Benchmarking Estimation of Distribution Algorithms

Patrick Stephen

November 13, 2017

1 Abstract

Eight Estimation of Distribution Algorithms were tested against six different bitstring fitness problems under strict conditions to see which methods would perform best in terms of time spent searching for solutions and the quality of solutions reached. The results showed that Stochastic Hill-Climbing with Learning by Vectors of Normalized Distributions was unable to perform on problems of large size, and Population-Based Incremental Learning, one of the most basic of EDAs, performed as well as the two other top performers, Mutual Information Maximizing Input Clustering and the Bivariate Marginal Distribution Algorithm, both bivariate EDAs. Among other tested algorithms, the compact Genetic Algorithm showed a propensity to find simple optima fast, and the Bayesian Optimization Algorithm showed promise in (slowly) always improving in fitness.

2 Introduction

Estimation of Distribution Algorithms (EDAs) are a meta-heuristic approach which evolve an explicit probability distribution for a given optimization problem over time. How this specific probability distribution is evolved varies by the given model, some of which construct Bayesian networks, Markov trees, or treat all variables as independent. EDAs combine work in probability with work in evolutionary computation, and have been shown to outperform similar approaches that don't use use explicit representations. [1] In this project, eight different EDA methods were developed in a unified framework as an extension on an existing evolutionary framework for Genetic Algorithms and Genetic Programs.[1]

Six different problems were fed into these eight methods, and this paper will detail the results of those runs in terms of speed to reach an optimal solution, average time taken per generation, and best solution reached for each method on each problem.

3 EDA Methods

All the implemented EDA methods are passed into this function as the variable eda:

```
func Loop(eda EDA, opts ...Option)
     (Model, error) {
  model, err := eda(opts...)
  if err != nil {
        return nil, err
  }
  for model.Continue() {
        model = model.Adjust()
        model.Mutate()
        (*(model.BaseModel().iterations))++
  }
```

[1]The code for this can be found at bitbucket.org/StephenPatrick/GoEvo. The relevant sub-packages where work was done for this project are eda and mut, which did not exist prior to this project, and env, which had significant modifications made to it as a part of this project. A separate repository at github.com/200sc/go-dist was also split off from goEvo, but most of the work in that package had already been done prior to starting this project.

```
  return model, nil
}
```

Where model.Continue() is the following:

```
func DefContinue(m Model) bool {
  b := m.BaseModel()
  return b.Fitness() > b.goalFitness &&
        *b.iterations < b.maxIterations
}
```

model.Fitness() returns the fitness of an underlying vector, which all models have, passed into a fitness function. Each problem definition tested in these experiments is coded as a fitness function on a vector. The fitness functions tested in this experiment are all fitness functions on bitstrings, and so each model is restricted so that each vector position holds something between 0 and 1, or in the case of samples, just 0 or 1[2]

model.Mutate() is method dependent, but is always a mutation of all of the indices of a vector or some set of vectors, with each index having some percent chance of being mutated. Mutation is hypothetically optional, and isn't touched on in the papers detailing any of these models in depth, but without sufficient mutation every model tested in this set of experiments failed to evolve anything worthwhile. Mutation is necessary to maintain a diverse population in any evolutionary method, and Mutation should serve both as a way to make many small changes in the composition of a population and rare large changes, to avoid local optima.

The remaining principles of a usual evolutionary algorithm, Selection and Crossover, are built into each model.Adjust(). Crossover is the more interesting of the two methods to discuss in relation to EDAs, as in principle EDAs remove the need for Crossover by representing the result of an infinite number of Crossover operations within a population through the probability model that they build. Selection, on the other hand, is usually handled classically, on occasion greedily instead of using a selection algorithm that tries to maintain diversity. All models which do greedy selection could probably be improved by using a non-greedy selection, but then they would no longer be the models we are seeking to judge.

In each following sub-section we'll detail abbreviated code for each model's Adjust method with descriptions, for each model implemented within the constraints of this project.

3.1 Univariate Models

Univariate models assume that the different indexes of a bitstring are all independent. There's little in the way of probability methods here, and these models are just different ways of approaching the idea of abstracting Crossover into a vector of probabilities.

3.1.1 Univariate Marginal Distribution Algorithm (UMDA)

```
func (umda *UMDA) Adjust() Model {
  p := umda.Pop()
  subPop := umda.SelectLearning(p)

  newenv := env.NewF(umda.length, 0.0)
  for _, ind := range subPop {
        for j, f := range *(ind.(*EnvInd).F) {
          newenv.Set(j, newenv.Get(j)+*f)
        }
  }
  newenv.Divide(float64(len(subPop)))
  newenv.SubF(umda.F)
  newenv.Mult(umda.learningRate)
  umda.F.AddF(newenv)

  return umda
}
```

UMDA's Adjust method generates a population based on its vector of floats, where each individual is a bitstring and each index of the bitstring is generated through a random roll

[2]The only exception to this is SHCLVND, which does not use bitstring samples but probability vector samples.

against the value in umda's vector. Selection is performed on this population, and a vector is generated as the average of the selected bitstrings. The difference between that vector and the UMDA's vector is then added to UMDA's vector at a predefined learning rate. [2]

All tests with UMDA were run with a learning rate of 0.2 and a mutation rate of 3%.

3.1.2 Population-Based Incremental Learning (PBIL)

```
func (pbil *PBIL) Adjust() Model {
  bcs := NewBestCandidates(pbil,
          pbil.learningSamples, nil)
  r := pbil.learningRate/float64(bcs.Length)
  pbil.F.Reinforce(r, bcs.Slice()...)
  return pbil
}
```

PBIL's Adjust method greedily picks some best samples from sampling it's underlying vector and reinforces that vector towards those samples each at an equal portion of PBIL's defined learning rate. PBIL is then very similar to UMDA, with a minor difference in how they learn from their samples and with PBIL's selection being potentially greedier than UMDA's.[3] [4]

All tests with PBIL were run with a learning rate of 0.2 and a mutation rate of 3%.

3.1.3 compact Genetic Algorithm (cGA)

```
func (cga *CGA) Adjust() Model {
  bcs := NewBestCandidates(cga,
          cga.samples, nil)
  bcs.Front.F.SubF(bcs.Back.F)
  bcs.Front.F.Mult(cga.learningRate)
  cga.F.AddF(bcs.Front.F)
  return cga
}
```

cGA's Adjust method is very similar to PBILs, with the difference being that cGA learns the difference between it's worst and best sample instead of just learning from its best samples. [5]

All tests with cGA were run with a learning rate of 0.1 and a mutation rate of 3%.

