CS4641: Machine Learning
Spring 2019
PS2 Randomized Optimization

Joel Ye

February 28, 2019

1 Neural Network

In this part of the assignment we consider the performance of random optimization on the same Diabetes dataset[2] used in PS1. Note that we are only swapping out the optimizer, the way we traverse our hypothesis space, which is constrained to neural nets. As a reminder the Diabetes dataset presents a binary classification task of whether the subject, reduced to eight medically diagnostic features. All the features were preprocessed in the same manner as before, to zero mean and unit variance. From overall analysis in the last assignment, this is a challenging dataset, with very noisy surfaces. As a baseline, all models were constrained to a single hidden layer with six units, so that the only variable was the optimization algorithm. This was sanity checked against backprop performance which matched results in PS1.
The three randomized algorithms compared were randomized hill climbing, simulated annealing, and genetic algorithms. The common requirement among randomized optimizers is a fitness function to optimize, and in the neural network problem we used accuracy as fitness. The specific implementations were provided by mlrose[1], a Python framework designed for comparing these randomized algorithms.

Figure 1 presents a comparison of each of the optimizers with respect to the number of iterations run. Overall caps to iterations across this assignment were chosen due to prohibitive runtime. All training and validation scores calculated in this section use 5 fold cross validation, for averaged results. Also note that since weight optimization is over a continuous domain and randomized optimizers generally require discrete ones, we arbitrarily choose a small step size that we discretize our domain with, much in the way we choose a small learning rate to step our weights in regular backprop - in fact implementation wise learning rate is used for this step size. We expect the optimizers to perform strictly worse than backprop, which is mathematically proven to take the 'best' step, as opposed to the random step (and we expect the surface to be well behaved in a medical problem). Details on algorithms are saved for section 2.

A few things are apparent - as from PS1, this problem converges quickly, so backprop with tuned regularization (here a hard threshold for weights) will have consistent performance. By 200 iterations, backprop sets the goal to reach at around 77%. Comparing the randomized models, we see trivial results for both simulated annealing and randomized hill climbing. This is due to the fact that the hypothesis space is full of local extrema (due to lack of data, input space is also not too well defined). It is hard to climb, and restarts and temperature cooling both do little to combat this - averaged over 5 trials there is still noise but presumably after rounds, we'd see mostly flat, trivial accuracy. Genetic algorithms perform a little better, shotgunning a large population increases the

