GROUND STATION SCHEDULING OPTIMIZATION

FOR A MODEL OF A REAL-WORLD PROBLEM INSTANCE

By

Bryce Wildish

Bachelor of Engineering, Ryerson University 2014

A thesis

Presented to Ryerson University

In partial fulfillment of the

Requirements for the degree of

Master of Applied Science

In the Program of

Aerospace Engineering

Toronto, Ontario, Canada, 2017

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

ABSTRACT

GROUND STATION SCHEDULING OPTIMIZATION FOR A MODEL OF A REAL-WORLD PROBLEM INSTANCE

Bryce Wildish

Master of Applied Science, Aerospace Engineering

Ryerson University, 2017

Effective scheduling of communication windows between orbiting spacecraft and ground stations is a crucial component of efficiently using spacecraft resources. In all but the most trivial cases, this forces the operator to choose a subset of the potentially available access windows such that they can achieve the best possible usage of their hardware and other resources.

This is a complex problem not normally solvable analytically, and as a result the standard approach is to apply heuristic algorithms which take an initial guess at a solution and improve upon it in order to increase its quality. Various such algorithms exist, with some being in common practice for this particular problem.

This thesis covers the application of several of the most commonly-used algorithms on a problem instance. Additionally, a real-world problem instance is used, and the resultant practical constraints are addressed when applying the heuristics and fine-tuning them for this application.

ACKNOWLEDGEMENTS

I must thank my supervisor Dr. Anton de Ruiter, without whose time, guidance, and understanding this work would not have been possible. Thank you also to exactEarth Ltd. for the information they provided, which was an essential starting point of this research.

Finally, thank you to my family for the support they gave while I focused on this thesis.

Table of Contents

List of Tables

List of Figures

List of Appendices

Chapter 1 - Introduction

1.1 Introduction

The successful operation of most orbiting spacecraft requires reasonably regular communication with one or more ground stations. This may be to directly receive control commands, as in the case of a surveillance satellite, to relay data, as in the case of an orbiting telescope like Hubble or WISE, or for other purposes. As the number of orbiting spacecraft in use grows, as it has done and will continue to do for some time [1], it is becoming increasingly important to be able to schedule and prioritize this communication, as the number of orbiting satellites – often far in excess of the number of ground stations, even for large, well-funded agencies like the European Space Agency [2] – can often stymie the ability for a spacecraft operator to remain in contact with all spacecraft at all times. This is further exacerbated if the range of locations of the ground stations is limited (for example due to budget restrictions or international politics) and the orbits of the spacecraft are such that the satellite only infrequently passes over those areas. Outside of the trivial case in which the number and location of ground stations is such that all spacecraft can be tracked whenever visible, it becomes necessary to plan communication windows with each spacecraft to ensure the most effective possible use of spacecraft resources to satisfy the operator's needs.

This is what is known as the "Ground Station Scheduling" problem. Depending on the specifics of the system, a solution will satisfy one or more criteria to varying degrees. One such example is to maximize the total useful communication time across all spacecraft, so that the minimum amount of time is wasted and to maximize the utility of all spacecraft. Another often-required criterion is to attempt to ensure that each spacecraft receives a roughly equal – or, in the case of differential requirements, proportional – amount of time with which to communicate, to avoid hindering the operation of some spacecraft at the expense of others. Other potential criteria include minimizing the number of conflicts, ensuring that no one spacecraft exceeds some maximum time between communication windows, or obtaining the maximum efficiency from the usage of ground station resources.

The ground station scheduling problem, more so than most other scheduling problems, is highly difficult computationally [3]. This is because the problem has many more constraints than is typical for a scheduling problem, and that the exact nature of these constraints varies strongly between problem instances [4] [5].

1.2 Requirements and Constraints

The classical elements of a ground-station scheduling problem are as follows:

- Access Window Fitness – If a scheduled communication does not fully intersect an actual visibility window between the involved spacecraft and ground station, it is effectively impossible, and thus is assigned no value
- Communication Time Satisfaction – How well a communication plan satisfies the communication needs of all spacecraft
- Conflict Avoidance – The degree to which a solution successfully schedules communication windows to avoid overlapping (conflicting), i.e. two spacecraft simultaneously being scheduled to communicate with one ground station
- Ground Station Usage – Because ground stations cost upkeep to operate, making as much use of the available stations as possible is desirable; this is the measure of how well that need is met in a given solution

1.3 Thesis Focus

The main focus of this work is to put several of the most common heuristic algorithms into practice, applying them to a model of a real-world problem instance, and comparing their relative performance in multiple criteria, such as final solution quality and relative computation time. With such information, it becomes more feasible to select which algorithm(s) may be most fit for a given application.

1.4 Thesis Contributions

In this thesis, a more detailed formulation of the problem is developed to properly account for several practical constraints. This formulation is then used to design a solver program which applies some of the above heuristic algorithms to problem instances, and its output is examined to perform the above analysis. Additionally, the data obtained is used to explore potential improvements to the solvers in order to yield better overall performance.

Chapter 2 - Literature Review

2.1 Problem Complexity

The high complexity of the problem (such as the large number of both variables and requirements of the final solution) [3], and the high number and often variability of the constraints [4] [5] mean that most traditional mathematical approaches, such as those mentioned in [3], become intractable and require infeasible amounts of computational time.

2.2 Heuristic Approaches

As a result of the computational difficulty of applying traditional mathematical solutions, almost all approaches seen in the literature for solving the ground station scheduling problem are heuristic methods. Such methods are those that, given input data regarding the constraints and requirements of the initial problem – in the case of a Ground Station scheduling problem, constraints like in 1.2, such as avoiding conflict and requirements such as satisfying access time – approach a solution. In most cases, some initial solution (or set thereof) is chosen, perhaps arbitrarily or with some preliminary tentative guidelines [4] [6], then successively improved upon – the process of which varies by algorithm and which will be explained in more detail later – until it is sufficiently capable of satisfying the requirements of the problem.

2.3 Fitness Function

Most approaches of the heuristic category create a *fitness function* with which to evaluate various solutions [7]. Under this evaluation, each criterion to be satisfied is transformed into a mathematical function whose output value is a reflection of how well the criterion is met.

In contrast to many heuristic approaches, which use a cost function which is intended to be minimized, the goal of a fitness function is to be maximized, i.e. a larger fitness value means an objectively better solution for the given criteria.

For example, whether the communication time requirements and the visibility window overlap would be termed the *Access Window Fitness*, and might be evaluated by defining that

communications that "fit" within the window are desirable while all others are not. Xhafa et al. use a method of this nature, depicted in Equation (2.1):

$$f(n) = \begin{cases} 1, if\ [T_{start,} T_{start} + T_{duration}] \subseteq AW(n_g, n_i) \\ 0, otherwise \end{cases} \quad \textbf{(2.1)}$$

That is, if the entire communication window, from T_{start} to $T_{start}+T_{duration}$, lies within the access window, the fitness function is correspondingly rewarded, being left zero for all other cases.

Once each criterion has been assigned a dedicated fitness function, these are then consolidated into a *Total Fitness Function*. This is usually done by simple summation with weighting factors applied [7], as it allows great freedom in selecting the relative importance – and thus influence on solution optimization – of each criterion.

Though the specifics of the problem may dictate minor changes to the function, such by modifying the individual weights, or by tweaking the individual fitness subfunctions, the general form is capable of accounting for most important concerns, including Access Window Fitness, maximizing communication time (*Communication Time Requirement Fitness*), reducing conflicts (*Clashes fitness*), Ground Station usage fitness, and more [4].

As the total fitness is intended to correlate directly to how well a prospective solution satisfies the problem criteria, it allows for an effective and comparable means with which to compare competing solutions during the heuristic process.

2.4 Exploitation vs Exploration

Heuristic solvers are dominated by two internal strategies, centered on how the solution space is to be explored, in order to best find the globally optimal solution**.** Each type of algorithm, including the ones discussed in detail later, is tailored to a specific balance of these two strategies, and may be modified to alter that balance [8] [7] [9].

The first, termed *exploitation*, is the process by which a solution is compared to its neighbors – very similar solutions, i.e. easily reachable with small permutations – and moves towards the one deemed most promising. This process repeats until a stopping condition, usually a local

maximum of optimization, is obtained. Exploitation covers a fairly small area of the search space, but is highly effective at finding the best solutions within that area. Conversely, however, it is prone to becoming trapped in local maxima, where all "nearby" solutions are less fit than itself, but where it is not necessarily similar in quality to the globally optimal solution. [9]

The second strategy, *exploration*, covers a much larger area of the solution space, either with multiple solutions being evaluated in parallel or with a (comparatively) freely-varying permutation process that allows for a single solution to move over a much larger search area. This approach is much less prone to entrapment within local maxima, but this often comes at the cost of either greater computational requirements (from larger populations which rather evidently require proportionally greater processing time to permute and evaluate in parallel) or a lower effectiveness at finding the optimal solution within a smaller area of the search space due to the higher tendency to jump to another area of the space. [7]

Ideally, a solution method would achieve some balance of the two strategies, in an effort to get the desirable traits of both without also suffering from their drawbacks [8].

2.5 Genetic Algorithms

By far the most common solution method found in the literature is some variation of a genetic algorithm, such as in seen in much of the work by Xhafa et al, like in [3], [5], and [9]. Genetic algorithms mimic the natural process of biological evolution, leveraging statistics of large populations to achieve an optimal solution, as a sort of "directed brute force" approach. They are commonly used to aid in the solving of computationally difficult problems, making them a natural choice for the ground station scheduling problem as well [4]. In such an approach, a population of randomly-varying "solution candidates" – i.e. collection of potential links, or communication passes between stations and spacecraft – is created, then evaluated based on how well they meet the constraints of the specific problem. The best fraction is then selected, with each solution candidate giving rise to one or more "offspring", slightly permuted from the original. This new pool of candidates is evaluated based on their ability to satisfy the requirements, and the cycle continues [10]. Each solution is encoded as a *chromosome*, which contains all of its inheritable traits (usually specific passes or elements thereof) [3] [11].

Genetic algorithms come in several forms. In the most basic form, each permutation is entirely random and no special inheritance process is applied. Additionally, with each generational cycle, the entire population of solution candidates is replaced with their descendants. More refined approaches modify the inheritance and population replacement behaviors, either to reduce computational time or improve effectiveness at finding the optimal solution.

Selection of the "surviving" solutions can come in various forms, including direct comparison, in which the worst-fit are eliminated, a roulette comparison, in which the probability of a solution's survival is tied to its fitness, and a tournament evaluation [10].

2.5.1 Advantages and Disadvantages

One of the main advantages of the genetic algorithm method of solving the problem is its relative ease of implementation; once a fitness function has been defined, it is comparatively straightforward to apply it to a varying population and compare solutions against one another. Another significant advantage is that genetic algorithms in a way run in parallel; because they operate on populations rather than individual solutions, there are effectively "more chances" to find an optimal solution and this is reflected in their increased ability to explore the search space [12]. Depending on the capabilities of the processing computer, and the implementation of the solving program, such parallelism might also lead to a faster solving time [13].

This approach does have some disadvantages; the most obvious is that being a variant of a brute-force approach, the number of required iterations can be large [5] [9], meaning a significant amount of computational time can be required. Another less-evident drawback is potentially more of a hindrance: evolutionary algorithms are a form of *low-level local search* approach [7], and as a result a solution can become "stuck" in local maxima, having iterated towards a solution that is more fit than its neighbors, yet not objectively ideal – i.e. not the global maximum – for the problem. This is worst for algorithms that do not have specialized mechanisms to combat it, such as the base genetic algorithm [12].

2.5.2 Crossover and Mutation Operations

In a genetic algorithm using this procedure, instead of the traditional "random mutation" method of generating offspring from a parent solution, the process is split into a *crossover operation* and a *mutation operation* [3]. The crossover operation is designed to ensure that parents pass on their "best traits" to their offspring, to improve the likelihood of the offspring being at least as fit as the parents [10]. The mutation operation is closer to the original process; it is usually fully randomized and small for any one generational change. However, the mutation process can be directed and consequently achieve higher effectivity [11]. One such example is an *adaptive feasible mutation function*, which is based off of some observations of real biological evolution, where mutations were observed in E. Coli bacteria, seemingly directed in such a manner as to better adapt the organisms to their environment [14]. Consequently, in the mutation model derived from this, mutations, rather than being entirely random, are applied so as to better adapt offspring to the environment. The result of this is an increased ability to converge onto optimal solutions [11].

Xhafa et al. evaluated the effectiveness of a crossover-and-mutation type genetic algorithm for the case of a single ground station and multiple spacecraft. Such an algorithm did indeed approach a solution that was far better than the initial state, but the fitness of the "near optimal" solutions was somewhat worse than those seen in more advanced genetic algorithms [3].

2.6 Steady-State Genetic Algorithm

Another variant of the standard genetic algorithm is the *Steady-State* algorithm. In this version, only a small proportion of the population is mutated each cycle, with the remaining offspring all being identical to their parents. The fitness criterion is also slightly relaxed, where less fit solutions are only less likely to survive rather than being eliminated outright. This approach trades off breadth of search for computing time; fewer mutations mean less processing but correspondingly less variety in the solving process [5]. This approach is unsurprisingly better suited to small problem instances (i.e. a relatively low number of ground stations and spacecraft), as larger populations take much longer for gene flow to propagate throughout [5].

Xhafa et al. found that the results from the steady-state algorithm, though potentially quite effective, are rather variable. In particular, it achieved varying performance for the different fitness types, more so than other heuristic approaches [5]. It was found to be effective, however, for small-size problems [5].

2.7 Struggle-Strategy Genetic Algorithm

Yet another variant is the *Struggle Strategy*, where each new solution candidate replaces the most similar older candidate, provided that it is the more fit of the two solutions. The level of similarity is determined by a *similarity function*, which returns a scalar value given two solutions; higher values correlate to more similar solutions. There are various forms of the similarity function, including Hamming Distance, comparisons in Vector space, and a *hash-based* similarity measure [9].

2.7.1 Hash Similarity

In contrast to other methods, which have quadratic computational time – $O(n^2)$ in traditional big-O notation – hash functions can be reduced to O(n) operations, greatly accelerating the process of finding the most similar solution to any other. [9]. Hashes are defined by a *key*, which can be calculated via several means, including based on fitness, position, or task-resource allocation [9] [15].

Xhafa et al. found that a Struggle-State genetic algorithm performed quite well, substantially better than a basic genetic algorithm [9]. Also in their findings was that a hash-based similarity function, if well-designed, gave the best performance, in a large part due to maintaining high solution diversity [9].

This method is highly effective compared to other genetic algorithms, but is computationally expensive: “This strategy is known for its effectiveness but suffers from a high computational cost. More precisely, given a new individual, finding a similar individual to it requires comparing against all individuals of the current generation.” [9].

This method is also very sensitive to the details of comparative evaluation, meaning that accuracy and effectiveness of the similarity and fitness functions is highly important to the success of the algorithm [9]. Also, like the Steady-State algorithm, it replaces only a small fraction of the population, and is ill-suited for large populations due to the increased time required for gene flow [15] Additionally, there is a rise in computational complexity [9] from needing to compare a greater number of solutions against one another.

2.8 Tabu Search

Another heuristic approach is the "Tabu Search". At first, it resembles a genetic algorithm: Given a starting condition, which may be chosen randomly or with certain preset heuristics [4], a space of solution variants is created, termed the *neighborhood* of solutions, by applying small perturbations (*movements*) to the original solution. At this point it diverges into a unique method, in particular with the *Tabu Status* and *Aspiration Criteria* evaluations to help direct the solution, with the presence of a *historical memory*, and due to the presence of *Intensification and Diversification* procedures.

Under the Tabu Status condution, already-visited solutions are "tagged" so as to prevent repeated visitation to the same solutions and avoid infinite program loops. The aspiration criteria are a set of criteria – whose exact nature are not defined by the authors, and possibly may be problem-specific – which can override the Tabu tagging and permit revisitation of solutions in the search of the optimal solution.

The historical memory consists of two components, a short-term memory of recently-visited solutions or movements, and a long-term memory about the entire solution process [4]This information is used to further expand the flexibility of the search, especially if combined with the intensification and diversification procedures.

Intensification and diversification procedures locally modify the fitness function, and are done to expand the range of the search and ensure it comprehensively covers the solution space. Somewhat of an inverse of one another, the former promotes uniformity among solutions, with

the frequency of a movement feature proportionally increasing the evaluated fitness, while the latter favors variety and penalizes the most common movements.

The main advantage apparent in the Tabu Search is that unlike other heuristic methods, it is much less vulnerable to becoming trapped in local maxima, and can better find the true optimal solutions [4].

As found by Xhafa et al., the Tabu search is very effective. Of particular note is that the process could always achieve 100% access window fitness – deemed the most important by the fitness function – and always allocated sufficient time to each mission [4]. Additionally, it converged rapidly, even the in the case of large-size instances, something not seen by them in other heuristic approaches.

The algorithm performed well in other respects as well; clashes were minimized to a high degree –less than 10% on average [4] – and even did fairly well optimizing ground station usage, especially when accounting for the much lower weight assigned to said fitness.

Furthermore, a Tabu search has a low deviation, meaning that its results are more likely to be representative of its true performance.

2.9 Simulated Annealing

The simulated annealing algorithm is another local (as opposed to population-based) heuristic approach. It gets its name from a similarity to a materials engineering process, where a material is gradually cooled and becomes more structurally cohesive. In the algorithm, the "temperature", i.e. the permissiveness of the permutation algorithm, starts high and is gradually reduced to progressively constrain further mutations in the solution [6]. The general design is to prevent premature optimization (and associated entrapment within local maxima) by initially permitting some movement from more-fit to less-fit solutions, with decreasing probability as the algorithm progresses.

This approach was found to be very effective by Xhafa et al., especially with regard to optimizing Time Requirement fitness. Such fitness was nearly always at a maximum, and

visibility window fitness was usually nearly as high [6]. It was also found to converge rapidly, more so than most other approaches [6].

2.10 Other Approaches

Several other approaches to solving the ground station scheduling problem have been explored in the literature, though they compose a minority of the total work.

2.10.1 Graph Coloring

The Graph Coloring procedure, examined by Zufferey et al., is based on the mathematics underlying the Tabu search algorithm [16]. It is similar to the Tabu search, but works in the solution space itself rather than the permutation space [16], and as a result is more efficient at solving the problem than a normal Tabu search [16].

2.10.2 Neighbor-Area Search

This process organizes the solution space into a tree, where each branch represents one data transmission schedule [17]. Lists of node neighbors are formulated such that the neighbors of node N_{ij} are $N_{i(j-1)}$ and $N_{i(j+1)}$ [17]. Neighboring nodes on the tree are then compared, with the best fit one being selected and used as a new point for further iteration. This process is illustrated in Figure 2.1:

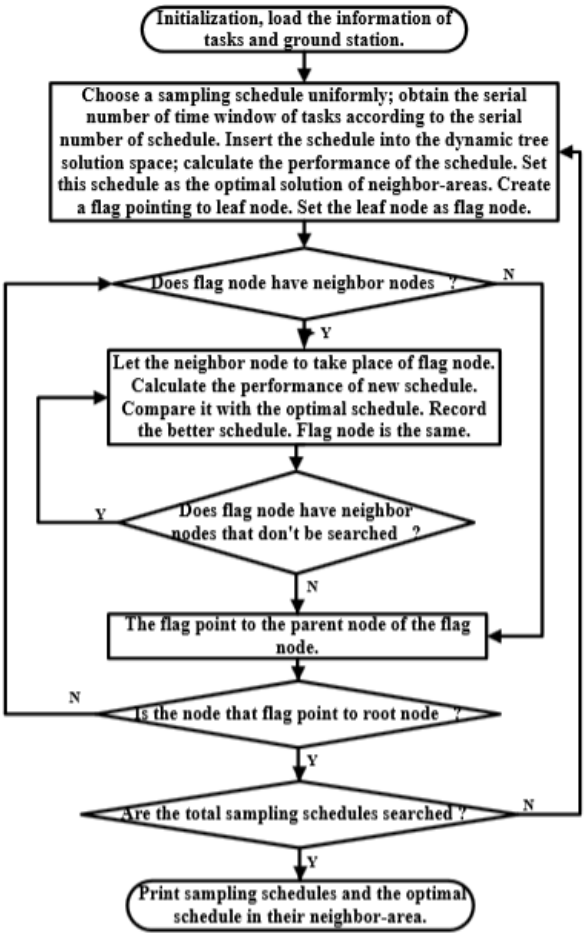

Figure 2.1 - Neighbor-Area Search Process [17]

This method is claimed to give better performance than traditional heuristic methods, in particular due to a reduced computation time without sacrificing solution quality [17].