3.1.4 Stochastic Hill-Climbing with Learning by Vectors of Normal Distributions (SHCLVND)

```
func (shc *SHCLVND) Adjust() Model {
  bcs := NewBestCandidates(shc,
          shc.learningSamples, shc.SigmaSample)
  mid := env.AverageF(bcs.Slice()...)

  mid.SubF(shc.F)
  mid.Mult(shc.learningRate)
  shc.F.AddF(mid)
  shc.Sigma = shc.lmutator(shc.Sigma)
  return shc
}
```

SHCLVND looks like halfway between PBIL and UMDA other than how SHCLVND samples new members. Instead of directly basing samples from an underlying vector, SHCLVND's samples are randomly offset from that underlying vector at rate σ, using a Box-Mller transform. σ is also reduced over time at a rate based on the original paper, so after each iteration $\sigma = \sigma * .97$.[4] [6]

Experiments showed that SHCLVND would not learn a better model with low mutation or high learning rate, which suggests that the mutation is doing most of the work for SHCLVND. All tests with SHCLVND were run with a learning rate of 0.05.

[3] Two other variants were considered for implementation but found to be functionally equivalent to PBIL, those two being IUMDA or Incremental UMDA and the very first considered EDA, the Equilibrium Genetic Algorithm. [3]

[4] This is also an option for learning rates in general for all of these algorithms, but no real gain in best fitness or speed of best solution found was found through adding a decay to the learning rate of any algorithm. This sigma decay stays because it is detailed by the paper itself as a part of the algorithm, and not as an optional modification.

3.2 Bivariate and Multivariate Models

Bivariate and Multivariate models assume either that each bitstring index is dependent on at most one other bitstring index, or that each bitstring index can be dependent on any number of other bitstring indices. These models are all of a similar complexity, however, and the multivariate models among them (EcGA, BOA) limit themselves in ways to prevent dependencies beyond bivariate dependencies from occurring frequently, so these are grouped together.

3.2.1 Extended compact Genetic Algorithm (EcGA)

```
func (ecga *ECGA) Adjust() Model {
  ecga.P = ecga.ECGAPop()
  selected := ecga.SelectLearning(ecga.P)
  ecga.F.SetAll(0.0)
  for _, s := range selected {
    ecga.F.AddF(s.(*EnvInd).F)
  }
  ecga.F.Divide(float64(len(selected)))
  ecga.MDMModel(selected)
  return ecga
}
```

EcGA's Adjust method creates a population based on the EcGA's existing population and building blocks, where building blocks are sets of vertex indices that EcGA has found should be assigned the same value in samples. The members of this population are sampled copying all values of a given building block in a random existing member to the new sample, until all blocks have been copied.

EcGA then replaces the lowest fitness members of its existing population with the new members created. EcGA updates its underlying vector for the purpose of judging that vector's fitness, then updates its building blocks based on the new population. This update process checks all pairs of blocks, starting with all blocks holding a single index, and calculates a complexity value that estimates how useful merging the two blocks would be relative to a standard punishment for large blocks. The pair with the best complexity value (if any pair should be merged) are merged until nothing would be gained by a merge.[5] [7]

All tests with EcGA were run with a learning rate of 0.2 and a mutation rate of 1%[6].

3.2.2 Mutual Information Maximizing Input Clustering (MIMIC)

```
func (mimic *MIMIC) Adjust() Model {
  fitnesses, samples :=
    SampleFitnesses(mimic, mimic.NSamples())

  ti := int(float64(mimic.samples)*
            mimic.learningRate)
  thetaFitness := fitnesses[ti]
  fi := len(samples)
  for i, f := range fitnesses {
        if f > thetaFitness {
          fi = i
          break
        }
  }
  filtered := samples[0:fi]
  mimic.UpdateFromSamples(filtered)
  mimic.PTF.Mutate(mimic.mutationRate,
                   mimic.fmutator)
  return mimic
}
```

MIMIC's Adjust method creates samples, sorts them and updates itself based on the samples which have a fitness better than or equal to some percentile of the sample set. These samples use two vectors and an index order, where the index order defines a conditional chain through all of MIMIC's indices. An index chain of [0,1,2,...] would imply that values at index 0 are independent, 1 is dependent on 0, 2 dependent on 1, etc. The two vectors then each store conditional probabilities (or univari-

[5] I would note, based on this explanation, that I'm not sure EcGA is a good name for this method– it doesn't seem anything like cGA.

[6] For all methods like EcGA which keep a persistent population around, mutation is applied to all members of their population instead of just on their underlying vector, and as a result they tend to need lower mutation rates.

ate probabilities for the single independent index), one stores $P(A|b = 1.0)$, and the other (PTF in the above code) stores $P(A|b = 0.0)$. Samples are generated by iterating through the index chain to find out what the probability of the next index should be. [8]

The index chain itself is generated through iteratively finding the index with the minimum conditional entropy given the previous index.

All tests with MIMIC were run with a learning rate of 0.07.

3.2.3 Bivariate Marginal Distribution Algorithm (BMDA)

```
func (bmda *BMDA) Adjust() Model {
  // Create a dependency forest
  // ... omitted, output: roots, children
  tenv := env.NewF(bmda.length, 0.0)
  fenv := env.NewF(bmda.length, 0.0)
  samples := PopEnvs(bmda.LastPop)

  for _, root := range roots {
    tenv.Set(root,
      UnivariateFromSamples(samples, root))
    bmda.SetChildren(samples, tenv, fenv,
      children, root)
  }
  newPop := bmda.NSamples(
          bmda.learningSamples, tenv, fenv,
          roots, children)

  bmda.ReplaceLowFitnesses(bmda.LastPop,
                           newPop)
  bmda.UpdateFromPop()
  // Mutate bivariate vectors (omitted)
  return bmda
}
```

BMDA's Adjust method generates a dependency forest and uses that forest to update its population just like EcGA updated its population, by replacing the lowest fitness members with sampled members.

The generation of the dependency forest uses a χ^2 function on each pair of indices in the bitstring length, where the value represents how likely it is that the two indices are independent versus one being dependent on the other. So long as indices are dependent on one-another, they are chained into the dependency forest by being labeled as the children of whatever they are dependent on.

To sample the dependency forest, a BMDA stores the conditional probability of every index in the bitstring length given any other index, as well as the univariate probabilities. Roots of the forest are based on the univariate probabilities and remaining indicies use the forest to determine what conditional probability they should be based on.

Following the sampling and population updating, the new population is used to recreate the mentioned conditional probability pairs. [9]

All tests with BMDA use a mutation rate of 25%.