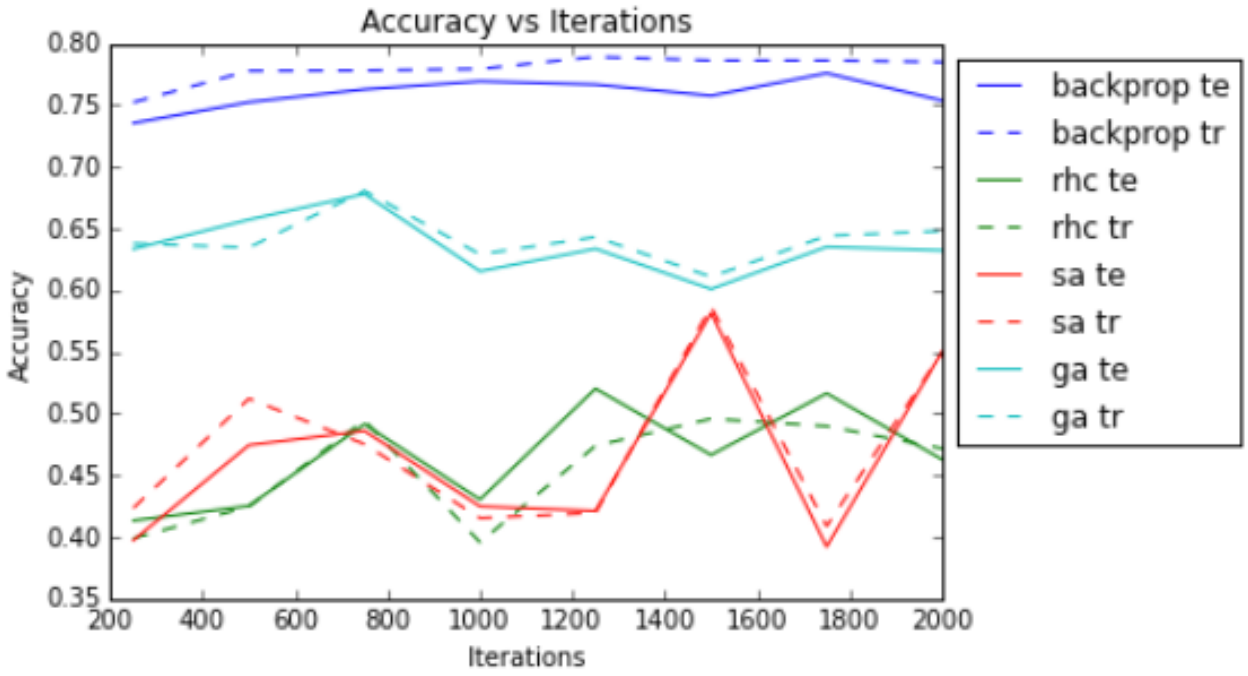

Figure 1: Comparison of All Optimization Methods

probabilities of some escaping local extrema (naturally at the cost of more computation). If the hidden nodes here compute meaningful features, then we can imagine two good individuals crossing over as attempting to save good feature representations. However, this roundabout approach overall does not do as well as regular backprop - it is also an order of magnitude slower, providing two reasons why backprop is the reliable backbone to optimizers in neural networks. One notable phenomena here is that in randomized algorithms, very little overfitting occurs - none of the random algorithms have non-negligible differences between validation and training curves, and this makes sense because we don't optimize closely aligned to training data. Not presented is a time versus iterations graph, which is generally trendless (all algorithms roughly run at around half a second, regardless of iteration count) suggesting early convergence and exit.

We can examine each of the optimizers on particular hyperparams to try to gain more insight. 2 shows trendless diagrams for randomized hill climbing with respect to number of restarts. In hill climbing, restarts enable what equates to multiple chances for the climbing to find better maxima - the final algorithm records the best one. This is like a genetic algorithm, if only individuals didn't communicate progress, and there were only one generation (we see why RHC is the simplest algorithm). Typically we expect more chances to equate to better overall best results, yet being trendless here indicates a great sparsity in maxima in weight space, such that even 10 restarts wouldn't give any better shot results in our splits. The time graph indicates that these hill climbing procedures are quite fast (forward passes with 100 attempts at finding a random neighbor that's better), so much so that restarts don't really factor in - this means several restarts begin near local maxima and terminate extremely quickly.

For simulated annealing, we can sample a few points in the cooling rate. Cooling rate controls how willing the annealing algorithm is to trying worse neighbors over time, and typically this flexibility makes annealing perform better than RHC. In this case, however, the space is so littered with local maxima that the one start annealing has is likely to be far away from any actual good

