NAVIGATING MULTI-DIMENSIONAL LANDSCAPES IN FOGGY WEATHER AS AN ANALOGY FOR GENERIC PROBLEM SOLVING

David RUTTEN
Robert McNeel & Associates, Austria

ABSTRACT: Developing algorithms that solve specific problems is a major area of research in computer science. Minimizing runtime complexities, reducing memory overhead, quickly returning intermediate or partial results, and balancing speed vs. accuracy are all important issues that need to be well understood when one wants to master algorithm design. It is safe to say that the people who need algorithms vastly outnumber those who can write them. However if we are willing to sacrifice performance, the application of *generic solvers* may democratize the field.
In principle, generic solvers are capable of solving most problems. This isn't quite as magical as it sounds, mostly because it involves a large amount of preparatory work by intelligent human beings. This includes defining the problem *phase space* and the relevant *fitness function*, which together make up the *fitness landscape*. Generic solvers are designed to find their way around these landscapes and to converge on high ground as quickly as possible.
This paper serves as an introduction to the theoretical side of generic problem solving with a strong focus on the geometry and topology of fitness landscapes. Along the way, several implementations of popular solver algorithms will be discussed.

Keywords: Generic Solvers, Phase Space, Fitness Function, Fitness Landscape, Peak Finding, Hill Climbing, Simulated Annealing, Evolutionary Algorithms

1. PROBLEM SOLVING

The problem of how to systematically solve problems goes back a long way. The history of *algorithmics* can be traced back to the mathematicians *Brahmagupta* and *Abū ‘Abdallāh Muḥammad ibn Mūsā al-Khwārizmī* from the 7th century India and 9th century Persia respectively. Algorithms are sets of unambiguously repeatable instructions which, when confronted with the same question, reliably give the same answer. The earliest known algorithms were devised to solve specific arithmetic problems such as multiplication or finding greatest common divisors.

Only in the last few centuries have mathematicians begun to approach algorithms from an analytic point of view, creating an *arithmetic of algorithms* which allows for their systematic classification and evaluation.

1.1 Problem categories

Computational mathematicians have come up with a set of classes to categorize different types of problems. Some of the better known classes are *P*, *NP*, *NP-Complete*, and *NP-Hard*. The class to which a problem belongs indicates roughly how quickly it can be solved. For example, it is always possible to generate answers to P class problems within a reasonable amount of time[1]. Problems that belong to NP-Complete on the other hand are more difficult as there exists no reliable way to generate a solution. However, *if* a solution is proposed, it is at least possible to recognize it as such. NP-Hard problems don't even have that luxury, not only is there no known

[1]Incidentally, ‘a reasonable amount of time’ in computational complexity means ‘before the universe ends’, so this is not necessarily a *practical* categorization.

way to generate an answer, it isn't even clear how to test a tentative answer for correctness.

For the purposes of this paper we shall comprehensively ignore the accumulated knowledge of the past thousand years of algorithmics and approach all problems in an NP-Hard-ish fashion. That is:

- it is not known how to generate the correct solution,
- it is not known how to test a proposed solution for correctness,
- but it *is* possible to compare two proposed solutions and select the more correct one.

1.2 Solver categories

Solutions to problems —or rather, *methods for finding* solutions to problems— can of course also be categorized. Algorithms can be described as *greedy* or *lazy*, *stochastic* or *deterministic*, *iterative* or *recursive*, *exact* or *approximate* and a million adjectives more. These characteristics are by no means mutually exclusive or indicative of quality. Different situations call for different types of algorithms. Take for example the age-old problem of sorting a collection of values. There exist at least a dozen famous sorting algorithms[2], each with its own strengths and weaknesses. Which algorithm is best depends on whether one is sorting large or small collections of data, whether the data is already somewhat sorted, whether the algorithm is allowed to use a lot of memory, whether the algorithm should yield intermediate results if aborted, how unsortable data is handled and so on and so forth.

Since a number of different solver algorithms will be discussed below, a passing familiarity with some relevant algorithm categories is important. A casual definition of each will suffice in this context.

[2]Visit sorting-algorithms.com for a visual comparison of the seven most common ones.

Greedy algorithms are very local-minded. They make decisions based on the immediate environment rather than taking long-distance or long-term goals into account.

Deterministic algorithms follow a fixed and predictable process which tolerates no chance or randomness.

Stochastic algorithms have a random component to them and are therefore less predictable than deterministic algorithms[3].

Exact algorithms are guaranteed to find the best possible solution(s) given the initial constraints.

Approximate algorithms will typically find some sort of solution without any guarantee that it is the best possible one.

Progressive algorithms compute solutions in an iterative manner, where each cycle yields —on average— a better answer than the previous one.

Adaptive algorithms can operate on a changing set of constraints and inputs. These algorithms run continuously within a dynamic environment.

Specific algorithms are designed to solve only one kind of problem. As a result they tend to be fast and reliable, but they are difficult to write. The vast majority of algorithms used on computers today fall into this category.

Generic algorithms are designed to tackle a wide variety of problems. This flexibility is accompanied by a significant drop in performance.

Open algorithms allow external entities (be they human beings or other algorithms) to participate in the solving process. Seemingly unpromising lines of inquiry can be investigated upon the request of an external actor.

[3]On digital computers, *all* processes are inherently deterministic, but pseudo-randomness is sufficient to classify an algorithm as stochastic.

2. PROBLEM ANALYSIS

So how exactly is a generic solver able to deal with many different problems? Doesn't that require a large amount of intelligence and understanding? The short answer is *yes*, but the key insight is that the intelligence need not reside in the solver itself. If the solver is *open*, another algorithm (which may well be classified as *specific*) can fill in the knowledge gaps. In other words, a generic solver doesn't need to know the first thing about nurbs geometry in order to find the intersection point between two curves, all it needs is a *companion algorithm* that does. By separating the 'knowing' and the 'solving' into two disjoint algorithms, they both become easier to write and easier to repurpose.

