Book Scanning Optimization with Metaheuristics

Igor Diniz[1], Ingrid Diniz[1] and Paula Ito[1]

[1] Artificial Intelligence, MSc Data Science and Engineering, Faculty of Engineering, University of Porto.

Abstract. The book scanning problem is characterized by a set of libraries, books and their constraints, and the objective is to maximize the total score obtained from scanning books within the given time frame. By implementing metaheuristic optimization techniques, including hill climbing, simulated annealing, tabu search, and genetic algorithms, the present work aims to solve the problem providing a friendly user-interface while comparing the performances from all algorithms in terms of score, memory and time consumption. The smaller datasets were easily solved with great score by simpler algorithms, while bigger problems had more benefits with more complex algorithms, although being more computationally expensive. Our results indicate that the choice of the most suitable optimization algorithm depends on the nature of the problem, computation resource constraints, and desired solution quality.

Keywords: book scanning, optimization, hill climbing, simulated annealing, tabu search, genetic algorithm, metaheuristics, artificial intelligence.

1. Introduction

Metaheuristics represent a powerful class of optimization techniques designed to tackle complex problems across diverse domains (Mohamed, 2018). Unlike exact algorithms that guarantee finding the global optimum, metaheuristics prioritize finding a good solution within reasonable timeframes. This makes them particularly suitable for real-world problems where finding the global optimum might be impractical, since they are, in general, NP-complete problems, and thus may not be solved in a polynomial time. In the present work, the authors use metaheuristics to solve the Book Scanning problem.

2. Problem Formulation

The Google Book Scanning problem, part of the Qualification Round of Hash Competition 2020, is characterized by the books and libraries as components, each with its singularity. There are ***B*** different books with IDs from 0 to ***B - 1***. Each book is unique in terms of identification (ID); however, one book can be stored in different libraries. In addition, scanning a book yields a specific score. Regarding the libraries, there are ***L*** different Libraries with IDs from 0 to ***L - 1***. Each library is characterized by the set of books in the library, the signup time required before scanning can commence and the number of books that can be scanned daily once signup is complete.

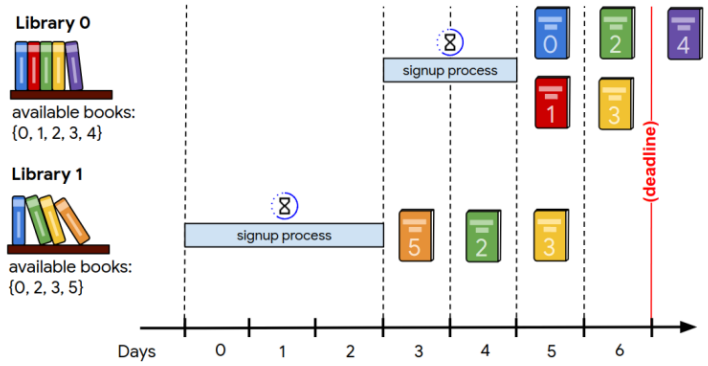

Figure 1. Book Scanning process.

Some important constraints about the problem are related to the time frame and the sign-up process (**Fig. 1**). There are ***D*** days from day 0 to day ***D - 1***. Library signup can commence on day 0, with day ***D - 1*** being the deadline for shipping books. Only one library can undergo signup at any given time due to logistical constraints. Libraries can be signed up in any order. Books from a library can be scanned immediately after the signup process is complete. Therefore, we have the following as constraints:

Constraint 1. Only one library can undergo signup at a time.

Constraint 2. Books from a library can be scanned only after the signup process is complete.

Constraint 3. Each library has a daily limit on the number of books that can be scanned.

Constraint 4. Books must be shipped, scanned, and returned within the specified time frame.

Constraint 5. A book can only be scanned once.

Through an interactive user interface, the present work aims to maximize the total score obtained from scanning books within the given time frame ***D*** with the use of hill climbing,

simulated annealing, tabu search and genetic algorithm, and compare the performance among them regarding total score, time consuming, and memory usage.

3. Methodology

3.1. Data

The dataset for the Google Hash Code book scanning optimization project consists of five files containing essential information. Each file includes the number of available books (***B***), the number of existing libraries (***L***), and the total number of days (***D***) available for scanning. Additionally, there is a list of scores associated with each book and details about each library, such as the number of books it contains, the signup time required, and the scanning rate (or checkout rate). Each library is also accompanied by a list of book IDs it possesses. These data form the basis for creating efficient strategies for book scanning, aiming to maximize the total score achieved within the available time.

Table 1. Datasets information: number of books, libraries, days, and file size. The files *a_example_2* to *a_example_5* were generated by the authors.

Dataset	Books	Libraries	Days	File size (KB)
a_example	6	2	7	1
a_example_2	50	10	7	1
a_example_3	100	50	10	2
a_example_4	1000	200	20	13
a_example_5	500	100	10	5
b_read_on	100000	100	1000	967
c_incunabula	100000	10000	100000	1382
d_tough_choices	78600	30000	30001	1731
e_so_many_books	100000	1000	200	3188
f_libraries_of_the_world	100000	1000	700	3318

While developing our application, different files structured similarly to the originals provided were generated due to computational constraints within the provided data, enabling the investigators to analyse our results within the time constraints of this project. The datasets had different metrics distribution (**Figs. A1 - A10**), different sizes, and different number of books, libraries, and days (**Table 1**).

3.2. Implementation details

The implementation was done entirely in Python using object-oriented programming. Book, library, Solution, and Solver classes have been created, of which the latter is a parent of inheriting classes for each metaheuristic algorithm created.

Furthermore, each algorithm was provided with the choice to set a timeout, which determines when the algorithm will stop iterating after a predefined period. Once the timeout is reached, the algorithm yields the best solution discovered up to that point.