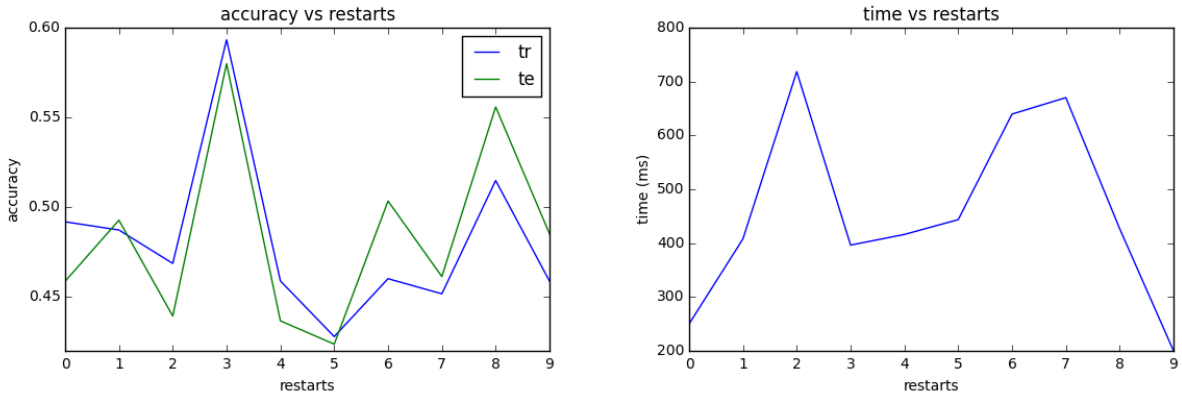

Figure 2: Varying Restarts in RHC

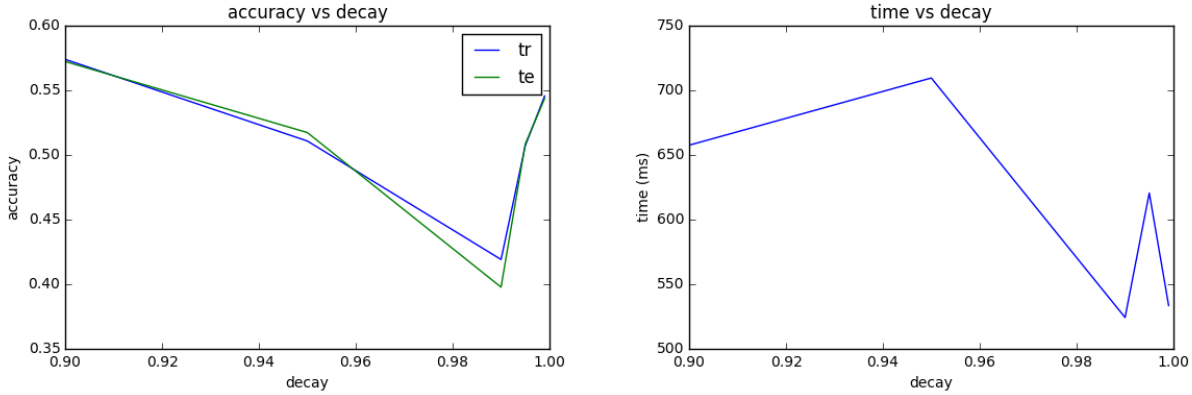

Figure 3: Varying Cooling in SA

solution, regardless of cooling rate - so it performs trivially as well, making it on par with RHC. Indeed, despite trying several different cooling schedules in 3, all performances are negligibly better or worse than random. The time graph may confuse with apparent downward trend, but the difference is negligible given the overhead of mlrose. For genetic algorithms, we can vary population size. Now, as every individual needs to be evaluated, we expect genetic algorithms to be at least the order of population size slower than hill climbing or annealing, and it does take a while - however this reflects back to the point that RHC and SA must evaluate many neighbors, while GA just assumes that two good individuals will combine to form a better individual - so runtime is roughly equal. In terms of performance, we again see no huge difference over population size in 4, which is surprising considering how large the discretized hypothesis space we present is - it's possible that on average GA simply settles on the best few samples in the seed populations and bounces around the local maxima in that range, and the variance we see is due to seed noise (for around 7% plus or minus, we can excuse this).

Overall for this particular problem, we see that randomized optimization generally fails for problems with complex solution topologies. Not much can be tweaked hyperparameter wise to change this. Even the slight structure of genetic algorithms is provides a big boost on performance, but GA makes big assumptions of smoothness about the input space. Wandering with high dependency on stochastic serendipity, especially as count of independent dimensions grow higher and higher, is unreliable, not to mention infeasible if we require very fine discretization of the input

