Solving Sudoku Puzzles with Particle Swarm Optimisation

Sean McGerty
ITEC808 student
Macquarie University
Sydney, Australia
sean.mcgerty@sudents.mq.edu.au

Contents

Figures

1. Abstract

This workshop paper addresses Heuristic approaches to solving Sudoku puzzles, with a particular focus on Particle Swarm Optimisation(PSO). Sudoku problems and their constraints will be discussed. Heuristics used to solve Sudoku will be identified. We will then propose a component framework for supporting PSO and other Heuristics. Doing so allows us to detail aspects of PSO such as initialisation, optimisation, randomisation and the fitness function separately. Conclusions are drawn, implications drawn for the other Heuristics, and suggestions for further work are made.

2. Introduction

Sudoku is a popular combinatorial challenge for enthusiasts worldwide. The simple 9x9 grid and 4 constraints are easily understood, and the $6{,}700 \times 10^{18}$ or so possible combinations ensures enough complexity to last hours.

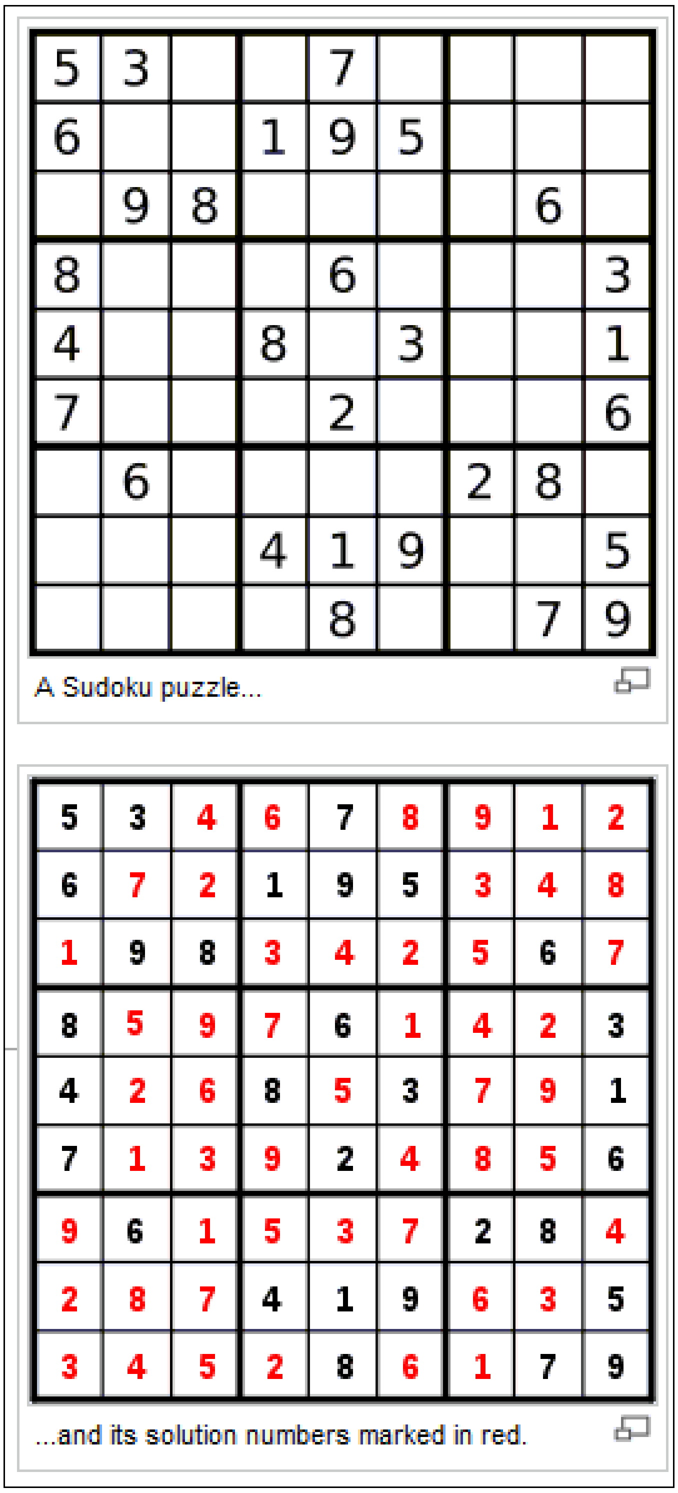
Sudoku is also significant as a target for heuristics research. Sudoku puzzles have been shown to be NP Complete(Yato, Seta - 2005), which means you may need to check all possible combinations to know you have the best solution. Finding reliable ways to quickly solve Sudoku problems may offer improved ways to solve NP Complete problems.

Generally speaking, Heuristics solve problems by working as a population. Each member of the population randomly changes, it is measured by a fitness function, and then information is shared about the more successful members.

There are two forms of Heuristics, Evolutionary Algorithms(EAs) and Swarming Algorythms(SA). EAs include Genetic Algorithms where particles simulate population lifecycles and the sharing of successful attributes from parents to children, or Simulated Annealing where elements are combined. SAs include Particle Swarm Optimisation where the particles move towards the most successful particle, in the way birds may flock in flight.

The weakness of Heuristics is a tendency to drive into local maxima. This happens because Heuristics chase the best solution, without knowing if it is really the best possible solution. A Heuristic might solve most of a puzzle, and only be able to tell it can’t improve. Our aim is to improve our heuristic to avoid these local maxima to fill all 81 cells more of the time.

The Heuristics have a lifecycle, and breaking their operation into components shows that there are multiple aspects to configuration and management. We suggest a component framework which will allow our PSO implementations to be optimised within each of these components.



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

A Sudoku puzzle...

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

...and its solution numbers marked in red.

http://en.wikipedia.org/wiki/Sudoku

Figure 1 - Sudoku puzzle

3. Heuristics

In this section we review Heuristics which have been used to try and solve Sudoku puzzles.

[1]A heuristic technique is one that attempts to find good fit solutions, often by trial and error or learning techniques, and avoids the requirement to evaluate every possible combination as a brute force method would.

Heuristics start with a collection of solutions which they attempt to improve. The result is the best so far, rather than the best possible solution. Our goal is to optimise heuristics so as to have confidence that the heuristic good solution is effectively the global optimum. If this were repeatable, solutions to other NPComplete problems may also be possible.

An initialised heuristic looks like a large collections of solutions which were used as the basis for improvement. The function of the heuristic compares and contrast the solutions with each other and random factors, attempting to discard weaker solutions and coalescing around stronger solutions. The simplest simulated annealing replica replicates a cooling metal just as the temperature would be dropping the effect of a random factor decreases and there's only improvements in solutions are accepted the population improves. Genetic Algorithms implement a survival of the fittest strategy with the best performed members of the population pass on their attributes to children who can replace the weaker members of the population. The process continues until the population stabilises. Particle swarm optimisation creates a large population of members each sure making individual choices about trusting a random factor, the global best solution, their own solution and the best solution amongst their neighbors. These implementations leverage the ability of the population to improve by comparisons between its own members; as a result by extension the larger the population the greater the points of comparison and the wider the initial randomisation of the problem space.

A key optimisation for heuristics is being able to avoid local maxima. We are lucky in that fitness functions for Sudoku can identify a correct solution, and in many cases there is only one unique solution.

Heuristics are good candidates for problem solving because they scale extremely well. At the most extreme implementations up to 5000 particles were seen in a Particle Swarm Optimisation implementation. This isn't to say however that you could scale beyond this range, and it would certainly make sense to do so. In particle swarm optimisation the most expensive operations might be related to social interactions with immediate neighbors, but there is ample evidence that this is understood. You can choose to implement social interactions, and while it is true that the social interactions need to be separately managed by a lookup table, it's not expected that such a management system would be expensive to implement. Most of the optimisations are inexpensive: the cooling effect in random simulated annealing interactions is simply the variation of a coefficient in a randomisation in the parent calculation. There aren't many heuristic operations which can't be performed in linear time order, thereby making this process more efficient than brute force methods, especially in larger domains.

That they scale so well is a great help when you consider that the widest possible distribution of the population is seen as a key benefit. Simply ensuring that the distribution of the population is purely random with the active throwing as many particles as possible into the domain and saturating puzzle could only help improve your chances of identifying as many local maxima as possible. This seems a contradiction, given the local maxima are a known problem of heuristics. But as demonstrated, and inability to bridge the entropy gap into a new partition has as much to do with an attraction to a local maxima as it does to a lack of population elsewhere in the puzzle domain.

[1] http://en.wikipedia.org/wiki/Heuristic

3.1 Brute Force Approaches

We will discuss successful brute force approaches and which optimisations work well with them.

So one of the first questions that might be asked when discussing computer based solutions for Sudoku might be along the lines "Can't they already solve Sudoku puzzles quickly?". The answer is yes, it's usually possible for a fairly simple script on a normal home computer to solve Sudoku puzzles in sub second time.

Sudoku puzzles are NPComplete, which in the worst case lead to a non linear solution times. For problems more complex than Sudoku this rapidly becomes unworkable, but our puzzles are of a size that people do them for fun, often in a few hours, and if they really needed to go through the $6{,}700\text{x}10^{18}$ combinations this wouldn't be the case. Most puzzles have enough givens to lead to a single result, and this approach usually takes up 20 or so cells. Humans then progress through the solution in a systematic way so as to reduce duplication and unnecessary work. Brute force algorithms can emulate these approaches, and the most popular two of these are Searches and Backtracking.

Searches attempt to reduce the number of bad moves by looking ahead and trying to balance both the Fitness Function and the accomplishing a traversal. Brute force ensures that all possible combinations might be checked, while also ensuring that invalid combinations in the namespace are avoided. This combination promises the best possible solution in the shortest possible time. This is the process humans go through when they choose how to proceed along the perceived optimal route without duplicating moves they've made previously.

Extending from the Searches approach, humans avoid re-entering dead ends by reversing moves they've recently made to a point where a different choice can be followed. In Brute Force approaches the iterative approach ensures that values for the present are tested before the current cell is cleared, attention moves back to the previous cell and the process continues. In this way the complete range of possible moves are checked while never producing the same combination twice, and the algorithm shows great reliability in climbing towards a local maxima and backing away from the dead end.

Neither of these approaches are understood to be consistent with Heuristics. The performance benefit of Heuristics comes from not needing to iterate through the entire namespace for the best solution, instead randomly varying their solutions in an attempt to spread out. Heuristics do not remember the path that got them to where they are. Instead SAs remember their own best result and the current best result among the rest of the population, while GAs value the current most successful in the population, allowing them to reproduce and replace the least successful members. Simulated Annealing grafts aspects of the more successful in the population into the weakest members.

