# A genetic addition to Fiduccia-Mattheyses

*Jan Stefan Kowalski* - October 3, 2020

`https://github.com/JanSKowalski/Fiduccia-Mattheyses`

Can I translate an engineering paper into working code?

Can I design a genetic algorithm?

Can I improve upon an algorithm to produce meaningful advancements?

”How do you place cells on a chip so that the area used for interconnect is minimized?”

This motivating question introduces an aspect of computer hardware engineering. Microchips function because of the many logic gates that make up their circuitry. Larger circuits with more logic gates are more capable, but occupy more space. The design complexity of a chip is limited by its internal area, so efficient use of that area is highly desirable.

Logic gates performing a specific function together can be grouped together as a cell. Cells communicate with each other through nets known as interconnet. One interconnect net can connect to multiple cells, and any one cell can be a part of multiple distinct nets. Unlike cells, however, interconnect has a variable area highly dependent on the placement of cells within the chip. Two cells distant from one another require a longer net, using more area than the interconnect between two close-by cells. So, assuming that all cells and nets are necessary, our problem of area reduction becomes one of positioning cells so that the number of long nets is minimized.

Using the terminology of the partitioning problem, we are looking to divide the cells into one of two partitions. The number of nets connecting cells across partitions (known as the cutstate/cutsize) is of great importance, and serves as a proxy measure for the number of long nets. Nets within a partition are considered localized and use a small area, whereas nets between partitions can potentially be very large and take up a lot of space. There are many approaches to the partitioning problem[1]; in this report we focus on Fiduccia-Mattheyses (FM).

This report and its accompanying code rely heavily on the 1982 IEEE paper[2] in which Messrs. Fiduccia and Mattheyses introduced their approach to placing cells. By carefully reading through the paper, we were able to reproduce the algorithm's storage and cutstate improvement mechanisms. Eighteen circuits from IBM's ISPD98 Benchmark Suite[3] were used to test our code, ranging in size from the 12,000 cell ibm01 to the 210,000 cell ibm18. Given that the code was compiled with modern optimization (gcc 4.8.4) and run on a modern CPU (intel i7-4700MQ), it is not surprising that our results far exceed those described in the original paper. Rather, the test of a successful implementation is the linear use of time by the code with increased circuit cell count[1], which we show in Figure 1.

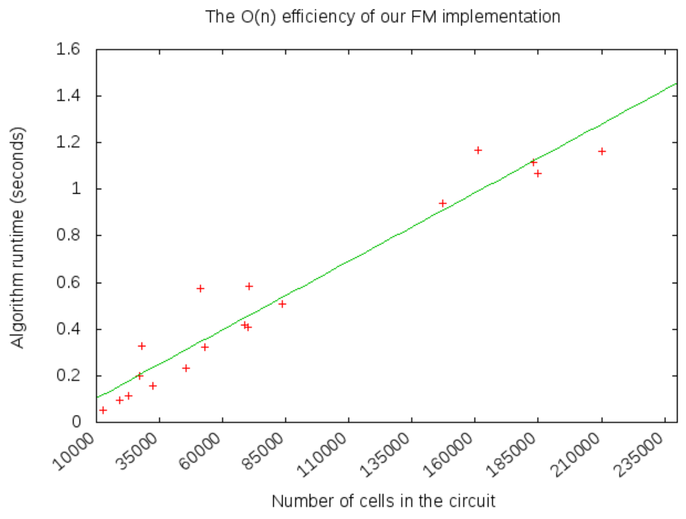

Figure 1: A linear increase in time with cell count

[1]Technically FM should be linear with the number of pins (cell-net connections) in the circuit, however the author did not find a significant difference between the two measures in practice.

Let's walk through the FM algorithm. We start by allocating all the cells into either one of two partitions. The paper doesn't specify how this should be done, so let's do this randomly for now. After generating random partitions, we eventually create two whose total areas (the sum of individual cell areas) are close to what we want, within a tolerance. The two partitions are now considered 'balanced,' and their areas will remain within that tolerance through the completion of the algorithm.

The purpose of FM is to reduce the number of nets in both partitions by reallocating cells. You can calculate that number (the cutsize) by checking every net for the presence of cells in each partition. Likewise, you can decide which cell to move by considering every potential movement and ranking cells by a 'gain' value. These are, however, very time expensive calculations to perform. Moreover, this approach was already incorporated in the Kernighan-Lin algorithm[4] a decade prior to FM. FM is important because it maintains the correct cutstate and gain values *without* repeated calculations, saving time in turning an $O(2^n)$ problem into $O(n)$.

FM accomplishes this feat through careful bookkeeping. Those expensive calculations are used once, at the start, to provide an initial global cutstate and individual gain value for cells. On every pass, the highest-gain cell that preserves balance iis chosen as the base cell to switch partitions. FM then updates only those nets directly connected to the base cell. Cells on those nets that would reduce cutstate if moved have their gains incremented. Cells that lose that ability have their gains decremented. Cutstate is modified incrementally as individual nets are accessed, keeping tally across passes without needing a complete recalculation.

