Comparative Analysis of Random Optimization Algorithms for Discrete Problems and Neural Network Weight Optimization

1st Hao Liang
(OMSCS) Georgia Institute of Technology
Georgia Institute of Technology
Markham, Canada
hliang82@gatech.edu

***Abstract*—Optimization algorithms are essential tools in solving complex computational problems across various domains, such as logistics, network design, machine learning, and artificial intelligence. This study explores and compares the performance of four optimization algorithms: Randomized Hill Climbing (RHC), Simulated Annealing (SA), Genetic Algorithm (GA), and Mutual-Information-Maximizing Input Clustering (MIMIC). The evaluation is conducted in two parts: the first part focuses on solving predefined optimization problems (the Traveling Salesman Problem (TSP) and the Knapsack Problem (KP)). The second part applies these algorithms to neural network weight optimization. The results demonstrate the strengths and limitations of each algorithm in terms of performance metrics such as fitness values, function evaluations, and computational time. The second part conducted a cross comparison between the baseline (back propagation) with the three optimization algorithms discussed in part1. The results also demonstrate the pros/cons of the algorithm in terms of weight optimization performance.**

***Index Terms*—Randomized Optimization, Hyper parameter Tuning, Weight Optimization, NN, SA, GA, RHC, MIMIC**

I. Introduction

Optimization algorithms play a crucial role in solving complex computational problems across various domains, from logistics and network design to machine learning and artificial intelligence. [1] This paper investigates the performance of four randomized optimization algorithms: Randomized Hill Climbing (RHC), Simulated Annealing (SA), Genetic Algorithm (GA), and Mutual-Information-Maximizing Input Clustering (MIMIC). Two optimization problems (the Traveling Salesman Problem (TSP) and the Knapsack Problem (KP)) have been selected as a performance baseline.

The study is divided into two parts. Part 1 examines the effectiveness of each algorithm in solving TSP and KP. Its target is to compare their performance in two dimensions: the ability to find optimal solutions, and the efficiency in finding the best solution. The comparision is conducted under two complexity levels (simple and complex) as a function test and pressure test for deeper understanding. This part involves detailed analysis and comparison based on fitness values, function evaluations (FEvals), and computational time. Part 2 extends the study by applying these algorithms in neural network weight optimization process. Back propagation, the most commonly used algorithm in this process, has been introduced as a baseline, to be compared with the optimization algorithm performance.

Based on the knowledge, we assume SA performs the best for TSP and GA performs the best for KP. And GA will be the best algorithm to be applied in NN weight optimization. However, the primary goal of this research is to identify the relative strengths and weaknesses of each algorithm in two optimization contexts. And introduce a method that can be applied in many other optimization contexts for studies in even more optimization algorithms.

II. Methodology - Algorithms

A. Randomized Hill Climbing (RHC)

Randomized Hill Climbing (RHC) is an optimization algorithm that iteratively attempts to improve a solution by making small, random changes. Derived from Hill Climbing (HC), it is often used in optimization problems where the search space is discrete [2]. Hill Climbing (HC) is a search technique that seeks to find the best solution by incrementally improving the current state based on a evaluation function [3]. The primary difference between HC and RHC is the concept of randomness. It helps to avoid to getting stuck in local optima as it will randomly start from different states and compare the optimum states.

B. Simulated Annealing (SA)

Simulated Annealing (SA) is an optimization algorithm inspired by the annealing process in metallurgy. Its objective is finding the global optimum of a given function in a large search space. [4] Unlike RHC and HC, SA can accept worse solutions, when a high temperature is set, to escape local optima. Moreover, it will only accept better solutions when the temperature cools down, which acts like a RHC approach. The ability to switch between the random walk and RHC mode makes it more robust in complex search bases.

C. Genetic Algorithm (GA)

The Genetic Algorithm (GA) is a type of optimization algorithms inspired by the process of natural selection. The abstract version of natural selection process involves selection, crossover, mutation and replacement. [5] It starts at a randomly selected initial population, going through iterations of derived natural selection process to create generations of population till a optima is reached. They are particularly useful for problems with large, complex search spaces.

D. Mutual-Information-Maximizing Input Clustering (MIMIC)

Mutual-Information-Maximizing Input Clustering (MIMIC) is an optimization algorithm that leverages probabilistic models to effectively search for optimal solutions. Unlike the introduced algorithms, MIMIC builds a probabilistic model of the best solutions found and uses this model to generate new possible solutions. Similar to GA, the key idea is to maximize the mutual information between variables, which can be gained from the best populations seen so far. By refining the probabilistic model and the candidate solutions, MIMIC performs well when dealing with complex and high-dimensional search spaces. This approach is particularly useful for problems where the relationships between variables are non-trivial and difficult to model using traditional techniques. [6]

III. Methodology - Framework

I studied the randomized optimization in two parts. For part 1, it focus on optimization performance in solving pre-defined optimization problems. And for part 2, it focus on the the optimization performance in tuning weight distribution in a Neural Network Model.