Finally, the aim of our work is to improve PSO for Sudoku and by extension PSO in general. Adding Search and Backtracking abilities to PSO leads PSO towards brute force style costs which would be highly problematic in more difficult contexts.

3.2 General approaches with Heuristics

Here we discuss the similarities in various Heuristics which will later form the basis of my later recommended component framework.

In general the same approach persists with each Heuristic implementation:

Initialisation - Namespaces

Here we discuss how we organize the data for the heuristic. The Heuristic is initialized by mapping the problem into a namespace which assists later processing.

In some cases the detail is organised very close to a physical representation of the problem; for Sudoku a casual observer would recognise the data for each member of the population as a direct representation of the board. As a result the namespace scales as a board per member.

In object orientated design it has been said that a strong relationship between physical objects and matching logical representations is more intuitive in a first pass. In the same way I argue that directly mapping the Sudoku board into a 9x9 integer grid is the most intuitive representation. More context is needed, such as flags for givens and randoms so that the original problem isn't rewritten, but the board representation remains.

The simpler the representation used by a Heuristic the more suited it is for optimisation. Heuristics such as Simulated Annealing and Optimistions like Geometric Crossovers do not have namespace requirements beyond the board representation. As a result they are better suited to being interleaved between iterations of other approaches. Mixing Geometric Crossovers(GCs)(stateless) within Particle Swarm Optimisation(state full) has been particularly successful. This interleaving is made possible by the GCs using little more than the board representations.

Some implementations favor mapping the problem space into a graph representation, which helps decision processing based on weighted decision states and the weighted transitions between them. Using graphs guides the effort of traversing the problem, and in some cases reduces the cost of processing. However some of the graph representations are suspiciously like directed searches, which devalues the Heuristic because it's biased towards a planned knowledge of the namespace rather than the strength of the Heuristic. Therefore we avoid this approach.

A subset of the graph namespaces are 'Belief Propagation' systems, which encode their trust values in decision chains onto the nodes of the graph. In this case the decision tree is primarily acting in support of the heuristic and a behavior we value. However for the reasons given above there doesn't seem to be enough opportunity with this approach to try most of the optimisations as they would also need to be adapted to the new graph namespace. Given the singular nature of the namespace, and its unsuitability to Particle Swarm Optimisation we will not draw too much from this approach.

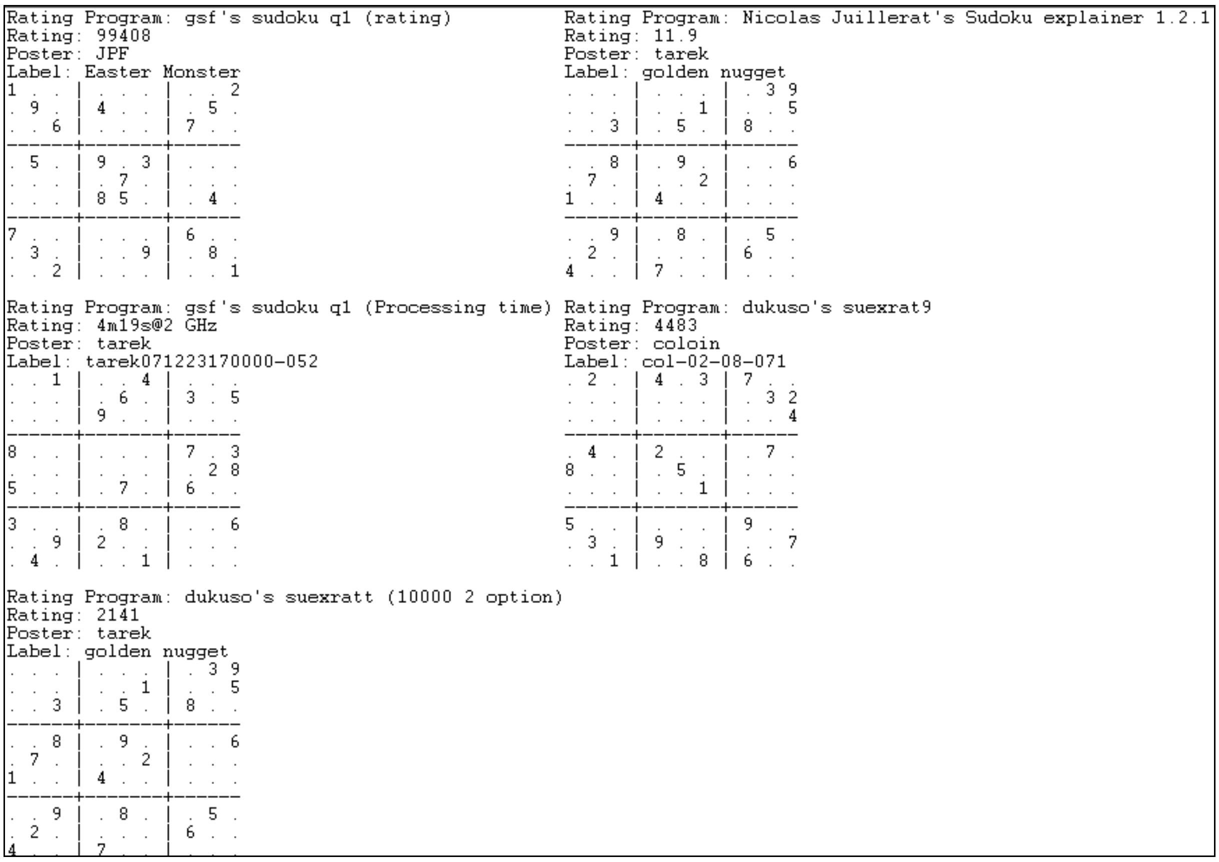
For similar reasons we will not pursue other representations that do not use namespaces which primarily operate from boards.

Initialisation – Givens

Givens are the numbers that define the original problem and remain unchanged as cells are filled.

Givens on published problems are selected by a Puzzle Master. The complexity of a problem is based on how difficult it is to solve with human approaches, so the Puzzle Master is a human adjusting the givens to adjust the complexity. The givens are usually selected in such a way that only one solution is possible. This is of great advantage to us evaluating Heuristics, because if we work a puzzle known to have only one solution, and our Heuristic manages to fill all 81 cells, then we know that we have in fact found the best possible solution and not just the best local maxima.

The fewest number of givens which has lead to a singular result appears to 48072 distinct configurations of 17 givens[2]. (Gordon Royle 2009). It seems remarkable that only 17 cells would dictate the contents of the final 64 cells.


```
Rating Program: gsf's sudoku q1 (rating)          Rating Program: Nicolas Juillerat's Sudoku explainer 1.2.1
Rating: 99408                                     Rating: 11.9
Poster: JPF                                       Poster: tarek
Label: Easter Monster                             Label: golden nugget
1 . . | . . . | . . 2                             . . . | . . . | . 3 9
. 9 . | 4 . . | . 5 .                             . . . | . . 1 | . . 5
. . 6 | . . . | 7 . .                             . . 3 | . 5 . | 8 . .
------+-------+------                             ------+-------+------
. 5 . | 9 . 3 | . . .                             . . 8 | . 9 . | . . 6
. . . | . 7 . | . . .                             . 7 . | . . 2 | . . .
. . . | 8 5 . | . 4 .                             1 . . | 4 . . | . . .
------+-------+------                             ------+-------+------
7 . . | . . . | 6 . .                             . . 9 | . 8 . | . 5 .
. 3 . | . . 9 | . 8 .                             . 2 . | . . . | 6 . .
. . 2 | . . . | . . 1                             4 . . | 7 . . | . . .

Rating Program: gsf's sudoku q1 (Processing time) Rating Program: dukuso's suexrat9
Rating: 4m19s@2 GHz                               Rating: 4483
Poster: tarek                                     Poster: coloin
Label: tarek071223170000-052                      Label: col-02-08-071
. . 1 | . . 4 | . . .                             . 2 . | 4 . 3 | 7 . .
. . . | . 6 . | 3 . 5                             . . . | . . . | . 3 2
. . . | 9 . . | . . .                             . . . | . . . | . . 4
------+-------+------                             ------+-------+------
8 . . | . . . | 7 . 3                             . 4 . | 2 . . | . 7 .
. . . | . . . | . 2 8                             8 . . | . 5 . | . . .
5 . . | . 7 . | 6 . .                             . . . | . . 1 | . . .
------+-------+------                             ------+-------+------
3 . . | . 8 . | . . 6                             5 . . | . . . | 9 . .
. . 9 | 2 . . | . . .                             . 3 . | 9 . . | . . 7
. 4 . | . . 1 | . . .                             . . 1 | . . 8 | 6 . .

Rating Program: dukuso's suexratt (10000 2 option)
Rating: 2141
Poster: tarek
Label: golden nugget
. . . | . . . | . 3 9
. . . | . . 1 | . . 5
. . 3 | . 5 . | 8 . .
------+-------+------
. . 8 | . 9 . | . . 6
. 7 . | . . 2 | . . .
1 . . | 4 . . | . . .
------+-------+------
. . 9 | . 8 . | . 5 .
. 2 . | . . . | 6 . .
4 . . | 7 . . | . . .
```

Figure 2 - Very Difficult Sudokus - the 'Unsolvables'

