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 (and the broader body of work to which it belongs, which I prefer to call "generative processing" and will also abbreviate as "GP") embodies a very particular *stance* towards the scientific and engineering work of modeling, design, analysis and—though least of all—optimization. A case could be made that the resistance we have all recounted towards GP from the technical and lay audience has little to do with the technical results so amply demonstrated over the last quarter-century, but rather from discomfort among that audience in GP's particular "way of working" on problems. There is a tacit assumption, even among GP theorists and practitioners, that science and engineering are "rigorous", or even successful, only when they proceed through ordered phases of

1. conceptualization, the "vision thing"
2. planning
3. design
4. architecture
5. implementation
6. testing
7. debugging

Very few scientists or engineers would admit that any *real* project has followed this narrative structure in practice, but in every phase of work life from fund-raising to published reports the story is framed in that way, especially in scientific work: Because of work gone before, an idea or insight was had. The idea was framed as a formal hypothesis. The hypothesis (and current Best Statistical Practices) suggested an experimental design obvious to any in the discipline. The experimental design was undertaken, the data collected, the hypothesis tested, and 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 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

religious belief.[1] 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 more today.

2.1 Pickering's "Mangle of Practice"

Andrew Pickering's *Mangle of Practice* is the one I'd like to use here. This approach is noteworthy because it is so close to our actual experience using GP, of the "didn't we already know this?" sort. At the risk of eliding a lot of extraordinarily well-considered structure, let me summarize:

The act of "doing science" is *at no point whatsoever* a behavior undertaken by an isolated "researcher" in an objective field of externalities. Rather, the researcher who initiates this process with a vision, intuition, hypothesis or hunch begins always by *making some artifact*: code, an instrument, a proof, a sketch, a maquette or even a thoughtful conversation at a conference.

The thing, inevitably, *resists*. That is, Pickering grants it agency, or rather makes it the agent of all those externalities that impinge on the work to make the *vision* differ from *practice*: the facts of the actual world, the cultural assumptions and habits of discipline, the raw materials and toolkit available to the practitioner, and so forth. "Resistance" here is not merely the obvious trouble of a software bug or some missing parts, but includes one's sense *on seeing it* that something's not quite right; the realization that more (or less) is needed. The language we use, in the face of this resistance, is that the *thing made* "feels wrong" or "points something out", that it "wants to do X instead of Y", or that "it's doing something too complicated for me to understand right now".

And—assuming science is indeed what's being done—the researcher *changes in response to this resistance*. That vision 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.

The "mangling" of Pickering's metaphoric name refers not to wounding but to the mechanical laundry apparatus, the antique wringer through which sheets and linens are run to drain the water, and which as a *side-effect* impose a novel higher-dimensional structure and juxtapose unexpected components with one another. Here, the response of the researcher to resistance she experiences in the course of her project is no less a part of the dynamics of "science" than the act of writing and running code, the authority warranted by particular statistical practices in her discipline, or the raw pressure of physical laws.

---

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

2.2 GP as "mangle-ish practice"

The broader field of machine learning leans 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 random forest on a given data set is not expected to be a *surprise* in any real sense, but rather the sufficiently robust result of applying numerical optimization to a particular mathematical programming model. Whether one describes the process as minimizing out-of-sample error or maximizing information gain, the focus of the discipline is on contingent reliability rather than exploratory modeling.

GP has the capacity to *tell us stories*, even in the "simple" domain of symbolic regression problems. The space under consideration is not merely a vector of numerical constants or a binary mask over a suite of input variables, but the *power-set* of inputs, and of functions over inputs, and of higher-order functions over those. While GP can be used to explore arbitrarily close to a paradigmatic model, we argue for it most when its application can produce unexpected insights. A number of us treat it as the best candidate for "real" artificial intelligence, and rightly so.

We can do that because GP surfaces Pickering's Mangle. When it "works", it does so by offering *helpful resistance* in our engagement with it, whether in the form of surprising answers, validation of our suspicions, or suggestions of ways to make subsequent moves. It *dances* with us, in a way that the other mere tools of machine learning do not.

Now a great deal of the last quarter-decade of work in the field, especially in the early days, seems to treat GP as a close relative of other machine learning techniques—as a methodology for producing *unsurprising* models from data, and within the traditional model of scientific work. That is, an implicit and idealized user is expected to proceed something like the users of any other machine learning framework:

1. frame your problem in the framework-specific language
2. "get" a GP system
3. "run GP on the data"
4. ???
5. you have solved your problem

There's strong pressure from our actual user community to enforce this stance, not least because it is exactly the stance of planning and public policy, and of the mythic science or programming project manager. That is, it frames GP as a *tool* to be invoked in a known and well-described planning situation.

The resulting resistance from early-adopting "users" who "get" an off-the-shelf GP system and try to apply it to their problem—at least within this narrative of work—should not be unexpected. *Being surprising* is perhaps the worst choice for any traditional project plan. It's no wonder that so much of GP research is focused on particular tweaks and staged horse-races among the unnumbered techniques: as long as there is a sentiment that GP might be "tamed" so that step (4) above *never*

happens, so that the framework might join its more traditional and popular relatives, there will be strong pressure to calculate statistics on, for example "proportion of successes in 30 replicates".

But what does a "replicate" mean for somebody using GP for a real project, whether theoretical or practical, where *GP itself* is not the focus of the work? Projects which authentically "use" GP must necessarily be searching for noteworthy answers, which is to say *surprising* and *interesting* answers, that they cannot otherwise obtain. To do otherwise is to somehow expect GP to be a form of "research automation", a premature expectation in light of our experience.

Therefore, any researcher who is using GP *realistically* is one who is watching, and adjusting, and engaging and interacting with the process of search itself. Seen as working within a traditional (and deeply misleading) narrative framework, we watch as she runs a population of 100 individuals for 100 generations, peers at the results in a CSV file and finds them wanting, and then adjusts the GP parameters to run another 100 generations… and repeating as necessary. But there is *no discernible difference* when we frame the same process as one of participating in the single application of an interactive algorithm that (remarkably, and unaccountably when you think about it) *throws away all intermediate results every 100 generations*.

Further, let's explore that traditional narrative which sees her workflow as a series of repeated attempts to use GP as a tool in a situation for which it is poorly suited, or maybe "badly tuned". She is carefully "not interfering" with any given run of 100 generations, observing our tradition in this field, and can only peer into the 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 vision-driven science are deeply intertwined. It is somehow "cheating" to admit in a scientific paper, even if no mistakes were made, that the original vision and plan changed over the course of the project; rather we describe research *results* as inevitable outcomes of an ahistorical process that erases the work actually done by human beings. Similarly, it is somehow "cheating" 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 tremendous potential to surprise.

But insofar as GP *surprises us*—and that is its sole strength over more predictable and manageable frameworks—we will inevitably see a good fraction of the surprises as *disappointments* rather than opportunities to adapt our selves.

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[2], 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.

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 test
2. Run all tests and fail
3. Make a little change
4. Run the tests and succeed
5. Refactor to remove duplication

Each stage offers a stumbling block for an experienced programmer, but the most salient for us now is the overarching flow of implementation (or "design") that it imposes: Each cycle, and also the overarching design process, begins with the 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 over all iterations of this cycle. 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

[2] 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.

produce their eventual "codebase". In other words, no code can be "produced" until *duplication in code added 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 he's run, the results of "design" of even simple algorithms in this artificial amplified setting seems much more *open-ended* than it would if the software were built within the typical framing of habits and assumptions that an experienced programmer carries along with her to any project.

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 As if we *really* 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*.[3]

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.

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.

---

[3] 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).

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.)