Communication between the generic and the companion algorithm can be thought of as a mixture of *twenty questions* and *hunt the thimble*. The generic algorithm confronts the companion algorithm with a tentative solution, to which the latter assigns a quality rating (cold, warmer, hot!). The generic algorithm is tasked with interpreting the clues and the companion is tasked with computing the 'temperature' or *fitness* of each proposed solution. For this exchange to work, both algorithms need to speak a common language. Happily this language is very succinct and it can be described by a single mathematical equation:

$$f(\tau) = q \tag{1}$$

$$f : \tau \to \mathbb{R} \tag{2}$$

Equation (1) defines the common language as a mathematical function, while equation (2) shows the mapping of this function, which is the mathematical way of specifying what sort of data goes in and what sort of data comes out. Terse though this notation may be, it hides a number of fairly abstract ingredients and deserves a detailed discussion.

In the above notation f represents the fitness function, which operates on τ (more on that later). The output of f is labelled q (for quality) and it represents the fitness as a single numeric value[4]. The mapping notation (2) merely states that f has to consume data in the form of τ while it should return data in the form of a real number.

This leaves us with τ, which is a type of *tensor*. A tensor is simply a collection of variables called *elements*, which together describe all possible answers to the problem at hand, both the good and the bad ones. In fact the precise layout of τ depends on the nature of the problem, and defining this layout is the responsibility of the companion algorithm. Since digital computers are only capable of dealing with numeric data, all elements that make up τ must be numbers.

Along with a tensor definition, the companion algorithm must also specify which transformations can be applied to τ. Usually this involves nothing more complicated than a list of directions in which τ is allowed to be pushed and —for each direction— how far one can push it.

Perhaps an example is the best way to explain what a tensor in this context is. Imagine one is asked to paint a 10 cm dot on the Mona Lisa so that it least disrupts the original painting. This is basically an optimization problem of two variables; *position* and *colour*. We need to find which combination of position and colour has the highest fitness. However position and colour are not numbers and thus cannot directly be elements of τ. Luckily it is possible to describe positions and colours using numbers, which provides a way forward for defining a tensor for this particular problem:

$$\tau = \{X, Y, R, G, B\} \tag{3}$$

The position variable is represented by two numbers that encode the horizontal and vertical offset of the paint dot, while the colour variable

[4] It has to be a single number as it is vital that two fitnesses can be unambiguously compared using relational operators. If the fitness were more than a single number, then the problem can be described as *multi-variate*, and in such cases solver algorithms are employed to map out *families of solutions* that balance multiple conflicting fitness metrics. This paper will not discuss such cases.

requires three numbers, one for each of the RGB channels. A tensor element is defined by two properties; namely the *index* at which it occurs in the tensor, and also a *set of possible values* it can assume[5]. The R, G and B elements for example can be assigned any integer value in the range 0–255. If we assume the dots can only be positioned on the painting at one centimetre intervals, then X and Y are constrained to the integers in the ranges 5–48 and 5–72 respectively[6]. These limits make it possible to calculate the total number of different *states* τ can assume. The number of states, also called the *variability* of a tensor, can be written using angle bracket notation:

$$\langle\tau\rangle = \langle 44, 68, 256, 256, 256\rangle \quad (4)$$

$$\langle\tau\rangle = 44 \times 68 \times 256^3 \quad \approx 5 \times 10^{10} \quad (5)$$

There is now enough information to do some actual maths. The tensor for this problem contains five elements which are entirely independent of each other, meaning that the amount of red in the colour is not limited or affected by the amount of blue or the x-position of the dot. Five independent elements define a five-dimensional tensor. The variability of τ equals the product of the variabilities of all the elements of τ, as per equation (5). Even if a single call to $f(\tau)$ took only ten milliseconds, it would still take nearly sixteen years to iterate over all unique states of τ. This is clearly not a practical approach.

The set of all states of τ is called a *phase space* and is denoted with the symbol $\mathcal{P}$ (a superscript integer is sometimes included to denote dimensionality). Every possible location in $\mathcal{P}$ is identified by a specific tensor. In this sense, tensors act as coordinates. Usually the phase space can be thought of as an N-dimensional hypercube, which is at least a reasonably fathomable concept[7].

A note on phase space sizes. The solver algorithm mentioned above (i.e. 'try every possible answer and remember the best one') is called a *brute force search.* For small problems brute force may be a reasonable solution, but as the problem grows larger the time it takes to enumerate and test all possible states increases rapidly. Table 1 shows brute force runtimes for several different tensor definitions, while assuming a 10ms duration for a single measurement. The rows contain tensors with one, two, three, or four elements, while the columns contain different variabilities of the tensor elements.

$\lvert\tau\rvert$	$\mathbf{10^1}$	$\mathbf{10^2}$	$\mathbf{10^3}$	$\mathbf{10^4}$
1	0.1s	1s	10s	2m
2	1s	2m	3h	12d
3	10s	3h	116d	317y
4	2m	12d	317y	3My

Table 1: Brute force runtimes

A tensor with three elements (6) where every element can take on one thousand unique values (7) results in a phase space size of one billion unique coordinates (8). Even when a single iteration takes only ten milliseconds, to evaluate each and every one would still take 116 days.

$$\tau = \{A, B, C\} \quad (6)$$

$$\langle\tau\rangle = \langle 10^3, 10^3, 10^3\rangle \quad (7)$$

$$\langle\tau\rangle = 10^9 \quad (8)$$

[5]In the case of interdependent elements, additional relational properties come into play.

[6]The dimensions of the Mona Lisa are 53 cm × 77 cm, however dots cannot be placed closer to the edge than 5cm, otherwise they would only partially cover the painting, which would be cheating.

[7]In the more complicated case of interdependent tensor elements, it could be that the variability of one element (i.e. whether it can assume 1, 14 or 3517 different states) depends on the value of another element. In these cases the dimensionality of $\mathcal{P}$ may differ from point to point. For the remainder of this paper we will ignore variable dimensionality.

