Flipping the switch on local exploration: Genetic Algorithms with Reversals

Ankit Grover\textsuperscript{1,2}, Vaishali Yadav\textsuperscript{2}, Bradly Alicea\textsuperscript{1,3}
\textsuperscript{1} Orthogonal Research and Education Laboratory, Champaign-Urbana, IL and Worldwide
\textsuperscript{2} Manipal University Jaipur, India
\textsuperscript{3} OpenWorm Foundation, Boston, MA
E-mail: agrover112@gmail.com

Abstract
One important feature of complex systems are problem domains that have many local minima and substructure. Biological systems manage these local minima by switching between different subsystems depending on their environmental or developmental context. Genetic Algorithms (GA) can mimic this switching property as well as provide a means to overcome problem domain complexity. However, standard GA requires additional operators that will allow for large-scale exploration in a stochastic manner. Gradient-free heuristic search techniques are suitable for providing an optimal solution in the discrete domain to such single objective optimization tasks, particularly compared to gradient based methods which are noticeably slower. To do this, the authors turn to an optimization problem from the flight scheduling domain. The authors compare the performance of such common gradient-free heuristic search algorithms and propose variants of GAs which perform well over our problem and across all benchmarks. The Iterated Chaining (IC) method is also introduced, building upon traditional chaining techniques by triggering multiple local searches instead of the singular action of a mutation operator. The authors will show that the use of multiple local searches can improve performance on local stochastic searches, providing ample opportunity for application to a host of other problem domains.

Keywords: NP-Hard, heuristic, gradient, Iterated Chaining, Genetic Algorithms

Introduction
In a wide range of complex systems, there are a number of problem domains where substructure, noise, and local variability are prevalent. These properties give rise to a very rough landscape, with many local minima. Genetic algorithms (GAs) can provide a means to navigate such landscapes, but standard operators such as mutation and recombination limit the ability to escape local minima without destroying accumulated search information.

Fortunately, a mechanism has been identified that is often used at the phenotypic or epigenetic levels of evolutionary systems: switching. Switching is the transition from one state to another by means of a quick, abrupt change. Switching can be characterized in forms such as electrical switches, molecular triggers, step functions, and first-order phase transitions. While switching can be observed in many various natural and mechanical systems, the application to algorithms requires a computational encoding of the phenomenon, such like what is done in the case of mutation and crossover operators in a standard GA chromosome.
One way in which switching is useful for computational environments is in modeling switching as a computational universal phenomenon. For example, cellular automata [1] to understand how the implementation of rules can produce complex patterns that approximate those seen in biological systems. When applied in parallel, these rules result in switching points at the macro-level, which introduces heterogeneity in the pattern. More relevant to the kinds of switching observed in evolutionary systems, adaptive behaviors enabled by switching and found in the \textit{lac} operon system, logic circuits, and RNA phenotypes. In two models of such systems, innovation via mutation (\textit{lac} operon, [2]) and increased speed of evolutionary innovations (logic gates and RNA phenotypes, [3]) serve as examples of how switching can be encoded in a computational framework.

Historically, chaining methods are used in a wide variety of algorithms and can be applied to many different types of problem domains. One example of this is in applications to multiple genome comparisons. In this problem domain, this is done by constructing a maximum weighted path on a weighted directed acyclic graph [4]. Optimal solutions to this type of problem [5, 6] allows for spatial and temporal specificity, which decomposes the problem domain to a series of local minima. Chaining is also used for optimization in a number of problem domains, including nearest-neighbor, lexical, and backwards [7]. Lexical chaining is also related to word-sense disambiguation [8], which uses a switching mechanism in a linguistic context. This particular problem poses switching as a means of discrete phase change, and lends itself to exploring subdomains of a problem.

\section*{Methodology}

\subsection*{Origins in Previous Work}

A flight scheduling problem can be used to demonstrate the efficacy of our approach. The \textit{fliscopt} approach\footnote{Github repository: \url{https://github.com/Agrover112/fliscopt}; PyPi (latest release): \url{https://pypi.org/project/fliscopt/}} utilizes a dataset of schedules consisting of features such as flight cost in dollars, departure, arrival times, city of origin, destination. This makes our problem NP-Hard [9] and heuristic algorithms with optimal solutions are preferred. Our dataset consists of six-city names and airport abbreviations [10]. A single person is assumed to be present at each city of origin. The domain $D$ in this case is a discrete vector $v(0, 9)$ of magnitude 12, representing round trips for the six-city example. The final schedule must take this condition into account. The search space is of the order of $10^{12}$, since 10 possibilities exist for each of the 12 trips. From this, a cost function can be created which can efficiently create flight schedules from origin cities to a particular destination city with least total price and total wait times.

Flight scheduling is often closely related to fleet assignment [11]. The authors [12] use a Lagrangian relaxation along with sub-gradient methods. Similarly the authors in [13] use an MAGS which is based on the Ant Colony Optimization Method [14]. Simple heuristic search techniques are used in lieu of gradient-based information, while alternate variants of a standard GA are implemented. Some of these variants have been used by [15] for image encryption, also
for Generation of S-Boxes [16] and by authors [17] using Reverse Hill Climbing for the Busy Beaver Problem [18], the authors in [19] use an Binary Differential Evolution algorithm for airline revenue management which is NP-Hard problem as well. Similarly the authors of [20] use a GA with 2-Dimensional mutation and crossover operations for Aircraft Scheduling.

The Nevergrad software package [21] uses a similar non-gradient based principle with single iteration of chaining, however our method involves the use of mutation operators and multiple iterations. By comparison, early stopping criteria combined with mutations are used to influence local minima. Also our second proposed variant is similar to [22] which uses Tournament Selection instead and has been applied to solve the parking lot path optimization problem [23]. GAs with reversals have also been by [24] for the Scary Parking Lot problem.

**Defining The Fitness Function**

In the following problem, there is a need to meticulously define the cost (fitness) function (shown in Appendix I) such that it takes into account both the cost of flights(dollars) but also the waiting times between different flights. The wait times need to be minimized and so does the cost. Thus, the final function is an additive of both quantities. The cost (fitness) function penalizes cases where flights are too far apart. The function \( \text{get}_{\text{minutes}}( ) \) is used to convert hours into minutes. The time complexity of our fitness function is \( O\left(\frac{L}{2} + \frac{L}{2}\right) \), where \( L \) is the length of the solution. More information about time complexity for all tested algorithms can be found in Appendix II.