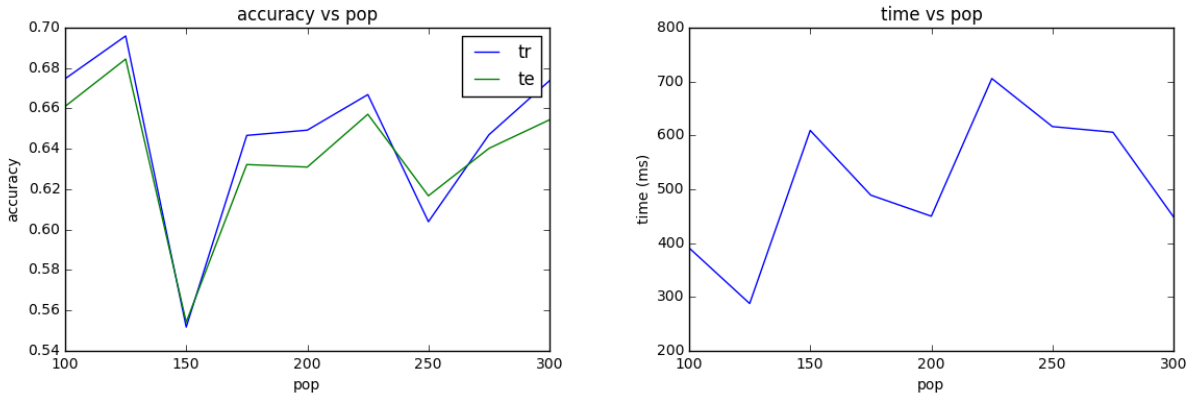

Figure 4: Varying Population Size in GA

space to not miss global optima. This is not to discount introducing elements of randomness into backpropagation, but this would have to be studied carefully. Thus the next section examines the strengths and weaknesses of each algorithm for overall insight into random optimization.

2 Optimization Case Studies

We now examine three different optimization problems and see how they highlight the differences in four randomized optimizers, the three we have already seen and MIMIC, a structured approach inspired by genetic algorithms as presented by Isbell[4]. The details of the problems are presented in respective sections, but overall for problems with variable setups fixed parameters were chosen, so that we could compare performance of the optimizers rather than study how the problem spaces change. Note that time graphs are largely omitted due to unremarkable results (more iterations means more time, roughly speaking), but they can be generated in the submitted code. In brief summary, simulated annealing and randomized hill climbing are blazingly fast, genetic algorithms come in slower as a function of population size, and MIMIC comes in orders of magnitude slower, running near a half second for one trial of one parameter setting, each iteration costing that much more due to the algorithm overhead.

2.1 Flip Flop: Simulated Annealing

The Flip Flop problem is a toy example where the fitness function counts the number of bits in an instance that are the negation of the previous bit. Thus given instance length, set in this study to 200, the maximum fitness is the length of the instance less 1. Initial consideration can show that this problem space has a well defined global minimum and exactly two global maxima, the number of neighbors better than an instance is exactly inverse to its own fitness, thus we can expect randomized algorithms to approach a 'random' optimal that is some scaled constant smaller than the true optimal.

Let's detail the algorithms used in mlrose's implementation of randomized hill climbing, and closely related, simulated annealing. Hill climbing is straightforward, it chooses the best neighbor - literally a rephrasing of gradient descent, but in most naive implementations the 'best' neighbor is calculated by sampling each neighbor instead of calculating gradient analytically. Randomized hill climbing tries to be more open minded by simply thresholding a certain improvement factor (in the simple case, as long as a neighbor shows any improvement), and stepping to a random neighbor

among these better options. In mlrose, instead of scanning all the neighbors, we simply sample at random until a certain number of attempts after which the algorithm concludes the input is a local maximum and exits. As mentioned earlier the hill climbing algorithm can be made more robust to local maximum by restarting the algorithm at random points and climbing from there, providing a very clear tradeoff between time and best fitness. Simulated annealing is similar to hill climbing, but establishes a smooth threshold instead of a sharp one, and uses a probability distribution that is smoother when its temperature parameter that is high, but sharper and more restrictive as the algorithm progresses and temperature is cooler. The exact decay schedule can be specified in mlrose, we tend to use geometric decay as after a comparison there is no clear victor. The exact details of the annealing formula are omitted - they are easily found online.

