Using Ant Systems to Solve Sudoku Problems

David Mullaney

School of Computer Science & Informatics
University College Dublin

Abstract

It has been shown in previous studies that Ant Systems have had reasonable success at solving NP-Complete problems, such as the Travelling Salesman Problem, and various Routing Problems. These classic computer science problems belong to a class of problems that are amongst some of the most interesting in mathematics, including the Sudoku problem, which has become very popular as a newspaper puzzle. As of yet, most approaches to automated solving of Sudoku problems involve brute force, or genetic algorithms. Given the success of Ant Systems with other problems within this class, it would be interesting to see how it handles these puzzles.

1. Introduction

Ant Colony Optimisation, as characterised by Marco Dorigo[1] in his ACO meta-heuristic, uses groups of simple agents, modelled as artificial ants, to find optimal length trails through problem graphs. The ants communicate information about solutions, or partial solutions, by laying 'pheromones' along the edges of the graph that yield good solutions. This allows later ants to use this information about the problem space to influence their path choice. This foraging behaviour, allows ants in biological systems to find the shortest path from the nest to a food source, or the quickest path around an obstacle.

In computer systems, we can expand upon the foraging behaviour of biological ants, and use the same system to build and combine partial solutions across many problem domains. In particular we will look at Sudoku, a specific case of the Latin Squares problem, using 9x9 grids. The mathematical properties of Sudoku are quite similar to the Travelling Salesman Problem[4] although the representation of problem space and the solution conditions impose a greater set of constraints upon the system.

1.1 Sudoku Puzzles

The Japanese Sudoku puzzle, often described as "the Rubik's Cube of the 21st century", achieved international popularity in 2005. Each puzzle consists of the 9x9 grid, which is subdivided into 9 3x3 sub-grids. The rules for completion are very simple, making the game easy to learn.

1. Every subgrid contains all 9 digits.
2. Every row contains all 9 digits.
3. Every column contains all 9 digits.

At the beginning of each game the grid will contain some 'start squares' which already have a value. These are then used to reason which value each of the empty squares should contain, and the set of start squares is determined such that it will provide a unique solution to the puzzle. So, like the Rubick's Cube, the rules are simple, the objective is clear, and inevitably the underlying mathematics are staggering. For the simple 9x9 version of the game, found alongside the crossword in every daily paper, there are 6,670,903,752,021,072,936,960[2] valid configurations for an empty grid, and just 1 of these is the solution that you're looking for. Moreover the general case of the problem has been proven to be NP-Complete[3] placing it amongst some of the most interesting problems in computer science.

When generating these puzzles it is important that the set of start squares ensure a unique solution, so in general, puzzles are created by calculation a valid solution, and then removing a subset of the grid values, leaving just enough that the configuration has a unique solution. In general a puzzles requires at least 17 start squares to maintain its uniqueness, although this problem remains formally unsolved.

2. Implementation

2.1 Representation

Given the success of Ant Systems when solving the Travelling Salesman Problem (TSP) it seemed sensible to use a similar representation when attempting to solve the Sudoku Problem. We will represent each square in the 9x9 grid, using their (x,y) co-ordinates, like the cities in the TSP, and then we extend the grid along the z-axis to create a triple (x,y,value). This results in a set of 729 nodes which make up the 'piece set'. Using permutations of length 81 from the piece set allows us to represent every complete Sudoku puzzle. We can then apply the simple rules of Sudoku to ensure that only valid solutions are chosen.

Each individual ant operates a tabu search[4], visiting as many nodes as possible, without breaking the rules of the game. After each new edge is added to the trail, the list of unreachable nodes is updated and added to the 'tabu list'. Once the tabu list contains all of the possible nodes, the ant will calculate the amount of pheromone[6] which it should dispense along the trail, and distribute it. The amount of pheromone deposited on each edge in a given path is show in *Fig. 1*

$$P(t+1) = \Delta\left(P(t) + \frac{|trail(t)|}{81}\right)$$

Fig 1. ***P*** *is the level of pheromone, after a given trail update*
trail *is the length of a given trail*
delta *is the pheromone decay constant*

Fitness for each solution is given by counting the number of nodes that the ant has traversed, and a complete Sudoku problem will have a fitness of 81.