3.2.4 Bayesian Optimiziation Algorithm (BOA)

```
func (boa *BOA) Adjust() Model {
  selected := boa.SelectLearning(boa.P)
  boa.F.SetAll(0.0)
  for _, s := range selected {
    boa.F.AddF(s.(*EnvInd).F)
  }
  boa.F.Divide(float64(len(selected)))

  envs := MemberEnvs(selected)
  bn := NewBayesNet(envs)

  sampleCt := int(boa.learningRate
                  *float64(boa.samples))
  samples := bn.Sample(envs, sampleCt)
  boa.ReplaceLowFitnesses(boa.P, samples)
  return boa
}
```

BOA's Adjust method samples from a persistent population, then creates a Bayes Network based on those samples. As in previous methods, samples from that network then replace the low fitness members of the persistent

[7] This approach would be referred to as elitism in a standard Genetic Algorithm (GA), but in this case the elitism factor is much higher than it would be in a usual GA, and there's no method preventing elites from being mutated, as you would expect in a GA. It's possible that standardizing this as real elitism and preventing the top n members from mutation would improve learning.

population.[7]

The creation of the Bayes Network is similar to the creation of the BMDA forest, where the best directional connection is made so long as there are good connections to make, maintaining that there are no cycles in the network and that no parent has more than two children. The function which judges the quality of a potential connection is the K2 algorithm.

Samples are taken from the network by ordering the indices by the network's nodes topographically and evaluating each index in order based on what has already been set in the new sample. [10]

All tests with BOA use a mutation rate of 3% and a learning rate of 0.1.

4 Experiment Parameters

Aside from the specific parameters listed above for each method, which were discerned through manual testing until a set of good, but potentially sub-optimal, parameters were found for all problems, each test run was given a set of parameters that was identical for every method. These parameters were identical to judge how the methods compare to each other when given the same tight restrictions on how well they could learn. For example, it's almost always better to generate more samples each generation and learn from more samples per generation, but we limit these values to 100 and 10 respectively for each method. There are some parameters in our defaults that may be overwritten by an individual method, in which case their customized value is used above the below default:

- Samples per generation: 100
- Learning Samples per generation: 10
- Value of the initial vector used for the first generation: 0.5 in all indices
- Mutation Rate: 15%
- Selection Method: Deterministic Tournaments of size 4[8]
- Mutation Method:
  49.5%: add .1 to the vector index;
  49.5%: subtract .1 from the vector index;
  1.0%: set the vector index to 0.5[9]
- Goal Fitness for early termination: 0[10]
- Maximum Generations: 2000

5 Benchmark Problems

For the following descriptions, let N be the length of the bitstring judged by a fitness function. In our implementation, the best fitness value for all fitness functions is 0.

5.1 Univariate Problems

These problems are univariate in the sense that the fitness is some sum total of index fitnesses, and no index fitness is dependent on any other index value.

5.1.1 Onemax

Onemax's optimal solution is a bitstring of 1s. This is a quintessential optimization problem and hypothetically the easiest to solve among all benchmarks. In our testing two methods were used for Onemax, one where the absolute difference between the floating point value at an index was compared to one, and the total difference was summed, and one where fitness was worsened by one for each index which was rolled under by a RNG. Ultimately

[8] Selection is only used for methods that have a population they select from. Deterministic Tournaments are those where, N times, L members are chosen from the population at random and whichever of the L members has the best fitness is selected. In this case, L is four.

[9] Following mutation, the vector index is rounded to 0 or 1 if it ends up outside of the 0 to 1 range.

[10] In other words, only terminate early if the optimal solution is reached.

both fitness functions evolved the same models.

The worst fitness value for Onemax is N. In our tests, Onemax is always tested with $N = 300$.

5.1.2 Alternating

A variant of Onemax, Alternating's optimal solution is a bitstring of alternating 1s and 0s. No fitness is rewarded for an incorrect value at any position. While something like this has probably been used before, this method was independently derived to attempt to test more univariate fitness functions than just Onemax.

The worst fitness value for Alternating is N. In our tests, Alternating is always tested with $N = 300$.

5.2 Multivariate Problems

The remaining problems are more complex, with some amount of fitness dependent on connectedness between indices.

5.2.1 Four Peaks

Four Peaks, as described in the MIMIC paper, is a variant of Onemax where a string of all 1s or all 0s both have a high but suboptimal fitness, and a string of NT 1s followed by $N(1-T)$ 0s, or $N(T-1)$ 1s followed by NT 0s are the optimal solutions, for $0 \leq T \leq 1$.

The worst fitness for Four peaks is $2N - T$ In our tests, Four Peaks is always tested with $N = 100$, $T = .1$. [8]

5.2.2 Six Peaks

A variant on Four Peaks where whether 1s are followed by 0s or 0s followed by 1s is no longer a part of the fitness function, adding two new optimal solutions where 1s and 0s are swapped from the two optimal solutions in Four Peaks.

The worst fitness for Six peaks is $2N - T$. In our tests, Six Peaks is alwys tested with $N = 100$, $T = .1$. [8]

5.2.3 Trap-M

The bitstring is split into chunks of size M, and in each chunk optimal fitness is attained by having all 1s. However, fitness is lowest at $M - 1$ 1s and increases towards a local optimum of all 0s. This is called "Trap" because the decreasing fitness of adding 1s makes it difficult to evolve into all 1s.

The worst fitness value for Trap-M is $N(M-1)/M$. In our tests, we run Trap-3 and Trap-5. Trap-3 is always run with $N = 99$, and Trap-M is always run with $N = 100$. [11]

5.2.4 Quadratic Assignment

The bitstring is split into pairs of adjacent indices, and optimum fitness is attained when all pairs share the same value as their partner. The form used in our testing has pairs of 1s weighed slightly better than pairs of 0s, so again a string of full 1s has the best fitness, but pairs of 0s can trick an algorithm out of finding that.

The worst fitness value for Quadratic Assignment is $N/2$. In our tests, Quadratic Assignment is always tested with $N = 100$. [12]

6 Analysis

We'll describe the performance of each method, in three categories of quality.

6.1 Poor Methods

One specific method stands out from the data as an outlier, to the point where it was removed from our figures to make the distinction between the remaining methods clearer.

6.1.1 SHCLVND

The paper for SHCLVND lists the tests that the algorithm was used on to demonstrate its effectiveness, and one parameter in these tests stands out as a problem for the tests we're using: the paper only uses vectors of a size between one and seven. We ran a few tests on our examples of size 7 to demonstrate that, yes, SHCLVND is successful at solving these, but with any significant number of samples, there's a very high chance that the optimal answer will occur due to random chance with that low of a vector length.