A. Part 1

Other than the four algorithms been used for study. The methodology involves a framework that define, tune and analize the optimization problems and algorithms. To ensure the exeprinment is consistent and reproducible, a Python virtual environment was configured with all necessary packages. The primary packages used include **mlrose** for implementing optimization algorithms. Below is a list of key sections included in part 1 framework:

1) Algorithm Implementation and Problem Definition: Four randomized optimization algorithms were implemented: **Genetic Algorithm (GA)**, **Mutual-Information-Maximizing Input Clustering (MIMIC)**, **Randomized Hill Climbing (RHC)**, and **Simulated Annealing (SA)**. These algorithms were executed using dedicated scripts (e.g., *ga.py*, *mimic.py*, *rhc.py*, *sa.py*).

2) Hyperparameter Tuning and Data Collection: Hyperparameter tuning was conducted for all algorithms especially **GA** and **MIMIC**, where one parameter was fixed while the other was varied. This approach contributed to evaluating the algorithms' performance under different configurations. Results were stored in CSV format and visualized using shared plotting functions, facilitating easy analysis and comparison. The *iteration* are collected as x-axis values for cross validation, where *time*, *feval* and *fitness* are collected as the y-axis values.

3) Results and Analysis: The results, including performance metrics and visualizations, were organized in the *optimization_results* directory, categorized by algorithm and problem type. Two level of problem types have been included (**simple** and **complex**) for better analysis of algorithms' performance.

B. Part 2

To further understand the application of optimization algorithms in neural network training, I extended the study by integrating the findings from Part 1 into Part 2. The process involved configuring a neural network framework, tuning hyperparameters, and evaluating model performance using various optimization algorithms. Same key steps and sections are included as the frameworks are high modular:

1) Data Preprocessing and Neural Network Configuration: The neural network models were configured using the 'pyperch' library, which allows easy switching between traditional backpropagation and optimization-based weight-tuning methods. Data was preprocessed (for example the boolean / category data has been converted to float values) to ensure compatibility with the neural network models.

2) Hyperparameter Tuning and Model Training: Grid search has been applied for hyperparameter tuning. I has been conducted for each optimization algorithm (GA, RHC, SA) and the default backpropagation method to find the best hyper parameters. The optimal parameters were then used to train the neural network models.

3) Performance Evaluation and Visualization: Performance metrics (accuracy and loss) were recorded during training and validation process. These metrics were plotted to visualize the learning curves, allowing for a detailed comparison of how each optimization algorithm impacts the neural network training process. The results were organized into the 'results' directory.

IV. Experiments

A. Problems

To better achieve the goal of this experinment, to compare the performance between the optimization algorithms we are interested in, I have chosen two discrete optimization problem so the

1) Traveling Salesman Problem (TSP): The Traveling Salesman Problem (TSP) is a classic optimization problem. An imaginary salesman has to visit all the cities exactly once in a sea, and returned to the original city [7]. The distances between the cities are known and the objective is to find the minimum total distances of the route. TSP has significant implications in logistics, route planning, and network design.

2) Knapsack Problem (KP): The Knapsack Problem (KP) is another well-known optimization problem. The problem asks a player to put treasures into a knapsack. Each treasure has a static value and weight, and the backpack has a maximum limit of weight it can carry. The objective is to find the most valuable combination of items that fits within the limit. It is an abstracted problem that can be applicable in many real-life applications such as: resource allocation, budgeting and decision making. [8]

B. Datasets

The **Adult-Census-Income** is a data sets derived from 1994 Census bureau databases. It includes the key information typically used to describe an individual's social economic status, such as age, income and working hours. These parameters are commonly collected by financial institutions to assess a person's affordability when applying credit loans. The dataset includes 14 input variables and 1 output variables which is whether a person's income exceed $50k or not. It has been chosen as grid search has not been applied in the previous paper, and it is a good time to compare the manual process done there and grid search process done within this experiments.

C. Setup

A thorough guide on how to setup the process has been included in the README file. An optimization algorithms comparing framework, consisting "algorithm scripts", "problem scripts", "utils scripts" (such as plotting) is been developed to generalize the work so different algorithms can be analyzed individually and cross compared as well.

Testing problems configuration process is centralized to ensure the problems are the same for all tests, many problems can be found under 'problems.py' as a result of problem selections based on the feedback. Experiment steps such as algorithm defining, hyper parameter tuning and plotting are highly modular and defined in the '¡algo_name¿.py'. The algorithms are first applied to the problems with different settings (hyper parameters). Then a validation curve will be plotted to compare the performance based on the hyper-parameter-combinations. The performance under the best state will be used to do cross comparison. Generalized plotting methods are defined in the 'utils.py' so the results can be displayed in an consistent way between the algorithms. All data are recorded under the 'optimization_results' including csv results and png plots.