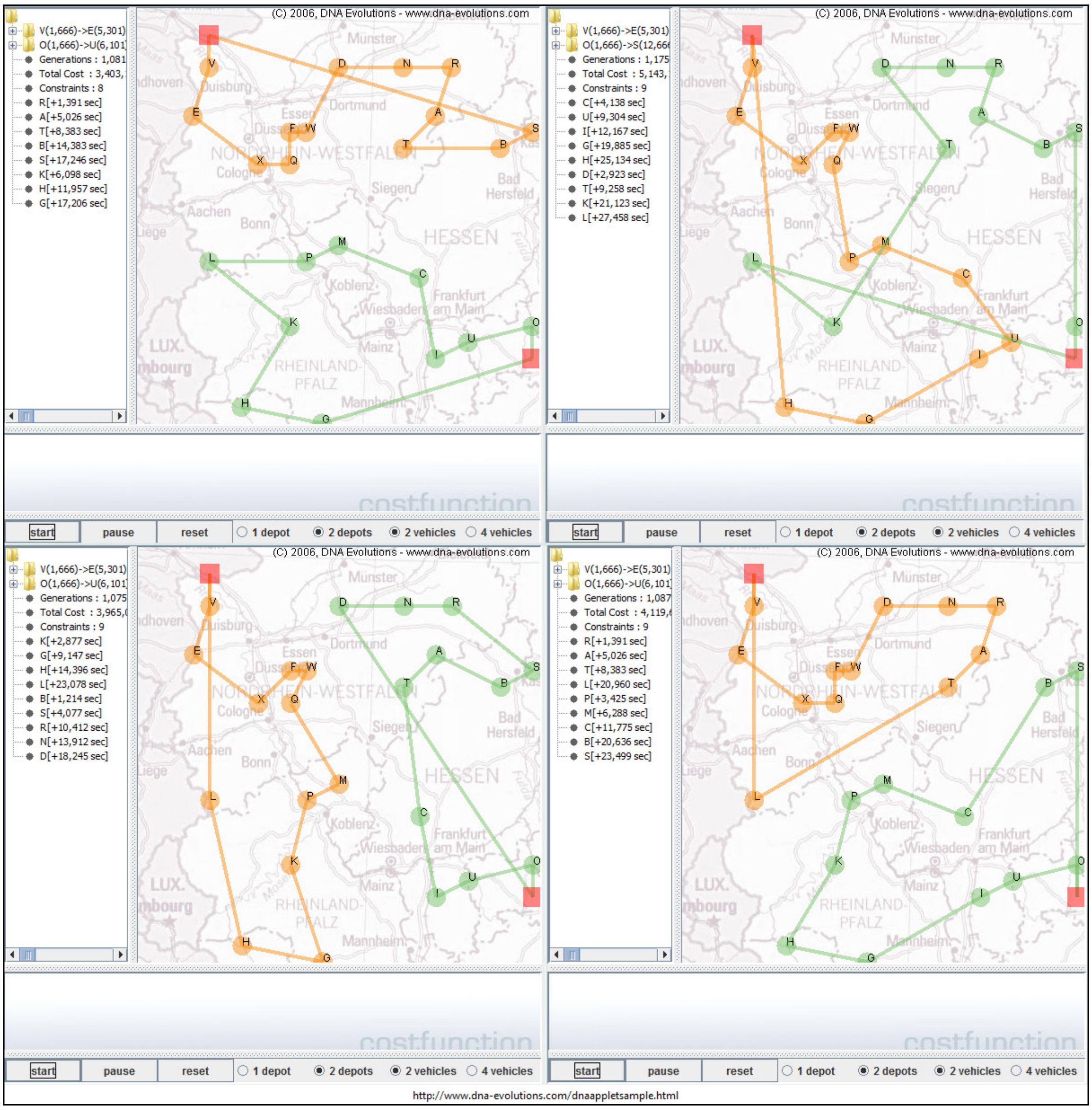
There also doesn't appear to be a correlation between the number of givens and the apparent complexity of the puzzle, of more importance seems to be the ability to defeat human Searching strategies. This reduces the problem to trial and error, which then makes the problem unsolvable in a reasonable amount of time. Figure 3 shows some of the 'unsolveable'[3] Soduku puzzles which need to be solved by Humans in brute force 'trial and error' fashion.

[2] http://people.csse.uwa.edu.au/gordon/sudokumin.php

[3] http://en.wikipedia.org/wiki/Algorithmics_of_sudoku#Exceptionally_difficult_Sudokus_.28Hardest_Sudokus.29

Avoiding Local Maxima

The greatest danger to Heuristics is also one of their best strengths: they aim to find good solutions without checking every possible combination – as a result they do not know for sure if their good solution is indeed the best solution.



```
V(1,666)->E(5,301)
O(1,666)->U(6,101
Generations : 1,081
Total Cost  : 3,403,
Constraints : 8
R[+1,391 sec]
A[+5,026 sec]
T[+8,383 sec]
B[+14,383 sec]
S[+17,246 sec]
K[+6,098 sec]
H[+11,957 sec]
G[+17,206 sec]
```

```
V(1,666)->E(5,301)
O(1,666)->S(12,66
Generations : 1,175
Total Cost  : 5,143,
Constraints : 9
C[+4,138 sec]
U[+9,304 sec]
I[+12,167 sec]
G[+19,885 sec]
H[+25,134 sec]
D[+2,923 sec]
T[+9,258 sec]
K[+21,123 sec]
L[+27,458 sec]
```

```
V(1,666)->E(5,301)
O(1,666)->U(6,101
Generations : 1,075
Total Cost  : 3,965,
Constraints : 9
K[+2,877 sec]
G[+9,147 sec]
H[+14,396 sec]
L[+23,078 sec]
B[+1,214 sec]
S[+4,077 sec]
R[+10,412 sec]
N[+13,912 sec]
D[+18,245 sec]
```

```
V(1,666)->E(5,301)
O(1,666)->U(6,101
Generations : 1,087
Total Cost  : 4,119,
Constraints : 9
R[+1,391 sec]
A[+5,026 sec]
T[+8,383 sec]
L[+20,960 sec]
P[+3,425 sec]
M[+6,288 sec]
C[+11,775 sec]
B[+20,636 sec]
S[+23,499 sec]
```

1 depot [✓ 2 depots] [✓ 2 vehicles] 4 vehicles

http://www.dna-evolutions.com/dnaappletsample.html

Figure 3 - 4 different stabilised Heuristic solutions to the same problem

Here Figure 3 is showing 4 stabilised solutions to the Travelling Salesman problem using 2 vehicles and 2 depots with the JOpt Genetic Algorithm Heuristic[4]. Most of them do not demonstrate waste, there are lots of short paths and reasonable procession from town to town. Notice however the least expensive solution is on the top left, the other 3 have collected around good but not best solutions. In truth, without checking all the solutions we can't even know for sure if the top left solution is in fact the absolute best possible.

[4] http://www.dna-evolutions.com/dnaappletsample.html

The Fitness Function

How we go about defining our Fitness Function might be one of the most important choices that we make. Being able to compare members of the population and potential changes with the hope of gaining an improvement above simple random change will be facilitated by the Fitness Function.

The simplest fitness function would be to simply count the number of filled cells. Given that a Sudoku puzzle's givens are usually formed to produce one solution, and all the cells have been filled, then a Fitness Function which counts the cells as 9x9 = 81 demonstrates the best possible solution.

Extensions to the Fitness Function would be extremely powerful. We've said earlier that we do not have the opportunity to perform the Search type functions that people take for granted. Fitness Functions do not have an ability to do look ahead searches, but they do interact with the constraints. Within Sudoku puzzles each cell interacts with 3 constraints representing the row, column and region the cell occurs in.

A strategy humans follow involves preferentially filling in constraints with the most cells. Heuristics can't search ahead like a human would, but they can interact with the constraints. Consider if we counted the population of each of the constraints for this cell. If we looked at a cell in a completed solution the score would be 3x9 = 27, giving a completed board value of 2187. Using this modified Fitness Function favors cells which help fill constraints. How much we trust filling in cells with busier constraints as against simply filling in cells may well be a trust factor that needs management over the lifecycle of the Heuristic.

It's worth noting that the fitness function is identified as a reusable component in our later discussions independent of any given heuristic. This is because while a heuristic lacks the ability to perform a search intelligence around the choice made it pursuing a better solution solely comes from the fitness function. And while the fitness function is responsible for interacting with the constraints, it is reasonable to assume that the fitness function is aware of the size of the constraints; if this assumption can be made in the fitness function requires the ability to preferentially fell constraints which have reduced degrees of freedom. In a small way this matches strategies employed by humans attempting to filling the easier parts of the puzzle first.

It is a very valuable choice to be able to make when you can avoid working in areas of the puzzle namespace which allow high degrees of variation, when potentially that work would be avoided when you return to congested constraints that you can no longer feel. If searches were allowed this could be described as selecting a path of least resistance by look ahead. In our case though, you're assessing the current state constraints and using it as a factor in the global value of the fitness function. When making a choice about where to place the next cell, we're simply selecting solutions which have already preferentially filled constraints .

Trust Factor Management

There are a range of factors that need management for a heuristic, and many of these may change while the Heuristic is running. As discussed with Simulated Annealing, the temperature coefficient falls, and the rate of change changes as a result, the longer the SA runs. In the same way initialization is nothing but random change before the Heuristic starts.

We have said earlier that heuristics scale particularly well because the calculations are simple and efficient. Trust factors represented in these calculations are simple linear co-efficients.

A management agent capable of setting an initial value for a trust factor in varying that value either over time, or as we approach a goal, or as we passed solution thresholds, is a reusable component that could hopefully sit across a range of such trust factors. As stated earlier it is the name of their implementation to be as reusable as possible and non-specific to any individual heuristic and as such a management component of trust factors is an ideal design goal.

Trust factor Management to be highlighted in each of the heuristic types were previously discussed. Simulated annealing has a trust factor the randomisation which we refer to as temperature which cools the time. The genetic algorithm has a number of controlling factors controlling such aspects as the rate at which members of the population are replaced, the number of children produced from each parental pair, and the number of parents which will be propagating; in simulated annealing terms it may make sense for faster rate of change earlier in stabilisation later. Particles full optimisation may have the widest range of trust factors controlling aspects such as randomisation belief in our best solution so far the best solution amongst our neighbors and the global best solutions found.

In addition to the core heuristics we have trust factors related to the optimisations we may interleave in between iterations of the main heuristic. For example, the simplest heuristic that we consider the simulated annealing is effectively stateless between iterations and so can be used as an optimisation of genetic algorithms or particles full optimisation. Trust factor related to temperature would be significant in this case as well as the amount of trust or variation that you allow to an optimisation is a contribution to the heuristic. For example do you introduce an optimisation with high levels of variation earlier in the process and trust that optimisations less the longer the process runs?

As mentioned earlier some of the controlling considerations to trust factors may be related to the date of the solution. It should be obvious that if we are more random at the start of the process and less random near the end, then it's easy to know your starting and harder to realise that you're closing in on a solution. So do you vary your trust factor with time – i.e. - the number of iterations? Do you vary a trust factor with the population of the board? Do you vary a trust factor with the fitness function value of the global best solution? Noting that some of the heuristics have congregating behaviors do you attempt an objective measure of the number of clusters and their size? Most of these options would seem to be possible, and they can be implemented independent of the workings of any of the individual heuristics.

Finally it also recommended that if trust factor Management is capable of assessing the progress of the population towards a solution that this would be useful to the purposes of reporting particularly in long runs or very large populations. References have been found to genetic algorithms being used to design jet engines and being left to run for many days[5].

[5] http://www.fortunecity.com/emachines/e11/86/algo.html

Heuristic Optimisations

You'll now discuss some of the considerations who was using optimisations with heuristics.

