Project 2
Random Optimization

Jean-Pierre Bianchi - GT ID jbianchi3

[ ]

1 A word about the graphs

I have implemented various types of graphs: fitness vs iterations (or time), function evaluations (or fitness) vs N, and plots with different values of a hyper-parameter, to do like a visual gridsearch. Even the time or iterations necessary to reach the optimum value (when we know it) vs N. I will mainly present fitness vs iterations for the following reasons. Fitness vs time will squeeze the fastest algorithms to the left, so they are useful when comparing algorithms with similar execution times.

I am perfectly aware of the fact that function evaluations can give a more accurate insight of a situation because, for instance, with HRC, we can only plot the best curve and it would be easy to forget there were 100 restarts or more to get there. For that reason, I have included in the legend the total execution times, so we have the best of both worlds. I feel that, by doing so, the iterations graphs give a very good insight of how the algorithm behave (steady/choppy rise, plateaus etc) while the timings give a reality check of the real power of an algorithm, which is what matters in the real world. Also function evals can be very deceptive when comparing different algorithms, since some do a lot of computation in one function eval, like MIMIC and GA for instance. On the other side, RHC and SA are orders of magnitude faster. So, I'll mostly plot function evals between RHC and SA, or MIMIC vs SA because each algorithm in those couples are quite similar in terms of behavior and timing.

I implemented gridsearch to be able to scan HP's, and when necessary, I'll show a graph to visualize the effect of some particular HP. I may not say it all the time, but I've done gridsearch extensively to try to find the best HP combinations before comparing the algorithms. Also, I've modified Mlrose-hiive, to implement restarts for all 4 algorithms, in order to be able to average their result. This can be a double-edge sword since it could mask a success, which could be important in problems really hard to solve. If you run the code, you'll have a lot more information, for instance the function evals at each gridsearch round, the final values retained, which can all guide a decision. I have put many logs at the bottom of my code for you to check.

2 Four peaks

The Four Peaks problem is very interesting since its optima are highly structured. I'll call 'heads', the ones at the beginning of the binary state vector, and 'tails', the zeros at the end. I'll use a hashtag to designate their amount. The fitness score is the maximum of (#heads, #tails) + a 'bonus'. The bonus is equal to N, only if both #heads and #tails are $>$ T, 0 otherwise.

Because of this particular structure, the basin of attraction for local optima is very wide, easy to fall into, while the basin of attraction of the global optima is much smaller, and almost impossible to find for algorithms that 'jump around' like RHC or SA, for two directly connected reasons. First, imagine a state where #heads is much smaller than both T and #tails, say 5 heads, 15 tails with T=20.

In such a likely starting configuration, RHC or GA will eventually add 1 head or 1 tail. If it adds 1 head, the score will remain the same because #tails is higher, so the algorithm will reject that change, even if it's a good one. That's why I said the basin of attraction towards the global optimum is narrow. Unless the initial state starts with balanced #heads and # tails, it's almost guaranteed that one-bit modifications to the small side will never generate improvements while any modification to the other one will increase the score. Then, second reason, as this happens, the score improves but the imbalance as well, and in fact, it's going straight to the local

optima.

As a result, the randomizer will keep increasing the side with the most 1's or 0's, until the state is full. It'll never get the bonus, which means it reached a local optimum with a fitness score of N only. That's how we recognize a local optimum in the graphs, by looking for plateaus at fitness = N. This phenomenon gets worse when we increase N, because the chances are low to start with a balanced state, ie #heads and #tails close to each other, especially if we also increase T (by increasing the t_pct parameter which initializes the FourPeaks class). RHC and SA are going to need a LOT of restarts to get a chance to find the global optimum basin.