3.3. Solution structure and operators

The solution is composed by both a list of libraries and a list of books objects. The books contain the library identification to indicate from which library it was scanned. Whenever an operator to generate a neighbour is applied on a solution, the general approach was to change the solution's list of libraries, then get the best books for each library without repeating books already scanned.

The operators developed to get new neighbours in the present work were classified into internal (*internal swap* and *deletion)* and external neighbours (*external swap*). As the names suggest, the *internal swap* operator switches two libraries' positions within the list of libraries, while *deletion* removes one library from the list. Besides that, *external swap* switch one library in the current solution to another one that is not present.

3.4. Metaheuristics algorithms

3.4.1. Hill climbing

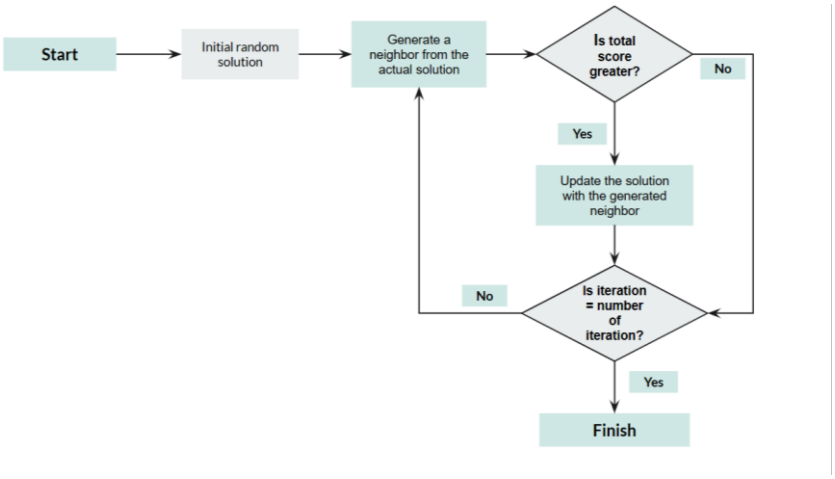

Figure 2. Hill Climbing Schema.

The Hill Climbing algorithm (HC) is a heuristic technique that "select any local change that improves the current solution value of the objective function" (B Selman, CP Gomes, 2006) and this process ends when the local optimal solution is found or another stopping criteria is met. In optimizing book scanning for the Google Hash Code competition, the Hill Climbing algorithm can be applied as follows: initially, an initial solution is generated randomly or through some simple greedy heuristic. Then, the algorithm generates a neighbor by making small changes, such as swapping or deleting libraries. After each modification, the algorithm evaluates the impact of the change on the total score of scanned books. If the score increases, the modification is kept; otherwise, the best solution maintains the last one. This process is repeated until a pre-set maximum number of iterations is reached. Ultimately, the algorithm returns the solution that maximizes the total score of books, representing the local optimal signup order for libraries and the selection of the most valuable books for scanning (**Fig. 2**).

3.4.2. Simulated Annealing

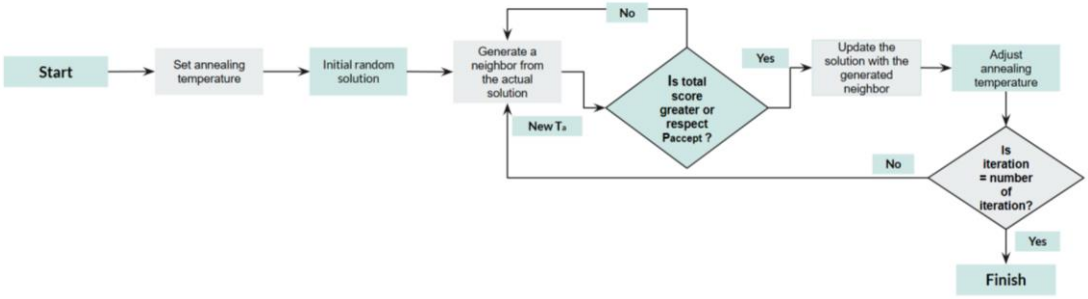

Figure 3. Simulated Annealing Schema.

The Simulated Annealing algorithm (SA) is a "probabilistic optimization technique inspired by the annealing process in metallurgy. The core concept adopted from metallurgy and applied to the algorithmic context is the notion of accepting poorer quality solutions in the early stages of the search and gradually tightening the acceptance criteria as the search progresses" (E Aarts et al., 2005).

In the context of optimizing book scanning for the Google Hash Code competition, Simulated Annealing can be applied as follows: an initial solution is generated randomly or through some simple greedy heuristic. Then, the algorithm iteratively explores neighbor solutions by making small changes, like HC. After each modification, the algorithm evaluates the impact of the change on the total score of scanned books. If the new score is greater or respects a *probabilistic acceptance* (**Expression 1**) the modification is kept; otherwise, the best solution maintains the last one. As the algorithm progresses, the temperature parameter is gradually reduced, leading to a more deterministic search towards the end (**Fig. 3**).

The probability of accepting a worse solution, is calculated using:

$$P_{accept} = exp\left(\frac{-\Delta}{T}\right) \quad (1)$$

Here, the exponential function models the probability of acceptance, with Δ representing the change in the objective function value and T_a representing the current temperature. As the temperature decreases, P_{accept} decreases, making it less likely to accept worse solutions. The exponential function is used to ensure that the acceptance probability decreases gradually as the difference in objective function values increases or as the temperature decreases. To decide whether to accept the worst solution, we compare the acceptance probability P_{accept} with a randomly generated number r between 0 and 1.