Benchmark functions are chosen such that it would be difficult to converge easily. This was done by choosing functions such that they followed certain properties [24, 25] of high dimensionality, since most real world problems are multi-dimensional in nature; they were non-separable, continuous, convex and unimodal. Average cost (\( \text{cost} \)), standard deviation(\( \sigma \)), number of function evaluations(\( n. f. e \)), and runtime (ms) are used as metrics for evaluation.

**Switching Mechanism**

Two types of switching are utilized here: intra-algorithmic and inter-algorithmic. For intra- algorithmic switching, the change in the phenomenon takes place inside the algorithm itself. In the case of inter- algorithmic switching, the switching mechanism occurs between algorithms, rather than within a single algorithm. The authors compared the following algorithms: Simulated Annealing (SA), Random Search (RS), Hill Climbing (HC), standard GAs, and their proposed variants: GA with Reversal Genetic Operations, GA with Reversals, GA with Stochastic Search Reversal, and the IC method.

**GAs with Reverse Operations**

In this algorithm, a GA in which the order is reversed is used to perform the genetic operations of mutation and crossover (see Figure 1). Thus, the GA consists of an Elitist selection step followed by crossover and then a mutation step. The probability of mutation \( P_{\text{mutation}} \) thus, becomes the probability of crossover. \( P_{\text{mutation}}, P_{\text{crossover}} \) are chosen as default of 0.2. The GA
with Reverse Operations converges a bit faster to optimal cost. This method can be understood as a coarse approximation of the Differential Evolution method without the differential weight $F$ and selection step at the start of the iteration. Being an approximation, this method provides some increase in performance when GA fails. Therefore, Figure 1 demonstrates that simply reversing operations is not enough. To potentially overcome this limitation, a reversal strategy is described in the next section.

![Figure 1: Plot for a standard GA (A), and for a GA with Reverse Operations (B) using the same seed for a single run.](image)

### GAs with Reversals

GA reversals are a form of switching which aims to introduce genetic diversity by means of reversing the maximization or minimization step. Since this reversal happens within the same GA it is a form of intra-algorithmic switching. Our idea is somewhat similar to [26] two neural networks where one tries to minimize whereas the other tries to maximize the other in a zero-sum game. Here the competition happens within the optimization algorithm itself.

Elitist selection is useful when the algorithm can get stuck in the loss landscape of functions with many local minima, and is slightly difficult to converge. This has been done by using the reversal step. Minimizing for the objective, the objective function is maximized for a fixed number of steps defined by the $\text{step}_\text{length}$ parameter, and the $\text{num}_\text{reversals}$ is controlled by the $n_k$ parameter using Equation 1. Reversals are performed for all iterations other than the first iteration. The iteration at which the reversal starts is the iteration $i$ divisible by $n_k$. By using such a short reversal process, local minima can easily be escaped. This forces the algorithm to find better search spaces.

### GAs with Stochastic Reversals

While an exemplar random search algorithm is implementable, performance on benchmarks may yield poor performance. Thus, GAs with stochastic reversals can be used
(Figure 2A), in which the algorithm performs a local random search in reverse objective maximization mode similar to how it does for GA with Reversals. This is shown in comparison to GAs with Reversals (Figure 2B), which itself demonstrates performance improvements compared to a standard (vanilla) GA. Figure 2A clearly shows how a reverse Stochastic Search perturbs the solution space, leading to worse solutions during the reversal process. In this mode the GA provides diversity in solutions by using Random Search instead. A consequence of this might be it reaching arbitrary search spaces in the landscape while also escaping local minima.

\[
\text{num}_{\text{reversals}} = \frac{\text{num}_{\text{generations}}}{n_k},
\]

where \( \text{num}_{\text{generations}} \in \mathbb{Z} \), \( n_k \in \mathbb{N} \) & \( n_k \leq \text{num}_{\text{generations}} \), \( \forall i \in \text{num}_{\text{generations}}, i \neq 0 \) \[1\]

Figure 2: Random Search Reversal (A), and GA with Reversals (B). The graph on the left figure clearly shows how a reverse Stochastic Search perturbs the solution space leading to worse solutions during the reversal process.

**IC Algorithm**

Solution initialization can be used in heuristic algorithms, which provides any algorithm a good starting point in the solution space. This involves providing an initial starting population/solution to a GA such as Random Search (RS) and Hill Climbing (HC). IC can be simplified to an Iterated Local Search by using two similar algorithms and removing the operators between them [27, 28]. Prior knowledge is used in the form of solution initialization. Full code for the IC algorithm can be found in Appendix III.
For the purposes of this paper, the initial solution will be referred to as weights, initialized weights, or solution weights interchangeably. This is the convention used in deep learning, and suits this context as well. In the following scenario, solution initialization is a discrete approximation of weight initialization in neural networks. Major difference being the initialized weights are from a pre-trained algorithm. Prior knowledge plays an important role in deep learning. This is especially true when the model is based on both prior knowledge and temporal recurrence. In such cases, a deep learning algorithm can outperform those without pre-training [29, 30]. A GA-based method will be used where instead of a single initialization, re-initialization is iteratively performed. GAs use mutation and crossover, which prevent stagnation and locally improve the solutions.

In our meta-algorithm, two algorithms are used to transfer its best solutions to one another over a certain number of rounds. Each algorithm learns from the other to a certain extent. The parameter rounds is analogous to epochs in training machine learning algorithms. Each round consists of running the Initial algorithm and the Chained Algorithm which uses weights from above. The Chained weights are then recursively implemented in the Initial algorithm. For the final iteration of the meta-algorithm (rounds $- 1$), weights are passed directly to the Chained instead until terminating.

