CS 7641 Machine Learning: Project-2

[Arti Chauhan: Mar-10-2018]

Abstract

This paper is split into two parts and explores below-mentioned four randomized optimization algorithms.

- *Part-1 presents evaluation of neural network with three optimization algorithms on Phishing dataset.*
- *Part-2 presents analysis of below-mentioned algorithms for three discrete optimization problems.*

Analysis was done in ABAGAIL. Please refer to README.txt for more details on test-code.

Random Optimization Algorithm

RANDOM HILL CLIMBING (RHC)

RHC conducts a series of hill-climbing searches from randomly generated initial states. It performs greedy searches as it makes a move to neighbor only if it yields a better solution. Random restarts reduces the chances of algorithm getting stuck in local optimum getting but doesn't eliminate it. RHC can run into hot waters if error surface has large flat regions (plateau). On positive side, RHC is fast and simple.

SIMULATED ANNEALING (SA)

SA models the physical process of heating a material and then slowly lowering the temperature to decrease defects, thus minimizing the system energy. It is similar to RHC but uses 'explore' and exploit' phase to explore the search space better and avoid getting stuck in local optima. At the beginning when temperature is high, probability of accepting worse solution is high but as temperature decays, algorithm favors better solutions. At low temperatures, SA acts like RHC. Key is to allow the system to cool slowly but downside of it is that it requires more computation and time.

GENETIC ALGORITHMS (GA)

Genetic Algorithms (GAs) are adaptive heuristic search algorithm based on the evolutionary ideas of natural selection and genetics. They are suitable for problems with large state-space, multi-modal state-space, or n-dimensional surface. Algorithm works by randomly initializing a population of solutions and selecting the fittest solutions to mate and mutate to create offspring. New population is created by replacing unfit members with these offsprings. Overtime, the population would evolve to optimal solution.

MIMIC

One drawback that above algorithm have is that they are amnesic- they don't preserve history or structure of problem space. MIMIC overcomes this issue by using probability densities to build the structure. It generates samples from a probability distribution, sets theta (n^{th} percentile), selects samples with fitness > theta and recalculates the distribution. This process is repeated until all suboptimal solutions are eliminated.
MIMIC is suitable of problems where cost of computing fitness function is high, as it can converge to an optimal solution in fewer iterations. On flipside, it is more complex and each iteration takes more time.

Part 1 – Evaluating neural networks with optimization algorithms

This section evaluates performance of three optimization algorithms namely randomized hill climbing, simulated annealing and genetic algorithms on phishing website dataset. This dataset was used in assignment-1 to evaluate different supervised algorithms.
Phishing dataset entails features that have proved to be sound and effective in predicting phishing websites. It has 2456 instances and 30 discrete attributes. Goal is to predict if a website is malicious or not.

1.1 METHODOLOGY

- 70-30 split is performed on dataset to create a training and test set. All three algorithms were tuned using training set. I used the same setting as in assignment-1 for neural –network here to have apple to apple comparison (1 hidden layer with 30 neurons). All results were averaged over 5 runs.
- **Parameter Tuning** : Following parameters were tuned to achieve best results.
  1) Simulated Annealing
     a. Cooling rate – this parameter is critical in balancing 'explore' and 'exploit' phase of the algorithm. Very fast cooling imply that algorithm will spend little time exploring the search space and may get stuck in local optima. At Low temperature, SA acts as RHC.
     b. Initial Temperature – if initial temperature is too low, then search gets restricted to the region around starting point. If it is too high then algorithm keep performing random walks and may result is unsuccessful search if number of iterations performed are low.
  2) Genetic Algorithm
     a. Population size – it is one of the important parameter that defines number of chromosomes (solutions) per generation. Too few chromosomes, GA have few possibilities to perform crossover and only small part of the search space is explored. Too many chromosomes, slows down algorithm.
     b. Mate - Crossover is done is to preserve and propagate parameters that might be responsible for generating a favorable solution ie create better solutions from existing solutions.
     c. Mutate – brings diversity and prevents GA from falling into local extremes.
  3) MIMIC
     a. Samples# - number of sample to generate from distribution.
     b. Tokeep – number of samples to keep post each iteration.
  4) Iterations

     Tests were performed for a range of iterations (100-8000) to evaluate point at which algorithm convergences. This parameter is applicable to all above-mentioned algorithms.

- **Performance metric** - Classification Accuracy is used as primary evaluator. It's a reasonable choice because Phishing dataset is quite balanced and value of F-1 measure is very close to Accuracy. (Elaborated in Assignment-1)

- **Analysis framework** -Training and Testing was performed in ABAGAIL. Relevant code can be found in ac_PhishingTest_*.java (one file for each algorithm).

1.2 RANDOMIZED HILL CLIMBING (RHC)

Charts below show train/test accuracy and time for different iteration count.

- Accuracy improved noticeably as number of iterations was increased from 50 to 2500 (Fig1.2a) but after that gains were too small to justify the increase in train time. In other words, algorithm seems to converge around 2500-3000 iterations.
- Training time increased linearly with number of iterations (Fig1.2c).
- Beyond 2500, gap between test and train accuracy started to widen, suggesting algorithm is trying too hard to fit the data and may lead to overfitting. Hence it would be wise to stop training around that point (Fig1.2b).