2.1 N$\frac{1}{2}$ dimensional landscapes

It is not feasible to properly convey what a complete phase space of more than two dimensions looks like on paper, which is why I will adopt the space-time diagram convention of cosmology and draw the phase space as a flat, two-dimensional plane. In such a representation the individual tensors that populate the space are all sitting side by side on the plane.

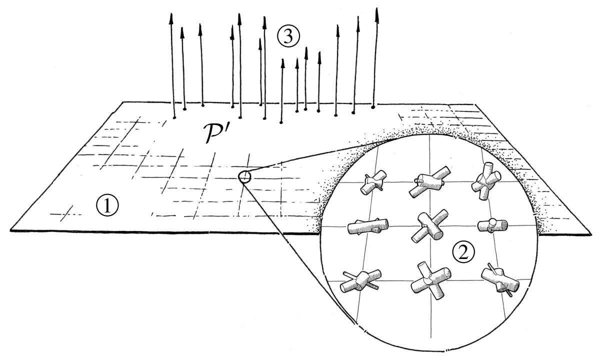

Figure 1: $\mathcal{P}$ as the set of all tensors

In the above figure ① represents the entire phase space $\mathcal{P}$ as collapsed to a plane[8]. The objects marked ② represent the individual tensors (visualised as cylinder permutations rather than collections of numeric values). The vertical bars marked ③ represent the fitness values of all tensors. These values are computed by the companion algorithm via $f(\tau)$.

The notion of a *fitness landscape* $\mathcal{L}$ as the complete set of all fitness values now starts to emerge. That is, if we apply $f(\tau)$ to every τ in $\mathcal{P}$ and thus draw all possible arrows, we get a scalar field of fitness values that pervades all of $\mathcal{P}$. The easiest way to imagine this geometrically is to treat the fitness value at each point in $\mathcal{P}$ as an elevation, thus 'pulling' $\mathcal{P}$ from a flat plane into a landscape.

[8]As a reminder that this is a *projection* of $\mathcal{P}$ onto two dimensions, this paper will include a $\mathcal{P}'$ symbol in any figure which contains such a collapsed space.

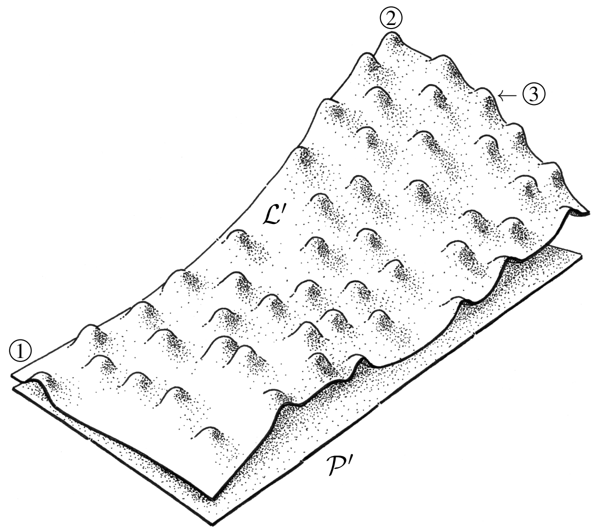

Figure 2: Fitness landscape as an extrusion of $\mathcal{P}'$

Figure 2 shows the fitness landscape resulting from a fitness function being applied to a phase space[9]. In the landscape one can immediately see certain characteristics of $\mathcal{P}$ and $f(\tau)$ such as low quality solutions ①, high quality solutions ②, and local optima ③.

Since the mapping of the fitness function is defined as $\tau \rightarrow \mathbb{R}$ (see equation (2), page 3), we only get a single fitness or 'elevation' for every point in $\mathcal{P}$. Which means that planes, mountains, and valleys are all possible features of fitness landscapes, but caves, bridges and overhanging cliffs are not, as that would require more than one elevation value per coordinate. In this sense the landscape falls ever so slightly short of being truly three-dimensional. This sort of geometry is typically referred to as two-and-a-half dimensional in the machining and graphics industries[10]. Since all phase spaces are of integer dimensionality, it follows that all fitness landscapes are of integer dimensionality + $\frac{1}{2}$.

[9] Again, to avoid misunderstandings, fitness landscapes that are drawn as $2\frac{1}{2}$ dimensional objects are marked with an $\mathcal{L}'$ symbol.

[10]Not to be confused with the partial or 'Hausdorff' dimensions that occur in fractal geometry.

Although it is not possible to draw landscapes whose dimensionality exceeds $2\frac{1}{2}$, it is perhaps possible to describe them verbally and get a sense of how the complexity increases with every additional dimension. A one-dimensional phase space can be represented by a line, like the x axis on a typical graph. The associated landscape must therefore also be a single curve that moves up and down as one travels from left to right along $\mathcal{P}^1$, see figure 3. The best solution can be found wherever the curve has its highest peak, which is a trivial exercise for the human eye (unless there are multiple peaks with very similar elevations). One could spot such a peak after looking at a curve for no more than half a second.

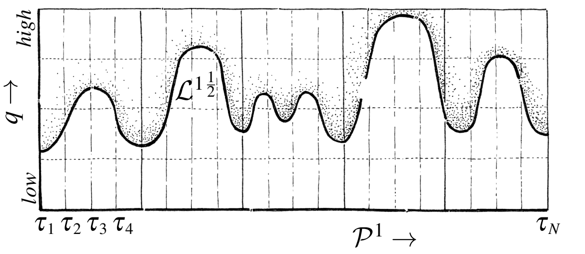

Figure 3: $f(\tau)$ for $\mathcal{P}^1$.