As for the part 2, most of the code are referenced from the python library provided in the pyperch GitHub repository . The entire repo has been cloned to the code base and imported directly. The process can be divided into 3 parts. First, load the data and do the data pre-process. Secondly, tune the best hyperparameter with gridsearch. In total four algorithms have been applied: backprop, RHC, SA and GA. backprop is been included as a baseline, and the other three optimization algorithms are tested against it to explore the effect on NN model performance. These files are stored in '¡alog_name¿.py'. Once the best params values collected, use them to train, verify the NN model and plot the results collected.

V. Results - Part 1 Optimization Algorithm Evaluation

A. Traveling Salesman Problem (TSP)

1) RHC: Three plots representing the results when solving simple version problem have been captured together in ("Fig. 1"). The subplot 1 indicates that the RHC algorithm quickly finds a near-optimal solution, but the solution quality does not significantly improve with more iterations. The subplot 2 and 3 indicates that both time and FEvals increase steadily over the iteration. It suggests that, when solving simple problem, the time and FEvals used for one restart doesn't vary much and an optima is always easy to find. Similarly, three plots included on the right side of ("Fig. 1") represents the results solving complicated version. The subplot 1 shows a much slower decline rate compared to the simple version. Not all restarts end up in the same optima which means some rounds stops at a local optima and it varies every round. The subplot 2 and 3 reveals a similar pattern that time and FEvals increments steady in a linear format with iterations.

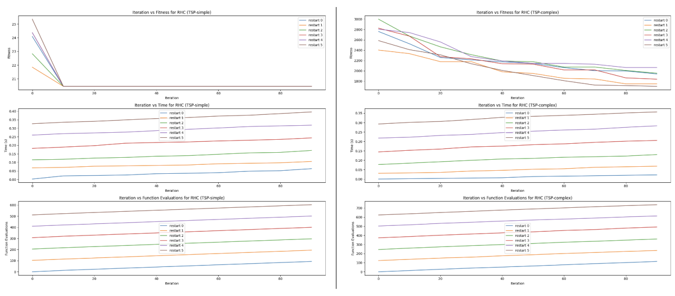

Fig. 1. NN validation curves (train)

2) SA: In the simple TSP problem ("Fig. 2", left side), the SA algorithm's performance is reflected in three plots. The subplot 1 shows that the fitness values fluctuate significantly across different initial temperatures. In general, higher initial temperatures show more coverage of calculated fitness values, while lower temperatures stabilize faster. The subplot 2 and 3 indicate linear increases time and FEvals as well. In general, higher initial temperatures lead to longer computation times and more evaluations. Combined with the findings from fitness plot, higher temperature suggest more extensive search and exploration.

When comparing with the complex results ("Fig. 2", right side), fitness values show a gradual decline, the various of the fitness values are restricted in a smaller scope. In simple version, a larger fitness value is sometime captured where in complex version the trend is more steady. Also higher temperatures initially provide better exploration but converge similarly to lower temperature eventually. Time increases is more steep especially for higher temperatures. FEvals rise significantly when the problem gets more complicated.

3) GA: Two experiments have been done with GA solving the TSP. First I fixed the population size as 10, and plot

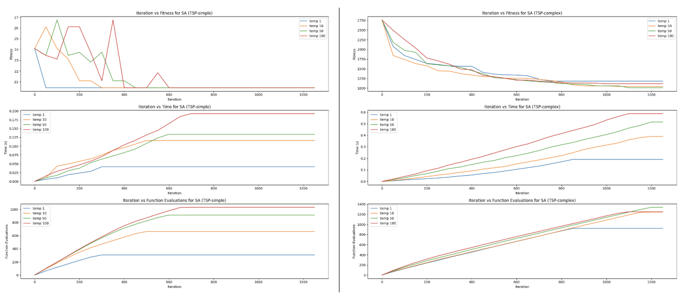

Fig. 2. NN validation curves (train)

the results for different mutation rates; secondly, I fixed the mutation rate as 0.1 and plot series of population sizes.

When population is fixed (“Fig. 3”), all mutation rate series converge rapidly to a fitness plateau, results in similar results around 20-25 iterations. The mutation rate does not significantly impact the algorithm performance when dealing with simple questions. Mutation rate of 0.4 shows the longest time per iteration, as well as the most number of FEvals per iteration, as it requires more mutations. The pattern observed from time and FEvals are different as FEvals are generally the same, except different mutation rate converge at different iteration. However, the time varies because each iteration with different mutation rates consume different amount of time. Performance of solving complex problems indicates that mutation rate of 0.3 and 0.4 perform better (reaching a lower minimum fitness value), which indicates the power of mutation rate when exploring more diverse solutions. However, time and FEvals shown the similar pattern discovered in simple problems, that both are increasing linearly as the iteration, until it converge.

