helpful in explaining *why*, or what to do in response to any given contingency. We have learned that *more CPU time* can sometimes get us past a tricky stage of "not working", but any project that lives very far from the bounds of the last century's version of "genetic programming" is unexplored territory.

This is not the fault of GP as a field, nor of our theoreticians and engineers. Rather I'll argue that a deep philosophical misconception presides over the work, both within and outside our field. This misconception undermines our understanding of what GP is *doing*, of the way it unfolds in theory and in practice, and even what it's *for*.

I will make my case here in the form of an exercise, or *kata*. This is not intended to be a "thought-experiment", nor is it a suggestion for a new way of working. Rather it is a formal exercise to be undertaken by those of us already working closely with GP systems. In the immediate case, my intent is to surface the three problems I've identified above:

1. What GP *does*
2. How GP *unfolds*
3. What GP is *for*

The rules for this exercise, "Doing GP as if you meant it", will feel the most artificial and restrictive to those of us with the most experience. Like the martial arts exercises from which it is ultimately inspired, it isn't intended to be simple or

even pleasant for the participants. But because it emphasizes the interaction between the GP system and the user, it not only surfaces the philosophical mistake I've mentioned, but immediately suggests tools with which we can address Rick's decade-old wish.

2 How we treat GP, and how it treats us in turn

Genetic Programming[1] embodies a very particular *stance* towards the scientific and engineering work of modeling, design, analysis and optimization. I increasingly suspect the resistance we've all recounted towards GP from our prospective technical and lay audience has little to do with our technical results, but rather arises from uncertainty among that audience with GP's very particular "way of working".

There is a tacit assumption, even among GP theorists and practitioners, that science and engineering are only "rigorous" when they proceed through a sequence of ordered phases from planning to implementation. The "scientific method" is most often represented something along the lines of

> 1. conceptualization; 2. planning; 3. design; 4. architecture; 5. implementation; 6. testing; 7. debugging

I am sure very few scientists or engineers of my acquaintance would admit any *real* project (in history) has ever followed this narrative arc in a literal sense. But that story nonetheless informs and constrains much of our work lives, from fund-raising to publishing reports:

> Because of the body of published work, an insight was had. The insight was framed as a formal hypothesis. The hypothesis (and current Best Statistical Practices) suggested an experimental design, one familiar and obvious to any in Our Discipline. The experimental design was undertaken, the data were collected, the hypothesis duly tested, and now we can be confident of its veracity because… well, you just heard me say "Best Statistical Practices", right?

Under trivial term substitutions—"cost–benefit analysis" and "requirements document" for "hypotheses" and "experimental design", for example—the same narrative can be used to describe almost any institutional project management and public policy planning process as well. The flow in every case is essentially from *vision* to *plan*, *plan* to *implementation*, *implementation* to *verification*, and *verification* to *validation*.

Of course, nobody "really believes" this narrative who has ever done any of the work. It is a matter for another day to draw parallels with the social construction of

[1] And not just Genetic Programming as such, but also the broader discipline to which I claim it belongs and which is not obliged to be either "genetic" or "programming". I prefer to call this looser collection of practices "generative processing", and will also abbreviate it "GP"; assume I mean the latter in every case.

religious belief.[2] There have been numerous philosophical challenges to this artificial narrative of course, from Peirce and Dewey nearly a century ago, to Kuhn and Lakatos in the 1970s, and many more to be found in the Table of Contents in any Philosophy of Science text.

One in particular is my focus today: Andrew Pickering.

2.1 Pickering's "Mangle of Practice"

Andrew Pickering's *Mangle of Practice* is a decade old, but surprisingly little-known outside the field of Science Studies. His approach is especially noteworthy here because I find it so close to our actual experience using GP. Indeed, most colleagues who hear it for the first time utter an inevitable "didn't we already know this?" I think the distinction Pickering's approach is able to highlight is exactly the problematic one in our work: between the illusory (but publishable) narrative of "scientific method", and the realized experience we have of *performing science*. At the cost of glossing a lot of his well-considered structure, let me summarize.

