

# OPTIMISED RANDOM MUTATIONS FOR EVOLUTIONARY ALGORITHMS

Sean McGerty and Frank Moisiadis

University of Notre Dame , Australia

## **ABSTRACT**

*To demonstrate our approaches we will use Sudoku puzzles, which are an excellent test bed for evolutionary algorithms. The puzzles are accessible enough for people to enjoy. However the more complex puzzles require thousands of iterations before an evolutionary algorithm finds a solution. If we were attempting to compare evolutionary algorithms we could count their iterations to solution as an indicator of relative efficiency. Evolutionary algorithms however include a process of random mutation for solution candidates. We will show that by improving the random mutation behaviours we were able to solve problems with minimal evolutionary optimisation. Experiments demonstrated the random mutation was at times more effective at solving the harder problems than the evolutionary algorithms. This implies that the quality of random mutation may have a significant impact on the performance of evolutionary algorithms with Sudoku puzzles. Additionally this random mutation may hold promise for reuse in hybrid evolutionary algorithm behaviours.*

## **KEYWORDS**

*Attention, adaption, artificial intelligence, evolution, exploitation, exploration, satisficing, Sudoku, particle swarm, genetic algorithm, simulated annealing, mutation.*

## **1. INTRODUCTION**

Evolutionary algorithms attempt to iteratively improve a population of candidate solutions. Each solution is randomly mutated. Random mutations are applied to each solution, and a fitness function is used to assess if an improvement has occurred. Evolutionary algorithms may then attempt to replicate attributes of the more successful candidates to the others. In this way weaker solutions become more like the better solutions and the cycle continues. This behaviour can be seen in both particle swarm optimisation and genetic algorithm heuristics [1] [2].

The optimisation in this approach can be seen as an accumulating behaviour for solution candidates around optimal points in the namespace. The forces of random mutation and fitness function assessment bring more candidates close to the best solution found so far. Diversification within the candidate population is being transferred into specificity. This accumulation of candidates can be seen as an exploitation strategy, which needs balancing against exploration [3][4]. We can describe non-optimal behaviours in evolutionary algorithms in these terms [5].

At higher levels of namespace complexity is inefficient to scan every possible candidate to know definitively which is the best. Doing so would be the ultimate in exploration, and in relational terms very little optimisation exploitation would be occurring. The strength in a heuristic is in the expectation of being ample to find a good solution and potentially the best solution without checking all possible solutions.

## 2. EXPLORATION VS. EXPLOITATION

Evolutionary algorithms face the same problems that people often do. Should we continue to try and solve a problem from where we are at the moment? Or should we diversify in case we are not making enough progress in the hope that there are better opportunities elsewhere?

The reality with most evolutionary algorithms is that they will only support one of these modes. For the most part evolutionary algorithms have their random mutation options for exploration inline with the rest of the optimisation. This means that the optimisation has to be held in balance with optimisation. The act of sharing successful attributes makes the candidates look more similar, while the actions of random mutation push them further apart. If we take the assumption that we are working towards an achievable solution in a logical way then the exploitative action of optimisation will have to overpower the explorative desire of randomisation to move apart.

This balance may work well in most cases. However where there is little change occurring because we have reached a local maximum, we have a problem. Those forces making the candidates exploit the incorrect but best solution so far hamper the ability of the randomisation to escape and find other better solutions.

Strengths in exploitation may lead to weaknesses in exploration. By replicating attributes among the solution candidates it is entirely possible that they may accumulate around a local maximum. In this case the desire to exploit has overpowered the entropy of the randomisation function, which now lacks the ability to break from the local maximum.

At this point the algorithm may relatively prioritise a repulsion factor between candidate neighbours [6]. The algorithm may de-prioritise the optimisation component allowing more random mutation. In either case the algorithm requires awareness that relative improvement is no longer occurring. There is also the question of how to parameterise these exploration modes, preferably in a non-namespace specific way.

This hybrid behaviour between exploration and exploitation is also seen when different evolutionary algorithms are combined [5]. If we compare particle swarm optimisation and simulated annealing we might consider particle swarm optimisation to be a relatively strong exploiter [7]. In the same terms simulated annealing may rely more on random mutation and therefore be a relatively strong explorer. If our implementation allowed each algorithm to share the same solution candidate population, then we would be able to swap between the two algorithms as needed. We would then be able to rebalance between exploitation and exploration at will.

The ideal partner for a normal evolutionary algorithm is therefore a randomisation algorithm that we optimised in such a way to preferentially find better solution candidates without traditional optimisation.

## 3. BROAD BASED ATTENTION

We draw many similarities between human vision and evolutionary algorithms. Much of the processing power at the back of your eye is devoted to peripheral vision. The right hemisphere of your brain is most likely dedicated to qualitative processing and broad based attention.

In these modes coverage and efficiency of operation appear to be primary concerns. Your peripheral vision is optimised to detect unexpected motion and changes in light intensity. This allows the majority of your left hemisphere and the central aspects of your vision to focus on specific tasks while not losing the bigger picture. If nothing else, consider it a survival mechanism where autonomic processing can save you while you think about something else.

We can achieve many of the same goals in an heuristic if we first notice that simulated annealing's optimisation modes are a bit different from the others. Rather than replicate attributes from solution candidates with better fitness function scores to weaker ones, simulated annealing has a random mutation step that discards it if the result is a net loss. This is an example of being able to direct changes in a beneficial way. You can also see that not needing to select or correlate solution candidates might have efficiencies over normal processing modes. Efficiency is the main consideration for an exploration mode broad-based attention agent. The most comprehensive mechanism of this type would be scanning every possibility in the namespace, but as we mentioned earlier this rapidly becomes unworkable for large namespaces.

A broad-based attention algorithm expects that we can disperse candidates through a data namespace and in so doing gain a better view. Each solution candidate is a mapping between causal input variables and a resulting fitness scalar. By varying these inputs as much as possible we gain a broader view of the distribution of this fitness curve.

Note too that the simplex algorithm is a use case for a select type of problem that evolutionary algorithms would be able to solve. The simplex algorithm understands that the best values will be the boundary values of one or more variables. This then leads to checks where correlated variables are set to boundary values and transitions between these combinations will maximise the fitness function. In a topographical sense we navigate the boundary of an n-dimensional object for a corner value we prefer.

In the same way we could, for example, recognise that we could add more values than we remove during random mutation. This is similar to saying that we expect a solution to be more likely with cells added than removed, and as we are solving Sudoku puzzles this is the case.

Using these ideas we will create a population of randomly mutating solution candidates that will move about sampling the namespace in a directed way. With optimising mechanisms these candidates will disperse, giving an aggregated view of a subsection of the problem. Note that we should be able to direct this mutator towards more interesting parts of the namespace without using evolutionary algorithm style optimisations. Visualise this as being more interested in the surface of a bubble than the air inside. We are beginning to make a case that there is benefit to thinking of random mutation as having a lifecycle.