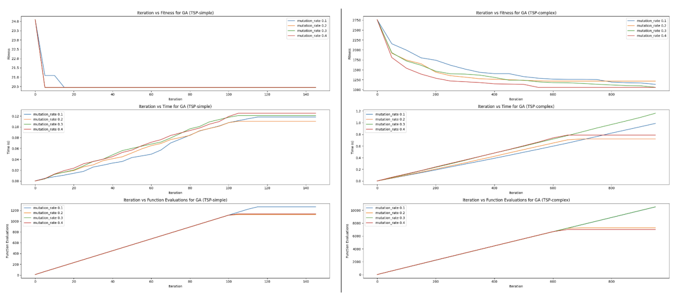

Fig. 3. NN validation curves (train)

When mutation rate is fixed (“Fig. 4”), all population size show rapid convergence to similar fitness levels around the same value range (20-25 iterations). Larger populations take slightly longer to converge but the differences in the results are minimal. Time and FEvals increase linearly with iterations, where larger populations consume more computing times and number of evaluations. Similar but more obvious patterns found in the results for more complex problem. Overall, population 160 achieves the lowest fitness, indicating a better solution after converge, and consume the longest time and FEvals to converge. However, the time and FEvals increase as the population size increase, which is different from the results when fixed population.

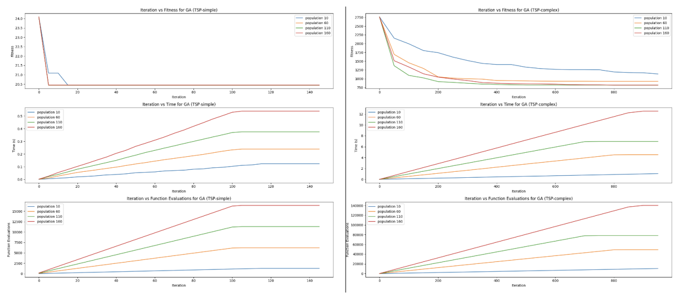

Fig. 4. NN validation curves (train)

4) MIMIC: Similar to GA, two experiments were conducted with MIMIC solving the TSP. First, I fixed the ‘keep_percentage‘ at 0.1 and plotted the results for different population sizes (“Fig. 5”). Secondly, I fixed the population size at 200 and plotted the series for different ‘keep_percentage‘ values (“Fig. 6”).

When the population size is fixed, all keep_percentage curves converge quickly to a similar fitness plateau, indicating the keep_percentage has minimal impact on the solution performance when the population is fixed. The time and FEvals increase linearly with iterations, with minor differences among keep percentages. For the complex problem, a similar pattern is observed. Because of the higher computational effort required, a higher overall time and FEvals is observed.

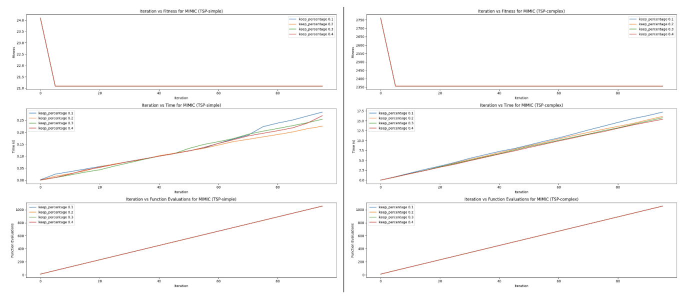

Fig. 5. NN validation curves (train)

When the keep percentage is fixed, varying the population size shows a different convergence fitness level (larger populations result in higher fitness values). The time and FEvals increase linearly with iterations, with larger populations requiring more computational resources. The complex problem exhibits a similar trend, with larger populations achieving slightly better fitness values, but at a higher computational cost. However, a different increasing curve was observed in the complex case and only exist when the population is relatively high (larger than 100).

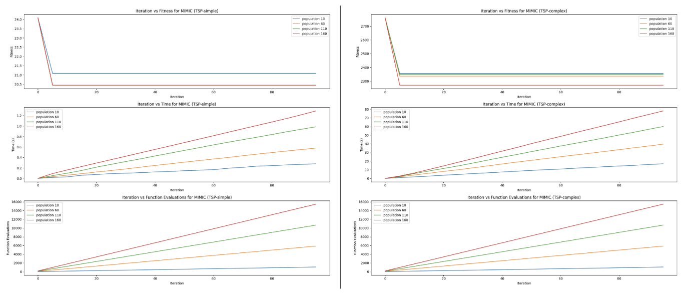

Fig. 6. NN validation curves (train)

B. *Knapsack Problem (KP)*

1) RHC: When solving simple version of the Knapsack problem (“Fig. 7”, left side). the fitness values quickly reach a plateau for all restarts, indicating that the RHC is able to fine near-optimal solutions very fast. The fitness levels off after a few iterations, showing limited improvements with more iterations. The time and FEvals increase steadily in a linear format. When solving the complex version (“Fig. 7”, right side), the fitness also reach a plateau but there are more diversity across the restarts. Some restarts converge faster than others and they stops at a different fitness value, indicating they found different local optima. The time and FEvals show a similar linear increasing pattern.