2.10.3 Particle Swarm Optimization

Particle swarm optimization (PSO) is inspired by the flocking behaviors of several animals, including migratory birds and schools of fish [18]. Instead of a population being progressively modified genetically, each member, or particle, is moved through the solution space, keeping memory of optimal solutions for itself or its neighborhood as it does so. The movement is informed by a *learning mechanism*, which is the core feature of such an algorithm, as it is this that gives PSO such high efficiency. In a variant of the heuristic, termed the global version, the neighborhood of a particle is defined as the entire swarm, and so any high-fitness solutions found are available to all particles, i.e. that any particle can move to those solutions, regardless of their current state [19]. Particle swarm optimization offers comparable or even superior performance to other heuristics, both in terms of solution quality and solution speed [18].

2.10.4 Ant Colony Optimization

As its name suggests, Ant Colony Optimization is based on observed behaviors of social insects, such as colonies of ants, as they venture among a collection of targets. There are several variants of the process, the most successful and commonplace of which is the *Max-Min Ant System* [20]. In this process, a series of nodes, corresponding to visibility windows, are linked by a system of edges, corresponding to access conflicts. Each edge is mapped to a "pheromone trail", akin to the ones used by real ants, at first assigned to a maximal value. Next, some number of "ants" are placed on an arbitrary node. Each of these then moves to a different node, with the selection probability being informed by relative fitness values and the strength of the pheromone trail. Pheromone trails decay over time (i.e. with each iteration, according to a fixed evaporation rate), but are replenished by some or all ants traversing an edge, thus increasing the chance of future ants taking the path. After successive iterations, edges of greater fitness become more and more likely to be taken, and as a whole an ideal solution is approached [20]. This process can rapidly converge, and can yield very consistent results [20].

2.11 Comparative Analysis

In addition to their work examining the methods individually, Xhafa et al. have performed comparative evaluations, applying many different forms of heuristic analysis to various sizes of problem (i.e. number of ground station and spacecraft), including various Genetic Algorithms, simple Hill-Climbing, Simulated Annealing, and the Tabu Search [7].

There is significant similarity between methods regarding the fitness of the optimal solutions, especially for Clash Fitness. However, certain differences are noticeable. In particular, a pure genetic algorithm (termed GA base by the authors) performed the worst among all methods for access window fitness, and the Tabu Search performed visibly better for Clash fitness, enough so as to be the only method that stands out for that type of fitness. Overall, the Hill-Climbing, Simulated Annealing, and Tabu Search appear to be more effective than the other approaches, especially for Time Requirement Fitness.

None of the approaches were as effective for Ground Station Usage fitness, but this is quite likely a result of the dramatically reduced weight (only 10% of Clash, and 1% of Time Requirement) of that fitness type.

Regarding the standard deviation, which is a measure of the consistency of the algorithm's performance (i.e. how likely the results for other problem sets will be of similar quality) [7] [12], the Struggle-Genetic Algorithm performed markedly better than all of the other approaches, most of which saw similar standard deviations more than 50% higher. [7] [12] The hash variant of the struggle algorithm, by contrast, performed substantially worse than all others, with almost twice the deviation of the non-hash version. This is probably due to the sensitivity of the hash on the specifics of the starting problem.

Interestingly, base genetic algorithms seem to take far fewer generations to "plateau" into their optimal solutions [10] [3], and are generally regarded as being fairly rapid solvers compared to other implementations [11]. However, as discussed above, this comes at the cost of (and may be directly related to) reduced comparative performance. The one exception to this pattern is seen in

[21], where the algorithm took longer to converge but after doing so had achieved results rather better than seen for most basic GA cases.

2.12 Limitations of Existing Approaches

2.12.1 Similar Traits

Given the similarities between many of the heuristic methods of solving the problem, a logical approach to improving their accuracy and/or performance is in the shared processes between them, such as in the creation of the individual or total fitness functions or to increase the rate of convergence while still preventing premature optimization and entrapment within local maxima, that is, find a balance between exploration and exploitation strategies. [7]

2.12.2 Fitness Function Improvements

Because of the importance of the fitness functions for defining the optimality of a solution, they are a potential point of improvement. In particular, the fitness functions used in most evaluations, such as those by Xhafa et al. [4] [3], are somewhat simplistic and may not optimize as well for real-world applications.

For example, the visibility window fitness subfunction is a simple binary evaluation, where every visible position is treated as equal to all others. In contrast, real-life constraints may dictate that, for example, certain angles are undesirable (perhaps due to atmospheric interference on specific wavelengths).

Similarly, the time requirement fitness subfunction makes no preference as to the nature of the communication window, whereas in reality one might expect, for example, that many short windows are less desirable than one long window of the same total length (perhaps due to “link setup” time).

The ground station fitness function makes a similar assumption, that all ground stations are equally desirable. In reality, they may have wildly different operational costs, which is the root impetus behind this component to the fitness function.

2.12.3 Ground Station Capabilities and Positioning

Contrary to that which seems to have been assumed by the above evaluations, not all ground stations are of equal capabilities. Some may well have the ability to track multiple spacecraft simultaneously [4], some may have less flexible angles of visibility (either due to hardware limitations or due to legal specifics of their location), and some may have different data rate capabilities, meaning that the same amount of communication time between two different ground stations is not necessarily of equal value. As an extension, not all stations may even be capable of continuous operation, at least in the same capacity; some may, for example, be weather- or schedule- sensitive, being at reduced capacity – or even offline – at certain times.

Additionally, many evaluations used a single ground station case [3], or appeared to use a fixed set of ground stations [5] [7]. This is of importance because there is the unaddressed possibly that a radically different arrangement of ground stations – perhaps clustered within one country, for example – could significantly alter the relative performance of the heuristic analyses.

2.12.4 Aspiration Condition

One other point that does seem to require further examination is the "Aspiration" condition for the Tabu Search. Xhafa et al. did not go into detail regarding the nature of the criteria, instead opting only to say that it could change for different solutions [4]. In practice, the aspiration condition is usually simply "fitness better than ever seen before" [22], with little variation.

2.12.5 Real-Time Scheduling

The vast majority of the literature focuses on "static" evaluations, where a given solution for the scheduling problem will be used in effective perpetuity, but many real-world cases diverge from this. For example, spacecraft may need to be rescheduled as new ones are added or old ones fail, or where real-time events needing satellite involvement have immutable deadlines [23].

Chapter 3 - Problem Definition

3.1 Problem Description

The input to the problem is a precomputed list of passes of a set of spacecraft over a number of ground stations, with each pass having a start and end time, and an involved spacecraft and station. The goal is to optimally divide these passes among the available spacecraft and stations to achieve the maximum possible fitness, within the practical restrictions.

The general format of this list was provided by exactEarth Ltd., and is representative of the actual scheduling problem that is part of their normal operation. exactEarth is a company based in Cambridge, Ontario, whose primary focus is tracking seafaring vessels with orbiting spacecraft. As a result, the problem considered here is representative of an actual practical application, rather than the purely theoretical constructs usually found in the literature. This is also the reason that the chosen input was a predetermined list of passes rather than orbit and location definitions from which to compute those passes, as that step had already been completed.

3.2 Problem Formulation

Let $\boldsymbol{S}$ and $\boldsymbol{G}$ be collections of spacecraft and ground stations, respectively. (For ground stations with multiple receiving dishes, each is represented individually as a unique station).

A *visibility window* is a 4-tuple (t_s, t_f, i, j) where $i \in \boldsymbol{S}$, $j \in \boldsymbol{G}$, and $t_s, t_f \in \mathbb{R}$ with $t_s < t_f$ such that spacecraft i is visible to station j over the interval $[t_s, t_f]$.

A *problem instance* is a triple $\boldsymbol{P} = \{\boldsymbol{V}, \boldsymbol{S}, \boldsymbol{G}\}$ where $\boldsymbol{V}$ is the set of visibility windows for $\boldsymbol{S}$ and $\boldsymbol{G}$ over a given time period $[t_0, t_1]$ where for all visibility windows in $\boldsymbol{V}$ $t_s \geq t_0$ and $t_f \leq t_1$.

A *link* is defined as a 4-tuple (t_s, t_f, i, j) where $i \in \boldsymbol{S}$, $J \in \boldsymbol{G}$, and $t_s, t_f \in \mathbb{R}$ with $t_s < t_f$, and is said to be *feasible* if there exists a visibility window $(t_s^*, t_f^*, i, j) \in \boldsymbol{V}$ such that $[t_s, t_f] \subset [t_s^*, t_f^*]$ (spacecraft i is visible to ground station j over the link's duration).

A *solution candidate* $\boldsymbol{T}$ is defined as a collection of links.

A pair of links $(t_{s1}, t_{f1}, i_1, j_1) \in \boldsymbol{T}$ and $(t_{s2}, t_{f2}, i_2, j_2) \in \boldsymbol{T}$ are said to be *in conflict* if $[t_{s1}, t_{f1}] \cap [t_{s2}, t_{f2}] \neq \emptyset$ (their time intervals overlap) and either $i_1 = i_2$ or $j_1 = j_2$ (they share either a spacecraft orS station).

A pair of links $(t_{s1}, t_{f1}, i_1, j_1) \in \boldsymbol{T}$ and $(t_{s2}, t_{f2}, i_2, j_2) \in \boldsymbol{T}$ cause a *ground station handover violation* if $j_1 = j_2$, and either $0 < t_{s2}\text{-}t_{f1} < \delta_{j1}$ or $0 < t_{s1}\text{-}t_{f2} < \delta_{j1}$, where δ_{j1} is the minimum handover time for ground station j_1.

Similarly, a pair of links $(t_{s1}, t_{f1}, i_1, j_1) \in \boldsymbol{T}$ and $(t_{s2}, t_{f2}, i_2, j_2) \in \boldsymbol{T}$ cause a *spacecraft handover violation* if $i_1 = i_2$ and either $0 < t_{s2}\text{-}t_{f1} < \delta_{i1}$ or $0 < t_{s1}\text{-}t_{f2} < \delta_{i1}$, where δ_{i1} is the minimum handover time for spacecraft i1.

Set $\boldsymbol{C}$ is the set of all pairs of conflicting links in $\boldsymbol{T}$, i.e.

$$\boldsymbol{C} = \{\{L_1, L_2\} : L_1, L_2 \in \boldsymbol{T} \text{ and } L_1, L_2 \text{ conflict}\}$$

Given two conflicting links $L_1 = (t_{s1}, t_{f1}, i_1, j_1)$ and $L_2 = (t_{s2}, t_{f2}, i_2, j_2)$, the interval of time in conflict is given by $[t_a, t_b] = [t_{s1}, t_{f1}] \cap [t_{s2}, t_{f2}]$, and its length is equal to $t_o = t_b\text{-}t_a$.

A solution candidate $\boldsymbol{T}$ is defined to be possible in problem instance $\boldsymbol{P}$ if all links within are feasible, and no pair of links in $\boldsymbol{T}$ are in conflict or trigger either form of handover violation.

3.2 Practical Restrictions

The limitations imposed by the relevant spacecraft and station hardware, as well as real-world logistics, impose restrictions not previously addressed in the literature.

In particular, the following additional constraints are imposed:

- A maximum per-orbit ‘total transmission duration’ and maximum downlink duration
    - Transmitting data to the ground is very expensive on energy, and as a result power budget concerns put a hard limit on how much time, per orbit, the satellite can be transmitting data, as well as the maximum length of any one downlink.
- Unequal priorities for spacecraft and ground stations

    - Not all ground stations are equal; some are more costly to run than others, or more capable than others. Similarly, not all spacecraft are equal; some may provide much higher-value data than others. The result of this is that the same amount of communication time may not be of similar value for different craft or stations, and the fitness function must reflect this.
- Minimum downlink durations
    - As one would expect, there is a finite, nonzero time to establish a communication link between a spacecraft and the ground station. Additionally, exceptionally short links are generally viewed as a waste of time and summarily ignored. As a result, durations of insufficient length (for this problem instance approximately three minutes) are discarded.
- Station 'Handover time'
    - Due to the need for recalibration and tracking of the antennas on a ground station, it takes a specific amount of time to switch targets from one spacecraft to another. This time, a few minutes in duration, means that two passes that immediately follow one another cannot be utilized with full efficiency. It also means that if a station finishes tracking one satellite, but the next spacecraft's pass has less time remaining than this handover time, the window is effectively lost.
- Spacecraft 'Handover time'
    - Similar to station handover, if somewhat shorter-duration (around a few seconds to a minute), a restriction is present for satellite handover, i.e. switching which ground station to which it is transmitting.
- Link quality during the pass
    - Due to atmospheric interference, the quality of a link with a spacecraft diminishes as the elevation angle with it lowers. As a result, two passes of equal lengths, but at unequal elevation angles, are not equally desirable.
- Transmission in eclipse is to be avoided if possible
    - In eclipse, the solar panels on a spacecraft are non-functional, and as a result all power must come from storages. Because these have a finite capacity, and often restrictions on number or depth of cycles, transmission in eclipse is undesirable.
- Specific pass quotas

- Due to limitations such as available power, it may be the case that certain spacecraft have minimum or maximum counts for the number of passes per day or week that is desired from them. As a result, this places a strict restriction on the viability of some solutions.
- Stations can have similar limitations, such as due to provider-imposed bandwidth limitations, or maximum allowable staffing times.

3.3 Overall Fitness Function Definition

The fitness function used is defined in Equation (3.1):

$$F = W_{AT}C_{AT} - W_{CT}C_{CT} + W_{GS}C_{GS} - W_{DLV}C_{DLV} - W_{LLV}C_{LLV} - W_{HV}N_{HV}$$

(3.1)

With the following terms, as defined in table 3.1:

Symbol	Value	Description
W_{AT}	Access Time Fitness Weight	Relative importance of access time satisfaction
C_{AT}	Access Time Satisfaction	Total communication time, weighted for spacecraft value
W_{CT}	Conflict Time Weight	Relative importance of conflict avoidance
C_{CT}	Total Conflict Time	Total time spent "in conflict", where two passes of a station or spacecraft overlap
W_{GS}	Ground Station Usage Weight	Relative importance of maximizing ground station usage
C_{GS}	Ground Station Utilization	Ground station usage time, weighted for station value
W_{DLV}	Duration Limit Weight	Relative importance of obeying total communication duration limits
C_{DLV}	Duration Limit Violation	Total time over maximum communication time
W_{LLV}	Link Length Weight	Relative importance of obeying link length limitations
C_{LLV}	Link Length Violation	Sum of all excess link length (i.e. the portion of a link

		after the prescribed max length value)
W_{HV}	Handover Violation Weight	Relative importance of avoiding handover violations
N_{HV}	Handover Time Violation	Number of handover violations
W_{PC}	Pass Centrality Weight	Relative importance of maximizing average pass centrality
F_{PC}	Pass Centrality Factor	The average "centrality" value of all passes

Table 3.1 - Fitness Function Terms

In Equation (3.1), all weights are required to be positive real numbers.

3.4 Fitness Function Term Definition

D_L – Length of a link L (t_s, t_f, i, j) in set of links T, given by t_f-t_s

C – The set of all conflicting links. For each pair, the overlap time is given by t_0

L_{Si} – The spacecraft i involved in link L

L_{Gj} – The ground station j involved in link L

W_{Si} – Weighted value of a spacecraft i in set S

W_{Gj} – Weighted value of a ground station j in set G

M_{Si} – Maximum link length for a spacecraft i

M_{Gj} – Maximum link length for a ground station j

A_{Si} – Allowable total communication time for a spacecraft i

A_{Gj} – Allowable total communication time for a ground station j

S_{Si} – Total communication time for a spacecraft i across a set of links T

S_{Gj} – Total communication time for a ground station j across a set of links T

L_{SH} – Whether link L violates the handover time for its spacecraft

L_{GH} – Whether link L violates the handover time for its ground station

$$C_{AT} = \sum_{T} D_L * W_{Si} * W_{Gj} \quad \textbf{(3.2)}$$

$$C_{CT} = \sum_{C} t_o \quad \textbf{(3.3)}$$

$$C_{LLV} = \sum_{T} \max\left(0, D_L - M_{L_{Si}}\right) + \max\left(0, D_L - M_{L_{Gj}}\right) \quad \textbf{(3.4)}$$

$$N_{HV} = \sum_{T} \left(\begin{cases} 1, if\ L_{SH} \\ 0, otherwise \end{cases}\right) + \left(\begin{cases} 1, if\ L_{GH} \\ 0, otherwise \end{cases}\right) \quad \textbf{(3.5)}$$

$$C_{DLV} = \sum_{S} \max(0, S_{Si} - A_{Si}) + \sum_{G} \max(0, S_{Gj} - A_{Gj}) \quad \textbf{(3.6)}$$

$$C_{GS} = \frac{\sum_S \left(S_{Gj} * W_{Si}\right)}{\sum_T D_L} \quad \textbf{(3.7)}$$

3.5 Treatment of Practical Constraints

Many of the constraints can be formulated as new terms in the total fitness function, allowing for their inclusion automatically. For example, the handover restrictions are accounted for in their own term with their own weight. Larger handover violation times then lead to lower overall fitnesses, especially for high weights on that term. With sufficiently large weights, the minimization of handover violations becomes the most influential force in the determination of the solution, making the satisfaction of that constraint as likely as is possible. It should be noted that this does not guarantee such violations do not occur, but the very nature of a heuristic algorithm cannot guarantee a viable solution, only make it more likely with an appropriate

fitness function. Forcibly imposing such constraints during the solution process would negatively affect the performance of the solver in finding the best solutions. This is because it reduces its ability to explore the search space by closing off potential paths to ultimately superior solutions, i.e. that it is not necessarily the case that all solutions between two feasible solutions are themselves feasible, or even of the same quality.

Other constraints are applicable by modifying existing terms in the fitness function. For example, while the total communication access time is normally defined as a simple sum of links' lengths, like in Equation (3.8):

$$C_{AT} = \sum_{T} D_L \tag{3.8}$$

Here it has been defined with relative weight parameters, as can be seen in Equation (3.2).

A few constraints, in particular eclipse avoidance, cannot easily be inserted directly into the fitness function. In the case of eclipse avoidance, it can still be treated programmatically, with the solver automatically ignoring or truncating certain passes in the raw input data as needed. However, in this work, eclipse avoidance was ignored.

3.6 Goals

Three primary goals are present. The first is to near-optimally solve the problem instance, preferably such that the solution obtained is implementable in the real world, and of comparable or superior quality to a solution obtained prior.

The second is to perform a general comparative analysis on various heuristic algorithms to determine which approaches appear best fit for the given problem and likely those similar to it.

The third goal is to explore potential improvements on the existing algorithms or their application, to achieve better final fitness, or improve time efficiency. One example of this is to hybridize a solution to use different algorithms at different times, based on data returns from single-algorithm solutions.

Chapter 4 - Solution Approach

4.1 Solving Process

In order to near-optimally solve the problem instance, multiple heuristic algorithms will be used and their results will be compared. Using multiple approaches will afford a greater likelihood of obtaining very high-quality solutions which may be missed if only using a small subset of the available algorithms. This process will also lend itself to solving the second goal of comparative analysis, as the solutions between algorithms can be compared directly.

4.2 Implementation

To aid in the solving process, a computer program was written for automatically applying various heuristic approaches to the problem instance and obtaining solutions for each. The programming language chosen for the program was Java, chosen for its familiarity, its portability, and for the fact that its syntax is straightforward and very similar to many other languages, allowing for the source code to be as easily comprehensible as possible.

All solvers were implemented directly into the program, rather than using external libraries. This was primarily done to allow for greater control over the solvers' operation and to leave open the possibility for improvements to the algorithms, but also for reasons such as guaranteeing compatibility with the overall program.

4.3 Program Description

The program is modular, consisting of four main modules:

- Input (Problem Specification)
    - Receives specific data about the problem instance allowing for its construction in the solver.
    - Selects the desired heuristic approach.
    - Specifies any additional specific parameters that might be required (such as number of iterations permitted).

- Heuristic Solvers (Simulated Annealing, Tabu, Genetic, Genetic Crossover, Genetic Steady-State, Struggle-Strategy Genetic)
    - Given a solution candidate or population thereof, constructs a new solution candidate or population.
    - Given the fitness of the solution candidate or of the population, chooses a new candidate or population .
- Solution Analyzer and Evaluator
    - Evaluates the fitness of solution candidates given to it, returning them to the solver instance so that it can choose the most fit.
- Output
    - Collects data during execution (such as average fitness over time)
    - Exports the data in the desired form
        - Output can be in CSV form, for graphical analysis.
        - Output can be in raw text (TXT) form, for simple usage.