Further Methodological Considerations

Overcoming Drawbacks with IC

One of the major issues of using such an initialization scheme is to account for Initial algorithm’s solution starting to deteriorate, and thus providing bad weights to the Chained Algorithm as a result decrease the global cost. Running the algorithm for a large number of rounds only aggravates the issues. Parameters are required which can quickly prevent divergence of the global cost during a deteriorating solution. Global cost in the following context refers to the cost of the meta-algorithm itself and cost refers to the local cost of the constituent algorithms. Our goal is to thus minimize the global cost. Thus, an Early Stopping mechanism such as that used in Deep Neural Networks controls the global cost [31].

The authors introduce two parameters $n_{obs}$: the number of observations over cost is averaged, and tolerance: the amount up to which a divergence of global cost is tolerated (see Equation 2). A larger value in $n_{obs}$ should handle divergence much more. In the given equation the right-side calculates the average of the global cost up to $n_{obs}$ and the left-side the tolerance tries to minimize the current cost by subtracting a factor. If the cost is substantially higher than the $n_{obs}$ (despite the tolerance factor), the meta-algorithm terminates early, preventing waste of resources. A default value of tolerance of 90 and $n_{obs}$ of 2 were used. Very high values of tolerance cause termination in early rounds. A demonstration of the early stopping algorithm and the consequences of limited diversity are shown in Figure 3.
The clock chose the Linux (WSL) 2. Moreover, Linux be native Subsystem since is Core(s) RAM 2.60GHz, were a pc the using carried 11 with Hardware and Performance Issues there might be cases where our Itertated Chaining method might fail. This method has some drawbacks. One notable consequence is a lack of transfer learning from one cost landscape to another. The model depends on the quality of the weights, here the performance is determined by how optimized our initialized weights are with respect to our minimization problem. Therefore, it is important to have good weights. Another drawback is that the Initial algorithm being weak, can often perturb the results to a degree that the Chained algorithm then fails to further reduce the cost and thus our improvements converge without providing a benefit with respect to an optimal solution. GAs were observed to converge the most with respect to Chained algorithms. A similar phenomenon is observed with two similar algorithms: each algorithm converges simultaneously, and this leads to the meta-algorithm being stuck as shown in Figure 3.

To overcome the lack of diversity in these representations, the OnePointMutation operator [32] is used. A mutation is introduced randomly after each Initial algorithm, while a definite mutation to the solution of the Chained algorithm. This provides the necessary diversity required and also prevents the algorithms from being stuck and explores the search space better. Introducing mutation greatly decreases the cost of our problem. The final performance of this method depends purely on the algorithms chosen and the problem at hand, therefore there might be cases where our Itertated Chaining method might fail.

**Hardware and Performance Issues**

All the experiments were carried out on a pc with Windows 11 using Windows Subsystem for Linux, using an Intel(R) Core(TM) i7-9750H CPU with 2.60GHz, 2592 Mhz, 6 Core(s) and 12 Logical Processor(s) with a RAM of 32.0 GB with available physical memory of 16.5 GB. It is to be noted that since experiments were performed on Windows Subsystem for Linux(WSL) 2, the wall clock time on a native Linux distribution might be faster. Moreover, background processes running on the system have a direct impact on the wall clock time. Wall clock time will be referred to as run-time. To further decrease the run-time the authors chose the PyPyv7.3.5 [33] interpreter instead of the native CPython implementation of Python along with asynchronous multiprocessing on all the cores.

\[
\text{if } \text{cost} - R > \sum_{n_{\text{obs}}}^{\text{scores}} \frac{\text{scores}}{-n_{\text{obs}}} \text{ then}
\]

\[
\text{return } \text{best soln', last score', scores, n.f.e}
\]

\[
\text{end if}
\]

, where \( \text{R} \in \mathbb{Z}, \text{tolerance} \leq \text{R} \leq 100 \) from a discrete \( \mathbb{U} \) dist.

\[
, \text{scores}[i : -n_{\text{obs}}] \text{ last n cost observations.}
\]
Figure 3. Clockwise from Upper Left: Upper Left: A demonstration of divergence in an IC algorithm (no stopping mechanism - converges towards a higher score). Upper Right: After applying Early Stopping criteria (converges towards a lower score). Bottom: A GA-GA Iterated Chain that fails to provide any diversity and converges without mutations.

Results

Results and Observations

The run-time and descriptive statistics for single runs of our various candidate algorithms are reported in Table 1. The time for all runs is 20 times that of a single run. In general, increasing the number of iterations (as was conducted for Hill Climbing, RandomSearch, and Simulated Annealing algorithms) demonstrates that increased run-time results in n.f.e (number of function evaluations) while showing marginal to no improvement relative to all other proposed algorithms.

By using Chaining (random search and hill climbing), minimum cost is achieved, albeit with a high standard deviation (σ). However, this is still less than the average of both constituent algorithms. The relative cost is also only 270 greater than that of our GA and
significantly less than the cost of both. However, the IC algorithm has an obviously high run-time and n. f. e due to the nature of its fundamental operations. The GA and its variants (see Table 2) converged completely using Brown[13 dim], Booth[33], Ackley_N2 [34], Sphere[35], Three_hump_camel [36], and Schwefel[37]. While these functions were completely minimized, other algorithms (Rosenbrock [38], Griewank [39], Zakharov [40] of 13 dimensions, Schaffer_N1 [41], Matyas [42, 43, 44]) did not. The full results for these functions are not reported. Run times and descriptive statistics for the different algorithms can be found in Appendix IV.

Table 1. Run times and descriptive statistics for our experimental conditions (different search strategies). Runtime measured in milliseconds (ms). Initial parameter values: \( num_{reversals} = 1 \) and \( step_{length} = 100 \).

|        | cost  | \( \sigma \) | \( cost_{min} \) | \( cost_{max} \) | n.f.e | ms   |
|--------|-------|------------|-----------------|-----------------|-------|------|
| SGA    | 2780.9| 205.75     | 2356            | 3081            | 1000  | 9.36 |
| GAwRo  | 2629.8| 213.79     | 2356            | 3004            | 1000  | 9.66 |
| GAwR   | 2593  | 183.89     | 2356            | 2973            | 1099  | 10.16|
| HC     | 4177.7| 817.72     | 2759            | 5839            | 328   | 0.33 |
| RS     | 4545.3| 271.95     | 4143            | 5165            | 100   | 0.17 |
| SA     | 3726.5| 578.16     | 2759            | 4679            | 512   | 0.24 |
| RS+HC  | 3050.7| 399.72     | 2356            | 3771            | 1657  | 2.23 |
| GARSRev| 2592.9| 168.45     | 2356            | 2888            | 1099  | 9.97 |

In most multi-dimensional functions it is observed our GA variants outperform GAs with better cost, \( \sigma \) with slightly larger run-times, and in terms of n. f. e. Moreover, another thing to note is the \( num_{reversals} \) was set at a minimum of 1, showing how a single reversal can drastically help prevent problems of getting stuck in local minima and improve performance (see Appendix 1 and Table 1). All of our variants, particularly a standard GA with Reversals, can be used in real world problems where the objective function of our problem is complex, multi-dimensional with a very small trade-off for slightly higher n. f. e and run-time.