3.4.3. Tabu Search

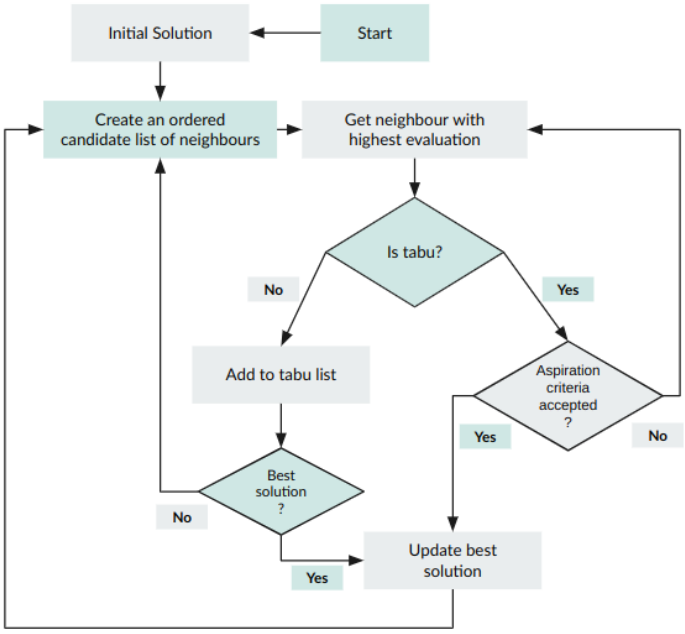

Figure 4. Tabu Search Schema.

Tabu Search (TS) is a heuristic procedure for solving optimization problems, designed to guide other methods (or their component processes) to escape the trap of local optimality (Glover, 1990). The most basic implementation of this algorithm maintains a *short-term memory*, namely the *tabu list*, to keep track of recently visited solutions (or attributes of the solutions) and prevents the algorithm from revisiting them in subsequent iterations, promoting diversification in the search process and avoiding getting stuck in a local optimal. More robust versions of Tabu Search also implement *frequency based long term memory* used to penalize elements of the solution that have appeared in many other solutions visited, and then promoting *diversification* and *intensification* on the search process.

To address the Book Scanning problem, both memories have been used and the swaps between two libraries (internal and external) have been used as *attributes of the tabu criterion* in the tabu list. Other than that, a *neighborhood reduction* was necessary, since generating all neighbors for the problem sizes presented by Google was inappropriate in terms of time. Therefore, the number of neighbors to be generated - *the candidate list size* - for each iteration, was a user-defined parameter along with the tabu tenure.

Regarding the *frequency-based long-term memory*, the implementation to this work was *transition-based*, storing the number of times two libraries had been already switched. Consequently, a penalty is introduced during the search process by subtracting this number

from the neighbor's score that is generated by this swapping, after the best solution remains the same during the chosen tabu tenure iterations. The introduction of this spreads the search, by actively prioritizing moves that give solutions with new composition of attributes improving the diversification of the search space and avoiding getting stuck in the solution's cycles.

3.4.4. Genetic Algorithms

A genetic algorithm (GA) is a model inspired by the biological evolution processes, such as natural selection and genetic evolution (Forrest, 1996). It operates by evolving a population of candidate solutions, referred as individuals or chromosomes, over successive generations through the application of genetic operators. In the present work, a steady-state GA was implemented, meaning that the least fit individuals are replaced in the population after each generation.

The first step is to generate an initial population (**Fig. 5**). In the present work, the initial population was generated 50% randomly, and 50% originated by variations of a greedy solution. Then, the parents are selected through tournament and roulette. This step is important, because it defines which individuals will generate offspring for the next generation. In the tournament, four randomly chosen individuals are selected from the population, and the one with highest score is selected. In the roulette wheel, one individual is chosen among all in the population based on its fitness: the highest score an individual holds, the highest probability it has of being chosen.

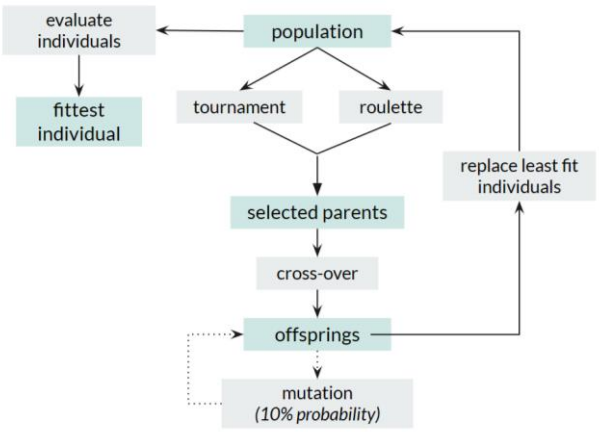

Figure 5. Genetic Algorithms Schema.

Once the parents are selected, crossover creates the offsprings. The crossover combines the genetic material between two individuals by exchanging subparts of their chromosomes, roughly mimicking biological recombination (Mitchell, 1995). Two techniques were implemented: mid-point crossover, which changes the genetic material based on the middle position, and random point crossover, in which the position for switching materials is randomly assigned. Based on a 10% probability of happening, the offsprings generated from the crossover may suffer a mutation procedure. As for mutation methods, the present work considered *internal swap*, *external swap* or *deletion* (explained in section 3.3).

Every time a population is generated, the algorithm keeps in storage the fittest individual encountered throughout the generations. After all generations, the fittest individual among all iterations is obtained.

3.5. User interface

The user interface was built using *tkinter* library in Python. In the menu created, two main options are available to be chosen: "Solve a problem with a chosen optimization algorithm" and "Compare algorithms performance", allowing the user to choose between running individual algorithms or comparing algorithms' results and performance through plots and table. After running each individual algorithm, the user was given the option to plot the results.

It was developed to facilitate the execution of each algorithm with its own user-defined parameters, besides providing graphical plots of the score evolution across each iteration allowing to see the effort of the chosen algorithm by the user to escape from local optimal. Additionally, a comparative analysis feature incorporated into the interface enabled simultaneous assessment of all algorithms implemented for the same file problem and same initial solution (in case of HC, SA and TS). Overall, the interface streamlines the evaluation process, enhancing the efficiency and effectiveness of algorithm selection for the book scanning problem.