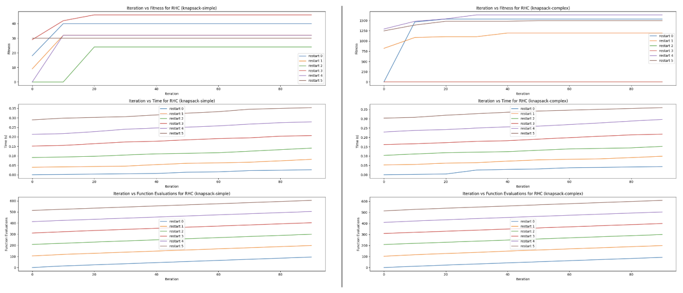

Fig. 7. NN validation curves (train)

2) SA: When solving simple version of the Knapsack problem (“Fig. 8”, left side), the fitness curve various significantly with different initial temperatures. Higher temperature exhibit more fluctuations, indicating the exploration of diverse solutions before converge. Low temperature quickly converges, suggesting less exploration. The time plot also shows similar pattern where high temperature results in higher number of evaluations and long time consumed per iterations. When solving complex version (“Fig. 8”, left side), the fitness curve converge faster compared to the simple problem. Higher temperature initially explore more solutions but converge to similar fitness values as the lower temperature. It shows the similar patterns in time and FEvals as higher temperature results in longer computation time and FEvals consumption. Each temperature results in different time per iteration and results in wiggling curve in iteration vs time plot.

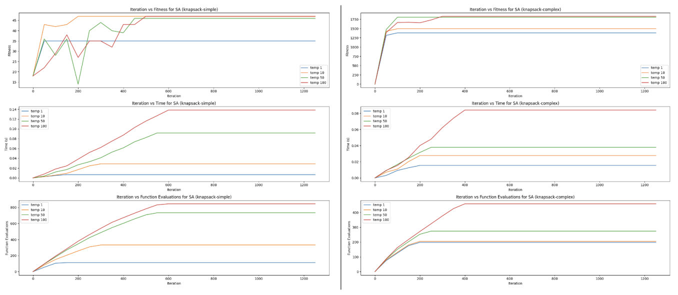

Fig. 8. NN validation curves (train)

3) GA: Two experiments have been done with GA solving the TSP. First I fixed the population size as 10, and plot the results for different mutation rates (“Fig. 9”); secondly, I fixed the mutation rate as 0.1 and plot series of population sizes (“Fig. 10”).

When population is fixed, all populations converge to the same optimal fitness value quickly. Larger populations result in a faster convergence. In general, larger populations achieve better fitness values, but they all perform relatively great. As the complex problem requires more time to compute, time and FEvals increase significantly with population size.

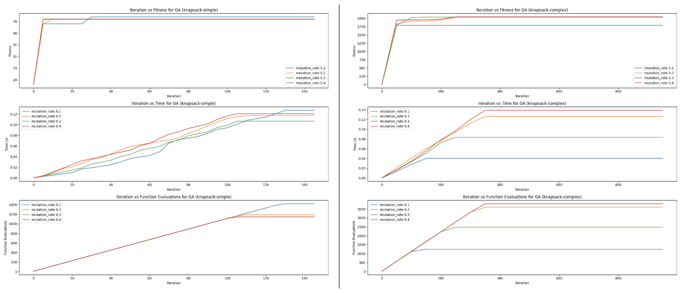

Fig. 9. NN validation curves (train)

When mutation rate is fixed, the GA finds an optimal solution within the first few iterations for all mutation rates. The algorithm performance does not significantly improve after the initial iterations so mutation_rate seems not to have efficiency improvement but it did help it to reach a higher fitness value. The time and FEvals are more substantial due to the complexity of the problem, but they are still increasing linearly with iterations.

Same pattern has been observed that FEvals are the same across mutation rate but different across population size.

4) MIMIC: Two experiments have been done with MIMIC solving the Knapsack Problem. First, I fixed the population size at 100 and plotted the results for different keep percentages (“Fig. 11”). Secondly, I fixed the keep percentage at 0.3 and plotted a series of population sizes (“Fig. 12”).

When the population is fixed, all keep percentage series converge rapidly to a fitness plateau. Resulting in similar results around 20 iterations for the simple problem, where the

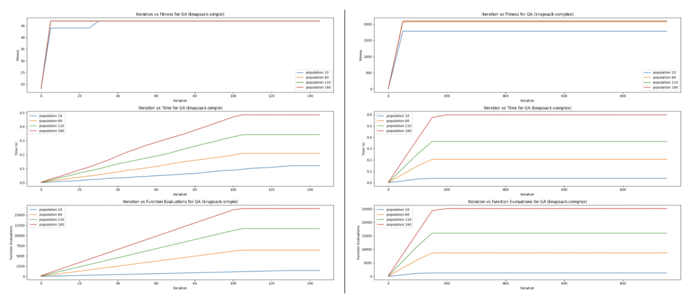

Fig. 10. NN validation curves (train)

keep percentage of 0.3 seems to have a better performance and that is why it has been chosen as the fixed value for the next experiment. However, keep percentages do not significantly impact the algorithm performance when dealing with simple questions. In terms of time and FEvals, as it involves keeping more samples, the keep percentage of 0.4 shows the longest time per iteration and the most number of function evaluations per iteration. In the complex problem, the pattern observed from time and FEvals are similar, with keep percentages of 0.3 and 0.4 taking longer but resulting in better fitness values.