We can also optimise the fitness function to our needs. We will always need to check that any solution is valid and consistent. Also if we accept that we may find an endpoint solution through random change then we want to know if we have reached the endpoint solution. We do not need the fitness scalar for comparing the solution candidates within this population, as we do not optimise by exchanging attributes. We are interested in the scalar sometimes, but only when we are interested to know if we have found a better solution via random change if we are running a separate random mutation solution candidate population. For the most part though, our processing for broad attention modes is simplified with respect to optimisation.

Take for example Sudoku puzzles, which are completed when all cells have been filled [8]. We are restricted to adding digits such that each digit can only occur once with in a row or column or

a region. We can separate these considerations: we are attempting to add digits, and solutions cannot be invalid.

The check for validity of the solution is a subset of the fitness function. Rather than returning a fitness score we can simply return true or false. We can combine our simplified fitness function with a random mutation agent with a bias for addition. We call this mechanism the greedy random.

Solution candidates are spread through the namespace by the greedy random. This behaviour attempts to fill as many cells as possible. If we correlate to human vision, the greedy random moves around the boundaries of what a human can see flagging changes. Note as well that the greedy random uses less resources as a subset of an evolutionary algorithm, so we can run more of them with less effort.

#### **4. ATTENTION ADAPTION**

Darwin was asked what he considered to be the most important attribute for continued success in his model of evolution. He avoided factors like strength, or speed, and instead suggested it was far more important to be able to adapt [9]. When an evolutionary algorithm collects around a local maximum we could see this specificity as a failure to adapt. Any candidate undergoing random mutation does not have the entropy to produce a candidate better than the current population. In these cases these insufficiently adapted mutations are removed or assimilated. We suggest that is need is a mechanism for being aware of candidates sufficiently outside the local maximum to allow us to adapt and escape.

Think of this as an attention mechanism, which allows the adaptation away from the local maximum to occur. By implementing this ability we gain understanding of a mechanism that has been known to plant sciences for most of a century. It is entirely possible to be able to separate changes into those based on internal factors from those that occur in response to their environment.

By being able to notice beneficial change in candidates undergoing random mutation we can adopt that change, even when it is outside the realm of experience for the evolutionary algorithm.

#### **5. FITNESS FUNCTION**

Evolutionary algorithms work by attempting to maximise a scalar fitness function by changing values within constraints [10]. For example we may attempt to maximise the sum of two numbers greater than zero but less than five. The constraints place acceptable input values from one to four. We may start the process with values of one, and a fitness of two. Over time we randomly change values and remember those pairs that lead to improved fitness function scalars. Eventually as a result of random changes and using the best candidates so far as a reference, our paired values improve. Eventually we reach a stable solution with four and four equalling eight, and we no longer see improvement with any random change.

A random mutation component capable of integrating with evolutionary heuristics will need to inter-operate with this evolutionary algorithm lifecycle. As previously mentioned we do not include optimisation, but the question remains how much of the fitness remains relevant to random change function.

At its simplest the fitness function returns a scalar value, which increases as our solution candidate improves if we are attempting maximisation. However there is also an expectation that candidate solutions that violate the problem constraints are invalid. In this case the fitness function may return a value equal to or less than zero to mark that this candidate is less valuable than any other current candidate. If we were to realise that we had produced an invalid candidate we could then choose to discard it, or to revert to the most recent valid version.

If we are not optimising, then we are not necessarily comparing candidates by fitness. It remains a serious issue if a candidate should receive random mutations that render it invalid. Therefore we still have interest in a subset of the fitness function outcomes. The assumption is that in most cases it should require less processing to validate a candidate than produce the complete fitness scalar.

## 6. RANDOM MUTATION

We have used the phrase random mutation, however not all random changes have equal effect [11]. If our fitness function is linear then we may prefer boundary conditions to our input values. If we are attempting to fill a region we may prefer adding cells than removing them. In any case, we have an opportunity to favour some types of random changes over others.

This leads to the idea of the greedy random. The greedy random understands in general terms that either setting values or removing values is preferential to the fitness function. For example we may set a probability to add as 0.8, and a probability to remove at 0.2. In this case before performing an operation we first choose whether we are in addition or removal mode. The net effect of this bias is to produce candidate solutions with as many added cells as possible. We call this process the greedy random because of this perspective of attempting to fill as many cells as it can before removal.

In this case the relative fitness function assessment of any candidate is less important than knowing if the candidate remains valid. So we can perform these greedy operations more efficiently as a result. In testing, this represented an opportunity for more random mutation cycles to each evolutionary algorithm cycle.

The risk of course is that a candidate solution may rapidly fill and lose degrees of freedom. This problem replicates the issue experienced by evolutionary algorithms around a local maximum. This was an important design consideration during testing. The solution to this problem became apparent during attempts to integrate with the evolutionary algorithm lifecycle. For iterations, the candidates in the population are assessed by the fitness function. In the case of attempting to maximise the process of filling a board such as Sudoku puzzle, performance was greatly improved by ensuring that the last change before fitness function assessment was a removal.

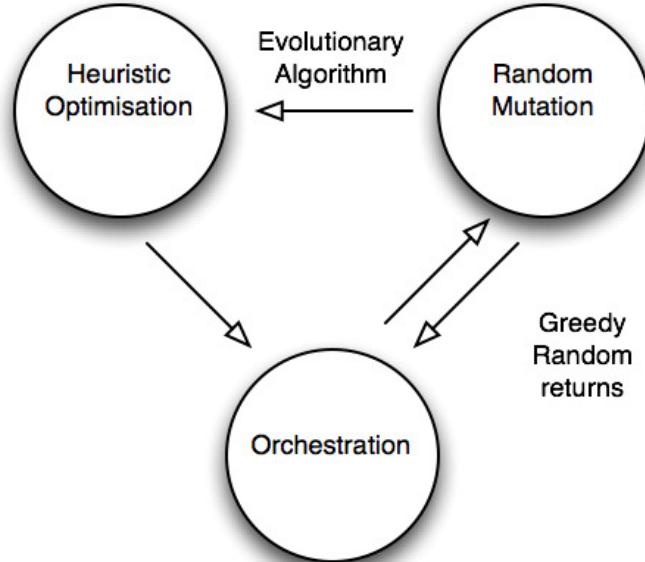


Figure 1. The modified heuristic iteration lifecycle

We realised a further implication of the fitness function lifecycle. In one respect the fitness function tells us when we have reached a global maximum. In the case of a Sudoku puzzle we may have a valid candidate with all the cells occupied by digits. If not there is a subtle difference between asking which candidate is the best, and which candidate has the best chance of improving with more random mutations. It seems that a strong candidate with additional degrees of freedom can be as valuable as a stronger candidate with more cells filled.

It is important to check fitness with as many cells as possible filled in order to find completed solutions. However if we are attempting to measure potential for improvement in a process where we are filling as many cells as possible, testing showed it was more meaningful to rank the candidates after a single removal. Doing so concentrates the random mutation entropy around the boundary conditions of the solution. Once again we have a collection pressure, but while evolutionary heuristics concentrate around local maxima the greedy random collects the candidates around input boundary values.

