[redacted] Randomized Optimization "RO" Algorithms

By **Sahil Gupta** [redacted]

I. Abstract

Theoretical Expectations of Randomized Optimization "RO" Algorithms

Let's briefly summarize learnings from lectures to compare theoretical expectations with empirical results we'll discuss soon:

1. **Backpropagation Algorithm with Gradient Descent**: This is the default algorithm used to find best weights in common place ANNs. Per Tom Mitchell[1], "In the case of multilayer networks, the error surface can have multiple local minima. *Unfortunately, this means that gradient descent is guaranteed only to converge toward some local minimum, and not necessarily the global minimum error*. Despite this obstacle, in practice Backpropagation has been found to produce excellent results". *In theory*, ***it's expected that incremental stochastic gradient Backpropagation algorithms may either come close to top or likely outperform all the RO algorithms***.
2. **Randomized Hill Climbing or "RHC"** (uses random restarts with hill climbing) finds maxima by iteratively moving towards a higher fitness score for the evaluation function. ***We expect this to get stuck in local maxima sometimes***.
3. **Simulated Annealing or "SA"** (uses generic hill climbing too) finds maxima by "randomly selecting a solution close to current one, measures its quality, & *moves to it according to the temperature-dependent random probabilities*. This could be accepting worse solutions as solution space is explored"[2]. ***This is expected to be better at finding global maxima***.
4. **Genetic Algorithms "GA"** will produce iteratively newer generations that have higher fitness scores than prior generations using concepts of natural selections such as "mutations" and "mating". ***GAs are effective in finding global maxima. For complex problems, GAs do not necessarily scale***. *Hence, it's expected that* ***this will likely underperform given the complexity of the chosen dataset*** *discussed below*.
5. **MIMIC**[3], by our beloved, Dr. Isbell *exploits "knowledge of the structure to guide a randomized search through solution space* & to refine our estimate of the structure by ***successively approximating the conditional distribution of the inputs given a bound on the cost function***. This technique obtains significant speed gains over other RO procedures." ***It's expected to be much faster than GA and equally (if not more) effective for finding optima for some problems***.

A Short Note on Experimentation Approach

Due to my expertise in both coding languages, the best of Java with ABAGAIL & the best of Python is used for experiments:

1. Python with scikit learn is used for pre-processing of data, performing a 75% train data & 25% test data split using `sklearn.model_selection.train_test_split`, `shuffle=true and stratify=y` i.e. data randomly split while maintaining target's distribution balance. This leads to 2 datasets on disk: adult_census_processed_test.csv & adult_census_processed_train.csv that are then used by Java code.
2. *A Java program with ABAGAIL* conducts all the RO experiments that output CSV files with performance metrics.
3. *Python with matplotlib & pandas* is then used to read performance metrics CSV files and create insightful plots below.
4. You are encouraged to follow along with the linked code if interested. This contains dozens of experimental-runs with images and each future run is a subsequent improvement from past runs.
5. Finally, ABAGAIL natively lacked CV functionality. ***Cross validation (CV) is not performed for each experiment due to simplicity & time constraints***. Train and test split is sufficient for the use cases here while CV would've been ideal.

II. Find Optimal Weights for Neural Networks

Introduce Dataset

To find the optimal weights using Randomized Optimization algorithms, the **UCI Adult Census Dataset**[4] from Assignment 1 is used here. The goal of the classification problem behind this dataset is to predict whether income exceeds $50K/yr based on 1994 census data. This dataset really puts the optimization algorithms to a challenge because of this dataset's attributes:

- This dataset contains 32561 instances with ***14 attributes each (multivariate) which are either categorical or integer (heterogeneous)***. Each entry contains demographic attributes about an individual such as age, sex, occupation, race, workclass, education, marital-status, relationship, etc. This ***dataset is imbalanced*** because there are 3 times more rows with <50k USD income than >50k USD income.
- As discussed in Assignment 1 report, *feature engineering and pre-processing is performed on this dataset*. These techniques included "binning for age & hours.per.week", "minmax scaling on binned age, number of education years, etc.", "one hot encoding on categorical data such as workclass, occupation, race, sex" and "removal of irrelevant features with little to no co-relation such as fnlwgt, relationship". As a result, the dataset became ***32561 rows x 38 features*** in final size with the final column for income >$50k, our target value to predict.
- Because of the large size of the dataset, 2 types of experiments were performed and compared with each other:
    a. To understand complex problems, it's best to start on a smaller simplified scale. Hence, a simplified "minimal" Adult Census dataset with only 3 preprocessed features & 1 target output is created.
    b. Finally, all RO algorithms were challenged using the full 38 features dataset & 1 target output.

A Note on Parameter Tuning & Experimental Approach

Extensive parameter tuning has been performed for each Neural Network experiment & each of the 3 strength problems and for each parameter for all the RO algorithms that require tuning (SA, GA, MIMIC) per problem. *The approach taken is very similar to grid search but also using RO Convergence graphs* where all other variables/parameters are kept the same except the one "***control parameter***" being tuned. It is impossible to list all tuning graphs below (8 parameters to tune x (3 + 1) problems given = ~32 graphs). Hence, for the sake of brevity, we'll only talk about key parameters that have the most effect on getting to optimal fitness score with some regard for faster training/execution time for an RO algorithm per iteration. ***For every experiment, at least 3-10 trials each per performed to get smooth graphs and remove variance from just a single trial. Finally, more than 25 experiments were conducted locally and on large AWS instances to get best results truly possible on these problem sets***. Of course, the code can very easily run on local instances. Each experiment had to be multi-threaded to save time.

Fig 1a. This image briefly explains the tuning approach taken throughout this paper.

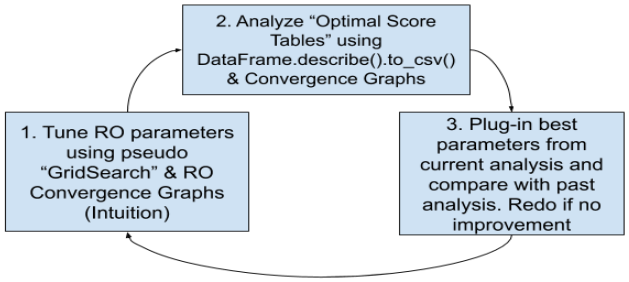
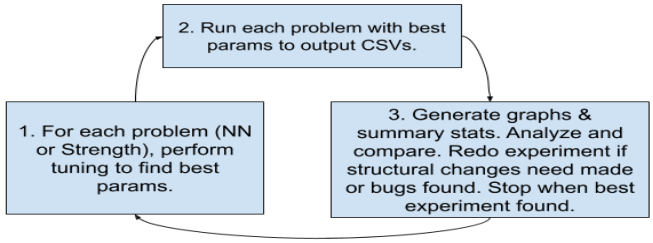


Fig 1b. The experimental approach to comparing all algorithms is described below. Almost 25 experiments had to be conducted to generate this paper.