It would be nice to program this algorithm in such a way that a) it stops iterating if improvement from one run to other is less than some threshold, b) perform random restarts when stuck in local optima.
Number of iterations should be sufficiently large to allow algorithm to explore search space and discover global optima. After evaluating both accuracy and time graphs, I believe a happy-medium is at 2500 iterations with test-accuracy =94.57 and train-time =17.14sec.

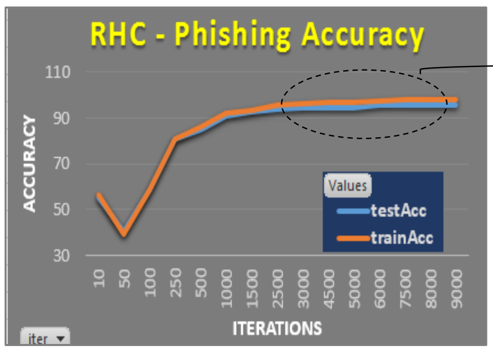


Fig-1.2a

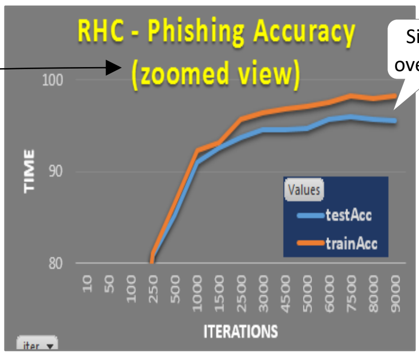


Fig-1.2b

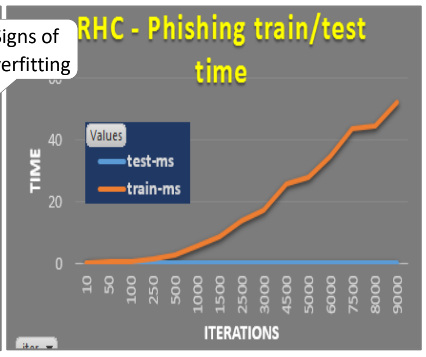


Fig-1.2c

1.3 SIMULATED ANNEALING (SA)

I explored a range of cooling rates between 0.15 to 0.95 for three different initial temperatures (1E7, 1E9, 1E11).Results were averaged over 5 runs for iteration count 10 to 8000. Results of one such set (temp=1E9) is shown below.

If cooling rate is too low, it required more iterations to achieve higher accuracy. This makes sense as when temperature is high, algorithm is in 'explore' mode, taking random walks. To allow temperature to drop sufficiently low so that algorithm can enter 'exploit' mode where optimal solution can be discovered, more iterations are required. This is evident in Fig-1.3.a where with cooling rate 0.95 (light pink line), it took 2500 iterations to cross 90% accuracy. Letting temperature to drop too quickly or too fast didn't allow algorithm to discover most optimal solution. Key is to pick a cooling rate, given an initial temperature such that 'explore' and 'exploit' phases are balanced.

For temp=1E9, optimal results were achieved at 5000 iteration with cooling rate=0.55. From Fig-1.3b, we can see that after 2500 iterations accuracy improves at snail's pace (showing signs of convergence), with higher cost of train-time. Eg: Going from 5000 to 8000 iteration, accuracy improves by 0.25% (95.38 vs 95.62) but train-time increases by 60 % (27.5 vs 44s). Moreover at higher iteration (8000), we see gap between train and test accuracy increases, showing signs of overfitting. For these reasons, I decided to stop at 5000 iteration.

