Assignment 2: Randomized optimization

Rohan D. Kekatpure

1. Organization of the report

The present report comprises of two parts corresponding to homework description: (1) neural network training using randomized optimization techniques and (2) comparison of randomized optimized techniques using three discrete optimization problems.

Since our origin neural network is implemented in R, we've used R for the first part. Specifically, the R libraries `GenSA` and `GA` are used for simulated annealing (SA) and genetic algorithm (GA) optimization respectively.

Unfortunately, these libraries have limited documentation for *discrete* optimization problems. So We've used the ABAGAIL library for PART 2.

2. Part 1: Neural network weights

Our neural network problem of choice is the Ionosphere problem in Assignment 1. This dataset contains 350 observations; each observation having 34 attributes. The objective is to classify each observation into a 'good' and 'bad' class. I believe the following factors qualify these datasets as 'interesting':

1. Ionosphere is a binary classification problem with 34 attributes. Its binary nature allows an easier comparative analysis of various binary classification algorithms.

2. Ionosphere data is small enough to allow us to efficiently train our models.

   For our studies, we include 150 observations in our training set and use the remaining 200 observations for testing. We don't know the train/test split use by the published studies, which might affect how closely we're able to reproduce these results.

2.1. Randomized hill climbing (RHC)

Not implemented

2.2. Simulated annealing

2.2.1. Methodology

As mentioned above, the `GenSA` package in R was used to implement the SA algorithm. The neural network itself was implemented using the `nnet` package. Our network had 1 hidden layer and 10 hidden units. The `GenSA` package provides the `GenSA()` function which takes a vector of inputs as the first value and a callable (`evalfunction`) as the second argument. Our `evalfunction` was the squared error objective function value of the neural network. To implement `evalfunction` we asked the `nnet` package to simply evaluate the objective function without any training iterations.

2.2.2. Results

Figure 1 shows the minimization of the objective function performed via the SA algorithm. The cost function value, after about 66,000 function evaluations has saturated at about $f_{\min} = 71.1$. The optimization run took about 235 seconds and final training and the test error were about 1.49% and 7.94% (average of 10 runs).

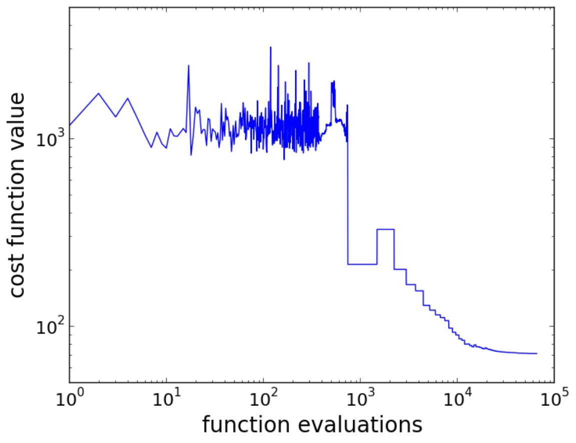

Fig. 1. Convergence of simulated annealing optimization. Initial temperature was 200.

2.3. Genetic optimization

2.3.1. Methodology

The general methodology was similar to §2.2.1. For the GA algorithm, we used the `GA` package in R. The only difference here was that since the `GA` package performs maximization, the cost function needed to be negated.

2.3.2. Results

Figure 2 shows the best results for neural network cost function minimization performed via the GA algorithm with local arithmetic (LA) crossover. After about 240,000 function evaluations, the final function value of $C_{\min} = 88.53$ was obtained. This corresponded to a final training and test errors of 1.99% and 16.56% respectively. The wall clock time for the optimization problem was 1200 seconds.

Compared to simulated annealing, we found it was more difficult to optimize the weights using GA. The performance above was the best we could obtain and it was after experimenting with four different crossover functions (described below) and a range of mutation probabilities.