**Discussion**

In this paper, the IC method is applied to a problem domain, which ultimately improves local performance in GAs. This technique (algorithmic switching) also relates to switching in

---

\(^2\) Abbreviations: SGA = standard GA; GAwRo = GA with Reverse Operations; GAwR = GA with Reversals; HC = Hill Climbing; RS = Random Search; SA = Simulated Annealing; RS+HC = Random Search/Hill Climbing Hybrid; GARSRev = GA with Random Search as Reversals.
biological systems, as well as earlier techniques in algorithmic design. When considering the overall effectiveness of IC, performance can be variable, even on the same problem or benchmark. Performance of the switching mechanism depends on the initialization conditions (seed values) of the algorithm. When the initial seed value is not fixed, it allows the algorithm to build the diversity necessary to engage in robust local searches as well as move to new regions to explore.

The IC method was only tested on a single optimization problem (flight scheduling). Other benchmarks were considered, but given the algorithm’s dependence on local searches, it would be futile to implement simple benchmark functions. Yet IC allows for new computational strategies to be refined. Consider two algorithms (Random Search and Hill Climbing) that are specialized for local optimization. IC outperforms a combination of such local optimization techniques most of the time. Yet reproducibility is difficult, largely but not exclusively due to the high variability between local searches.

Future Work

The current work is limited to flight scheduling optimization in the discrete domain. As such, the computational experiments were performed for a limited dataset, and more esoteric evolutionary methods such as Ant Colony Optimization (ACO) [45], Particle Swarm Optimization (PSO) [46], or Evolution Strategies (ES) [47] were not fully evaluated. Although the IC method with genetic operators can be theoretically used with any algorithm, in-depth investigation of chaining algorithm performance using more algorithms and problem domains is necessary.

There are other application domains which might also be well-suited to this approach. One example is to use the Braatenberg Vehicle developer BraGenBrain [48] as a benchmark for GA performance on the Traveling Salesman Problem (TSP). TSP benchmarks are particularly informative, since the flight scheduling problem is a special case of this complexity class, and our initial tests involved a limited dataset. BraGenBrain is a GA that controls the growth and configuration of a neural network that interacts with a BV body and the external environment. This problem domain provides a mixture of optimization goals (and thus local optima) that might highlight the strengths of IC. More generally, IC might excel at multi-objective problem domains.

Acknowledgements

The authors would like to thank Chaitanya Karyal for their support. Thanks also go to Jones Granatyr for providing the dataset. Additional feedback came from members of the Saturday Morning NeuroSim group.

References

[1] Wolfram, S. (2002). A New Kind of Science. Wolfram Media, Champaign, IL.

[2] Kashtan, N., Noor, E., and Alon, U. (2007). Varying environments can speed up evolution. *PNAS*, 104(34), 13711–13716.
[3] Mayo, A.E., Setty, Y., Shavit, S., Zaslaver, A., and Alon, U. (2006). Plasticity of the cis-Regulatory Input Function of a Gene. *PLoS Biology*, 4(4), e45.

[4] Abouelhoda, I.M. and Ohlebusch, E. (2005). Chaining algorithms for multiple genome comparison. *Journal of Discrete Algorithms*, 3(2–4), 321-341.

[5] Myers, E.W. and Miller, W. (1995). Chaining multiple-alignment fragments in sub-quadratic time. *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 38–47.

[6] Zhang, Z., Raghavachari, B., Hardison, R., and Miller, W. (1994). Chaining multiple-alignment blocks. *Journal of Computational Biology*, 1, 217–226.

[7] Nelken, R. and Shieber, S.M. 2007. Lexical Chaining and Word-Sense-Disambiguation. *Harvard Computer Science Group Technical Report*, TR-06-07.

[8] Schutze, H. (1998). Automatic word sense discrimination. *Computational Linguistics*, 24(1), 97–123.

[9] Mezentsev, Y., and Estraykh, I. (2018). An optimal fleet assignment and flight scheduling problem for an airline company. *Proceedings of the Russian Higher School Academy of Sciences*, 3, 74–90. [https://doi.org/10.17212/1727-2769-2018-3-74-90](https://doi.org/10.17212/1727-2769-2018-3-74-90).

[10] Granatyr, J. (2016, October 25). *Programando a Inteligência Coletiva*. iaexpert.academy. [https://iaexpert.academy/2016/10/25/review-de-livro-programando-a-inteligencia-coletiva/](https://iaexpert.academy/2016/10/25/review-de-livro-programando-a-inteligencia-coletiva/)

[11] Sherali, H.D., Bish, E.K., Zhu, X. (2006). Airline fleet assignment concepts, models, and algorithms. *European Journal of Operational Research*, 172, 1-30.

[12] Yan, S. and Tseng, C-H. (2002). A passenger demand model for airline flight scheduling and fleet routing. *Computers and Operations Research*, 29(11), 1559–1581.

[13] Caetano, D. J., and Gualda, N. D. F. (2015). Solving the Integrated Schedule Generation and Fleet Assignment Problem: an ACOBased Metaheuristic Approach. *Journal of Transport Literature*, 9(3), 30–34.

[14] Dorigo, M. and Stutzle, T. (2005). Ant Colony Optimization. MIT Press, Cambridge, MA.

