ARE EVOLUTIONARY ALGORITHMS REQUIRED TO SOLVE SUDOKU PROBLEMS?

Sean McGerty and Frank Moisiadis

University of Notre Dame Australia
sean.mcgerty@gmail.com
frank.moisiadis@nd.edu.au

ABSTRACT

Sudoku puzzles are an excellent testbed for evolutionary algorithms. The puzzles are accessible enough to be enjoyed by people. However the more complex puzzles require thousands of iterations before a solution is found by an evolutionary algorithm. If we were attempting to compare evolutionary algorithms we could count their iterations to solution as a indicator of relative efficiency. However all evolutionary algorithms include a process of random mutation for solution candidates. I will show that by improving the random mutation behaviours I was 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 out rhythms may then attempt to replicate attributes of the more successful candidates to the others. In this way we can solutions become more like the better solutions and the cycle continue. 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 around 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 for sure 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

David C. Wyld et al. (Eds) : CCSIT, SIPP, AISC, PDCTA, NLP - 2014
pp. 365–377, 2014. © CS & IT-CSCP 2014 DOI : 10.5121/csit.2014.4231

expectation of being ample to find a good solution and potentially the best solution without checking all possible 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 maxima. 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 maxima.

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.

2. EXPLORATION VS. EXPLOITATION

Evolutionary algorithms face the same problems that people often do. Should I continue to try and solve a problem where I am at the moment? Or should I diversify in case I am 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 optimization. The act of sharing successful attributes makes the candidates look more similar, while the actions of random mutation pushes 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.

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.

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

3. BROAD BASED ATTENTION

I 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 a heuristic if we first notice that simulated annealing's optimisation modes are a bit different than 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 it discards 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 name spaces.

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 occur at 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. I 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 for 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. I suggest that we need a mechanism for being aware of candidates sufficiently outside the local maxima to be able to escape.

Think of this as an attention mechanism, which allows the adaption away from local maximums 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, which 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 at four and four of eight, and we no longer see improvement with any random change.

A random mutation component capable of integrating with evolutionary heuristics will need to interoperate with this evolutionary algorithm life cycle. 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, which 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 however 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]. Consider our example from earlier though we had two numbers between zero and five. Random change can either the increase or decrease of value. In the case of a crossword puzzle this could be the addition or erasure of the character. In general terms if we are attempting to fill positions in a solution should attempt to add values in preference to removing them.

This leads to an idea that we call 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, which have added as many sales as possible before rendering the candidate invalid. We call this process the greedy random because of this perspective of attempting to fill as many cells as possible.

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 as 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 local maxima.

This was an important design consideration during testing. The solution to this problem became apparent during attempts to integrate with the evolutionary algorithm lifecycle. During each iteration the candidates in the population is 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.

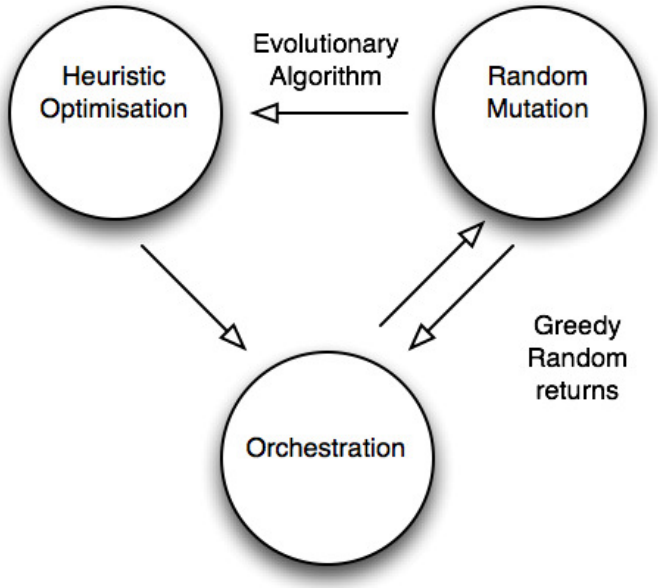

Figure 1 - This diagram shows the modified heuristic iteration lifecycle

I 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 is 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 less 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 name space we can 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 8x8 x3x9 = 2781 as the score for a solved board and 0 for an empty one.