Research has shown that the inclusion of optimisations with mystics greatly improves the probability of success. Heuristics have a natural tendency to congregate towards local maxima, and other trust factors such as randomisation do not appear to have enough of a mitigating effect. In general we see two approaches. In the case of simulated annealing the trust factor to temperature which controls randomisation is reduced over time, and there doesn't appear to be much of a reliance on other optimisations. PSO however is usually discussed with reference to an optimisation; and we see examples such as Simulated Annealing, the inclusion of repulsive factors, or Geometric Crossovers.

Optimisations are also expected to have low expectations that a full heuristic and as such some of the more popular optimisations work from simple board representations and are effectively stateless. Simulated and yearling works as an optimisation because it has simple needs and stateless operation. Genetic algorithms however churn through the population replacing large numbers of members with new children; as such thing to a the operation of the parent heuristic and so are not well suited for use as an optimisation.

Propulsive affects considered on an optimisation that actually represent is a modification to the trust factors in the calculation of new solutions. At the point where the calculation attempts to move towards another strong solution whether it be global social or local is also a factor which attempts to push the particle and opposite direction hoping to sustain a gap between this particle and the attractive force that may be taking the particle towards a local maxima. To be honest I find the description of this implementation could be confusing. My attempt to use a hubristic to improve the quality of his solution and then intentionally reverse the actions in the other direction? No that the repulsion isn't against other particles in a similar area of the problems is because the distance calculation involved would be extremely expensive. We are simply talking about moving the particle way from what we perceive to be an improved solution state.

You will notice that if separately identified as consideration and are attempting to use an alternate strategy in the mechanism for initialisation of the members before the heuristic starts. The attempt to increase the separation of the members by allocating as many Randoms is possible there by increasing the separation before the heuristic would have an opportunity to help the members congregate. We would actually recommend this process along with a reduced trust factor congregation earlier the process is something that matches the temperature coefficient behaviours of simulated annealing.

Geometric crossovers are potentially very complex but we have attempted to target a simple implementation. As one of the final steps in the heuristic iteration, we will progress through the members and attempt to swap the value of random cells with another member, while still ensuring that the solution remains viable. The trust factor for this operation will determine the rate of change.

Simulated Annealing *(Lewis 2007)* is sometimes used as an optimisation technique on other heuristics. The basic idea is that the rate of random change is highest where we start in decreases over time, and this process is simple enough that it can be interleaved between heuristic iterations.

Finally it is worth noting that we intend to apply to trust factor to his optimisations which may vary with time.

4. Evolutionary Heuristics

[6]Evolutionary Algorithms(EAs) have generations of solutions that attempt to optimise themselves by combining the best elements of each other.

The general approach is outlined as follows:

- A generation of solutions are created
- They are assessed by a fitness function.
- The more successful members of the population share attributes or propagate.
- The least successful solutions are eliminated or attempt to gain the attributes of the more successful.
- A random mutation factor is applied.
- This process continues until the population stabilises around a collection of local maximums.

Genetic Algorithms(GAs) are EAs based around Darwinian evolution and survival of the fittest. *(Perez, Marwala 2008), (Mantere, Koljonen 2007), Darwin 1859.* GAs are similar to the other heuristics in that the population is randomly distributed and iterations are performed where the population undergoes random change, and the quality of any member is measured by the Fitness Function.

Simulated Annealing is a type of EA which combines attributes from good solutions into weaker solutions, and is based on how mineral crystals grow. *(Perez, Marwala 2008)*

Simulated Annealing holds extra value for Heuristics because it can be used as both a Heuristic and also as an optimisation for other Heuristics. For iterations as a Heuristic Simulated Annealing modifies each member of the population randomly, then evaluates each member against the fitness function, and copies attributes from the most successful members to the weaker members. Note that each time an iteration occurs the population is re-evaluated. For this reason simulated annealing can be interleaved between iterations of another Heuristic as an optimisation, and will not be adversely affected by changes made by the parent Heuristic. Genetic Algorithms are not as useful in this context as they completely remove weaker elements of the population.

Geometric Crossovers have a similar behavior to Simulated Annealing in their use as an Optimistion. However Geometric Crossovers do not leverage a Fitness Function, and so the changes they make are purely random. For this reason Geometric Crossovers are more of an optimisation that relies on a Heuristic to attempt to improve the population.

[6] *http://en.wikipedia.org/wiki/Evolutionary_computation*

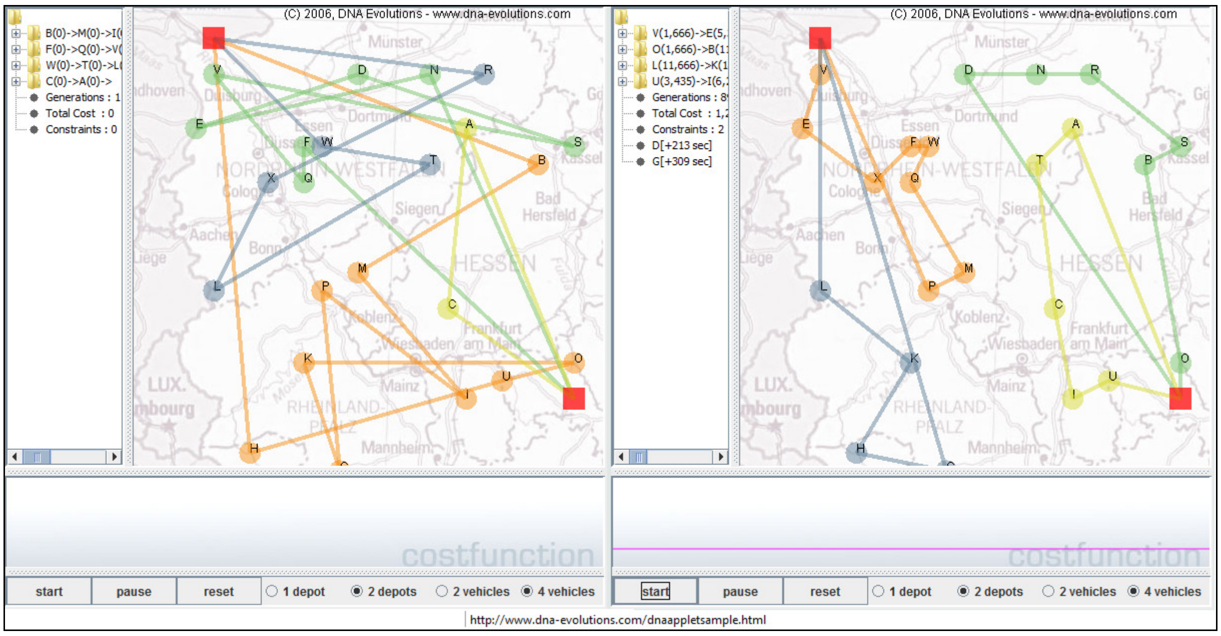
4.1 Genetic Algorithms

In Darwin's "On the Origin of the Species by Means of Natural Selection", he outlines a process whereby competitive advantage is gained after random change, leading to a survival of the fittest when they out compete the rest of the population. After successive iterations of this process the population improves and their ability to compete for resources has improved.

The following is an example when implemented as pseudo-code[7]:

```
1) Select a population
2) Apply the fitness function to the population
3) Repeat until [time limit or fitness achieved]
     a. Select the best ranking individuals to reproduce.
     b. Breed a new generation via crossover or mutation, giving birth to offspring.
     c. Evaluate the fitness of the offspring.
     d. Replace the worst ranked in the population with the offspring.
```

The mutation component is a randomisation to the solution. In our case randomisation would include changing non given cells in such a way that the constraints were not violated. Blank cells might be assigned values, existing values might be changed or erased. Randomisation independent of the rest of the algorithm has a very low probability of success in any reasonable time order.



```
B(0)->M(0)->I(
F(0)->Q(0)->V(
W(0)->T(0)->L(
C(0)->A(0)->
Generations : 1
Total Cost  : 0
Constraints : 0
```

```
V(1,666)->E(5,
O(1,666)->B(1
L(11,666)->K(1
U(3,435)->I(6,
Generations : 8
Total Cost  : 1,2
Constraints : 2
D[+213 sec]
G[+309 sec]
```

○ 1 depot [✓ ● 2 depots] ○ 2 vehicles [✓ ● 4 vehicles]

(C) 2006, DNA Evolutions - www.dna-evolutions.com
costfunction
start
pause
reset
http://www.dna-evolutions.com/dnaappletsample.html

Figure 4 - An example of a Genetic Algorythm Solving the Travelling Salesman problem

Figure 2 is a visual representation of the famous travelling salesman problem being approached by a genetic algorithm. The problem is to reduce the distance travelled by 4 vehicles visiting every town at least once and returning to 2 depots. Initially the GA is seeded by a completely random solution shown on the left; the Heuristic hasn't begun iterations using the fitness function as yet, and so there's little evidence of optimisation – the "Total Cost: shows as 0 as a result. The right image shows the best result so far from 1000 or so iterations, and it's evident that many of the cities have been joined by quite short paths, there's less evidence of paths crossing each other randomly. Four loops have appeared, one per vehicle, and where long paths exist it's usually connecting the depot to the first or last town in the loop.

[7] http://en.wikipedia.org/wiki/Genetic_algorithm#Simple_generational_genetic_algorithm_pseudocode

Note that no direction is given to the Heuristic beyond the fitness function and constraints, and this represents as an effort to reduce cost and for solutions to remain valid. We do not see a 5th car appearing, and we see all the towns being visited at least once and being connected to a depot, which shows the constraints were followed. By the same token we didn't tell the GA to use all the cars, and we didn't say that both depots needed to be involved; the GA has decided that cost is minimised by involving all 4 cars and both depots. Finally we have no guarantee that the solution that has been found is in fact the best possible of all solutions, we just know that the GA has stabilised around this solution and that it's the best solution it's found so far.

Crossovers mitigate stabilizing around local maxima by swapping cells between solutions in such a way that the constraints are not violated. Crossovers are seen as a dispersal factor that helps offset the tendency of the population to move towards the most successful members and potentially be caught in local maxima. How to tell if the population is approaching a potential solution, or merely running into a local maximum is a genuine problem. In general it is argued that GAs should assign more trust to crossovers earlier in the process where there are considerable degrees of freedom, and less near the end of the iteration cycle when a larger percentage of the potential solution has been found. The degree to which we trust crossovers might be the probability that cells are exchanged, or the number of cells exchanged.