[15] Gholamiyan, F., and Yaghoobi, M. (2015). Utilizing reverse genetic algorithms and fractal theory for encrypting images. *International Congress on Technology, Communication and Knowledge (ICTCK)*, Curran Associates, Red Hook, NY.
[16] Ivanov, G., Nikolov, N., and Nikova, S. (2016). Reversed genetic algorithms for generation of bijective s-boxes with good cryptographic properties. Cryptography and Communications, 8(2), 247–276.

[17] Jones, T. and Rawlins, G.J.E. (1993). Reverse Hillclimbing, Genetic Algorithms and the Busy Beaver Problem. Proceedings of the 5th International Conference on Genetic Algorithms, Morgan Kaufmann, San Francisco, CA.

[18] Radó, T. (1962). On non-computable functions. Bell System Technical Journal, 41(3), 877–884.

[19] Karbassi Yazdi, A., Kaviani, M. A., Hanne, T., and Ramos, A. (2020). A binary differential evolution algorithm for airline revenue management: a case study. Soft Computing, 24(18), 14221–14234.

[20] Tsai, M. W., Hong, T. P., and Lin, W. T. (2015). A Two-Dimensional Genetic Algorithm and Its Application to Aircraft Scheduling Problem. Mathematical Problems in Engineering, 2015, 1–12.

[21] Rapin, J. and Teytaud, O. (2018). Nevergrad: a gradient-free optimization platform. Github, https://github.com/FacebookResearch/nevergrad

[22] Bobukh, E. (2017). Genetic Algorithm, in Reverse Mode. Microsoft Research blog, July 24. https://www.microsoft.com/en-us/research/blog/genetic-algorithm-in-reverse-mode/

[23] Wang, Y., Wu, J., and Wang, Y. (2017). An improved hierarchical A * algorithm in the optimization of parking lots. AIP Conference Proceedings, 1864(1), 020031.

[24] Bozinovski, S. and Fulgosi, A. (1976). The influence of pattern similarity and transfer learning upon the training of a base perceptron B2. Proceedings of the International Symposium Informatica, 3-121-5.

[25] Manickam, R. and K. Mahesh, K. (2020). A Performance Analysis of Pre-trained Neural Network and Design of CNN for Sports Video Classification. International Conference on Communication and Signal Processing (ICCSP), 0213-0216.

[26] Schmidhuber, J. (2020). Generative Adversarial Networks are special cases of Artificial Curiosity (1990) and also closely related to Predictability Minimization (1991). Neural Networks, 127, 58-66.

[27] Yang, Z., Salakhutdinov, R., and Cohen, W.W. (2017). Transfer Learning for Sequence Tagging with Hierarchical Recurrent Networks. arXiv, 1703.06345.
[28] Raskutti, G., M.J. Wainwright, and Yu, B. (2011). Early stopping for non-parametric regression: An optimal data-dependent stopping rule. *Allerton Conference on Communication, Control, and Computing*, 1318–1325.

[29] Obitko, M. XI. Crossover and Mutation. *Marek Obitko blog*, http://www.obitko.com/. Retrieved 2011-04-07.

[30] Katoch, S., Singh Chauhan, S., and Kumar, V. (2020). A review on genetic algorithm: past, present, and future. *Multimedia Tools and Applications*, October 31, 1–36.

[31] Ansari Ardeh, M. (2016) BenchmarkFcns: A collection of mathematical test functions for benchmarking optimization algorithms. Github. https://github.com/mazhar-ansari-ardeh/BenchmarkFcns

[32] Jamil, M. and Yang, X-S. (2013). A literature survey of benchmark functions for global optimization problems. *International Journal of Mathematical Modelling and Numerical Optimisation*, 4(2), 150–194.

[33] Booth Function. http://www.sfu.ca/~ssurjano/booth.html

[34] Ackley, D. H. (1987). A connectionist machine for genetic hillclimbing. Kluwer Academic Publishers, Boston, MA.

[35] Sphere Function. http://www.sfu.ca/~ssurjano/spherref.html

[36] Three-hump Camel Function. http://www.sfu.ca/~ssurjano/camel3.html

[37] Schwefel, H.P. (1981). Numerical Optimization for Computer Models. John Wiley Sons, New York.

[38] Generalized Rosenbrock's function. Retrieved 2008-09-16.

[39] Virtual Library of Simulation Experiments: Test Functions and Datasets. http://www.sfu.ca/~ssurjano

[40] Zakharov Function. http://www.sfu.ca/~ssurjano/zakharov.html

[41] Mishra, S.K. (2006). Some New Test Functions For Global Optimization And Performance of Repulsive Particle Swarm Method. Available at SSRN: https://ssrn.com/abstract=926132
[42] Al-Roomi (2015). Modified Schaffer’s Function Number 1. Power Systems and Evolutionary Algorithms. Al-Roomi website, https://al-roomi.org/benchmarks/unconstrained/2-dimensions/96-modified-schaffer-s-function-no-1

[43] Matyas Function. http://www.sfu.ca/~ssurjano/matya.html

[44] Test functions for Optimization. https://en.wikipedia.org/wiki/Test_functions_for_optimization

[45] Kaushalm, J. (2012). Advancements and Applications of Ant Colony Optimization: A Critical Review. International Journal of Scientific & Engineering Research, 3(6), 1-5.

[46] Ab Wahab, M.N., Nefti-Meziani, S., Atyabi, A. (2015). A Comprehensive Review of Swarm Optimization Algorithms. PLoS One, 10(5), e0122827.

[47] Beyer, H-G. (2007). Evolution Strategies. Scholarpedia, 2(8), 1965.