2.2 Simulation Platform

The implementation of this system is loosely based on an applet developed by Tobias Oetzel, which addressed the Travelling Salesman Problem(TSP)[5] using Ant Colony Optimisation. While the algorithms used by Oetzel were very helpful, the overall design of the applet made it difficult to extend outside its specific problem space, so a new tool was written based on Oetzel's core model, but without the overhead of maintaining a graphical front-end.

The tool was written in Java and used simple single-threaded agents to represent the individual Ants, allowing for performance optimisation when performing experiments with many ants, but without adding unnecessary complexity, given the simple decision-making required by the ants. The core search engine used was tabu search.

3. Experiments

The reason for choosing tabu search was mainly because it had been used reasonably successfully in Oetzel's experiments, and it also lends itself to problems that have a clear defined rule set. In this case a list of nodes that are included in the solution thus far is kept, and this list can be traversed, applying the simple Sudoku rule set at each node, to determine which nodes in the piece set are tabu, and which are still legal node choices. By enforcing the rules in this way we never search the areas of the search-space which are disallowed by the game rules. This does of course still leave a fairly large search-space.

Once the system had been adapted to explore sudoku problems, some control tests were run to establish a baseline for comparison. The results for this control test are shown *Fig 2* below. As a mater of contrast consistency the zero-th run will be calculated using standard tabu search, as outlined already. The dataset being used is the'Top 95 Hardest Sudokus' as compiled by Geunter Stertenbrink[7].

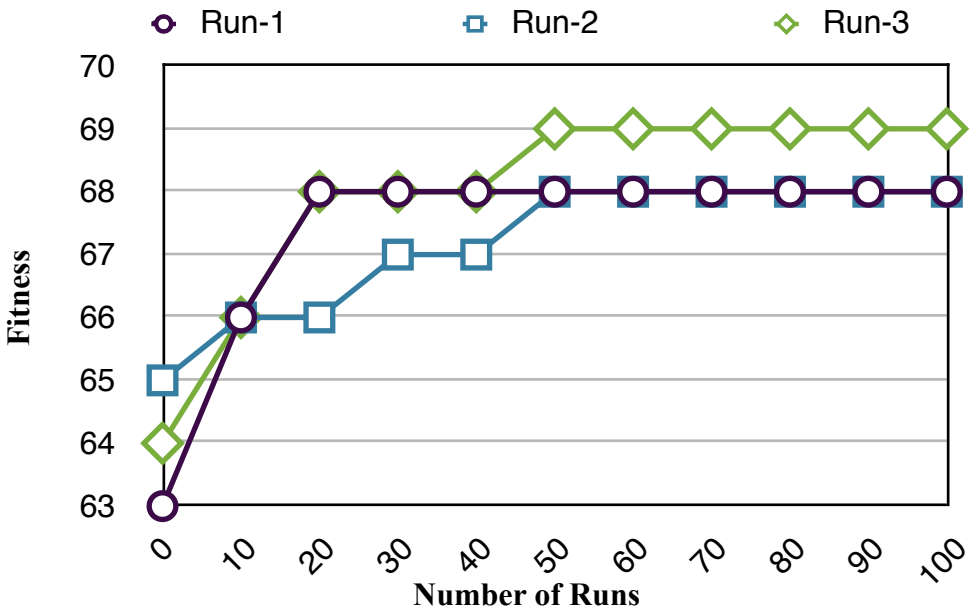

Fig 2. Control Results using Tabu Search
Run 1 — using 1 Ant, for 100 runs
Run 2 — using 2 Ants for 50 runs each
Run 3 — using 4 Ants for 25 runs each

As you can see, using simple tabu search yields pretty poor results, due to the essentially random nature, and the sheer size of search space. As a measure of how well the algorithm is performing it is best to look at the progress which has been made towards the solution rather then simply the total fitness. In the above case out best score would be 5.

3.1. Testing the Standard ACO

As an initial test of the system, three trials were performed, and the results are given in *Fig 3*. These trials were performed using 0.15 as the decay constant, which was applied at the end of each pheromone update operation. The tabu list update is performed after every edge traversal, and the next node is selected from the new list of acceptable nodes, with a 20% probability that pheromone trails will be ignored.