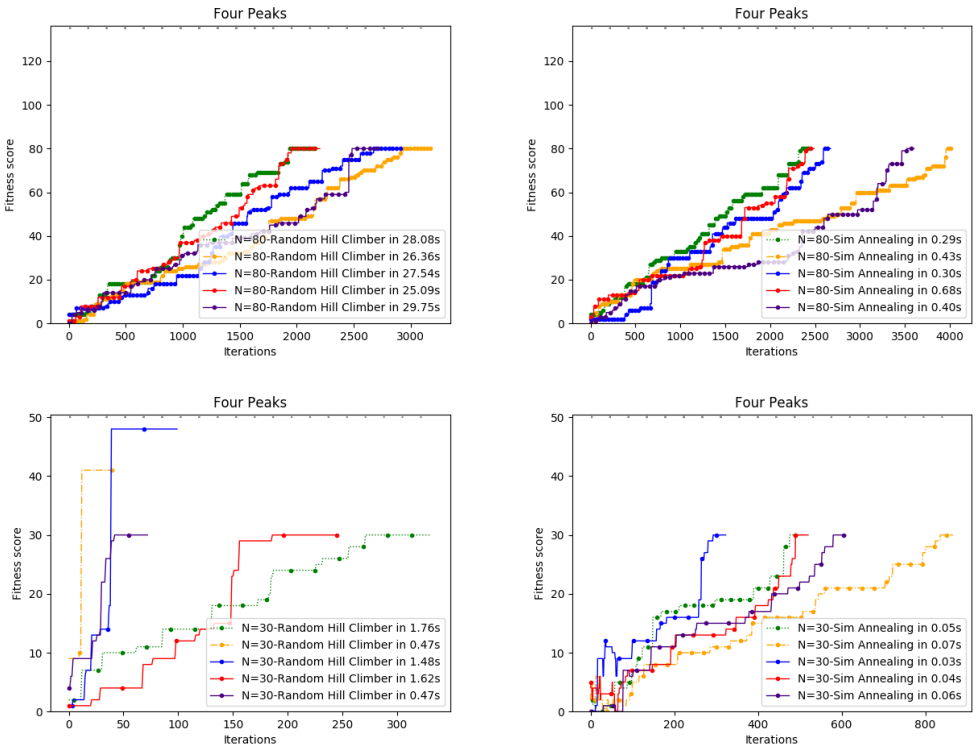

Figure 1: 1:RHC with N=80 - 2: SA with N=80 - 3: RHC with N=30 - 4: SA with N=30

To get the bonus and reach the global optimum, the optimizer should be able to change several bits at a time. RHC can't do that. SA can't either but it can accept bad neighbors before the T gets too low, so it's still possible to modify a few bits in a row, which can work, especially at low N. In the first two graphs in Fig 1, N=80 is already too high and we see how both SA and GA can't make those typical sudden vertical jumps when they find a neighbor with a much better score, but instead can only grow almost linearly, inevitably, towards a local minimum, as described, even with 250 restarts (which I implemented for all algorithms). There are small jumps, at best, when, say, tails grow and merge with a small cluster of zeros that were present from the start. Still, if the other side is not growing, this leads inevitably to a local optimum.

When N is smaller, say 30, like in the last two graphs, it is more likely to start with a well balanced initial state, with #heads and #tails close to T, in which case we may get the bonus quickly. That's why a few plots for RHC jump directly above the local optimum. It can happen to SA too even if doesn't show in Fig 1.4. However, the nature of those algorithms is incompatible with what needs to be done to solve this problem, so, even at low N (30), it is still difficult to reach the global optimum (the dotted line at the top) because as soon as there is significant imbalance, it's impossible for the smallest side to catch up.

This is clearly demonstrated by the blue line in Fig 1.3, that plateaus a few points from the optimum. It's an example of a local optimum, very close to the global optimum, but impossible to reach it. Increasing max_iters, max_attempts or restarts will have little to no effect since only luck (and only at low N), not the algorithm, can

produce a state very near the 'bonus barrier', ie heads and tails very close to it, or above it right from the start.

Fig 2.1 shows how RHC fades, in terms of function evals, vs SA. To generate it, I accumulated the function evals of all runs until the algorithm solves the problem, like in the MIMIC paper [1]. We can see that SA fades against the other two, even at low N, but that's not a valid comparison because they operate fundamentally differently, and one function eval of SA is really fast, but comparing GA and MIMIC is meaningful. Fig 2.3 shows a curve similar to what is found in MIMIC paper [1], ie an advantage to MIMIC as N increases. But I don't find that impressive because, if you look at the timings, MIMIC takes more time, so its function evals are heavier. I said it before, function evals should be used on one algorithm only. But, for sure, at this point we can conclude for FourPeaks, RHC and SA are out of the picture, even at low N.

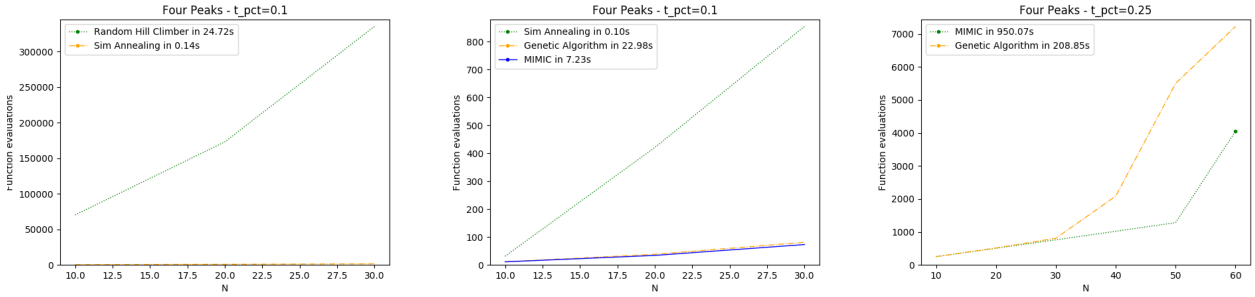

Figure 2: 1:RHC vs SA - 2: SA vs MIMIC and GA - 3: MIMIC vs GA

From Dr Isbell's paper [1], we can read "even in problems where fitness is obtained through the combined effects of clusters of inputs, the GA crossover operation is beneficial only when its randomly chosen clusters happen to closely match the underlying structure of the problem... which is a rarity... a fortuitous occurrence". That's what we have here: clusters of 1's and 0's on both ends. So, even if this problem was actually designed to show GA's strength (over RHC and SA...), I could catch all 3 stuck in a local maximum, as shown in Fig 3. It is a very nasty type of local minimum because it's impossible to get out of it, even by correctly flipping a lot of the right bits, ie more than N/2 and all at once... One could really call them 'basins of oblivion', like a Black Hole.