First, the act of "doing science" is *at no point whatsoever* to be understood as one in which an isolated "researcher" works in an objective and static field of "externalities" and "facts". Rather, research begins only when the researcher undertakes to *makes some artifact*: a block of code, a technical instrument, an equation, a sketch, a maquette or even a thoughtful conversation at a conference. Everything before the researcher begins to make these things is a matter of building "vision" (my term); research proper only occurs when work is done in the real world. I'll call this work product *the thing made*, and keep in mind that it is a proxy for the vague collection we might otherwise call "the project".

We see Pickering's model moving quickly away from more traditional views of science when we realize he has given the inanimate *thing made* an agency of its own. In any real project, we perceive the *thing made* resisting—whether we imagine that it resists "in itself", or as an agent of externalities that impinge on the work in progress to make our driving vision differ from reality. The *thing made* comes to represent, in other words, the facts of the actual world, the cultural assumptions and norms of one's discipline, the raw materials and toolkit available to a practitioner, and so forth. "Resistance" here is not merely a problem with a software bug or lost raw materials, but includes one's sense *on seeing it* that something's not quite right. The *thing made*'s resistance, as perceived by the researcher, leads her to react in turn.

Consider the language we use in the face of this resistance: it "feels wrong"; it "points something out"; it "wants to do X instead of Y"; it's "doing something too complicated for me to understand right now".

And—assuming science, engineering, art or any other creative process is indeed what's being done—the researcher *changes in response to this resistance*. That vi-

[2] Paul Veyne's excellent *Did the Greeks Believe in Their Myths?* might be an interesting starting point, though.

sion changes, the plan adapts, or in some other way the *thing made* causes a response in the state of the researcher herself. Pickering's Mangle is this emergent dance of inanimate agency: the researcher starting to follow a vision by making (or altering) a *thing*, and the *thing made* in turn acting as a channel for the world itself to steer the researcher in another direction.

Pickering's "mangle"[3] is this emergent dance of agency, undertaken between the researcher and the project: the researcher makes, the *thing made* resists, the researcher is influenced and redirected, *accommodating* the resistance. It may seem glib to say that "no plan survives contact with the enemy," but Pickering's narrative of creative work emphasizes the fact that no *revised* plan survives unchanged, either. In the traditional narrative, we elide the work as it unfolded and re-frame it as a sort of idealized, apersonal Platonic truth: we use the passive voice, we hide the missteps and confusion, we paint a story moving from vision to plan to success.

In a GP setting, the many modes of resistance we perceive are the very "symptoms" Rick mused about. Even though we as researchers know we've written all the code and set all the parameters, we're nonetheless willing to speak of a GP run "doing" things, as opposed to merely unfolding according to our plan. A GP system does not "resist" by, for example, "having the wrong population size". Rather it resists by *causing concern or dissatisfaction in the user*, which in turn sparks in that user a practical (if only explanatory) response, which leads them to change the *thing made*.

2.2 GP as "mangle-ish practice"

The broader field of machine learning seems to take a much more traditional stance towards its subject matter: machine learning frameworks (excepting GP) are each discrete tools aimed at producing standardized and reproducible results to particular statistical questions. The result of training a neural network or even a random forest on a given data set is not expected to be a *surprise* in any real sense, but rather the reliable and robust end-product of applying numerical optimization to a well-specified mathematical programming problem. Whether one describes these machine learning processes as "minimizing out-of-sample error" or "maximizing

[3] The word "mangle" he has chosen is itself interesting and insightful:

> . . . I find "mangle" a convenient and suggestive shorthand for the dialectic because, for me, it conjures up the image of the unpredictable transformations worked upon whatever gets fed into the old-fashioned device of the same name used to squeeze the water out of the washing. It draws attention to the emergently intertwined delineation and reconfiguration of machinic captures and human intentions, practices, and so on. The word "mangle" can also be used appropriately in other ways, for instance as a verb. Thus I say that the contours of material and social agency are mangled in practice, meaning emergently transformed and delineated in the dialectic of resistance and accommodation. . . .

information gain", the supposed strength of most machine learning approaches is the *unsurprising* nature of their use cases and outputs.