This problem was chosen to highlight the strengths of simulated annealing, as there's not much structure for more complex algorithms to exploit. In terms of local maxima, the problem makes better functions appear more and more maximal, annealing outperforms hill climbing here mainly because it is willing to press onward even if every step is not strictly improvement. This comparison is more apparent in the graphs in 5. Forgiving the varying scales on axes, it is clear that given a reasonable number of attempts, simulated annealing does pull ahead, to almost optimal fitness. It's interesting that RHC does better given low number of attempts; this is an artifact of the default number of restarts giving RHC an implicit 'higher attempt' count - this peaks significantly below SA. MIMIC is unable to make much of the data, because it is unstructured, and probabilistic bounds cap its performance. GA performs quite well because it incorporates random mutations and the slight chain structure of flip flop being easy to exploit in GA, making a viable contender for SA if not for its extra complexity.

2.2 Four Peaks: Genetic Algorithms

Though in the original paper the problem was designed to highlight the strengths of MIMIC, in this particular case genetic algorithms pull ahead due to their faster evaluation. MIMIC has the overhead of its sampling algorithm (to be detailed), while genetic algorithms can simply evaluate the simple fitness function. To elaborate, four peaks was a function that takes a bitstring and a threshold T, and returns a fitness based off of leading 0s, trailing 1s, and capitalizing on the threshold (again, formula omitted). More importantly, the problem is designed with two basins of attraction for hill climbing based algorithms (RHC and SA) at either end, all 0 or all 1, and two global maxima that are closer to the center. The larger the provided threshold T, the wider these basins of attraction. This problem will assuredly lure in RHC and SA because the threshold is so sharp, so unless you initialized extremely luckily the basins of attraction would be the most visible maxima.

However, crossover, the generally unintuitive and non-hill climbing like approach to optimization presented in genetic algorithms, won't get pulled towards these basins. Genetic algorithms are evolutionarily inspired, and in each iteration or generation, a pool/population of individuals are evaluated. The best individuals are selected, in these experiments using the DEAP[3] python framework for genetic programming and its default option, which is tournament selection, and a random crossover point is selected at which point two new individuals are created, as combinations of these parents. These new individuals are randomly mutated with a certain probability, and then thrown into the next generation's population pool. The fundamental assumption made by genetic algorithms is that two individuals contain strengths that are characterized by subsequences of the overall individual, and that we can use good parents to produce good children. In the case of four peaks, the flexible swapping may enable individuals that previously approached the local maxima and completely change the trajectory of evolution - the point is that there is no pull.