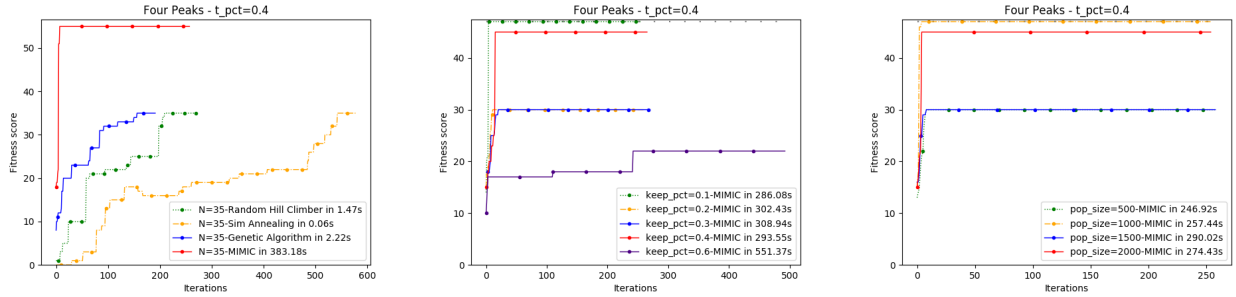

Figure 3: 1:MIMIC going straight to the optimum - 2: keep_pct optimization (N=30) - 3: Population optimization

Fig 3.1 shows that MIMIC shoots straight up, in a few iterations, which is impressive. It is not a fluke, I have observed this phenomenon very often. That's what led me to do the following analysis. One can only conclude that it fits with the structure of this problem very well. Unlike RHC and SA, it carries information through steps, to very quickly improve the fitness score, get the bonus and fly to the global optimum. If you look at the title, you'll see that I have increased t_pct to 0.4 in order to make it harder for the other algorithms and expose their weaknesses. To be honest, GA can also do very well, but I wanted to show a clear algorithmic superiority of MIMIC for this problem. In terms of execution time, both are quite close, with MIMIC usually

faster, because a Genetic Algorithm can take some time to reach an optimum, global or local. See the stair-case like nature of GA in Fig 3.1.

I have done some parameter optimization, and, as Fig 3.3 shows, the ideal population for N=30 is around 1,000, and taking more is detrimental since it can lead MIMIC into local optima more frequently. You'll also see in the code, that I have used quite large values for the HP's for all algorithms in order to not skew the results. Also, I have modified Mlrose-hiive to add 'iterations' on SA, GA and MIMIC to be able to run them multiple times, but not 100 times like RHC, just a handful, to have more robust results.

We're not done yet because, as I said, GA also performs well, so I had to find a good reason to declare MIMIC the winner. I have set the stage for a final comparison between MIMIC and GA. I'm going to run them both over different values of N, with 250 max_attempts, population=1000 and the same probability. These values are not optimal for every N, but they are good enough to demonstrate the clear difference between MIMIC and GA.

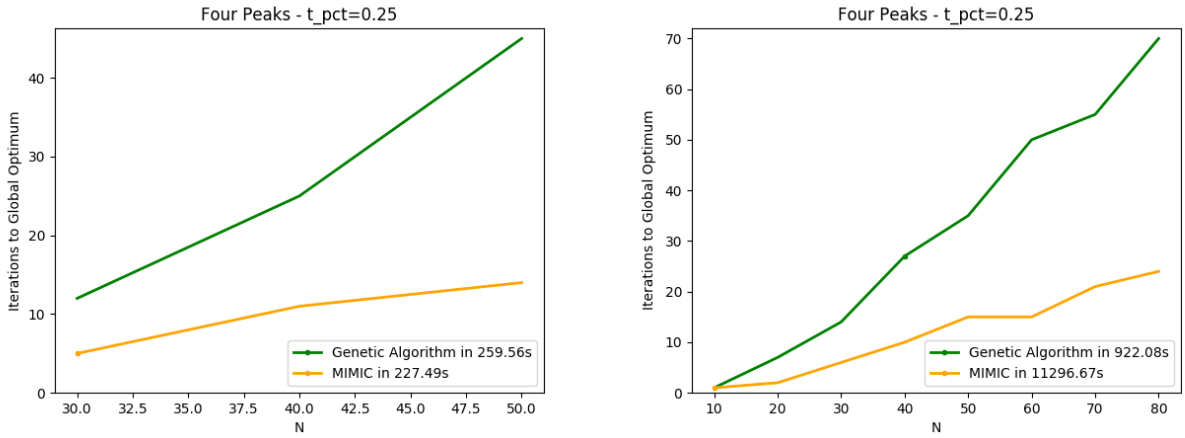

Figure 4: 1:MIMIC reaches the global optimum in less iterations (times are for the biggest N)

As we can clearly see in Fig 4, GA takes significantly more iterations to reach the global optimum. It is really impressive that MIMIC can get to the global maximum in such a few iterations. Even if it does a lot of things at each iteration, the timings in the graph show that both algorithms took more or less the same time. Well, with N=70 and 80, MIMIC started to show some weakness, as one can see in the timing.