Metrics Chosen to Compare

To analyze the performance, we'll be employing the same metrics from Assignment 1 i.e. f1_score is used instead of just accuracy or MSE. Accuracy and MSE are ineffective in measuring performance of models built on imbalanced datasets. To recap, f1_score is the harmonic mean of precision & recall. Graphs are generated to compare iterations vs f1 score, precision, recall, MSE, accuracy, function evaluations, total elapsed time, etc. but only iterations vs **f1 score, accuracy, function evaluations and total elapsed time will be used below. F1 score will be a number between 0 & 1 instead of percentages.**

ANN Implementation

To compare the performance of different RO algorithms, we'll use a neural network constructed in assignment 1 as the baseline and see if we can beat this. To recap, a Multi-layer Perceptron (MLP)[5] classifier with 1 hidden layer of 30 neurons with 'relu' as the activation function and max iterations set to 10,000 was used (anywhere between 8-64 neurons produced very similar results per model complexity (MC) analysis): `MLPClassifier(alpha=1e-06, hidden_layer_sizes=(30,),max_iter=10000,random_state=0,warm_start=True)`. Lesser hidden neurons can lead to underfitting of data while a high number of neurons and layers leads to overfitting. Of course, to avoid the pitfall of comparing scikit learn's MLPClassifier, this ANN is constructed in ABAGAIL with the same structure. A feed forward backpropagation network/algorithm is used to determine the optimal weights (similar to 'adam' stochastic gradient-based optimizer). This network uses 1 hidden layer with 30 neurons, relu activation function, resilient backpropagation (RPROP) with learning rate in range {min=0.000001, max=50 & initial learning rate of 0.064}, sum of squares as gradient error measure. Since the dataset has 38 attributes and 1 target value, an input layer of 38 & output layer of 1 is used:

```
BackPropagationNetworkFactory factory = new BackPropagationNetworkFactory();
BackPropagationNetwork network = factory.createClassificationNetwork(new int[] {38, 30,1}, new Relu());
RPROPUpdateRule rule = new RPROPUpdateRule(0.064, 50, 0.000001);
GradientErrorMeasure measure = new SumOfSquaresError();
BatchBackPropagationTrainer optimizationAlgorithm = new BatchBackPropagationTrainer(new
DataSet(trainingInstances), network, measure, rule);
```

ANN Randomized Optimization Parameter Tuning

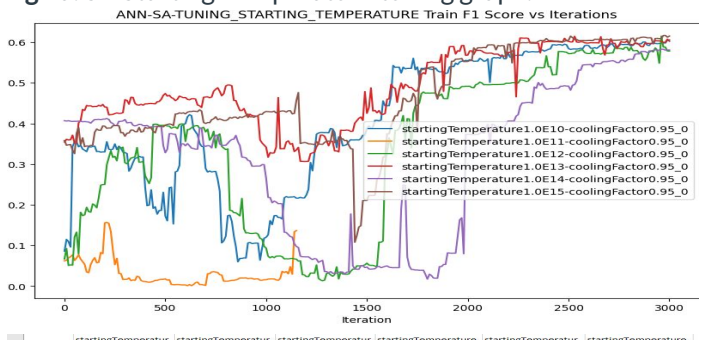
For SA, ***higher temperature*** *implies higher probability to accept worse solutions.* As time progresses, the cooling factor affects the cooling schedule. "*The* ***cooling factor*** *is described as the method for which simulated annealing reduces the temperature to its next value*"[6]. T = 0 gives hill climbing behavior (greedy only) and higher T (infinity) imitates random walk. Following are the best parameters we can see from graphs below:

1. Starting temperature of 1E13 when cooling factor is kept at 0.95 performs best and gets to max f1_score of 0.6145.
2. Starting temperature of 1E11 when cooling factor is kept at 0.35 performs best and gets to max f1_score of 0.63.

Empirically, we can see that higher T performs well only with higher cooling factor and lower T performs well with lower to medium cooling factors. Given above, **let's select the starting temperature of 1E11 when cooling factor is kept at 0.35**.
Another interesting point to note is that *on several un-tuned parameters*, we can see that SA has a "***slow start***" which kind of looks like a "sigmoid or tanh like hyperbolic line". This slow start likely occurs because cooling factor and starting

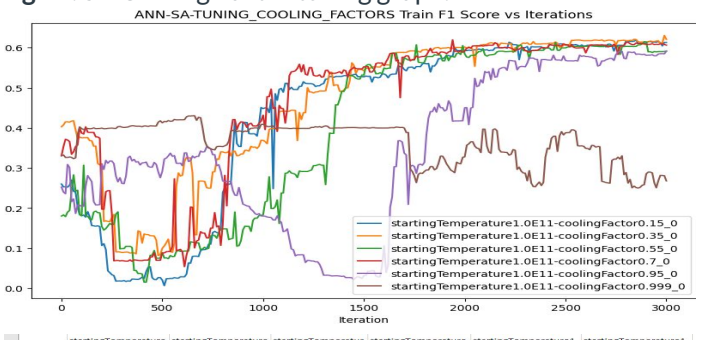
temperature are far apart from their usual matches e.g. high cooling factor and low temperature or low cooling factor and high temperature. ***This means it takes several iterations (~1500) to reach convergence as T is varied via cooling rate***.

Fig 2a. SA Starting Temperature tuning graph.



		startingTemperature1.0E10-coolingFactor0.95_0	startingTemperature1.0E11-coolingFactor0.95_0	startingTemperature1.0E12-coolingFactor0.95_0	startingTemperature1.0E13-coolingFactor0.95_0	startingTemperature1.0E14-coolingFactor0.95_0	startingTemperature1.0E15-coolingFactor0.95_0
1	count	301	116	301	301	301	301
2	mean	0.411068771	0.032641379	0.344219269	0.488526246	0.315998671	0.465261462
3	std	0.166232129	0.034164131	0.196021985	0.092385686	0.18340263	0.114563591
4	min	0.0598	0.001	0.0131	0.3066	0.0172	0.1079
5	25%	0.2913	0.0138	0.1312	0.425	0.0949	0.384
6	50%	0.3998	0.01775	0.3896	0.4692	0.3733	0.4291
7	75%	0.5685	0.035925	0.5019	0.5889	0.4217	0.5968
8	max	0.6055	0.1569	0.6149	0.6145	0.5828	0.6166

Fig 2b. SA Cooling Factor tuning graph.



		startingTemperature1.0E11-coolingFactor0.15_0	startingTemperature1.0E11-coolingFactor0.35_0	startingTemperature1.0E11-coolingFactor0.55_0	startingTemperature1.0E11-coolingFactor0.7_0	startingTemperature1.0E11-coolingFactor0.95_0	startingTemperature1.0E11-coolingFactor0.999_0
1	count	301	301	301	301	301	301
2	mean	0.425670764	0.464259468	0.394386047	0.455677076	0.342052824	0.362477409
3	std	0.216920834	0.171269061	0.213562749	0.190082181	0.187010031	0.050745772
4	min	0.0063	0.0814	0.015	0.0538	0.0212	0.2499
5	25%	0.2485	0.3444	0.1728	0.3777	0.2068	0.3269
6	50%	0.5284	0.5484	0.332	0.5506	0.3176	0.392
7	75%	0.6003	0.6086	0.5907	0.5999	0.5524	0.4005
8	max	0.6196	0.6307	0.615	0.6198	0.5917	0.4304

For GA, there are 3 parameters under discussion. The ***population size*** *is the number of chromosomes present in every generation*[7]. ***To mutate*** *is the number to mutate (flipping the value in each bit position of each chromosome) in each iteration/generation.* ***To mate*** *determines the number to mate in each iteration.* Following are the best parameters per graphs:

1. For toMate=75, toMutate=20, the best population size comes from 300 and even better 2000.
2. For populationSize=600, toMutate=10, the best toMate value is 100.
3. For populationSize=300, toMate=100, the best toMutate is 10.

Empirically, we can see that around higher population size is longer to train but produces better results always[7] ***(size of 2000 took around 1.5 times longer than a size of 100 to train 2000 iterations).*** *A toMate of around 15-30% of population size usually gave best results. A toMutation of around 5-10% of population size led to ideal results.* We'll also see similar behavior in 3 optimization problems below! Keeping training time constraints in mind, ***let's choose a populationSize=600, toMate value=100 & toMutate=30***. Finally, a default uniform crossover in ABAGAIL is used while defining the `NeuralNetworkOptimizationProblem`.

Fig 3. GA Population Size tuning graph.

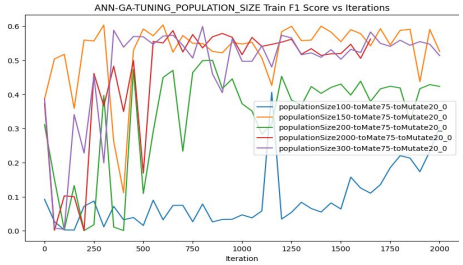


Fig 3. GA "To Mate" tuning graph.