7. COMPARING DIFFERENT EVOLUTIONARY ALGORITHMS

Consider the situation where we are attempting to evaluate suitability of different evolutionary algorithms on 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 life cycle. 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 forgiven example of a problem may be suboptimal in the more general case. I argue that when comparing the performance of evolutionary algorithms we need to separate those components, which are less related to optimisation and more likely to be shared between implementations.

I would make an argument that the 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 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 life cycle, which accepts differentiated optimisation components in the data management, and fitness functions should be reusable.

In the same way as 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 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 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 life cycle.

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 up 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 are fitness is the height of the swell and the endpoint condition is distance up the beach as a boolean consistent with a causal relationship. Increase the swell, 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 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 life cycle. 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:

- I have identified a common lifecycle among evolutionary algorithms
- I 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.
- I argue that random mutation can be a collection of different random action types. I 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 optimization 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 I realised random mutation is capable of solving Sudoku puzzles without an optimisation component.

This question 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 I 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 optimise able. My 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 life cycle with out the optimisation components. At this point it became apparent that the random mutation is capable of solving the problems by itself.

- This of course implies that all of the optimisations within this life cycle 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 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 in these cases. This result seemed counterintuitive. You would hope that by applying focused attention should be more capable of solving problems in all conditions. Yet we appear to see evidence that 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 maxima 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 however 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 optimisation concentrates candidates, random mutation distributes them through the namespace topology. This matches how we look for something we've lost. The attempt to remember where we have left an object as though it were a problem to solve, and we are also broadly paying attention, as we look in case we have forgotten something or someone else may have moved the object in question.

8. IMPLEMENTATION

We will demonstrate these ideas with a python-based component framework implementation of heuristics for solving Sudoku problems. Sudoku problems are defined on a 9 x 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 can occupy heuristics for thousands of iterations [12][13].

I have collected a sample of 60 or so Sudoku puzzles which were all solved by the evolutionary algorithms and greedy random. Most significantly I 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".

My 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 I verified that all the sample Sudoku puzzles could be solves 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) Easter Monster
2) tarek071223170000-052
3) Golden Nugget
4) col-02-08-071

1) 1..|...|..2 2) ..1|..4|... 3) ...|...|.39 4) .2.|4.3|7..
   .9.|4..|.5.    ...|.6.|3.5    ...|..1|..5    ...|...|.32
   ..6|...|7..    ...|9..|...    ..3|.5.|6..    ...|...|..4
   ---+---+---    ---+---+---    ---+---+---    ---+---+---
   .5.|9.3|...    8..|...|7.3    ..8|.9.|..6    .4.|2..|.7.
   ...|.7.|...    ...|...|.28    .7.|..2|...    8..|.5.|...
   ...|85.|.4.    5..|.7.|6..    1..|4..|...    ...|..1|...
   ---+---+---    ---+---+---    ---+---+---    ---+---+---
   7..|...|6..    3..|.8.|..6    ..9|.8.|.5.    5..|...|9..
   .3.|..9|.8.    ..9|2..|...    .2.|...|6..    .3.|9..|..7
   ..2|...|..1    .4.|..1|...    4..|7..|...    ..1|..8|6..
```

Figure 2 - The four most difficult sudoku puzzles tested.