[48] Dvoretskii, S., Gong, Z., Gupta, A., Parent, J., and Alicea, B. (2020). Braitenberg Vehicles as Developmental Neurosimulation. arXiv, 2003.07689.
Appendix I. Definition of the Cost Function

1. Definition of the cost function.

**Input:** $solution_{1}, \ldots, solution_{N}$

**Output:** $cost$ (Total cost of the solution)

```plaintext
function fitness_function(solution[], dest)
    \( total_{price} = 0, last_{arrival} = 0 \) \# 0:00 time
    \( first_{departure} = 1439 \) \# 23:59 for initialization
    \( flight_{id} = -1 \)
    for \( i \leftarrow 1 \) to \( \frac{\text{length(solution)}}{2} \) do
        \( origin = people[i][1] \)
        \( flight_{id} += 1 \)
        \( going = flights[(\text{origin, dest})][solution[flight_{id}]] \)
        \( flight_{id} += 1 \)
        \( returning = flights[(\text{dest, origin})][solution[flight_{id}]] \)
        \( total_{price} += going[2] \)
        \( total_{price} += returning[2] \)

    if \( last_{arrival} < \text{get_minutes}(going[1]) \) then \# Find last arrival
        \( last_{arrival} = \text{get_minutes}(going[1]) \)
    end if

    if \( first_{departure} < \text{get_minutes}(returning[0]) \) then \# Find first departure
        \( first_{departure} = \text{get_minutes}(returning[0]) \)
    end if
```

15
```plaintext
end for

total\_wait = 0
flight\_id = 1
for i ← 1 to \(\frac{\text{length(solution)}}{2}\) do
    origin = people[i][1]
    flight\_id += 1
    going = flights[(origin, dest)][solution[flight\_id]]
    flight\_id += 1
    returning = flights[(dest, origin)][solution[flight\_id]]

    # Waiting time for all arrived
    total\_wait += last\_arrival - get\_minutes(going[1])

    # Waiting time for all to depart and reach location
    total\_wait += get\_minutes(returning[0]) - first\_departure
end for

# 3PM - 10AM
# 11AM - 3PM
if last\_arrival > first\_departure then
    # Penalize if arrival and departure are not on same days
    total\_price += 50
end if

return total\_price + total\_wait  # The total cost associated
```
Appendix II. Time Complexity Derivations

1. Time Complexity Derivations

Worst Case analysis is assumed, and maximum values a set to $N$

1.1 Cost Function

$S \rightarrow$ Length of initial Solution/Individual

Time Complexity: $T(P) = S/2 + S/2$

$O(T(S)) = S/2$

$\therefore O(N/2) = O(N)$

1.2 OnePoint Mutation Mutation and Crossover

By ignoring the operation of copying N elements $(O(N) \text{ time})$, gene selection then takes $O(1)$, as random.randint uses the Mersenne-Twister algorithm which is $O(1)$.

$\therefore$ Time Complexity is $O(1)$.

1.3 Random Search

$E \rightarrow$ Epochs

$D \rightarrow$ Length of Domain

$S \rightarrow$ Length of initial Population/Solution

Time Complexity:

$T(N) = D + E * (S/2) + (E - 1) * D$; $O(T(N)) = O(ES/2 + E - 1 * D)$

$\therefore O(N^2/2) = O(N^2)$

1.4 Hill Climbing

$E \rightarrow$ Epochs

$D \rightarrow$ Length of Domain

$S \rightarrow$ Length of initial Population/Solution

$n \rightarrow$ Number of neighbouring solutions
\[ T(N) = D + \frac{S}{2} + n \times (S/2); \]
\[ \therefore O(T(N)) = O(N^2) \]

1.5 Standard GA Configuration

\[ P \rightarrow \text{Population Size} \]
\[ G \rightarrow \text{Number of Generations’} \]
\[ D \rightarrow \text{Length of Domain} \]
\[ S \rightarrow \text{Length of Solution/Individual} \]
\[ C \rightarrow \text{Length of array of Costs} \]
\[ P_m \rightarrow \text{Probability of Mutation} \]
\[ P_c \rightarrow \text{Probability of Crossover}, \quad P_c = 1 - P_m \]

\[ T(N) = P \times D + G \times \left( (P + 1) \times \frac{S}{2} + C \log C + P \times (P_m \times 1 + P_c \times 1) \right) \]

this can be simplified to \( O(N) = \frac{N^3}{2} + N^3 \log N, \)
\[ \therefore O(T(N)) = O(N^3 \log N) \]

The choice of sorting function is responsible for \( \log N \). Python’s default \textit{Tim Sort} instead of doing Heapify operations which reduces worst case complexity from \( O(N^3 \log N) \) to \( O(N^3) \).

1.6 GA with Reverse Operations

\[ P \rightarrow \text{Population Size} \]
\[ G \rightarrow \text{Number of Generations} \]
\[ D \rightarrow \text{Length of Domain} \]
\[ S \rightarrow \text{Length of Solution/Individual} \]
\[ C \rightarrow \text{Length of array of Costs} \]
\[ P \rightarrow \text{Probability of Crossover} \]
\[ P_m \rightarrow \text{Probability of Mutation} \]
\[ P_m = 1 - P_c \]

\[ T(N) = P \times D + G \times \left( (P + 1) \times \frac{S}{2} + C \log C + P \times (P_c \times 1 + P_m \times 1) \right) \]

this is of the form
\[ O(N) = \frac{N^3}{2} + N^3 \log N, \]
\[ O(N) = N^3 \log N \]

1.7 GAs with Reversals

\( P \rightarrow \) Population Size \\
\( R \rightarrow \) Number of Reversals; \\
\( G \rightarrow \) Number of Generations \\
\( \text{step}_{\text{length}} \rightarrow \) The number of reverse steps/epochs

The number of reversals is calculated as follows:

\[ R = \frac{G}{n_k} - 1 \]

\[ T_R (N) = C; \quad \text{if step}_{\text{length}} = 1 \ \text{else} \]

\[ T_R (N) = (\text{step}_{\text{length-1}})^* (C + \frac{S}{2} + P * (Pc * 1 + Pm * 1)) \]

Now actual Time Complexity of GA with Reversals is,

\[ O(T(N)) = O(T(P)) + O(T_R(N)) \]

\[ T(N) = P * D + G * ((P + 1) * \frac{S}{2} + ClogC + R * C + P * (Pm * 1 + Pc * 1)); \]

\[ \text{if step}_{\text{length}} = 1 \ \text{else} \]

\[ T(N) = P * D + G * ((P + 1) * \frac{S}{2} + ClogC + R * ((\text{step}_{\text{length-1}})^* (C + \frac{S}{2} + P * (Pc * 1 + Pm * 1) + P * (Pm * 1 + Pc * 1)))) \]

This further reduces to these 2 forms:

\[ O(T(N)) = O(N^3/2 + N^3) = O(3/2N^3) = O(N^3); \text{ if step}_{\text{length}} = 1 \ \text{else} \]

\[ O(T(N)) = O(N^3/2 + N^4) = O(N^4) = O(N^4); \]