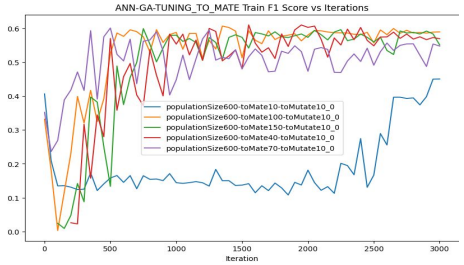
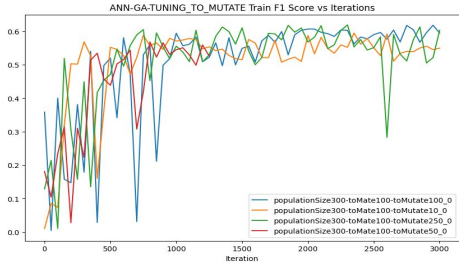


Fig 3. GA "To Mutate" tuning graph.



Here is the final tuned implementation of the optimization algorithms that we'll be talking about in the next section:

```
optimizationAlgorithmsArray[0] = new RandomizedHillClimbing(neuralNetworkOptimizationProblem[0]);
optimizationAlgorithmsArray[1] = new SimulatedAnnealing(1E11, .35, neuralNetworkOptimizationProblem[1]);
optimizationAlgorithmsArray[2] = new StandardGeneticAlgorithm(600, 100, 30, neuralNetworkOptimizationProblem[2]);
```

III. Comparison of RO Algorithms to Find Best Weights for ANN

Expressiveness of Simplistic "Minimal" Problem with 3 Features

Given the large number of attributes and it's size in Adult Census dataset, I ran the experiments on a minimalistic dataset of only 3 features (`age_binned_normalized, work_hours_binned_normalized, education_num_binned_normalized`) first to understand how ANNs can express datasets with the target feature `income_>50K`. The ANN setup is the same as discussed above except it has an input layer of 3, output layer of 1 and a single hidden layer of 3 neurons. This meant 29 links total = 3 inputs, ((3 inputs * 5 Neurons) + (5 neurons * 1 Output)) 20 weights & 6 biases (5 hidden neurons + 1 output)[8].

```
Network Links BEFORE training are: [0.9312,-0.5620,-0.3323,
....,-0.6560,-0.2989,0.3972,0.2212,-0.2262,0.3291]
```

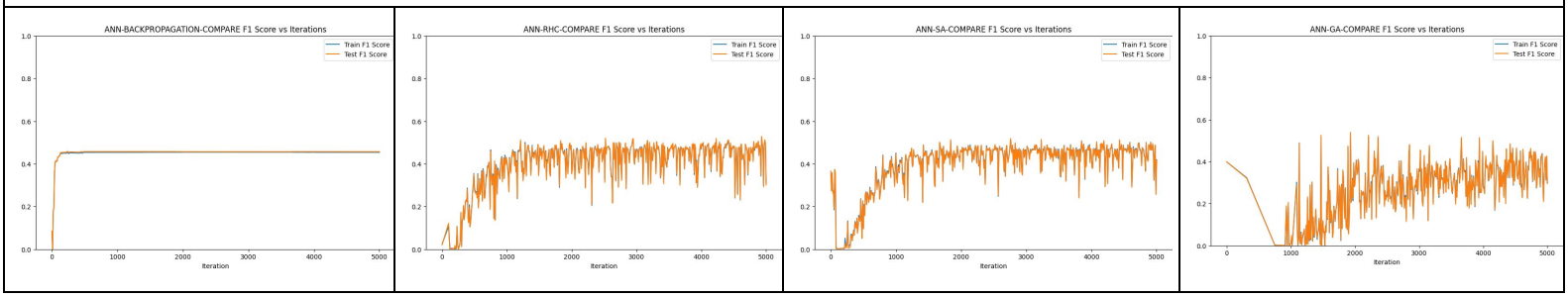
```
Network Links AFTER training are: [1.1545,-0.6811,-0.5492,
....,-4.6147,0.8787,828.6494,-5.4704,-2.6022,-0.0414]
```

We can see that *after training, the absolute values of the weights are higher which shows how RO algorithms are tuning weights to find optima per weight that minimizes the sum of squared error*. Here is an example of how the RO ANN problems are defined:

```
NeuralNetworkOptimizationProblem nnop = new NeuralNetworkOptimizationProblem(new
DataSet(trainingInstances), new BackPropagationNetworkFactory().createClassificationNetwork(new int[]
{INPUT_LAYER,HIDDEN_LAYER,OUTPUT_LAYER },ACTIVATION_FUNCTION), new SumOfSquaresError());
```

Upon plotting the convergence graphs below, it's evident that this ANN is not very expressive as the F1 scores are significantly lower than what we expect per our analysis from assignment 1. From lectures, we know that 5 neurons are likely capable of representing some boolean relationships (e.g. 1 neuron can represent AND & 2 neurons of 1 per 2 layers can represent XOR) and linear data. Of course, 3 features do not contain sufficient information to model the target value income. Finally, per Tom Mitchell[1], "*Every possible assignment of* ***network weights represents a syntactically distinct hypothesis*** *that in principle can be considered by the learner. The hypothesis space is the n-dimensional Euclidean space of the n network weights. Notice* ***this hypothesis space is continuous***." ***VC dimensions of a neural network in this case are finite***: "If the weights come from a finite family (e.g. the weights are real numbers, then the VC dimension is at most O(|E|)."[9] ***This allows for PAC learning***.

Fig 4. F1 score vs iterations convergence graphs for different RO algorithms and Backpropagation on minimal Census dataset.



Full Dataset with 38 Features Analysis

Now that we understand the ANNs a little better with minimal dataset, let's put it to test with all 38 attributes. The weights discussion cannot be shown due to them being 43 times more than minimal dataset i.e. here we have 1239 links = (38*30 +30*1+38 +1+30) that need optimization. This ANN should be far more expressive in estimating the target value than above.

Convergence Curves - F1 Score vs Iterations

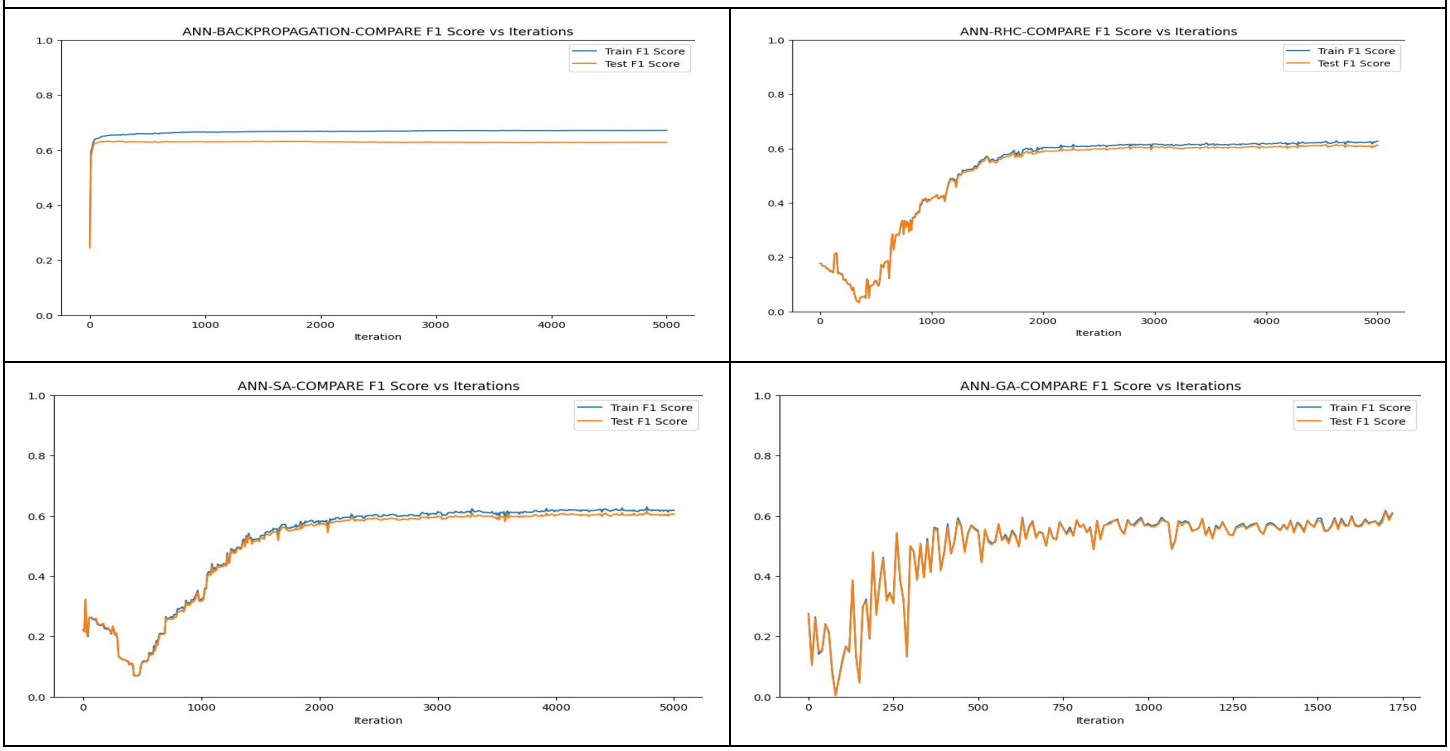
In all ANN experiments, 3 trials per experiment were performed to provide smoother curves. Accuracy & MSE (not shown due to brevity) gave a misleading and over-exaggerated picture of performance with all algorithms near 80-85% accuracy or low MSE of 0.05-0.06 for in-sample and out-of-sample datasets. F1 score shows very insightful results:

1. ***Backpropagation reaches convergence fastest*** *in less that 200 iterations! Upon convergence, backpropagation reaches the highest F1 score among all RO algorithms of around 0.68 on training set and 0.63 on test set. It also had the* ***most consistent and highest precision and recall*** *in comparison to all other RO algorithms (not shown). Though it also* ***suffers from overfitting****! The out of sample test F1 score is lower than in-sample train F1 score. Whereas other RO algorithms such as RHC, SA & GA don't suffer (as bad) from this. Moreover,* ***Backpropagation is very slow in comparison to other RO algorithms.*** The mean training time per iteration was 146ms whereas RHC & SA took 60ms & 61ms respectively. ***Given enough computing resources and train time, Backpropagation is an ideal choice because of it's superlative performance!***
2. ***SA suffered the least from overfitting***. We can see that it's mean F1 score upon convergence is 0.63 for training set and 0.62 for test set. Though, ***due to the "cold start" we discussed in the tuning section above, it suffers initially from underfitting the data set*** as the test set and training set F1 scores are very low. The recall and precision graphs displayed similar behavior though both were smooth. Again, SA is more than 2.5 times faster than Backpropagation. ***SA is a great choice for a weight optimization algorithm on ANNs that requires lesser training time and where datasets suffer from high noise. RHC also suffered less from overfitting. It has similar results as SA with a slightly faster training time per iteration***. Though, SA has slightly better recall and precision before convergence occurs. ***Moreover, SA shows better F1 score performance before convergence than RHC***. SA reaches convergence faster at 1350 iterations in comparison to 1700 iterations for RHC. This is likely because RHC gets stuck on local optima frequently whereas SA finds the global optima faster because it moves according to the temperature-dependent random probabilities.
3. ***GA suffers the least from overfitting and on the bright side, reaches convergence in 700 iterations***. Though it's got the worst performance both in terms of F1 score vs iteration and also training time. GA was restricted to 2000 iterations because ***each iteration took on average 6760 ms i.e. 110 times slower than RHC & SA and 46 times slower than backpropagation. It also suffers from highest volatility***. Clearly, this is a poor choice for dataset & weight optimization.

Unlike all RO algorithms here, it's important to mention that **backpropagation is the only one that can infer back from past errors due to it's feed forward network. Per Tom Mitchell[1], "*Limited depth feedforward networks provide a very expressive***

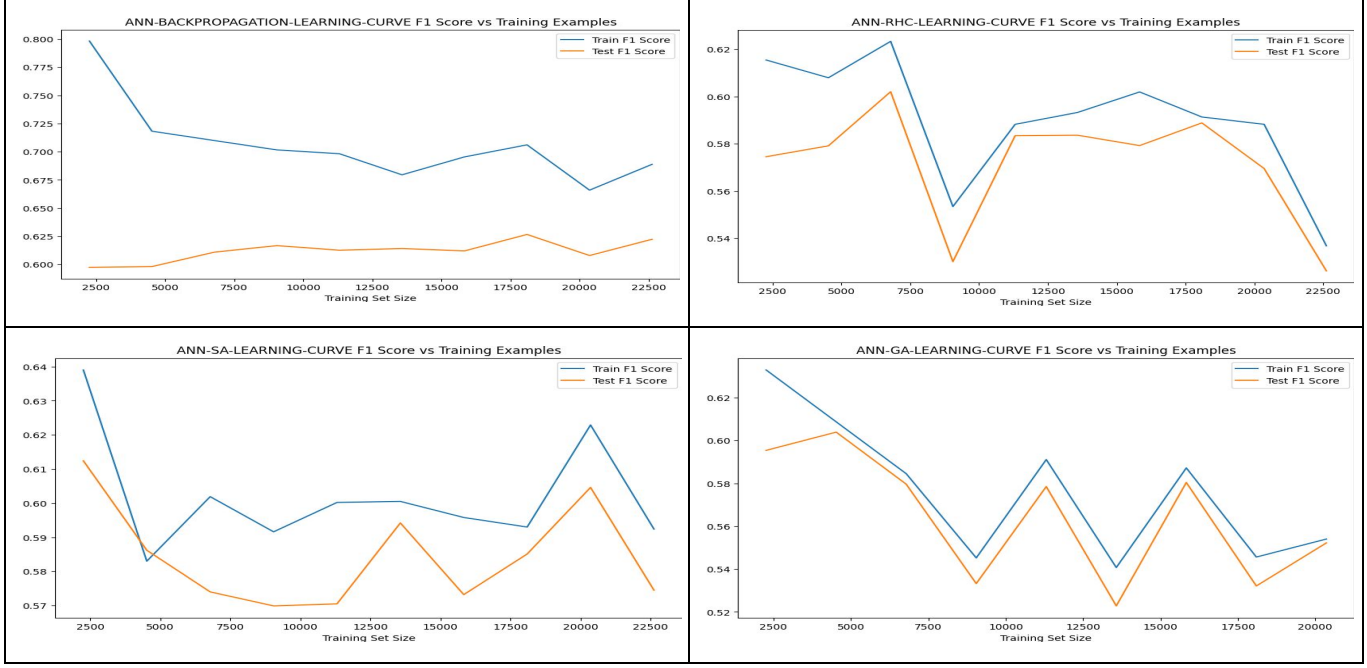
hypothesis space for backpropagation. *It searches the space of possible hypotheses using gradient descent to iteratively reduce the error. Gradient descent converges to a local minimum in the training error with respect to the network weights.*"

Fig 5. F1 score vs iterations convergence graphs for different RO algorithms and Backpropagation on full Census dataset.



F1 Score vs Varying Dataset Size - The "Learning Curves"

Fig 6. F1 score vs Number of Training Examples randomly chosen and tested against the test set for RO algorithms.