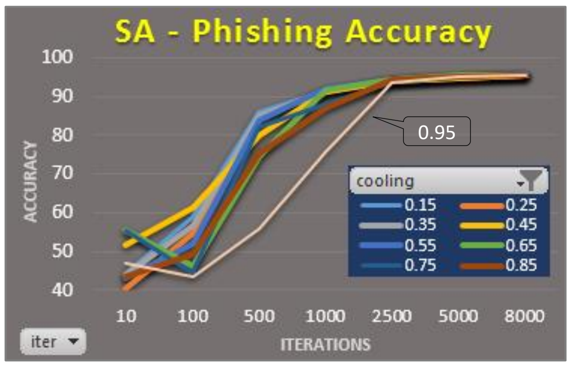


Fig 1.3a

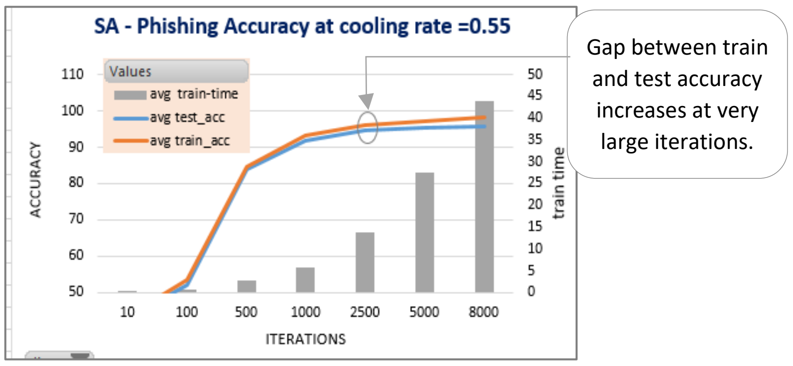


Fig 1.3b

1.4 GENETIC ALGORITHMS (GA)

ABAGAIL's implementation provides population-size, mate and mutate rate options to tune for genetic algorithm. I explored different range of these parameters to achieve the best results. I ran 3 sets of experiments where one parameter was changed while keeping other two constant. Charts below show test accuracy as function of population size, mating and mutating rate. After evaluating results (accuracy and time) from these experiments, Config '*pop_sz=200, mate=120, mutate=6, iteration=500'* gave best accuracy in reasonable time.

Insights from these experiments

a) Low mating rate didn't yield good accuracy, which intuitively makes sense as low mating rate diminishes chances of preserving and propagating good solutions to next generation. (Fig-1.4a)

b) Small population size (<100) were not ideal for this problem as it gave low accuracy. Population size should be sufficiently large to be able to capture some good solutions to begin with. On flipside, as population size increases so does train time. Hence we need to strike a right balance. (Fig-1.4b)

c) Fig-1.4c shows the impact of mutation rate on test accuracy. Very low or high mutation rate didn't yield good results. Most optimal results were seen at mutation rate =3%. Very high mutation rate can perturb the good solutions, especially during the early phases of GA run and too low can diminish diversity which is important during later phase of run. Ideally we want mate and mutation rate to dynamically adapt as run progresses.

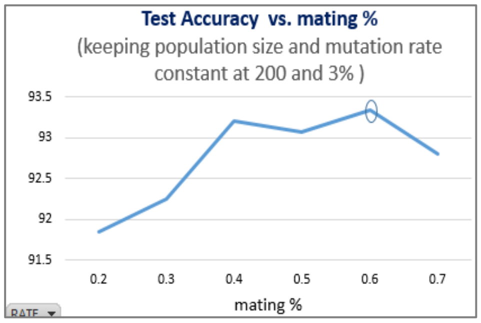


Fig-1.4a

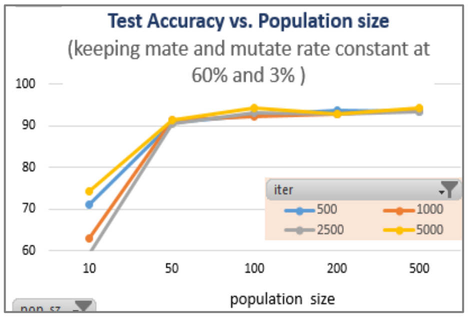


Fig-1.4b

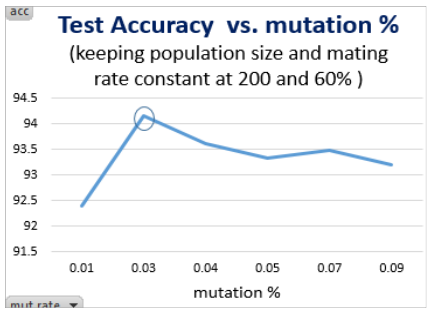
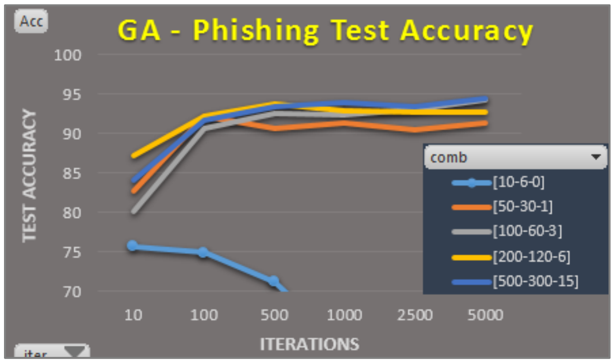
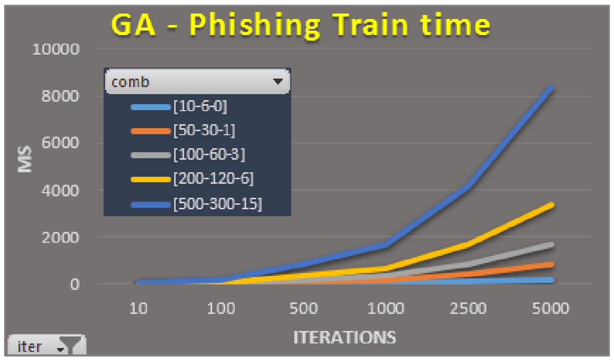