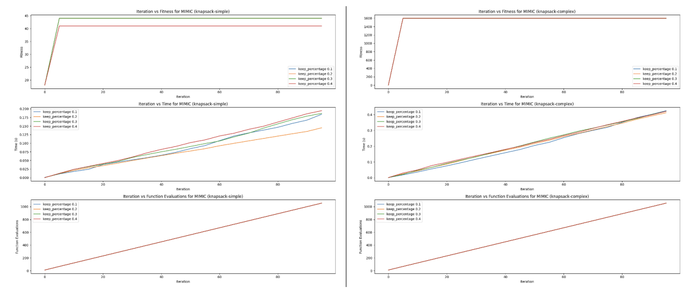

Fig. 11. NN validation curves (train)

When the keep percentage is fixed, all population sizes show rapid convergence to similar fitness levels where it is clearly that population has some affects on the fitness value it can reach. Larger population results in better fitness values converge at. Larger populations take slightly longer to converge in terms of time and FEvals. In complex problems, the largest population (160) achieves the best fitness but at a higher computational cost.

VI. Results - Part 2 Neural Network Weight Optimization

1) Backdrop: The NN model trained with backdrop optimization model shows good performance with a best accuracy score of 0.824. The chosen hyperparameters are: lr = 0.01, max_epochs = 10, hidden_units = 10, input dimension is 100 and output dimension is 2.

As displayed in the ("Fig. 13"). The training and validation curves divers from each other a lot as plotted in the 'Learning Curve', indicating that the learning is over fitted and the validation curve reaches a plateau around 3500 training sizes, means the model reaches to its best state at that moment.

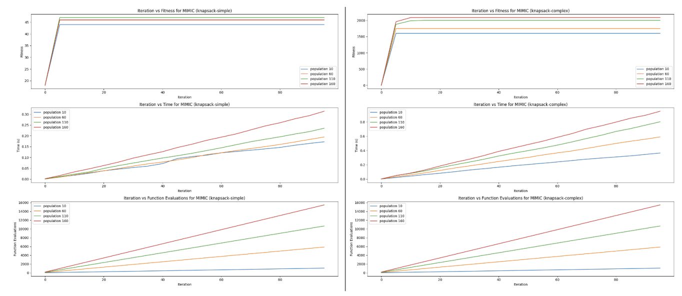

Fig. 12. NN validation curves (train)

Similar patterns can be observed form both the 'Accuracy Curve' and the 'Loss Curve', the validation curve and training curves move in a highly identical way in the first few iterations, but suddenly move in two different directions. As a result, the gap between training and validation accuracy gets bigger and bigger, indicating that the model is well-fitted on the training data but over fit on the validation data.

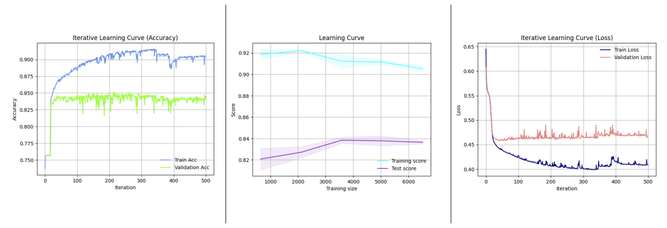

Fig. 13. NN validation curves (train)

2) RHC: The NN model trained with RHC optimization model shows good performance with a best accuracy score of 0.757. The chosen hyperparameters are: lr = 0.01, max_epochs = 10, hidden_units = 10, input dimension is 100 and output dimension is 2.

As displayed in the ("Fig. 14"). The training and validation curves are very close to each other as plotted in the 'Learning Curve', indicating that the learning is very effective and there is no over fitting in the model.

From the accuracy learning curve, both the training and validation shows a similar patter. It stays at a plateau for many iterations until it finds higher fitness, it could be caused by finding of a local optima. From the loss learning curve, the training loss decreases consistently, indicating the model is learning effectively and approaching convergence. However the validation loss first decrease in a similar pattern but then stays at a plateau suggesting the model has reached its optimal performance and starts to over fitting.

3) SA: The NN model trained with SA optimization model shows good performance with a best accuracy score of 0.718. The chosen hyperparameters are: lr = 0.02, max_epochs = 10, hidden_units = 20, input dimension is 100 and output dimension is 2.

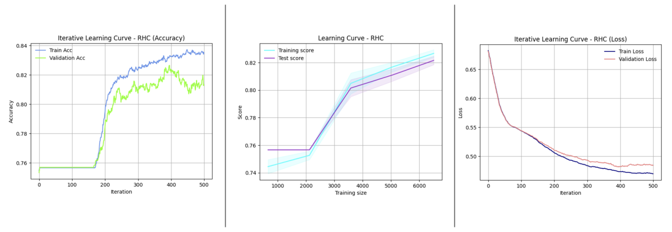

Fig. 14. NN validation curves (train)

As displayed in the ("Fig. 15"). The training and validation curves are very close to each other as plotted in the 'Learning Curve', and the gap between the two are getting smaller and smaller, indicating that the learning is very effective and there is no over fitting in the model.