Finally, the Figure 6. above shows the learning curve for Backpropagation and all RO algorithms. Notice the different scales on the left - we can see the following order of performance of F1 score: ***Backpropagation (best) > SA > RHC > GA (lowest).*** With backpropagation, we see the typical graph from assignment 1: "*As the training examples size increases, the F1 score on the test set increases and training score decreases.* ***When the training examples size is small, the difference between training score and CV score is high. This implies overfitting to the dataset and after 15000 examples both lines become flat***." RHC (and slightly GA/SA) overfit to the dataset when only <5000 training examples are given. But, RHC, GA and SA quickly recover after 7500 samples and are not as prone to over/under fitting as training samples increase in comparison to backpropagation. The gap

(indicator of overfitting) between training and test F1 score as training size increases is as follows: ***GA (best) > RHC > SA > Backpropagation (highest)***. **Finally, all RO algorithms have high volatility except backpropagation**. ***Backpropagation has the most predictable performance as the number of training examples increases.***

IV. Summary of Neural Networks

In conclusion, backpropagation is undoubtedly the go to choice and the reason why popular ANNs use it for weight optimization. ***It's got the best F1 score on test set, least iterations to converge and most predictable performance. Though, if the dataset suffers from significant noise and training time is an important trade-off, SA & RHC are both good choices with SA being favorite*** among all RO algorithms discussed so far. ***Finally, after thorough analysis, it can be confidently concluded that theoretical and empirical expectations discussed unsurprisingly match-up***! This table below summarizes it all:

	Mean Train F1 Score	Mean Test F1 Score	Max Train Score	Max Test Score	Train Time (ms)	Iterations to Converge
Backpropagation	0.666	0.628	0.672	0.632	146.0858283	200
RHC	0.517	0.508	0.628	0.615	60.52694611	1700
SA	0.511	0.501	0.629	0.616	61.6091151	1350
GA	0.6173	0.6135	0.6173	0.6135	6760.607558	700

V. Comparative Analysis Using Three Strength Optimization Problems

Let's now compare 3 unique problems where we expect one RO algorithm to perform best because of the task at hand and task's nuances. First, we'll talk about why the problem is interesting and discuss its setup. Discuss structure and extensive tuning that was performed on all RO algorithms to ensure they all got their best chance to perform. Notes that ***for all "Strength Problems" experiments, 3 trials per experiment were performed to provide smoother curves***. Extensive reasoning for parameter choices & tuning is similar to the discussion above for ANN and beyond the scope here due to brevity.

Traveling Salesman Problem (TSP) - Best RO Algorithm Found is GA

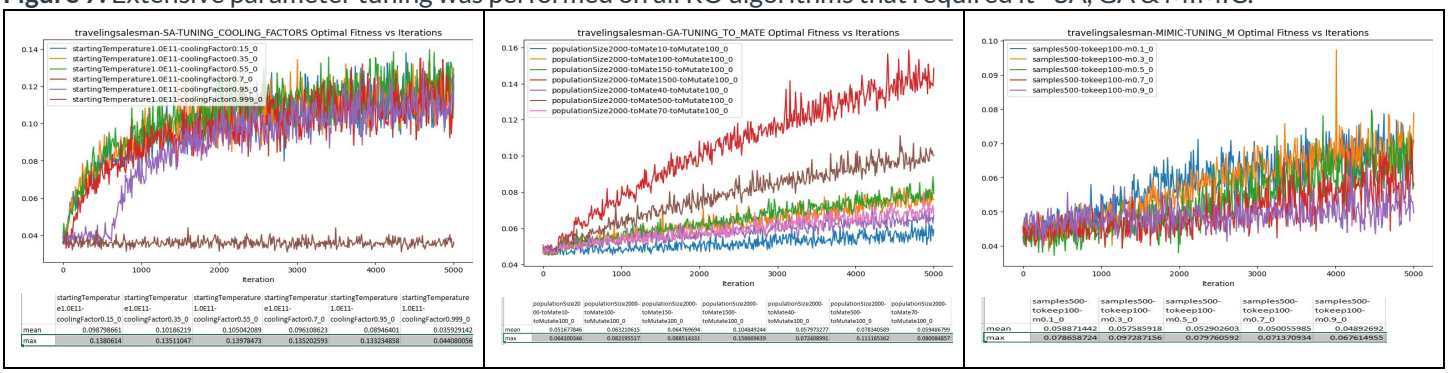
Problem Introduction

In layman words, TSP can be described as a traveling salesman who has to make a choice to visit n cities one at a time traversing with the shortest route possible and then return back to the original city. "***The optimization of vehicle routing problem (VRP) and city pipeline optimization can use TSP to solve***. TSP is an NP typical problem. In a complete graph with n node, find a shortest Hamiltonian loop that traverses all nodes and each node is only accessed 1 time"[10]. Mathematically:

Given $P(X)=\{V_1,V_2,V_3,...,V_n\}$ ***where X={1, 2, 3, ..,n} for n cities.***	***Minimize:*** $T_d=\sum_{i=1}^{n-1} D(V_i,V_{i+1})+D(V_n,V_1)$	***Where D(V_i, V_{i+1}) represents the distance between cities V_i & V_{i+1}***

Parameter Tuning

Figure 7. Extensive parameter tuning was performed on all RO algorithms that required it - SA, GA & MIMIC.



The tables above highlight why decisions were made. In SA, 1E14 outperforms all startingTemperature (not shown for brevity) values. Cooling factor of 0.55 leads to a maximum of 0.139 fitness score. For GA, a large populationSize (not shown) of 2000 outperforms all sizes. Large size wasn't prohibitively problematic as we've seen before. The best toMate is 1500 and leads to the highest optimal fitness score of 0.158! It is evident that larger population sizes require larger toMate and toMutation values too as we've seen with other problems. In MIMIC, m=0.5 and 0.3 are ideal. We'll set m=0.5. Below are parameters of final tuned algorithms for TSP in Java with arguments names shown for ease:

```
simulatedAnnealing = new SimulatedAnnealing(startingTemp=1E14,coolingFactor=.55, ...);
standardGeneticAlgorithm = new StandardGeneticAlgorithm(populationSize=2000,toMate=1500,toMutate=30, ...);
mimic = new MIMIC(200, 30, new GenericProbabilisticOptimizationProblem(mimicEvaluationFunction, new
DiscreteUniformDistribution(ranges), new DiscreteDependencyTree(m=.5, ranges)));
```

Results Summary - Convergence Graphs, Function Evaluations & Wall Clock Times

1. **GA outperformed** all RO algorithms to ***find the approx. global max fitness score of 0.156. After convergence, it's got least volatility or standard deviation from optimal fitness score. GA is the fastest to converge in less than 300 iterations.***
2. SA, MIMIC and RHC never find an approximate global optima and take considerably long to converge.

Fig 8a. A convergence plot of Optimal fitness score vs iterations. *Note that 200 times fewer iterations were performed for GA/MIMIC due to slower training times and to showcase relative convergence with RHC and SA*. Hence the different x-axes.

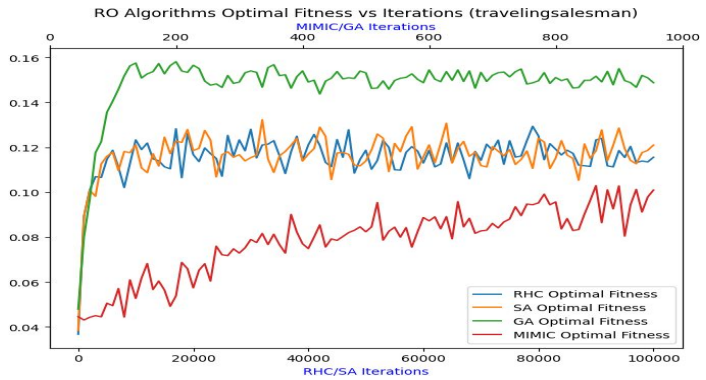


Fig 8b. To avoid the pitfall of iterations mismatch, another experiment with 10000 iterations was performed. Here we can see that ***GA converges fastest and stays at global optimal fitness score***. MIMIC & SA converge at ~4000 iterations.

Fig 8c. ***SA & RHC are the fastest***. GA is ***4100 times slower than SA but 2.4 times faster than MIMIC***. Each MIMIC & GA evaluation takes a longer fixed cost per function evaluation.