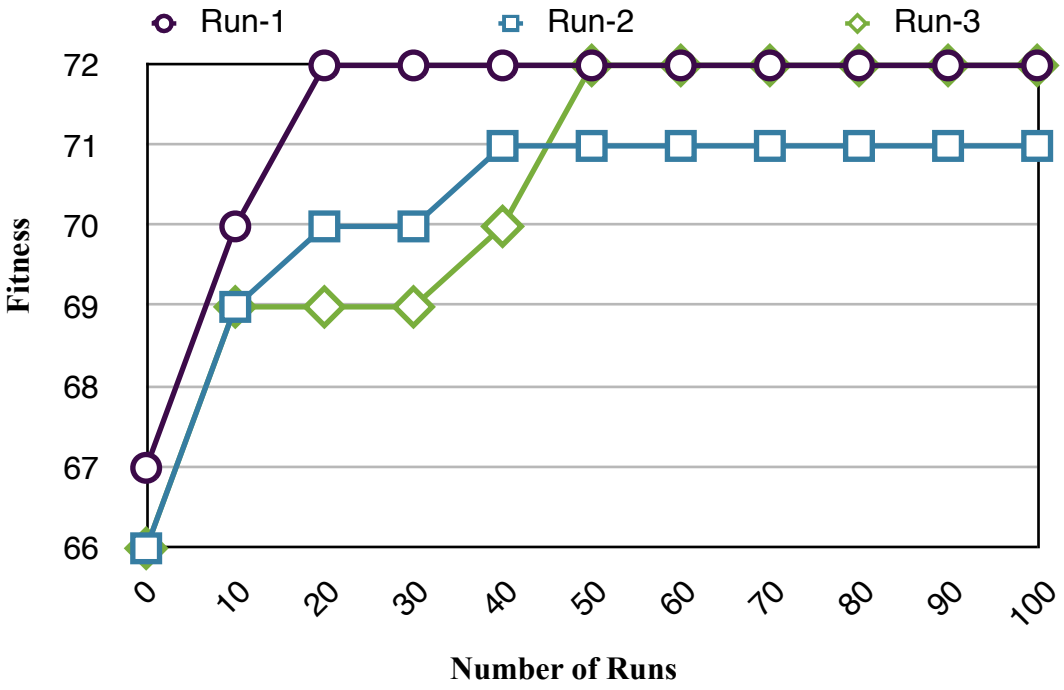

Fig 3. Results using Standard ACO

As we can see from the graph, the systems seems to find 'good' solutions early on, however, it seems to converge very quickly, leaving little room for exploration. That being said we can rate the performance of the algorithm, using the progress metric above, and we can see a 7 point increase in Run-1.

3.2 Shortest Tabu List

Although the previous set of tests seem to imply that ACO finds good solutions fairly early on in the run, and we know that the goal of the algorithm is to find the longest path through the puzzle, i.e. length 81, It seemed worthwhile to investigate whether simply choosing the path that results in the Shortest Tabu List (STL), might allow the algorithm to find the longest path more quickly. Obviously this method negates the need for multi-agent ant systems, so it's not an ideal solution, but for the sake of completeness, this method for solving the problem was carried out, using the same multi-agent design used in both the control and standard ACO experiments.