Fig-1.4c



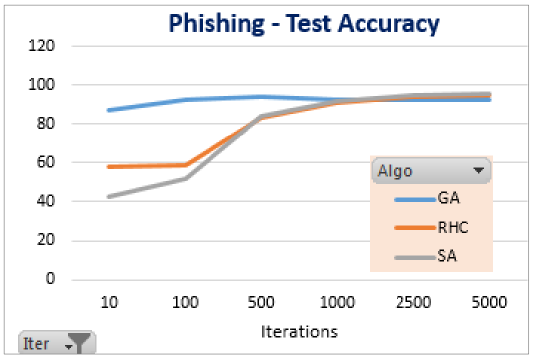


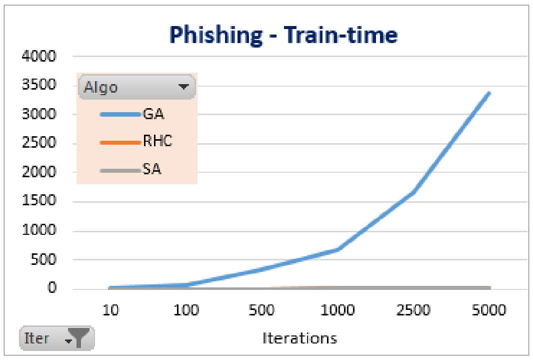
1.5 ALGORITHMS COMPARISON

Table below shows train/test accuracy and time for all three algorithms for different iteration count.

- Though GA was able to provide good accuracy for small iteration count (100-500), time taken by GA in each iteration is quite high. That's because in each iteration it performs crossover and mutation (as per the rate defined by user) and evaluates fitness of each individual in the population.
- Winner here is SA with 95.27 test-accuracy, though RHC was also very close for both accuracy (94.77) and time (27.69s).
- In terms of accuracy, I would say all three algorithms were close but it's the time-complexity where RHC and SA outshined GA.
- Fact that SA and RHC did so well may suggest that global optimum of this classification problem didn't depend on learning the structure of the problem.

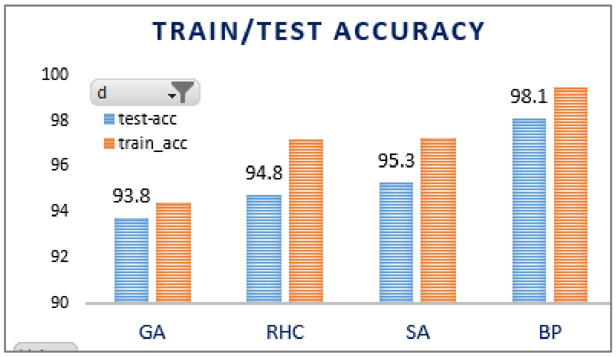
Iterations	Avg train time			Avg test-accuracy			Avg train-accuracy			Avg test time		
	GA	RHC	SA	GA	RHC	SA	GA	RHC	SA	GA	RHC	SA
10	6.72	0.09	0.07	87.16	58.08	42.98	87.85	58.87	43.50	0.004	0.016	0.005
100	66.40	0.65	0.56	92.19	58.42	52.04	92.88	57.35	53.51	0.004	0.008	0.006
500	341.42	2.74	2.80	93.75	83.42	83.99	94.42	84.27	84.71	0.005	0.005	0.004
1000	676.76	5.49	5.69	92.87	90.96	91.94	93.75	92.35	93.15	0.004	0.005	0.005
2500	1677.77	13.73	13.95	92.80	93.75	94.47	93.66	95.64	95.96	0.004	0.006	0.004
5000	3364.83	27.69	27.53	92.80	94.77	95.27	94.83	97.15	97.26	0.005	0.004	0.004

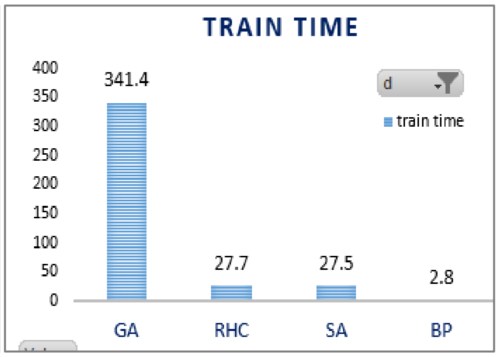




1.6 COMPARISON WITH BACK-PROPAGATION NEURAL NET (ANN-BP)

As can be seen from graphs below, neural network with BP outperformed other optimization algorithms in both accuracy and train time. *(Test time was similar for all four, hence not shown below)*





So what made BP perform better than Random Optimization algorithms for Neural Net?

Weights computed in NN take continuous values. Gradient based approach employed by Backpropagation (BP) is more suited for continuous valued space, compared to these optimization algorithm that are more suited for discrete valued parameter space.
BP is a gradient descent search algorithm, which is based on minimization of the total mean square error between actual output and a desired output. It takes advantage this error, learnt from actual labels provided in data, to guide its search in the weight space.
Also, Back Propagation (BP) had a distinct advantage that other randomized algorithms didn't have in current implementation –

a) BP made use of adaptive learning-rate and momentum. This helps in avoiding getting stuck in local minima and faster converges. This would explain lower execution time of ANN-BP.
b) In addition, ANN-BP used early stopping (adaptive iterations) which avoids overfitting and better generalization.

Part 2 – Optimization problems analysis

This section presents analysis of three different optimization problems, namely Travelling Salesmen, MaxK coloring and CountOnes, using aforementioned optimization algorithms.