We can also better conform to the problem namespace by prioritising changes with fewer degrees of freedom. In this way additions are validated against the cells that are already filled in this solution candidate. If we were to choose a more empty part of the namespace we could choose from more values for a cell, however we may be introducing a combinatorial issue with later additions.

Therefore we reduce rework by using a fitness function that can be thought of as the count of neighbours that each filled cell has. For Sudoku we are checking each row, cell and region, so we are looking for  $8 \times 8 \times 3 \times 9 = 2781$  as the score for a solved board and 0 for an empty one.

## 7. COMPARING DIFFERENT EVOLUTIONARY ALGORITHMS

Consider the situation in which we are attempting to evaluate the suitability of different evolutionary algorithms for the same problem. If we were simply reusing algorithms then we would find a way of encoding the problem in a format that each algorithm could recognise. This would give us a time to completion for each of the algorithms, but we would not be able to

differentiate the performance factors within each algorithmic lifecycle. This would leave us less capable of classifying performance by problem type, and less capable of predicting performance with other problems in the future. Finally we are also at the mercy of the quality of the implementation of each component. What might be a specialised optimisation for a given example of a problem may be suboptimal in the more general case. We argue that when comparing the performance of evolutionary algorithms we need to separate those components that are less related to optimisation and more likely to be shared between implementations.

We would argue that a fitness function that is better suited to the namespace topology has better alignment to the data than the optimisation. In this case it would make more sense to rework the fitness function to each problem data type than to attempt to re-use the fitness function between different problem types. If we separate these shared functions from the optimisation then we can better evaluate the efficiency of the optimisations in isolation. All we need to do is create an encapsulating lifecycle, which accepts differentiated optimisation components in the data management, and fitness functions should be reusable.

In the same way that we have managed to isolate the fitness function from the optimisations we can also isolate random mutation. As mentioned the reasons for differentiating random mutation are less obvious. When we look at randomisation it soon becomes apparent that not all random variations are the same. We have stated that in the middle time period of solving the problem it may make sense to add as many cells as we remove during random mutation. However as we fill more of the board we encounter reduced degrees of freedom and so more cell additions will fail consistency checks. In effect it will be more successful removing cells than adding cells and random mutation may be detrimental to optimisation in that case. To remediate we may decide to bias in favour of cell additions rather than cell deletions, or we may retry additions until successful.

If we follow this path we introduce another anti-pattern in that we may leave the solution candidates with no degrees of freedom entering the optimisation component processing. During experimentation we had greater success when we left deletion steps at the end of the process. We expect there are two factors for the observed behaviours. Firstly it may be preferential to leave a degree of combinatorial flexibility for the process of attribute replication during optimisation to occur. Secondly the question arises of the optimal time to evaluate fitness in the optimisation lifecycle.

If we accept that producing a scalar for comparison and evaluating the possibility of an end point solution are different questions then we open the possibility that it may make sense to check for these conditions at different times in the life-cycle. Consider waves at a beach. Our endpoint condition may be a wave reaching a distant point along the shore. However the fitness of any wave might be better measured by the height of the swell before the wave approaches the beach as an indicator of future success. In these terms the fitness is the height of the swell and the endpoint condition is achieving the required distance up the beach as a Boolean consistent with a causal relationship. Increase the swell and the waves drive further up the beach. So if we are attempting to solve a Sudoku problem then it may be more valuable to rank candidate solutions with better fitness and degrees of freedom than fitness alone.

In any case if we are complicating the random mutation, particularly if we are doing so to suit conditions in the data namespace, then it also makes sense to separate the random mutation from the evolutionary algorithm component. We can see by a process of optimisation that we have extracted reusable components and encapsulated complexity to the point where optimisation components have become more specialised. The fitness function and random mutations have become more specialised to the namespace topologies of the data. These components are

orchestrated by an extensible component lifecycle. We can now test different evolutionary algorithms by implementing their optimisations within the shared component framework.

At this point we have evolved a component framework that allows us to differentially optimise and orchestrate discrete components:

- We have identified a common lifecycle among evolutionary algorithms
- We argue that the fitness function can be better suited to the data namespace than the optimisation. The fitness function is modal as the checks for consistency, endpoint solution and fitness scalar have different processing and usage modes.
- We argue that random mutation can be a collection of different random action types. We argue that differentiating these modes leads to performance optimisations and further that these can be orchestrated in their own lifecycle to optimise degrees of freedom.
- If we follow this path we come to the conclusion that optimisation may be best implemented as a component within a framework that uses a fitness function and a random mutation that are detailed to each data namespace.
- A framework of this type allows us to compare and contrast the suitability and performance of different evolutionary algorithms.
- If we accept that random mutation can reach solutions then a subset of fitness function modes will allow bypassing of optimisation if an endpoint solution has been reached.

Most interestingly we have also gained the ability to test the efficiency of the random mutation in isolation by deactivating the optimisation components entirely. We have already made an argument that the random mutation can be better suited to the data than the optimisation, and so it makes sense that the random mutation be optimised independently for each dataset before integration with evolutionary optimisations. It was during this optimisation that we realised random mutation is capable of solving Sudoku puzzles without an optimisation component.

The question of the optimising component is an intriguing one. Sudoku has regularly been used to demonstrate the ability of evolutionary algorithms. On various occasions particle swarm, genetic algorithms and hybrid Meta heuristics have been shown to be capable of solving Sudoku problems. Using the component framework above we managed to confirm that indeed particle swarm, genetic algorithms and simulated annealing could solve these problems.

However as noted each of these heuristics has a random mutation component which is separately optimisable. Our aim therefore was to improve this function in isolation, which would then improve the baseline performance of each of the evolutionary algorithms. Doing so involves operating lifecycle with out the optimisation components. At this point it became apparent that the random mutation is capable of solving the problems.

This of course implies that all of the optimisations within this lifecycle may be able to solve the same problems as the random mutation, as long as they do not sufficiently overpower the random mutation function.

This also implies that an inability to solve Sudoku problems may be implementation specific. Results indicated a benefit in using evolutionary algorithms for most problems, for which evolutionary algorithms received lower main iteration counts to the endpoint solution. However, in the same way that some problems are more difficult for humans, the evolutionary algorithms also seem to struggle by concentrating around local maxima. In these cases the optimised random mutation lacked concentrating behaviour and achieved faster solution times. This result seemed counterintuitive. We would have hoped that by applying focused attention we should be more capable of solving problems in all conditions. Yet we appear to see evidence that a broader solution mode can lead to answers for more difficult problems in a shorter time frame.

Consider too that the value of each solution candidate is a sample of the fitness of that point in the namespace. Aggregating the population of candidate solutions might be considered as analogous to an awareness of the solution candidates so far.