3.6. Performance metrics

3.6.1. Computational efforts: time and memory

For each algorithm, we kept of the computational efforts to execute. With analysis of memory usage and time to perform, it became possible to understand which algorithm and its parameters are more computationally expensive, besides being able to improve code efficiency.

3.6.2. Evaluation function

The evaluation function (or fitness function for GA) used in the present work was the total sum of book scores found in the solution. In other words, it is the sum of all books scanned for all libraries in the solution.

4. Results

The algorithms differed in performance related to the score obtained, memory consumed and time to process for each file problem. For HC, SA and TS, the algorithms were run twice per file – once from a greedy and once from a random initial solution. For GA, the algorithm was run once per file, since the population had fixed proportion of random and greedy-originated individuals.

The HC algorithm was run with a fixed parameter of 1000 iterations twice for each file instance - one for random initial solutions and one for greedy. However, for data where it took a long time to reach the solution, a time limit of 1 hour was added. As predicted, the results showed a progressive improvement of the solution throughout the iterations, affirming the expected performance of the algorithm. It exhibits behavior where it refrains from accepting lower values than the existing book, occasionally resulting in the solution stagnating at a local optimum (**Table 2**).

As the algorithm does not store any values, low memory consumption is noted, with a gradual increase depending on the data. The same is observed for time.

Table 2. Hill Climbing Performance.

File	Score		Memory (MB)		Time (s)	
	Random	Greedy	Random	Greedy	Random	Greedy
a_example	21	21	0.63	0.03	2.29	1.57
a_example_2	173	288	0.16	0.07	1.85	3.76
a_example_3	475	849	0.18	0.3	0.56	16.46
a_example_4	2170	6066	1.15	1.43	77.20	156.03
a_example_5	633	2993	0.39	0.44	8.20	11.26
b_read_on	5008800	5.82	522.69	216.41	3600.71	9210.313
c_incunabula	975204	4941042	104.21	99.62	5208.53	21152.41
d_tough_choices	4346225	4815525	241.02	263.71	3616.41	3607.755
e_so_many_books	1904856	4888825	222.88	242.83	3601.66	3607.20
f_libraries_of_the_world	1149559	4946061	80.95	96.77	16436.72	3607.45

The initial parameters established for SA were maximum iterations of 100, 0.99 cooling rate and temperature of 1000, and it was run twice for each file instance - one for random initial solutions and one for greedy.

In the results obtained from the SA algorithm (**Table 3**), we noticed a tendency to accept worse scores as solution, unlike HC. Consequently, the algorithm can bypass local optima, potentially reaching a global optimum without incurring significant computational cost, as it does not require storing values to generate a neighbouring solution. The memory and time values grow gradually, confirming the size of the dataset, but are not significant.

It is also important to note that at initial temperatures the SA is more likely to accept worse solutions, and at final temperatures it is more difficult due to the implemented acceptance parameter P_{accept}.

Table 3. Simulated Annealing performance.

File	Score		Memory (MB)		Time (s)	
	Random	Greedy	Random	Greedy	Random	Greedy
a_example	21	21	0.14	0.12	0.15	0.09
a_example_2	173	288	0.15	0.17	1.38	1.94
a_example_3	536	862	0.26	0.28	4.50	5.92
a_example_4	264	6178	0.88	1.01	24.69	34.90
a_example_5	633	3082	0.41	0.46	7.97	11.20
b_read_on	464220	5828100	125.04	131.93	758.13	796.67
c_incunabula	958032	4941239	83.39	73.77	561.97	1113.42
d_tough_choices	4348890	4815395	245.51	159.24	3821.77	3614.25
e_so_many_books	1844505	4877499	201.44	159.97	3601.4	2693.13
f_libraries_of_the_world	1109563	4946061	81.91	72.24	1060.68	295.71

The TS algorithm was run twice for each file instance - one for random initial solutions and one for greedy - by using tabu tenure equal to 9 and different candidate list sizes but fixing the maximum number of iterations to 100, since it is the most time-consuming algorithm in this work. For the smallest instances, named by *a_example_X.in*, the TS metaheuristic was the one with the greatest performance in terms of scores reached for the best solutions, although always being the worst regarding time consumed, for both random and greedy initial solutions.

In relation to the larger files provided by Google, a timeout of 2 or 3 hours needed to be introduced as a *stop criterion* in the algorithm's implementation to meet the deadline to deliver the work, because it was taking even more than that to reach only 100 iterations. Thus, as shown **in Fig. A13** in the Appendices section (8*)* the algorithm could optimize more and more the solutions described in **Table 4** if more time could be spent in this work. Consequently, different number of iterations was achieved by this timeout, from 20 to 70, implying different qualities of final scores – worse for smaller iterations and better for the bigger ones. To address these files instances, tabu tenure equal to 9 have been used and candidate list size equal to 100.

Table 4. Tabu Search performance.

File	Score		Memory (MB)		Time (s)	
	Random	Greedy	Random	Greedy	Random	Greedy
a_example	21	21	0.025	0.026	0.024	0.029
a_example_2	248	288	0.23	0.29	4.22	12.97
a_example_3	536	901	1.83	1.27	66.04	76.07
a_example_4	2188	6225	4.14	5.03	227.16	274.19
a_example_5	2914	3268	3.56	1.38	136.19	184.19
b_read_on	5371300	5822900	8504.26	3085.88	7591.41	11020.6
c_incunabula	1051792	4974961	1498.79	674.39	7295.83	15073.4
d_tough_choices	4347655	4815785	1703.70	3546.85	7255.27	11086.6
e_so_many_books	2009769	4882134	450.32	420.58	7223.79	11217.3
f_libraries_of_the_world	1579790	5031953	1012.23	2017.62	8055.19	11765.9

The GA was run with different population and generation parameters depending on the file, but all runs had random mode for both mutation and crossover. For the small datasets *a_example* and *a_example_2*, the population size was set to 5 and number of generations was set to 10. As for the remaining datasets, the algorithms ran with the population size equals to 100, and number of generations as 50.