On the other hand, GP has the capacity to *tell us stories*—even in the relatively "simple" domain of symbolic regression. The space under consideration by GP is not merely a vector of numerical constants or a binary mask over a suite of input variables, but the *power-set* of inputs, functions over inputs, and higher-order functions over those. We who work in the field can be glib about the "open-endedness" of GP systems, but that open-endedness puts them at odds with their supposed relatives in machine learning. While GP *can* be used to explore arbitrarily close to some paradigmatic model, its more typical use case leads to the production of *unexpected* insights—to the degree that a number of us feel justified in treating it as a strong candidate for "real" artificial intelligence.

I argue we have that leeway because of the way GP surfaces Pickering's Mangle. When the methodology "works", it does so by offering *helpful resistance* in our engagement with the problem at hand, whether in the form of surprising answers, or validation of our suspicions, or simply legible suggestions of ways to make subsequent moves. GP *dances* with us, while most other machine learning methods are "mere tools".

2.3 Against replication

Nonetheless, there seems to be a widespread desire, inside and outside our field, to frame GP as a methodology for producing *unsurprising* models from data, more in keeping with the traditional linear of scientific work. That is, an idealized user is expected to proceed something like the users of any other machine learning or mathematical programming framework:

1. frame your problem in the correct formal language
2. "get" a GP system
3. run GP "on your data"
4. (whatever this is, it's not our problem)
5. you have solved your problem

This is not original with the GP community; there is strong pressure from our peers in other disciplines and our users to promote this use case, not least because it is *exactly* the stance expected in any planning or public policy setting, or in any scientific or programming project management setting. That is, we are under tremendous social pressure to treat GP as a *tool* to be invoked in a known, predictable and well-described planning situation.

The resulting resistance from early-adopting users shouldn't be unexpected. *Being surprising* may well be the worst conceivable behavior for any tool to be used in a traditional project setting. And given that pressure, it's no wonder that so much of GP research is focused on constraining tweaks to bring GP "into line". If only GP could be "tamed" or made "adaptive" so that step (4) above *never happens*.

I imagine this is why so many GP research projects strive for rigor in the form of counting replicates which "find the solution": they aim not to convince users of the known strengths of GP, but rather demonstrate to critical peers that GP can be "tamed" into a mere tool.

What would a "replicate" stand for, to a user who sought to exploit GP's strength in a project (whether theoretical or practical) where *search* rather than the algorithm itself is the primary focus? Projects which authentically "use" GP *must necessarily* be those searching for noteworthy answers—which is to say *surprising* and *interesting* answers—that they could not otherwise obtain. Thus, we should better think of a "replicate" as a sort of proxy for user frustration in step (4) above: that is, it represents a project in which search begins, stalls, and for which the user cannot see a way to move search forward towards more interesting and useful answers.

But I think we would agree: it is a poor researcher who, when faced with a stumbling block in the form of a black box's misbehavior, doesn't attempt to work around that obstacle. Not to *begin the project again from scratch*, but to make changes and continue. Any researcher who is using GP *realistically* will be watching, and adjusting, and engaging and interacting with the process of search itself.

When a GP user is viewed as working within the traditional *linear* narrative of research, we see her run a population of 100 individuals for 100 generations, peer at the results that have been dumped into a CSV file, and find them wanting. Desiring an actual *answer*, she adjusts the GP parameters and begins another run of 100 generations… and repeats as necessary.

But there is *no discernible difference* when we frame this same process of "many runs" as a single project involving initial researcher moves, resistances thrown up by the *thing made*, and subsequent accommodations made by the researcher to the new perceived truths. Except, that is, in the researcher's view of her own response to resistance: if she comes to the project with *plans* for running "ten replicates", we can only assume she has learned from someone that a search algorithm is *supposed* to forget everything it has learned every 100 generations….