This brings an intriguing correlation to the anatomy of human vision, and the implications of having too narrow a focus. Split brain theory shows the left hemisphere has a predilection for focused mono-procedural tools based processing, much like the way optimisation acts as a concentrating force for solution candidates around the best solution found so far. The longer an evolutionary algorithm spends around a maximum the more attributes are copied from the better solutions to the weaker, the more similar the population becomes. We can see this as a concentration of attention focus around the best solution so far.

The human brain uses both strategies at the same time. The left hemisphere has a preference for focused attention, much in the way evolutionary algorithms concentrate solution candidates around local maxima. The right hemisphere prefers a broader attention mode. Human vision in particular has peripheral perception modes, which have strengths in motion detection and changes in light intensity. These modes benefit from the widest possible distribution of attention, which correlates to the idea of reduced optimisation for solution candidates, and more of a bias towards random mutation. In the same way that optimisation concentrates candidates, random mutation distributes them through the namespace topology. This matches how we look for something we've lost. The way we attempt to remember where we have left an object is dealt with as though it were a problem to solve, while we were also broadly paying attention by looking around, as we look in case we have forgotten something or someone else may have moved the object in question.

## 8. PARALLELISATION

Consider the approaches we would take if we were intending to parallelise heuristic or iterative approaches across many processor cores. We would be intending to balance the workload across the cores to minimise aggregated execution time. This might typically be achieved by the introduction of a queuing mechanism whereby each thread of execution requests the next piece of work. Or we may ‘shard’ the problem into discrete domains that each core can then work on in isolation. There are multitudes of successful implementations of these types, particularly with the advent of software as a service and cloud computing.

Along with implementations of this type we also assume the anti-patterns required for synchronisation and delegation. There is an overhead required for multi core processes that share cached information. There are overheads for delegation processes whereby the client acquires the next piece of work. There are inefficiencies in sharing approaches, which may lack the adaptability to devote more processes to current interval areas of attention. If we look at implementations of this type then our random mutation has some distinct advantages.

Most evolutionary algorithms represent with challenges to parallelisation. The optimisation process uses the results of the fitness calculation to replicate attributes from stronger candidates to the weaker, which implies identifying differences between the two and changing attributes where possible. There are some exceptions to this process as in the case of simulated annealing where optimisation represents more as a backtracking mechanism against retrograde fitness changes. In general though, the processes we see in genetic algorithms and particle swarm optimisation require granulated comparisons and replications between candidates in the same logical group. Parallelisation therefore implies attribute comparisons and replications between different processing groups.

A possible remediation to this granulation problem can be seen in some of the swarm grouping behaviours in particle swarm optimisation. Rather than have the entire population follow the generic population wide candidate we instead break the population into separate groups, which allows for each group to move separately. We then track the global best solution as the best of the lead candidates in each swarm.

Mapping this consideration to multiple cores of execution in multiple domains we use the fitness check to identify best candidates in local populations that can be copied into other groups of optimisers. At that point the best candidate across any of the processing groups us accessible to the local optimisation processing and granular processing can continue within this core processing group.

For random mutation processing to participate in this style of processing we can simply reuse the results of the fitness function check that we are already performing to see if we have reached and endpoint solution. If we have improved our candidate beyond any other population then we can replicate this candidate to other optimiser populations. We have noted however that we do not need to replicate candidates between the randomisation populations, so there is little incremental effort to their participation in the scheme.

## 9. ADAPTIVE SHARDING OF SURFACES

These behaviours also serve to illustrate the advantages of random mutation as a sharding strategy. A natural consequence of parallelisation is the segmentation of the namespace into more than one population. An evolutionary algorithm identifies the best solution candidate so far and replicates its attributes. When we segregate the populations this behaviour maps to the best candidate in that population. As we have mentioned particle swarm optimisation can vary this behaviour by maintaining multiple parallel swarms. In general however we rarely see large numbers of small swarms.

The random mutation population does not replicate attributes in this manner. So if we argue that the best solution candidate for an evolutionary algorithm is the focus of attention, then we might also argue that each candidate in a random mutation population is a separate focus of attention. By this metric each of the random mutation solution candidates works like a focus of attention implemented in a green thread time-sharing scheme.

Additionally the random mutation life cycle is a subset of the evolutionary algorithm life cycle and so more iterations are achieved through random mutation for the same resource spends. The optimisers and the random mutator life cycles are decoupled by the introduction of a push mechanism towards the optimisers, which allows inclusion of new best candidates at the commencement of the next optimiser life-cycle iteration. This removes more common parallelisation strategies such as queuing or synchronisation between processes.

## 10. QUESTIONS OF HOMOGENEITY

Evolutionary algorithms work in part because changes in the fitness function are not large for small changes in the input variables. More gradual differentials allow for more predictable results and help remediate the possibility of spike maxima that we might not find, or might have difficulty escaping from.

The extreme of this condition where fitness function deltas become too radical might be where the set of combinatorially permissible inputs are not contiguous. If we were to randomly start in a region not connected to the global maximum then no degree of optimisation may lead us to it. Using a population of random mutators helps reduce these occurrences by placing hundreds of candidates throughout the input namespace. Initially we place them in a uniformly random manner, which implies that the percentage of random mutators starting in each region is related to the ratios of their volumes. In testing we often use populations of 500 or more greedy random candidates. By weight of numbers we would expect a workable number of candidates to fall into most non-contiguous regions.

We also start evolutionary algorithms in the name space. When they start they begin with they identify their best candidate so far and from that point the population receives attributes from this candidate. Time passes, and with each iteration the dispersed candidates make an attempt to acquire the attributes of this candidate. With time, where possible, candidates may achieve the entropy required to make the jump between regions and join the best candidates' region.

At the same time the greedy random continues to mutate in a directed fashion. At some point a greedy random candidate may achieve a state better than the evolutionary algorithm has found. The greedy random candidate then becomes the lead evolutionary algorithm candidate. If possible the evolutionary algorithm will bring other candidates to this new region. This ongoing revalidation allows modes of mutation entropy that the evolutionary algorithms lack on their own.

## 11. ON REVALIDATION AND PROBLEM COMPLETION

The complete model for these interactions includes a number of evolutionary algorithms and two populations of random mutators. We use one population as a dispersion agent throughout the namespace. The second is a random mutation behaviour applied to the satisficing cache, which orchestrates these interactions.

The satisficing cache is a normal population of the greedy random. However if any other population of candidates finds a solution candidate better than the current best then this is replicated into the cache and a random candidate is removed. This allows each optimising population to be able to check at the start of iteration in the satisficing cache to see if a new best candidate has evolved. Interactions with segregated parallel models can then occur between these caches.

We note in related work that this design is intended to have similarities with split brain theory, where the right hemisphere is more aware of the environment you are in and the passing of time, while the left hemisphere is more focused, mono-procedural and within the current time frame. In this case the satisficing cache works like attention around the most recent region of work. In this way the satisficing cache is a good contributor for improving random mutation entropy around local maxima.