To be totally honest, unlike Fig 2 where I accumulated all the function evaluations, even the unsuccessful runs, this time I rejected the cases where both algorithms converged to a local optimum, and counted only the first time they hit the global optimum. It happens to both algorithms, especially at higher N because this problem is quite hard, and I made it harder by taking t_pct=0.25. In

However, the trends in the graph are clear: not one time did GA reach the optimum in fewer iterations, otherwise the lines would cross. No one can claim that MIMIC got the perfect initial state vector every time. To me, this indicates a superior algorithmic capability of MIMIC for this type of problem, where the solution is very structured. In practice though, GA can really challenge MIMIC and it is hard to pick one over the other, but we were asked to "highlight advantages of our algorithm", and I've found one aspect of MIMIC that is better.

3 Continuous Peaks

This is an easy one. The analysis is quite straightforward. By the way Continuous peaks is built, there are more potential basins of attraction compared to Four Peaks, and here Simulated Annealing really shines due to its ability to 'jump around', like RHC, but not give up too quickly, accept a few bad neighbors, to be able to

continue looking for better ones. Fig 5 tells the whole story (if one also looks at the times in the legend).

In Continuous peaks, there are 'blobs' of 1's and 0's everywhere, and although the fitness function is still a maximum, it accepts blobs situated anywhere in the binary string. In other words, it is possible for SA to connect blobs of the same value by flipping the few bits between them, and suddenly a new, much bigger blob appears. This was also possible in FourPeaks but to be meaningful, it had to happen on both extremities only. Here, SA can flip bits randomly, and connect blobs. RHC can do that too, but the distance between two blobs has to be one single bit, while SA can flip several at a time, even some that don't increase the fitness function immediately.

This is a crucial advantage over RHC, and also MIMIC and GA, which flips maybe too many bits at a time, which can totally destroy blobs. We can say that this problem is less structured than FourPeaks, so it's no surprise that MIMIC doesn't shine here. It is 1000 times slower... I think the best strategy to solve this problem quickly is: flip just a handful of bits to connect blobs, slowly but surely, to avoid breaking big blobs too much. SA seems to do that perfectly. Even if it breaks a blob, it's only by a few bits, and it can be rebuilt stochastically later. In other words, SA flips bits one by one and tests each modification, while modifying lots of bits at once (and doing a lot of damage), or resampling a big part of the population don't seem the optimal things to do.

Although all algorithms managed to reach the optimum, as Fig 5 shows, SA got there blisteringly fast, 50 ms, ie 30 times faster than RHC and 20 times faster than GA. It is interesting to see how MIMIC and GA get, again, much closer to the optimum right off the bat, in a few iterations, no doubt due to their capacity to modify several bits at a time. But maybe, at some point, they flip too many bits, and I think that is not the optimal strategy for this problem as I just explained.

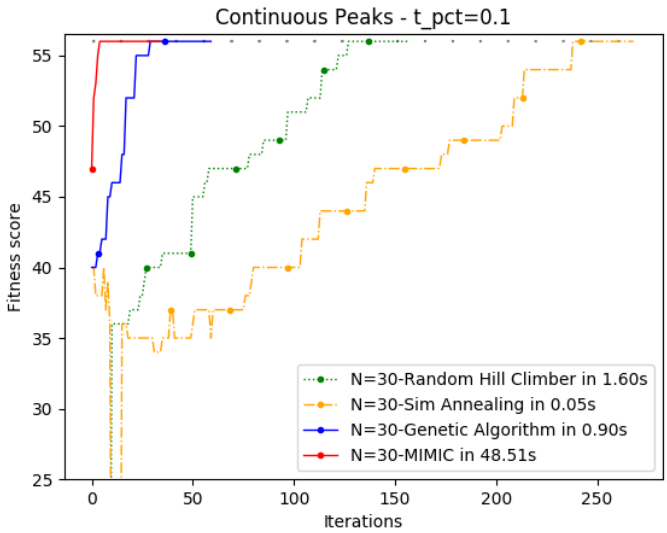

Figure 5: Continuous Peaks performances

There isn't much more to say here. You'll find in the comments of my file the various runs I did to try to optimize GA and MIMIC, but I basically got the same runtimes as in Fig 5. The advantage of SA is such that I don't think there's a need to look at fevals here, or anything else to show the advantage of SA. This type of problem seems to be made for SA. I think it's useful to look at function evals when algos do a lot of thing under the hood at each iteration, but here SA basically calls the fitness function, and does an exponential. Even if SA does 10 times more func evals, it does them so much faster, that I'll always pick it for this type of problem.

4 Custom J-Peaks

Now I have to prove that GA can beat the other algorithms in some situations. It seemed easy at first, I tried several of the classical problems but MIMIC kept beating GA in problems with structured solutions. In problems with no structure, SA beats them both, so the problem is harder than it looks. It seems that TSP is a possibility, but I was not excited to do the same thing again, ie change the name of the algorithm and run the same code, do a gridsearch etc... In fact, looking into how Fourpeaks was conceived to beat RHC and SA by luring them in nasty local minima, and how SA came with a revenge in Continuous peaks really gave me a good insight in how these algorithms work in practice.