I cannot help but be reminded of the fallacy, surprisingly common both in and outside of our field, that "artificial intelligence" must somehow be a self-contained and non-interactive process. That is, that an "AI candidate" loses all authenticity as soon as it is "tweaked" or "adjusted" in the course of operation. It is as if every new-born "AI" must be jammed into an air-tight computational container and isolated *until it learns to reason*, and that without exceeding a finite computational budget. If humans creating *real* intelligences treated them anything like the way computer scientists insist we treat nascent *artificial* ones, murder charges would be forthcoming.

There are several practical reasons for us to try something different.

Consider our hypothetical GP user. In keeping with the Behaviorist standards of GP, she is carefully "not interfering" with any given run of 100 generations; she can only peer at a results file after the fact. During the course of any 100 generations, all sorts of dynamics have happened: crossover, mutation, selection, all the many random choices. Imagine for a moment we were given perfect access to the entire dynamical pedigree of the unsatisfying results she receives at the end, and were

able to backtrack to any point in the run and change a single decision. Before that point, it's unclear how badly things will actually turn out at the 100-generation mark; at some point after that juncture, it's obvious to anybody watching that the whole thing's a mess.

If such miraculous insights were available, then surely the correct approach would be to intervene and adjust the situation when the crucial point was reached... and then continue. Lacking (as we do) this miraculous insight, *why then does it seem reasonable to stop any run arbitrarily at a pre-ordained time point and begin again from scratch?*

It is, I think, because the myths of artificial intelligence and the linear narrative of science are so deeply intertwined. It is frowned upon to admit in a scientific paper, even when no mistakes were made, that the original vision and plan changed over the course of the project; rather we are expected to describe research *results* as the inevitable outcomes of an ahistorical process, and erase all resistance and accommodation actually done by human beings in the context of their projects. Similarly, it feels somehow wrong to admit in a GP project, even if every parameter was set correctly, that the original vision and plan gave way to the inevitable surprises thrown up by GP's inherent tendencies to do just that.

But insofar as GP *surprises us*, and since that is its sole strength over more predictable and manageable frameworks, we must inevitably see a good fraction of those surprises at least in part as *disappointments* rather than encouraging opportunities to change our plans. Let's learn new ways to accommodate those disappointments, and stop trying to make them go away.

3 "TDD as if you meant it"

As far as I can tell, Keith Braithwaite first described his training exercise for software developers in 2009. The target of the exercise is "Pseudo-TDD": the noted habit among software developers who claim to "know" and "do" test-driven development as part of their daily work towards a sort of thoughtless approximation of the technique.

I should note that a number of agile software development practices share informative relations to genetic programming's dynamics[4], but in this work I'll focus on those of TDD. In particular, test-driven development (or more accurately "test-driven design") *when done correctly* can break down the complex design space of a software project into a value-ordered set of incremental test cases, focus the developers' attention on those cases alone, inhibit unnecessary "code bloat" and feature creep, and produce low-complexity understandable and maintainable software.

[4] I imagine there is an Engineering Studies thesis in this for some aspiring graduate student: Genetic programming and agile development practices arose in the same period and more or less the same culture, and both informed by the same currents in complex systems and emergent approaches to problem-solving.

TDD *as such* is a rigorous process, to the point where it can be described as "painful" (though also "useful") by experienced programmers. The steps are deceptively easy to trivialize and misunderstand, especially for those whose habits of thinking about code are ingrained:

1. Add a little (failing) test which exercises the next behavior you want to build into your codebase
2. Run all tests, expecting only the newest to fail
3. Make the minimal change to your codebase that permits the new test to pass
4. Run all tests, expecting them all to succeed
5. Refactor codebase to remove duplication

Each stage offers a stumbling block for an experienced programmer, but the most salient for us now is the iterative flow of implementation (or "design") that it imposes: Each cycle begins with a choice of *which little test should next be added*; each cycle ends with a rigorous process of refactoring, not just of the new code but of the *entire cumulative codebase* produced so far. The middle three steps—implementing a *single* failing test and modifying the codebase *by just enough* so that all tests pass—feel when one is working through them as if they could be automated easily. The *mindfulness* of the process lives in the choice of next steps and (though somewhat less so) of standard refactoring operations.