```
for entire population
  initialise each candidate as the Sudoku problem,
  mark problem cells as locked.
while solution not found
  ##random mutation
  for each candidate in population
    if randomisation factor == 0 break
    for randomisation factor
      choose add or remove by bias
      if add then attempt to set a digit in a vacant cell
        check fitness valid candidate
        if not valid revert
        else if fitness completed solution break
      if remove then blank a non problem cell
    if not solved and last action add then blank a non problem cell
  ##Note below here is not required for greedy random in isolation
  ##fitness ranking
  for each candidate in population
    calculate fitness scalar
  sort solutions by fitness scalar
  ##evolutionary heuristics
  ##particle swarm optimisation
  if Particle Swarm Optimisation
    for each candidate
      attempt to replicate an attribute from the best solution
      check fitness valid solution
      if not valid revert
  ##genetic algorithm
  if Genetic Algorithm
    mark top 2/3rds of candidates as parents, bottom 1/3 as children
    for each child and pair of parents in random order
      for all cells in child
        if matching cells in both parents the same replicate in child
        else set blank in child
  ##simulated annealing
  if Simulated Annealing
    for each candidate
      if random mutations were not an improvement on last iteration
        then revert candidate to previous version
        else reduce randomisation factor
```

Figure 3 - The component hierarchy as pseudo code.

9. TESTING OUTCOMES

U01 "Easter Monster" - Iterations			Populations	
Heuristic	8000	4000	2000	1000
Genetic Algorythm	640	3030	2020	1680
Particle Swarm	1080	1000	770	1460
Pure Random	1140	1050	450	2290
Simulated Annealing	250	1640	350	5520

U02 "tarek071223170000-052" - Iterations			Populations	
Heuristic	8000	4000	2000	1000
Genetic Algorythm	3670	5920	1480	3690
Particle Swarm	800	1260	1300	1050
Pure Random	330	850	3070	380
Simulated Annealing	440	1100	920	770

U03 "Golden Nugget" - Iterations			Populations	
Heuristic	8000	4000	2000	1000
Genetic Algorythm	1080	1860	2250	3020
Particle Swarm	790	400	1370	1800
Pure Random	730	1240	380	530
Simulated Annealing	750	1030	420	1230

U04 "col-02-08-071" - Iterations			Populations	
Heuristic	8000	4000	2000	1000
Genetic Algorythm	3250	1290	1550	2230
Particle Swarm	780	580	1300	1550
Pure Random	1440	780	8260	9530
Simulated Annealing	1620	1430	3000	4160

Average iterations for U*			Populations	
Heuristic	8000	4000	2000	1000
Genetic Algorythm	2160.0	3025.0	1825.0	2655.0
Particle Swarm	862.5	810.0	1185.0	1465.0
Pure Random	910.0	980.0	3040.0	3182.5
Simulated Annealing	765.0	1300.0	1172.5	2920.0

Thousand boards for solution of U*			Populations	
Heuristic	8000	4000	2000	1000
Genetic Algorythm	17280	12100	3650	2655
Particle Swarm	6900	3240	2370	1465
Pure Random	7280	3920	6080	3183
Simulated Annealing	6120	5200	2345	2920

Figure 4 - Results

The pure random mutation (with no optimisation) and all three evolutionary algorithms were shown to be able to solve all 60 Sudoku puzzles. For 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) thousand boards.

3) Where optimisation out performed 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) thousand boards for populations of 1000.

4) Simulated annealing held the closest correlation to pure random (2920 to 3183) thousand boards for populations of 1000. This is to be expected, as there is no real propagation of attributes in this optimization. Rather there is an additional random mutation, which is only significant on improvement.

10. DISCUSSION

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

The major danger to completion would therefore appear to be in the balance between the random mutation and the optimisation. If the action of copying attributes from the strongest candidates is 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. Optimisation will then replicate these new preferable attributes among the evolutionary algorithm population.

11. 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 would 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 I would enjoy retesting against some of the newer higher ranked puzzles.

ACKNOWLEDGMENTS

I thank Mehmet Orgun of Macquarie University.

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 Bernab´e Dorronsoro. The exploration/exploitation tradeoff in dynamic cellular genetic algorithms. Evolutionary Computation, IEEE Transactions on, 9(2):126–142, 2005.
[5] Urszula Boryczka and Przemyslaw Juszczuk. Solving the sudoku with the differential evolution. Zeszyty Naukowe Politechniki Bialostockiej. 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 Buche, 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 – A Research PHD Student at the University of Notre Dame Australia Sydney Campus