It's no surprise then that SHCLVND performs the worst by far out of all of the methods tested. At univariate tests, SHCLVND barely outperforms the expected fitness of a single random sample. It doesn't look to perform as poorly on the peak tests or the trap tests, and does well at the Quadratic test with an average peak fitness of 10, but even these results are still the worst of the pack. SHCLVND serves as a baseline for the remaining methods to be judged against.[11]

On top of performing the worst, SHCLVND takes far longer to evaluate the average generation compared to the other univariate models. CGA, PBIL, and UMDA all take a little over a millisecond to evaluate a generation while SCHLVND takes three and a half milliseconds.

6.2 Average Methods

We call methods average if they did not perform the best at any fitness test without ties.

6.2.1 UMDA and CGA

With a few exceptions, UMDA and CGA perform about as well as each other through each test. Thankfully that includes both of them performing optimally on the univariate tests, which is what we would expect from the univariate methods.

UMDA and CGA also take approximately the same time to evaluate an average generation, the lowest of all methods at around one millisecond.

6.2.2 BOA

BOA ties for first on the univariate problems with the optimal solutions, and is the second best method for Four Peaks and Trap 5, but isn't the best at anything based on our current data. BOA also takes by far the longest to evaluate a single generation, at almost 25 minutes to run 2000 generations. BOA showed one specific feature that wasn't found in any other model, however, and that was that BOA never completely converged on a solution, always improving in fitness, if slowly. Presumably at some higher max iterations, every method other than BOA would converge far sooner than BOA would, but 2000 is too low for BOA to take advantage of its flexibility in this respect. BOA also only showed this trait under very specific input parameters, whereas most methods performed about as well under a wide spread of parameters.

For BOA's ability to avoid convergence, it deserves more attention than the other average methods.

6.2.3 EcGA

EcGA's performance is middling, taking some third places and being the only method that was not the best at any problem but also failed to optimize the univariate problems, other than SHCLVND. EcGA is one of the slower methods to evaluate an average generation, but far outperforms BOA in that respect.

[11] It's worth considering that SHCLVND could perform better with different parameters, but the parameters we use are similar to those used in the paper that introduced SHCLVND and were tested and found to be better than many alternatives, at the very least.

6.3 Top Methods

Each of the top methods performed the best without ties at at least one fitness test.

6.3.1 BMDA

BMDA performs the best on Trap-3 and Quadratic, is second best on the peaks problems and is near-optimal on the univariate problems. Just looking at these results BMDA appears to be the clear winner among the methods tested, but like BOA, BMDA performs analysis on all pairs during its generation function and as a result it takes around 200 milliseconds for an average generation. A single test result for BMDA takes about fifteen minutes to generate for our maximum generation count of 2000.

As another knock on BMDA, while it does perform near-optimally on the univariate tests it does not perform optimally, and most methods did perform optimally.

6.3.2 MIMIC

MIMIC performed the best on Trap-5, and tied for best with an optimal solution for Alternating and Four Peaks. The remaining tests aren't very good for MIMIC, however, as outside of SHCLVND, MIMIC performed the worst on Onemax, Trap-3, and Quadratic Assignment. That said, MIMIC did have the best average time per generation before the univariate methods, as its algorithm does not involve any calculations on all pairs.

6.3.3 PBIL

PBIL performed the best on Six Peaks, was second best on Trap-3, and tied for best with an optimal solution for Alternating, Onemax, and Four Peaks. PBIL is perhaps the most effective algorithm for the time it takes to write and run, with the lowest average runtime for a generation along with cGA and UMDA, outperforming both of them on most tests. The code for PBIL is the smallest out of all of the methods and PBIL still outperforms the others.

6.4 Generations to Reach Optimum

Unfortunately most methods did not reach the optimum solution on most problems, so an overarching comparison of how long it takes for each problem to reach the optimal solution given the constraints used is not available from our data. We can do a partial analysis of the methods that did reach an optimum before 2000 generations, however.

On OneMax, PBIL reached the optimum the fastest followed by CGA and BOA, with UMDA the slowest. Given that CGA also found the optimum on Alternating the fastest, this might imply that CGA outperforms UMDA in this metric.

Otherwise of note, PBIL reached the optimum solution on Four Peaks marginally faster than MIMIC did, giving it more credit as a very good method to use over the more complex approaches.

7 Conclusion

The ultimate recommendation that this data would suggest is that if you are considering using an EDA, you can probably write up PBIL and have an effective solution that will perform very well and has a good chance of being optimal regardless of the complexity or interconnectedness of the distribution of the problem, and this solution will definitely be calculated faster than or as fast as any other method you could try. After writing PBIL, then consider more complex methods as more optimal solutions are required, potentially preferring MIMIC, BMDA, and BOA in that order.

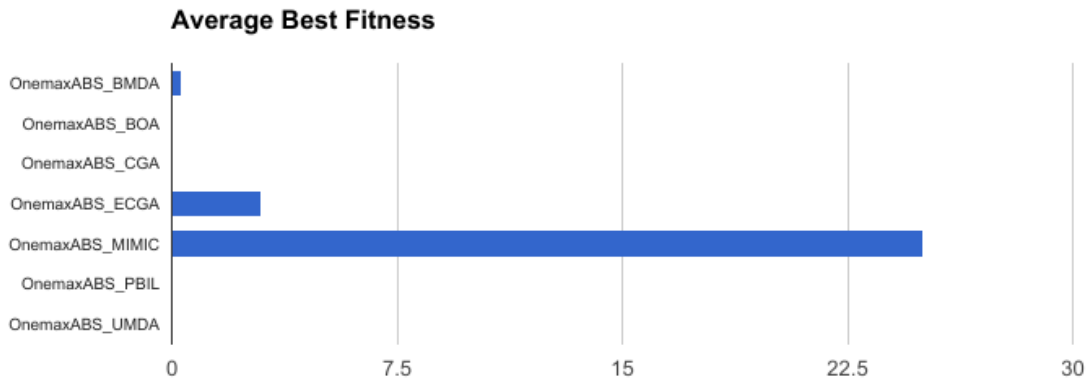
This data should be taken with the significant caveat that we did not have access to a