This structure was primarily chosen for its flexibility, as each module can have multiple variations to modify the behavior as desired. For example, the Heuristic Solver module has a variant for each algorithm, with a given form being used to apply that approach to the problem instance.

The implementation of the solution analyzer also benefitted from the modular design of the program, as concurrent programming is often complex and somewhat sensitive to the program state, and as such being a separate module simplified the design and made it more robust.

4.4 Program Overview

Because of the modularity of the program, the pseudocode has been split into multiple sections, one for the “general program process”, one for the fitness evaluator, and a third for the solver.

4.4.1 General Program Process

Input: Problem instance (spacecraft/station definitions, passes, et cetera) (*P*), Heuristic Solver (*S*), FitnessEvaluator (*E*)

1. Settings C = loadSettings()
2. SolutionCandidate sc = createStartingSolutionOrPopulation(P)
3. Output O = initializeOutput()
4. Fitness f = E.evaluate(P, sc)
5. N = 0
6. N_{max} = C.maxIterations; N_{min} = C.minIterations
7. O.recordData()
8. **while** ([N < N_{min} **OR** NOT(S.stopCondition(P, sc, f))] **AND** N < N_{max})
   a. sc = S.createChildSolutionOrPopulation(P, sc)
   b. f = E.evaluate(P, sc, C)
   c. O.recordData()
   d. N = N+1
9. **endwhile**
10. O.exportData()

Given a problem instance, the fitness function, and the program settings, an initial-guess solution is generated and evaluated, and passed into the solver, which runs until it either reaches an iteration limit or an internal stop condition. Each iteration, the permuted solution is evaluated and recorded by the output. At the end of the program, all output data is exported.

4.4.2 Fitness Evaluator

The fitness evaluator is the module that actually calculates the fitness of a solution or population (and each member within), based on the specified fitness function. This is called any time two solutions need to be compared, as well as for output purposes.

Input: Problem instance (*P*), Heuristic Solver (*S*), SolutionCandidate or Population (*sc*), Settings (*C*)

1. **for** all SolutionCandidate si in sc
2. localfitness lf = evaluateFitness(si, P, C)
3. S.recordFitness(sc, lf)
4. **endfor**

4.4.3 Heuristic Solver

Because the process used for the heuristic solver for evaluating, selecting, and generating solutions varies from algorithm to algorithm, the general process is explained here. Consult section 4.6 for details on each solver's implementation.

For a given solution candidate (or in the case of a population thereof, for each solution within), it requests individual fitnesses from the FitnessEvaluator. As the fitness evaluator computes the fitness of the solutions, it notifies the solver of the specific fitness of each candidate. Once the evaluator notifies the solver that it has finished all of its "requested" fitnesses, the solver will then, based on its underlying algorithm, use that data to select one or more new solution candidates as the next population.

4.5 Implementation Details

4.5.1 General Structure

Certain traits and procedures are general to the solver program and are shared by all modules. In all cases, the general structure is as follows:

For the given problem instance, the "first guess" solution candidate (or population) is generated. For simplicity, this first solution is the problem as-is, with every possible access time being used in all cases. In general, this solution has a large negative fitness, as it is usually physically impossible and has very large amounts of conflict.

Each iteration, the solution or solutions are mutated, and compared to their progenitors – i.e. the solution from which they were derived by permutation – within the solver's framework. In general, mutated solutions with higher fitnesses tend to replace the previous ones. Depending on the algorithm, not all solutions may be mutated, or not all solutions may be replacing ones of higher fitness, or their direct ancestors.

Once the new solution or population has been generated and used to replace the previous ones, the iteration repeats, until one of the stop conditions has been reached, either an iteration limit, an average fitness-per-iteration threshold, or one inherent to the solver algorithm. The vast

majority of these stop conditions are user-specified, either directly as in the case of an iteration limit, or indirectly via other parameters like annealing decay factor.

With each iteration, data such as fitness and solution time is logged for output as needed.

4.5.2 Program Cycles

Because the solvers include a large degree of randomness when mutating solutions, any one solution process risks being an outlier. To alleviate this, the program is designed to be able to run multiple successive attempts to solve a given problem instance, using the same initial guess each time. Though the number of cycles defaults to one to save computation time, more accurate data benefits from larger values, and some outputs like the statistics are mostly meaningless without it.

4.5.3 Chromosome Implementation

In the genetic algorithms, the chromosome implementation was such that a solution candidate was effectively its own chromosome; that is, its list of links were its permutable and transferrable traits, with each link being a ‘gene’. A crossover implementation, then, for example, would consist of transferring some number of links directly from one solution to another. A visualization of this can be found in Figure 4.1:

G1	Link 1
G2	Link 2
G3	Link 3
...	...
G_{n-1}	Link n-1
G_n	Link n

Figure 4.1. A solution as its own chromosome, with links serving as genes.

4.5.3 Mutation

There are five types of mutations that can be applied, and are selected using a weighted random (i.e. a random-number generator with unequal chances for each potential result). These mutation types, as well as their selected defaults, are listed in Table 4.1:

Type	Description	Default Weight	Default Chance
Link Shift	Translating a link, unmodified, by some amount of time; moves both the start and end times the same amount For a link (t_s, t_f, i, j) and the mutation amount d, the link becomes (t_s+d, t_f+d, i, j)	5	5/14.1 = 35.4%
Link Edit	Changing the length of a link, by moving either the start or end time by some amount For a link (t_s, t_f, i, j) and the mutation amounts d_1 and d_2, the link becomes (t_s+d_1, t_f+d_2, i, j)	7.5	7.5/14.1 = 53.2%
Link Removal	Deletion of a link entirely	0.1	0.1/14.1 = 0.7%
Link Split	Splitting a link into two links, at any point between 0.01% and 99.99% of the length; this limit is to avoid creating zero-length links Given a link (t_s, t_f, i, j) the new links become (t_{s1}, t_{f1}, i, j) and (t_{s2}, t_{f2}, i, j) where $t_{s2} = t_{f1}$	1	1/14.1 = 7.1%
Link Merge	Merging two links if they are consecutive and share spacecraft and station; the new link has the start time of the first and the end time of the second, making its total	0.5	0.5/14.1 = 3.5%

	length the sum of the two original links plus any empty time in between. Given two links (t_{s1}, t_{f1}, i, j) and (t_{s2}, t_{f2}, i, j) where $t_{s2} > t_{f1}$, the new link becomes (t_{s1}, t_{f2}, i, j)		
		Sum: 14.1	Sum: 100%

Table 4.1 - Mutation Settings

The weight, and thus relative probability, of each mutation type is configurable. Very dramatic changes like link removal generally have small weights, while small edits like a link shift have larger weights, and this is reflected in the chosen defaults. Also user-specified is the number of mutations to run per iteration and the maximum magnitude of a mutation. The default values are five mutations per iteration, of a magnitude up to 100. These defaults were chosen because excessively small values cause the solution to converge very slowly, while overly large ones generate a great deal of noise and can stymie proper improvement. As link times are usually imagined to be in seconds, a magnitude limit of 20, for example, would then impose a maximum translation of up to 20 seconds. Note that the sign of a mutation is random, i.e. any magnitude-type mutation can be applied in either direction. For example, it is equally likely to shift a link forwards in time as to do so in reverse.

Larger mutation values, and more mutations per iteration, generally accelerate the solution in that more space is explored more rapidly. However, by virtue of introducing much more randomization, it increases the risk of missing small-scale optima.

For the sake of simplicity and speed, all mutations are done entirely randomly within the assigned parameters; for example, any value of link translation up to the maximum is equally likely, as is the chance of any one link being selected should the chosen mutation be a link removal.

All randomizations are performed using the standard Java random libraries, which offer a built-in capability to generate pseudorandom numbers.

4.5.4 Output

There are four main output streams (one optional) to the program:

- Logger – general program state, status updates, debug info (if enabled), final results
- Mainline Output – transient fitness values; customizable format
- Statistics – overall solution statistical analysis, like average iteration time, improvement per iteration, or the average cycle fitness
- Profiling (optional) – Records the amount of time spent on different parts of the solution program, such as fitness evaluation or file I/O

The mainline output is customizable in format, determined automatically based on the specified output filetype; the two current implementations are CSV for generating graphs and TXT for raw text. In either case, the usual content is the fitness information on a per-iteration basis.

For long solutions that may generate excessively large files, the output density can be configured, so that the data is only logged every N iterations.

For each algorithm type, the statistics output records the following parameters, across all cycles, designed to expedite a comparative analysis of different solution types:

- Worst Final Fitness encountered
- Best Final Fitness encountered
- Median Final Fitness
- Average Final Fitness
- Fitness Range
- Fitness Variance
- Total Execution Time
- Total Iteration Count
- Average Time Per Iteration
- Average Time Per Cycle
- Average Improvement Per Iteration
- Average Improvement Per Cycle
- Average Improvement Per Millisecond

- Average Iterations Per Cycle

The profiler is a debug feature, designed to identify where execution time is being spent (or wasted). If enabled, it sorts time into nine categories:

- Init – Program initialization; making initial guesses, loading settings
- Fitness – Fitness evaluation of solutions
- Check – Solution somparison
- Mutation – Mutating solutions
- Iteration – Overall iteration
- Caches – Updating caches of best or other notable solutions
- IO – File I/O
- Logging – Generating log messages
- Output – Generating output data, compiling output file data

A typical profiler output would look as seen in Table 4.2:

	Total Time (ms)	Block Count	Fraction	Average Block Time
Init	93.902402	49	13.48%	2.751 µs/block
Fitness	347.360479	1566	49.86%	0.318 µs/block
Check	14.908991	4130	2.14%	0.005 µs/block
Mutation	19.415059	1565	2.79%	0.018 µs/block
Iteration	171.730296	5130	24.65%	0.048 µs/block
Caches	0.691046	1006	0.10%	0.001 µs/block
IO	2.868024	1213	0.41%	0.003 µs/block
Logging	0.136044	8	0.02%	0.024 µs/block
Output	45.632599	1000	6.55%	0.066 µs/block

Table 4.2 - Typical Profiler Output

A block is defined as one operation in a given category, such as one log message or one mutation.

4.6 Solver Implementations

4.6.1 Simulated Annealing

The simulated annealing solver is a standard implementation of such a solver like that seen in [6], where specific values for the constants (defined below) were tuned as they appeared to give the best results. The solver starts with a 'temperature' parameter initialized to 1.0. Each iteration, the temperature is multiplied by the decay factor 'α', which defaults to 0.95 – a common value in the literature [6] – but can be modified at will. At each step, the permuted solution candidate can be chosen over its predecessor if its fitness (F_{new}) is greater than that of the old (F_{old}), or, if it is not, with a chance proportional to the temperature (T), as per Equation (4.1):

$$e^{(\frac{F_{new}-F_{old}}{T})} \quad \textbf{(4.1)}$$

This formula is the standard annealing formula; as can be seen, the chance of a less-fit solution being chosen drops as it becomes more dramatically less fit, and as the solution progresses.

This process continues until the temperature reaches the chosen minimum value, defaulting to 10^{-6}, again chosen because it is a common choice [6].

The pseudocode for this algorithm is as follows:

Input: Initial Guess Solution Candidate (*sc*), FitnessEvaluator (*E*), Settings (*C*), Mutator (*m*), Output (*o*)

1. Temperature T = 1.0
2. **while** T > C.minTemperature
   - **a.** Fitness fprev = E.evaluate(sc)
   - **b.** improved = false, attempts = 0
   - **c.** **while** (attempts < C.attemptsPerStep **AND** [**NOT** improved])
     - **I.** attempts = attempts+1
     - **II.** SolutionCandidate scm = m.mutate(sc)
     - **III.** Fitness f = E.evaluate(scm)

IV. Chance ch = $e^{(\frac{f-fprev}{T})}$, range 0 to 1

V. **if** doWithChance(ch) **then**

VI. sc = scm

VII. improved = true

VIII. **endif**

a. **endwhile**

3. **endwhile**

4. T = T*C.decayFactor

5. o.logOutput()

6. **endwhile**

For as long as the temperature remains above the threshold temperature, the solver continues to iterate. Each iteration, N attempts are made to improve the solution (with N being the number of attempts per step specified in the settings). Each attempt, the permuted solution is compared to the parent. If more fit, or with a chance based on the fitness reduction, it replaces that parent. If this replacement occurs, the iteration is complete, and the temperature is multiplied by α and output data is logged.

4.6.2 Tabu Search

In the Tabu solver, the main solution determination happens not in whether to step to a newly generated solution or not, but in the generation of that solution itself. As opposed to the process used in simulated annealing, where in each step a solution candidate generates a variant against which it is compared, with a Tabu search a whole 'neighborhood' of solutions is generated and the best of those is chosen as a successor, which *always* replaces the original, irrespective of relative fitness. This is a standard form of such an algorithm, and is based on examples from the literature [22] [4] [24]. The determination of the best successor is based on three factors:

- If a solution is in the Tabu list (the list of visited solutions), it is not a valid successor, unless:
- If a solution meets the Aspiration Condition, its Tabu status is ignored

- Of all the solutions meeting the above two listed conditions in a given neighborhood, the one with the highest total fitness is selected

Here the aspiration condition has been defined as if the solution candidate has a fitness greater than has ever been seen before, i.e. its fitness value is greater than the maximum recorded across the entire solution process. This is very common aspiration condition, and it helps move the search to a more promising area of the space.

Additionally, with each step of the iteration, some caches of solutions are updated:

- A list of all solution candidates iterated through is maintained, with each new step added to the end of the list
- A shorter "recent memory" list is also added to, but continuously trimmed to keep its length below a maximum value (configurable and here defined, somewhat arbitrarily, as 12 iterations)
- The solution just iterated through is added to the tabu list. Periodically, the tabu list is cleared.

This iteration process continues until either the iteration limit has been reached, or there is a step where there is no viable successor, i.e. all solutions in the neighborhood are tabu and do not meet the aspiration condition.

An important note is that the current implementation forgoes some of the diversification and intensification procedures often seen in a Tabu search, as those usually involve restarting the search at a location midway through the last one's process, something not easily compatible with the overall design of the solver program.

The pseudocode for this algorithm is as follows:

Input: Initial Guess Solution Candidate (*sc*), FitnessEvaluator (*E*), Settings (*C*), Mutator (*m*), Output (*o*)

1. tabuList = EmptySet(), solutionPath = EmptyList(), recentPath = EmptyList()
2. Fitness maxFit = 0
3. ViableSolutions = true
4. **while** ViableSolutions
   a. Neighborhood N = generateNeighborhood(sc)
   b. Fitness f = E.evaluate(sc)
   c. ViableSolutions = false
   d. Fitness f1 = -Infinity
   e. Initialize “best fitness of neighborhood” to negative infinity (so anything is better)
   f. **for** all SolutionCandidate si in N
      I. Fitness f2 = E.evaluate(si)
      II. **if** (f2 > f1 **AND** [f2 > maxFit **OR NOT** tabuList.contains(si)]) **then**
      III. f1 = f2
      IV. sc = si
      V. ViableSolutions = true
      VI. **endif**
      VII. tabuList.add(sc)
      VIII. solutionPath.add(sc)
      IX. recentPath.add(sc)
      X. **if** (recentPath.size() > C.shortMemorySize) **then**
      XI. recentPath.removeFirst()
      XII. **endif**
      XIII. **if** (f2 > maxFit) **then**
      XIV. maxFit = f2
      XV. **endif**
   g. **endfor**
   h. o.logOutput()
5. **endwhile**

As long as available moves remain, the solver generates a population of permuted solutions for the current solution each iteration. Of these, the best is selected unless it is both tabu and not meeting the aspiration condition. Once selected, a solution is added to the tabu list and the memory caches, the availability of moves is reconfirmed and output is logged.

4.6.3 Genetic Solver (Basic)

The most basic genetic solver, intended primarily as a simple comparative baseline, applies only raw mutation, skipping improvements like crossover. In this approach, a solution candidate population of size N – configurable in the program, defaulting to 50 since it is large enough to be effective at diversifying the search without being so large as to greatly slow computation – is randomly generated from the input data by taking the initial guess and permuting it (by applying the settings-defined number of mutations) for each member of the population. With each iteration, each of these candidates generates N descendants, the most fit of which is selected as their successor, if it is also more fit than them. Once all N solutions have been replaced with their most-fit children, the iterations continue.

This solver has no specialized stop condition; it continues until the overarching iterator's stop conditions (such as max iteration count or lack of improvement in some number of generations) signals it to terminate.

4.6.4 Genetic Solver (Crossover)

A variant of the basic solver, the crossover solver inserts an additional step during the mutation process. Crossover has two variants; in the first, each randomly-generated offspring is crossed with its parent, with the parent transferring it part of itself. This is the approach used by Xhafa. et. al [3]. The second approach crosses between parents, before any offspring are generated. This approach is more similar to the real-world genetic process upon which crossover is based, but it is more logistically complex, in particular when solutions are not readily paired. For that reason, the first approach was chosen, where each offspring, once generated, is given one component of

its parent, with a component here being defined as one link (i.e. one connection between station and spacecraft).

Like the basic genetic solver, this process continues until externally stopped. A sample crossover operation, where a child is permuted and then crossover overrides part of that permutation, is depicted in Figure 4.2.

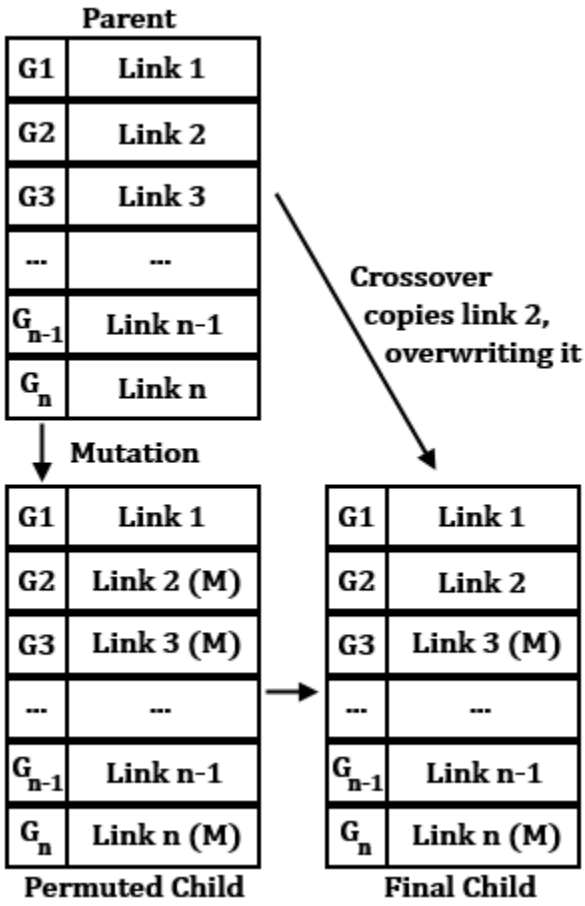

Figure 4.2. A Sample Crossover Operation.

4.6.5 Genetic Solver (Steady-State)

The steady-state genetic solver is an extension of the crossover solver, with one modification: Where in previous algorithms each solution is always replaced with its successor if the latter is more fit, in this algorithm there is only a chance for this to happen. This chance is up to the user to define, and controls the tradeoff between improvement and iteration time; here its value has been chosen to match much of the literature at 20% [5], so that roughly only one-fifth (less, if

there are solutions yielding no superior offspring) of the population is replaced in any one iteration. Like most other constants, this parameter is configurable.

Both the basic genetic solvers, and the steady-state genetic solver, have nearly identical pseudocode:

Input: Initial Guess Population (*P*), FitnessEvaluator (*E*), Settings (*C*), Mutator (*m*), Output (*o*)

1. **while true**
2. **for** all SolutionCandidate sc in P
   a. **if** (**NOT** isSteadyState **OR** doWithChance(C.steadyStateReplacement)) **then**
   b. Fitness fprev = E.evaluate(sc)
   c. Population children = emptyPopulation()
   d. **while** (children.size() < C.childCount)
      I. SolutionCandidate child = m.mutate(sc)
      II. crossover(sc, child) *(Note: Only in the Crossover and Steady-State algorithms)*
      III. children.add(child)
   e. **endwhile**
   f. SolutionCandidate best = E.getBest(children)
   g. Fitness f = E.evaluate(best)
   h. **if** (f > fprev) **then**
   i. P.replace(sc, best)
   j. **endif**
3. **endfor**
4. **endif**
5. o.logOutput()
6. **endwhile**

Until an outside stop condition is imposed, the solver continues to run, with each iteration generating a population of children for each member of the parent population (or fraction

thereof, in the case of steady-state). If applicable, crossover is applied, and for each child population the best-fit is selected and compared to the parent, replacing it if superior.