The fittest individuals were found in the first generation for most of the problems, except for file *e* and *f*, where the best solutions were in the second generation, and file *a_example_3*, where the fittest individual was discovered in the 28th. Although the early finding

of the best solution for most datasets, the least fit individuals were clearly being removed from the population as it evolved, showing great functionality for population evolution (e.g., *f_libraries_in_the_world*, **Fig. A12** and *a_example_3*, **Fig. A11**).

Table 5. Genetic Algorithm performance.

File	Score	Memory (MB)	Time (s)
a_example	21	0.16	0.15
a_example_2	288	0.24	0.27
a_example_3	823	5.59	29.39
a_example_4	5913	24.98	143.58
a_example_5	2914	5.59	33.79
b_read_on	5825600	8504.26	5001.98
c_incunabula	4940693	1734.86	3638.86
d_tough_choices	4815395	2975.56	9926.16
e_so_many_books	4874476	3790.3	3809.47
f_libraries_of_the_world	4945834	963.08	3633.48

As expected, it was more computationally expensive to run the files with bigger size (**Table 5**). The memory is a fundamental resource for GAs, since the entire population must be kept in storage after each iteration. Its consumption not only relies on the population size, but also on the individual size, especially because the individual (i.e. solution) for the Book Scanning problem can be very large. Depending on the size of the population, it also led to higher time to find the fittest individual among all.

When comparing computational efforts from all algorithms, they differed in terms of memory, time and score (e.g., Fig**. A13**). It is noticeable that GA is more expensive when it comes to memory: most of the runs had larger consumption than the other algorithms. This is likely a result from the constraints mentioned above. Furthermore, TS also has significant memory consumption because of the tabu list, even though not significantly as GA because it is not storing the whole solutions, but attributes from solutions as mentioned before in section 3.4.3.

Regarding time-consuming factors, TS stood out as the most demanding algorithm, probably due to its iterative nature and the need to explore a large solution space and examine a large set of neighbouring solutions in each iteration. This exploration ensures it doesn't get stuck in local optima, but it also requires significant time consumption, especially for complex problems with many possible neighbours (Pirim et al., 2008).

Concerning the scores reached by the algorithms, they were very similar – when not equal – for files *a_example* and *a_example_2*, which are the smaller ones. For HC, SA and TS,

the algorithms performed much better when initiating from a greedy solution, as it is a good starting within the search space. However, TS was able to find better scores in general when comparing to other algorithms.

Based on the performance results presented, the HC and SA algorithms perform better for small datasets, since these two algorithms do not store any information. When comparing HC and SA, HC achieved faster execution times compared to SA. However, SA achieved better score results in some cases, especially in larger input files, albeit at the expense of significantly longer execution times. In terms of memory usage, both algorithms demonstrated similar behaviour. Therefore, the choice between the two algorithms for small datasets depends on project priorities, with HC being faster but SA potentially producing higher-quality solutions in certain scenarios.

5. Conclusion

Our analysis of optimization algorithms for the book scanning problem has revealed that each approach offers specific advantages depending on the time constraints and desired solution quality. In scenarios where a solution is needed in a very short time and suboptimal results are acceptable, such as files *a_example*, HC is a viable choice. It can provide a reasonable solution in minimal time, which is useful in high-pressure situations.

On the other hand, when there is a bit more time available and a better solution than HC is desired within an acceptable timeframe, SA is a more suitable option. Its ability to accept unfavourable moves with a certain probability allows for a more comprehensive exploration of the solution space, potentially achieving higher-quality solutions.

For problems where solution quality is a priority, and more time can be allocated to finding the optimal solution, TS and GA stood out. Although they require more computational time, these algorithms can find high-quality solutions that significantly optimize book scanning in libraries. Furthermore, as TS performed overall better than GA, it revealed the necessity of providing greater parameters for the latter to perform its best potential.

Therefore, the choice of the most suitable optimization algorithm depends on the nature of the problem, time constraints, and desired solution quality. By considering these factors, we can select the most effective approach to solving the book scanning problem efficiently and accurately.

6. Future work

As future work, the authors suggest implementing a hyperparameter tuning method, such as grid search, to find the best parameters combination that results in the best solution. Furthermore, it would also be advantageous for GA to make use of heuristic search for creating the initial population (as suggested by Syberfeldt et al., 2009), which could improve the performance.

7. Acknowledgments

The authors would like to thank Professor Luís Paulo Gonçalves dos Reis for the Artificial Intelligence lessons, and for all the support that was fundamental to the development of this project.

8. References

Abdel-Basset, Mohamed (2018). Metaheuristic Algorithms: A Comprehensive Review. *Computational Intelligence for Multimedia Big Data on the Cloud with Engineering Applications*, 185–231. doi:10.1016/B978-0-12-813314-9.00010-4

Aarts, E., Korst, J., Michiels, W (2005). Search Methodologies. Retrieved from: Simulated Annealing | SpringerLink

Forrest, S. (1996). Genetic algorithms. *ACM computing surveys* (CSUR), 28(1), 77-80.

Glover, Fred. (1990). Tabu Search: A Tutorial. Interfaces 20(4):74-94.doi.org/10.1287/inte.20.4.74

John, K. (2020, October 30). How we achieved 98.27% of the best score at Google HashCode 2020. [Medium]. Retrieved from: https://medium.com/@kevinjohn1999/how-we-achieved-a-98-27-of-the-best-score-at-google-hashcode-2020-94314a904190.

Mitchell, M. (1995, September). Genetic algorithms: An overview. *In Complex*. (Vol. 1, No. 1, pp. 31-39).