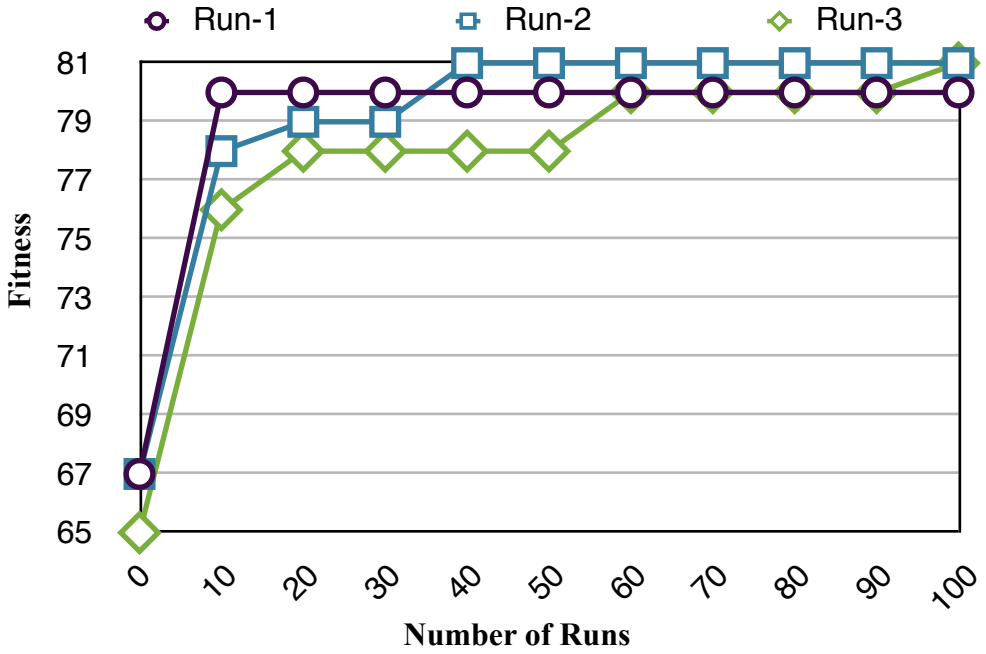

Fig 4. Results for the Shortest Tabu List approach

Fig 4 shows some strong results, hitting our first complete solution after just 40 iterations, using the 2 thread configuration. This of course proves that throwing serious amounts of CPU at a problem works well when dealing with complex problems like Sudoku, however, what the graph fails to convey is the sheer brute force required to use this kind of algorithm. When you consider that on the machine used to perform the above experiment, a single iteration took 68.46 seconds with the STL algorithm, compared to 0.58 seconds for standard ACO, and 0.42 for tabu search.

In an effort to exploit the obvious advantages of STL I propose that we replace the initial tabu search step, that we used to initialise previous experiments, with an STL search. Hopefully this initialisation stage can place a broad marker towards a good solution, which the artificial ants can follow. We can then use the lightweight ant search to refine the search, hopefully finding complete solutions without the huge cost of brute force.

This initial STL search will be carried out by a special case of the Ant agent, the navigator, and its behaviour can be summarised as searching for diversity within the graph, with respect the to the tabu criteria. For this reason, I will refer the this type of search as Navigator Initialised Ant Colony Optimisation (NI-ACO).

3.3 STL initialised ACO—Navigator Initialised ACO

As is evident from results, shown in *Fig 5,* that the navigator step in NI-ACO allows use to improve the search results, by defining a good trail at the outset, and allowing the ants to concentrate the search around the initial solution.

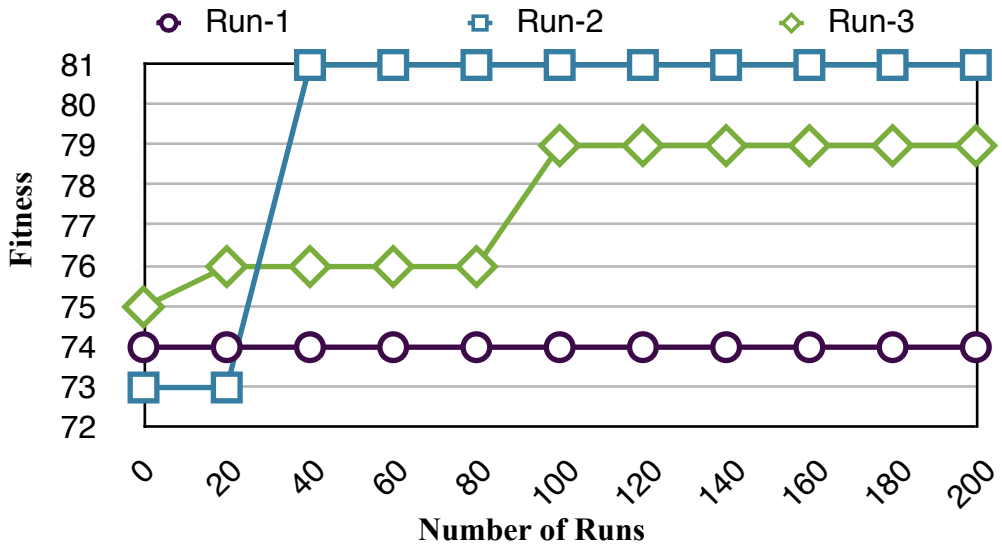

Fig 5. Control Results using Tabu Search
Run 1 — using 1 Ant, for 200 runs
Run 2 — using 2 Ants for 100 runs each
Run 3 — using 4 Ants for 50 runs each

In order to better understand the results we'll take a look at the behaviour of the algorithm, and see how this reflects the behaviour that we expect from ACO. In particular lets look at the fitness distribution using 4 ants, over 200 iterations.