What makes these algorithms desirable for these problems?

Typically, random search algorithms sacrifice a guarantee of optimality for finding a good solution quickly with convergence results in probability. All the above-mentioned problems are NP-hard and using exact-algorithm will increase the computation time exponentially as problem complexity increases. With high-quality admissible heuristics, these algorithms can find a 'good-enough' solution in a fraction of the time required compared to brute force approach.

2.1 METHODOLOGY

- Each of the above-mentioned problems was run for four different optimization algorithms.
- **Metrics** - Fitness function evaluations, fitness-score and execution time was computed for different iteration counts (100-10000) and averaged over 5 runs for each iteration to account of randomness.
- **Tuning -** Parameters described in section 1.1 were tuned to obtain the best metrics for each algorithm.
- **Criteria for best algorithm selection** – pick the one with highest fitness score and lowest execution time.

2.2 TRAVELLING SALESMAN (TSP)

Travelling salesman problem is a classic NP-hard problem and one of the most widely studied combinatorial optimization problems. Objective of this problem is to find a complete, minimal-cost tour where a salesman is required to visit each of the given cities only once. Since TSP has often been a touchstone for evaluation of different algorithms proposed to solve combinatorial optimization problem, it was an ideal candidate for this exercise.

<u>**Parameter tuning**</u>

1. GA: Selecting right population size and operator probabilities play an important role in the process of solving a problem based on GA. Knowing nothing better, I resort to a disciplined trial and error method to evaluate impact of pop size and mating and mutating rate.
    a. *Fig2.2a evaluates impact of initial population size, keeping mating and mutation rate constant. Very large population size (>5000) didn't yield good results. Optimal population size was in 200-500 range.*
    b. *Fig2.2b shows impact of mutation rate, keeping populating size and mating rate constant. I tried different mutation rates [1%,3%,5% ,7%,10%] however didn't see a discernable difference in results.*
    c. *Similarly, I tried mating rate in range [25% to 75%] and most favorable results were seen at 60%*

2. SA/MIMIC: I ran the experiment with a range of cooling rates (Fid 2.2c) and [sample#, toKeep] for MIMIC.
3. For each algorithm, parameters that gave the best results were picked.

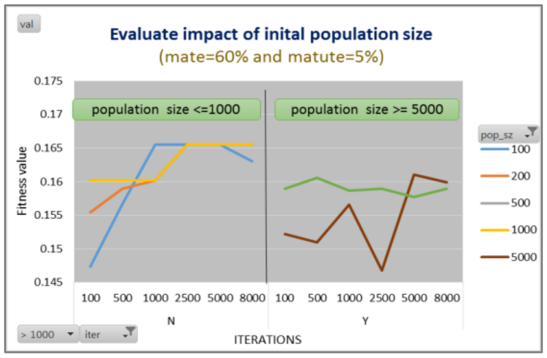


Fig 2.2a

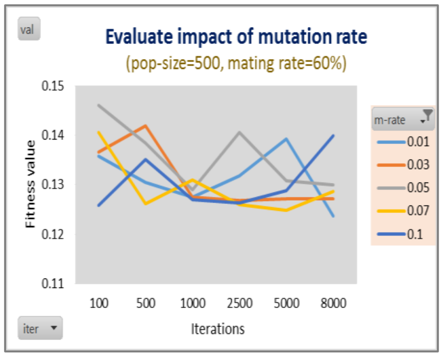


Fig 2.2b

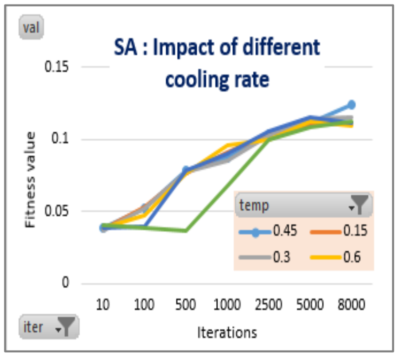


Fig 2.2c

Results

Charts below shows comparison of all four algorithm for N (number of cities) =50. GA outperformed all other algorithms for fitness value. It required more computation time compared to RHC and SA but it took much less time than MIMIC for all iteration#.GA gave best fitness value (0.159) with config [pop_sz=200, mate=60%, mutate=5%, iteration=1000] taking 0.87sec