Pirim, Harun & Bayraktar, Engin & Eksioglu, Burak. (2008). Tabu Search: A Comparative Study. 10.5772/5637.

Selman, B. & Gomes, CP. (2006). Hill-climbing search. Retrieved from: Paginated-E 1..104 (cornell.edu)

Soares, A., & Costa, J. P. (2013). Principled Modelling of the Google Hash Code Problems for Meta-Heuristics [University of Coimbra]. Retrieved from: https://estudogeral.uc.pt/retrieve/265403/tese_final_pedro_rodrigues.pdf.

Syberfeldt, A., Persson, L. (2009) Using Heuristic Search for Initiating the Genetic Population in Simulation-Based Optimization of Vehicle Routing Problems. In: Proceedings of Industrial Simulation Conference EUROSIS-ET

9. Appendices

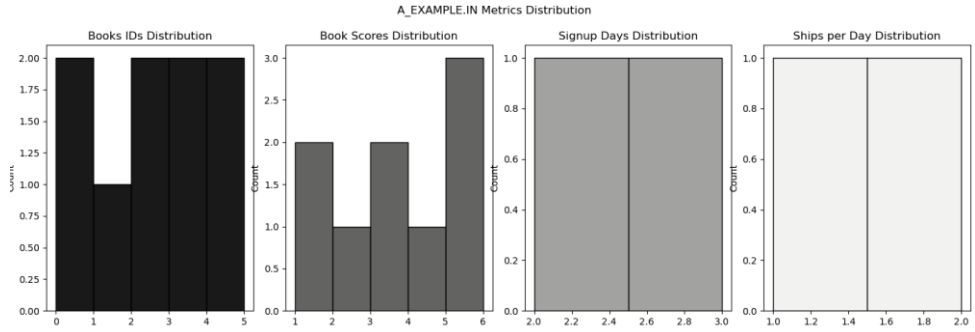

Figure A1. Metrics distribution from file problem a_example.

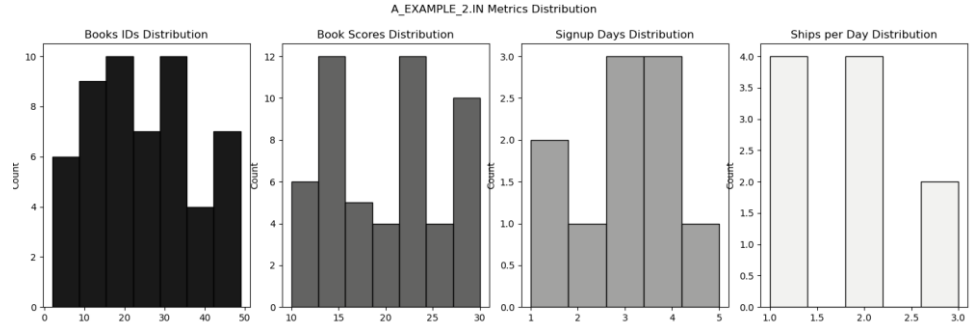

Figure A2. Metrics distribution from file problem a_example_2.

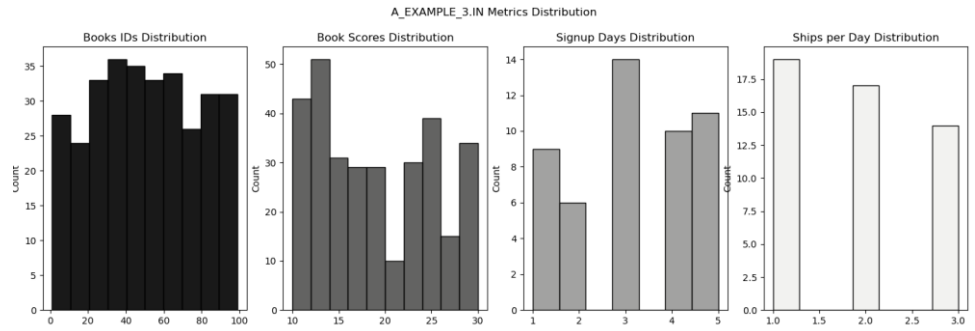

Figure A3. Metrics distribution from file problem a_example_3.

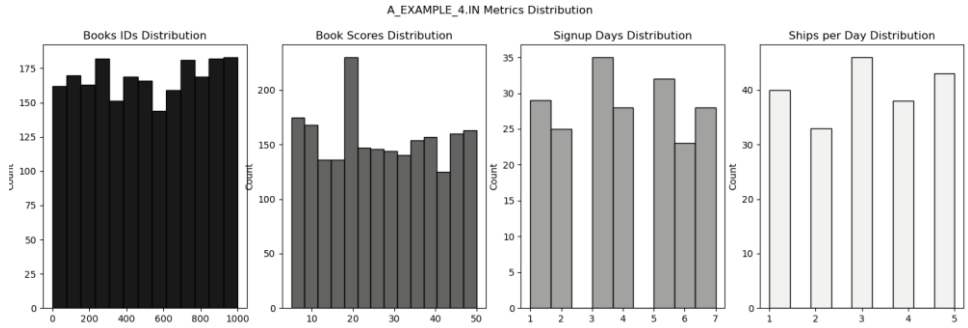

Figure A4. Metrics distribution from file problem a_example_4.

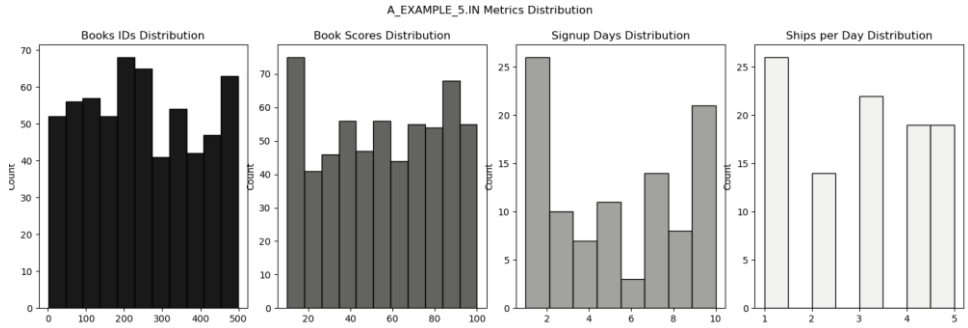

Figure A5. Metrics distribution from file problem a_example_5.