So, I decided to try making my own fitness function for a made up problem on my own. At first, I couldn't conceive a problem where GA was beating MIMIC, and keeping SA at bay. MIMIC is really powerful... It was frustrating, but at the same time, it was fun and instructive to try to outmsart those algorithms by, say creating different types of basins of attraction, especially local minima to try to trap one algorithm but not another. It's a tall order because if you're too close to Fourpeaks or to Continuous peaks, GA doesn't stand out. I even tried a problem where the solution roll-shifts by a random number of positions in both directions at every fitness evaluation. It was possible for the fitness function to know what the solution was since I passed the function that created it as a parameter.

Believe it or not, it was not hard enough, and even SA could solve it at low N as we can see in Fig 6. It is really interesting to see that all algorithms find the solution despite it's constantly changing. One can imagine that MIMIC and GA have a big enough population to correspond to any shift by the fitness function, so once they reach the global optimum, they stay there. It's more of a mystery for RHC, but it has to be because of the 250 allowed attempts, which allow to find the solution after every shift. SA, on the other side also finds the optimum in a few iterations, but then loses it as shows the choppy line that keep touching and leaving the optimum level. It has to become much harder for SA and RHC at higher N even with 250 max_attempts.

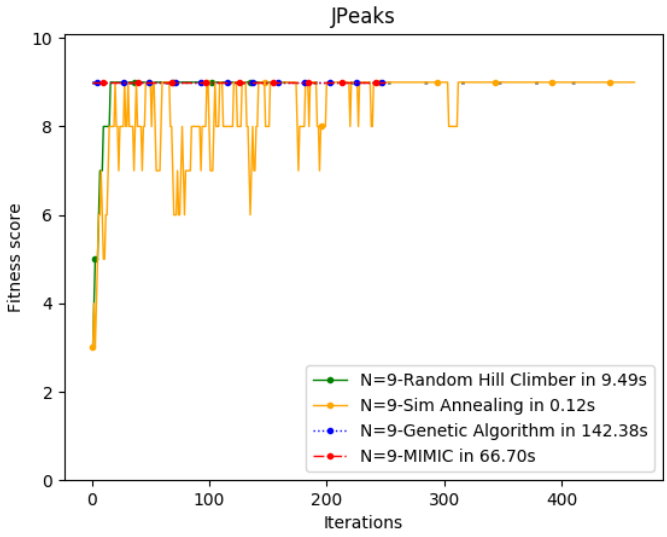

Figure 6: JPeaks with Random barrel-shifting

This was really interesting, but depressing because I thought randomly changing the solution was going to create a wedge between MIMIC and GA. I almost gave up, but I re-read what they were saying in the MIMIC paper, about the only situation where GA should work better: "GA crossover operation is beneficial only when its randomly chosen clusters happen to closely match the underlying structure of the problem or can be easily determined beforehand by the programmer." This gave me the idea of a dynamic problem, where the optimal solution changes, in a way similar to the kind of thing a crossover function would do.

In fact, I did the same as in the OnePointCrossover function in Mlrose, ie select a cutoff point randomly, and then split the initial state in two parts at that value, and swap them. Of course, the cutoff point that GA would pick would be different, but GA would 'match the underlying structure' of my problem. And it worked! Even with UniformCrossover! Fig 7 plot 1 & 2 shows that GA doesn't move until it hits the moving target and jumps straight near to the global optimum and, more amazingly, stays there, which can only be due to the large population. MIMIC, however, got really hurt by this problem. We can see it in the first and second graph, how it seems to quickly jump to a position close to the optimum, and then loses it and the fitness score crashes down to zero. Amazing! I managed to sabotage MIMIC as I wanted.