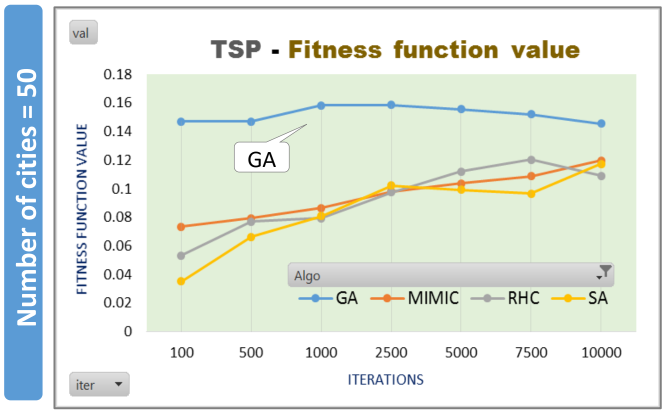


Fig 2.2d

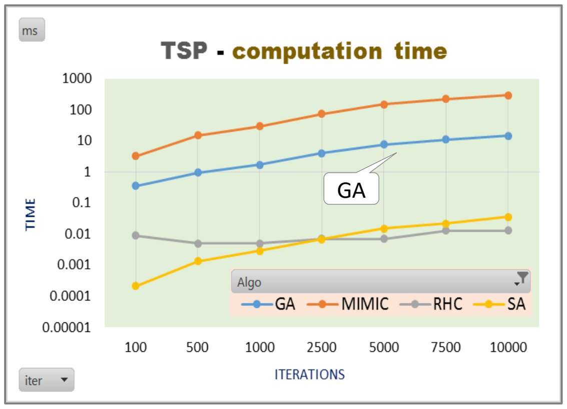


Fig 2.2e

How does Algorithm behavior change as problem complexity increases?

- Fig-2.2f below shows comparison of all four algorithms with N increased to 100. Here again GA outperforms other algorithms. Similar results were seen N=250 as well.
- Fig-2.2g shows GA execution time as number of cities grow from 50 to 100 and to 250 for different iteration count. As N increases from 50 to 100, solution space explodes from 50! to 100!, however computation time increased 2.5 folds at 1000 iteration. This highlights the power of randomized algorithms.

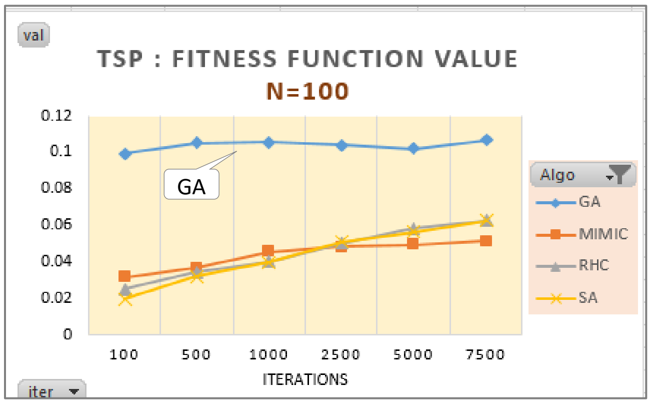


Fig 2.2f

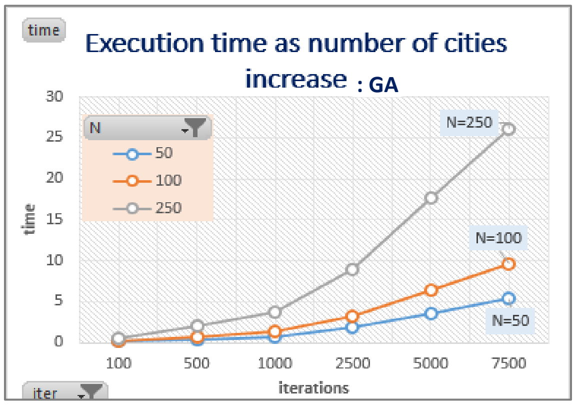


Fig 2.2g

Why GA performed best for TSP?

Success of GA for depends on how problem is represented via selection, evolution, encoding etc. Though GA doesn't know anything about the problem it is trying to solve, domain knowledge can be injected via implementing effective encoding, selection, fitness, recombination, evolution scheme and ABAGAIL must have done a really good job implementing these schemes such that it captures underlying structure of TSP.
I assume RHC and SA would be leveraging same fitness function, GA has this additional arsenal of crossover and mutation where it's able to preserve some information of problem structure from one population to other that RCH and SA can't.

2.3 MAX-K COLORING

The maximum k-coloring problem is to color a maximum number of vertices using k colors, such that no two adjacent vertices have the same color. Graph coloring is computationally hard and hence a good candidate for randomized optimization algorithm. Based on the lectures, I expect MIMIC to perform better for this problem as solution relies on the structure, rather than the values attributes take on.
In addition to parameters listed in section 1.1, there are 3 more parameters for this problem, namely number of vertices (N), adjacent nodes per vertex (L) and possible colors (K), which required to be tuned. From problem description, it is evident that number of optimal solutions will be sparse if K approaches L (ie fewer global optimas).On the other hand, there will be multiple optimal solutions if L is small but K is high. In other words, algorithms like RHC which are more likely to get stuck in local optima will have harder time with former configuration (K approaches L) as this configuration effectively creates very narrow basin of attraction. Hence even with random restarts, algorithm might not be able to converge to global optima. This is evident in the results below (Fig 2.3a).