The `GA` package in R provides a choice of four cross-over functions for real-value encoded GA problems. Below are the results of our experiments with each of the cross-ver methods. First we provide the results and analyze them together with simulated annealing in the next section.

1. Single-point crossover: This is the standard single-point method where two children (c_1 and c_2), each getting a contiguous part of the bit value of weights of two parents (p_1 and p_2).

$$c_1 = p_1[1:k] + p_2[k+1:n]$$
$$c_2 = p_2[1:k] + p_1[k+1:n]$$

   where the operator '+' is overloaded to mean string concatenation.

   Weights generated using a single-point crossover were suboptimal. The minimum cost function value obtained after varying the mutation probability from 10% to 99% was $f_{\min} \sim 506$. Compare that with $f_{\min} \sim 70$ obtained using backprop and simulated annealing. One possible reason for the bad performance of single-point crossover GA is that perhaps this crossover method is not well suited for real-valued features. After all what is the interpretation of generating floating points by crossing over bit patterns? In fact, for small weights ($\simeq 1.0$), a half-way single-point cross often (but not always) produces children that are identical to parents. This fact can be verified by the Python script in the Appendix. This greatly undermines the crossover operation. Despite these limitations, the `GA` package provides it as an option for real-valued features.

2. Local arithmetic (LA) crossover: This crossover method generates two children from two parents by altering only a few features (co-ordinates) of the parents. The alternation consists of performing a linear combination of parent features. For example if the two parents are:

$$\begin{aligned} p_1 &= (x_1, x_2, \ldots, x_n) \\ p_2 &= (y_1, y_2, \ldots, y_n) \end{aligned}$$

   then the children are

$$\begin{aligned} c_1 &= (x_1, x_2, \ldots, \alpha x_k + (1-\alpha) y_k, \ldots, x_n) \\ c_2 &= (y_1, y_2, \ldots, \alpha y_k + (1-\alpha) x_k, \ldots, y_n) \end{aligned}$$

   For the LA crossover, I went with the default settings and varied mutation probability between 10% and 99%. We obtained best performance with LA crossover with `pmutation` = 99%. Figure 2 represents the 240,000 iteration of the genetic algorithm with LA crossover.

3. Whole arithmetic (WA) crossover: The WA crossover is similar to LA crossover except the entire parent vector is used.

$$\begin{aligned} p_1 &= (x_1, x_2, \ldots, x_n) \\ p_2 &= (y_1, y_2, \ldots, y_n) \\ c_1 &= \alpha p_1 + (1-\alpha) p_2 \\ c_2 &= (1-\alpha) p_1 + \alpha p_2 \end{aligned} \tag{1}$$

   WA crossover worked significantly better than single-point crossover, but 50% as worse compared to LA crossover. The cost function minimum obtained using WA crossover at `pmutation`=99% is $f_{\min} = 150$. It appears that big deviations from parents don't lead to a better search performance.

4. Blended crossover (BLX): Blended crossover expands the search boundary and adds a mutation-like randomness. Assume the parents, like before, are n-component real-valued feature vectors.

$$\begin{aligned} p_1 &= (x_1, x_2, \ldots, x_n) \\ p_2 &= (y_1, y_2, \ldots, y_n) \end{aligned}$$

   Define

$$\begin{aligned} d_i &= |x_i - y_i| \\ m_i &= \min(x_i, y_i) - \alpha d_i \\ M_i &= \max(x_i, y_i) + \alpha d_i \end{aligned}$$

Components x_i and y_i for the next generation are chosen uniformly randomly from the interval $[m_i, M_i]$.

BLX crossover performed slightly better than single-point crossover but significantly worse then WA and (therefore) LA crossovers. Cost function minimum obtained using BLX crossover at `pmutation` = 99% is $f_{\min} = 350$.