1.8 Iterated Chaining

\( Rounds \rightarrow \) The number of iterated Chaining rounds

\[ T(N) = Rounds - 1 * T_{algo_1} (N) + Rounds * T_{algo_2} (N) \]
The authors chose \( algo_1 \) as Random Search and \( algo_2 \) as HillClimbing, thus:

\[
T(N) = \text{Rounds} - 1 \times T_{RS}(N) + \text{Rounds} \times T_{algo_2}(N)
\]

\[
T(N) = \text{Rounds} - 1 \times D + E \times (S/2) + (E - 1) \times D + \text{Rounds} \times D + S/2 + n \times (S/2)
\]

\[
\therefore O(T(N)) = N^3 - 1 + N^3 = O(N^3)
\]

**Appendix III. Iterating Chaining Algorithm**

IC Algorithm

**Algorithm : Iterated Chaining with Early Stopping**

*Input:* Domain \( D_i \ldots \ldots \ D_N \) rounds, fitness_function, \( n_{obs} \) tolerance

*Output:* \( soln_{final}, \text{best}_{cost}, \text{scores}, \text{NFE}, nfe, seed \)

```plaintext
1 scores ← [], NFE ← 0 //Global record of cost and no. of function evaluations
2 for i ← rounds do
3     if i == 0 then
4         soln, cost, scores, nfe, seed ← algorithm_1(domain, fitness_function, seed)
5         soln ← OnePointMutation(domain, random.randint(0, 1), soln)
6         scores ← append cost to list
7         NFE ← nfe+1
8     end if
9     else if i == rounds - 1 then
10        soln_{final}, cost, scores, nfe, seed ← algorithm_2(domain, fitness_function, seed)
11        scores ← append cost to list
12        return soln_{final}, scores [-1], scores, NFE
13        NFE ← nfe+1
14     end if
15     else
16        soln, cost, scores, nfe, seed ← algorithm_1(domain, fitness_function, seed)
17        soln ← OnePointMutation(domain, random.randint(0, 1), soln)
18        scores ← append cost to list
19        NFE ← nfe+1
20     end else
```

20
$\text{soln}_{final}$, cost, scores, nfe, seed $\leftarrow$ algorithm_2(domain, fitness_function, seed)

scores $\leftarrow$ append cost to list

$\text{NFE} \leftarrow \text{nfe} + 1$

if rounds == 1 then

\[ \text{return soln, scores[-1], scores, NFE} \]

end if

if \( \text{cost} - \text{random.randint(tolerance, 100)} > \text{int} (\text{sum(scores[-n_{obs}:])} / n_{obs}) \) then

\[ \text{return soln}_{final}, \text{scores}[-1], \text{scores}, \text{NFE} \]

end if

Where \( \text{scores[-1]} \) is \( \text{best}_{cost} \), i.e the final cost; \( \text{NFE} \) is the global list of costs

**Algorithm: Genetic Algorithm with Reversals**

**Input:** Domain \( D_i \ldots D_N \), mutation \( P_{mutation} \), crossover \( P_{crossover} \), \( n_k \), step length, num generations, fitness_function, \( n_{obs} \)

**Output:** \( \text{soln}_{final}, \text{best}_{cost}, \text{scores}, \text{nfe}, \text{seed} \)

for \( i \leftarrow \text{num generations} \) then

population $\leftarrow$ Initialize population randomly

if \( i/n_k = 0 \) and \( i \neq 0 \) then

if step_length == 1 then

Sort costs list in descending order instead

rev $\leftarrow$ rev+1

end if

else

rev $\leftarrow$ rev+1

while \( i \leftarrow \text{step length} \) do

\[ \text{costs.sort(reverse=True) //Decreasing order of costs} \]

ordered_individuals = [individual for (cost, individual) in costs]

end if

end if
```
12  population ← Get to a list of top $n_{\text{eltism}}$ from ordered_individuals
13  scores ← fitness_function(population[0])
14  nfe ← nfe + 1
15  while length of population list $< n_{\text{population size}}$ do
16    if random.random() $< P_{\text{mutation}}$ then
17      population ← Append result of Crossover of 2 randomly
18      chosen individuals from ordered_individuals
19    end if
20  else
21    population ← Append result of OnePointMutation of a randomly
22    chosen individual from ordered_individuals
23  end else
24  end while
25  end while
26  end if
27  else
28  costs.sort() //Increasing order of costs
29  ordered_individuals = [individual for (cost, individual) in costs]
30  population ← Get to a list of top $n_{\text{eltism}}$ from ordered_individuals
31  scores ← fitness_function(population[0])
32  nfe ← nfe + 1
33  while length of population list $< n_{\text{population size}}$ do
34    if random.random() $< P_{\text{mutation}}$ then
35      population ← Append result of Crossover of 2 randomly
36      chosen individuals from ordered_individuals
37    end if
38  else
39    population ← Append result of OnePointMutation of a randomly
40    chosen individual from ordered_individuals
41  end else
42  end while
43  end else
44  end for
45  return $s_{\text{oln}}$, $s_{\text{final}}$, $s_{\text{best}}$, $s_{\text{cost}}$, scores, nfe, seed
```

**Algorithm: Genetic Algorithm with Random Search Reversals**

**Input:** Domain $D$, ..., $D_N$, $P_{\text{mutation}}$, $P_{\text{crossover}}$, $n_k$, step length, num generations


for $i \leftarrow \text{num generations}$ then

1. population $\leftarrow$ Initialize population randomly
2. if $i/n_k = 0$ and $i \neq 0$ then
3.   if step_length == 1 then
4.     Sort costs list in descending order instead
5.     rev $\leftarrow$ rev + 1
6.   end if
7. else
8.     rev $\leftarrow$ rev + 1
9.     while $i \leftarrow$ step_length do
10.    costs.sort(reverse=True) //Decreasing order of costs
11.    soln $\leftarrow$ Randomly initialize within U.B and L.B of $D$
12.    population $\leftarrow$ Get to a list of top $n_{\text{eltism}}$ from ordered_individuals
13.    scores $\leftarrow$ fitness_function(population [0])
14.    nfe $\leftarrow$ nfe + 1
15.    if cost $>$ best_cost then
16.      : 
17.      best_cost $\leftarrow$ cost
18.      best_solution $\leftarrow$ solution
19.    end if
20.    scores $\leftarrow$ Append best_cost
21.    population $\leftarrow$ Append best_soln
22. end while
23. end else
24. else
25. end if
26. costs.sort() //Increasing order of costs
27. ordered_individuals = [individual for (cost, individual) in costs]
28. population $\leftarrow$ Get to a list of top $n_{\text{eltism}}$ from ordered_individuals
29. scores $\leftarrow$ fitness_function(population [0])
nfe ← nfe + 1