Braithwaite's exercise does an interesting thing to surface the formal rigor of this approach. In it, the participants (willing, of course, because the exercise is a *kata* or "refresher" for experienced software developers to hone their skills) are asked to implement a nominally simple project like the game of Tic-Tac-Toe, given an *ordered* list of features to implement and the artificial restriction that they must go farther than normal TDD practice asks. Rather than producing a suite of tests and a self-contained codebase, they are forced to use *only* refactoring of code added to tests to produce their eventual "codebase". In other words, no code can be "produced" until the "smell" of duplication in the code added to multiple passing tests *provides a warrant* for refactoring it out.

Further, a facilitator patrols ongoing work and deletes *any and all code not called for by a pre-existing failing test*. Words like "irritating" and "annoying" crop up in participants' accounts of this onerous backtracking deletion the first few times it happens, as one might imagine. But as Gojko Adzik emphasizes in his descriptions of workshops, the resulting designs for even simple algorithms in this artificially amplified setting seems much more *open-ended* than it would if the software were built under the typical norms and habits an experienced programmer uses in normal conditions.

A number of contextually positive benefits are attributed to agile software development practices, and to TDD within that suite of practices. But the one that brings us here today is that aspect surfaced particularly in Adzik's account of Tic-tac-toe:

> By the end of the exercise, almost half the teams were coding towards something that was not a 3×3 char/int grid. We did not have the time to finish the whole thing, but some interesting solutions in making were:

- a bag of fields that are literally taken by players—field objects start in the collection belonging to the game and move to collections belonging to players, which simply avoids edge cases such as taking an already taken field and makes checking for game end criteria very easy.
- fields that have logic whether they are taken or not and by whom
- game with a current state field that was recalculated as the actions were performed on it and methods that could set this externally to make it easy to test

In other words: innovative approaches to the problem at hand began to arise, though there wasn't enough time to finish them in the time alloted for the exercise. The analogy to our collective experience to date with genetic programming should start to peek through at this point—though the thoughtful reader will hopefully wonder what utility there is in an analogy drawn between two similarly unsatisfying outcomes.

[more here]

4 GP as if we meant it

In the same way that Braithwaite's onerous coding exercise is intended to drive the attention of its participants toward test-driven design with its obligation to write "real" code *only as a refactoring*, I'd like to be able to demand a *warrant* for every step that moves our changing genetic programming setup away from just plain random guessing. Braithwaite's target of "Pseudo-TDD" suggests an analogous "Pseudo-GP": one in which the fitness function is the only "interface" with the problem itself, and where the representation language, search operators, search objectives and other algorithmic "parameters" are *fixed*.[5]

Not only do traditional search operators like crossover, mutation and [negative] selection not come "for free" in this variant, but in every case we must develop a cogent, data-driven argument in favor of starting them *as part of an ongoing search process*. Similarly, the initial selection criteria will be limited to a minimal subset of the training data, and expansion (and other alterations) of the training set will have to be made in light of measured progress, not assumptions that "more is better" in every case.

The result will be an incremental process of refinement of an ongoing search, carried out not at the level of externally-assigned parameter "tweaks" but rather by *opening* the black boxes we typically demand and demanding we do surgery to correct their "pathologies" (and understand their mysteries) without killing them outright. It is not intended as an "algorithm" to supplant those used today, but rather as a forced re-description of what we actually already do.

[5] In much the same way that Braithwaite's participants often acknowledge they *know* and *use* TDD as it's formally described, but rarely take the time to do so unless "something goes wrong", I imagine many GP users might say they *know* and *use* all the innumerable design and setup options of GP, but treat them as adjustments to be invoked only when "something goes wrong". I offer no particular justification for either anecdotal stance here, but the curious reader is encouraged to poll a sample of participants at any conference (agile or GP).

4.1 The tableau representation of GP systems

I find it helps to present a simplified but formalized description of GP systems, and one which highlights the particular features I'm considering.