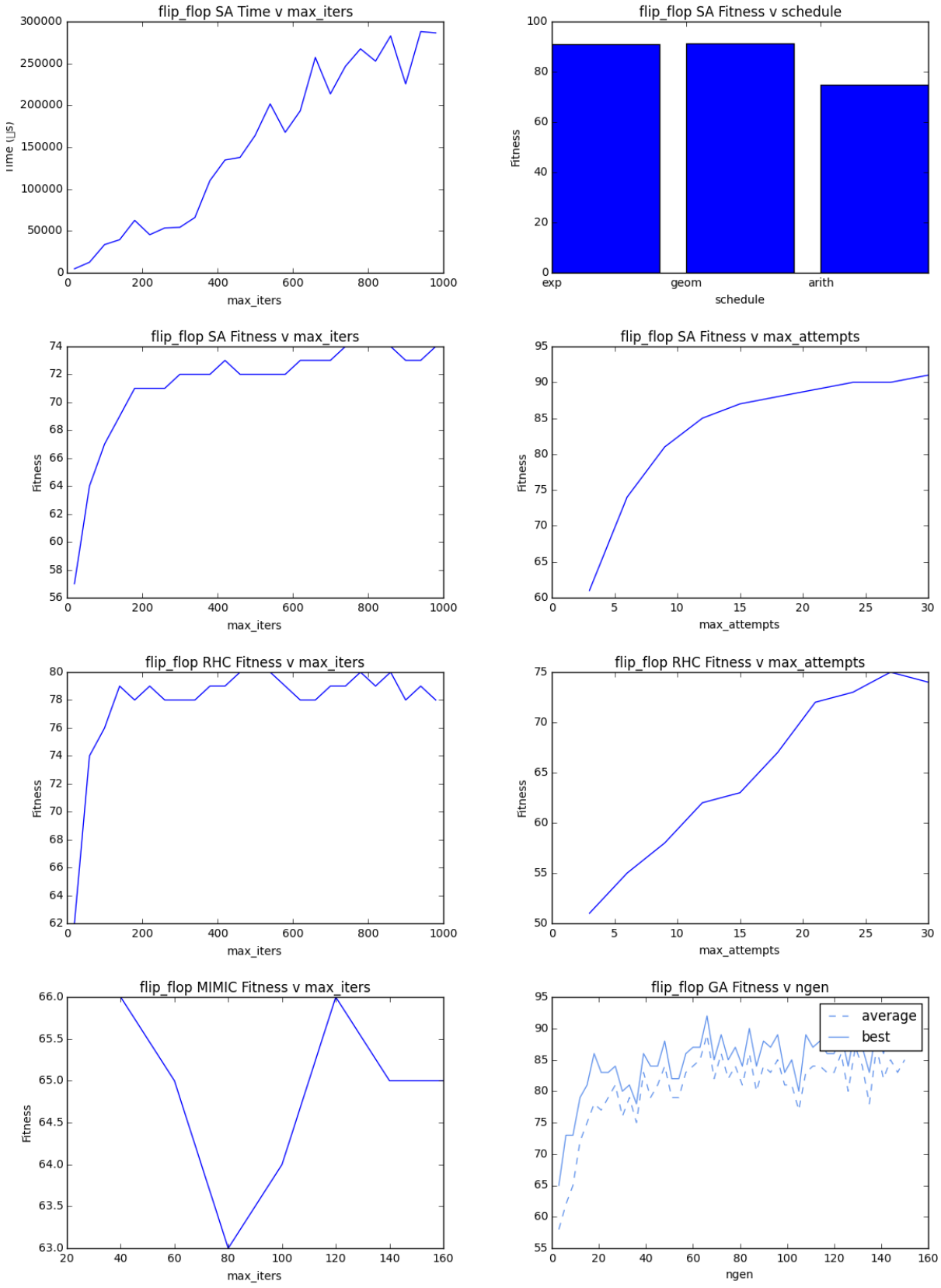

Figure 5: Simulated Annealing in Flip Flop

An individual of length 50 was evaluated with threshold 10, implying maximum fitness of 90. Default settings on GA were .2 mutation probability, .5 crossover probability, 100 individuals and 100 generations. We see in 6 that both increasing population size and mutation probability (to a certain extent) allow GA to find the optimal results. It's curious how large the gap is between average performance and best performance is, indicating single good individuals cannot propagate their genes across a population, at least with tournament selection mechanism.

Examining other algorithms, we see that both annealing and RHC perform poorly (which makes sense, given the problem design). It's promising that annealing has positive fitness trend, but other trials suggest SA fitness peaks at around 30. The most curious behavior is in MIMIC's spike. This is likely due to the necessary amount of iterations to converge on the solution, and we would see a similar spike at different times on other seeds. Compared to the amount of time it takes GA to find the same solution or even better, however, GA is the clear winner in this case.

Other hyperparameters were tuned but generally found to be trendless (RHC vs iterations, RHC on different decay schedules) or intuitively positive trending (MIMIC fitness vs population size).

2.3 Knapsack: MIMIC

The Knapsack problem is a classical optimization problem where we try to optimize the value of the items we take given a capacity of our knapsack, represented in this problem by max weights. In order to make the input space tractable for random initialization, I reduced the problem to 0-1 Knapsack, where we can take at most 1 of any given item. Weights and values were randomly initialized, and weight threshold was 60% of the total weight of the bag.