In the two-dimensional case, $\mathcal{P}^2$ and $\mathcal{L}^{2\frac{1}{2}}$ actually look like the diagrams in this paper, but the geometry becomes significantly more complex with $\mathcal{P}^3$ and $\mathcal{L}^{3\frac{1}{2}}$, as it is not possible to imagine an extrusion at right angles to a volume. However instead of a landscape, one can think of an elevator car with cigarette smoke. The interior of the car is an analogy for $\mathcal{P}^3$ and the density of the smoke represents the fitness at each point in $\mathcal{P}^3$. The smoke will be thicker in some places and more spread out in others. Instead of a peak in a landscape, the search is now for the cubic millimetre with the thickest smoke. This will certainly require a lot of squinting and head bobbing, if one can see it at all.

Extending this analogy to one more dimension ($\mathcal{P}^4$ and $\mathcal{L}^{4\frac{1}{2}}$) requires that one take into account not just the smoke at any one instant in time, but the motion of smoke particles during an entire time interval. In this case the goal is to find the cubic millimetre with the highest density at any given second between the doors closing in the lobby and the doors opening on the fifth floor. Every additional dimension increases the size of $\mathcal{P}$ in an exponential fashion, which is why the runtimes in table 1 increase so dramatically with every row.

3. THE FAMOUS FOUR

Up to this point the purpose of this paper has been to introduce the theory of tensors, phase spaces, fitness functions, and fitness landscapes. While it is possible to draw entire landscapes as a theoretical exercise, one should bear in mind that at the start of a new search, the actual shape of a fitness landscape is entirely unknown. The diagrams in this paper serve only as visual aides and one should not infer from them that these landscapes are at any time entirely computed. There is one thing —and one thing only— that can be done to gain information regarding the shape of the landscape, and that is to pick some tensors and see what fitness values $f(\tau)$ assigns to them.

One can think of it as standing in the landscape and having a GPS receiver but no map, while being surrounded by such thick fog that even the terrain underneath is barely visible. Given these limited tools, how does one find high-ground? This is the issue that generic solvers must deal with, and different solvers take different approaches. Like a GPS reading, a single sample may take a significant amount of time. Testing a bunch of pixels underneath a circle may take no more than 10ms, but performing a sunlight analysis on a specific window shape for every daylight hour for every day of the year could easily take seconds if not minutes. The solver must find high-ground in the landscape before the passage of the time renders the problem irrelevant.

The remainder of this paper will discuss four common generic algorithms in order of complexity while highlighting their strengths and weaknesses against the backdrop of different landscape geometries and topologies.

3.1 Divide and conquer

Divide-and-conquer (DC) is the name for an entire paradigm of algorithms, both specific and generic ones. The core idea behind DC is to break a problem into smaller and smaller pieces until the fragments are small enough to be solved directly (for example using a brute force algorithm). An implementation of DC within the context of peak finding might work as follows:

1. Define a domain $\mathcal{D}_i$ for the search. When the algorithm begins, $\mathcal{D}_i$ will be identical to the boundaries of $\mathcal{P}$.
2. Sample the tensors at the vertices v of a grid $\mathcal{G}_i$ within $\mathcal{D}_i$.
3. Find the highest quad(s) within the grid.
4. Shrink $\mathcal{D}_i$ to contain only these quads. This may involve splitting $\mathcal{D}_i$ into one or more disjoint $\mathcal{D}_{i+1}$ regions.
5. Repeat (1) through (4) until:
   (a) a solution with an acceptable (predefined) fitness is found,
   (b) or $\mathcal{D}_i$ becomes so small as to make further subdivisions pointless.

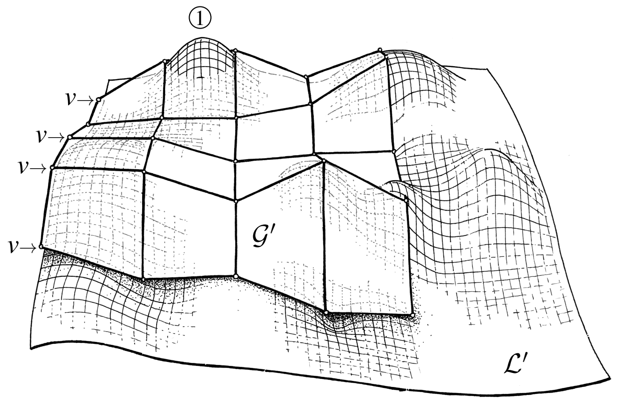

Figure 4: Sampling grid on $\mathcal{L}$

Such an algorithm is easy to implement, but it does rely on a number of assumptions regarding $\mathcal{L}$ that will not be true for all cases. First of all the algorithm treats the sampling grid $\mathcal{G}$ as a proxy for $\mathcal{L}$ at every iteration. If $\mathcal{L}$ contains relevant details that are smaller than the spacing between adjacent samples, they may well go overlooked. Narrow peaks with a small footprint are particularly likely to go unnoticed, as ① in figure 4 shows. To rephrase that in a way that points the accusing finger at the algorithm instead of the landscape; DC assumes that $\mathcal{L}$ is uninteresting over short distances and can thus be reliably approximated with a low density grid.

The extent of a peak, usually called its *basin of attraction*, can be described as the area from which one can reach the peak while only walking straight uphill. The height of a peak and the extent of its basin are completely unrelated properties, and quite often peaks representing low quality solutions have larger basins than peaks representing high quality solutions.

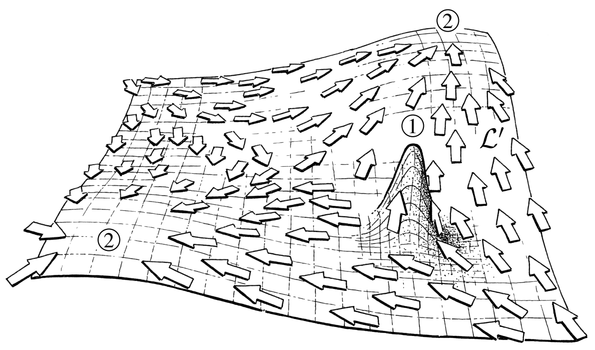

Figure 5: Basins of attraction

The above figure shows a landscape with three peaks, where ① represents a high quality solution, while the remaining ones marked ② represent low quality solutions. The arrows show the steepest way up from a sampling of points across $\mathcal{L}'$. As is now visible, the vast majority of arrows lead to low quality peaks.