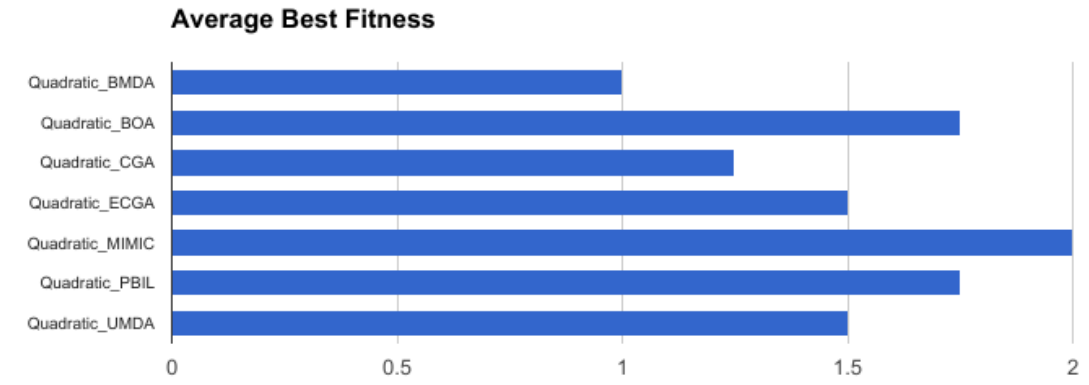
super-computer and due to how long the more complex methods took, we were not able to run a significant number of cycles of the tests to have absolute confidence in the averages obtained. The next steps for this project should begin with repeating these tests to reinforce the accuracy of this data.

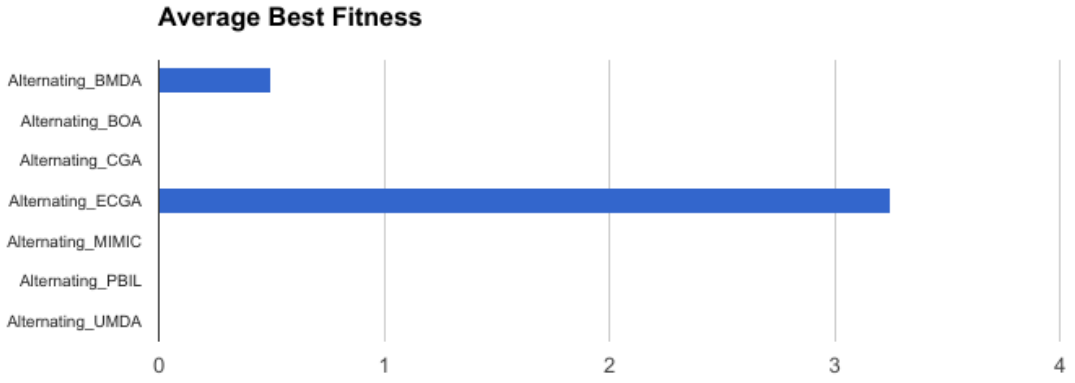
Further next steps should involve visualization and integration of the EDA API into the API used for non-EDA evolutionary algorithms in the library. Following this, yet more methods should be built and tested with similar restrictions as used in these tests, and yet more fitness problems should be built to test on.