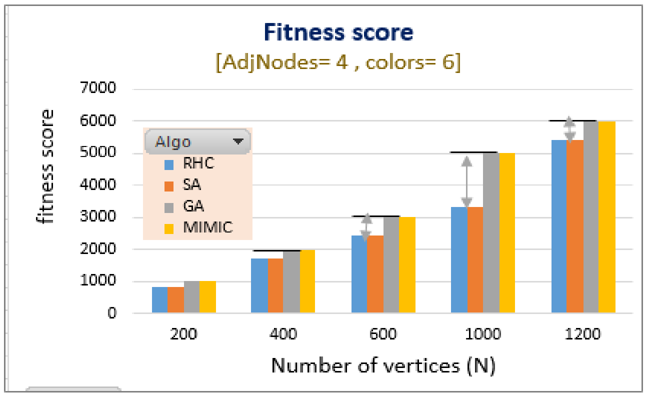


Fig 2.3a

Only configuration where RHC and SA was able to find optimal solution was when L was very low (2). I ran tests with multiple configuration of L & K [(2,2), (4,6) ,(5,5)] and vertices in range 100-1000. In all cases, RHC and SA failed to find optimal solution.
Chart on the left shows fitness scores for all four algorithms (using best parameters). Score is shown for different N (keeping L & K constant at 4 & 6).

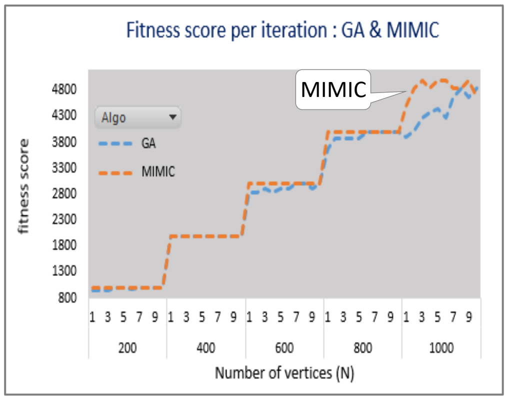


Fig 2.3b

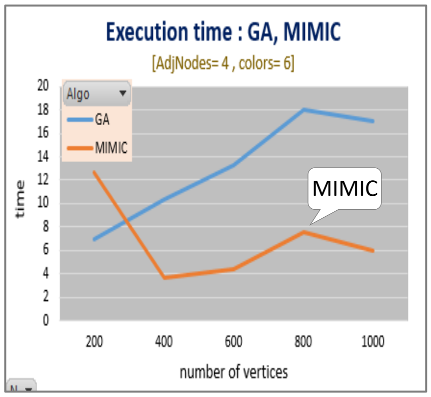


Fig 2.3c

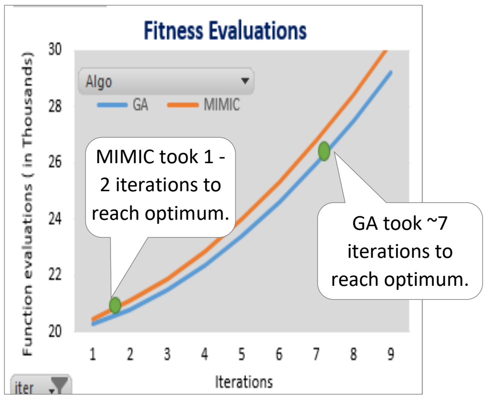


Fig 2.3d

Charts above compare performance of MIMIC and GA in terms of number of iterations and time required to converge to global optima (average of 5 runs). For N <=400, GA and MIMIC gave similar results. However, MIMIC outshined GA as problem complexity grew with large N. For N>=600, MIMIC consistently gave good results within one iteration, whereas GA took 7 iterations on average. So MIMIC is the best fit for this problem

Why MIMIC performed well for Max-K coloring problem?

For this problem color of a node is dependent on its adjacent vertices. In other words, we need to maintain a structure that captures this dependency and that's precisely is forte of MIMIC. It uses structure to guide it through solution space, rather than getting confused with the values like RHC and SA. GA was eventually able to catchup with MIMIC because notion of crossover captures structure but it took lot more iterations. Moreover, GA's performance was not consistent especially for N>=600, which alludes to '*randomness in choosing appropriate cluster that represents underlying structure*' mentioned in Prof Isbell's paper.

2.4 COUNT ONES

Objective of this problem is to search for a solution that maximizes the number of ones in given vector. Choice of this problem is based on the lecture, with expectation that SA will excel here since solution to this problem doesn't rely on the structure of the problem, but rather on the value of the input itself.