This idea became highly significant with related work attempting to model human vision processing modes into heuristics. Having a central cache allows us to better track when a population that are meeting more success supplies the best candidate. When there are few local maxima we might see particle swarm optimisation providing more of the best candidates. If we have problems with local maxima we might see the particle swarm take the lead and then wait until the greedy random finds a new approach.

Most surprisingly we saw that the conditions around final completion represent in a similar way to a local maximum. In both cases it seems to the optimiser that there are few opportunities for

improvement with a lack of combinatorial freedom around the best candidates so far. In these conditions we saw the satisficing cache take over more often than not, and finish the solution more often than the evolutionary algorithms or cache.

In this work we were attempting to model human decision patterns. The comparison we drew was to how humans check to see if they have really finished a job. If a person is engaged in a tool based exercise they might check that a nail has been left at the correct depth, or a screw is flush to a surface, before deciding if they should continue or correct current work. Continuing to hammer or screw until forward progress ends might be successful in the middle scenario but a wider focus is required for fine detail. We appear to have heuristically simulated something like this validation interaction.

We expect this occurs in the byplay between two behaviours. The first behaviour comes from the ability of the random mutation to be able to chain together multiple iterations of change uncorrected by optimisation. This gives a decided advantage in such situations. The second behaviour comes from the replication of the best candidate into the satisficing cache. The best candidate is replicated each iteration into the satisficing cache until a significant number of candidates surround the area with random mutation. These accumulate and then diversify in a stream of candidates until they probabilistically find the global solution.

## 12. COMBINATORIALLY ADAPTIVE EFFECTS

The greedy random represents as an optimisation along one or more input variables where we have an expectation of a boundary condition leading to higher solution candidate fitness values. If we have an expectation that the boundary value of a variable, or the value next to it, represents as 2 in perhaps 100 possible values for this variable then we have excluded a requirement to combinatorially test the remainder of the range for this variable. This leads to as much as a 50x efficiency in randomisation across the namespace.

Of course problems of consequence often have complicated combinatorial restrictions. The greedy random behaviour adapts to the surface between valid and invalid inputs irrespective of the remainder of the input variables.

Perhaps most significantly empowering greedy random behaviours across multiple variables lead to a multiplicand efficiency improvement. If for example we were able to apply the greedy random behaviour to two different variables similar to our earlier example we would expect a  $50 \times 50 = 2500$  speed improvement. We have identified that we do not need to test the majority of the namespaces for these two variables, and yet will remain valid across the entire namespace if the problem becomes combinatorially difficult.

## 13. IMPLEMENTATION

We will perform two validations of the framework. The first will be a python-based component framework implementation of heuristics for solving Sudoku problems. Sudoku problems are defined on a  $9 \times 9$  grid where the digits from 1 to 9 are arranged such that no digit is repeated on any column, row or 3 by 3 cell grid of which there are 9. Sudoku puzzles are simple enough to be enjoyed as a diversion, and yet the more complex ones can occupy heuristics for thousands of iterations [12][13].

We have collected a sample of 60 or so Sudoku puzzles which were all solved by the evolutionary algorithms and greedy random. Most significantly we tested against 4 Sudoku puzzles, which

have been known as some of the most difficult created: "The Easter Monster", the "Golden Nugget", "tarek071223170000-052" and "col-02-08-071".

Our implementation shows the greedy random acting as the usual random mutation agent for each of the evolutionary algorithms. During testing each of the evolutionary algorithms can be selected or deselected individually via a command line option.

During initial testing the algorithm was run separately with each evolutionary algorithm selected and we verified that all the sample Sudoku puzzles could be solved with each.

It then became apparent that it would be a useful comparison to produce a baseline where no optimisation was selected. This would help identify the net benefit of the optimisation action above random mutation.

```

1 1) Easter Monster
2 2) tarek071223170000-052
3 3) Golden Nugget
4 4) col-02-08-071
5
6 1) 1...|....|.2 2) ..1|..4|... 3) ...|....|.39 4) .2.|4.3|7..
7 ..9.|4...|.5. ....|.6.|3.5 .....|..1|..5 .....|....|.32
8 ..6|....|7.. ....|9...|... ..3|..5.|8.. ....|....|..4
9 ---+---+--- ---+---+--- ---+---+--- ---+---+---
10 ..5.|9.3|... 0...|....|7.3 ..6|..9.|..6 ..4.|2...|.7.
11 ....|.7..|... ....|....|.28 ..7.|..2|... 8...|.5.|...
12 ....|85..|.4. 5...|.7..|6.. 1...|4..|... ....|..1|...
13 ---+---+--- ---+---+--- ---+---+--- ---+---+---
14 7...|....|5.. 3...|.8.|..6 ..9|..8.|..5. 5...|....|9..
15 ..3.|...9|..8. ..9|2...|.1... ..2.|....|6.. ..3.|9...|..7
16 ..2|....|..1 ..4..|..1|... 4...|7...|... ..1|..8|6..

```

Figure 2. The four most difficult Sudoku puzzles tested.

```

1   for entire population:
2       initialise each candidate as the Sudoku problem
3       mark problem cells as locked
4   while solution not found:
5       # random mutation
6       for each candidate in the population:
7           if randomisation factor == 0: break
8           for randomisation factor:
9               choose to add or remove by bias
10              if add then attempt to set a digit in an adjacent cell
11                  check fitness valid candidate
12                  if not valid: revert
13                  if fitness completed solution: break
14                  if remove: then blank a non problem cell
15                  if not solved and last action add: blank a non problem cell
16      # note below here is not required for the greedy random in isolation
17      # fitness ranking
18      for each candidate in population:
19          calculate fitness scalar
20      sort solutions by fitness scalar
21      # evolutionary heuristics
22      if particle swarm optimisation:
23          for each candidate:
24              attempt to replicate an attribute from the best solution
25              check fitness and valid solution
26              if not valid: revert
27      if genetic algorithm:
28          mark top 2/3rds of candidates as parents, bottom 1/3 as children
29          for each pair of parents in random order:
30              for all cells in child:
31                  if attributes match in both parents replicate to the child
32                  else set blank in child
33      if simulated annealing:
34          for each candidate:
35              if random mutations were not an improvement since last iteration:
36                  revert candidate to previous version
37              else: reduce the randomisation factor