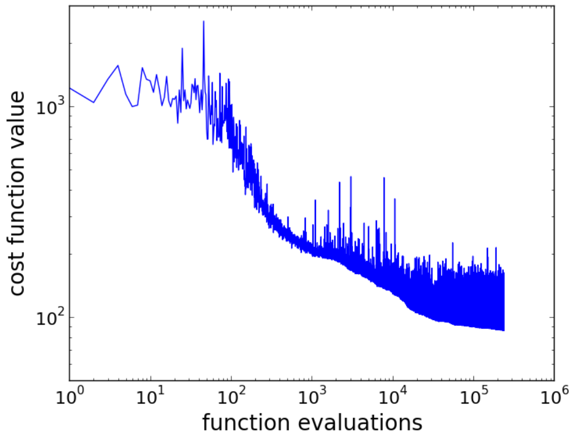

Fig. 2. Convergence of genetic algorithm with LA crossover and `pmutation` = 99%.

2.4. *Summary and discussion*

Table 1. Comparison of simulated annealing and genetic algorithms with different crossover functions.

Algorithm	**Training time**	$f_{\min}$	**Training error**	**Test error**
Backpropogation	10 s	70.5	1.49%	6.5%
Simulated annealing	235 s	72.8	1.49%	7.9%
GA, SP crossover	1200 s	450	23.4%	45.2%
GA, LA crossover	1200 s	88.5	1.99%	16.5%
GA, WA crossover	1200 s	168.3	13.4%	32.4%
GA, BLX crossover	1200 s	345.2	20.1%	39.5%

In general, randomized optimization approaches take significantly longer computation times compared with direct approaches like backpropagation. The obtained solutions are also suboptimal. However, memory or computational constraints might mandate the use of randomized optimization algorithms.

Simulated annealing (SA) algorithm achieved almost the same $f_{\min}$ and test error as Backpropagation, but took almost a hundred times longer to compute because of random search. One technique to reduce the run time is to apply a faster cooling rate. Unfortunately the `GenSA` package does not seem to allow altering the cooling rate. The documentation [1] has no mention of this control parameter. One of the publication by the `GenSA` team [2] mentions that the cooling

[1]https://cran.r-project.org/web/packages/GenSA/GenSA.pdf

[2]https://journal.r-project.org/archive/2013-1/xiang-gubian-suomela-etal.pdf

rate can be altered by passing the argument, but there is no example in the documentation or the general web showing how to do that. As a result I was unable to experiment with this parameter. Another technique might be to cache function values to avoid repeated function computations. However, it is unclear how we can do that since we're dealing with a multidimensional real values feature vector at machine precision (e.g. evaluation is requested at $x = 1.5$ but value in lookup table is at $x = 1.45$. Lookup is even less beneficial in multiple dimensions.)

Our SA experiments were therefore limited to varying the initial temperature. All of these yielded approximately similar performance indicating that there aren't any faraway maxima.

Genetic algorithms performed worse by comparison both when evaluated by computation time as well as final results. We experimented with two parameters: (1) four different crossover functions and (2) varying mutation probability from 0.10 to 0.99. A mutation probability close to 1 was *necessary* to obtain a minimum function value; without that the results were much worse than reported in Table 1. Local arithmetic (LA) crossover provided best performance.

Mutations in genetic algorithms allow for searching a small area around the current location by providing a small random wiggle room around the current location. Crossovers, on the other hand, provide an ability to make large leaps somewhere 'in between' the parents. Also, for real-valued inputs, mutation adds random new information to the solution, whereas crossovers combine already available information.

Taking both the above observations into account, it seems that our neural network cost function has relatively sharp minimums, which can be captured only if our optimization algorithm has ability to perform random search around in its current vicinity. This reasoning can also be applied to understand the variation in the performance of different crossover functions. The single-point crossover does not result in different offspring for small floating-point inputs. The LA crossover makes very small alterations to the parent genes and therefore results in children that resemble the parents closely. WA and BLX crossovers, on the other hand, make large changes to parent genes and result in children far off from their parents, preventing local search.