Crossover:

Input: SolutionCandidate (*sc*), Child SolutionCandidate (*child*)

1. Link l = pickRandomLink(sc)
2. child.replaceSameIndexLink(l)

4.6.6 Genetic Solver (Struggle)

In contrast to the above algorithms where each solution is replaced with its own offspring, in this solver a solution replaces that which it most closely resembles. [9] [15] This is done by comparing each the offspring to each member of the parent population, and finding the one with the highest 'similarity value'. This is done with a simple Hamming Distance, where two solution candidates are more similar if they have the same number of links N (marked as "entirely dissimilar" if they do not), and where each matching link adds 1/N to the similarity value (so two completely identical solutions have a similarity value of 1.0). Links are defined as matching if they have the same start and end points between the same spacecraft and stations. This is the implementation used by Xhafa et al [15].

The struggle GA pseudocode is identical to the crossover algorithm, except on the replacement step:

Input: Initial Guess Population (*P*), FitnessEvaluator (*E*), Settings (*C*), Mutator (*m*), Output (*o*)

1. **while true**
2. **for** all SolutionCandidate sc in P
   - **a.** Population children = emptyPopulation()
   - **b.** **while** (children.size() < C.childCount)

I. SolutionCandidate child = m.mutate(sc)
II. crossover(sc, child)
III. children.add(child)

c. **endwhile**
d. SolutionCandidate best = E.getBest(children)
e. SolutionCandidate replaced = findMostSimilar(P, best)
f. Fitness fprev = E.evaluate(replaced)
g. Fitness f = E.evaluate(best)
h. **if** (f > fprev) **then**
i. P.replace(replace, best)
j. **endif**

3. **endfor**
4. o.logOutput()
5. **endwhile**

This is mostly the same as the other genetic implementations, except for the replacement logic. In this case, the best child of each population replaces the most-similar member of the parent population, rather than their direct parent.

Similarity Evaluation:

Input: Population (*P*), SolutionCandidate Target (*sc*)

1. maxSimilarity = 0
2. SolutionCandidate closest = none()
3. **for** all SolutionCandidate si in P

a. **if** (si.linkCount != sc.linkCount) **then**
b. similarity = 0
c. **else**
d. similarity = countIdenticalLinks(si, sc)/si.linkCount
e. **endif**
f. **if** (similarity > maxSimilarity) **then**

    g. maxSimilarity = similarity
    h. closest = si
    i. **endif**
4. **endfor**
5. **return** closest

To find the most-similar member of the population, it is iterated over. For each member, a similarity fraction is determined based on the number of matching links between the two populations (zero if the link counts differ). The population member with the highest fraction is selected.

Chapter 5 – Results and Analysis

Sections 5.1 to 5.7 primarily contain the raw results; most of the analysis of these results can be found in Section 5.8.

Details of problem definitions and a sample portion of a problem can be found in Appendix A. The generation of random problems is detailed in Appendix B.

A problem instance is defined by a set of definitions for all involved spacecraft and ground stations, specifying the relevant parameters such as time requirement, weight value, and limitations, and a list of all potential passes.

5.1 Setup

For each solver type, 40 cycles (full solving processes) were run consecutively on the same problem instance, i.e. 40 solution attempts were performed on the same problem. The same solvers were then run on 40 randomly-generated problems to increase the likelihood of statistical validity of the results. In both cases, 40 cycles were run for the same reason; while any individual solution attempt may underperform, running a large number makes it very likely that the behavior seen is representative of the actual nature of the solvers.

The solvers were run on a computer with an 8-core, 3.5GHz i7-3770K processor, 12GB of 1800MHz DDR3 RAM, and a 6 Gb/s OCZ solid-state drive.

The fitness function defined above in Equation (3.1) was used, with the following weight coefficients, shown and explained in Table 5.1:

Symbol	Term	Value	Reasoning
W_{AT}	Access Time Fitness Weight	2.0	This is the most important parameter in a practical use case
W_{CT}	Conflict Time Weight	1.0	Nearly as important as satisfying access time
W_{GS}	Ground Station Usage Weight	0.01	Worth considering, but not nearly as important as effective usage of spacecraft

W_{DLV}	Duration Limit Weight	10^{-6}	A comparatively minor concern compared to other parameters
W_{LLV}	Link Length Weight	10^{-6}	A comparatively minor concern compared to other parameters
W_{HV}	Handover Violation Weight	1000	Very large due to physical impossibility, but finite to still allow numerical comparison of solution fitnesses

Table 5.1 - Fitness Function Weight Values

The program settings were as shown in Table 5.2:

Iteration Minimum Count	50
Iteration Maximum Count	2000
Fitness Per Iteration Threshold	50
Mutations Per Iteration	5
Maximum Mutation Magnitude	100
Annealing Attempts Per Iteration	250
Annealing Decay Factor	0.99948
Annealing Temperature Limit	10^{-9}
Child Count (Genetic)	8
Population Size (Genetic)	30
Steady-State Replacement	20%
Tabu Short Memory Length	12 Entries

Table 5.2 - Solver-Specific Settings

5.2 Generated Problem Instance Results

5.2.1 Fitness

Comparing the final solution fitnesses obtained by the different solver types for a fixed problem (a sample portion of the problem can be found in Appendix A), the data obtained is shown in Table 5.3 and Figure 5.1 (‘Average’ denotes the average final fitness across all cycles). Percentage improvements are relative to the initial-guess fitness (essentially that of the raw data).

Solver Type	Worst Final Fitness	Worst Percent Increase	Average Final Fitness	Average Percent Increase	Best Final Fitness	Best Percent Increase
Annealing	-1.720×10^6	0.1867	-5.520×10^5	67.97	-2.243×10^5	86.99
Tabu	6.715×10^3	100.4	8.212×10^3	100.5	1.053×10^4	100.6
Genetic	7.636×10^3	100.4	8.776×10^3	100.5	9.916×10^3	100.6
Crossover	-9.822×10^4	94.30	-6.660×10^3	99.61	1.012×10^4	100.6
Steady	-6.254×10^5	63.71	-4.755×10^5	72.41	-2.198×10^5	87.25
Struggle	1.171×10^4	100.7	1.412×10^4	100.8	1.733×10^4	101.0

Table 5.3 - Final Fitness Results by Algorithm

As can be seen, the Tabu search, the Struggle-Strategy genetic algorithm, and the base GA have comparable performance, with the Crossover GA having the capability of also achieving the same fitness, if somewhat less reliably. The Steady-State solver yields far worse fitnesses in all respects, and the Annealing heuristic, while its maximum and average fitness is similar to that of the Steady-State, can yield solutions that are far worse than any other algorithm.

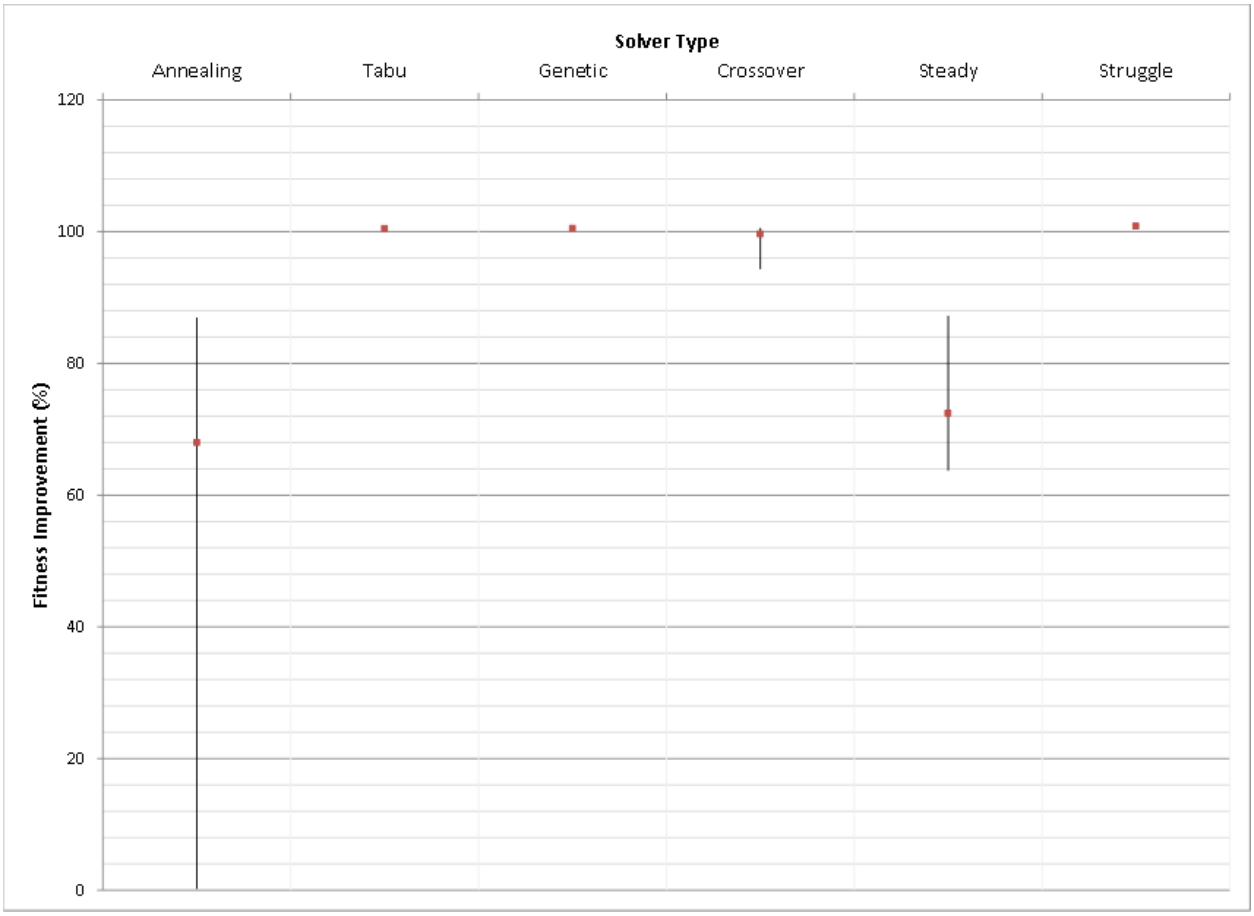

Figure 5.1 – Average Fitness Improvement and Range by Algorithm for Generated Problem; Square points indicate the average values, with the lines extending to show the minimum and maximum values

5.2.2 Execution Time

The total execution time across all cycles, as well as the average time per-iteration, is shown in Table 5.4 and Figures 5.2 and 5.3:

Solver Type	Total Time (ms)	Average Time Per Iteration (ms)
Annealing	49900	0.689
Tabu	4510000	56.3
Genetic	6490000	81.1
Crossover	6270000	78.3
Steady	1370000	17.2
Struggle	6180000	77.2

Table 5.4 - Execution Time Data by Algorithm

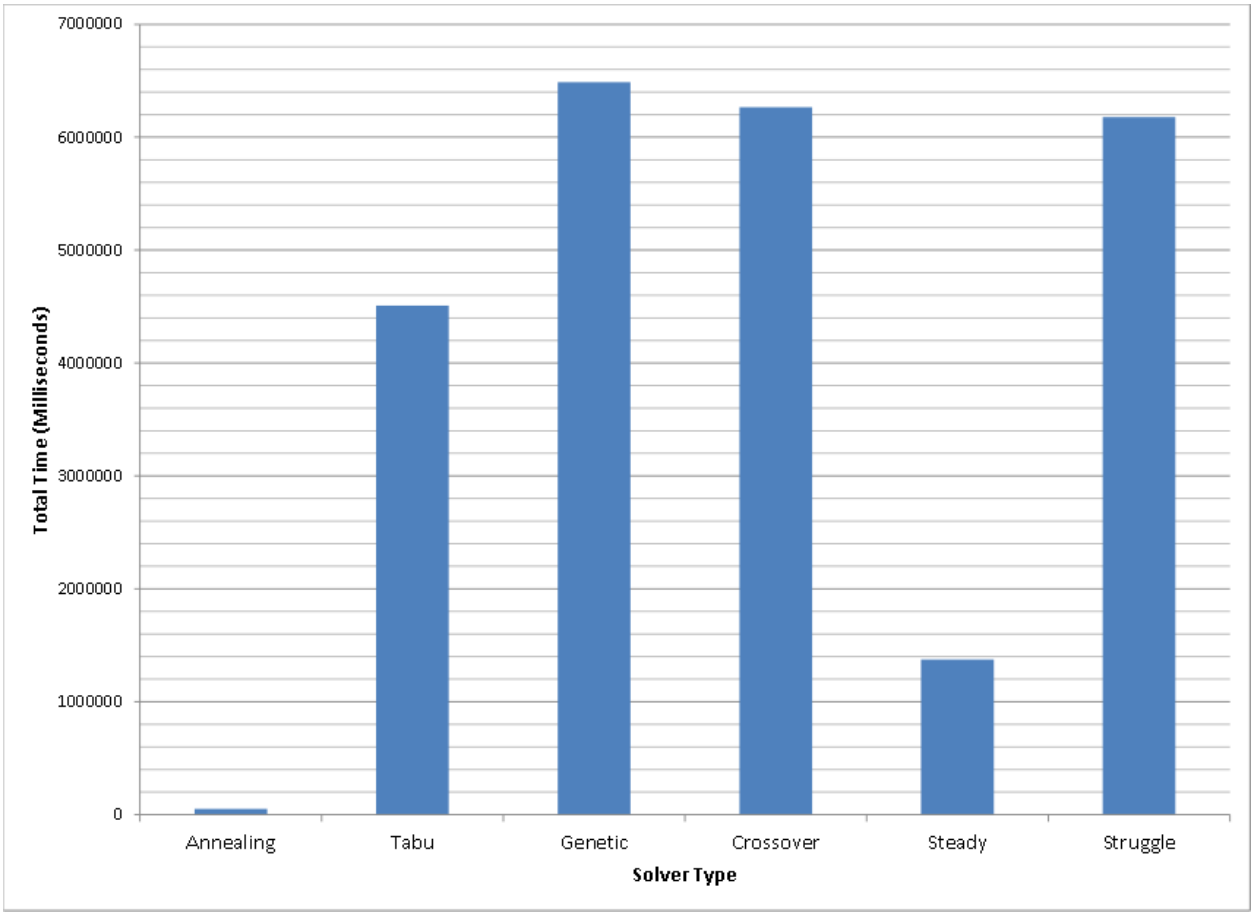

Figure 5.2 – Total Execution Time by Algorithm

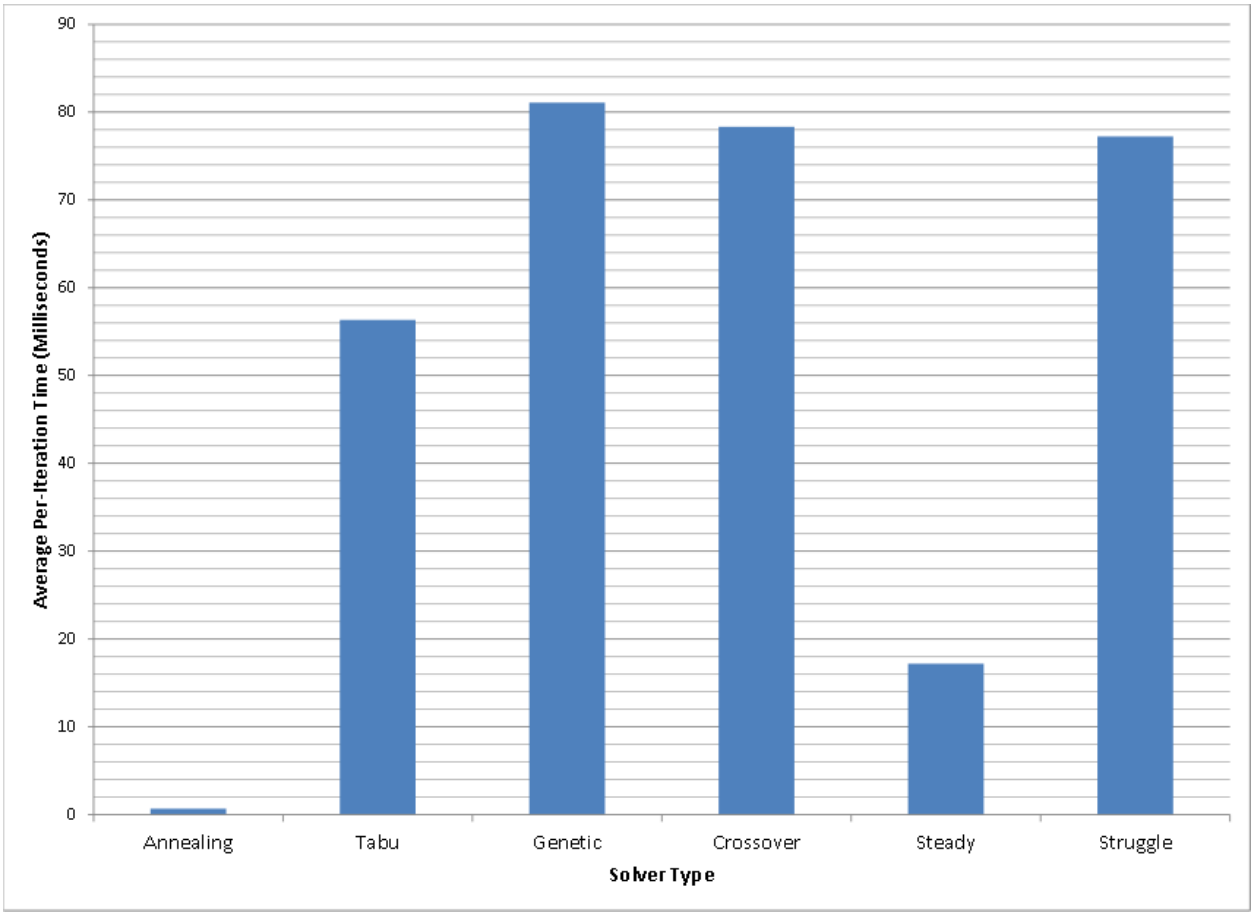

Figure 5.3 – Average Time Per Iteration by Algorithm

Aside from Steady-State, the genetic algorithms have near-identical execution times – a few milliseconds per iteration difference at most – while the Steady-State unsurprisingly runs in about one fifth the time. Tabu search is somewhat faster, while Annealing demonstrates itself to be orders of magnitude faster yet.