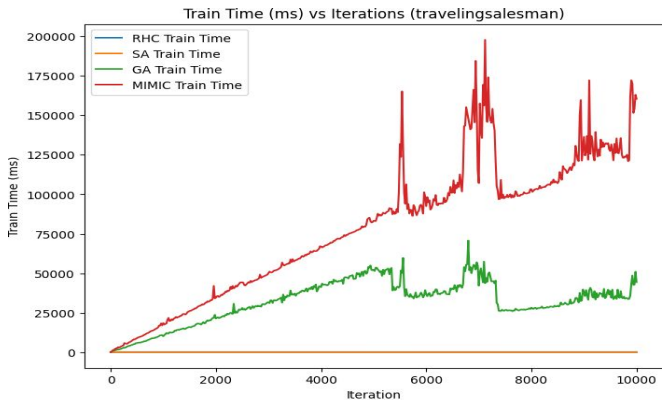
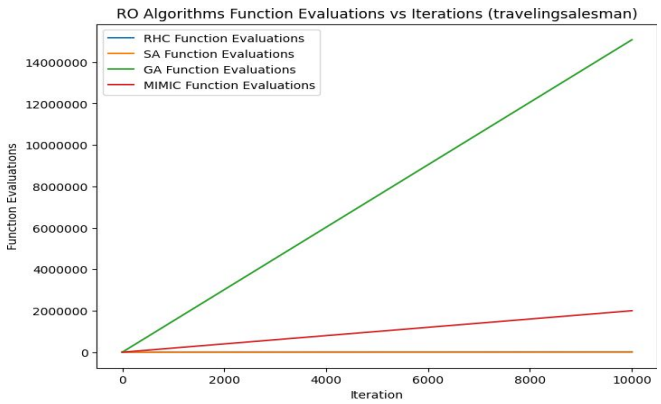


Fig 8d. ***Surprisingly GA takes the most function evaluations vs iterations:*** GA (most) > MIMIC > SA = RHC (low). More function evaluations per iteration and longer evaluation time explains slowness for GA.



Continuous Peaks Problem (CPP) - Best RO Algorithm Found is SA with a Close Second-Best MIMIC

Problem Introduction

The "Continuous-Peaks" problem (CPP) is derived from the Four-Peaks problem. Four-peaks is defined as "Given an input vector X, which is composed of N binary elements, maximize the following[11]:

$$FourPeaks(T, X) = MAX(head(1, X), tail(0, X)) + Reward(T, X)$$

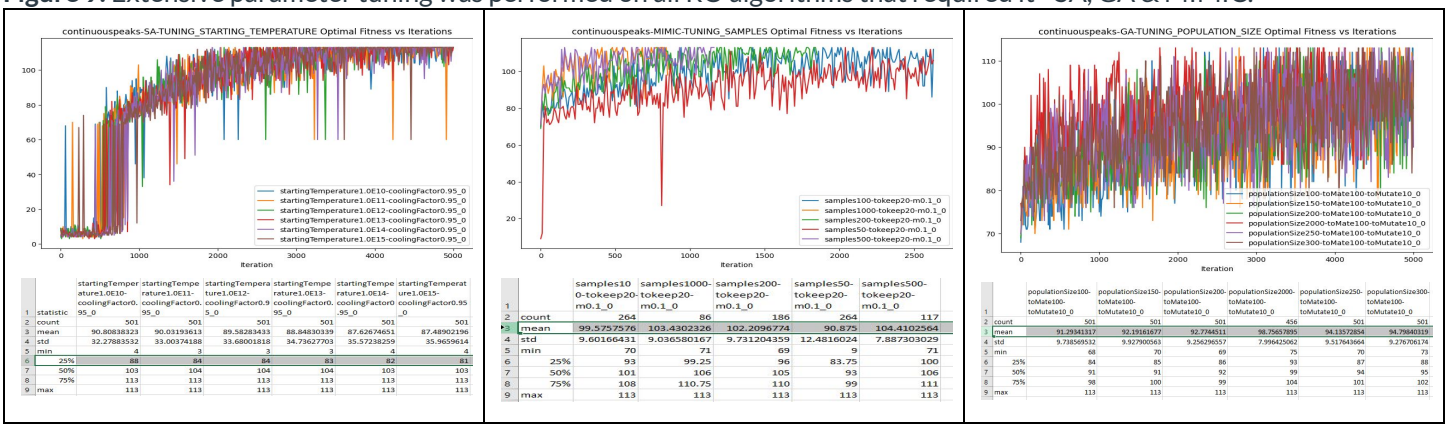
$$Reward(T, X) = \begin{cases} 100 & \text{if } (head(1, X) > T) \wedge (tail(0, X) > T) \\ 0 & \text{otherwise} \end{cases}$$

This means a fitness is maximized if a string is able to get both the reward of 100 and if the length of one of head(1,X) or tail(0,X) is as large as possible. ***Continuous-Peaks doesn't force 0's & 1's to be at the ends of solutions string, they are allowed to form anywhere in the string***." Because of this modification, ***CPP has several local optimas in comparison to 4-peaks***, which only contains 2 sub-optimal local optima controlled by T and a global optima inversely proportional to these 2 local optimas. ***For the CPP problem, N=60 & T = 6 was set during the experiments.***

Parameter Tuning

The tables in Fig 9. highlights why decisions were made. Again, larger population sizes & samples are preferred for GA and MIMIC respectively but they are prohibitive due to high train time. E.g. in GA train time for 100 iterations & size=2000 is 2985ms while it's 613ms for size=300.

Figure 9. Extensive parameter tuning was performed on all RO algorithms that required it - SA, GA & MIMIC.



	statistic	startingTemperature1.0E10-coolingFactor0.95_0	startingTemperature1.0E11-coolingFactor0.95_0	startingTemperature1.0E12-coolingFactor0.95_0	startingTemperature1.0E13-coolingFactor0.95_0	startingTemperature1.0E14-coolingFactor0.95_0	startingTemperature1.0E15-coolingFactor0.95_0
2	count	501	501	501	501	501	501
3	mean	90.80838323	90.03193613	89.58283433	88.84830339	87.62674651	87.48902196
4	std	32.27883532	33.00374188	33.68001818	34.73627703	35.57238259	35.9659614
5	min	4	3	3	3	4	4
6	25%	88	84	84	83	82	81
7	50%	103	104	104	104	103	103
8	75%	113	113	113	113	113	113
9	max	113	113	113	113	113	113

		samples10 0-tokeep20- m0.1_0	samples1000- tokeep20- m0.1_0	samples200- tokeep20- m0.1_0	samples50- tokeep20- m0.1_0	samples500- tokeep20- m0.1_0
2	count	264	86	186	264	117
3	mean	99.5757576	103.4302326	102.2096774	90.875	104.4102564
4	std	9.60166431	9.036580167	9.731204359	12.4816024	7.887303029
5	min	70	71	69	9	71
6	25%	93	99.25	96	83.75	100
7	50%	101	106	105	93	106
8	75%	108	110.75	110	99	111
9	max	113	113	113	113	113

		populationSize100-toMate100-toMutate10_0	populationSize150-toMate100-toMutate10_0	populationSize200-toMate100-toMutate10_0	populationSize2000-toMate100-toMutate10_0	populationSize250-toMate100-toMutate10_0	populationSize300-toMate100-toMutate10_0
2	count	501	501	501	456	501	501
3	mean	91.29341317	92.19161677	92.7744511	98.75657895	94.13572854	94.79840319
4	std	9.738569532	9.927900563	9.256296557	7.996425062	9.517643664	9.276706174
5	min	68	70	69	75	70	73
6	25%	84	85	86	93	87	88
7	50%	91	91	92	99	94	95
8	75%	98	100	99	104	101	102
9	max	113	113	113	113	113	113

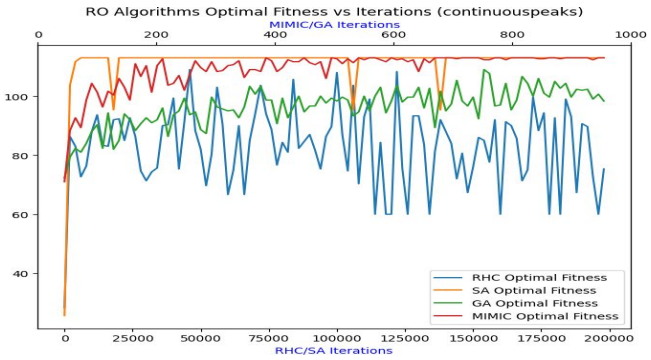
Below are parameters of final tuned algorithms for CPP in Java with arguments names shown for ease:

```
simulatedAnnealing = new SimulatedAnnealing(startingTemp=1E10,coolingFactor=.55, ...);
standardGeneticAlgorithm = new StandardGeneticAlgorithm(populationSize=500,toMate=200,toMutate=100, ...);
mimic = new MIMIC(300, 20, new GenericProbabilisticOptimizationProblem(mimicEvaluationFunction, new
DiscreteUniformDistribution(ranges), new DiscreteDependencyTree(m=.7, ranges)));
```

Results Summary - Convergence Graphs, Function Evaluations & Wall Clock Times

Fig 10a. A convergence plot of Optimal fitness score vs iterations. *Note that 200 times fewer iterations were performed for GA/MIMIC due to slower training times and to showcase relative convergence with RHC and SA*. Hence the different x-axes.