This problem was chosen to highlight the strengths of MIMIC - in particular we do not expect genetic algorithms to succeed because a good solution is probably not the combination of two high fitness parents, as it'll probably go over weight and thus fall to zero fitness. On the other hand RHC and SA are not complex enough to traverse this space riddled with local maxima (because many neighbors are overweight and thus invalid). MIMIC's structured sampling is most likely to yield good results, as we can imagine the heaviest values rooting the dependency tree. However empirically we see that MIMIC and genetic algorithms perform roughly equal at best; in fact, MIMIC is edged out by the best in generation of genetic algorithms in some tunings of the parameters. Currently the best explanation for this assuming that MIMIC does actually converge much faster than other algorithms is that the process is truly random and would best be examined by averaging over a high number of trials, given variable initializations of the knapsack problem. However, optimization time was prohibitive, so we're left with only post-mortem analysis and conclusions that more iterations are necessary to see convergence, especially in MIMIC's case with its odd curves.

We can dive more specifically into MIMIC's graphed quirks by examining the algorithm. At its core MIMIC assumes we can generate more good samples by building a probability distribution for the 'good' individuals in its current generation. It's worth thinking about other selection mechanisms, rather than a simple keep percent threshold, as genetic algorithm's strength lays in its configurability. The probability model presented is a dependency tree, which requires that each input element depends on at most one other, which is not quite sophisticated enough for Knapsack or many real world problems, but can serve as a useful approximation.

Now observing the graphs in 7, focusing on where we see the peak performance of MIMIC, there's an explainable trend in the percentage kept graph. After a certain threshold of around .7 performance drops, presumably because the number of iterations run on default (100) is not enough to converge on a good solution when we are conservative with our culling. On the other hand, lower percentage kept (around .2) mean we can hone in too quickly and converge on a local maximum