Another problem with DC is that it does not scale well to higher dimensions. A sampling grid accuracy needs to be picked for every iteration, and this accuracy needs to be tight enough to not miss significant landscape features. The number of actual fitness samples that needs to be taken for a single DC iteration[11] is defined as the product of the number of samples for every tensor element. For a sampling density of ten steps per tensor element in $\mathcal{P}^4$, a single iteration requires $10^4 = 10,000$ samples. At half a minute per sample for something as computationally intense as for example a daylight analysis, the first DC iteration will have a runtime of about three and a half days.

Yet for low-dimensional problems DC is often a good approach and it is certainly one of the more popular search algorithms out there. A straight-forward implementation of DC can be categorized as greedy, deterministic, approximate, and progressive (see page 2 for an explanation of these categories). Since the algorithm quickly discards large areas in the landscape as it shrinks the search domain, it is not adaptive.

3.2 Hill climbing

While divide-and-conquer at least attempts to find the best solution, a hill climbing algorithm is designed to only take local conditions into consideration. Starting from an arbitrary location on the landscape it will try and walk uphill as fast it can. It achieves this by sampling the tensors adjacent to the current position to see which one is the fittest and then moving in that direction until it finds a peak, or until something bad happens. A simple implementation of a hill climber might work as follows:

1. From a location τ_i, sample adjacent tensors in all meaningful directions. Remember τ_j as the neighbour with the highest fitness.
2. Define a travelling vector $\vec{v} = \tau_j - \tau_i$, and multiply by a scalar S representing the step size.
3. Take a step in $\mathcal{P}$, defined as $\tau_{i+1} = \tau_i + \vec{v}$.
4. Evaluate the fitness q_{i+1} at τ_{i+1}.
5. Repeat (3) and (4) until:
   (a) q_{i+1} is less than q_i,
   (b) or until a boundary of $\mathcal{P}$ is reached.
6. Repeat (1) through (5) until:
   (a) a solution with an acceptable (predefined) fitness is found,
   (b) or there are no adjacent tensors with higher fitness than the current one.

[11] At least assuming that a search domain does not split into multiple subdomains, in which case more samples are required still.

To put that in plain English; pick a direction that seems to go uphill the steepest, then keep walking in that direction until you start walking downhill, or until you can walk no further. Then pick a new best direction and start walking again.

Hill climbers are greedy algorithms that tend to find the peak whose basin of attraction they start in. This behaviour isn't necessarily a drawback, but it does mean that using hill climbers to find the best possible solution(s) is only possible if $\mathcal{L}$ doesn't have too many local optima. Bumpy landscapes such as shown in figure 2 are unsuitable for a hill climbing approach. In extreme cases $\mathcal{L}$ can start to exhibit fractal properties (bumps upon bumps upon bumps). Since most algorithms depend on at least a small level of continuity in $\mathcal{L}$, such landscapes often defy navigation algorithms, but hill climbers in particular are vulnerable to such geometry.

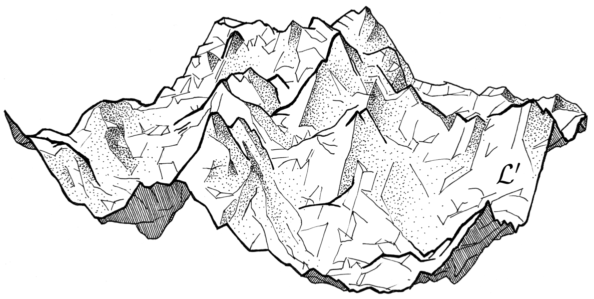

Figure 6: Landscape with fractal properties

Hill climbers are significantly more difficult to implement than divide-and-conquer algorithms, mostly because there is a lot of fine-tuning involved. Although in theory adjacent tensors can be evaluated to get the local *slope* or *gradient* of $\mathcal{L}$, it is sometimes better to pick tensors that are not directly adjacent but a bit further away. Short distance sampling is liable to suffer from tiny amounts of numeric noise that are inherent in $f(\tau)$, but how far away should a sample be taken? It depends on the nature of $\mathcal{L}$, which the algorithm is not supposed to know anything about.

Another fine-tuning problem arises with the selection of a step size S. The best sampling radius for the gradient approximation is unlikely to be the same number as the ideal distance for subsequent steps. But what is the ideal value for S? Again, it depends on the geometry of $\mathcal{L}$.

It is also reasonable to expect S to decrease over the course of the search. While running towards a far away peak, it makes sense to take long strides in order to get there quicker. But near a peak big leaps will probably just overstep the summit and make things worse. But how sharply should S decline over time? Nobody knows.

In addition to the problems mentioned above, which are at least in principle solvable, hill climbers suffer from poor scalability to higher dimensional spaces. It is computationally cheap to walk across a landscape, but as the dimensionality of $\mathcal{P}$ increases it becomes more and more expensive to determine the local gradient[12]. One would need at least two samples in opposite directions for every principle axis in $\mathcal{P}$ [13].

When discontinuities in $\mathcal{L}$ are rare (even if they are large), they should not interfere too much with a hill climbing approach, as there are still plenty of connected areas where a sequence of progressive steps can be taken.

[12]Measuring the local gradient is only expensive if it has to be *inferred* by sampling the immediate surroundings. If a derivative fitness function $f'(\tau)$ is known then the search would scale very well to higher dimensions. However, a lot of fitness landscapes are not readily differentiable.

[13]This is true for tensors on the interior of $\mathcal{P}$ only. Locations along the boundaries and edges of $\mathcal{P}$ require fewer samples.