while length of population list < population_size do
    if random.random() < pmutation then
        population ← Append result of Crossover of 2 randomly chosen individuals from ordered_individuals
    end if
else
    population ← Append result of OnePointMutation of a randomly chosen individual from ordered_individuals
end else
end while

end else
end for

return soln_final, best_cost, scores, nfe, seed

Appendix IV. Descriptive Statistics for Different Candidate Algorithms

Run times and descriptive statistics for our experimental conditions (different search strategies) using different candidate algorithms. Runtime measured in milliseconds (ms). Initial parameter values: $num_{\text{reversals}} = 1$ and $step_{\text{length}} = 100$. Abbreviations: SGA = standard GA; GAwRo = GA with Reverse Operations; GAwR = GA with Reversals; HC = Hill Climbing; RS = Random Search; SA = Simulated Annealing; GARSRev = GA with Random Search as Reversals.

Rosenbrock (13 dimensions)

|          | cost | $\sigma$ | $cost_{\text{min}}$ | $cost_{\text{max}}$ | n.f.e | ms  |
|----------|------|----------|---------------------|---------------------|-------|-----|
| SGA      | 134.6| 194.47   | 0                   | 609                 | 1000  | 0.12|
| GAwRo    | 134.4| 186      | 0                   | 518                 | 1000  | 0.13|
| GAwR     | 72.5 | 145.9    | 0                   | 510                 | 1099  | 0.25|
| HC       | 645001| 804257   | 0                   | 3025941             | 1092.6| 0.02|
| RS       | 209620| 90604.6  | 93099               | 354460              | 100   | 0.01|
| SA       | 2131676| 1080542  | 114786              | 3904163             | 512   | 0.01|
| GARSRev  | 122.9| 197.8    | 0                   | 609                 | 1099  | 0.2 |
### Zakharov (13 dimensions)

|          | cost  | σ     | cost_{min} | cost_{max} | n.f.e | ms |
|----------|-------|-------|------------|------------|-------|----|
| SGA      | 84.47 | 26.83 | 40.31      | 159        | 1000  | 0.12 |
| GAwRo    | 86.28 | 36.68 | 33.13      | 150        | 1000  | 0.13 |
| GAwR     | 55.29 | 23.52 | 27.31      | 108        | 1099  | 0.22 |
| HC       | 298.65| 110.16| 65.31      | 505        | 698.5 | 0.02 |
| RS       | 394.11| 385.05| 138.31     | 1644       | 100   | 0.01 |
| SA       | 4.03E+08| 5.42E+08| 758507    | 2.16E+09  | 512   | 0.01 |
| GARSRev  | 58.59 | 32.32 | 9.31       | 136.31     | 1099  | 0.17 |

### Griewank (13 dimensions)

|          | cost  | σ     | cost_{min} | cost_{max} | n.f.e | ms |
|----------|-------|-------|------------|------------|-------|----|
| SGA      | 15.66 | 7.58  | 2.23       | 36.33      | 1000  | 0.13 |
| GAwRo    | 32.07 | 18.53 | 6.88       | 77.19      | 1000  | 0.12 |
| GAwR     | 29.92 | 15.04 | 4.0        | 62.53      | 1099  | 0.20 |
| HC       | 24.00 | 12.36 | 8.26       | 47.49      | 78026 | 0.15 |
| RS       | 178   | 32.96 | 110.82     | 228.66     | 100   | 0.02 |
| SA       | 385.29| 76.41 | 275.12     | 600.31     | 512   | 0.01 |
| GARSRev  | 33.8  | 14.88 | 11.18      | 67.90      | 1099  | 0.16 |

### Matyas 2D

|          | cost  | σ     | cost_{min} | cost_{max} | n.f.e | ms |
|----------|-------|-------|------------|------------|-------|----|
| SGA      | 0     | 0.01  | 0          | 0.04       | 1000  | 0.11 |
| GAwRo    | 0     | 0     | 0          | 0          | 1000  | 0.12 |
| GAwR     | 0     | 0     | 0          | 0          | 1099  | 0.14 |
| HC       | 0.39  | 0.93  | 0          | 3.86       | 44.1  | 0.01 |
|        |    cost |     σ  | cost\_min | cost\_max | n.f.e | ms   |
|--------|---------|--------|-----------|-----------|-------|------|
| **RS** | 0.13    | 0.14   | 0         | 0.5       | 100   | 0.01 |
| **SA** | 29.99   | 24.43  | 3.14      | 90.26     | 512   | 0.01 |
| **GARSRev** | 0     | 0      | 0         | 0         | 1099  | 0.16 |

**Schaffer\_N1 (2 dim)**

|        |    cost |     σ  | cost\_min | cost\_max | n.f.e | ms   |
|--------|---------|--------|-----------|-----------|-------|------|
| **SGA** | 0.15    | 0.13   | 0         | 0.42      | 1000  | 0.09 |
| **GA\_w\_Ro** | 0.14  | 0.11   | 0.02      | 0.42      | 1000  | 0.10 |
| **GA\_w\_R**  | 0.14   | 0.10   | 0.02      | 0.33      | 1099  | 0.13 |
| **HC**   | 0.56    | 0.09   | 0.43      | 0.78      | 9.5   | 0.01 |
| **RS**   | 0.24    | 0.14   | 0         | 0.47      | 100   | 0.01 |
| **SA**   | 1.34    | 0.15   | 0.86      | 1.49      | 512   | 0.01 |
| **GARSRev** | 0.12  | 0.11   | 0.02      | 0.46      | 1099  | 0.12 |