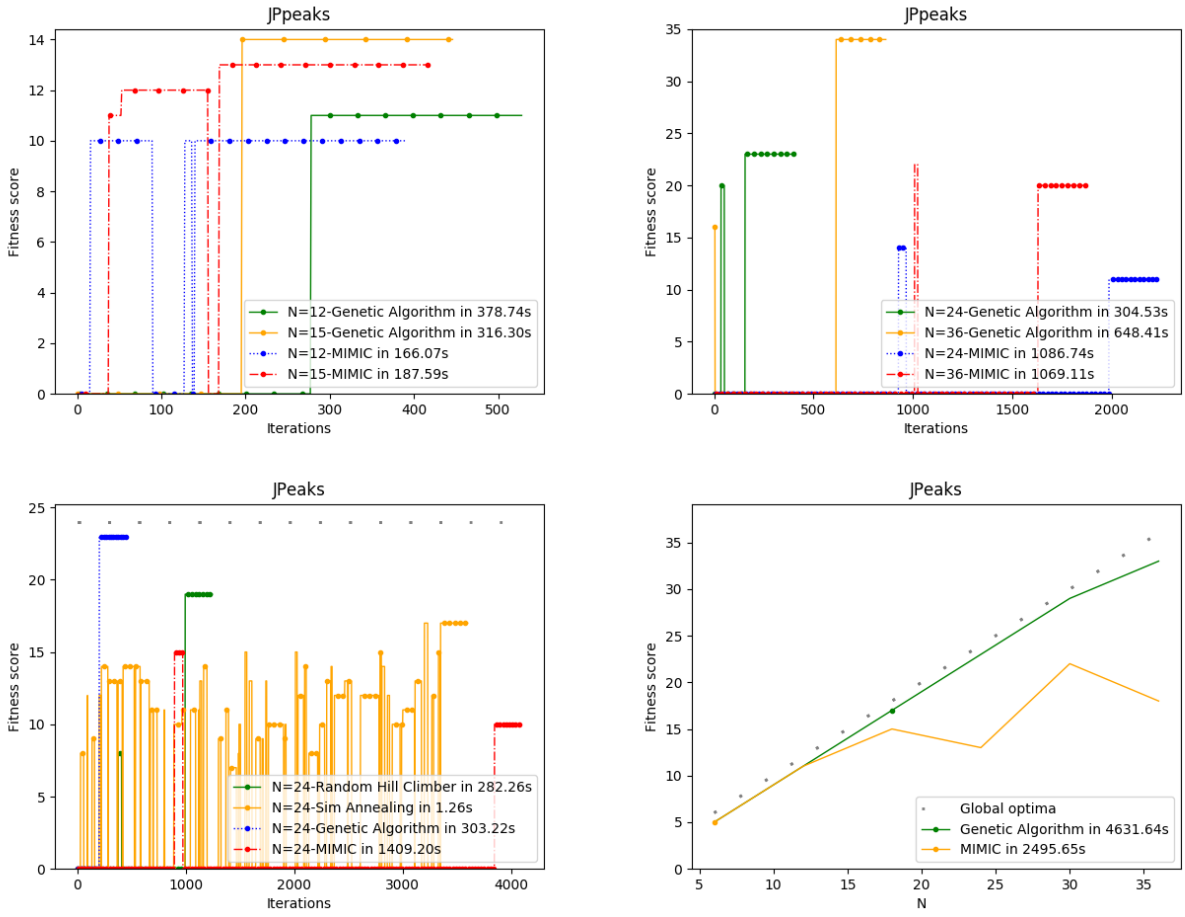

Figure 7: JPeaks with dynamic Crossover

I tried to optimize the HP of all algorithms via gridsearch but here, the problem is structural. Max_attempts would have to increase exponentially with N for RHC and SA to stand a chance. Fig 7.3 tells the whole picture, ie all algorithms with 250 max_attempts keep grasping at straws, except GA which get very close to the optimum in a time comparable to RHC...

The graphs also reveal another facet of those algorithms. GA finds the solution, even if it's changing, by its sheer power of crossovers (mutations probably don't play a big role here), and then jumps straight from zero to close to the optimum. I thought I had a bug in my program because it always stops 1 unit under the optimum, but higher N's indicate it is really like that. I haven't found an explanation to that. Of course, it is hard to hit a moving target, but I was expecting it to be possible at low N's anyway. Interestingly, MIMIC also has issues at low N. On the other hand, MIMIC, even if it is not close to the optimum, keeps trying, re-assessing the situation at every iteration, trying to find the best population possible. We see it crashes to zero at times, and

then tries again, always with big vertical jumps, like we analyzed in Fourpeaks.

Also interesting, is that, in the end, both MIMIC and GA manage to stay at a 'good' level, without crashing, but I'm sure it comes from the rich population. We can imagine that parts of the population correspond to every possible value of the cutoff, as if there were N different sub-populations, to cope with the moving target. This is, of course, all theoretical, but it manages to expose the strength of GA, even if it is for a very unlikely type of problem, as the paper says. The fourth plot shows that, no matter the length of the state vector, GA stays very close to the optimum values (mysteriously never reaching it), while MIMIC keeps doing worse as N increases. I could probably improve that by giving them both a higher population, but it would only postpone the problem, because it is fundamental here. I think all this fits the requirement quite nicely, and it was way more interesting that doing a standard problem. Trying to make an algorithm fail is actually a very insightful exercise.

5 Neural Nets Weights Optimization

I had never thought of using Random optimizers to search for the weights of a NN, but it works quite well. At least for my relatively 'easy' data set from P1 (the Wisconsin breast cancer data [2], a slightly imbalanced binary set with 600 instances and 10 features). Before starting, I verified that both Mlrose and scikit-learn were using the same loss function (Softmax) and the activation function (Relu). I used the same Neural Net from P1, with 25 nodes in one internal layer, and I got nice results almost at the first try.

I coded up gridsearch to be able to scan many HP, so I ended up with good results quite quickly. Here, I'll discuss RHC, SA and GA at the same time since I'm more familiar with their strengths and weaknesses now. In P1, the best F1 accuracy of my dataset was 97.6% after boosting. Here, I'm reaching a F1 score of 96.7% with RHC in 43s, 96.5% with SA in 22s, and 97.7% with GA in 509s. These numbers are quite good, but I wouldn't recommend GA for searching NN weights because it's 25 times slower than SA as shown in Fig 8.