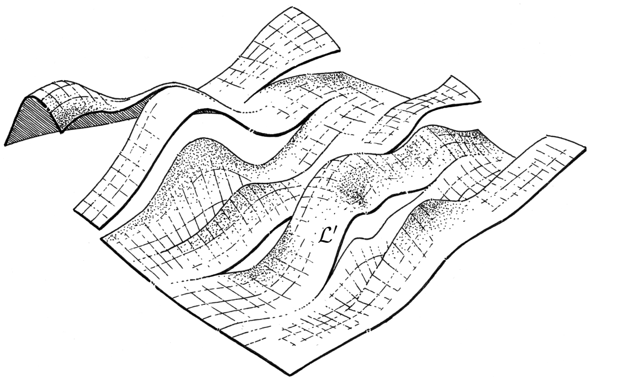

Figure 7: Landscape with few discontinuities

Hill climbing algorithms can be categorized as greedy, deterministic, exact (though only in a *local* optimum sort of way), and progressive. It is possible to introduce a certain amount of randomness and turn the categorization away from the deterministic and towards the stochastic. Instead of sampling tensors parallel to phase space axes, a randomized sampling could be employed. One benefit of such a change is that it may reduce the number of turns required, especially in the case of ridges that exist at angles to tensor axes in $\mathcal{P}$.

3.3 Simulated Annealing

The previous two algorithms may not be simple in their details, but they mimic the way humans would solve problems and it is therefore easy to see how they might work. *Simulated annealing* is not like that and it appears to follow some very unusual and suspect rules. ‘Annealing’ is a term that comes from a treatment process in metallurgy which is closely related to crystallization.

The atoms in most metals and alloys like to sit on a regular lattice called a *crystal*. Atoms which are aligned with the lattice exist in a *minimum energy state*. However in the real world, a sample of metal will contain dislocations and cracks and adjacent regions of differently oriented atoms. These departures from a perfect crystal are called *defects*. By heating up the sample, one can weaken the bonds between atoms which enables them to randomly jump around a bit until they find a lower energy state than they had before. As the sample cools down, atoms will tend to settle in these new states and the sample will end up with larger regions of pristinely crystallized atoms and fewer defects.

The analogy with *simulated* annealing as a problem solving approach is somewhat weak, but the mathematics are very similar. A simulated annealing solver takes the thermodynamic equations that describe metallic annealing and applies them to tensor movement. As a result the algorithm is highly stochastic, fairly progressive, somewhere in between exact and approximate (depending on the complexity of $\mathcal{L}$), and slowly transitions from global to greedy during the progression of each search.

A basic simulated annealing solver is actually remarkably easy to implement and could work as follows:

1. Define a temperature $\mathcal{K}$. At the start of the search $\mathcal{K}$ should be high.
2. From the location of τ_i, randomly pick a τ_{i+1} somewhere within a maximum radius d, where d is covariant with $\mathcal{K}$. The higher the temperature, the larger the distance.
3. Evaluate the fitness q_{i+1} at τ_{i+1}.
   (a) If $q_{i+1} > q_i$ then move to τ_{i+1}.
   (b) If $q_{i+1} < q_i$, then maybe *still* move to τ_{i+1}, depending on the difference in fitness (Δq) between q_i and q_{i+1} and $\mathcal{K}$. The higher the temperature and the smaller the difference in fitness, the more likely a move to a less fit tensor will be accepted.
4. Redefine $\mathcal{K}$ as a percentage of its current value, simulating a cooling environment.
5. Repeat (2) to (4) until:
   (a) a solution with an acceptable (predefined) fitness is found,
   (b) or until $\mathcal{K}$ drops below a predefined limit.

The surprising feature of simulated annealing is of course that it will sometimes deliberately move to a worse solution. The benefit of being willing to make things worse is that it sometimes allows one to break out of a local optimum and find higher ground that is more than one move away. Since $\mathcal{K}$ is reduced during every iteration though, the algorithm becomes less and less willing over time to accept a move to a lower quality tensor. One can imagine simulated annealing as two solvers wrapped into a single package. At first the search includes all of $\mathcal{P}$ and the algorithm is clearly global minded. It is during this phase that high ground is sought. Then as $\mathcal{K}$ gets lower and lower the algorithm instead starts behaving like a greedy stochastic sampler, only willing to climb the local peak.

There are very few drawbacks to simulated annealing and the things that do plague it tend to wreak havoc no matter what generic solver you apply. In particular high peaks with small basins of attraction as shown in figure 5 can easily be missed by the random nature of the sampler. On the other hand simulated annealing can deal very well with discontinuous and fractal terrain.

The problem I would like to discuss in this context is not associated with simulated annealing solvers but rather with poor implementations of $f(\tau)$. It is entirely possible to write a fitness function that yields identical fitness values for adjacent tensors. This will result in flat areas or *plateaus* in $\mathcal{L}$ as shown in figure 8.

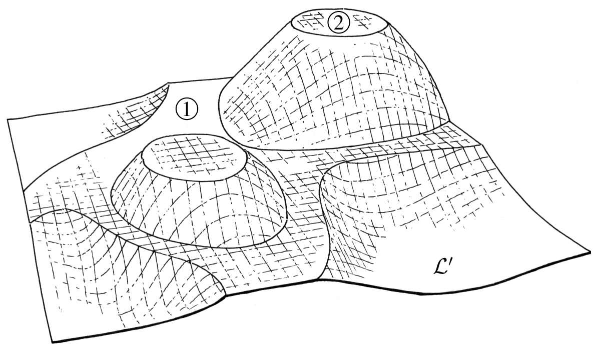

Figure 8: Geometry of under-constrainedness

Plateaus in fitness landscapes are a manifestation of *under-constrained* fitness functions. Let us repurpose the painting-a-dot example discussed on page 3, but instead of the *Mona Lisa*, this time the painting in question is *Who's Afraid of Red, Yellow and Blue*. In this new case, a lot of red dots can clearly be painted which are all equally fit in the sense that they do not change the appearance of the painting in any way. In other words, the 'peak' in the fitness landscape defined by this $\mathcal{P}$ and this $f(\tau)$ is more like a *mesa* than a summit.