Fig 10b. To avoid the pitfall of iterations mismatch, another experiment with 10000 iterations was performed. Here we can see that ***MIMIC converges faster than SA in 1000 iterations and stays at optimal fitness of 113 throughout***. SA converges to optimal fitness at around 4000 iterations.



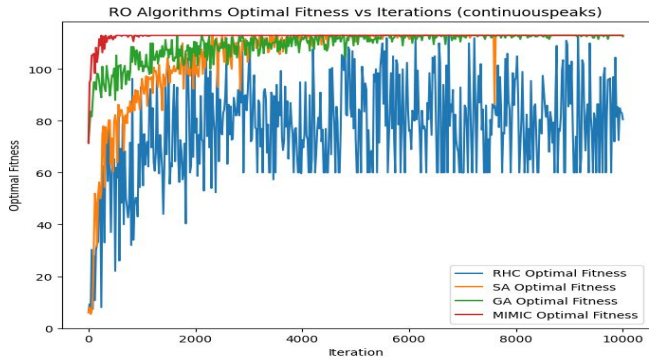
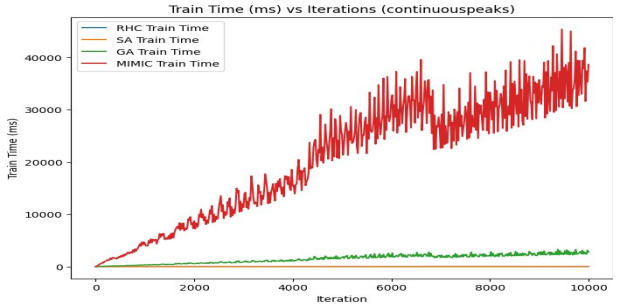
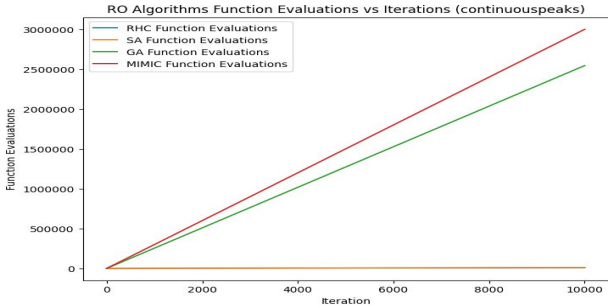


Fig 10c. ***SA & RHC are the fastest*** and MIMIC is ***4600 times slower than SA***. Each MIMIC & GA evaluation takes a longer fixed cost per function evaluation.

Fig 10d. Function evaluations vs iterations graph shows that ***MIMIC (most) > GA > SA = RHC (low). More function evals per iteration and longer evaluation time explains slowness***.





1. ***It is clear from the graphs below that SA performs best when it comes to finding max fitness score in appropriate training time.*** Figure 10. We can see that SA, MIMIC & GA are capable of discovering the global maxima fitness score of 113. RHC gets stuck on local optimas and hence is disqualified from this analysis for further comparison.
2. ***SA's mean train time per iteration is 4.47ms i.e. 4600 times faster than MIMIC (20656ms) & 334 times faster than GA***!Hence, it's prudent to pick SA as it'll reach convergence faster than MIMIC in terms of train time (see Fig 10a).

Knapsack Problem (KP) - Best RO Algorithm Found is MIMIC with a Close Second-Best GA

Problem Introduction

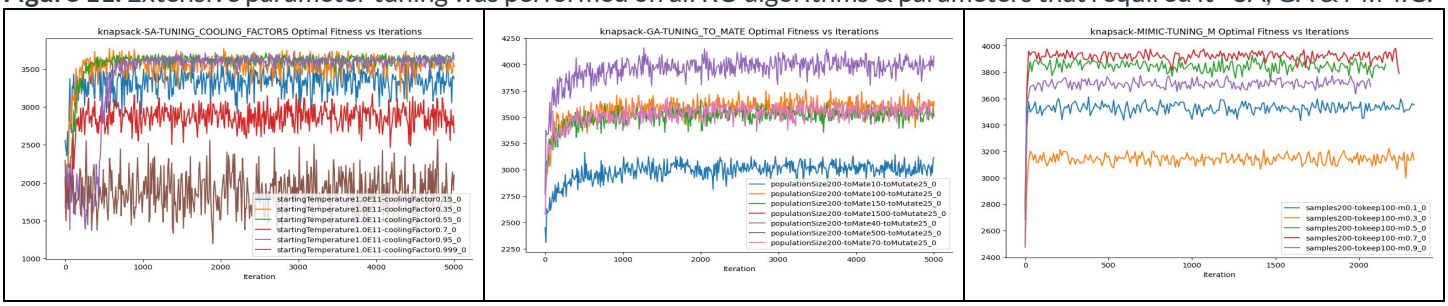
The Knapsack problem (KP) is a combinatorial optimization problem[12] where "given a set of items, each with a ***weight*** w_i and a ***value*** v_i, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible." Again, here the maximum weight of knapsack is the limit and *our goal is to carry items which maximize value.* KP is very interesting because of its application in resource allocation, figuring out optimal tasks under budget constraints, finance - portfolio optimization, etc. ***In ABAGAIL, it's a bounded KP as no restriction is placed on the number of copies*** x_i ***of each kind of item except the length of Integer.MAX_VALUE (c)***. Mathematically, this is defined as:

$$\text{maximize} \sum_{i=1}^{n} v_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq W \text{ and } x_i \in \{0, 1, 2, \ldots, c\}.$$

Parameter Tuning

The Figure 11. highlights why decisions were made. For GA, a high populationSize=2000 (not shown for brevity) is a clear winner but very slow. In fact, 200 does better than 250 & 350 and is also faster in train time. For population size of 200, toMate=40 outperforms and reaches an optimal score of 4000-4050. For MIMIC, m=0.7 is ideal. In ABAGAIL's MIMIC, a ***DiscreteDependencyTree probability distribution is used*** *when defining the optimization problem.* ***This m is the small positive value that used to add when making this tree***. Remember, this probability distribution is used for the first part of MIMIC's partial solution[13] i.e. "***a randomized optimization algorithm that samples from those regions of the input space most likely to contain the optima for C(). By varying this positive number, we're affecting the probability distribution.*** *"If we had access to* $p^{\Theta m}(x)$ *for* $\Theta_m = \min_x C(x)$ *a single sample would be sufficient to find an optimum."* ***Hence, m=0.7 gives us the ideal probability distribution we require.***
Figure 11. Extensive parameter tuning was performed on all RO algorithms & parameters that required it - SA, GA & MIMIC.



Below are parameters of final tuned algorithms for KP:

```
simulatedAnnealing = new SimulatedAnnealing(startingTemp=1E12,coolingFactor=.55, ...);
standardGeneticAlgorithm = new StandardGeneticAlgorithm(populationSize=200,toMate=40,toMutate=20, ...);
mimic = new MIMIC(200, 20, new GenericProbabilisticOptimizationProblem(mimicEvaluationFunction, new
DiscreteUniformDistribution(ranges), new DiscreteDependencyTree(m=.7, ranges)));
```

Results Summary - Convergence Graphs, Function Evaluations & Wall Clock Times

Fig 12a. A convergence plot of Optimal fitness score vs iterations. Here is the order of performance on KP: ***MIMIC > GA > (RHC ~ SA).*** In earlier iterations <500, it's a close competition between GA, SA & RHC but MIMIC converges fastest to approx. global maxima fitness score of 4100.