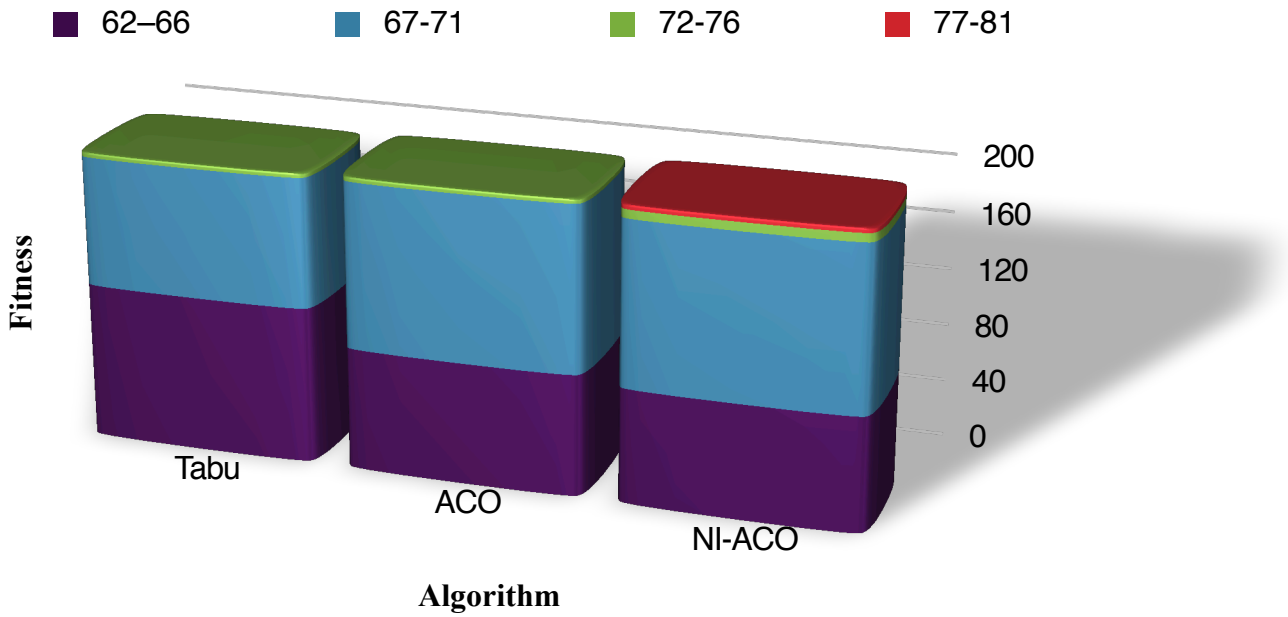

Fig 6. Distribution of Fitness across 3 key algorithms over 200 runs

Looking at *Fig 6* it is clear that although the overall performance of the NI-ACO algorithm is an improvement over the previous iterations, there is still a great deal of effort wasted on areas of the search space which yield poor fitness results.

3.4 Navigator Assisted ACO

In order to refine the search we will extend the behaviour of the simple creating a second class of agent. The original ant agent, the drone ant, will maintain the original behaviour, using a standard Tabu search for its first iteration, and and pheromone influenced ACO search for all of the subsequent iterations. In addition we will define a Navigator ant, which uses parameterised STL and ACO searched. As a base we will use 20% probability of using STL, and otherwise we use ACO