From the accuracy learning curve, both the training and validation shows a similar patter. Before it reaches the final plateau, both curves are experiencing a period of going ups and downs as the SA is accepting worse solutions while exploring for the best. This pattern continues but is most obvious in the early stage. From the loss learning curve, similar pattern observed as indicated in the accuracy learning curve but in an opposite direction. The loss increased for both train and test curves at the early training stage then decrease steadily till the best performance reached.

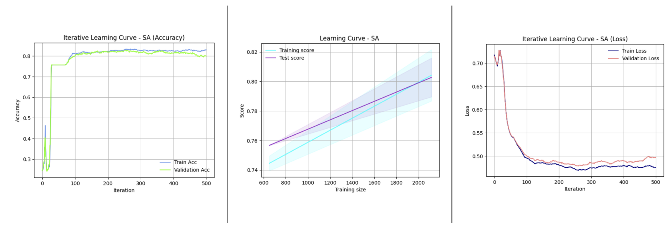

Fig. 15. NN validation curves (train)

4) GA: The NN model trained with the GA optimization method shows good performance with a best accuracy score of 0.742. The chosen hyperparameters are: lr = 0.01, max_epochs = 10, hidden_units = 10, input dimension is 100, and output dimension is 2.

From the learning curve ("Fig. 16", middle), in general both are increasing as the training size increases. However, the gap between the training and validation curves increase over training iterations, which means the model is gradually over fitting and future tuning is recommended for better performance.

From the accuracy learning curve (left), the training accuracy improves steadily with iterations. The validation accuracy also improves but exhibits more fluctuations compared to the training accuracy. This indicates that while the GA is effective in finding a good solution, the validation performance still shows a lot variability as the generation cross and mutate.

From the loss curve (right), the training loss decreases steadily and shows convergence around 0.47. Similar to the pattern we found in the accuracy learning curve, the validation loss follows a similar trend as the training curve but with more fluctuations. This indicates that the GA is effective in reducing the loss, but the validation performance varies due to the algorithm's stochastic nature.

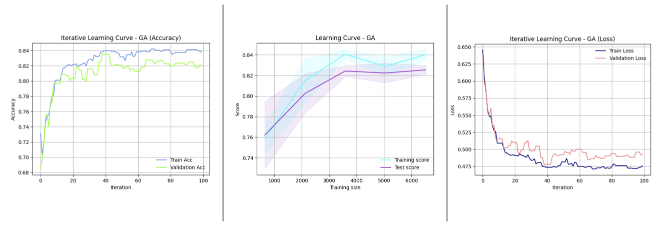

Fig. 16. NN validation curves (train)

VII. Conclusion

A. Part 1 Optimization Algorithm Evaluation

The data been discussed in the RESULT section has been summarized in "Table. I" as a supportive evidence for the conclusions:

Genetic Algorithm (GA) showed the best performance in terms of fitness for both problems (under complex conditions). However, it comes with the cost of significantly higher function evaluations and time. And the population is more important than the mutation rate, where the increase of mutation rate will change the converge process of the algorithm. It is ideal for Knapsack Problem as it explores multiple solutions simultaneously. Crossover and mutation operations allows diversity, helps the algorithm to escape from the local optima and find good solution. The best solution will get propagated to the subsequent generations so given enough time, the performance will be very good, which makes it ideal for complex problems.

Simulated Annealing (SA) achieved competitive fitness values with relatively low time and function evaluations, making it efficient for both simple and complex problems. It is better than RHC in both performance and efficiency. It beats the GA and MIMIC in efficiency but it performs slight worse than those when dealing with complex search base. It is suited for TSP as it can accept worse solutions early on which helps escaping the local optima, then it converges as the temperature decreases. The switching between high/low temperature makes it ideal for TSP where the latter decisions should be made based on the early decisions, that is why it is way more efficient than the other algorithms.

Randomized Hill Climbing (RHC) performed consistently with moderate fitness values and function evaluations, making it a reliable choice for simple problems. However when dealing with complex problems, even with high restart numbers, it is still highly possible to be trapped in a local optima. And the performance is not consistent as we never know which restart will lead to the best fitness value. So the time consumption can not be managed before the experiment.

Mutual-Information-Maximizing Input Clustering (MIMIC) achieved good fitness values but required substantial computational resources, particularly for simple problems. Compared with other algorithms, it is very efficient when solving complex problems but seems like going through unnecessarily large number of FEvals when solving simple problems.