At the highest level of abstraction, we will treat GP as a collection of particular *decisions* made by the user, plus an otherwise autonomous stochastic process executed by software and hardware, which can be "started", "paused" and "resumed" but which cannot be *restarted*.

The decisions available to the user apply to three core components of the stochastic process: `operators`, `answers`, and `rubrics`.

4.1.1 Operators

An `operator` is any function which takes as argument a (possibly empty) collection of `answers` and produces a new collection of `answers`. Operators thus include "random guessing", which in GP systems is often used to build an initial population, any "crossover" and "mutation" functions, but also any arbitrarily complex function which might be used to *create a new answer*. So for example "particle swarm on an abstract expression tree" would be an `operator` working on a single `answer` and producing a very large collection of results.

In the exercise (but not in the examples that follow) *every intermediate result* which is an `answer` is considered to be part of the return value. That is, `operators` are obliged to return all intermediate `answers` they produce while building their "actual" results: a function that implements "crossover-and-mutation" will is obliged to return *all* crossover products it builds, in addition to the results of a subsequent internal mutation.

Note that no process is provided for `answers` to be *removed* from a tableau. This framework is purely cumulative.

4.1.2 Answers

An `answer` is what might traditionally be called an "individual" in GP literature, though there are subtle differences. We can model them programmatically as key–value hash, typically beginning with a "genome" or "script" field set to an abstracted representation for a solution to the problem at hand. As an `answer` "matures" in the unfolding tableau, various other attributes will be appended and set by other algorithmic processes, such as fitness scores or measured attributes like "age" or "alive?" states.

In the tableau layout, we will represent the unfolding collection of `answers` as the *rows* of a spreadsheet-like table, and their attributes and scores as the *columns*.

4.1.3 Rubrics

A `rubric` is any function which returns a scalar numerical value, given arguments of a collection of one or more `answers` (and possibly additional arguments), and assigned to a particular `answer`.

Insofar as any given `rubric` is associated with an objective of search, it should be framed as a *minimization* form. For errors, this should be obvious; for `rubrics` used in ALPS-like systems, realize that the desired `rubric` to select *younger* `answers` is not "age" but "youth".

In our final exercise (though not in the examples to follow) there will also be a strict requirement that any attribute associated with an `answer` is also available as an *implicit* `rubric`: the script, the creation time (if recorded), or any other `rubric`. We will not permit `rubrics` to store intermediate values, *except in other rubrics*: thus a `rubric` which specifies "sum squared error over a training set" cannot be applied without also *first* creating `rubrics` for "measured error when provided input *i*" every element *i* of the training set. If the training set has 100 elements, then the SSE `rubric` implicitly represents a suite of 101.

However, in the examples to follow that sketch "traditional GP", we can relax this restriction.

4.1.4 Search process

4.1.5 "traditional GP" tableau

It's tempting to be glib about defining "traditional GP", and for the sake of brevity let me succumb to that temptation: Say it is a fixed-size population of 100 `answers` which are created initially at random and subsequently by crossover and mutation, a single `rubric` which returns a single SSE score calculated over input–output pairs measured over a static collection of training cases.

individual	genome	SSE	(notes)
0.1	...	0.2	
0.2	...	0.8	
...			
0.N	...	0.1	

The analogy to SQL.

4.1.6 "lexicase selection tableau"

The analogy to SQL.

4.1.7 “lazy lexicase selection tableau”

The analogy to NoSQL.

4.1.8 “mutually lazy tableau”

“Send me a message.”

4.1.9 “as if we really meant it”

I need your help.

5 Leveraging “resistance”: the problem of the dots and lines

5.1 dots and lines, in pushforth

5.2 the problem of the unknown language

5.3 the problem of the objective(s)

5.4 expanding the goals when it’s boring

6 Exploration and exploitation interfaces and affordances

7 Final thoughts: What should it mean to *act intelligently*?

(It would mean the sort of self-awareness it takes to notice that something is wrong, and to ask for help.)