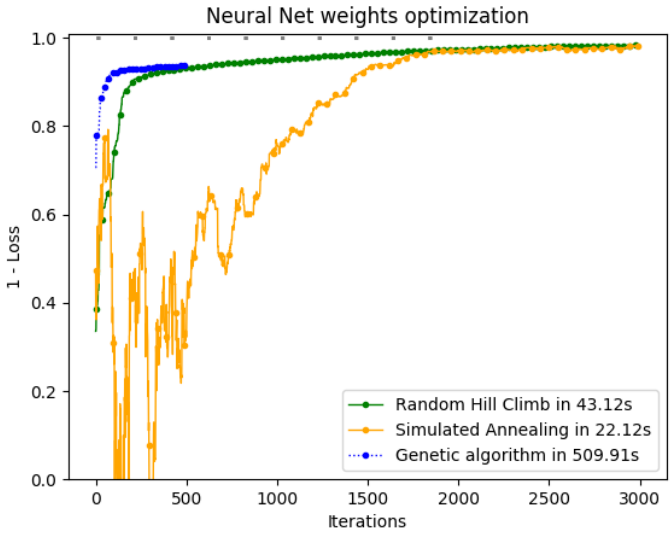

Figure 8: Convergence of Loss functions

RHC shines here, with SA being a close second. Mlrose returns the loss, so I took 1-Loss to have a rising metrics as requested. RHC shoots right from the start and converges steadily, without hiccups. At least with this dataset. GA too but is way too slow. SA is really choppy at the start, and I don't like that because this is an easy dataset, which is why I prefer RHC which seems to be able to converge in a steady way. After seeing this graph, and from my experience with the first part, I'd be inclined to conclude that this type of problem has no structure at all, to the point that it's not beneficial at all to keep bad neighbors like SA does. If I had more time, I'd test this on my second dataset, which was much bigger and nasty.

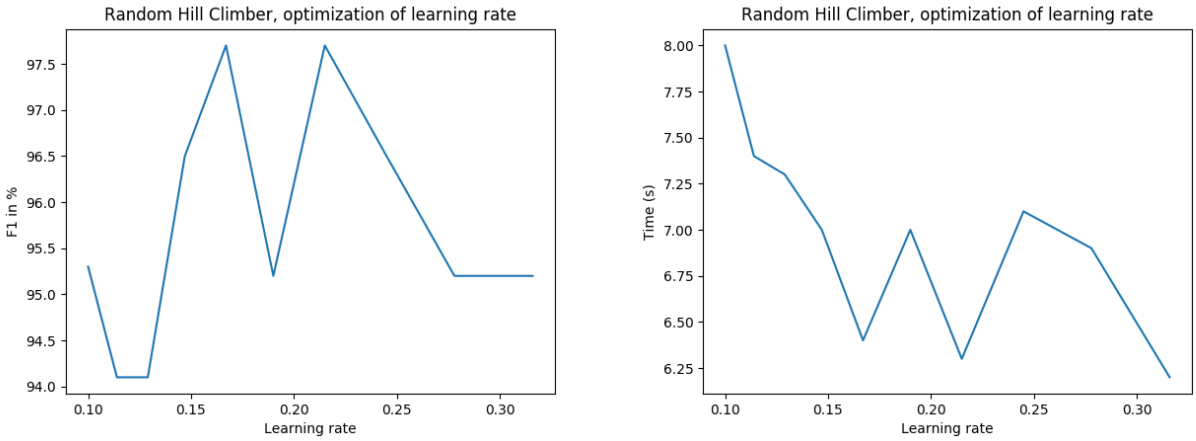

Figure 9: Optimization of learning rate

There are lots of HP that one can tune, the learning rate for sure, and some specific ones like the T decay for SA, or the population for GA. In fact, GA was extremely slow at the beginning, and I had to reduce the population size thanks to gridsearch. Here, I'll show the optimization of the learning rate for RHC, in Fig NN5. The first plot shows that there are two possible values to reach 97.5%, ie 0.16 and 0.21. The second plot shows that increasing the learning rate decreases the time, so 0.21 is preferable. It is not super critical here, but it's a nice example of having to take into account of various parameters durin HP optimization.

Boost

	0	1
0	94	1
1	1	41

97.6%
1.63 s

NN P1

	0	1
0	93	2
1	1	41

96.5%
0.1 s

RHC

	0	1
0	93	2
1	0	42

97.7%
43 s

SA

	0	1
0	93	2
1	1	41

96.5%
22 s

GA

	0	1
0	93	2
1	0	42

97.7%
509 s

Table 1: Confusion tables, F1 scores and convergence times

Table 1 sums up the performances of our algorithms but also some from P1. We can see that the confusion matrices are pretty much identical, with very few false positives and false negatives. However, scikit-learn using gradient descent is so much faster. I turned Mlrose's NeuralNetwork into a Mklearn classifier so I could re-use my code from F1 to generate learning curves, which revealed much slower times, which mean that the Mlrose-hiive could be improved in terms of performance. So, let's take a look at RHC and SA learning curve. I'm going to lack time and space to do GA too, but I'm not sure it was a requirement. And GA is not recommended anyway. First let's look at the learning curve of our NN from P1. It was quite nice, with very small bias and zero variance.