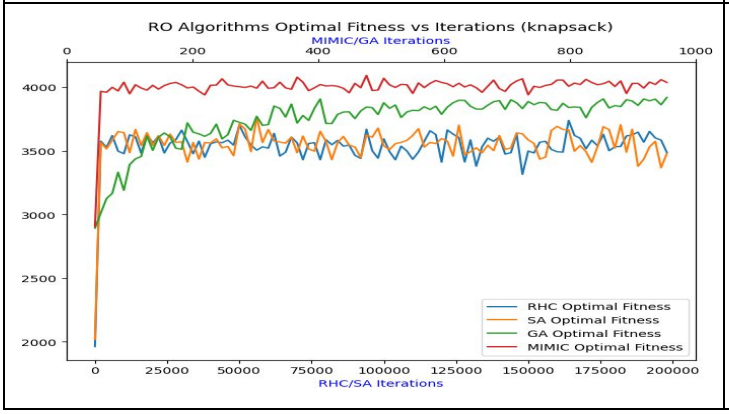


Fig 12b. To avoid the pitfall of iterations mismatch, another experiment with 10000 iterations was performed. Here we can see that ***MIMIC converges the fastest and stays near approx. optimal fitness of ~4100***. GA is second best in finding optima.

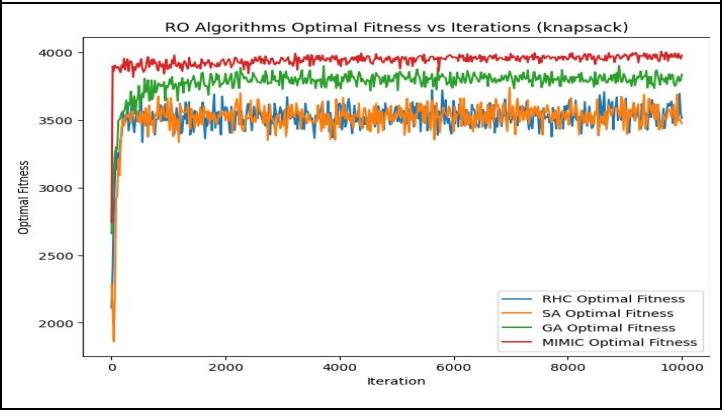
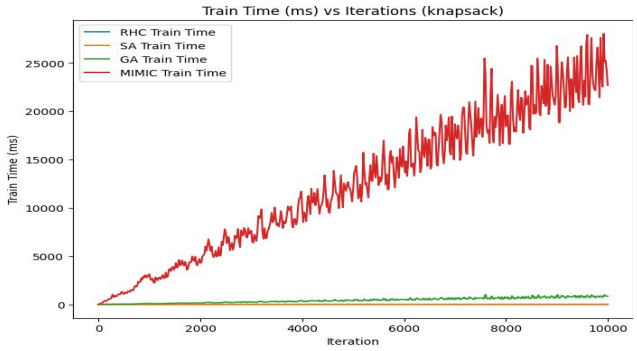
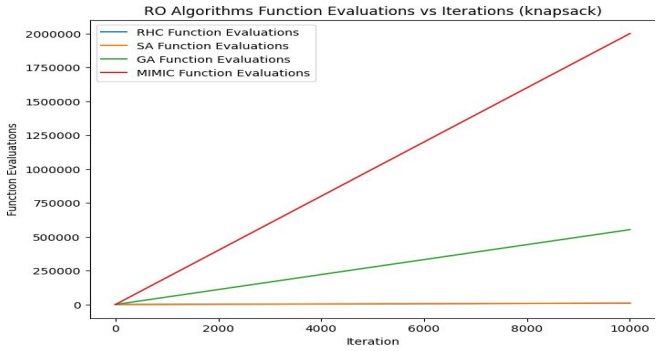


Fig 12c. ***Unsurprisingly MIMIC is the slowest & has a mean training time of 12395ms.*** The ordering is: RHC(1.569ms - fastest) > SA (5.67ms) > GA (435ms) > MIMIC (slowest).	**Fig 12d.** Function evaluations vs iterations graph shows that ***MIMIC (most) > GA > SA = RHC (low). More function evals per iteration and longer evaluation time explains slowness.***
	

1. ***Graphs show that MIMIC performs the best when it comes to finding the max fitness score.*** Figure 12. We can see that GA & MIMIC are both very capable of *discovering the approx. global max fitness score with MIMIC still performing better*. RHC & SA are nowhere close to finding the global optima and are likely stuck on local optima after convergence.
2. ***MIMIC remains the slowest in terms of training time likely due a high number of function evaluations per iteration. GA is 29 times faster than MIMIC, so if train time is a consideration, GA can be a good choice too*** *with a sacrifice in performance.*

V. Summary of Three Optimization Problems & Their Performance

This heat map summarizes it all. **Dark green showcases the best** performing algorithms on that problem, **light green/yellow** showcases ones that **do fairly well too** and **red are ones that got disqualified** from analysis due to poor performance.

	Traveling Salesman				Continuous Peaks				Knapsack			
RO	**Mean Train F1 Score**	**Max Train Score**	**Train Time (ms)**	**Iterations to Converge**	**Mean Train F1 Score**	**Max Train Score**	**Train Time (ms)**	**Iterations to Converge**	**Mean Train F1 Score**	**Max Train Score**	**Train Time (ms)**	**Iterations to Converge**
SA	0.103290979	0.134641	7.7837658	4000	105.90319	113	4.47838	4000	3514.1386	3740.542	5.6739854	500
GA	0.14628419	0.156872	32005.273	300	109.44844	113	1495	5000	3780.318	3902.648	435.27212	750
MIMIC	0.108314695	0.133873	78147.209	3500	112.66401	113	20655.9	1000	3942.2805	4092	12395.186	200

It is unsurprising to see that our theoretical understanding matches very closely with empirical results:

1. For TSP: ***GA works best on NP hard problems & TSP is NP hard***! GA doesn't get stuck in local optimas and converges fastest to approx. global maxima. GAs don't ensure optimal solutions; but, they give good approx. usually in time[14].
2. For CPP: ***SA works well because it doesn't get stuck in local maxima*** since it moves according to its temperature-dependent random probabilities. Moreover, ***SA is the fastest RO algorithm in terms of training time***.
3. For KP: ***MIMIC's two phased solution and "knowledge of the structure to guide a randomized search through solution space & to refine our estimate of the structure" works well on very hard combinatorial optimization problems***. This is the reason why MIMIC does best followed by GA.
4. **RHC while the fastest doesn't perform well** to our needs on any problems and isn't a canonical choice to solve these.

VI. Citations

[1] Tom Mitchell, "Machine Learning", McGraw Hill, 1997. [Book]
[2] Wikipedia, "Simulated Annealing". [URI]
[3] Charles L. Isbell, "Randomized Local Search as Successive Estimation of Probability Densities". [URI]
[4] Ronny Kohavi & Barry Becker, Data Mining & Visualization, Silicon Graphics, "Census Income Data Set". [URI]
[5] Scikit Learn, "MLPClassifier". [URI]
[6] Arya Vigeh, "Investigation of a Simulated Annealing Cooling Schedule used to Optimize the Estimation of the Fiber Diameter Distribution in a Peripheral Nerve Trunk". [Thesis]
[7] Shumeet Baluja & Rich Caruana, "Removing the Genetics from the Standard Genetic Algorithm". [E-Journal]
[8] Stack exchange, "How many parameters does a neural network have?" [URI]
[9] Wikipedia, "Vapnik–Chervonenkis dimension". [URI]
[10] Chunhua Fua, Lijun Z., Xiaojing W., Liying Q., "Solving TSP Problem with Improved Genetic Algorithm". [E-journal]
[11] Shumeet Baluja & Scott Davies, "Using Optimal Dependency-Trees for Combinatorial Optimization: Learning the Structure of the Search Space". [E-Journal]
[12] Wikipedia, "Knapsack Problem". [URI]
[13] J. S. De Bonet, C. L. Isbell, and P. Viola (1997), "MIMIC: Finding Optima by Estimation Probability Densities". [E-journal]
[14] Hindawi, Computational Intelligence and Neuroscience, "Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator". [E-journal]