5.2.3 Fitness vs Time

Meta-analysis data, regarding the average fitness improvement per iteration as well as per millisecond, is shown in Table 5.5 and Figure 5.4:

Solver	Improvement Per Iteration	Improvement Per Millisecond
Annealing	647	9.40×10^{2}
Tabu	866	1.54×10^{1}
Genetic	866	1.07×10^{1}
Crossover	858	1.10×10^{1}
Steady	624	3.63×10^{1}
Struggle	869	1.12×10^{1}

Table 5.5 - Fitness vs Time Data by Algorithm

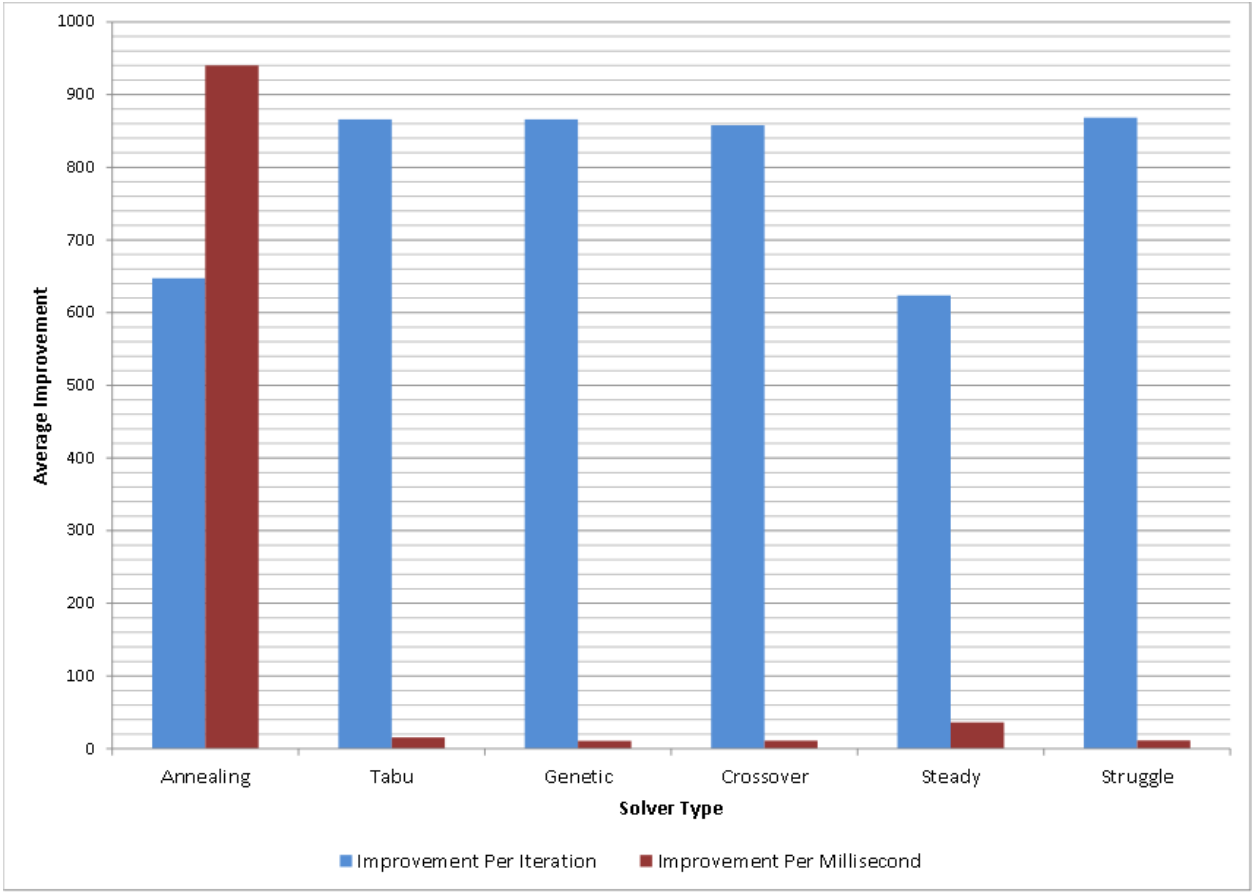

Figure 5.4 – Average Fitness Over Iteration and Millisecond by Algorithm

This data shows that even though that – compared to the others – the Simulated Annealing heuristic is not normally of comparable performance, at least in terms of output fitness, it is much more time-efficient in that it improves far more rapidly per millisecond.

Figures 5.5 and 5.6 display the fitness and improvement thereof during the solver process for the best run, i.e. the cycle with the highest final fitness. Note that each step on the graph represents 25 iterations, as the solver only logged output with that frequency.

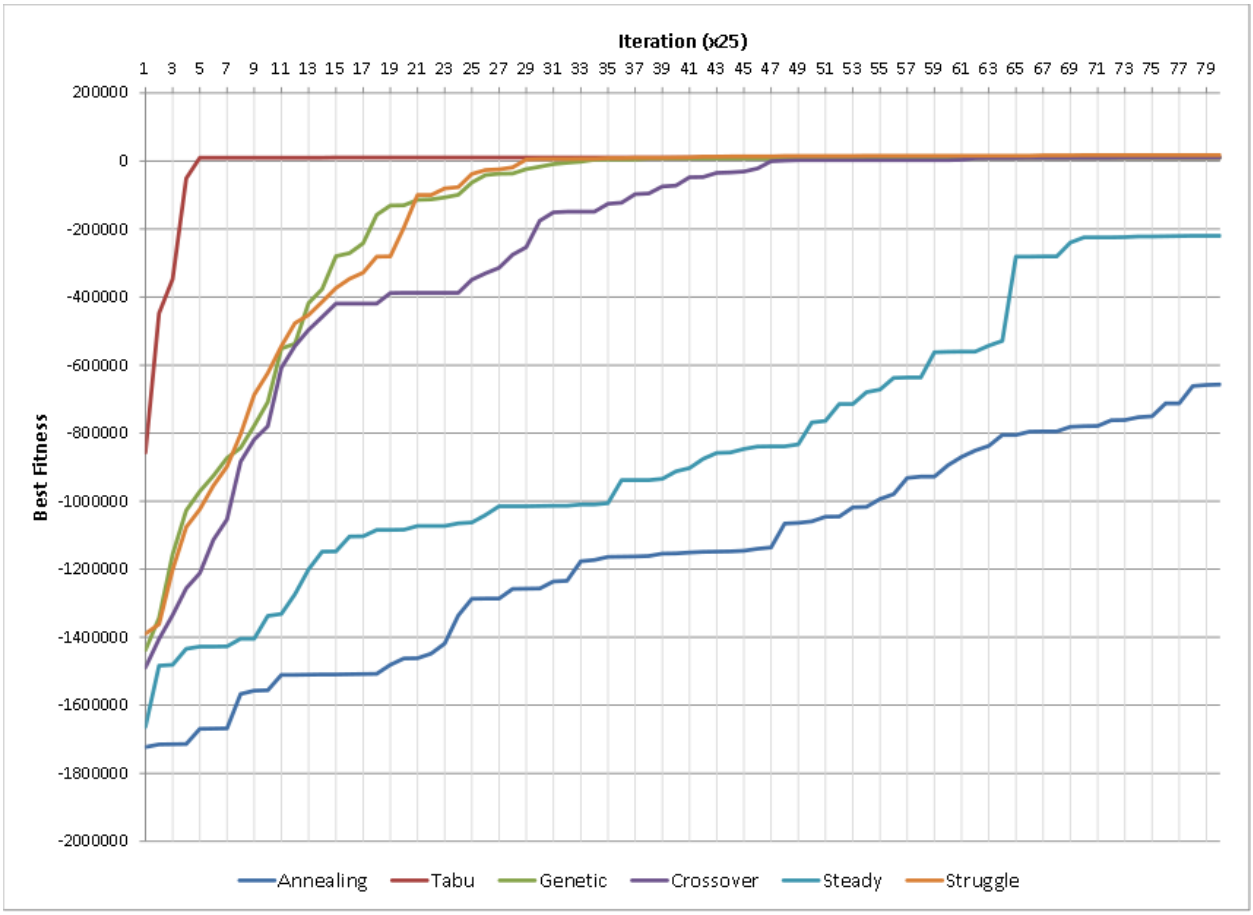

Figure 5.5 – Best Fitness History by Algorithm

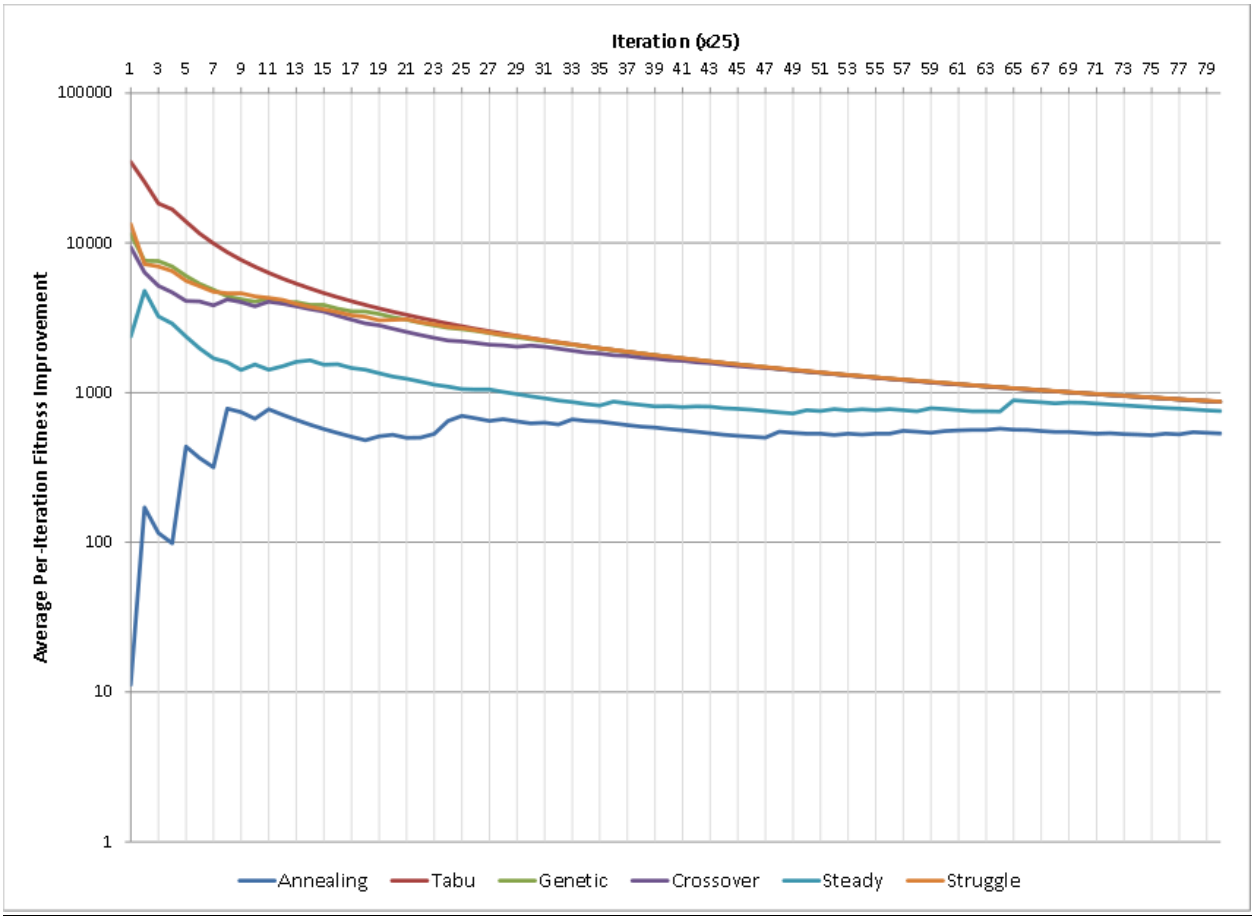

Figure 5.6 – Average Improvement Per Iteration by Algorithm

As can be seen in the fitness histories, the Tabu search converges extremely rapidly, with most of the genetic algorithms not far behind. Steady-State converges much more slowly, whereas the Annealing does not converge to a comparable fitness at all, at least in the time allotted.

5.3 Randomized Problem Instance Results

5.3.1 Fitness

The data obtained for randomized problem instances is shown in Table 5.6 and Figure 5.7. As before, percentage improvements are relative to the initial guess.

Solver Type	Worst Fitness Percent Improvement	Average Fitness Percent Improvement	Best Fitness Percent Improvement
Annealing	15.02	88.07	102.1
Tabu	100.0	101.1	117.7
Genetic	99.93	100.8	105.4
Crossover	85.89	99.51	116.3
Steady	0.8698	85.75	103.0
Struggle	98.92	101.7	133.7

Table 5.6 - Final Fitness Results by Algorithm

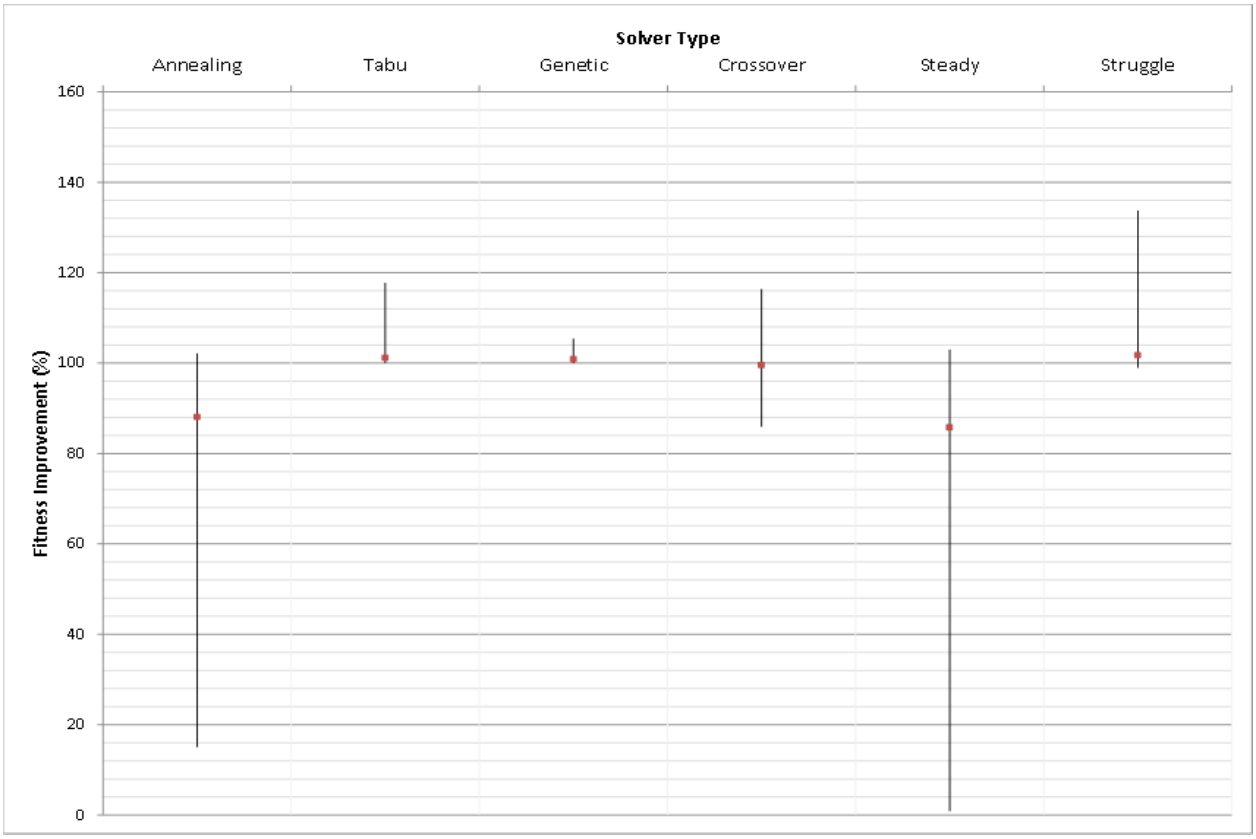

Figure 5.7 – Average Fitness Improvement and Range by Algorithm for Random Problems; Square points indicate the average values, with the lines extending to show the minimum and maximum values

Very similar to the results obtained using a non-random problem, the Tabu Search and Struggle-Strategy yield the highest potential and average fitnesses. Crossover offers potentially larger fitness improvement than the base GA, but not reliably. As before, Steady-State and Annealing fail to achieve similar improvements to the other algorithms.

5.3.2 Execution Time

The total execution time across all cycles, as well as the average time per-iteration, is shown in Table 5.7 and Figures 5.8 and 5.9:

Solver Type	Total Time (ms)	Average Time Per Iteration (ms)
Annealing	41500	0.578
Tabu	6580000	84.2
Genetic	5540000	72.7
Crossover	5910000	76.5
Steady	1200000	15.6
Struggle	5620000	71.4

Table 5.7 - Execution Time Data by Algorithm

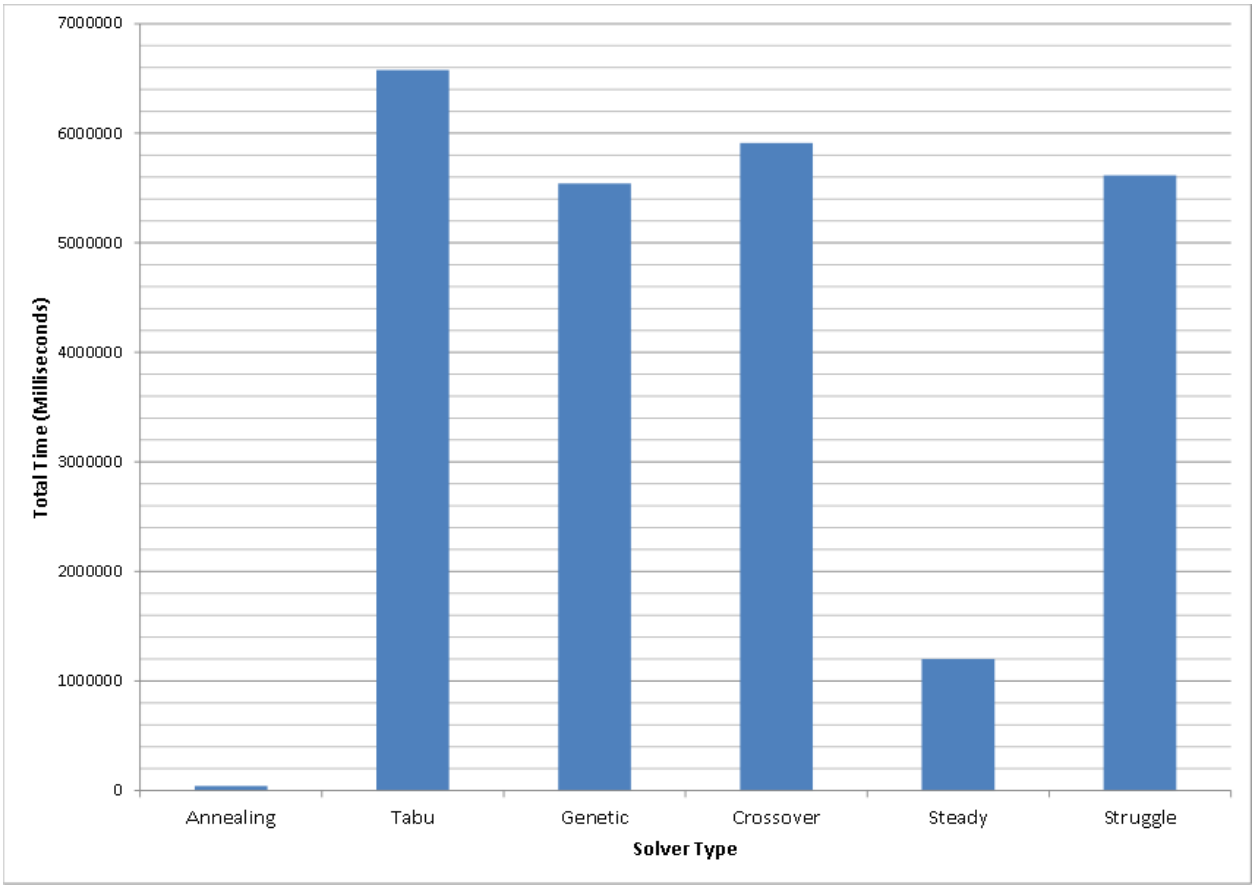

Figure 5.8 – Total Execution Time by Algorithm

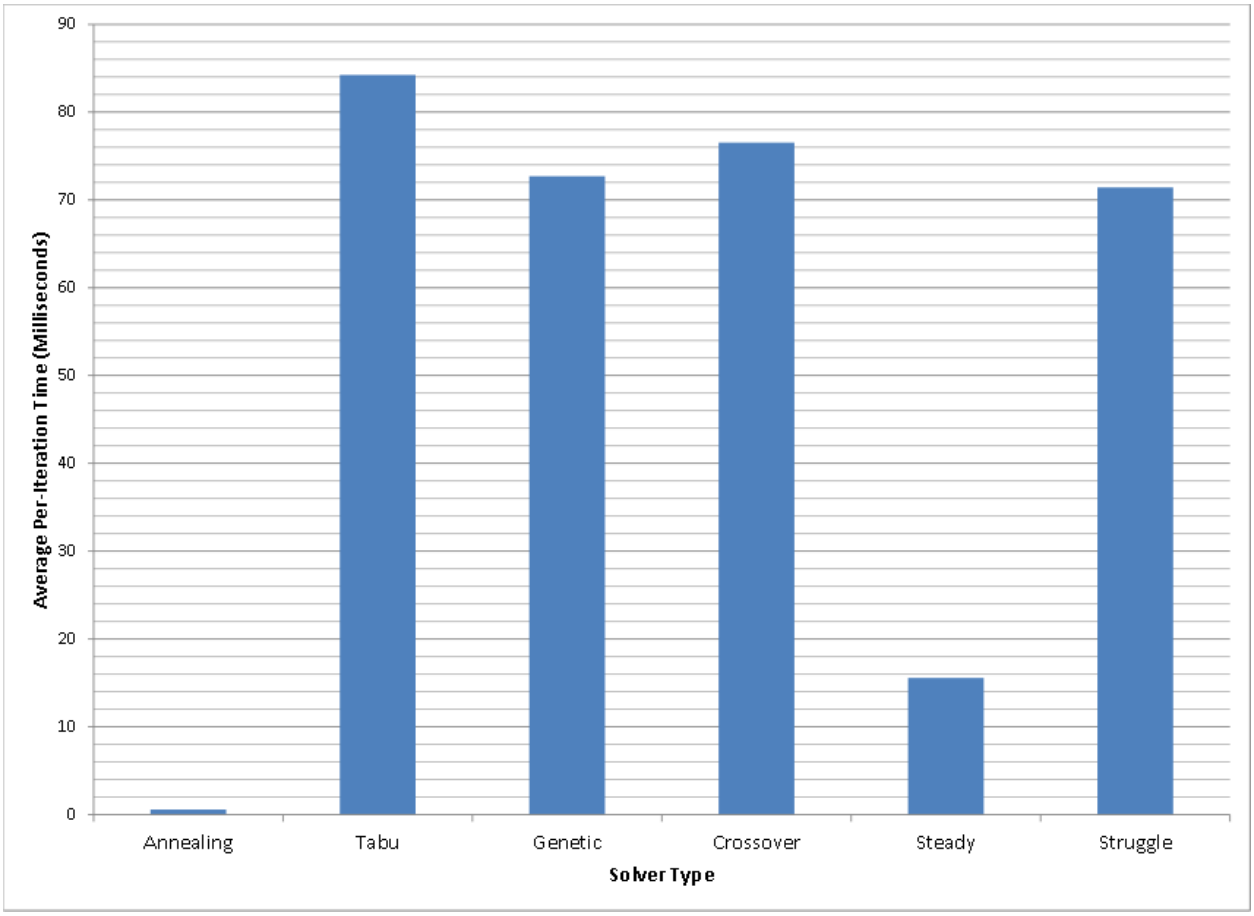

Figure 5.9 – Average Time Per Iteration by Algorithm

These results are nearly identical to those of a fixed problem instance, with the biggest difference being that the Tabu search is now the slowest to execute. However, this difference is not large, and the main trends remain present. In particular, the Tabu Search and Struggle GA remain the best improvers per iteration.