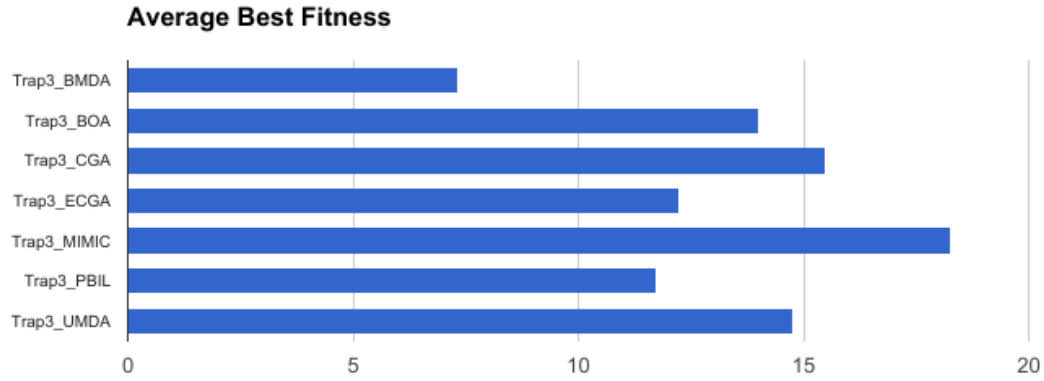
8 Figures

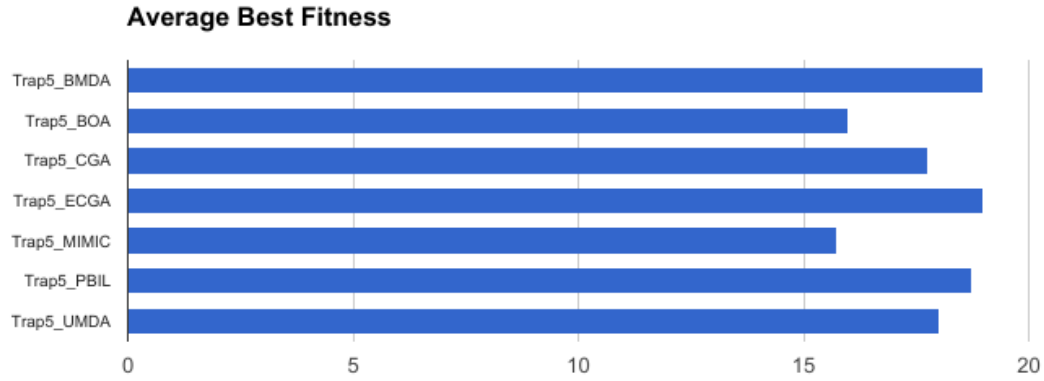
8.1 Without SHCLVND

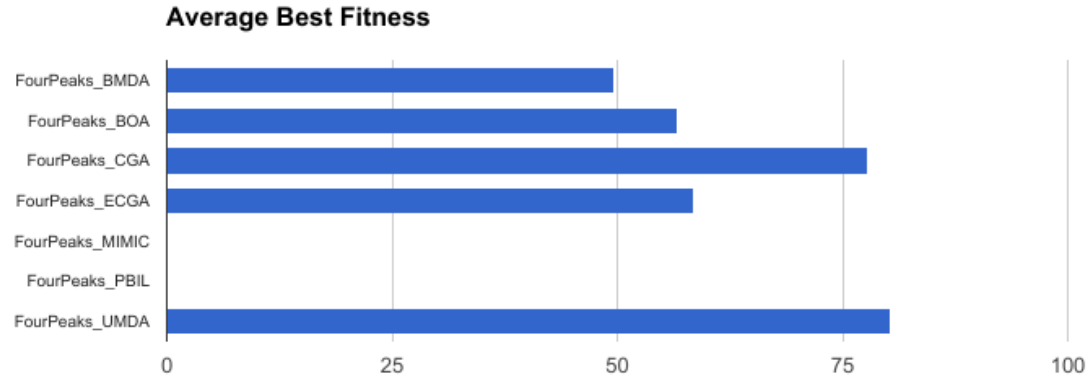


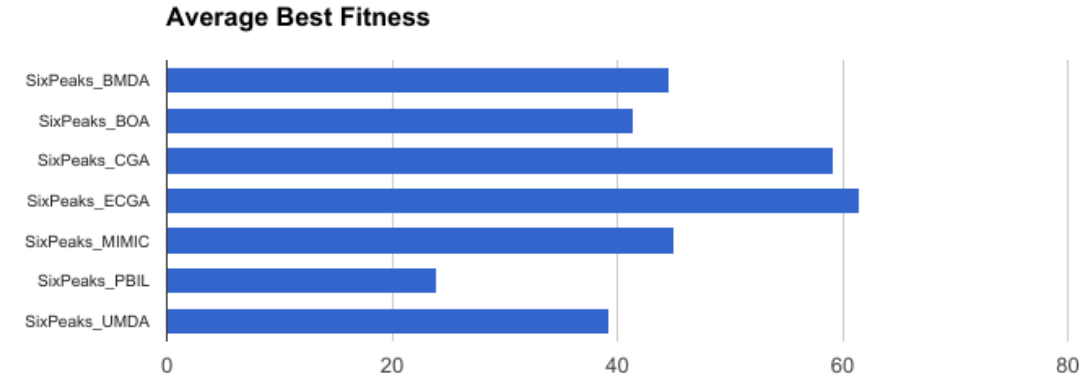


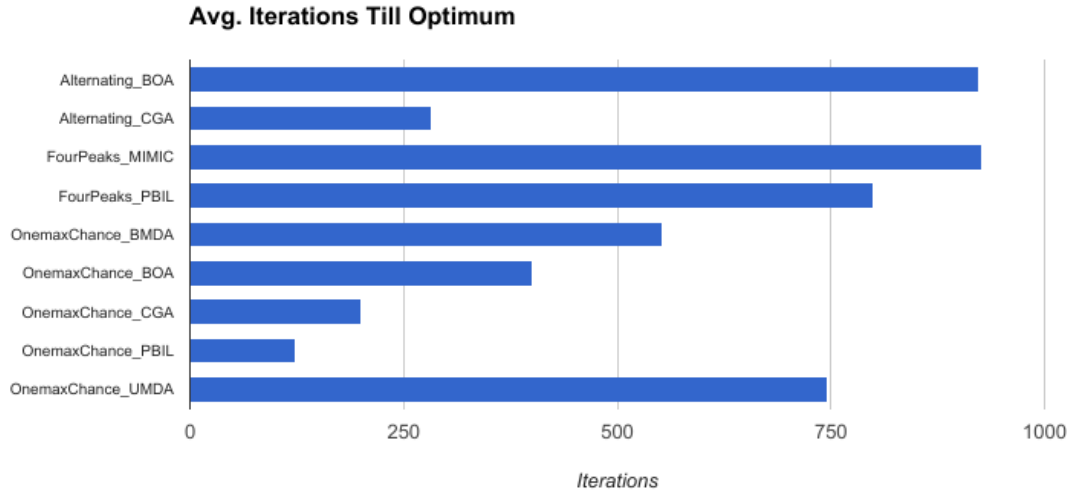

Average Best Fitness
Alternating_BMDA
Alternating_BOA
Alternating_CGA
Alternating_ECGA
Alternating_MIMIC
Alternating_PBIL
Alternating_UMDA
0
1
2
3
4


Average Best Fitness
Trap3_BMDA
Trap3_BOA
Trap3_CGA
Trap3_ECGA
Trap3_MIMIC
Trap3_PBIL
Trap3_UMDA
0
5
10
15
20


Average Best Fitness
Trap5_BMDA
Trap5_BOA
Trap5_CGA
Trap5_ECGA
Trap5_MIMIC
Trap5_PBIL
Trap5_UMDA
0
5
10
15
20