3. Part 2: Three discrete optimization problems

3.1. Algorithm evaluation method

In part 2 of the assignment, we study three algorithms, each highlighting strengths of genetic algorithm (GA), simulated annealing (SA) and MIMIC. Different aspects can be stressed in evaluating the strength of an algorithm and defining a figure of merit for its measurement. One can consider the achieved final (maximized) value of the objective function, $f_{\max}$. However, this ignores the cost in terms of time. After all, given infinite time, even a random hill climbing algorithm can find the global maximum. Therefore one also needs to consider the runtime of the algorithm. Runtime can be impacted by two independent factors. Either the function could be very expensive to evaluate or the function landscape could have a lot of local minima which necessitate a lot of iterations from some 'naive' algorithms such as GA and SA. 'Naive' here is used in the sense of throwing away information from function evaluations.

If the total runtime is limited by expensive function evaluations, a general improvement strategy may be to use a method like MIMIC which extracts more information from function evaluations. If the total runtime is limited by cost function landscape then it might be more effective (time wise) to use relatively faster methods like SA or GA.

To evaluate the various algorithms, we'll plot $f_{\max}$ vs runtime. The number of iterations is the implicit parameter in all these plots since each value of runtime and $f_{\max}$ is obtained by varying the number of iterations. In this scheme, the best algorithm is that which obtained largest $f_{\max}$ in least amount of runtime. So on a graph of $f_{\max}$ on the y axis and runtime on the x axis, the best algorithm would be on the top left (max value in min time).

The first two problems (count ones and Rastrigin function) demonstrate strength of the Simulated annealing method, the next two (traveling salesman and four peaks) demonstrate the strengths of MIMIC and the last problem highlights the strength of genetic algorithm.

3.2. *Count ones*

The count ones is an already implemented algorithm in the ABAGAIL package. The input to this function is bit strings and the objective function value is simply the number of 1's in the string. This function is easy to evaluate, but not necessarily easy to optimize. For our test we used 100 bit strings so $f_{\max} = 100$. Figure 3(a) shows the results of running the three algorithms on the CountOnes problem. It is clear that simulated annealing outperforms the other algorithms for this problem. The reasons are exactly as mentioned previously: CountOnes objective function is easily evaluated and trading off number of function evaluations for costlier sampling in MIMIC is not warranted.

3.3. *Rastrigin test*

Rastrigin test problem can be thought of as a continuous as well as a discrete optimization problem. Here we evaluate is as a real-valued cost function of discrete inputs. It is defined as:

$$\begin{aligned} x &= (x_1, x_2, \dots, x_n) \\ R(x) &= nA + \sum_{i=1}^{n} \left[x_i^2 - A\cos(2\pi x_i) \right] \\ f(x) &= B - R(x) \end{aligned}$$

$R(x)$ is the actual Rastrigin function (As defined in the Wikipedia). We've modified it by adding the last equation for $f(x)$ which merely converts a minimization problem into a maximization. Our modified Rastrigin function, as defined above, has a global maximum at $x = (0, 0, \dots, 0)$ with the value $f_{\max} = B$.

For our experiments, we used $n = 100$ and $A = 10$. Rastrigin function has a relatively shallow global maximum and is surrounded on all sides by local maximums. As such, it is a difficult optimization problem. However, the function evaluation itself is not particularly costly. Because of this characteristic, simulated annealing once again outperforms MIMIC as seen in Figure 3(b). Note that both MIMIC and SA solve the problem (i.e. achieve $f_{\max} = 100$, but SA achieves it 100 times faster. Notably, in our experiments, genetic algorithm is unable to solve the problem with either a single-point crossover or a uniform crossover. Upon monitoring the GA solution, we found that the algorithm gets stuck in one of the local maxima, whose location is dependent on the choice of initial position.

Our next two optimization problems illustrate the strengths of MIMIC.