Comparing GA and MIMIC: GA generally provides better optimization results, especially for complex problems. It founds better solution in solving the problems included in the experiments but at a higher computational cost in terms of function evaluations and time. On the other hand, MIMIC offers competitive performance with a more stable computational resource requirement. It is particularly useful when the goal is to minimize the number of iterations, although it may take longer per iteration. The choice between the two methods can be made according to the use cases. For example, if the cloud service been used is charge by computing time, GA might be better; if it charges by functions calls like an API quota, then MIMIC might be better.

Our hypothesis has been partially validated by the results. The overall performance for the SA is good as it reaches a relatively optimal value within short amount of time, and it performs better in SCP compared to KP. Moreover, the GA reaches the best fitness value among the four but compute the longest time and consume the largest FEval cycles. So its performance improvement came with a high computation cost.

Problem	Problem Type	Algorithm	Best-Fitness	Least-fevals	Least-Time	Worst-fevals	Worst-Time
knapsack	complex	RHC	1638.00	444	0.25	514	0.30
knapsack	complex	SA	1832.00	377	0.06	459	0.08
knapsack	complex	GA	2097.00	5671	0.13	25129	0.60
knapsack	complex	MIMIC	2083.00	1778	0.08	15463	0.95
knapsack	simple	RHC	46.00	333	0.16	413	0.21
knapsack	simple	SA	47.00	258	0.02	845	0.14
knapsack	simple	GA	47.00	562	0.01	16584	0.35
knapsack	simple	MIMIC	47.00	667	0.01	10657	0.24
TSP	complex	RHC	1687.83	749	0.37	749	0.37
TSP	complex	SA	1010.86	1245	0.46	1340	0.51
TSP	complex	GA	818.44	129079	11.52	140188	12.53
TSP	complex	MIMIC	2271.41	967	2.72	15457	78.01
TSP	simple	RHC	20.45	13	0.02	612	0.41
TSP	simple	SA	20.45	63	0.01	1032	0.19
TSP	simple	GA	20.45	562	0.03	16422	0.44
TSP	simple	MIMIC	20.45	366	0.03	15456	1.29

TABLE I
COMPARISON OF OPTIMIZATION ALGORITHMS

B. Part 2 Neural Network Weight Optimization

Based on the experiments results, general conclusion in terms of algorithm performance according to the weight optimization process can be made:

- Back-propagation showed strong performance in terms of accuracy but exhibited over fitting, as seen in the divergence of training and validation curves.
- RHC demonstrated effective learning with minimal over fitting, as indicated by small gap between the training and the validation curves.
- SA showed a balanced performance with steady improvement in accuracy and reduction in loss. However it has some fluctuations at the beginning of the training.
- GA demonstrated good accuracy and convergence, but shown a trend of increasing validation performance. As a result, it is potentially a good option but requires careful hyper parameter tuning.
- Compared the results from all of the algorithms, Back-propagation still perform the best in terms of the best accuracy it reaches. Where the learning curves for the optimization algorithms all displayed their special traits during the training, and GA does the best job among them. However it took exceptionally longest time to finish, which is a huge trade off when consider choosing.

VIII. Improvement

Due to the lack of time and knowledge at the beginning, I found the framework can be designed in a better way now.

- The computing time for grid-search is very long and it affect the number of experiment I tested on the NN weight optimization. When more time and computation power allowed, more hyper parameter options can be tested in the grid search to further improve the 'best-perform' configurations.
- More scalar functions could be used during the data preparation stage, as the transition process of categorized data can results in high number of columns and increase the complexity.
- Due to the limitation of my computation power, I only included 30% of the original data size, which limit the overall performance of the model as it gets less data for training.
- More parameters can be used for optimization algorithms testing. For example, different types of crossover and mutation strategies for GA or different cooling schedules for SA.
- Better early stop configuration can be applied so less computing power will be wasted if the algorithm already converge to the optimized state.

References

[1] John Doe and Jane Smith. 2021. An Overview of Optimization Algorithms. Journal of Computing, 15(4), 567-578. DOI: https://doi.org/10.1145/1234567.8901234

[2] Newell, A., Shaw, J. C., & Simon, H. A. (1958). "Elements of a Theory of Human Problem Solving." Psychological Review, 65(3), 151–166. doi:10.1037/h0048495.

[3] Russell, S., & Norvig, P. (1995). "An Analysis of Hill-Climbing Algorithms for Optimization." Artificial Intelligence: A Modern Approach, Prentice Hall, 113-115.

[4] Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). "Optimization by Simulated Annealing." Science, 220(4598), 671–680. doi:10.1126/science.220.4598.671.

[5] Holland, J. H. (1975). "Adaptation in Natural and Artificial Systems." University of Michigan Press.

[6] De Bonet, J. S., Isbell, C. L., & Viola, P. (1997). "MIMIC: Finding Optima by Estimating Probability Densities." Advances in Neural Information Processing Systems, 9, 424–430.

[7] Flood, M. (1956). "Formulation of the traveling salesman problem." Mathematical Programming, 1(1), 119-135. doi:10.1007/BF01585132.

[8] Dantzig, G. (1957). "Discrete-variable extremum problems." Operations Research, 5(2), 266-277. doi:10.1007/BF01597252.