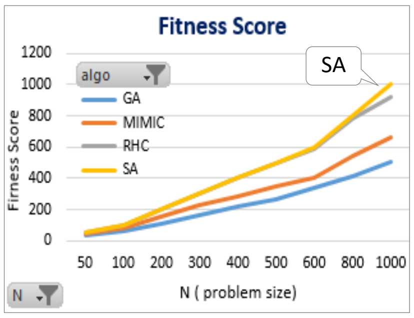


Fig-2.4a

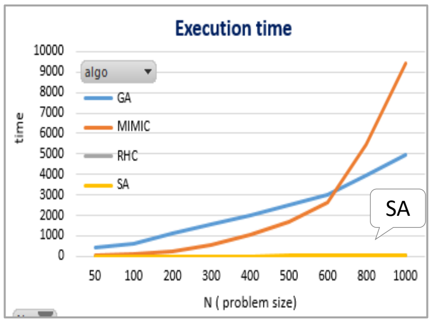


Fig-2.4b

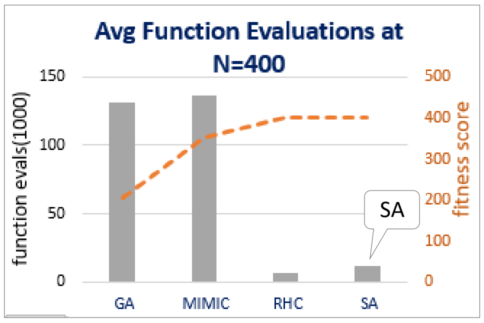


Fig-2.4c

Fig-2.4a and b above show fitness score and exectution time for all four algorithms for different values of N (vector sizes).SA performed best for this problem in terms of execution time and fitness score and GA the worst. SA and RHC were close but SA performed better for higher N .
Time taken by MIMIC and GA was very high compared to SA/RHC. I ran experiments with iterations in range 100-20000 , different permutaions and combinatoins of parameters described in section 1.1 but that didn't change the relative performance gap between algorithms. It can be seen that as problem size increases from 50 to 1000, gap between performance of GA/MIMIC and RHC/SA widens non-linearly. Fig-2.4c shows average function evaluations required by each algorithm – GA and MIMIC took 10x more fitness evaulations compared to SA and still not able to achieve higher fitness score.

Why SA performed best for count-ones problem?

Reason for this can be traced back to lectures and MIMIC paper – success of MIMIC and GA (to some extent) depends if solution of the problem depends on the structure or not. If the solution can be arrived at independent of the problem structure, as is in this case, then added complexity of MIMIC and GA is not justified and hence is not a good choice for such problem. When there is no discernable structure, neither GA nor MIMIC is not able to gain much knowledge as population/theta evolves, there by not able to arrive at optimal solution and taking lot more function evaluations.
This problem contrasts well with previous two problems (TSP and MaxK coloring) where GA and MIMIC performed well due to solution's dependency on underlying structure. But when there exists no such dependency, SA performs well. SA, due its probabilistic acceptance (of good solution) is able to avoid local minima's, which RHC can't.

Possible improvements

GA:

1. Instead of selecting initial population randomly, chose heuristically. For example, KNN can be used generate a set of relatively good tours (based on proximity) to begin with in case of TSP. This can result in better solutions in fewer iterations.

2. Use of dynamic mating and mutating rate – As Algorithm is converging, we will have similar solutions left and hence its mutation rate should be increased, while at the beginning higher crossover rate is desired.

SA:

1. Keep track of the neighborhood that have already been explored, that way same point is not visited more than once.
2. Dynamically change the number of iterations as the algorithm progresses. At lower temperatures it is important that many iterations are done so that the local optimum can be fully explored. At higher temperatures, the number of iterations can be less.
3. Acceptance criteria based on Boltzmann distribution uses exponents calculation, computationally intensive. May be use a different Acceptance criterion that approximates the exponential.

RHC:

1. Perform random restarts
2. Keep track of the neighborhood that have already been explored and to avoid in future.

Common to all:

1. Incorporate '*early-stopping*' criteria where algorithm terminates if it has converged or gain per iteration has become too low, whichever comes first. In fact this improvement is applicable for all four algorithms. *(Use ConvergenceTrainer() instead of FixedTrainer())*
2. Cost function is calculated at every iteration and hence responsible for a large proportion of the execution time.
    - One can approximate evaluation function, which can be faster than carrying out an exact evaluation of the given solution. *(desirable where cost of fitness evaluation is very high)*
    - If that's not acceptable, may be use a cache to store solutions (partial and complete) that have already been evaluated so that we can retrieve the value of the cost function from the cache rather than having to go through the evaluation function again.

Conclusion

Above experiments are reminiscent of 'No Free Lunch' seen in assignment-1 – '*there is no one algorithm that will work of all problems'*. Moreover, there is always a tradeoff between best performance and execution time. Hence it is important for us understand the nature of the problem and connect it with the strength and weakness of a given algorithm.

Each algorithm makes some assumptions (analogous to inductive bias) and when these assumptions are violated performance is sub-optimal. Eg: MIMIC assumes that by learning complex relationship between parameters, it can learn the structure of the problem and hence can arrive at optimal solution more efficiently. But what if solution doesn't depend on this interrelationship or problem doesn't have any discernable structure.

MIMIC and GA are strong candidate for problems like Max-K coloring and TSP where problem's structure can be learnt over time.

- However it's worth noting that both these algorithm make use of structure in a very different way. Random crossover and unguided mutation in GA are not always successful at detecting underlying structure of the problem. Whereas MIMIC is able to create clusters of highly correlated parameters by learning from the data itself.
- MIMIC is well suited for problems where cost of evaluating fitness function is high as it takes less iterations to arrive at optimal solution. GA is more suited for problems with high dimensions, where each dimension contribute differently to global optima.

On the other hand, we have RHC and SA which are effective at tackling problem like count-ones where optimal solution doesn't depend on learning the structure. RHC is more suited for time sensitive applications but has higher chances of getting stuck in local optima.