5.3.3 Fitness vs Time

Meta-analysis data, regarding the average fitness improvement per iteration as well as per millisecond, is shown in Table 5.8 and Figure 5.10:

Solver	Improvement Per Iteration	Improvement Per Millisecond
Annealing	380	6.57×10^{2}
Tabu	416	4.93×10^{0}
Genetic	337	4.64×10^{0}
Crossover	426	5.57×10^{0}
Steady	293	1.89×10^{1}
Struggle	450	6.30×10^{0}

Table 5.8 - Fitness vs Time Data by Algorithm

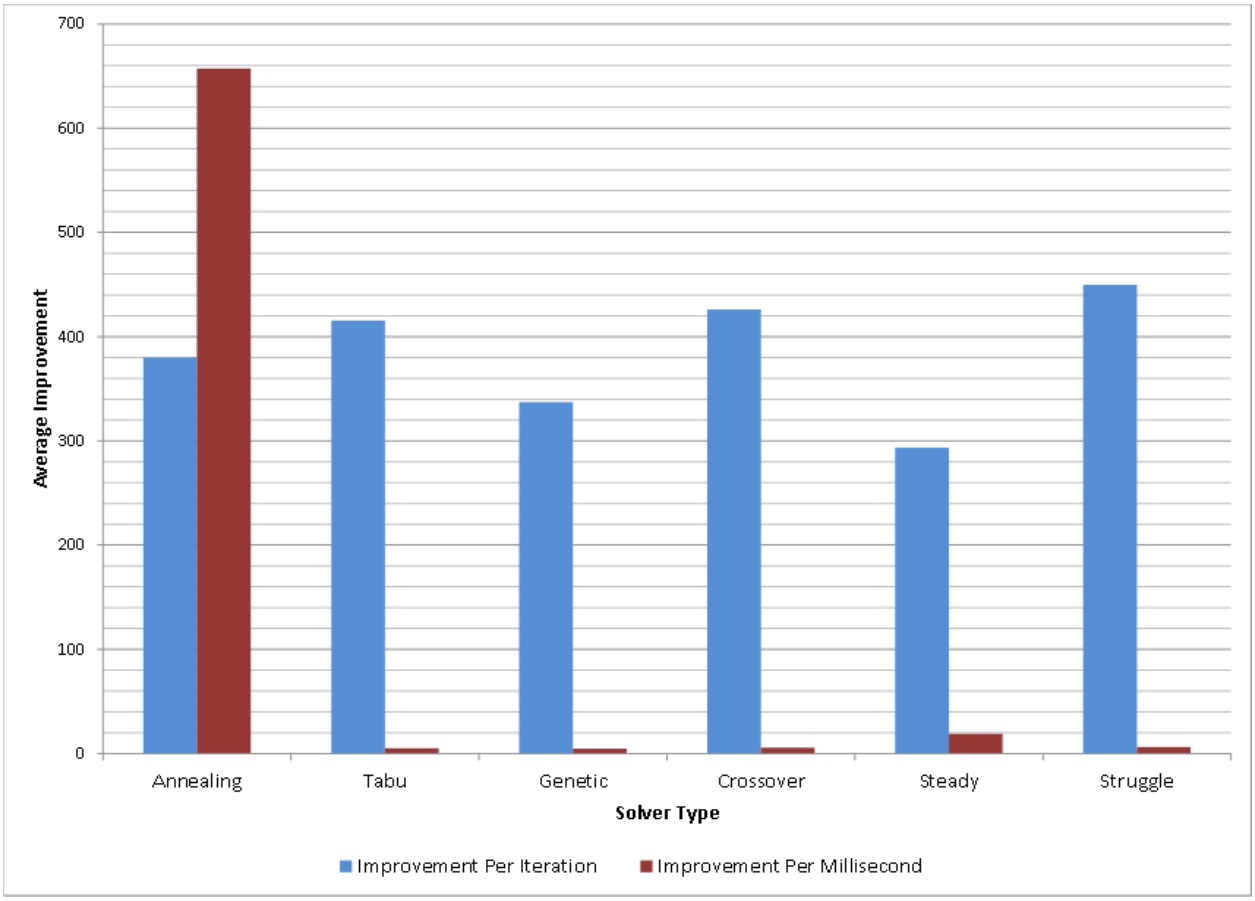

Figure 5.10 – Average Fitness Over Iteration and Millisecond by Algorithm

Figures 5.11 and 5.12 display the fitness and improvement thereof during the solver process for the best run, i.e. the cycle with the highest final fitness.

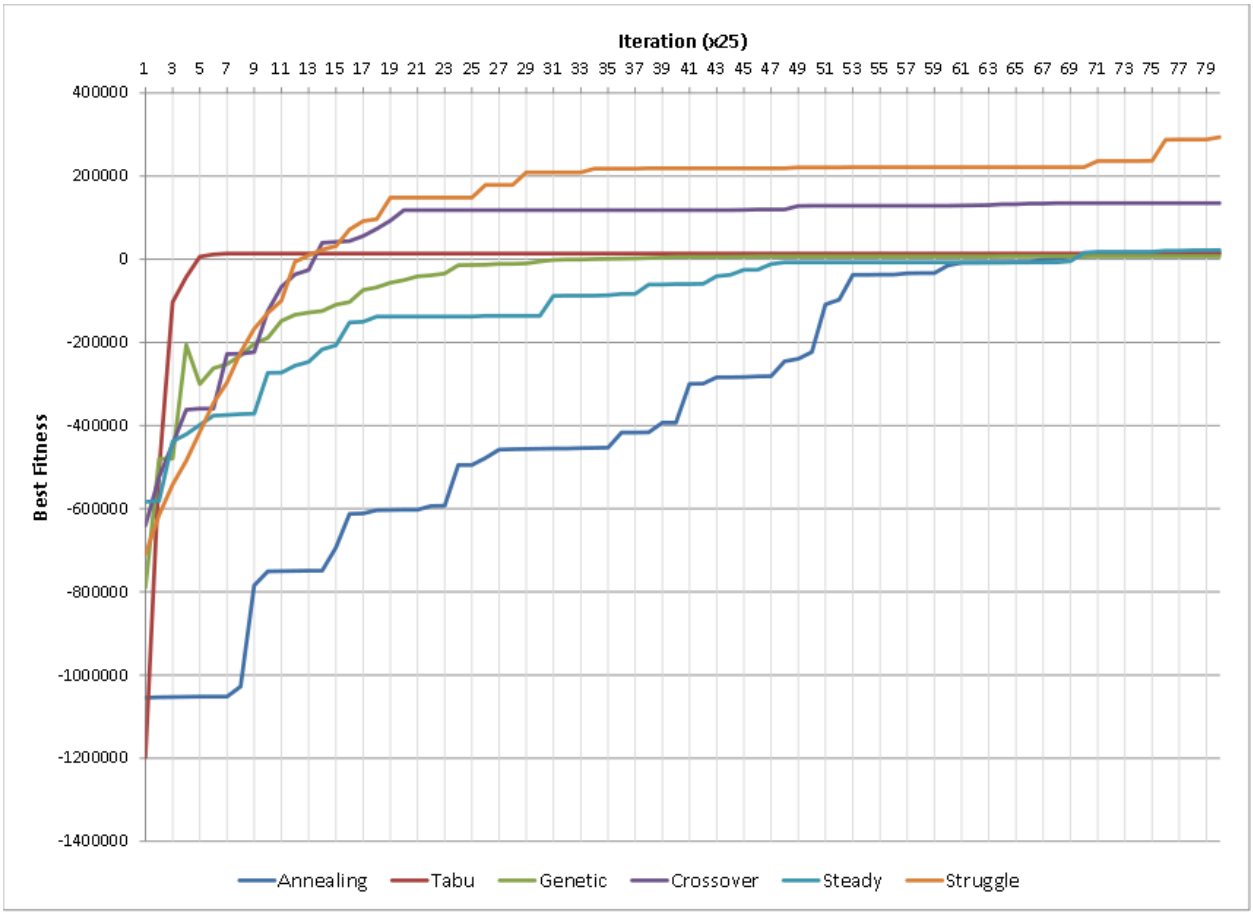

Figure 5.11 – Best Fitness History by Algorithm

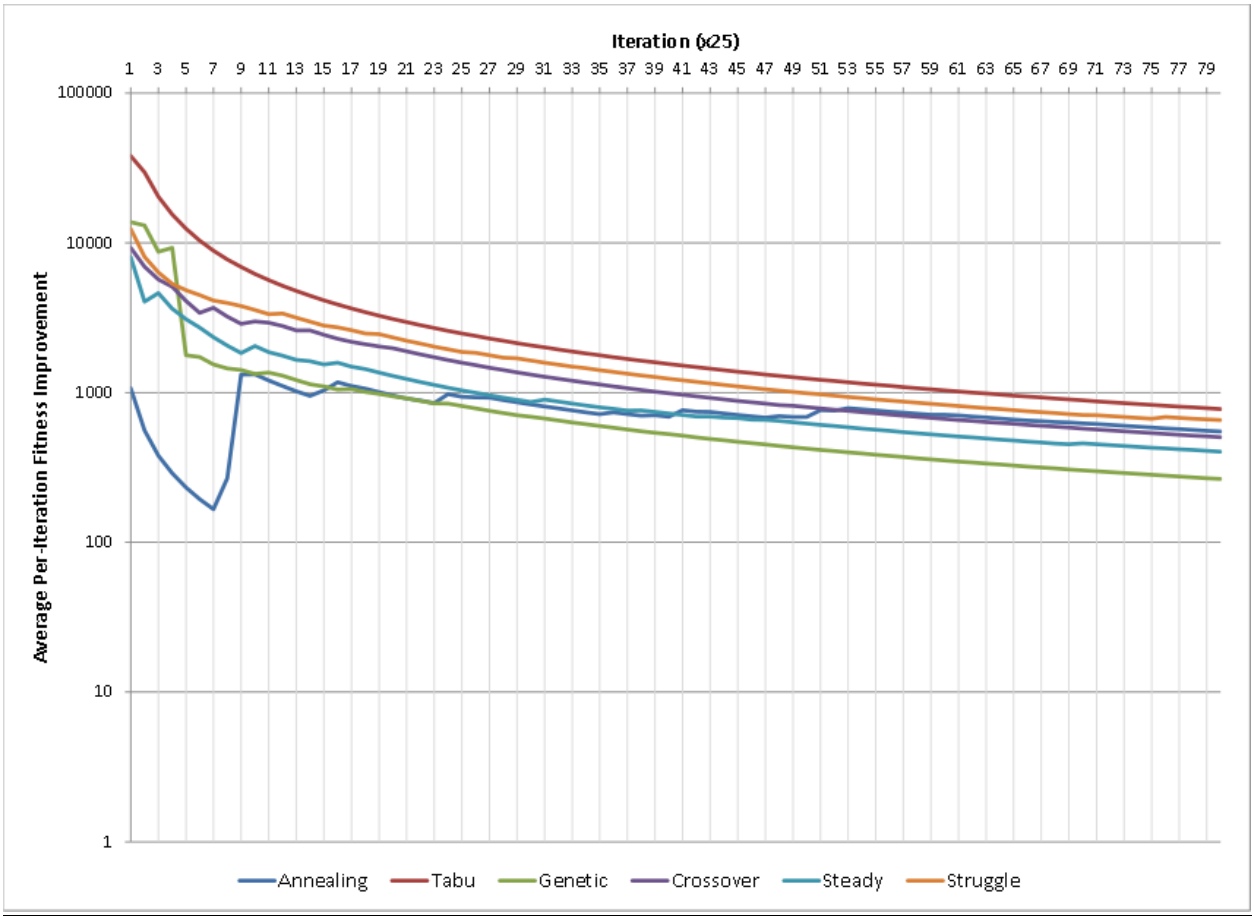

Figure 5.12 – Average Improvement Per Iteration by Algorithm

The fitness histories show a similar story to with a fixed problem instance. One important result here is the Struggle and Crossover GAs demonstrating a superior capacity for exploration of the search space, as both found solutions better than the plateau that trapped the other solvers.

5.4 Long-Run Annealing Results on Generated Problem

Given a large number of iterations, a Simulated Annealing algorithm can sometimes obtain solutions of competitive quality to the other solvers, and still much more rapidly in terms of elapsed time, as shown in Table 5.9 and Figure 5.13:

Solver Type	Worst Fitness	Average Fitness	Best Fitness	Average Total Time (ms)
Long-Run Annealing	-1.722×10^6	-3.808×10^5	9.804×10^3	932000
Tabu	6.715×10^3	8.212×10^3	1.053×10^4	4510000
Genetic	7.636×10^3	8.776×10^3	9.916×10^3	6490000
Crossover	-9.822×10^4	-6.660×10^3	1.012×10^4	6270000
Steady	-6.254×10^5	-4.755×10^5	-2.198×10^4	1370000
Struggle	1.171×10^4	1.412×10^4	1.733×10^4	6180000

Table 5.9 – Long-Run Annealing Fitnesses Compared To Other Algorithms

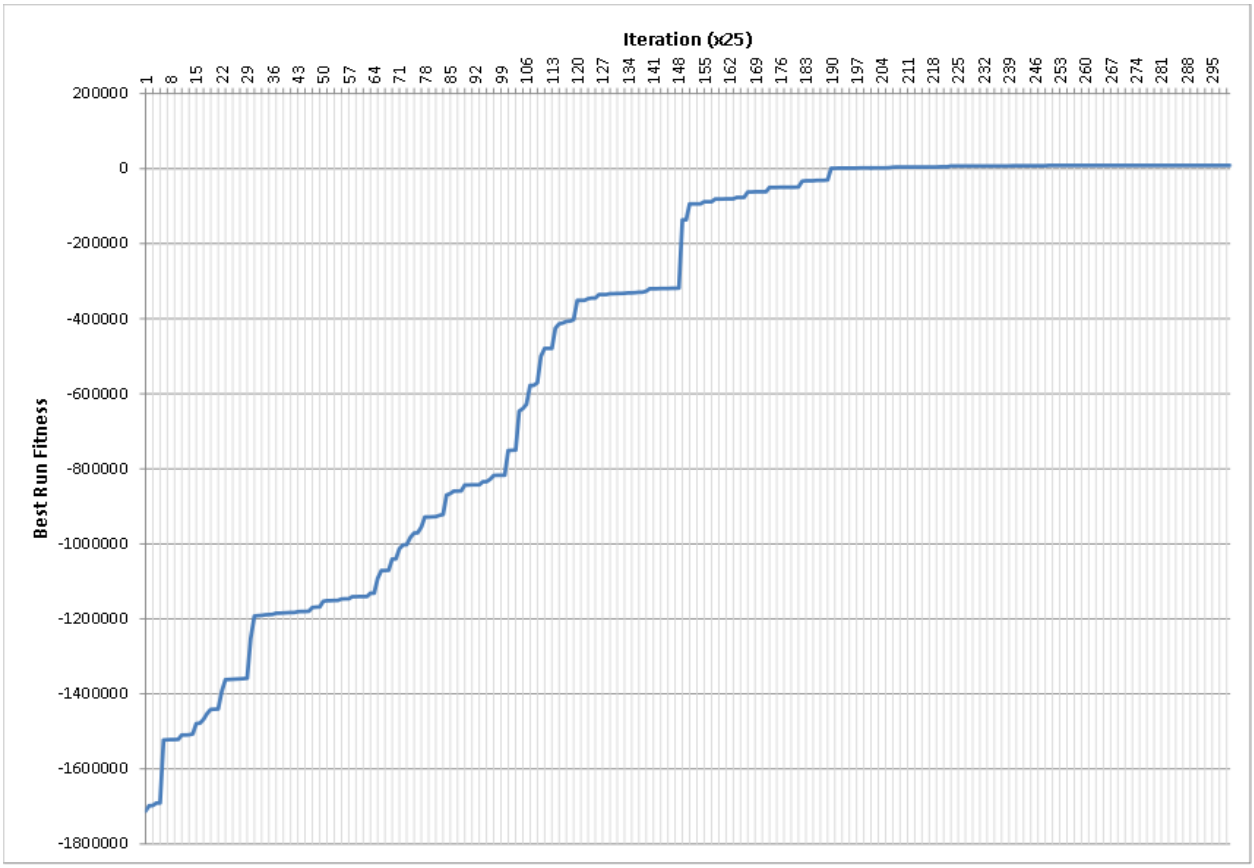

Figure 5.13 – Long-Run Annealing Fitness History

5.5 Hybrid Solution Results on Random Problems

In an attempt to achieve even better solutions than the individual algorithms, a hybrid algorithm was designed so as to be able to take advantage of the rapid improvement of the Tabu search as well as the exploratory power of the genetic algorithms. Using the data from Section 5.3 as a guide, the hybrid algorithm was chosen to be a Tabu search to iteration 250, then a Struggle-Strategy GA to iteration 1000, followed by 1000 iterations of annealing to scan for potential improvements. The goal was to improve upon the initial guess first, then expand the search to find nearby maximums of greater quality than the local. This hybrid was tested against the other solvers for ten randomly-generated problem instances.

This ultimately did not prove to be the case; as can be seen in Figure 5.14, this hybrid, at least in terms of final fitness, does not offer noticeably better performance than the other solvers.