In Fig 11, we can clearly see that both models overfit because the blue line is almost at 1 even with the whole population, while the validation set stays around 95%. It's not horrible, but I would have liked to see that blue line come join the green line, line in Fig 10. There are a few solutions or ideas to try to remedy to this.

First of all, I did not do cross validation, which is an important tool against variance. Also, I could have done more complexity analysis, like in P1. Also, scikit-learn NN uses a parameter 'alpha' to do regularization, which is not possible in Mlrose. Also, maybe compare the weights from P1 to those found here, to see if we don't see some strange pattern (not an obvious thing to do but worth trying. The learning curves seem to indicate that if we had more data, maybe the curves would get closer to each other. In my case, maybe it would be possible

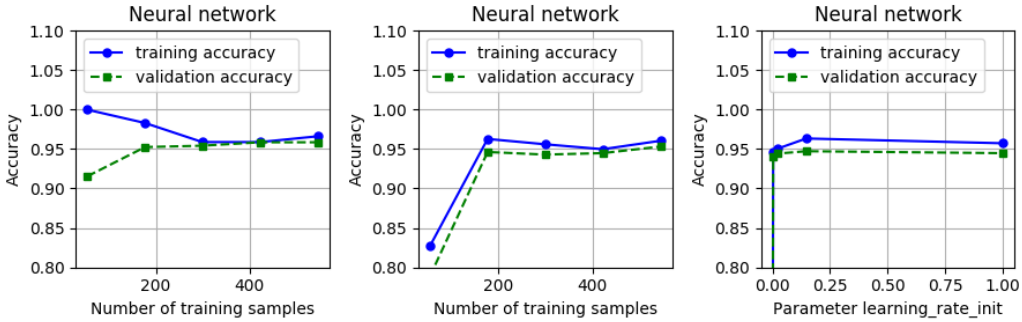

Figure 10: Learning curve before fitting, After and Complexity analysis

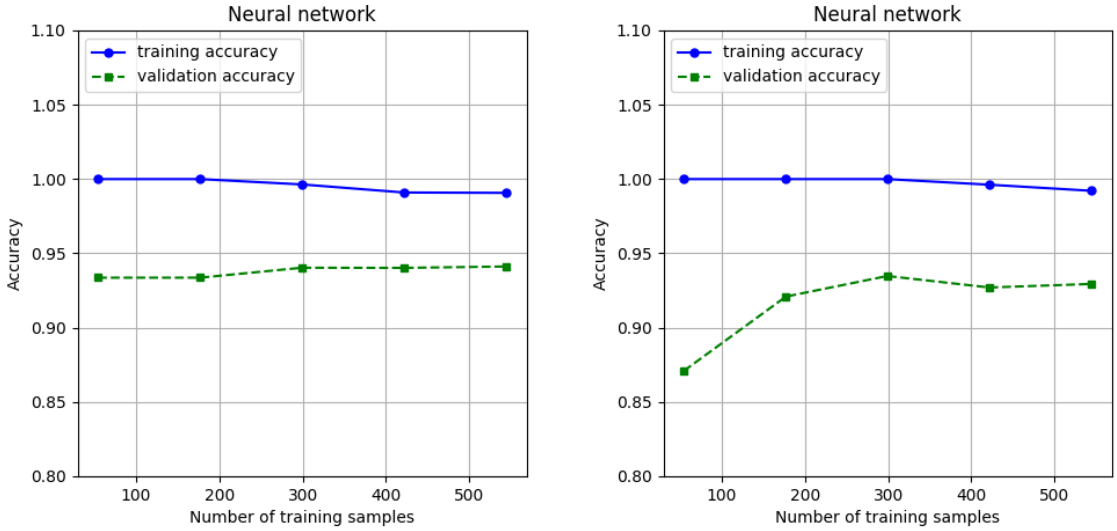

Figure 11: Learning curve after weights optimization with RHC and SA respectively

to find other databases on breast cancer. Another idea would be to add bias and play on the bias / variance tradeoff, but I'm not quite sure how to proceed concretely here.

I kept for the end the question about using binary state vectors to represent weights which are continuous. It immediately comes to mind "Isn't it what everyone on the Planet is already doing? I mean, on computers, floats are encoded on 64-bit. So, after thinking twice about it, I don't see the problem. The only question is how we link any array of bits to something imaginary, like numbers. Well, whatever meaning we put in a specific binary notation, those randomizers will find it without knowing what they're doing. Remember my first attempt in JPeaks, where I was randomly shifting the solution, and yet the algorithms could find it.
In the N-Queens problem, I had assumed that each column was represented by $log_2(N)$ bits. The algorithsm didn't know that, and yet it worked because I think it doesn't matter. Whatever is the convention on the bits to encode some type of information, they'll find it. Also, now that I think of it, the algorithms can select any bit configuration, either by flipping them one by one or in group. To me, it means that they can shatter a the hypothesis space of these models, which are all based on a finite number of bits, so the VC dimension should be finite, and the problems are PAC learnable.

6 References

[1]De Bonet, JS., Isbell C, and Viola P. "MIMIC: Finding Optima by Estimating Probability Densities". Massachusetts Institute of Technology, 1997.