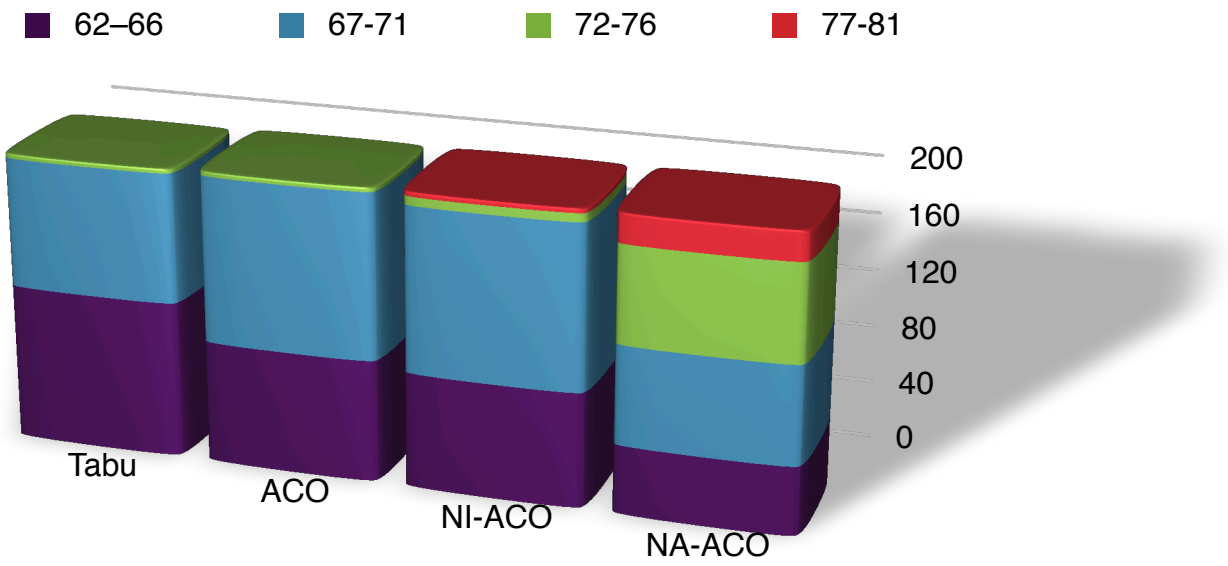

Fig 7. Distribution of Fitness, showing the improvement in performance when STL is combined with ACO

Fig 7 shows a distinct change in the distribution of fitness values as we introduce more structured search to lay the initial pheromone trail, and allow the presence of a Navigator ant to periodically enforce this pheromone trail. This have the effect of reducing the amount of time spend searching poor fitness areas of the search space. This does not however, prove the presence of communal learning that should be taking place, because we have yet to examine the change in fitness over the course of an execution.

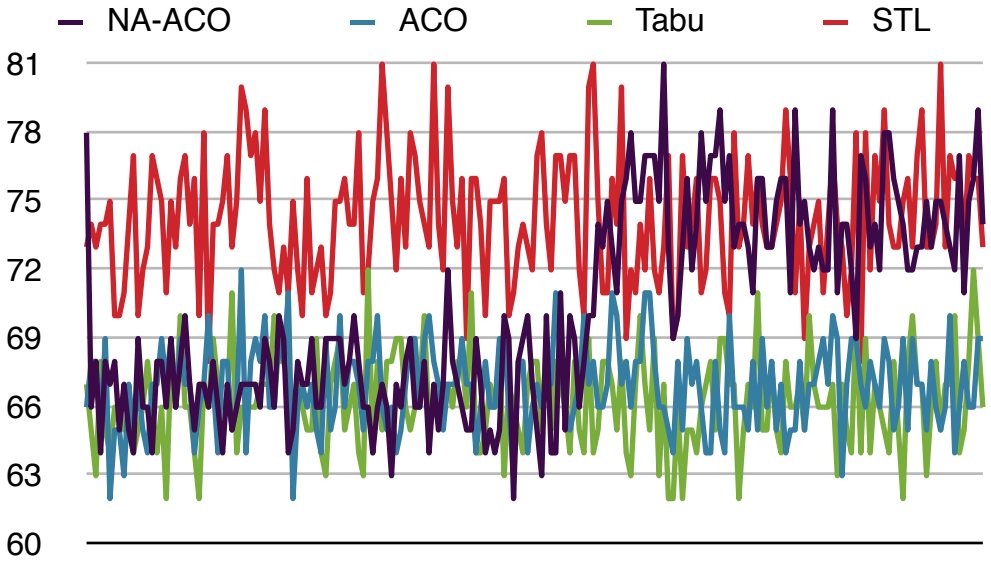

Fig 8. Distribution of Individual Fitness, across a 200 run execution, of each of the 3 key algorithms, suggesting that using a small number of complex agents can greatly improve the effectiveness of ACO