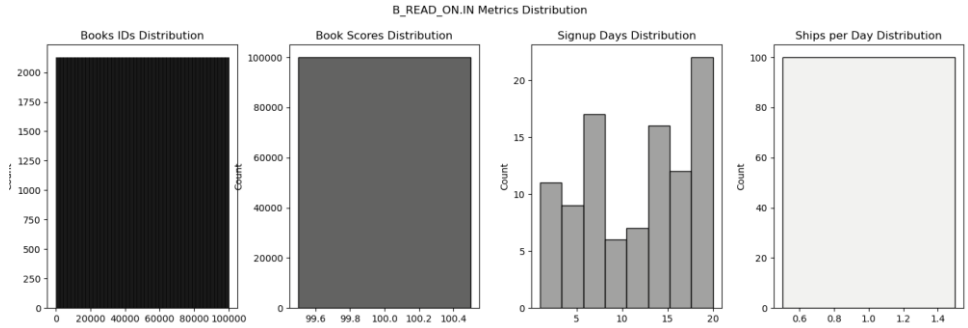

Figure A6. Metrics distribution from file problem b_read_on.

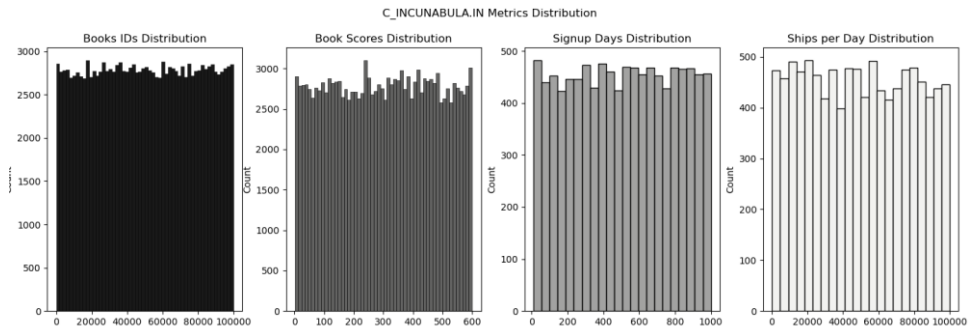

Figure A7. Metrics distribution from file problem c_incunabula.

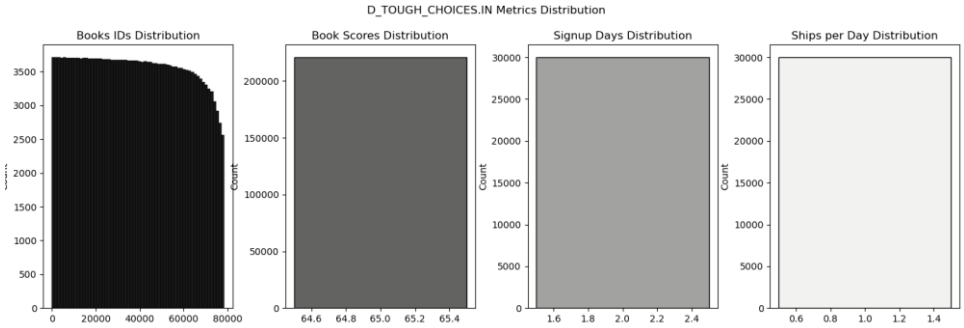

Figure A8. Metrics distribution from file problem d_tough_choices.

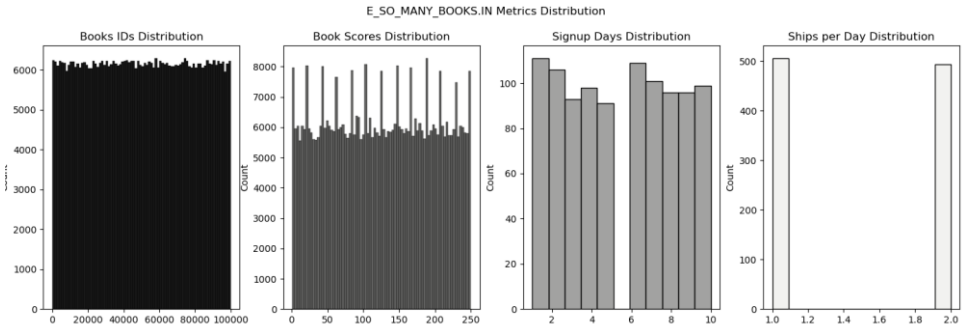

Table A9. Metrics distribution from file problem e_so_many_books.

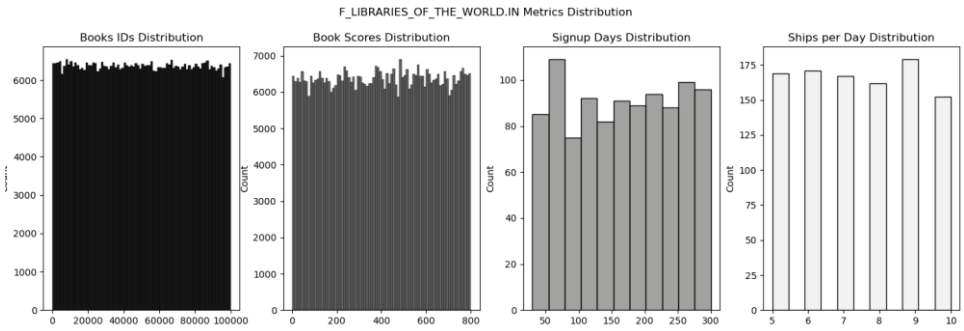

Figure A10. Metrics distribution from file problem f_libraries_in_the_world.