The uniformity of the results across datasets surprised the author, who subsequently wrote several verification functions looking for the nonexistent error. The example shown in Figure 2 could, with a scaled axis, have been taken from any of the benchmark circuits. FM undergoes a period of steep decline, followed by a shallower drop until the apex is reached. At that point the algorithm runs out of favorable high gain cell swaps. Cell movement continues with increasing cutstate until every free cell (that doesn't violate balance) has been moved.

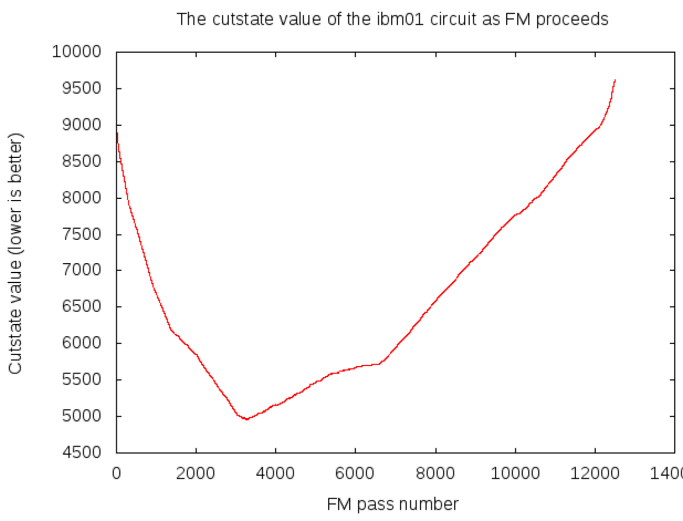

Figure 2: A common output curve

We notice that the passes after the apex are extraneous. We can introduce a condition stop the algorithm if the current cutstate exceeds the lowest cutstate by a predetermined amount. We see in Figure 3 that the cutoff mechanism improves the execution time of every benchmark. Note that too low a cutoff value will prevent normal variations that occur in the algorithm as it climbs out of local minima. A cutoff of 20 was sufficiently high for the largest variances during testing.

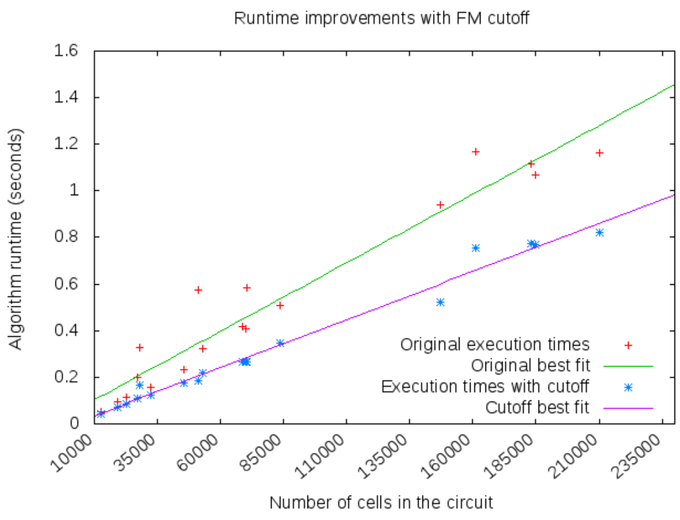

Figure 3: The benefits of cutoff

Perhaps a natural next step in minimizing the cutstate is the application of FM multiple times to the circuit. We saw in Figure 2 that a single run cuts the cutstate by about half. If we pass the output of one run as the initial partition of the subsequent run, we would expect an exponential decay ($0.5^n$ decrease in cutstate for n runs). Experiment validates this intuition, with Figure 4 showing the decreasing cutstate with application of FM. The partition information is only copied and stored once – at the end of the algorithm – to minimize execution time. This approach is acceptable because out cutoff addition guarantees that the final partition is within cutoff-threshold edits of the lowest-cutstate partition encountered.

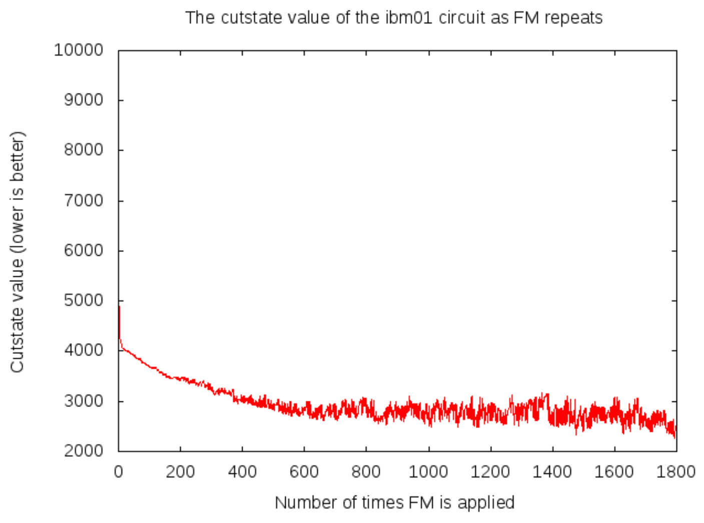

Figure 4: Exponential decrease in cutstate with repeated FM runs. (The initial cutstate is 9122)