3.4. *Traveling salesman*

Travaling salesman cost function is costly to evaluate. As such, it is chosen so that SA and GA techniques, which require relatively more function evaluations, would not be able to solve it in our number of allowed iterations. This is exactly what we observed in our experiments. As seen in Figure 3(c), MIMIC clearly outperforms SA and GA in finding the maximum value. Both the latter algorithms are unable to find the optimum solution in the maximum allotted 5000 iterations. SA and GA find similar (sub)optimum values but SA reaches there faster.

3.5. *Four peaks*

We used a four peaks (FP) problem with an array of dimension $N = 200$ and $T = 20\%$. The four peaks problem illustrates strengths of MIMIC in a different way. While the FP function itself is not particularly costly to evaluate, it has two broad basins of attraction which tend to trap SA and GA in local maxima. The regions of sample space that MIMIC eliminates in every iteration prove crucial to its ability to reach the *global* maximum. As Figure 3 shows, in our allotted 5000 iterations, none of the algorithm has achieved $f_{\max}$, but MIMIC is close to achieving the theoretical maximum of $f_{\max} = 240$.

3.6. *Knapsack problem*

The knapsack problem we chose has 40 items with 4 copies each and the maximum weight is restricted to 50. For the knapsack problem, all three algorithms perform almost equally. However, GA performs *slightly* better, as seen from Figure 3(e). However, this advantage seems marginal when compared to its increased time in comparison to SA.

This is a general observation about SA and GA. We were hard-pressed to come up with problem in which GA outperformed SA in terms of runtime and convergence rate. Research articles comparing the two algorithms also conclude similarly (paper).

One reason for a lack of clear benefit for GA may be that there is no obvious reason why GA's mutation + crossover strategy for 'escaping' local optimums is better than SA. Also the choice of mutations and crossover functions provide more knobs to GA, which must be precisely tuned (much like hyperparameters). Any suboptimal choice of mutation and crossover functions can severely impact performance of GA. In spite of this, there is no systematic procedure for scanning though different mutation and crossover functions.

In our present studies, we retained the default mutation strategy, but experimented with uniform and single-point crossover. However, the choice of crossover function did not provide any visible benefit to the problem solution.

3.7. *General observations*

From Figure 3 it seems that if a problem is solvable for SA, it will be the fastest one to solve it. However, throwing more runtime at SA (i.e. increasing the number of iterations) does not necessarily lead to an improved performance for SA. In that sense it is stochastic. This is because SA is sampling the entire space for every iterations, including the regions where the function value is not likely to be optimum.

MIMIC, on the other hand, approaches the optimum value much more systematically. This is built into the algorithm. It successively eliminates the 'suboptimal' regions of the search space and retains only the regions where the optimum will be found. As a result, with MIMIC, we can be confident that any extra runtime thrown at the algorithm is likely to be beneficial. This is an important confidence to have in an algorithm.

Appendix: Python script to perform single-point crossover

```python
import struct
def float_to_bits(fnum):
    hexnum = struct.pack('>f', fnum)
    bits = ''.join([bin(ord(c))
              .replace('0b', '')
              .rjust(8, '0')
              for c in hexnum])
    return bits

def spxover(f1, f2, k):
    f1_bits = float_to_bits(f1)
    f2_bits = float_to_bits(f2)
    c1_bits = f1_bits[:k] + f2_bits[k:]
    c2_bits = f2_bits[:k] + f1_bits[k:]
    return f1_bits, f2_bits, c1_bits, c2_bits

if __name__ == '__main__':
```

```python
f1 = 1.0
f2 = 2.0
f1b, f2b, c1b, c2b = spxover(f1, f2, 16)
if f1b == c1b:
    print "f1b == c1b"

if f2b == c2b:
    print "f2b == c2b"
```