```

Figure 3. The component hierarchy as pseudo-code

The second validation will be an attempt to validate the greedy random behaviours in an accessible problem. We will initialise a population of 100 candidates in the (x,y) plane such that  $-1 < x < 1$ , and  $-1.1 < y < \cos(\pi * x)$ . We will then perform 100 iterations of random mutation on these candidates allowing them free movement, excepting we will enforce  $y < \cos(\pi * x)$  as a constraint. We will manage two of these populations. The first will be a control group undergoing normal random mutation. The second will be a group undergoing greedy random mutations. If the greedy random is successful then we should see more candidates closer to the  $y = \cos(\pi * x)$  surface. This action will normalise mutations for greedy random candidates to be along the surface. In practice we would have configured the greedy random to attempt to maximise expecting this would improve fitness values.

```

5   SAMPLES = 1000
6   NUMITERATIONS = 100
7   NUMCANDIDATES = 200
8   INVALIDFITNESS = 9999
9   # the first half of candidates are a control group
10  controlgroup = int(NUMCANDIDATES / 2)
11
12  sampleResults = []
13  runResults = []
14  for k in range(SAMPLES):
15      # fill the array with points below the surface
16      candidates = []
17      for i in range(NUMCANDIDATES):
18          next = [random.uniform(-1,1), INVALIDFITNESS]
19          while(next[1] >= math.cos(math.pi *next[0])):
20              next[1] = random.uniform(-1.1,1)
21          candidates.append(next)
22
23      # for each candidate mutate NUMITERATIONS times rejecting movement above the surface
24      for i in range(NUMCANDIDATES):
25          for j in range(NUMITERATIONS):
26              # redo until solution is below the surface
27              ok = False
28              while(not ok):
29                  next = [candidates[i][0] +random.uniform(-.2,.2), candidates[i][1] +random.uniform(-.2,.2)]
30                  if(next[1] < math.cos(math.pi *next[0])):
31                      # ingest the mutation
32                      candidates[i] = next
33                      ok = True
34
35          if(i >= controlgroup):
36              # an aggressive greedy component that ensures we are on the surface
37              ok = True
38              while(ok):
39                  check = candidates[i][1] +.1
40                  if(check < math.cos(math.pi *candidates[i][0])):
41                      candidates[i][1] = check
42                  else:
43                      ok = False
44
45      # sum the distances from the surface to each candidate[i][1] for the surface cos(math.pi *candidate[i][0])
46      deltas = [0, 0]
47      runResults = ""
48      for i in range(NUMCANDIDATES):
49          idx = 0
50          if(i >= controlgroup):
51              idx = 1
52              if(deltas[1] == 0):
53                  runResults += "\n"
54                  deltas[idx] += abs(math.cos(math.pi *candidates[i][0]) - candidates[i][1])
55                  runResults += "%f,%f\n" % (candidates[i][0],candidates[i][1])
56
57      print "%u: control group delta %f = for %u iterations and %u candidates" %(k, deltas[0], NUMITERATIONS, NUMCANDIDATES/2)
58      print "%u: greedy random delta %f = for %u iterations and %u candidates" %(k, deltas[1], NUMITERATIONS, NUMCANDIDATES/2)
59      sampleResults.append(deltas)

```

Figure 4. Greedy random test code

## 14. TESTING OUTCOMES

| U01 "Easter Monster" - Iterations |      |      |      | Populations |                     |      |      | U04 "col-02-08-071" - Iterations |      |                     |      | Populations |      |      |  |
|-----------------------------------|------|------|------|-------------|---------------------|------|------|----------------------------------|------|---------------------|------|-------------|------|------|--|
| Heuristic                         | 8000 | 4000 | 2000 | 1000        | Heuristic           | 8000 | 4000 | 2000                             | 1000 | Heuristic           | 8000 | 4000        | 2000 | 1000 |  |
| Genetic Algorythm                 | 640  | 3030 | 2020 | 1680        | Genetic Algorythm   | 3250 | 1290 | 1550                             | 2230 | Particle Swarm      | 780  | 580         | 1300 | 1550 |  |
| Particle Swarm                    | 1080 | 1000 | 770  | 1460        | Pure Random         | 1440 | 780  | 8260                             | 9530 | Pure Random         | 1440 | 780         | 8260 | 9530 |  |
| Pure Random                       | 1140 | 1050 | 450  | 2290        | Simulated Annealing | 250  | 1640 | 350                              | 5520 | Simulated Annealing | 1620 | 1430        | 3000 | 4160 |  |

| U02 "tarek071223170000-052" - Iterations |      |      |      | Populations |                     |       |       | Average iterations for U* |        |                     |        | Populations |        |        |  |
|--|------|------|------|-------------|---------------------|-------|-------|---------------------------|--------|---------------------|--------|-------------|--------|--------|--|
| Heuristic                                | 8000 | 4000 | 2000 | 1000        | Heuristic           | 8000  | 4000  | 2000                      | 1000   | Heuristic           | 8000   | 4000        | 2000   | 1000   |  |
| Genetic Algorythm                        | 3670 | 5920 | 1480 | 3690        | Particle Swarm      | 862.5 | 810.0 | 1185.0                    | 1465.0 | Genetic Algorythm   | 2160.0 | 3025.0      | 1825.0 | 2655.0 |  |
| Particle Swarm                           | 800  | 1260 | 1300 | 1050        | Pure Random         | 910.0 | 980.0 | 3040.0                    | 3182.5 | Pure Random         | 910.0  | 980.0       | 3040.0 | 3182.5 |  |
| Pure Random                              | 330  | 850  | 3070 | 380         | Simulated Annealing | 440   | 1100  | 920                       | 770    | Simulated Annealing | 765.0  | 1300.0      | 1172.5 | 2920.0 |  |

| U03 "Golden Nugget" - Iterations |      |      |      | Populations |                     |      |      | Thousand boards for solution of U* |      |                     |       | Populations |      |      |  |
|----------------------------------|------|------|------|-------------|---------------------|------|------|------------------------------------|------|---------------------|-------|-------------|------|------|--|
| Heuristic                        | 8000 | 4000 | 2000 | 1000        | Heuristic           | 8000 | 4000 | 2000                               | 1000 | Heuristic           | 8000  | 4000        | 2000 | 1000 |  |
| Genetic Algorythm                | 1080 | 1860 | 2250 | 3020        | Particle Swarm      | 6900 | 3240 | 2370                               | 1465 | Genetic Algorythm   | 17280 | 12100       | 3650 | 2655 |  |
| Particle Swarm                   | 790  | 400  | 1370 | 1800        | Pure Random         | 7280 | 3920 | 6080                               | 3183 | Pure Random         | 7280  | 3920        | 6080 | 3183 |  |
| Pure Random                      | 730  | 1240 | 380  | 530         | Simulated Annealing | 750  | 1030 | 420                                | 1230 | Simulated Annealing | 6120  | 5200        | 2345 | 2920 |  |

Figure 5. Sudoku Results

The pure random mutation (with no optimisation) and all three evolutionary algorithms were shown to be able to solve all 60 Sudoku puzzles. These are the results for the 4 hardest puzzles:

1. The effect of being caught in local maxima had a significant effect on average times. If the algorithm catches a local maxima on harder problems in 20% of runs average iteration counts can double or triple. The algorithms recover and complete, but at large time scales.
2. The genetic algorithm had median performance. This is thought to be of a consequence of a relatively higher complexity in the optimiser combined with a slower propagation rate for good attributes. This idea is correlated in the genetic algorithm showing less benefit from larger population sizes (17280 to 12100 to 3650 to 2655) of thousand boards.
3. Where optimisation outperformed random mutation on the harder problems it was usually particle swarm optimisation. If we multiple the size of the population by the number of iterations as a number of boards then particle swarm achieves end point solution in less than half the number of (1465 to 3138) of thousand boards for populations of 1000.
4. Simulated annealing held the closest correlation to pure random (2920 to 3183) of thousand boards for populations of 1000. This is to be expected, as there is no real propagation of attributes in this optimisation. Rather there is an additional random mutation, which is only significant on improvement.