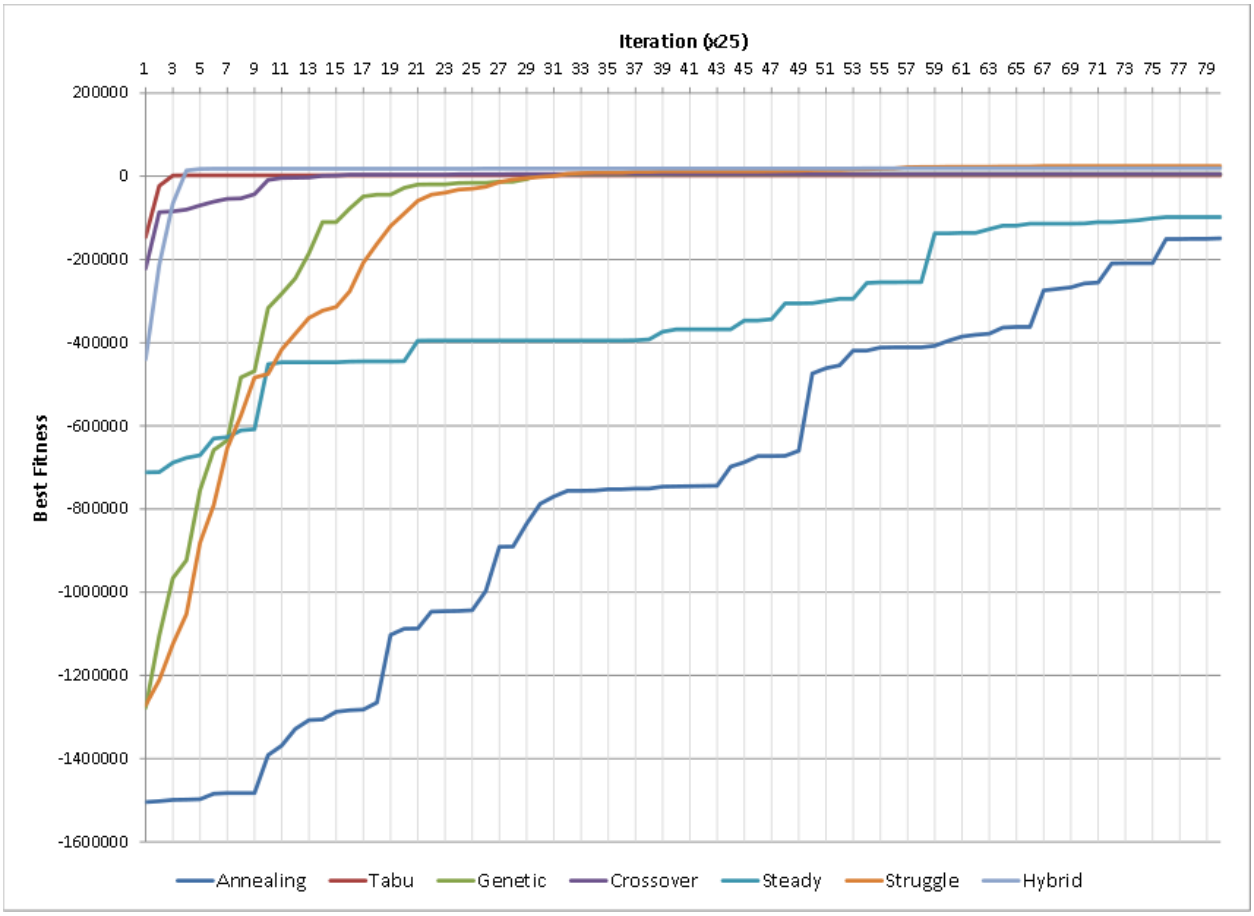

Figure 5.14 – Hybrid Solver Performance Comparison, Best Fitness History

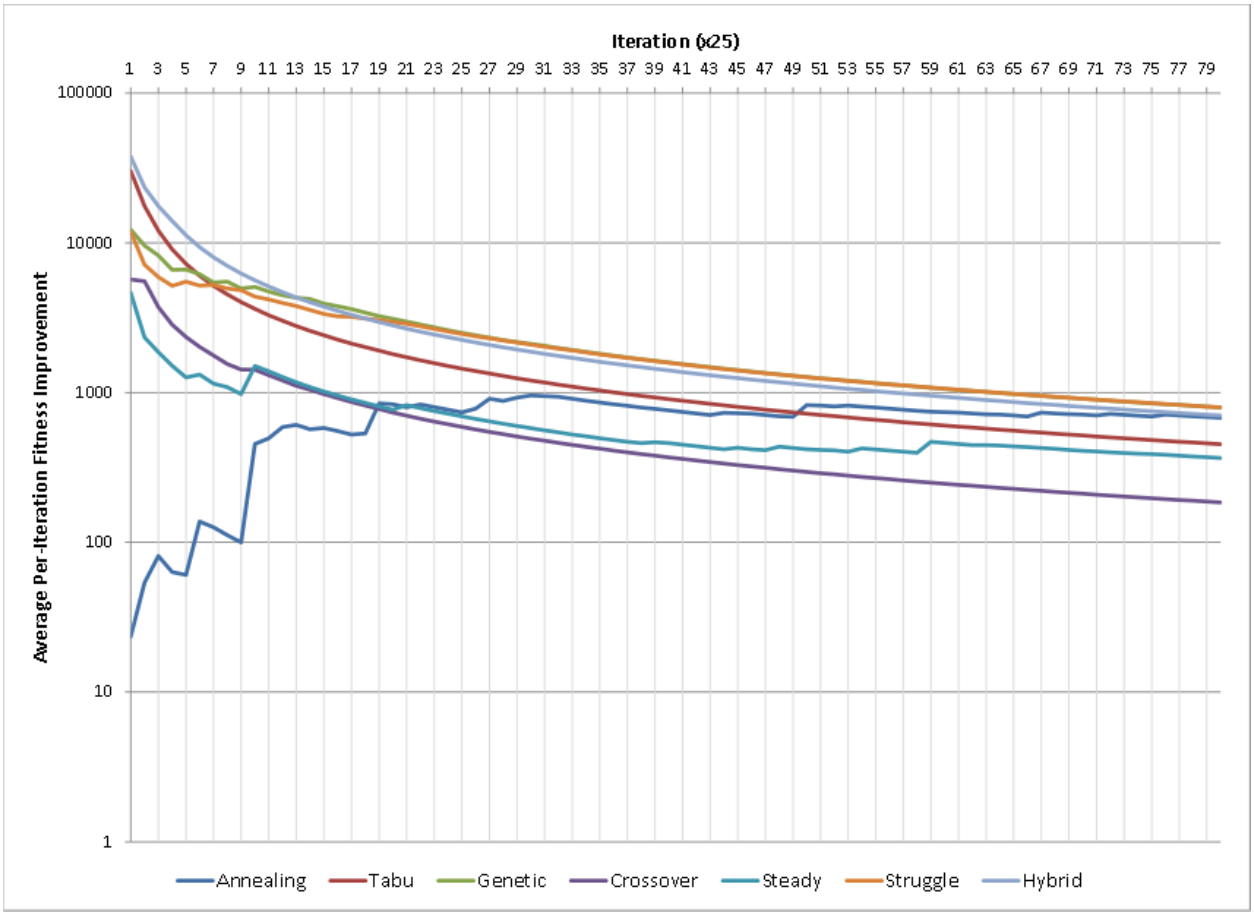

Figure 5.15 – Hybrid Solver Performance Comparison, Average Improvement Per Iteration

5.6 Effect of Annealing Attempt Count

Though it is not normally present in a standard implementation of Simulated Annealing, the number of attempts per iteration, if increased, does improve the performance of the solver, both in terms of output quality and convergence speed. However, this effect is not particularly large, and ceases to be significant once the number of attempts reaches five to ten attempts per iteration. This can be seen in Table 5.10 and Figures 5.16 and 5.17:

Annealing Attempts	Worst Fitness	Average Fitness	Best Fitness	Iterations to Convergence (approx.)
1	-1.723×10^5	-7.582×10^5	-9.302×10^4	>4000
2	-1.722×10^5	-4.056×10^5	6.020×10^3	3900
5	-1.722×10^5	-3.507×10^5	7.790×10^3	3850
10	-1.722×10^5	-2.997×10^5	9.801×10^3	3425
50	-1.722×10^5	-2.300×10^5	9.149×10^3	3900
100	-1.722×10^5	-3.010×10^5	8.374×10^3	3850
250	-1.723×10^5	-2.826×10^5	8.957×10^3	3125
1000	-1.723×10^5	-2.834×10^5	9.432×10^3	3475

Table 5.10 – Effect of Annealing Attempt Count Per Iteration

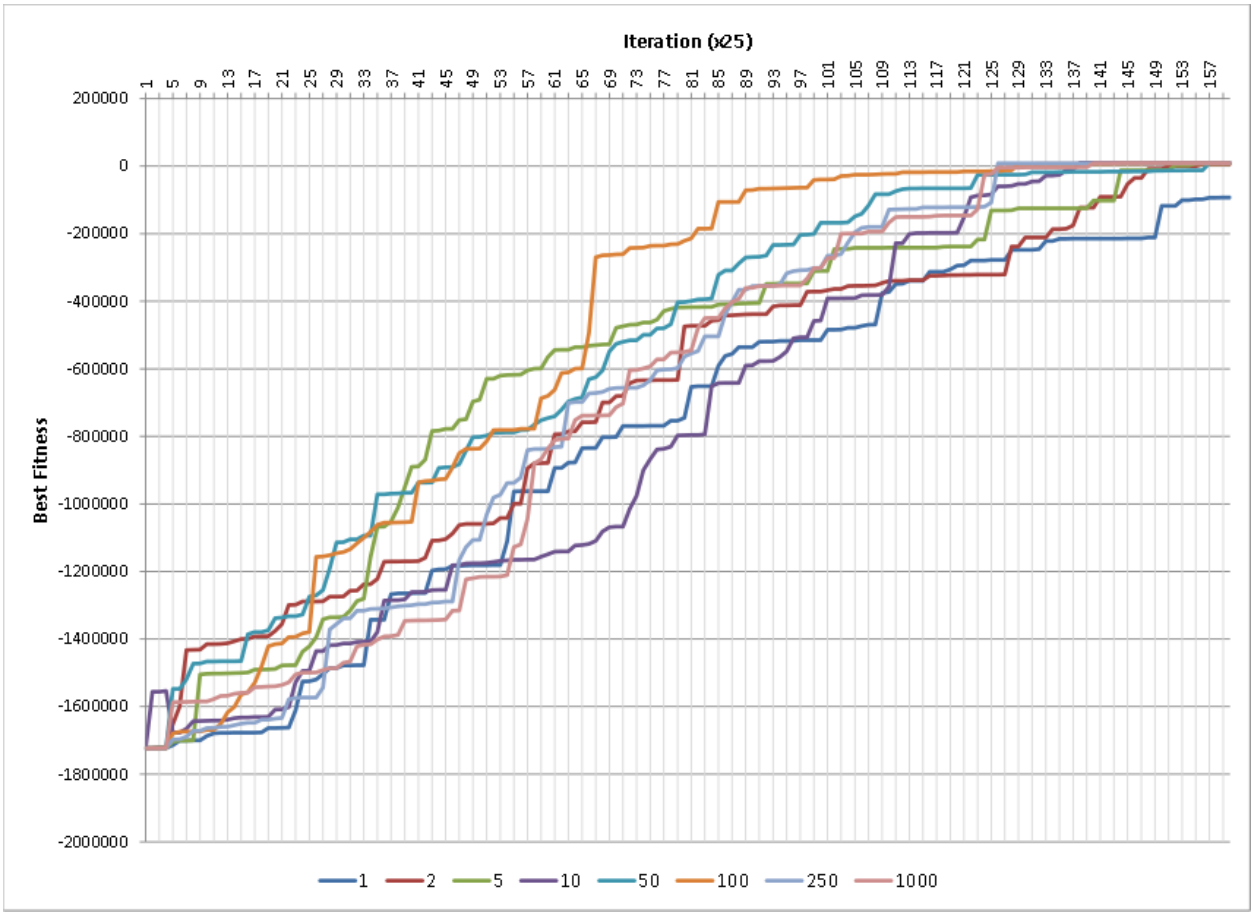

Figure 5.16 – Fitness History by Annealing Attempt Count

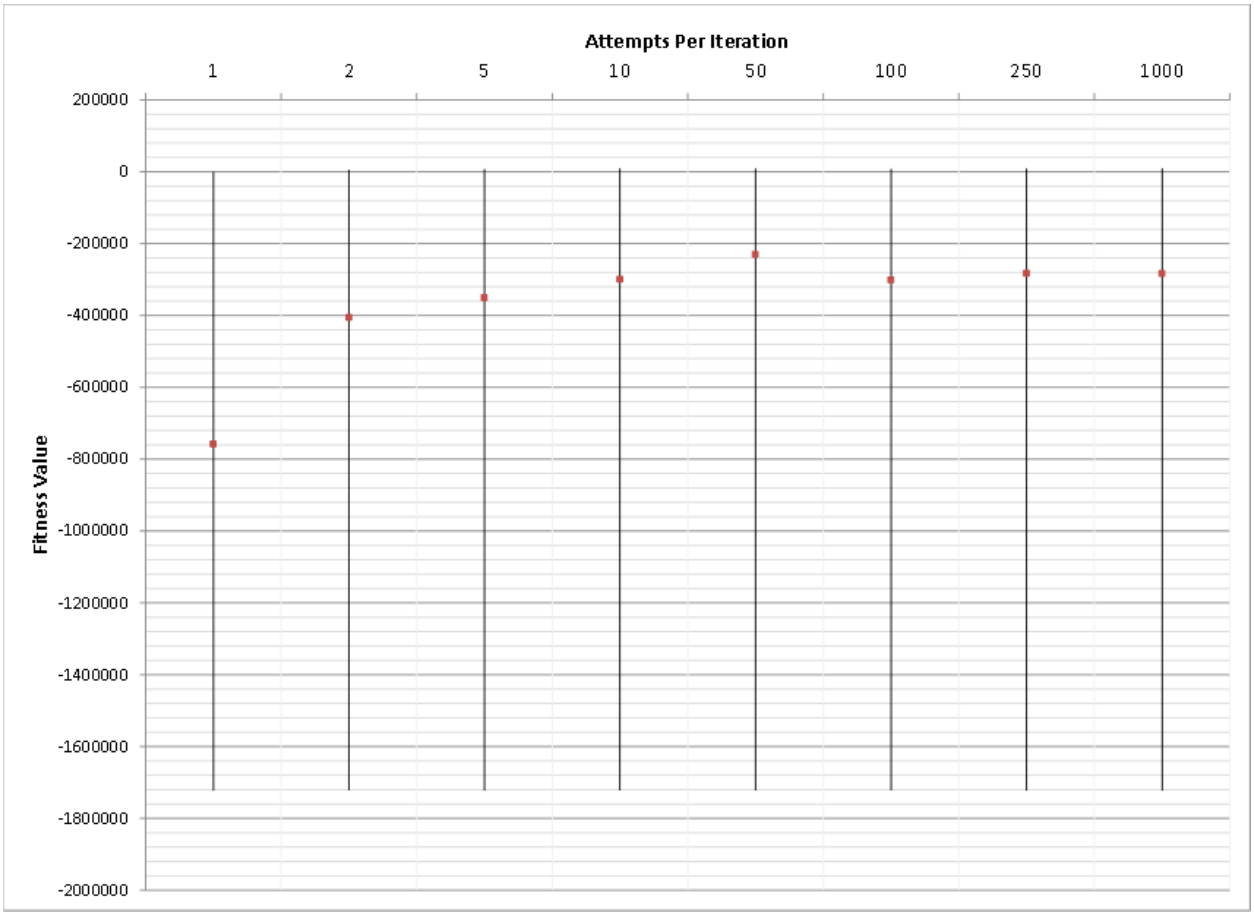

Figure 5.17 – Fitness and Variation by Annealing Attempt Count; Square points indicate the average values, with the lines extending to show the minimum and maximum values

This improvement also comes at the cost of time, as seen in Figure 5.18:

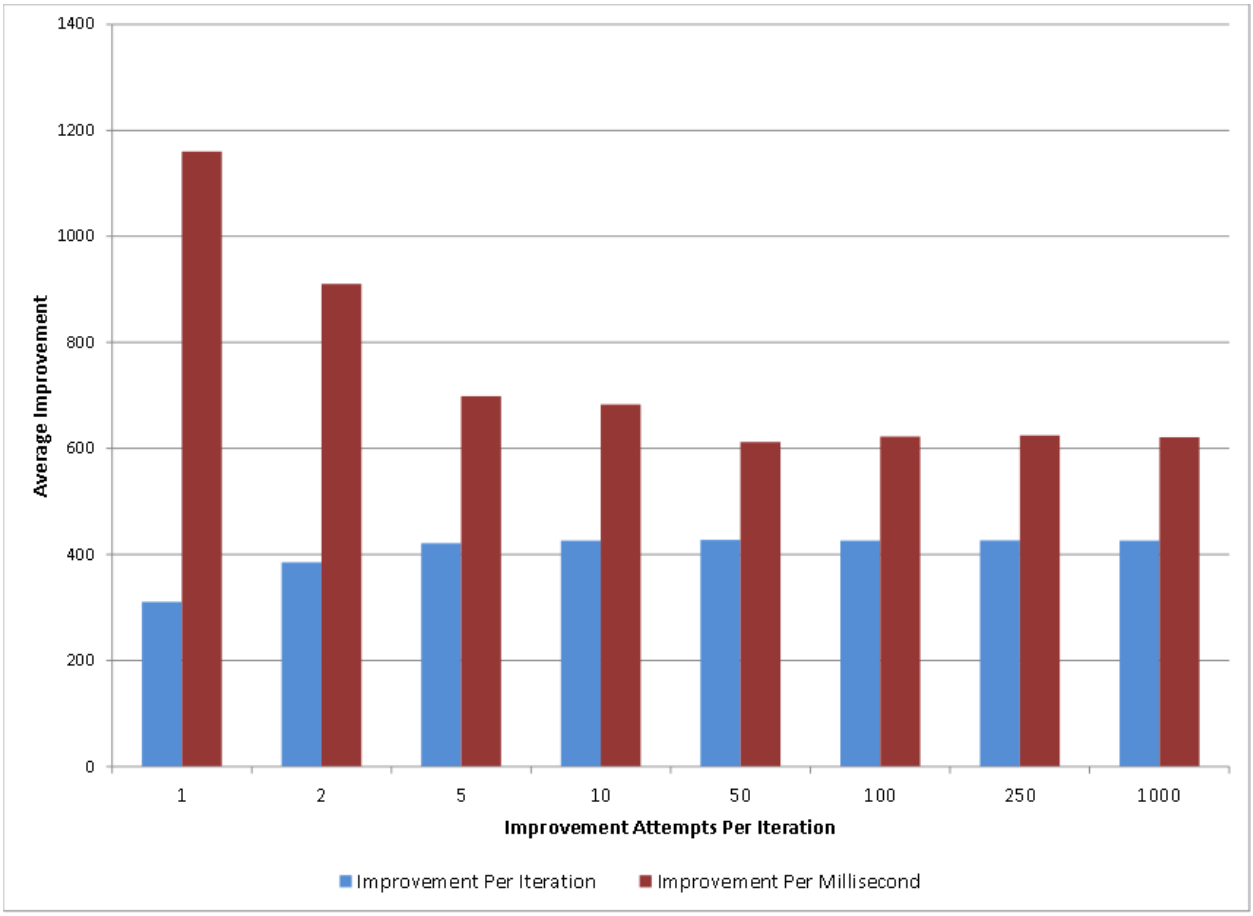

Figure 5.18 – Improvement Over Iteration and Millisecond by Annealing Attempt Count

5.7 Feasibility Improvements

All of the solvers improved the feasibility of randomized problem instances, reducing both conflict time and handover violation count. A sample of this can be seen in Table 5.11 and Figures 5.19 and 5.20, which average these results across six randomly-generated problem instances:

Solver	Average Conflict Time	Average Spacecraft Handover Violations	Average Ground Station Handover Violations
<Raw Problem>	1.04×10^6	0.250	6.5
Annealing	7.06×10^5	0.167	2.0

Tabu	4.70×10^{-1}	0.000	0.0
Genetic	2.60×10^{-1}	0.000	0.0
Crossover	6.29×10^{3}	0.000	0.0
Steady	3.85×10^{5}	0.000	0.0
Struggle	1.80×10^{2}	0.000	0.0
Hybrid	1.73×10^{-1}	0.000	0.0