The Fitness function is a measure of how well the solution fits the requirement. In the case of Sudoku we are very lucky in that most Sudoku puzzles' Givens are designed such that there will be only one solution which fills the board. So should a fitness function simply not find any blank cells, then the better solutions have lower values and the final solution has a value of 0. Conversely the fitness function might count non blank cells in which case the better solutions have higher values and the final solution has a score of 9x9 = 81. Trust in the fitness function is implicit.

The size of the population has a range of effects. We want to introduce enough variability so that the initial spread of the population captures enough of the potential solution space. However each member of the population tracks their own solution state, and so resource consumption might become prohibitive fairly quickly. Note too that each member of the population is traversed for each generation and potentially used and replaced, therefore we're sliding towards the same non linear performance factor that we are trying to avoid by not pursuing a brute force implementation. In any case, scaling the population to be as large as possible is the easiest way of helping the probability of success.

To what scale the members of the population are selected to reproduce, and how many children are generated from them is a specific requirement of GAs. For example, if we wanted to involve the entire population in each iteration, and we assumed that two parents each produced a child, we might then select the top two thirds of the population as parents and replace the weakest third with their children. Then again, if there was a degree of randomisation in the way the parents combined in the child we might produce two children from each pair; in this case the bottom half is replaced by the children of the top half. Or we may find that this is too high a rate of change and we put more trust in mutation rather than propagation, in which case we might select the top quarter as parents for children replacing the bottom quarter, and leave the middle half unaffected by the propagation factor and instead varied by mutation alone. In short, the trust factor related to propagation appears to vary how many of the population reproduce and how many are replaced by children.

Finally an optimisation might be introduced between iterations to improve variability or further share the attributes of the most successful members. However it should be noted these optimisations are rarely attuned to much more than the fitness function, and they do not perform comparisons to other members in the population to the same degree as the main Heuristic. Optimisations might help redistribute solutions where they collect near a local maxima(crossovers). Or they might share attributes in a simple way via weak Simulated Annealing. In any case, the optimisations rarely retain state between each interleaved opportunity to affect the population, and so every member is often checked each time they are run. Once again we begin to approach the non linear costs of brute force processing, so there are real benefits to making the optimisations as efficient as possible. A key ability in this regard is being able to work from the namespace of the surrounding heuristic – i.e. – share the same boards.

Once again we note that these trust factors may not be constant throughout the solution lifecycle. Randomisation and spreading factors might be most significant early on to help stop hitting local maxima too early, while propagation factors may become more important as you approach a full board.

Some of the largest implementations of heuristics observed appear to be related to genetic algorithms attempting to solve massively complex problems. Jet turbine engines are the subject of development projects which are known to run to 5 years and into multiple billions of dollars[8]. Upwards of 100 variables are controllable in a design with more than 50 constraints and the resulting namespace has more than 10^{387} data points. In such a scenario an experienced engineer might take two months to produce a design. The genetic algorithm proposed a design with twice as many improvements in under a day. Obviously this doesn't occur in all cases but the scale of the problems which can be approached by a heuristics is notable.

[8] http://www.fortunecity.com/emachines/e11/86/algo.html

4.2 Simulated Annealing

Simulated annealing is based on the idea that matter will tend to a lower state of energy. A population will be allocated, most likely in the same fashion as with other EAs such as GAs. With each iteration each of the members selects a neighboring state. If the neighboring state represents a lower energy state there is a probability that a transition occurs for this member to this state and the process repeats. This probability of a transition is referred to as a temperature, and the temperature is often managed in such a way that the population cools – i.e. – the rate of change reduces with time.

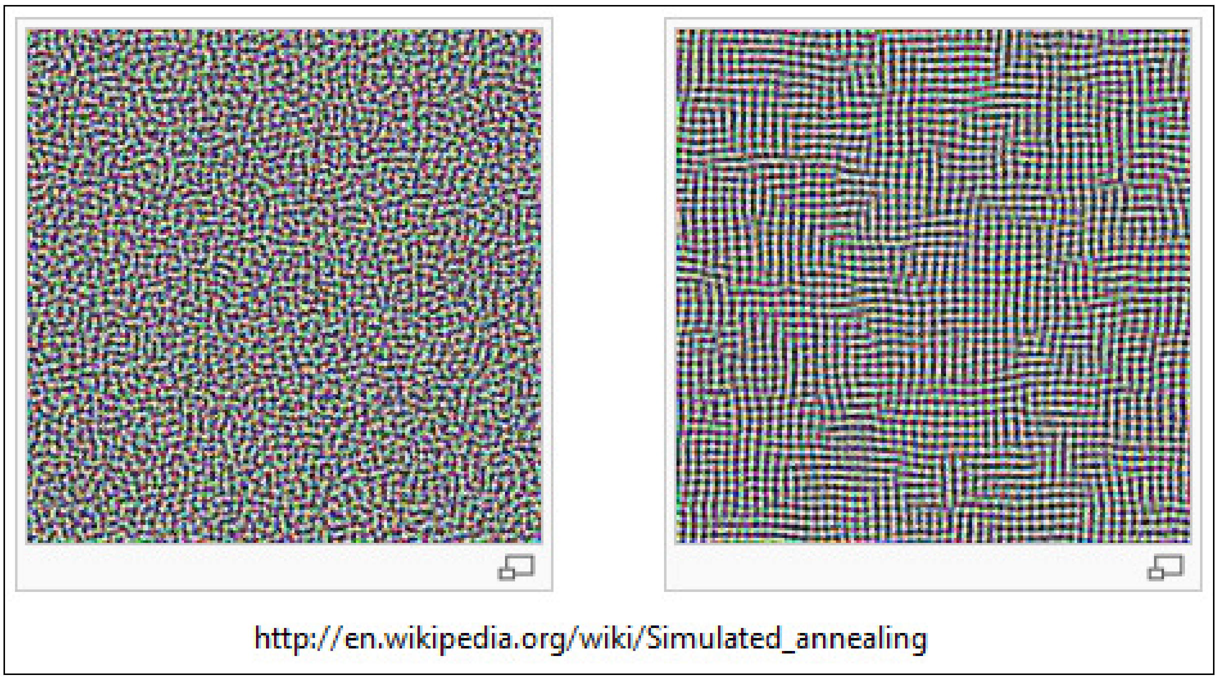


Figure 5 - Temperature Effect Example for Simulated Annealing

The variation of the temperature co-efficient with time is interesting in that it supports earlier statements we made with respect to managing the effects of change throughout the solution lifecycle. SAs have higher randomization effects earlier in the process, indicated by higher temperature probabilities, but this effect is reduced later in the process as they hopefully near a solution. The rate at which this reduction in randomisation takes place will be a consideration for testing, In Figure 5 - Temperature Effect Example for Simulated Annealing the SA is attempting to ensure that the nearest neighboring colours are different, while also matching the colours at slightly longer distances. The SA on the left was allowed to cool much faster than the SA on the right, creating a noticeable randomisation on the left and the apparent grid nature of the pixels on the right. This visually demonstrates the dangers of too much randomisation late in the process, as well as perhaps not running low temperatures for long enough later on.

Another consideration for randomisation for this and any heuristic is the number of purely random number added to the givens during initialization. Adding Randoms is an inexpensive way of distributing the population as far as possible before the Heuristic starts, but just as mentioned just adding Randoms will in most cases fail. So before we start the Heuristic iterations changes are purely random. As the Heuristic starts changes for improvement are highly random in probability. Then as the Heuristic proceeds the actions of change decrease.

A consideration for Simulated Annealing is Barrier Avoidance[9] where a local minima is so strong that it overpowers the transitions of members around it. An example of Barrier Avoidance, and cooling pressure, are Optimisations which help the algorithm avoid proceeding into a local minima too early. Geometric Crossovers for example help migrate aspects of the solution around the solution away from a point of congregation.

[9] http://en.wikipedia.org/wiki/Simulated_annealing

5. Swarming Heuristics

Swarming heuristics involves particles which attempt to improve their position by moving towards better performing neighbors.

A population of points are randomly thrown into the solution space. Each point knows its "location" and its "velocity". Points know their current solution fitness and their best solution so far. Points also know the fitness and best solution so far for their neighbors. *(Li, Tian, Hua, Zhong 2006)*

Neighbors can be defined by distance or relationships. Distance calculations for points can be computationally expensive. Relationships can be maintained via lookup tables. As particles converge either neighbor algorithm may yield similar results.

Each movement iteration combines the current particle 'velocity' and a randomisation factor with a range of trust factors including: current location, this particles' best solution so far, the positions of its neighbors, and the success of its neighbors.

Repulsive Particle Swarm Optimisation adds a factor for ensuring particles do not get too close together in the hope of avoiding local maxima. *(Perez, Marwala 2008)*

Geometric Crossovers are an important variant of PSO where the velocity component is replaced by the ability to crossover cells with other solutions. *(Moraglio 2007), (Moraglio, Di Chio, Togelius, Poli 2008).*

It is notable that most of the optimisations for PSO appear to be trying to redistribute the particles away from local maxima. Three separate optimisations were seen: a simulated annealing example, a propulsive factor which simply reversed the attraction to the best solutions, and geometric crossovers. Of the ease the most successful appeared to be geometric crossovers which coincidentally may have the strongest ability to redistribute the particles throughout the problem space.

Geometric crossovers deserve special consideration as they are not simply random; they are attempting to cross pollinate solution aspects between the members. The randomisation trust factor inherent in PSO remains so a balance can be managed between true randomisation and the crossovers; but each crossover operation is only using values which are different between each member and yet potentially part of the solution. One would expect therefore that the effect of crossovers is strongest earlier in the process as the particles were more widely distributed which matches the management practices around randomisation exhibited in heuristics and in particular in simulated annealing. I wonder if geometric crossovers would be implemented along with the notion of distance and attempting to ensure that the crossovers occurred between the most widely distributed solutions. The question of how to implement such an ability is of course problematic and I also expect that the time order of such implementation would be a expensive.

Noting that our research leveraged three different optimisations with three different PSO implementations, our PSO discussions will often lean towards to compensate the perceived weakness with the appropriate optimisation.

5.1 Particle Swarm Optimisation

A generic PSO algorithm might look like this[10] with a slight variation:

```
1) Initialise global fitness to worst possible
2) For each particle
    a. Initialise fitness to worst possible
    b. Initialise neighbor relationships
    c. For each dimension
        i. Initialise position
        ii. Initialise velocity
3) While iterating or stable:
    a. For each particle:
        i. Update the current fitness
        ii. Update the local best if current better
        iii. Update the global best if current better
        iv. Update the neighborhood best if current better
        v. Update the particle velocity and position.
            1. Add random factors towards the total, local and neighbor best solution.
    b. Optionally include Optimisation.
```

The algorithm is pleasingly simple at first glance. Particles have 6 aspects: position, velocity, random movement, a random movement towards the best solution so far, and a random movement towards the global best position, and a random movement towards the best solution of the particle neighbors.