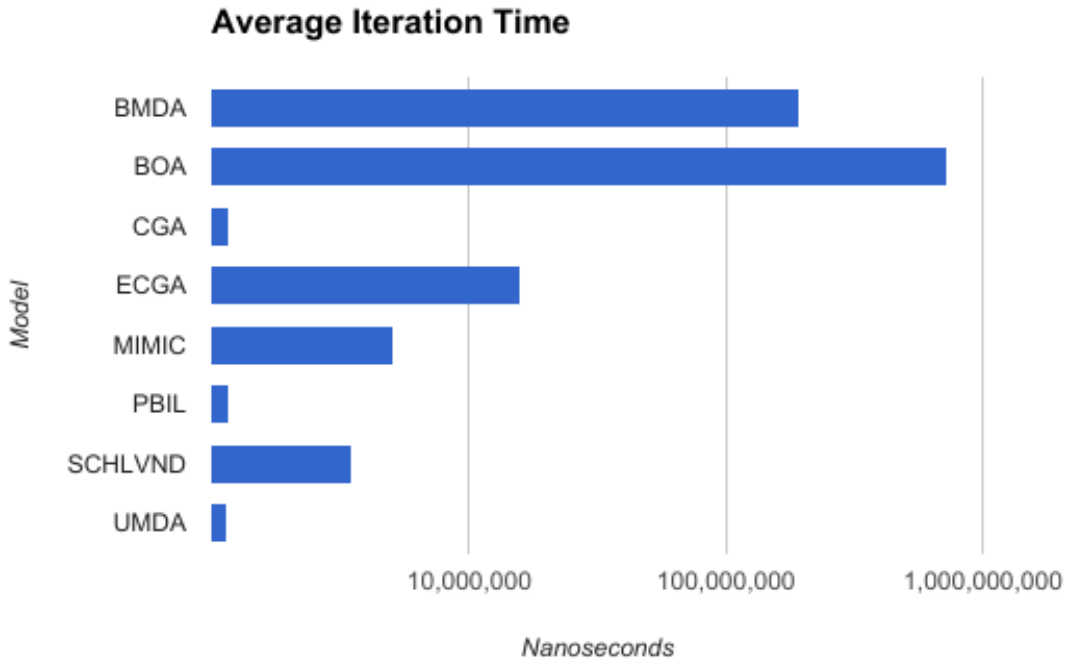


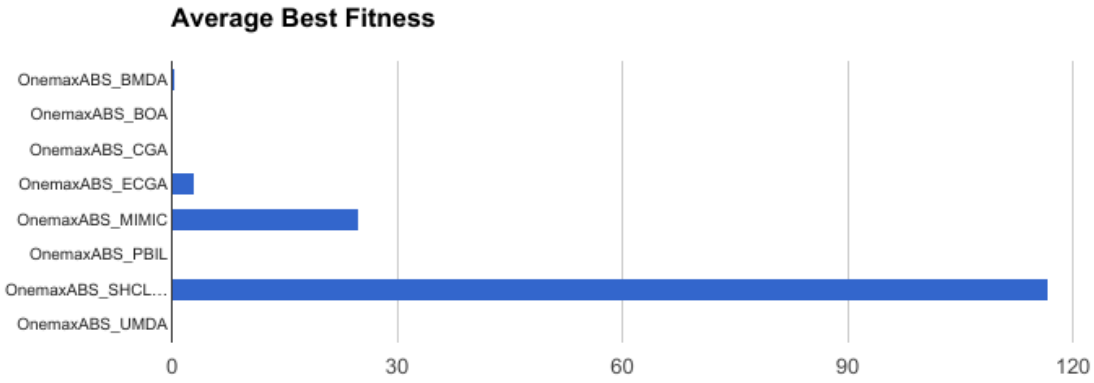


12

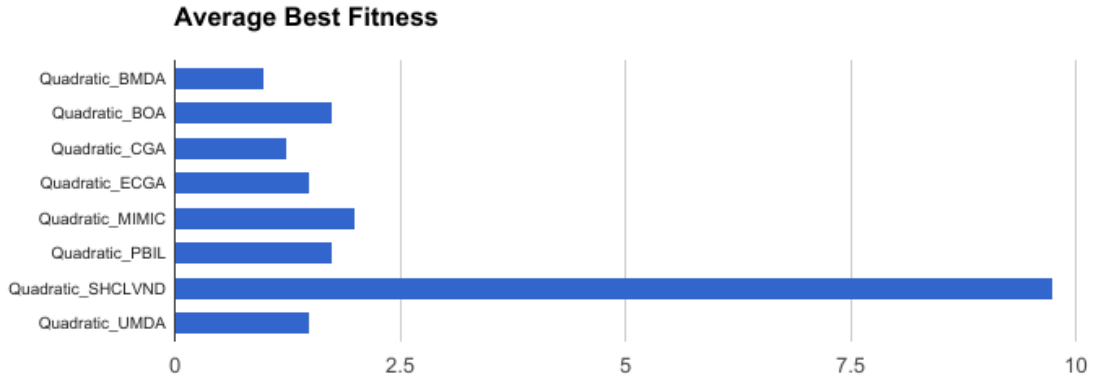
[12]This graph only shows the tests where it took less than 2000 generations to find the optimum more often than