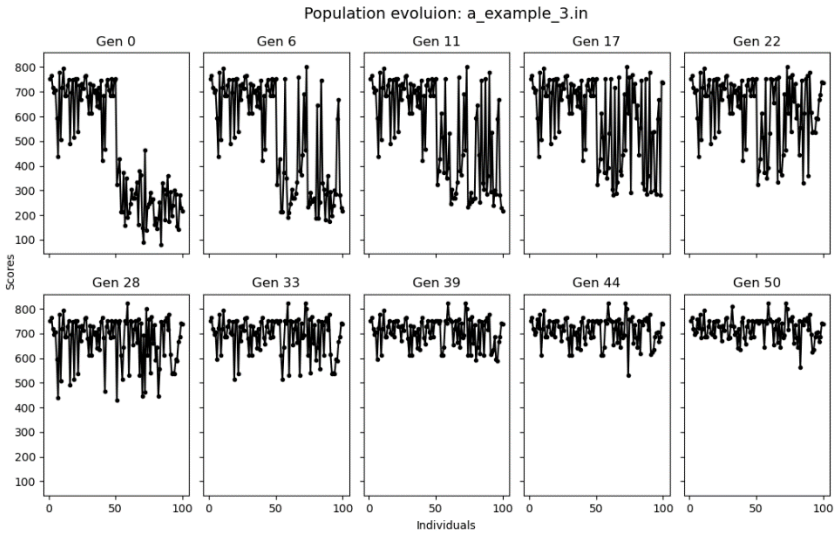

Figure A11. Population evolution for problem file a_example_3.

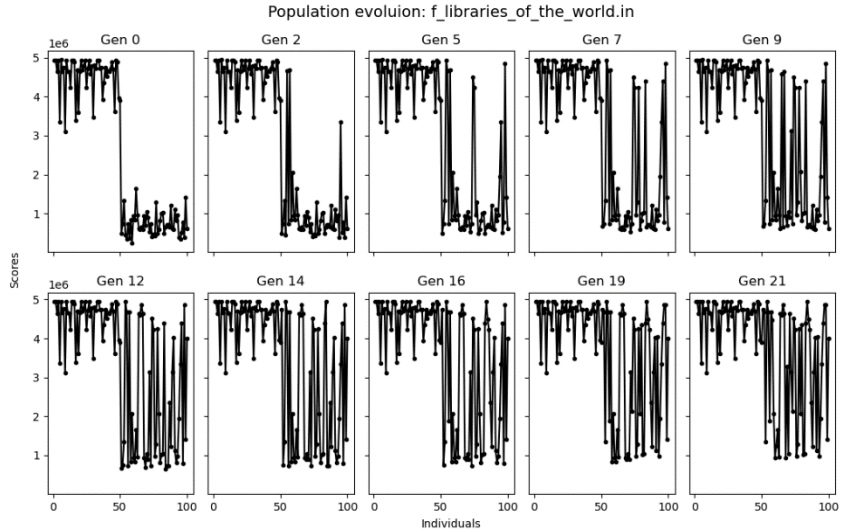

Figure A12. Population evolution for problem file f_libraries_in_the_world.

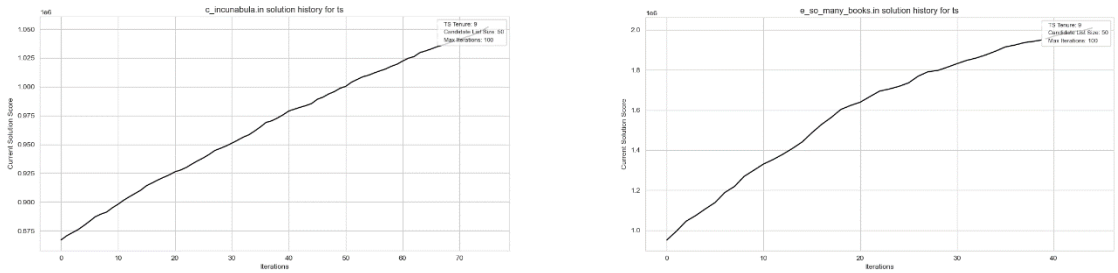

Figure A13. TS solution's score evolution for larger datasets.

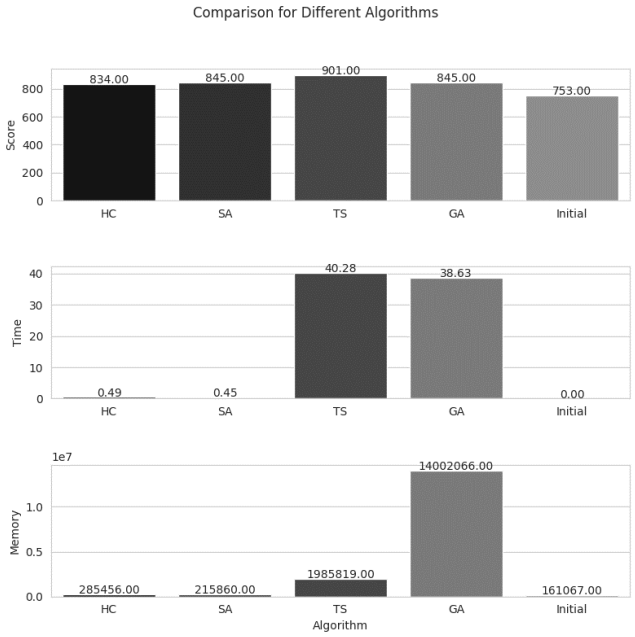

Figure A14. Comparison among metaheuristics for a_example_3.in file instance.