Particle Neighbors come in two types: particles that are in similar solution states, or those that have been defined at initialisation as social neighbors. Either implementation requires additional processing. The distance calculations required to identify the particles closest to each other are expensive. On the other hand maintaining a lookup table relating social neighbors is also a computational overhead. Of the two I am more in favor of the social neighbors variant in that it would seem to help distribute the particles more effectively.

The trust factors we've deal with on other Heuristics are also applicable to PSO. All but the position aspect can each be weighted by trust factors that can be managed over the lifecycle of solving the puzzle.

The namespace requirement is mostly dictated by the number of particles. Each has a board that represents their present position, their best solution so far and a Velocity.

Note that PSO appears to have a much stronger idea of comparisons against the stronger elements, with three trust factors dedicated to the local global and neighborhood best solution so far. Each of these comparisons has particular considerations. The local best is the best solution that has been found in this particular particle, while simulated annealing only ever improves itself, the ability of PSO to retreat from lower local maxima would be an advantage not possible with simulated annealing. Global best solution hopefully be completed puzzle should be fairly obvious.

The implementation of the social neighborhood is an interesting one. If we use distance-based groupings and we would expect the particles to have higher cohesion factors and collect in groups, but this distance calculation is expensive and approaching n^2 time order. Maintaining a separate lookup table of social groupings is far more efficient, but has a downside that particles may be dragged considera-

[10] http://en.wikipedia.org/wiki/Particle_swarm_optimization#Pseudo_code

ble distance across a solution space simply because of the notion that social grouping rather than as a function of how much of a potential solution they share. I'm in favour of the social grouping context if another reason that I would have discovered population as much as possible and I would hope that with enough groupings within the population that it has a better opportunity of condensing within the same local maxima. This once again reinforces the idea that randomisation distribution of members of the population is a good thing, at least earlier on in the process

PSO Velocity

The per particle Velocity component looks to be highly variable and implementation dependent. At times it's described as a matrix operation, in other implementations it's described as a consistent change pattern, and in Geometric Crossover PSO implementations the Velocity component is even removed completely in favor of swapping some cells between solutions that would not violate the constraints. To generalise, Velocity varies all the way from being a consistent change pattern to non existence in favor of spreading the particles around.

An interesting consideration I haven't seen implemented is the possibility that the board cells could be numbered sequentially from 1 to (9x9 cells) x 10 (9 allowed values and a blank) = 810, and that the velocity could be a relatively prime value to 810 which would be added to the current position cell number each iteration. The operation is complicated by the requirement that no solution ever be invalid, so we would adapt by either: skipping the velocity induced change if it produces an invalid solution, or continuing to add the velocity component until a valid changed solution is found. A by-product of using velocity in this fashion would be an extremely wide distribution of movement, while still ensuring the entire namespace was being traversed.

The reason for hope that this might help lies in the optimisations that are usually applied to Particle Swarm Optimisation. PSO shows a tendency to accumulate at maxima, global or local. Attempts for unmodified PSO to solve Sudoku showed limited success, and results were improved with Repulsive PSO varients and Geometric Crossover Optimisations. These two changes, which I will describe in detail later, acted to help distribute the particles away from collecting locally. An implementation of Velocity in this manner may help achieve some of the same goals, and it would seem to be worth testing.

This of course leads to the question, is a proposal of this type devaluing the Heuristic by operating like a Traversal? Is this implementation too much like iterating in a Sudoku specific manner? Perhaps this is true. It's an interesting idea though.

In any case, variation on the operation of the Velocity component is a key point of variation among the PSO variants.

5.2 Repulsive Particle Swarm Optimisation

Repulsive Particle Swarm Optimisation involved adding a repulsive factor to the standard PSO implementation. This factor acted to repel particles from each other in the exact opposite fashion as the attractive force pulling particles towards the best performing solution.

Modifying our PSO algorithm from earlier for the repulsive behavior:

```
1) Initialise global fitness to worst possible
2) For each particle
    a. Initialise fitness to worst possible
    b. Initialise neighbor relationships
    c. For each dimension
        i. Initialise position
        ii. Initialise velocity
3) While iterating or stable:
    a. For each particle:
        i. Update the current fitness
        ii. Update the local best if current better
        iii. Update the global best if current better
        iv. Update the particle velocity and position.
            1. Add random factors towards and away from the total, local and
               neighbor best solution.
    b. Optionally include Optimisation.
```

Note there isn’t much of a change away from normal PSO, excepting we’re using both towards and away factors. Both these factors would seem to be good candidates for lifecycle management of the type we’ve mentioned before: increasing the randomisation factors (read repulsion) earlier and decreasing them as you’re hoping to close on the best solution. The balance of the attractive and repulsive factors is critical – should the repulsive factors remain overpowering for any period of time it’s expected that you would be driving away from a solution.

References to implementations offer policies PSO do not seem to indicate the high success factors. It is difficult to compare PSO implementations with each other when they occur between different studies and in some cases PSO implementations are compared to evolutionary algorithms. It is notable that in the case of propulsive PSO the results were seen as less favourable than the genetic algorithms while the reverse was true in the other references which also had what more than one heuristic. I do not see the the implementation of this repulsing factor is doing enough that isn't actually taking the particles away from favourable solutions or doing enough that more than normal randomisation. If on the other hand the repulsive factor was attempting to ensure that there was enough variation between the particles based on particle caparison or ensuring enough variation between sells solutions that this might make sense. However such a comparison would be expensive and we do not see examples of this being attempted.

To summarise, the results for Repulsive PSO appear to be an improvement on normal PSO, however it’s unclear if there was any additional improvement over simply applying a generic Optimisation (Simulated Annealing as an example).

5.3 Geometric Particle Swarm Optimisation

Geometric Particle Swarm Optimisation appeared to gain the best results of any of the Heuristics detailed in our research. The Velocity component is removed in favor of an optimisation called Geometric Crossovers which conditionally swaps values between solutions where the constraints aren't violated.

Once again modifying our PSO algorithm from earlier for the Geometric Crossover:

```
4) Initialise global fitness to worst possible
5) For each particle
    a. Initialise fitness to worst possible
    b. Initialise neighbor relationships
    c. For each dimension
        i. Initialise position
        ii. Initialise velocity
6) While iterating or stable:
    a. For each particle:
        i. Update the current fitness
        ii. Update the local best if current better
        iii. Update the global best if current better
        iv. Update the particle velocity and position.
            1. Add random factors towards the total, local and neighbor best solution.
        v. Randomly pick another particle.
            1. Attempt to exchange Randoms between the solutions that do not violate the constraints.
```

The net effect of the Geometric Crossover is to redistribute variants of solutions around the board, once again offsetting the tendency of PSO to want to congregate near maxima (local or otherwise).

Interactions with Givens aren't expected to be an issue because both particles are based on the same givens, and so these values will be the same for both particles and it will be obvious that there's no point swapping these cells as they'll be equal.

The focus of their implementation is based on the reported results that geometric crossovers when applied to PSO has the potential to solve Sudoku puzzles. When references made statements that puzzles were solvable, discussion sometime moved away from solving puzzles and towards other areas. As demonstrated in our applet example heuristics do produce what humans would recognise as good results. The only implementation which combined heuristics and optimisations with the promise of reliable results appears to be the combination of PSO and geometric crossover.

Of course having said that this paper hedges its bets. Our primary implementation in a tent is the PSO with geometric crossovers however the component-based framework we intend to use the implementation has high levels of reuse for other combinations of heuristics and optimisations some of which we haven't seen in other papers. Of particular interest we intend to consistently leverage areas of initialisation and life cycle management. By reusing these components a stronger comparison can be drawn between the relative contributions of heuristics and optimisations with each other.

6. Assessing Heuristic Components

In this section we extend from our review of sources and suggest a breakdown of the Heuristics into pieces that can be separately optimized. These components can then be reassembled in a framework improving a range of Heuristic and optimisation combinations.

Heuristic approaches appear to lend themselves to a componentisation and reuse. If the components are inputs to the heuristics, then improving the quality of the components increases the probability of success for the heuristics.

Each heuristic: is seeded with a board defining the problem givens, is seeded with random solutions including the givens, and it can be defined to work with the same representation of the Sudoku board.

The fitness function has more to do with assessing the constraints than any particular heuristic.

Optimisations appear to be interleaved between heuristic iterations, and can therefore be independent.

Trust factors such as randomisation may be consistent between heuristics. As well the mechanism controlling trust factor prioritisation over the lifecycle of a heuristic would be reusable if it remained configurable per heuristic. This is described in Figure 6.

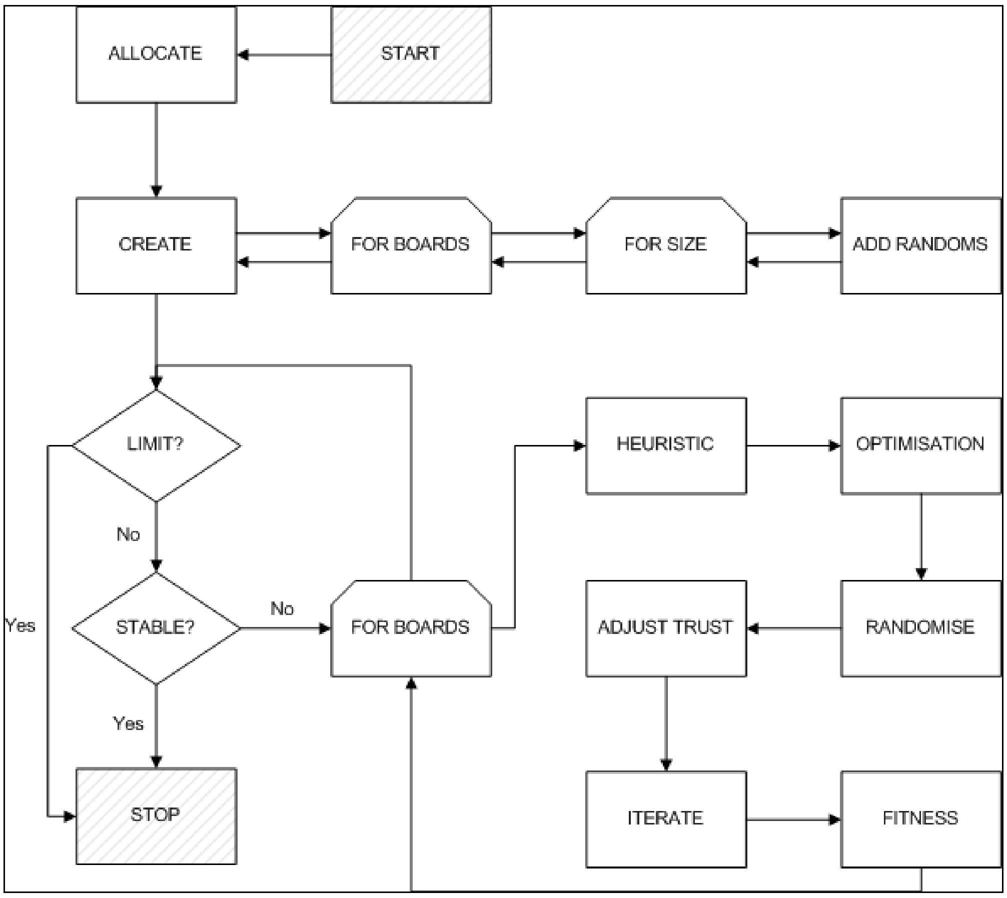


Figure 6 - Component workflow

6.1 *Why component design?*

In object oriented design there is often a comparison drawn between hierarchical implementations and component-based implementations.