Plateaus in the low or medium ground of a fitness landscape may well retard the progress of many solvers as it becomes very difficult to decide in which direction it is smart to move. But plateaus in the high ground of a landscape, specifically if they are the highest features, may result in different yet equally valid solutions. This is rarely what people are after but luckily it is often possible to add additional terms to $f(\tau)$ which introduce a gradient to $\mathcal{L}$ so that perfectly horizontal plateaus become ever so slightly tilted or curved.

Note that a plateau is defined as two or more adjacent tensors with identical fitness. This means that the dimensionality of plateaus can be any number up to and including the dimensionality of $\mathcal{P}$ as shown in figure 9.

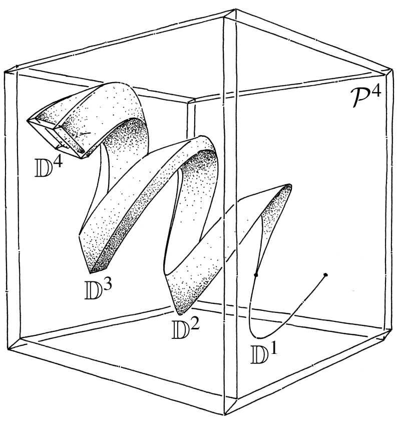

Figure 9: Plateaus of different dimensionality in a 4-dimensional phase space.

3.4 Simulated Evolution

Evolutionary solvers and *genetic algorithms* have long been popular amongst the computationally minded and there are many different ways to implement them. The ideas behind such algorithms are derived from biological evolution and its two main components of mutation and selection. Evolution is nature's answer to systematic problem solving, but apart from being very successful, it is also rather slow. In nature that tends not to matter too much because of the availability of massive parallel processing (every atom simulates itself in real-time), but when simulating the process digitally, the performance can become prohibitively slow.

The core idea of both biological and simulated evolution is the *heritability of traits*. Biologically, traits are phenotypic expressions of allele-complexes and the fitness of a specific complex is measured by its frequency in the gene-pool over time. Things are much simpler in a simulated evolution environment as the fitness is not an intrinsic property that is manifested statistically over several generations, but rather it is defined directly by $f(\tau)$. The analogy between the biological terms and mathematical symbols can be summarised as follows:

- Population $\longrightarrow \widehat{\mathcal{X}}$
- Generation $\longrightarrow \mathcal{X}_i$
- Genome $\longrightarrow \tau$ (tensor)
- Gene $\longrightarrow \tau_i$ (tensor element)
- Allele $\longrightarrow |\tau_i|$ (element value)

A typical evolutionary solver is an iterative algorithm which maintains a *population* $\widehat{\mathcal{X}}$ of individuals and selectively culls and recombines them to form successive generations $\mathcal{X}_i$, $\mathcal{X}_{i+1}$, $\mathcal{X}_{i+2}$, etc. The point is to only use the fittest individuals to form the next generation in order to select from and expand upon the best genes from a randomly generated gene-pool. The founder generation $\mathcal{X}_1$ can either be distributed randomly across all of $\mathcal{P}$, or it could be limited to a smaller patch in order to explore local optima. Not all generations need have the same number of individuals. Especially $\mathcal{X}_1$ may benefit from being much larger than subsequent generations, as a dense sampling of $\mathcal{L}$ will reduce the chances that solutions with small basins of attraction will go unnoticed. The entire process can be described as:

1. Populate $\mathcal{P}$ with a founder generation $\mathcal{X}_1$ of randomly picked genomes.
2. Evaluate each genome in $\mathcal{X}_i$ using $f(\tau)$ and create a hierarchy $\mathcal{H}_i$ based on their fitness distribution.
3. Combine genomes from $\mathcal{X}_i$ based on $\mathcal{H}_i$ to create offspring genomes for $\mathcal{X}_{i+1}$.
4. Repeat (2) and (3) until:
   (a) a genome with an acceptable (predefined) fitness is found,
   (b) or $\mathcal{X}_i$ fails to significantly improve upon $\mathcal{X}_{i-1}$, $\mathcal{X}_{i-2}$, ...

But succinct as the basic approach may be, there are a lot of details left unsaid and a lot of decisions left unmade. Item (3) especially hides a lot of complexity. The purpose of iteration in this algorithm is to evolve a population of tensors that occupy high peaks in $\mathcal{L}$. For this to work, the fittest tensors in generation $\mathcal{X}_i$ have to combine to produce even fitter tensors in generation $\mathcal{X}_{i+1}$, but a lot can go wrong in this process.

Offspring from nearly coincident parents will not contribute significantly to the representation of $\mathcal{L}$ that the algorithm builds over time, since that particular region has already been sampled. Measuring the fitness of highly related tensors very quickly suffers from diminishing returns. On the other hand, parents which are too far apart may well belong to different *sub-species*, each in the process of climbing a different peak. Since offspring will most likely fall somewhere in between the parents in $\mathcal{P}$, they are likely to end up in a valley, as valleys tend to separate peaks in a landscape. These two mating extremes are called *incestuous* and *zoophilic* respectively and they are as detrimental in nature as they are in simulations.

Ideally one would select parents that are positioned on opposite sides of the same peak. But even in the ideal case there are still more decisions to be made regarding the combination of two parent genomes to produce offspring. This process is called *coalescence* and several approaches exist.