We now consider the results of the greedy random efficiency testing. The tests show a control population of normal random mutators and a greedy random each varying 100 candidates starting in the range  $x=(-1, 1)$  100 times each. We impose  $y < \cos(\pi * x)$  as an uneven constraint surface.

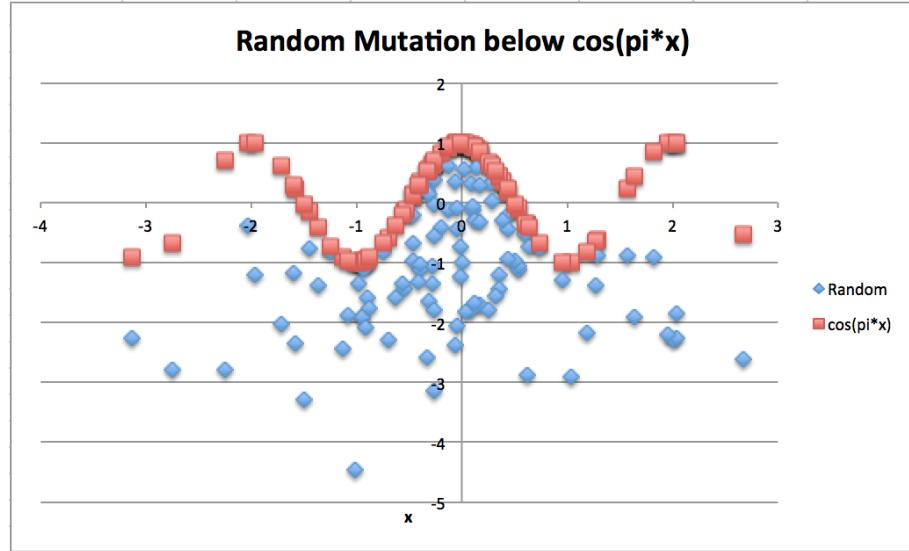


Figure 6. Random mutation efficiency

We can see the random mutator starts in the  $x=(-1,1)$  and  $y>-1.1$  range and expands in all directions. The candidates near the curve seem relatively compressed and lateral movement past the  $x=-1$  and  $x=1$  values seems reduced unless  $y < -1$ . The uneven constraint initially appears to effect expansion rates in an uneven way.

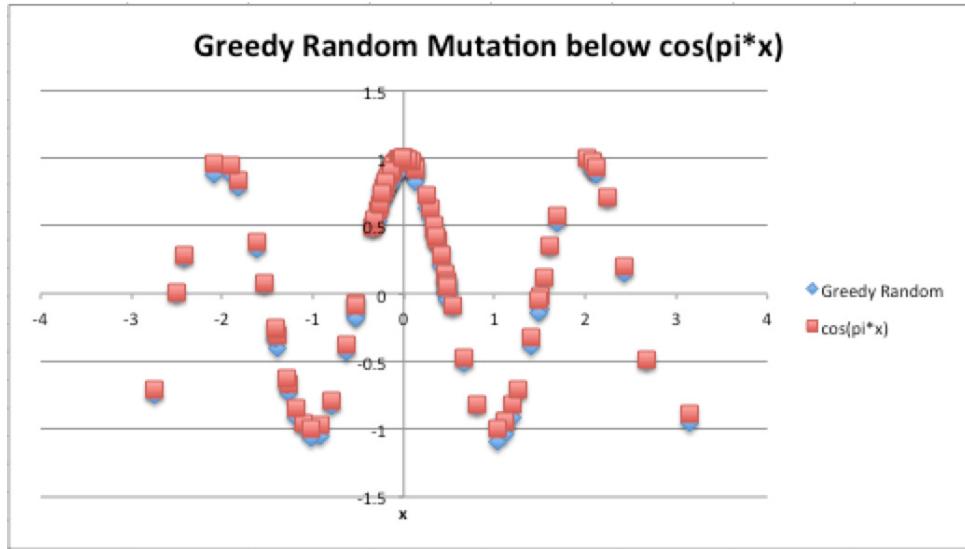


Figure 7. Greedy random efficiency

Here we see the greedy random solution candidates adjacent to but just below the constraint surface. Movement is forced along the constraint surface consistent with the expectations of a simplex algorithm.

We were not required to set expected optimal or control values. Linear optimisation can show preferences for boundary conditions for input values, and we are doing the same. However the boundary conditions can vary due to combinatorial restrictions or the behaviours of other input variables. While we cannot plan for a specific input boundary value we can ask the greedy random to prefer values with a directional bias. In this case, increase values while still remaining valid below  $y=\cos(\pi * x)$ . In usual terms we would not know this constraint, or how it varies with combinations of inputs. The greedy random would adapt.

The net effect is that we are now treating a 2-dimensional area below the curve as more like a linear position along the curve itself. We believe improved fitness occurs with these inputs on the line, so we randomise along it. We are converting the examination of the larger volume into the more specific boundary in a probabilistic way.

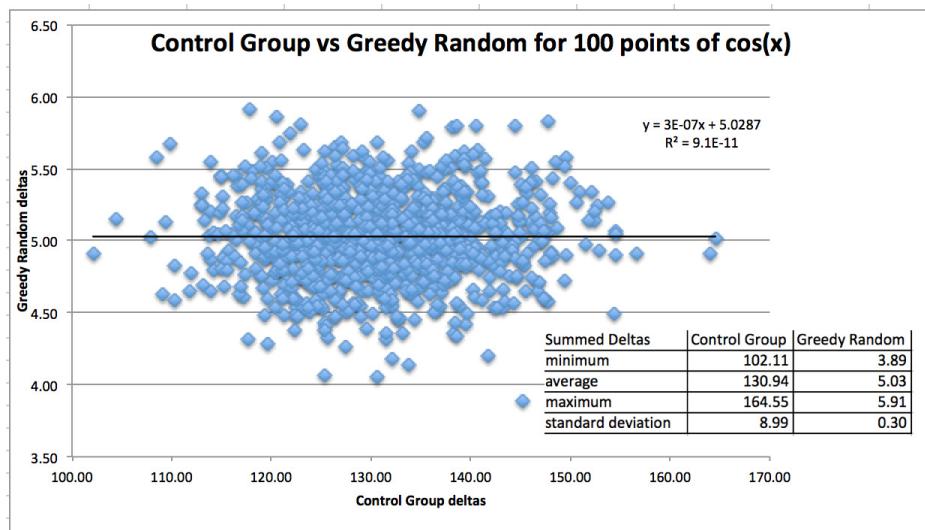


Figure 8. Check for correlating behaviours

These results show the plots of delta between the randomised (x,y) and the  $y=\cos(\pi * x)$  surface for 1000 runs. With an  $R^2$  of 9.1E-11 we show there is no appreciable correlation between the truly random mutation and the greedy random. We can therefore expect the greedy random is as relatively random, just within a smaller range.