Hierarchical architectures are obvious because support for inheritance and polymorphism are usually fundamental to the syntax of the object languages. So if you're beginning to program in Python or Java you can immediately experiment with creating base types of extending them; such as the common example of a furniture object that is then used to create a table and chair.

A different style is a component-based implementation which resist the urge to create trees and objects inheriting from each other. Instead the implementation focuses on working to interfaces, and this is the basis of the popular Spring framework, and as such the supporting argument is usually that component-based implementations are flatter and they have less run-time overhead are easier to understand or readily extended.

In our case we gain benefits from a component-based approach in a number of ways:

1) firstly because each of our components is intended to work through an interface and interoperate with other components in the standard way we benefit from a clear demarcation of responsibilities. In our model heuristics are not responsible for initialisation of the population instead reusable component will allocate the members of the population in a consistent way with consistent optimisations.
2) Along with consistency in implementation of the components we can also benefit from consistency and management aspects such as reporting and life cycle management.
3) Finally, because hubristic is will be implemented to an interface they will be interchangeable. This ability to mix and match, and test different combinations, will help us to compare heuristics with each other.

6.2 Empty Cells and Degrees of Freedom

An optimisation that is particularly successful during brute force solutions is working with cells with the lowest degrees of freedom. Working in open areas of the board is not necessarily helpful if it can be invalidated by cells which have fewer valid options.

Identifying cells with restricted degrees of freedom can be done by search mechanisms, however this devalues the heuristic approach and we will not pursue this avenue. The Fitness Function could contain a factor that favors solutions which are using cells with more populated constraints. Filling these cells earlier on in the process increases the flexibility available to the heuristic later on in the process. This also delays the appearance of local maxima.

Related to the discussion regarding the fitness function is the concept of degrees of freedom in the constraints which we have highlighted as a potential major benefit. One of the key action bits used to identify the highly difficult and/or unsolvable problems is their ability to defeat humans searching abilities reducing the participant to a trial and error approach much like an optimised brute force implementation. One of the optimisations which give such an advantage to human participants is the searching ability to preferentially select cells which reduce the number of choices which may be needed later on. This choice is often based on preferentially filling constraints which have the least number of empty cells. In other words we prefer working with constraints which have lower degrees of freedom.

The impact of targeting these cells with these constraints is that we reduce the number of selections in open areas of the problem which would be defeated by later comparison back to the busy constraints this eliminates retry and backtracking. We will not have the opportunity to implement a search algorithm as a human would use, but we do have the opportunity to value solutions which preferentially fill busy constraints.

Visually this process looks as though it's leaving regions and roads and columns of the puzzle empty the later work.

Note random interactions with the solution should include the ability to clear a non-given cell. If as planned the fitness function is modified so that filling constraints is preferred, then changes which clear cells in less populated areas will also be preferred.

In the same mechanism processing constraints should not be expensive or it could rapidly increase processing time. It also said that we were attempting to avoid namespace requirements beyond physical representations of the board so as yet we do not have a recommendation to how this would best implemented. The heuristic could maintain account of the number of cells which had been allocated in each constraint this would remove the requirement to count them each time by the text action bits to manage above the requirements of the physical board.

6.3 Random Candidate Solutions

The size of the initial random solutions is expected to be significant.

Consider the process of adding valid random numbers to a board. It is expected that the number of valid combinations remaining drops as numbers are added to the board. However the rate at which combinations are eliminated is relative to the complexity of the problem presented by the numbers placed, as stated earlier.

Therefore it is expected that a board might transition from many possible solutions, to one valid solution, to no valid solutions, as random numbers are successively added. The point at which no more random numbers can be added matches the situation for a heuristic where a local maxima has been found. This is indicated in Figure 7.

A higher degree of randomisation in the initial population distributes the population and particles as far as possible across the namespace. It is expected that this also improves the benefits seen from optimisations such as Geometric Crossovers as there are more cells to work from when comparing particles.

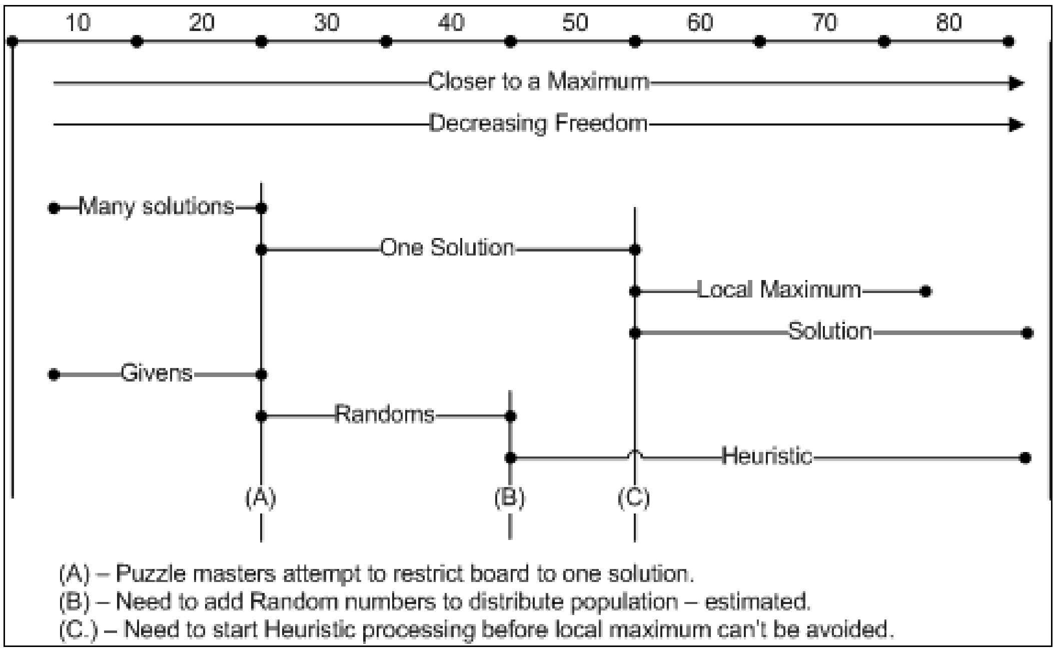

10
20
30
40
50
60
70
80
Closer to a Maximum
Decreasing Freedom
Many solutions
One Solution
Local Maximum
Solution
Givens
Randoms
Heuristic
(A)
(B)
(C)
(A) – Puzzle masters attempt to restrict board to one solution.
(B) – Need to add Random numbers to distribute population – estimated.
(C.) – Need to start Heuristic processing before local maximum can't be avoided.

Figure 7 – Considerations for Randoms

Just adding random numbers has a very low probability of success. Continuing to add random numbers before commencing the heuristic brings the solution closer to local maxima.

It is expected the optimum size of random solutions can be validated by attempting to solve puzzles with a brute force algorithm as Randoms are added. As the transition of through the states of "many solutions" to "one solution" to "unsolvable" is expected, we can generalize a good number of target Randoms for truly random solutions.

6.4 Optimisations Interleaving Heuristics.

When we look at the namespace requirements for Evolutionary and Swarming algorithms it becomes apparent that there are similarities.

The suitability of an optimisation for interleaving with a heuristic appears to be determined by its namespace and its behaviors:

Optimisations that do little more than work with boards are favored. It would be possible to see Simulated Annealing, or Geometric Crossovers used as Optimisations as they simply compare boards and swap values. Others like PSO which manage additional information such as velocities, prior solutions or neighborhood relations are unsuitable.

Optimisations that would need to discard solutions, such as Genetic Algorithms, may be inconsistent with the management requirements of heuristics that try and improve a solution over time like PSO. To continue to operate PSO would need to reset the particle from the deleted solution to the new one, which looses the value of the particle knowing its best solution so far.

As a result Geometric Crossovers, Simulated Annealing and Repulsive factors are candidates as optimisations because they are immediate while Genetic Annealing and PSO are not.

Evolutionary algorithms use populations of boards which vary by iteration.

Swarming Algorithms need to track: the current solution, the best solution so far, the current ‘velocity’, and any neighbor relationships (distance or social).

As a result it would be possible to perform an iteration of an Evolutionary Algorithm in between iterations of PSO if the EA used the current solution as its board. The reverse would not necessarily be true as the PSO’s extra details wouldn’t be available.

This explains why relatively stateless optimisations such as Simulated Annealing and geometric crossovers have been used as an optimisation for PSO.

It's worth repeating that we had every intention of testing various optimisation types with our PSO implementation and indeed against other heuristics as long as they conform to our component framework.