Coalescence where some elements are copied intact from τ_1 and others from τ_2 most resembles the biological process of fertilization where gametes combine to form a new genome. This kind of merging only works well when τ contains many elements:

$$\begin{gathered}\tau_1\{\ \underline{3,8},1,17\ \}\\ \tau_2\{\ 5,2,\underline{9,25}\ \}\\ \downarrow\\ \tau_3\{\ 3,8,9,25\ \}\end{gathered}$$

Coalescence where alleles are interpolated between τ_1 and τ_2 treats the genes as smoothly varying properties. In this sense the tensor elements do not so much resemble different alleles as entire allele-complexes. The interpolation need not always be halfway, parental genomes can be weighted either randomly, or based on Δq:

$$\begin{gathered}\tau_1\{\ \underline{3,8,1,17}\ \}\\ \tau_2\{\ \underline{5,2,9,25}\ \}\\ \downarrow\\ \tau_3\{\ 4,5,5,21\ \}\end{gathered}$$

Finally, there's no reason why (in a simulation) offspring requires two parents. One could instead take all τs in $\mathcal{X}_i$ into consideration during coalescence, or perhaps simulate asexual reproduction.

In addition to breeding selection, partner selection and coalescence algorithms, an evolutionary solver should also provide ways for introducing mutations into a genome. Again, there are many ways to approach this problem, whether one wants to treat genes as independent or covariant, whether mutations are associated with gene blending due to coalescence, whether mutations are dependent on the uniformity of all tensors in the (sub)species, whether mutations are in some way related to $\mathcal{H}_i$ or even the recent history of $\mathcal{H}$ over several generations, and so on and so forth.

Like all peak finding algorithms, evolutionary solvers perform better on certain landscape geometries. As with all solvers, peaks with small basins of attraction are easily missed and plateaus are crossed slowly. Unlike many other algorithms however, evolutionary solvers are capable of exploring multiple peaks simultaneously by populating each peak with its own sub-species. As a result evolutionary solvers are decidedly non-greedy, reasonably adaptive (especially if $\mathcal{L}$ changes *slowly* over time), particularly open, and very stochastic.

Unlike hill climbers and simulated annealing, evolutionary solvers don't really 'walk' across the landscape in search of peaks. Instead they act more like expanding and thinning clouds of tensor particles. The velocity of the cloud front is typically quite low and it is easily deflected by obstructions such as plateaus.

While plateaus, narrow peaks and fractal terrain are all geometric properties, *over-constrainedness* causes *topological* defects in fitness landscapes. When a fitness function is over-constrained, it becomes impossible to evaluate certain states of τ, and as a result $\mathcal{L}$ will contain gaps. Depending on the size and connectivity of these gaps, and depending on the solver in question, this may or may not be a problem. Even solvers that seem to move across $\mathcal{L}$ in a continuous fashion do not actually travel from τ_i to τ_{i+1} in a smooth fashion, they jump instantaneously from one to the other. If a piece of $\mathcal{L}$ is missing in between τ_i and τ_{i+1} nobody will be the wiser. But as gaps grow bigger, so do the chances of stepping in one.

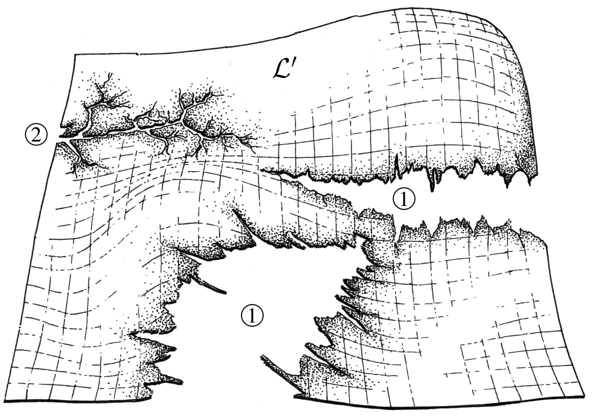

Figure 10: Geometry of over-constrainedness

There are two important gap metrics in dealing with over-constrained landscapes: *area* and *circumference*. Gaps covering large portions of $\mathcal{L}$ will delay or even halt a solver as it is unable to cross over to the other side. Small gaps on the other hand may go completely unnoticed as the solver never attempts to sample an over-constrained tensor. The lower limit for the circumference of a gap is at least equal to the circumference of a circle of equal area[14], but there is no upper limit. The boundary of a gap can exhibit fractal properties which can trap algorithms such as hill climbers and evolutionary solvers into ever narrowing tendrils of $\mathcal{L}$, see ① in figure 10. This sort of topology acts as an over-abundance of local optima. However if the area is small —even in the case of fractal boundaries— then the gap can go unnoticed as it will tend to fall between successive samplings of $\mathcal{L}$, see ② in figure 10. Simulated annealing in particular is adept at dealing with over-constrained problems, as it often traverses large distances and can thus easily span gaps that cover significant percentages of $\mathcal{L}$.

[14]Technically the minimal circumference (as measured in $\mathcal{P}$, *not* $\mathcal{L}$) is equal to the area of a hyper-sphere with the same volume and dimensionality as the gap.

4. CONCLUSIONS

This paper discussed the fundamentals of generic solvers. Although a detailed understanding of such algorithms is not a prerequisite for their application, potential users should at least familiarise themselves with the notions of phase-spaces, fitness functions and fitness landscapes, as defining these aspects is the responsibility of the user. It is important to realise that generic solvers are stochastic rather than analytic processes, and that they may take a prohibitively long time to run, depending on the specifics of the problem at hand.

ACKNOWLEDGEMENTS

This paper was written in $\LaTeX\,2_\varepsilon$, proofread by $\mathbb{K}$, and funded by Robert McNeel & Associates. Thanks to all.

ABOUT THE AUTHOR

David Rutten is a graduate of the faculty of Architecture and Urbanism at TUDelft. Since 2006 he has been employed by the *Andrew leBihan partnership* in *Turku, Finland*, and, more recently, by *Robert McNeel & Associates* in *Seattle, WA* where his main task is the continued development of and support for the Grasshopper® plug-in for Rhinoceros 3D®. Grasshopper is a visual programming environment aimed at those who wish to partake in computational design.