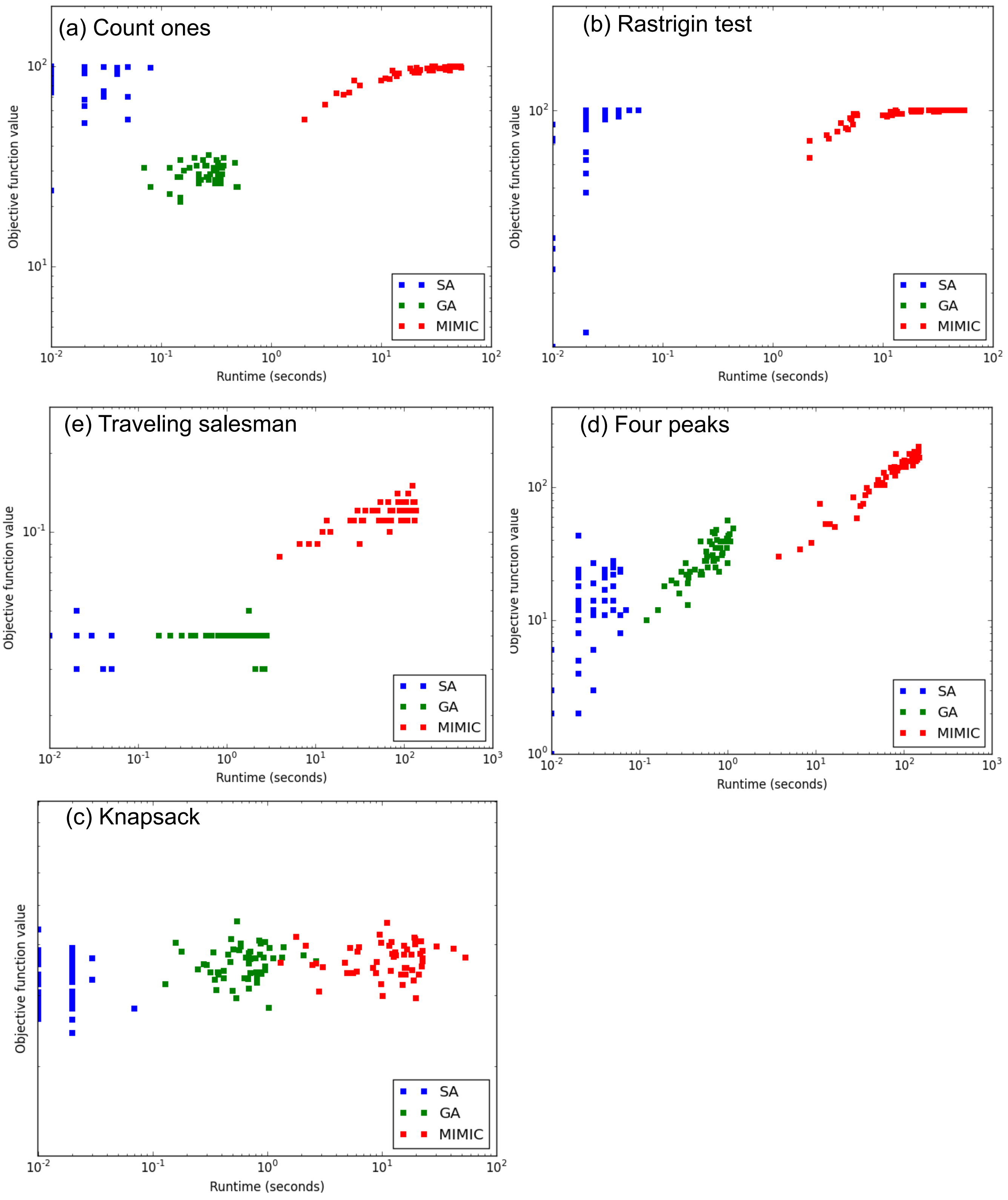

Fig. 3. Maximization of various discrete functions described in the text: (a) count ones (b) Four peaks (c) Knapsack (d) Rastrigin and (e) Traveling salesman. The blue, green and red plots correspond to results of simulated annealing, genetic algorithm and MIMIC respectively. The results of genetic algorithm are not visible in part (d) because of the scale used.