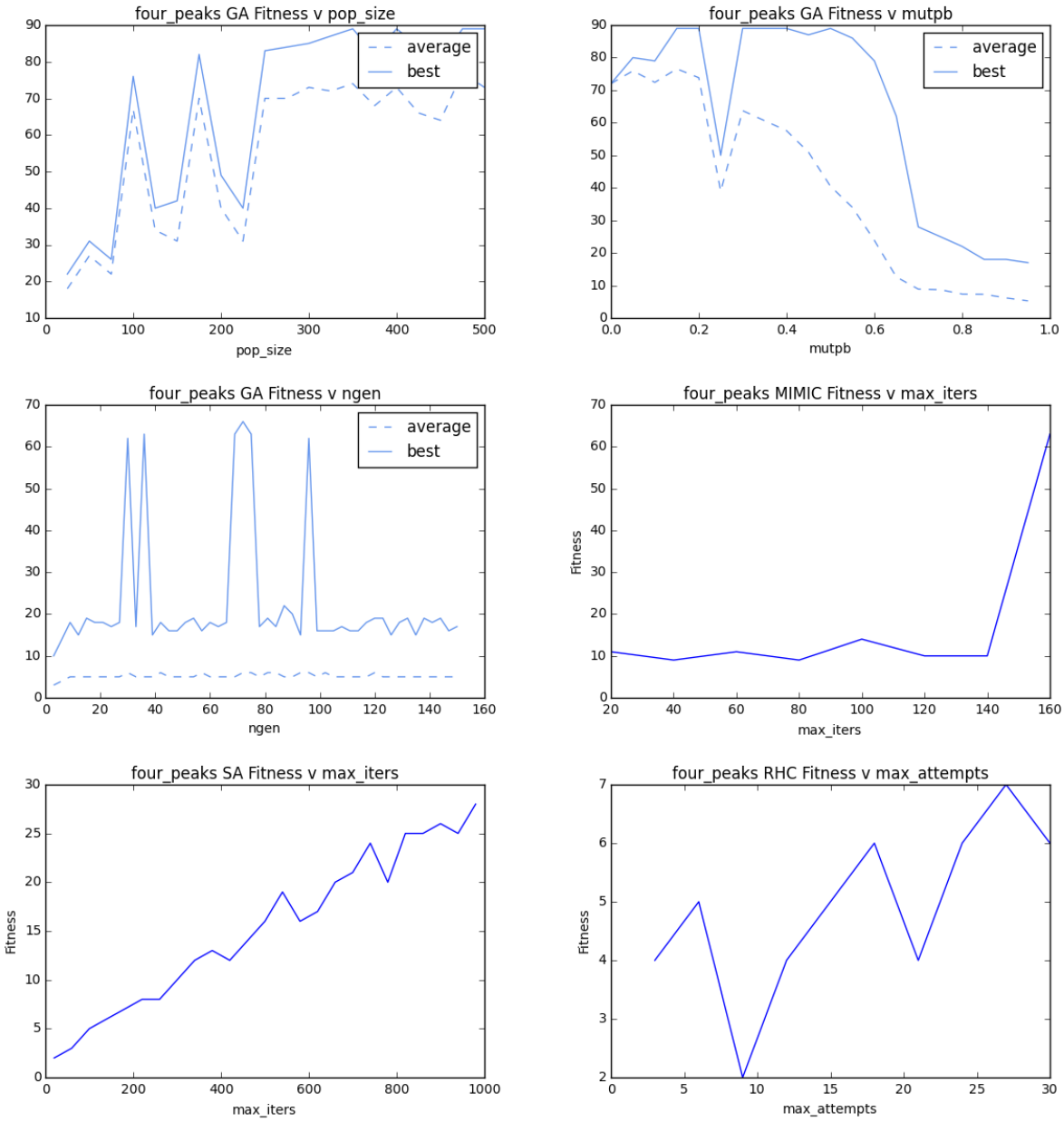

Figure 6: Genetic Algorithms in Four Peaks

Figure 7: MIMIC vs 0-1 Knapsack

(though in this case, still fairly good solutions). SA and RHC hold a somewhat stable baseline at around 3400, which is presumably the "approximately good enough" solution of this instance of the knapsack problem, where we inlude a bunch of valuable items and cross our fingers, which MIMIC reliably beats, but not by much (it chooses just slightly more carefully). It's worth seeing the time graph as well, to get a sense of how long MIMIC takes—a while. In this case iterations don't have too much of an effect on MIMIC as we appear to converge to around 4000 (save some noise); it's much more important the way we update our population rather than how many times we update it. It's worth benchmarking this against GA, which on average scores around 4400 with default parameters. The smooth downward trend on mutation probability indicates there's a lot of stable structure (good combinations of items) that should be kept between individuals, as mutations break this kind of structure down. MIMIC should capture this kind of structure as well. In this case though MIMIC find a better (ish) optimal GA wins because the individuals are cheap to evaluate.

There's a lesson here about what hypothetically works being applied to what empirically works, e.g. simulated annealing is the simplest algorithm that tends to still be used because it is reasonably effective. Since MIMIC failed to pull ahead, we should train for higher iterations, but also consider if its assumptions about sampling are valid - given such a knapsack of size 50, it's highly likely a sample size of 100 is not nearly enough to appropriately model conditional probabilities and mutual information for many of the parameters.

3 Wrap Up

In quick conclusion, we can find problems that illustrate the strengths of each algorithm. Though inconsistent results in this paper were chalked up to be due to insufficient compute, some of the more curious phenomena are worth investigating, to understand more of the cases that a randomized algorithm can go down. In the world of randomized algorithms, all algorithms benefit from more iterations, more random initializations, larger populations, and the like. Few tunings are trickier, such as decay schedule in SA and mutation probability in GA, but these did not affect the toy domain too much. Generally, choosing the right one will boil down to domain knowledge - however, if possible, the default option should be backpropagation.

References

[1] Hayes, G., 2019 *mlrose: Machine Learning, Randomized Optimization and Search package for Python*, Accessed February 26, 2019
`https://github.com/gkhayes/mlrose`

[2] Pima Indians Diabetes Database
`https://www.kaggle.com/uciml/pima-indians-diabetes-database/home`

[3] Félix-Antoine Fortin and François-Michel De Rainville and Marc-André Gardner and Marc Parizeau and Christian Gagné, July 13, 2012 from Journal of Machine Learning Research *DEAP: Evolutionary Algorithms Made Easy* Accessed February 26, 2019

[4] Isbell, Charles L. Jr., 1996, *Randomized Local Search as Successive Estimation of Probability Densities*, Accessed February 26, 2019
`https://www.cc.gatech.edu/ isbell/tutorials/mimic-tutorial.pdf` to insert