Table 5.11 – Solution Feasibility Improvement by Solver

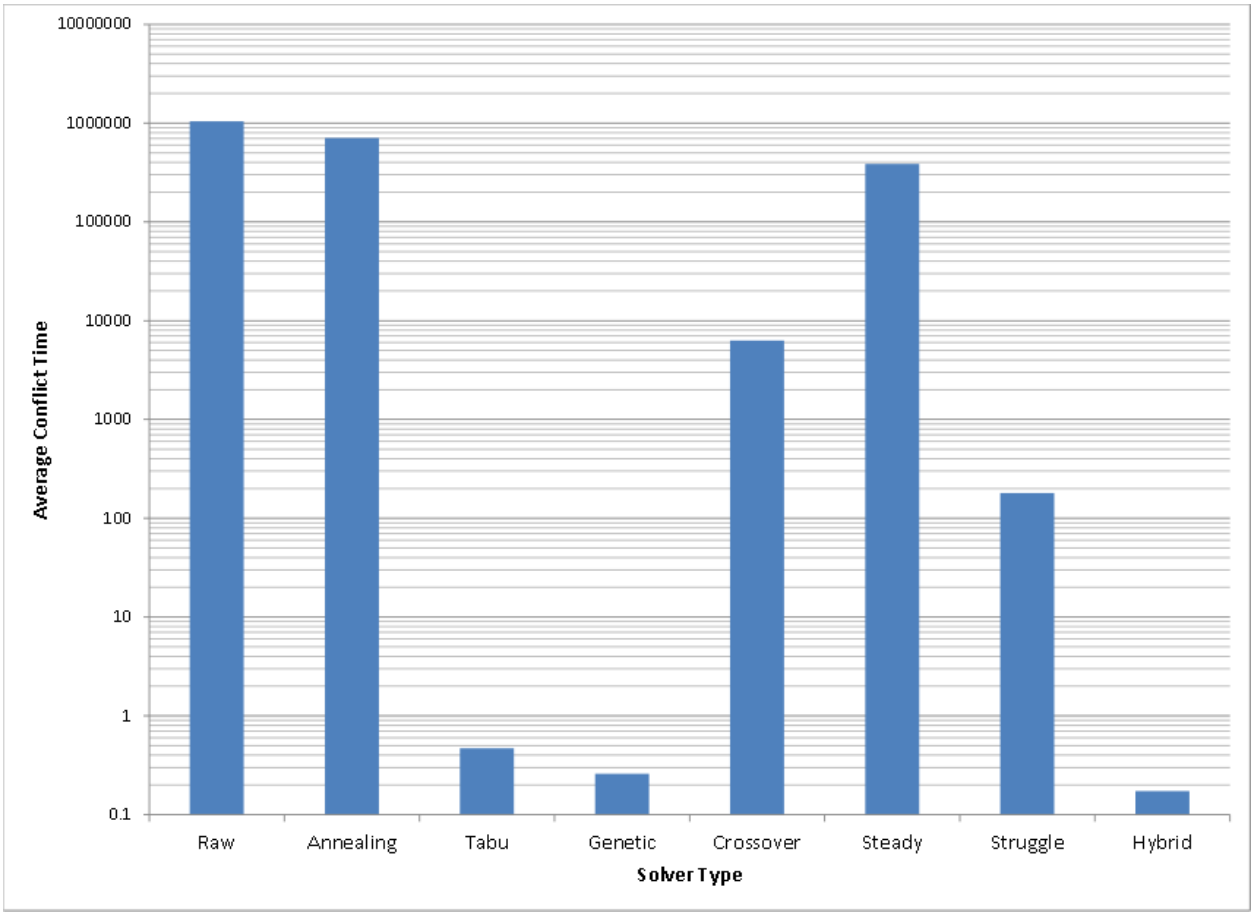

Figure 5.19 – Average Total Conflict Time in Final Solution by Solver

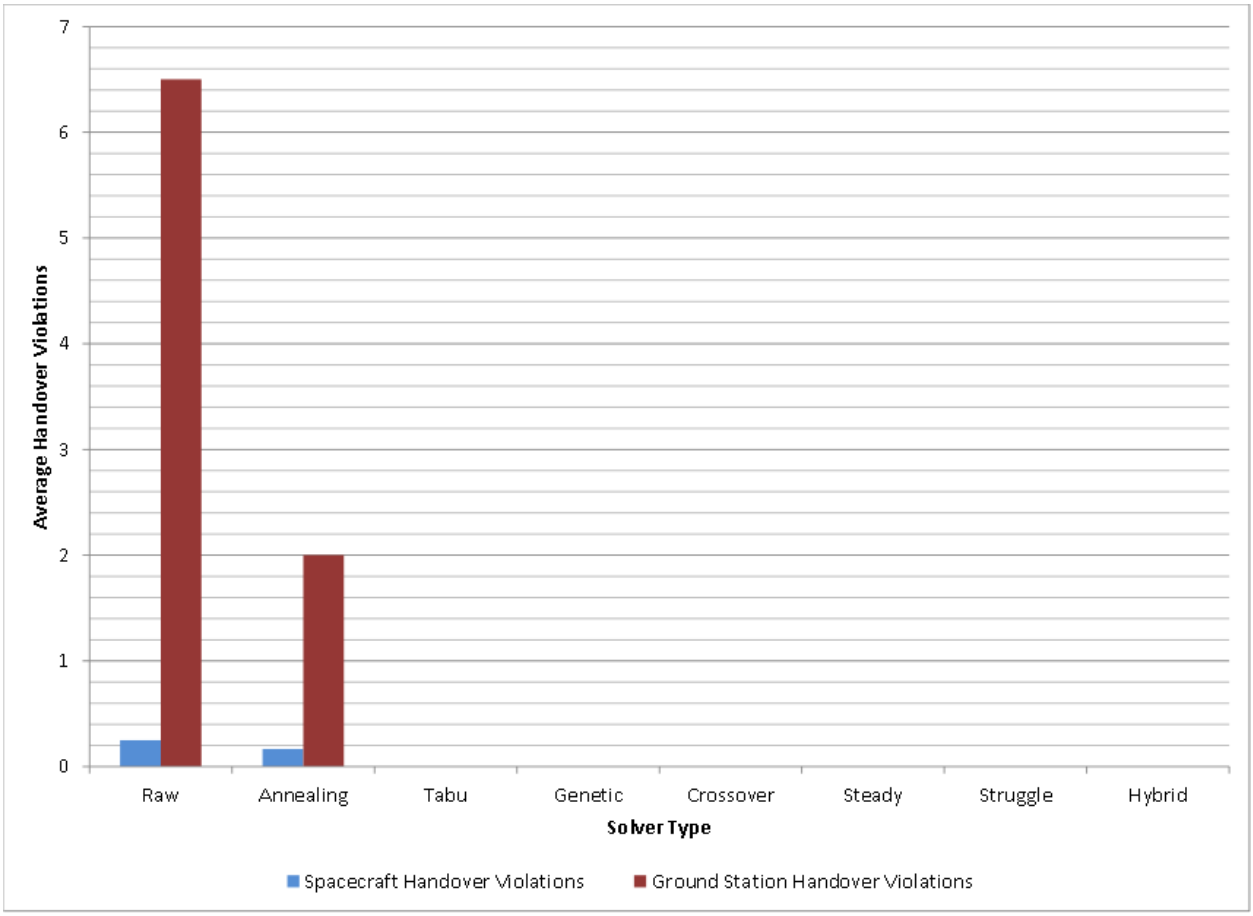

Figure 5.20 – Average Handover Violation Count in Final Solution By Solver

5.8 Analysis

5.8.1 – Exploration vs Exploitation

As expected, different algorithms and their results demonstrate the effects of a different balance between exploration and exploitation. In both the generated and the random problems, the Tabu search demonstrated easily the greatest capacity for exploitation, converging extremely rapidly compared to the other solvers. This is most clearly demonstrated in Figures 5.5, 5.11, and 5.14. However – as seen in the same figures – it demonstrated fairly poor ability to then explore the larger search space, and given enough time was often surpassed by the genetic algorithms, whose large population size and thus large amount of variation give them much better ability to explore. This differs somewhat from the literature results [4], but it is almost certainly explained by the

simplified implementation of the Tabu search in this work, where it is fairly similar to a Hill-Climbing algorithm, a heuristic known for its lack of exploration ability [12].

5.8.2 – Genetic Algorithm Comparison

As can be seen in Figure 5.7, and is consistent with the literature [15], a simple base genetic algorithm offers mediocre performance in its obtained fitnesses, significantly outperformed by the crossover and struggle-strategy versions. Similarly, also visible in both Figure 5.7 and Figure 5.1, and again consistently with the literature, the Steady-State variation offered much poorer solutions, most likely because the algorithm is ill-suited for cases like the ones in this work, with large population (N=30) and large problem instances [12] [5].

Also as expected, the Struggle-Strategy performed well more reliably than the Crossover algorithm, where the nature of the crossover implementation has a chance of worsening a solution as well as improving it. The large variation in final fitness from the Crossover solver reinforces this, especially given that the average fitness is not substantially different from that of the base genetic algorithm. This relationship is most obvious in Figure 5.7.

The most noticeable difference seen between these results and those generally found in the literature is the execution time (displayed in Figures 5.2, 5.3, 5.8, and 5.9), especially for the Struggle-Strategy heuristic. Where most literature, like in [9] and [15], describes Struggle-Strategy as very slow – especially for the non-Hash version used in this work – compared to a base or even crossover GA, but the results obtained here show times, both total and per-iteration, that are mostly the same – a few milliseconds per iteration difference at most – between all genetic solvers. One possible explanation of this is that the main reason a non-Hash Struggle GA is slow is due to iteration over the population, and it may well be the case that the size here is not large enough to create a significant effect. Another possible explanation may be that the similarity evaluation used in the literature is more detailed when compared to this work's computationally-simple direct-link-comparison implementation, which would translate to a larger effect.

5.8.3 – Speed vs Solution Quality

Unsurprisingly, the algorithms which perform simpler or fewer operations per iteration – namely the Simulated Annealing and Steady-State algorithms – yield comparatively poor final solutions compared to the other solvers, both for the given problem instance and for randomized problems, as can be seen in Figures 5.1 and 5.7. However, the tradeoff is not zero-sum; that is, the final solutions are often worse given the same number of iterations, but the time required to perform these iterations is markedly less than for the other heuristics.

One significant result from the data obtained is that the improvement-per-millisecond seems almost wholly determined by the absolute speed of the solver, not its per-iteration quality. This then implies that given enough iterations to converge, an algorithm that appears poor may actually prove in a way superior due to its much higher time efficiency.

This is especially true for the Simulated Annealing algorithm, which while it converges far more slowly *per-iteration* than any other solver, does so enormously faster *per-millisecond* (Figure 5.10). As a result, though the Simulated Annealing algorithm may at first appear ill-suited to this sort of application, it offers significant promise in that it has good capability to offer solutions of reasonable quality in a fraction of the time required for any other algorithm. Given enough iterations, it can achieve good-quality solutions, as can be seen in Figure 5.13.

Though it is true that Simulated Annealing does not reliably achieve competitive fitness, this is of little consequence; the algorithm completes so rapidly compared to the others – on the order of seconds as opposed to 15 minutes or more – that it is trivial to simply run a large number of cycles and select the best solutions.

5.8.4 – Hybrid Analysis

The hybrid solvers designed – of which the best was shown earlier – did not prove superior to the base algorithms. In particular, as was shown in Figure 5.14, the hoped-for fast improvement followed by exploration did not occur. The two most immediate hypotheses for why this may be the case are that the solver had already reached a global maximum, or that it was too late in the solver process for the exploration to be useful. That is, the early solution given by the Tabu

search may have locked the solver into a region of the solution space such that it was at a sufficiently isolated a local maximum so that no small permutation could reach a better solution.

However, the hybrid did prove promising in one way: Where a 2000-iteration Tabu search would usually take on the order of five to fifteen minutes, the Hybrid, by virtue of using Simulated Annealing for the second half of its cycle, used roughly half that, with no loss in solution quality. This is because both the hybrid and the Tabu search see almost all the improvement in the first few hundred iterations, something seen in Figure 5.14.

5.8.5 – Solution Feasibility

As one would hope, and is shown by Figures 5.19 and 5.20, the solutions generated by the solvers are far more feasible than the raw input data. Where the initial guess, taken from the raw visibility data, has large amounts of conflict and a large number of handover violations, the final result is massively better, with a very large reduction in conflict time – sometimes eliminating all conflict entirely – and a significant reduction in the number of handover violations, with the better solvers often fully resolving them. Even though this solution may not be fully feasible directly, this is still fairly close to an ideal result, as a handover violation is fairly easy to resolve by manually tweaking the solution whereas a conflict is not. This is also a fairly standard outcome; it is not uncommon for the final output from these algorithms to require some manual improvement missed by the solver.

5.8.6 – Optimal Solver Choice

Using the obtained results, the Tabu search seems to be a fairly good candidate as the ideal solver for this application and problem formulation; though it may not be as capable of achieving global maxima quite as frequently as the better genetic algorithms, it still obtains high-quality solutions, and does so in fewer iterations than any other heuristic.

Chapter 6 – Conclusions

6.1. Conclusion

The primary goals of this work were to:

- Apply heuristic algorithms as seen in the literature to a real-world problem instance, or approximation thereof
- Perform a comparative evaluation of these algorithms with regard to their efficiency in solving the problem
- Attempt to find an improved algorithm to achieve even greater performance

To achieve these goals, a solver program was designed and written to apply the selected heuristics to randomly-generated problem instances, as well as any other problem provided in a similar format. This solver was designed to account for several real-world constraints normally ignored in the literature, including handover violations, communication limits, and unequal asset values. The output of the program, when using each solver, was then analyzed.

Through this analysis, the Tabu search positioned itself as a viable candidate for the best choice of algorithm of those tested for this application.

Though the third goal did not ultimately end up being fully achieved – the designed hybrid algorithms were no better than the defaults – it remains a feasible goal as only a limited amount of time was focused on this objective, and future work here may prove more rewarding.

6.2 – Future Work

Due to time limitations, not every practical constraint was able to be modelled in this work. In particular, the centrality of a pass and the effect of an eclipse penalty were not studied. Though this should not affect the comparative results from the different solvers, it may significantly alter the nature of the generated solutions. Therefore an exploration of these two constraints would be a significant contribution.

Another improvement would be to try more solver types, including some of the more advanced heuristics present in the literature which were forgone here. This includes a Hash version of the

Struggle-Strategy genetic algorithm and a full implementation of a Tabu Search, as well as some more exotic, complex, and effective algorithms mentioned in the introductions of several of the cited works.

It would also be worth investigating the effect of the fitness function or mutation settings in more detail; the settings in this work were based on the literature where available and some testing and small-scale optimization elsewhere, and it is possible that different settings may affect convergence speed or reliability of the solution quality.

A large expansion to this work would be to use an actual problem instance. The randomly-generated problems, though intended to reflect the nature of a real-world problem, ultimately may not be fully representative of one. For example, the randomly-generated problems usually had few spacecraft handover violations, where a real-world case may have significantly more.

Finally, more hybrid algorithms could be tested to see if there are potential improvements to be made to the existing algorithms, either in terms of solution quality, speed, or both.

Appendices

Appendix A – Subset of Generated Problem Instance

Where multiple cycles of a solver were used on the same problem instance, as in Sections 5.2, 5.4, and 5.6, this is a portion of the problem instance that was used, selected to demonstrate its structure. Its format is identical to that used as input to the solver program.

Sample Spacecraft Definition:

id:RandSC_4	- An identifier string used internally
handover:2.3792548609530506	- Minimum handover time
weight:2.0545891680755677	- Relative spacecraft weight value
required:89604.97370507567	- Required communication time amount

Sample Ground Station Definition:

id:RandGS_3	- An identifier string used internally
handover:9.623904416641736	- Minimum handover time
latency:1.9595304162542604	- Network latency value
weight:6.3342855864701955	- Relative station weight value
dishes:3	- Number of usable dishes

Sample Visibility Window:

satellite:RandSC_6	- Which satellite is involved
station:RandGS_1	- Which ground station is involved
start:386170.86910751404	- Start time
end:411524.54620013153	- End time

Appendix B – Generation of Random Problems

Where a randomly-generated-problem was used, it was generated dynamically at the start of the program. First, an overall time frame was created, from $t_0 = 0$ and t_1 being a random real number between 1000 and 500000. All random numbers were generated using the native Java pseudorandom libraries.

Then 12 ground stations were generated with bounded random parameters, as seen in Table A1:

Parameter	Minimum Value	Maximum Value	Notes
Dish Count	1	8	Integer; 67% chance of only one dish
Weight	0	10	Any decimal value
Latency	0	4	Any decimal value
Handover Time	0	40	Any decimal value
Communication Limit	1000	500000	Any decimal value; 87.5% chance of being Infinity

Table B1 – Randomized Ground Station Parameters

Next, 32 ground stations were generated with bounded random parameters, as seen in Table A2:

Parameter	Minimum Value	Maximum Value	Notes
Required Time	0	120000	Any decimal value
Weight	0	10	Any decimal value
Handover Time	0	10	Any decimal value
Max Link Length	1000	100000	Any decimal value
Communication Limit	1000	500000	Any decimal value; 87.5% chance of being Infinity

Table B2 – Randomized Spacecraft Parameters

Finally, for each pair of spacecraft and ground stations, with a 12.5% chance, between one and six visibility windows were created, with start and end times randomly-chosen in the overall time frame (with a check to ensure two windows do not overlap).

References

[1] Pelton et al, *Handbook of Satellite Applications*. New York: Springer, 2013.

[2] European Space Agency. (2015) ESA'S DEEP SPACE TRACKING NETWORK. Brochure. [Online]. http://download.esa.int/esoc/estrack/esa_estrack_brochure_2015_EN.pdf

[3] Xhafa et al, "Evaluation of Genetic Algorithms for Single Ground Station Scheduling Problem," in *2012 26th IEEE International Conference on Advanced Information Networking and Applications*, March 2012, pp. 299-306.

[4] Xhafa et al, "A Tabu Search Algorithm for Ground Station Scheduling Problem," in *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, May 2014, pp. 1033-1040.

[5] Xhafa et al, "Steady State Genetic Algorithm for Ground Station Scheduling Problem," in *2013 IEEE 27th International Conference on Advanced Information Networking and Applications*, March 2013, pp. 153-160.

[6] Xhafa et al, "A Simulated Annealing Algorithm for Ground Station Scheduling Problem," in *2013 16th International Conference on Network-Based Information Systems*, September 2013, pp. 24-30.

[7] Lala et al, "On Local vs Population-based Heuristics for Ground Station Scheduling," in *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, July 2015, pp. 267-275.

[8] Kanta Vekaria and Chris Clack, "Selective Crossover in Genetic Algorithms: An Empirical Study," University College London, London, n.d.

[9] Xhafa et al, "Evaluation of struggle strategy in Genetic Algorithms for ground stations scheduling problem," *Journal of Computer and System Sciences*, vol. 79, pp. 1086-1100, February 2013.

[10] Junzi Sun and Fatos Xhafa, "A Genetic Algorithm for Ground Station Scheduling," in *2011 International Conference on Complex, Intelligent, and Software Intensive Systems*, July 2011, pp. 138-145.

[11] Hongrae Kim and Young Keun Chang, "Mission scheduling optimization of SAR satellite constellation for minimizing system response time," *Aerospace Science and Technology*, vol. 40, pp. 17-32, October 2015.

[12] Xhafa and Herrero et al, "A Comparison Study on Meta-Heuristics for Ground Station Scheduling Problem," in *2014 International Conference on Network-Based Information Systems*, September 2014, pp. 172-179.

[13] Alda Kika and Silvana Greca, "Multithreading Image Processing in Single-core and Multi-core CPU

using Java," *International Journal of Advanced Computer Science and Applications,* vol. 4, no. 9, pp. 165-169, 2013.

[14] Cairns et al, "The origin of mutants," *Nature,* vol. 335, pp. 142-145, September 1988.

[15] Xhafa et al, "A Struggle Genetic Algorithm for Ground Stations Scheduling Problem," in *2012 Fourth International Conference on Intelligent Networking and Collaborative Systems*, September 2012, pp. 70-76.

[16] Zufferey et al, "Graph colouring approaches for a satellite range scheduling," *J Sched,* vol. 11, pp. 263-277, May 2008.

[17] Xu and Lou et al, "Task Scheduling of Satellite Ground Station Systems Based on the Neighbor-area Search Algorithm," in *2013 Ninth International Conference on Natural Computation (ICNC)*, July 2013, pp. 1830-1834.

[18] Andrea Olsson, *Particle Swarm Optimization: Theory, Techniques and Applications*. Commack: Nova Science Publishers, 2011.

[19] Zhan et al, "Orthogonal Learning Particle Swarm Optimization," *IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION*, vol. 15, no. 6, pp. 832-847, December 2011.

[20] Zhang et al, "Multi-satellite control resource scheduling based on ant colony optimization," *Expert Systems with Applications*, vol. 41, pp. 2816-2823, 2014.

[21] Xhafa and Herrero et al, "Using STK Toolkit for Evaluating a GA base Algorithm for Ground Station," in *2013 Seventh International Conference on Complex, Intelligent, and Software Intensive Systems*, July 2013, pp. 265-273.

[22] Fred Glover and Rafael Marti, "Tabu Search," Leeds School of Business; Universidad de Valencia, Boulder, Colorado; Valencia, Spain, n.d.

[23] Wang et al, "Towards dynamic real-time scheduling for multiple earth observation satellites," *Journal of Computer and System Sciences*, vol. 81, pp. 110-124, July 2015.

[24] Michel Gendreau and Jean-Yves Potvin, "TABU SEARCH," Universite de Montreal, Montreal, Canada, n.d.

[25] Spangelo et al, "Optimization-based scheduling for the single-satellite, multi-ground station communicationproblem," *Computers & OperationsResearch*, vol. 57, pp. 1-16, December 2015.