Fig 8 shows how each iteration, across the 200 iteration execution of each algorithm, performed when solving hard[1] sudoku problems. The data seems to suggest that after several iterations of the Navigator ant reenforcing strong trails—found using the STL algorithm— the overall performance of the algorithm is increased. The trails marked by the Navigator[2] would appear to help concentrate the ant's search around areas of high fitness. Although marking these trails is expensive, in terms of processing, relative to the basic ACO, the performance once the trail are established is comparable to the brute-force like STL algorithm.

4 Discussion and Future Work

It was established early on that Sudoku problems are not easy, and therefor it is not surprising that simple ant algorithms failed to produce startlingly effective results in such a short spaces of time—200 iterations or less. It also unsurprising that a brute force technique—requiring hours of process time—scores consistently better then a stochastic algorithm. What was surprising is how well the stochastic algorithm performed when given a small amount of training using brute force techniques. We saw how the fitness levels for the Navigator Assisted ACO shift from following the ACO trend to the STL trend, even though less then 10% of the ants were performing the STL-like algorithm, which is interesting if we consider how we might apply similar principles to other biologically inspired algorithms.

In Particle Swarm Optimisation, each individual particle has an associated velocity and position, which are used to calculate the trajectory of that particle. As the particle moves around the search space, the velocity changes based on the best fitness level seen by that individual particle (P-Best), and the best seen by any member of the swarm(G-Best). It might be interesting to see how the algorithm would perform if the system contained some particles which performed hill-climbs to seek out local optima, and used the swarming particles to local nearby peaks.

5 Conclusion

On the test set of 95 sudoku puzzles, we tested several methods for finding solutions. The most successful was the STL algorithm, which used brute force to find the longest paths through the puzzle, ensuring a majority of high fitness iterations. The STL was able to find the solution to every problem it was test against, but the runtime was approximately 120 minutes for each problem. Given that it takes most people less then 30 minutes to crack a sudoku puzzle, this is a pretty ineffective method for solving problems of this type.

Conversely the ACO algorithm was very fast, and took only 3–7 minutes to complete its execution, but was only able to find solutions to about 20% of the problems that it was tested against. This method may have had more success, were it left to run for the 100–120 minutes that the STL algorithm ran for, but experimentation of this type is outside the scope of this paper.

Finally the Navigator Assisted ACO, which took about 20–25 minutes to execute, was able to crack about 60% of the puzzles that is was tested against, making it only marginally less effective then human computation. This highlights the interesting behaviour of these simple ant actions, when combined with some other method that harnesses the domain constraints of the problem.

---

[1] Sample problems were chosen from the 95 hardest Sudoku Problems[7]

[2] The Average number of Navigator runs in a 200 run execution is 7.5–8.5%

References

[1] Marco Dorigo, Gianni Di Caro, Luca M. Gambardella. 1999. *Ant Algorithms for Discrete Optimization*. MIT Artificial Life. Available at http://www.idsia.ch/luca/ij_23-alife99.pdf

[2] Bertram Felgenhauer, Frazer Jarvis. 2005. *Enumerating possible Sudoku grids*. Mathematical Spectrum. Available at http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf

[3] Takayuki Yato. 2003. *Complexity and Completeness of Finding Another Solution and its Application to Puzzles*. Available at http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/MasterThesis.ps

[4] Pintér, János and Weisstein, Eric W. "*Tabu Search*." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/TabuSearch.html

[5] Tobias Oetzel. *Implementierung von stochastischen Optimierungsverfahren für das TSP*. Available at http://www.math.tu-clausthal.de/Arbeitsgruppen/Stochastische-Optimierung/tspapplet/applet.html

[6] Stephen Gilmour, Mark Dras. 2005. *Understanding the Pheromone System within Ant Colony Optimization*. Available at http://www.ics.mq.edu.au/~gilmour/publications/gilmour2005b.pdf

[7] Guenter Stertenbrink. *Sudoku Webpage*. Available at: http://magictour.free.fr/sudoku.htm