7. Our suggested Geometric PSO implementation

```
1. initialise global best as worst possible
2. initialise member social mapping table
      i. initialise each member social best
3. for each member of the population
      i. initialise member best as worst possible
     ii. allocate member to social group
    iii. initialise bored with givens
     iv. while population of board is less than initialisation limit
            i. if random value in random cell is valid
                  1. set new value at cell
      v. calculate fitness function value
     vi. set personal best solution as current if required
    vii. set social group best if required
   viii. set global best if required
4. until limit reached or population stabilised
      i. reassess trust factors
     ii. for each member of the population:
            i. while new solution is invalid:
                  1. calculate new solution given:
                        a. trust in a random factor
                        b. trust in our current position
                        c. trust in our previous best solution
                        d. trust in the global best solution
                        e. trust in the social group best solution
                  2. calculate fitness function value
                        a. for each cell on board:
                              i. for each constraint for that cell
                  3. if the new solution is valid
                        a. update our best previous solution
                        b. update the global best solution
                        c. update the social group best solution
    iii. for each member of the population:
            i. choose another member of the population at random
           ii. identify cells which are different between the two solutions
          iii. for geometric crossover limit:
                  1. choose a cell that is different
                  2. if this is our value would be valid in the original member:
                        a. move the cell value to the original member.
     iv. report if the global best solution has improved
      v. terminate if the board has been filled.
```

7.1 Breaking down the implementation

```
1. initialise global best as worst possible
```

Here we track the best solution found so far. Default in the best solution to be terrible ensures that it acquires the first member solution and any improvements on the initial value. At best solution type will be represented as a board and its corresponding fitness function value.

```
2. initialise member social mapping table
     i. initialise each member social best
```

Here we create a list for associating members of social groups. Primarily each of these groups is simply a list of members, however we need to track the best solution found within these groups so far. Once again this best solution will be represented by a board at its corresponding fitness function value.

```
3. for each member of the population
      i. initialise member best as worst possible
     ii. allocate member to social group
    iii. initialise bored with givens
     iv. while population of board is less than initialisation limit
            i. if random value in random cell is valid
                  1. set new value at cell
      v. calculate fitness function value
     vi. set personal best solution as current if required
    vii. set social group best if required
   viii. set global best if required
```

Here we initialise each member of the population. First we initialise the best solution to this member. We include this member and a social group. Then we create the initial puzzle board.

There we randomly fill in values. As previously charmed this is a very important step as a significant factor in the distribution of the population. Should there not be enough randoms the population may be clustered and too close to local maxima. Should there be too many randoms we may be left with an initial state which will valid can no longer lead to a viable solution.

We continue by calculating the fitness function for our initialise member, and reset the best solution so far to be the first port state. The update the social group best if required and once again checked to see if this is the best member so far in a global context.

```
4. until limit reached or population stabilised
     i. reassess trust factors
```

This is where our component that manages trust factors adjusts at behaviours throughout the life cycle of the heuristic. For example the randomisation component may be more trusted earlier in the process and were so later on. Obviously to accomplish this goal is trust management ability needs to be able to identify the solution processes matured; whether this is based on a population count or a number of iterations or subtracted to be identified is presently not known.

```
ii. for each member of the population:
    i. while new solution is invalid:
```

Here we attempt to change the solution for each member. We said that changes we may attempt to make may lead to invalid solutions which we will not want. Should that happen we simply try again.

```
        1. calculate new solution given:
            a. trust in a random factor
            b. trust in our current position
            c. trust in our previous best solution
            d. trust in the global best solution
            e. trust in the social group best solution
```

This is obviously a fairly complicated calculation which represented fairly simply. We've previously mentioned that we are tracking each of these aspects, and that we apply a trust factor to each. Please do not forget random change also includes the possibility of clearing the value of a cell to be blank, so any cell blank or otherwise may have it's value changed and/or be made blank. The only cells remain unchanged are givens.

```
        2. calculate fitness function value
            a. for each cell on board:
                i. for each constraint for that cell
```

There is that this function calculation that we mentioned previously. More than just performing account of the population as used in some implementations with seen we'll also give extra value to cells whose constraints are more populated. They should hopefully fill constraints faster moving relatively open sections of board for later work. So rather than 81 cells each value of 1, we may have 81 cells each including factors from the three constraints related to that cell. Most likely these constraint factors will be based on the constraint population of power with the power to be identified. In any case larger fitness function values will be perceived to be better.

```
        3. if the new solution is valid
            a. update our best previous solution
            b. update the global best solution
            c. update the social group best solution
```

We only wish to change our state if we produced a valid result, otherwise we simply try again..

```
iii. for each member of the population:
    i. choose another member of the population at random
    ii. identify cells which are different between the two solutions
```

This is the start of our geometric crossover optimisation. For each member of our population we randomly pick another member and try and identify cells which are different between the two. Note that the given should be the same in both solutions so the only difference it should be inconsistencies between the randoms.

```
        iii. for geometric crossover limit:
                1. choose a cell that is different
                2. if this is our value would be valid in the original member:
                        a. move the cell value to the original member.
```

We are only interested in valid solutions, and so it checking the performing crossover should never lose in an invalid state.

```
    iv. report if the global best solution has improved
     v. terminate if the board has been filled.
```

Finally we check to see if we've produced a new best candidate, and if it successfully filled the board and solved the puzzle we could stop.

8. Conclusions

Particle Swarm Optimisation appears to have shown the best success rates when combined with Geometric Crossovers. Unlike Evolutionary Algorithms, the Geometric Crossovers aren't driving towards improving the solution and approaching a local maximum. The Crossovers are redistributing the particles without violating the constraints. As always, a balance is formed between the trust factors for approaching solution and the trust factors for randomisation. However in this case, randomisation extends to redistribution of the solution as well as incremental change.

The suggested component framework offers significant advantages often not apparent in reference implementations. Instantiation, lifecycle management, trust factors and interleaved optimisations can all be controlled and measured outside of any given heuristic.

We are obviously proponents of PSO which will be the focus of our work. However we are hoping to extend beyond any given heuristic into a more generalised approach to solving his puzzles. To a degree this is possible if the component-based implementation works; from a purely design perspective this would appear to be a reasonable goal.

By separating the components related to initialisation and the heuristics is hoped that the heuristics can become more generalised and reusable in other contexts without too much modification. Whether the component framework is generalised enough to enable the same will be a matter of validation but once again it is a valid design consideration for our implementation even if we only did PSO and geometric crossovers.

I have often wondered how much of the success of some of his implementations is directly attributable to their ability to scale the populations. To be honest the puzzle board is a very big place the idea that you would throw thousands of particles into such a confined space would seem to increase the probability of success as much as anything else. What is not obvious is how rapidly the nature of the puzzle leads to tens of thousands of variants; each one and capable of stabilizing around their own local maxima with slight variation. As discussed distance calculations are expensive to implement, so we will attempt to ensure that each of the particles has enough variation of each other that there is enough diversity in the members of the population revolutionary algorithm to there is always the risk your collapse into a fairly confined space. Some of the notations used to so do could puzzles look like an 81 character string which is obviously very friendly way of encoding aboard as far as the computer is concerned that doesn't lend itself towards human readability. The advantage of such an encoding is a string comparison is hopefully obvious in, and the possibility of leveraging this strength hasn't as yet been fully considered. It definitely bears more discussion.

Other most pleasing findings from research was the visual impact of the genetic algorithm applet. I would suggest that consideration be given to a human interface all reporting ability which should allow assessments of the solutions in a far more natural way. During some implementations we found the process of encoding the puzzles took longer than solving them. A simple input method for the puzzles would be a great help. Once again though these reusable components to any efficiency in the input output abilities are reusable among the heuristics and components.

8.1 Further work

A reference implementation of the component framework can be created for PSO with Geometric Crossovers. Care should be taken for development considerate of later extension. Instantiation and Trust factors can then be varied to maximise success.

Once this is completed new interleave optimisations can be developed such as Simulated Annealing. The framework configuration can again be optimised.

We would like to investigate various forms for the velocity component in particle Swarm optimisation. Some variants completely remove velocity, closely with other trust factors. We do wonder about the possibility that velocity might be something of an iterative function which helps distribute the members in the problems space. Such implementation distribute the particles more uniformly and help offset the tendency for clustering new local maxima, it would seem too much like Searching which does not suit the aim of the heuristic.

New heuristics can be implemented within the framework, and optimisations and configurations tested / optimised for each.

We would like to validate our assumptions regarding boards is the fundamental unit of our namespaces. We know PSO has been successfully implemented in graph representations, and some of the belief propagation systems and other heuristics use coding methods such as binary strings. Finally some implementations abandon physical considerations and instead focus on decision trees. A lot could be learned from successfully implementing these namespaces consistently within the framework.

We would then be curious about the possibilities of reusing our approaches in other problems such as the travelling salesman problem, or the knapsack problem, or indeed a range of other problems also known to be NP complete.

9. Acknowledgements

This study is the result of a proposal by Mehmet Orgun. Content is primarily based on a sequence of papers by Alberto Moraglio on Particle Swarm Optimisation. Genetic Algorythms discussion largely comes from by Sudoku Genetic Algorythm approaches by Mantere and Koljonen. Simulated Annealing by Lewis. Generalised Stoachistic approaches by Perez and Marwala.

9.1 References

[1] T. Yato and T. Seta. *Complexity and completeness of finding another solution and its application to puzzles.* Preprint, University of Tokyo, 2005.

[2] Alberto Moraglio, Julian Togelius *Geometric Particle Swarm Optimization for the Sudoku Puzzle.* University of Essex, UK 2007.

[3] R. Lewis. (2007). *Metaheuristics can Solve Sudoku Puzzles.* Journal of Heuristics Archive, 13(4), 387-401.

[4] Meir Perez and Tshilidzi Marwala - *Stochiastic Optimization Approaches for Solving Sudoku* 2008.

[5] Mantere, T.; Koljonen, J. *Solving, rating and generating Sudoku puzzles with GA Evolutionary Computation, 2007.* CEC 2007. IEEE Congress on Volume , Issue , 25-28 Sept. 2007 Page(s):1382 - 1389

[6] Darwin, C.: *The Origin of Species: By Means of Natural Selection or The Preservation of Favoured Races in the Struggle for Life*, Oxford University Press, London, 1859, A reprint of the 6th edition (1968)

[7] Alberto Moraglio, Cecilia Di Chio, Julian Togelius, and Riccardo Poli - *Geometric Particle Swarm Optimization* (2008)

[8] Helmut Simonis - *Sudoku as a Constraint Problem* – 2005 Imperial College London

[9] Ines Lynce, Joel Ouaknine - *Sudoku as a SAT Problem* – 2006

[10] Todd K. Moon and Jacob H. Gunther - *Multiple Constraint Satisfaction by Belief Propagation: An Example Using Sudoku* - July 2006

[11] Elwyn R. Berlekamp, fellow IEEE, Robert J. McEliece, Henk C. A. Van Tilborg - *On the Inherent Intractability of Certain Coding Problems* - IEEE TRANSACTIONS ON INFORMATION THEORY, VOL. IT-24, NO. 3, MAY 1978

[12] Xiangyong Li, Peng Tian, Jing Hua, and Ning Zhong - *A Hybrid Discrete Particle Swarm Optimization for the Traveling Salesman Problem* – 2006

[13] Gordon Royle – Minimum Sudoku http://people.csse.uwa.edu.au/gordon/sudokumin.php – *University of Western Australia* – 2009