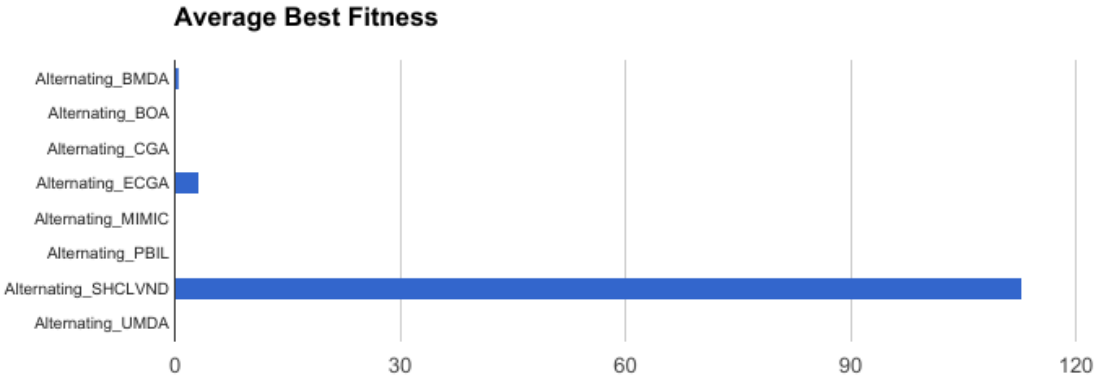
8.2 With SCHLVND

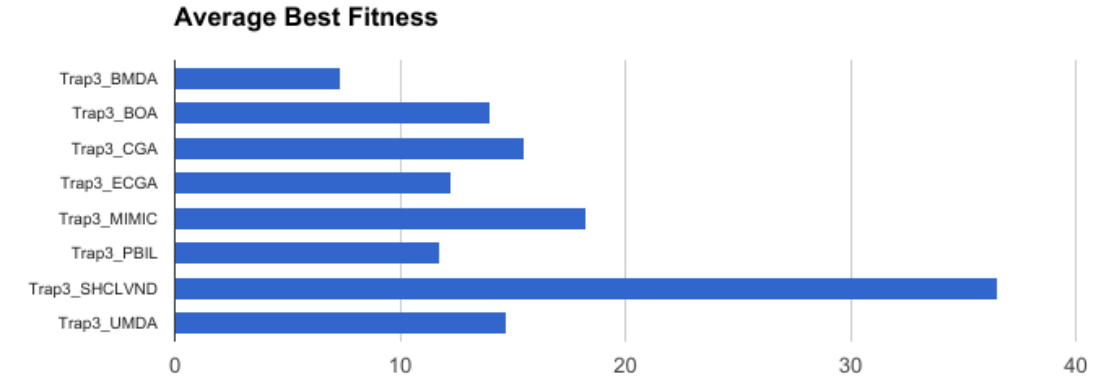


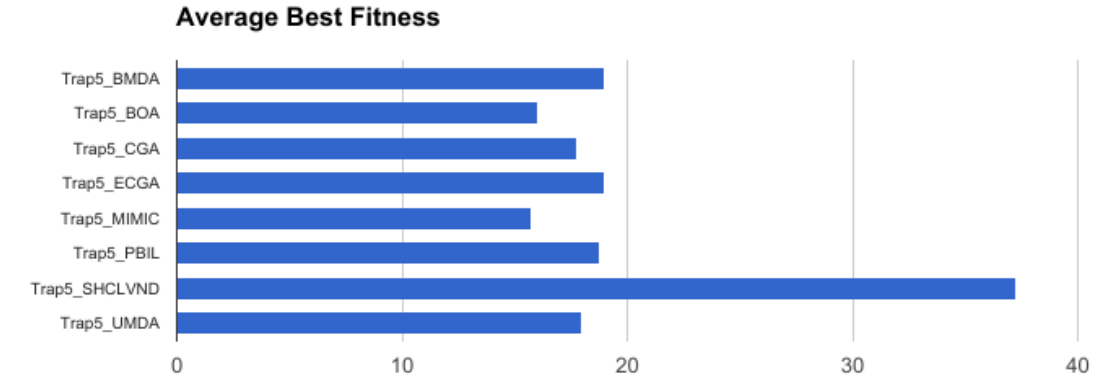


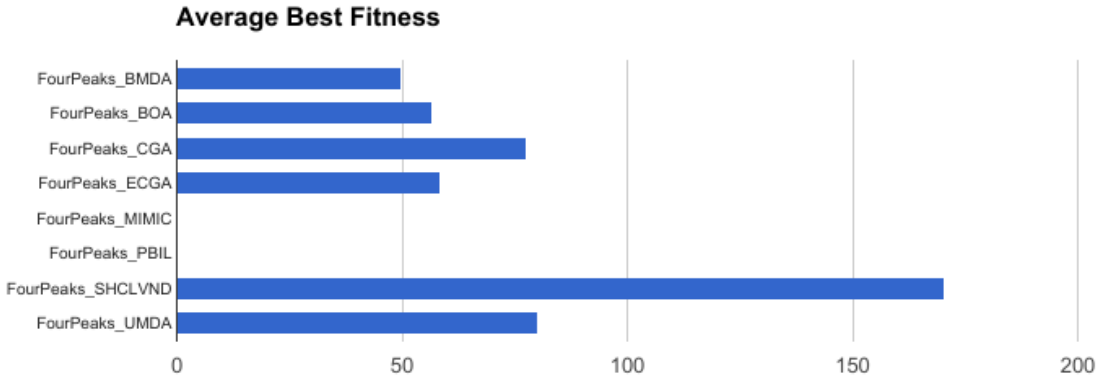
not. This is a little disingenuous, as the implementation will not leave early if any given sample found during a generation reaches the goal fitness, but only if the vector judged in the continue function reaches the goal fitness.

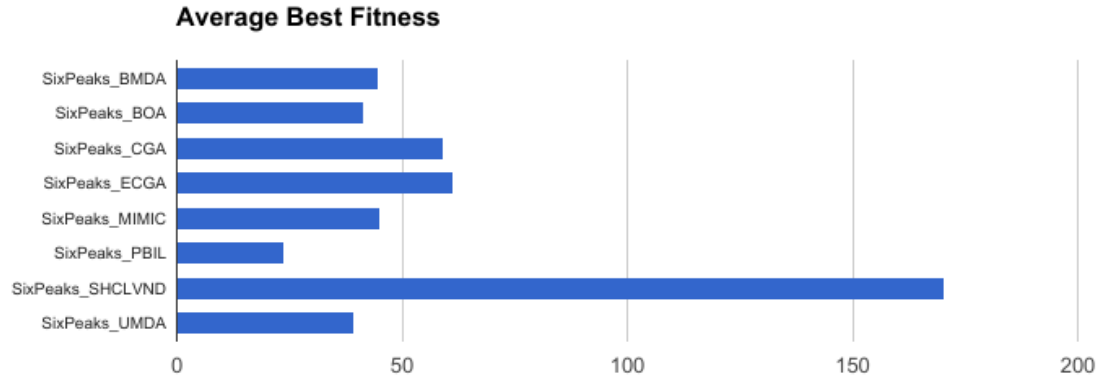

Average Best Fitness
Quadratic_BMDA
Quadratic_BOA
Quadratic_CGA
Quadratic_ECGA
Quadratic_MIMIC
Quadratic_PBIL
Quadratic_SHCLVND
Quadratic_UMDA
0
2.5
5
7.5
10


Average Best Fitness
Alternating_BMDA
Alternating_BOA
Alternating_CGA
Alternating_ECGA
Alternating_MIMIC
Alternating_PBIL
Alternating_SHCLVND
Alternating_UMDA
0
30
60
90
120


Average Best Fitness
Trap3_BMDA
Trap3_BOA
Trap3_CGA
Trap3_ECGA
Trap3_MIMIC
Trap3_PBIL
Trap3_SHCLVND
Trap3_UMDA
0
10
20
30
40







References

[1] M. Hauschild and M. Pelikan, "An introduction and survey of estimation of distribution algorithms," *Swarm and Evolutionary Computation*, vol. 1, no. 3, pp. 111 – 128, 2011.

[2] H. Mühlenbein and G. Paaß, *From recombination of genes to the estimation of distributions I. Binary parameters*, pp. 178–187. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996.

[3] F. G. Lobo, "Lost gems of ec: The equilibrium genetic algorithm and the role of crossover," *SIGEVOlution*, vol. 2, pp. 14–15, July 2007.

[4] S. Baluja, "Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning," Tech. Rep. CMU-CS-94-163, Computer Science Department, Pittsburgh, PA, 1994.

[5] G. R. Harik, F. G. Lobo, and D. E. Goldberg, "The compact genetic algorithm," *IEEE Transactions on Evolutionary Computation*, vol. 3, pp. 287–297, Nov 1999.

[6] S. Rudlof and M. Kppen, "Stochastic hill climbing with learning by vectors of normal distributions," pp. 60–70, 1997.

[7] G. R. Harik, F. G. Lobo, and K. Sastry, *Linkage Learning via Probabilistic Modeling in the Extended Compact Genetic Algorithm (ECGA)*, pp. 39–61. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006.

[8] J. S. D. Bonet, C. L. Isbell, Jr., and P. Viola, "Mimic: Finding optima by estimating probability densities," in *Advances In Neural Information Processing Systems*, p. 424, The MIT Press, 1996.

[9] M. Pelikan and H. Muehlenbein, *The Bivariate Marginal Distribution Algorithm*, pp. 521–535. London: Springer London, 1999.

[10] M. Pelikan, D. E. Goldberg, and E. E. Cantú-paz, "Linkage problem, distribution estimation, and bayesian networks," *Evol. Comput.*, vol. 8, pp. 311–340, Sept. 2000.

[11] D. H. Ackley, *An empirical study of bit vector function optimization*, pp. 170–204.

[12] T. C. Koopmans and M. Beckmann, "Assignment problems and the location of economic activities," *Econometrica*, vol. 25, no. 1, pp. 53–76, 1957.