If we process 1000 runs of 100 points for the control group and the greedy random we do not find a significant correlation ( $R^2 = 9.1e-11$ ). On average the greedy random was better than 20 times more efficient than the test control group at reducing distance to the test function.

Each input variable for a problem can be represented this way, and the effects multiply.

## 15. DISCUSSION

We have shown that an optimised random mutation is capable of solving Sudoku puzzles on its own. We have shown that evolutionary algorithms, which used this random mutation, were also capable of solving the puzzles.

The major danger to completion would therefore appear to be in the balance between random mutation and optimisation. If the action of copying attributes from the strongest candidates were capable of offsetting randomisation then any attempt to break away from a local maximum would be lost.

An amenable solution to this problem would appear to be the addition of a separate random mutation population to the action of the evolutionary algorithm. In this way one population would always be capable of random mutation. Whenever the random mutation population finds a better solution than the evolutionary algorithm then this can be replicated across to the evolutionary algorithm population. Optimisation will then replicate these new preferable attributes among the evolutionary algorithm population.

We have also attempted to demonstrate and quantify the strength of optimisation that the greedy random represents. We have shown how operating in this way can convert the behaviour of candidates from occupying a volume into populating a complex boundary. We have shown this operation does not affect the random nature of the mutations, excepting that they now occur in a more confined space. Finally we have discussed how these effects multiply by the number of participant input variables, which lends hope for major performance improvements in more complex problems.

Finally it is worth noting that the greedy random mutator and related optimisations better model human broad based attention modes by distributing solution candidates without the attractive forces of optimisation.

## 16. FUTURE WORK

This implementation was created to test optimisation of the evolutionary algorithms. In this case the random mutations are inline with the rest of the evolutionary algorithm, and the candidate population has one heuristic mode. During the discussion on satisficing behaviours we noted possibilities for additional modes.

We have seen that the greedy random has promise with problems that challenge evolutionary algorithms. Creating a hybrid with a population for each will allow us to displace the evolutionary algorithm from local maxima by replicating better candidates from the mutation population.

We could also be able to create a secondary population of mutation that was seeded from the evolutionary algorithm. This population:

1. Improves randomisation around the best exploitation targets.
2. Can operate as a satisficing cache [13][14][15] between different algorithms where the best candidates can be shared between populations.

Since this work began new evaluations of Sudoku puzzles have emerged and we would enjoy retesting against some of the newer higher ranked puzzles.

We see promise in representing combinatorial boundaries as n-dimensional surfaces. In these conditions we have shown the greedy random may provide significant performance improvements. More than this however, this geometric analogy of problem spaces may yet yield more improvements.

## REFERENCES

- [1] Carlos M Fonseca and Peter J Fleming. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. 423:416–423, 1993.
- [2] J Kennedy and R Eberhart. Particle swarm optimization. . . . 1995 Proceedings, 1995.
- [3] Sylvain Gelly and Yizao Wang. Exploration exploitation in go: UCT for Monte- Carlo go. 2006.
- [4] Enrique Alba and Bernabé Dorronsoro. The exploration/exploitation tradeoff in dynamic cellular genetic algorithms. Evolutionary Computation, IEEE Transactions on, 9(2):126–142, 2005.
- [5] Urszula Boryczka and Przemysław Juszczuk. Solving the sudoku with the differential evolution. Zeszyty Naukowe Politechniki Białostockiej. Informatyka, pages 5–16, 2012.
- [6] Kyun Ho Lee, Seung Wook Baek, and Ki Wan Kim. Inverse radiation analysis using repulsive particle swarm optimization algorithm. International Journal of Heat and Mass Transfer, 51(11):2772–2783, 2008.
- [7] Scott Kirkpatrick, D Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. science, 220(4598):671–680, 1983.
- [8] Rhyd Lewis. Metaheuristics can solve sudoku puzzles. Journal of Heuristics, 13(4):387–401, 2007.
- [9] Charles Darwin. On the origin of the species by natural selection. 1859.
- [10] Dirk Buc̄e, Nicol N Schraudolph, and Petros Koumoutsakos. Accelerating evolutionary algorithms with gaussian process fitness function models. Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on, 35(2):183–194, 2005.
- [11] Ajith Abraham, Rajkumar Buyya, and Baikunth Nath. Nature’s heuristics for scheduling jobs on computational grids. In The 8th IEEE international conference on advanced computing and communications (ADCOM 2000), pages 45–52, 2000.
- [12] Sean McGerty. Solving Sudoku Puzzles with Particle Swarm Optimisation. Final Report, Macquarie University, 2009.
- [13] Sean McGerty, Frank Moisiadis. Managing Namespace Topology as a Factor in Evolutionary Algorithms. Artificial Intelligence in Computer Science and ICT 2013.
- [14] Herbert A Simon. Theories of bounded rationality. Decision and organization, 1:161–176, 1972.
- [15] Herbert A Simon. Rationality as Process and as Product of Thought. The American Economic Review, 68(2):1–16, 1978.
- [16] Francesco Grimaccia, Marco Mussetta, and Riccardo E Zich. Genetical swarm optimization: Self-adaptive hybrid evolutionary algorithm for electromagnetics. Antennas and Propagation, IEEE Transactions on, 55(3):781–785, 2007.

## Authors

Sean McGerty is a PhD Research student at the University of Notre Dame Sydney. Sean studies relationships between human satisfying behaviour patterns and evolutionary algorithms. His current research areas include modelling broad based attention modes, namespace topology optimisations and satisfying solution behaviours. Using these taxonomies Sean has mapped the anatomy of human vision into services architectures. From these he has produced artificial intelligence models, which predict observable behaviours. Most recently Sean is working on modalities of human task sequence prioritisation.



Dr Frank Moisiadis is the Head of Mathematics and a Senior Lecturer at the University of Notre Dame, Sydney. Frank has a PhD in Software Engineering (thesis on Prioritisation Algorithms for System Requirements Using Fuzzy Graphical Rating Scales., a MSc in Computer Science (thesis on Improving Search Algorithms and Path Planning for Autonomous Robots) and a BSc(Hons) in Mathematics. His current research focuses on using fuzzy graphical rating scales for requirements engineering and optimising information flows in health systems. He has authored over 30 international and national research papers, supervised PhD students in Artificial Intelligence, Health Informatics, Information Systems and Software Engineering and authored two editions of the textbook, “Principles of Information Systems” published by Cengage Learning (2007 and 2010).