After the first hundred runs we see the cutstate climbing and dipping within a range of values. To explain this, consider a solution space of all possible cell configurations (either partition A or B). We define distance on our solution space as a measure of similarity, with single-cell differences being the closest. The algorithm has a specific approach to looking for nearby solutions due to FM's gain value mechanism. It only considers solutions that can be accessed by partition-switching the highest gain cell, followed by the next highest, until no beneficial cell-swaps are available. FM is able to execute so quickly because it considers a limited number of solutions.

The volatile behavior indicates that the solutions immediately accessable to FM are exhausted. Accessing the next lowest cutstate solution requires travel through solutions with poor cutstate. How do we address this? One method is to increase the number of solutions considered by FM. Attempting to access every nearby solution and consider every possible series of edits to our partition isn't reasonable. Far too many resources would be consumed to be practical. What we can do instead is *sample* the nearby solution-space for low cutstates with genetic algorithms.

Our solution-space has many peaks and valleys, making it difficult to find the global minimum, and even harder to generalize sample results. Gradient descent methods are likely to get caught in a local minima unless they have momentum or annealing mechanisms. In situations like ours, genetic algorithms (GA) are a popular choice.

GAs work by pseudo-randomly sampling the solution-space near solutions known to be good. These samples are generated by recombining two or more good solutions and mutating the result. A cost function is then applied to determine the quality of our offspring, getting rid of the bad ones. Ideally, this cost function would correlate to the final cutstate output of a single FM pass on the offspring. Unfortunately, there is not (to the author's knowledge) a clear indicator to determine which partitions will perform well. In the absence of a true cost function, the author uses the offspring's initial cutstate as a measure of fitness. We note that, as shown in Figure 5, the initial cutstate doesn't strongly correlate to a low final cutstate on the first FM pass. Future work should endeavor to identify such indicators and encorporate them into the cost function.

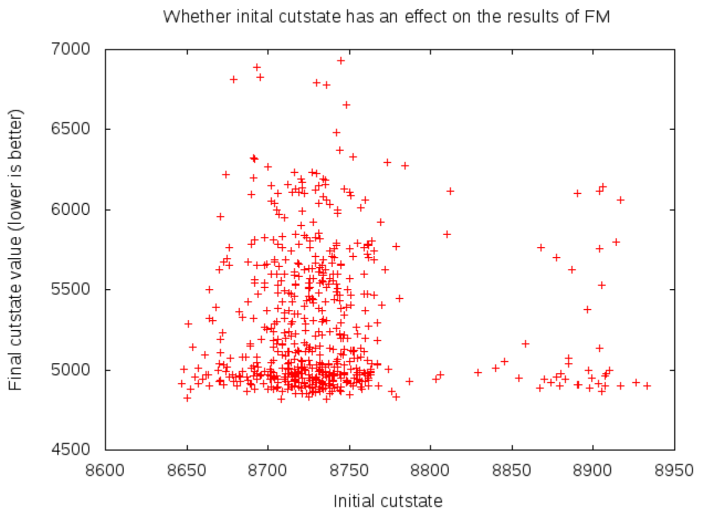

Figure 5: For a single FM pass, the initial cutstates do not strongly affect the results. Most trials cluster around 5000 – half the initial cutstate. The final cutstate values above 5000 are attributed to the FM cutoff mechanism

The recombination process for solutions is a straightforward one. We are interested in whether a particular cell belongs to either partition A or partition B. Every solution is an array of size num_cells containing either 0 for p.A or 1 for p.B for individual cells. In recombination, we copy part of one array, and insert the remainder of a different array.

Most of the time this operation leaves us with a bad, high cutstate result. It's the repetition and pseudo-randomness of this process that eventually leads to a successful result.

If we can recombine two arrays by switching at one point, why not increase the frequency of switching? Or add a third solution to copy from? We can alter any number of aspects of the genetic algorithm to improve our GA performance[5], but in the spirit of prototyping a working algorithm, we will only add recombination variability to the GA.

Now that we can change the recombination frequency, let's choose two values (3, 160) and see how it impacts the GA. In Figure 6, we see that limiting the recombination frequency to a max. of three decreases the final output of the GA by an average of about 50 nets. From this we know that we prefer the GA recombination parameter to be set at three, *barring any change to the other parameters.* This is important. We cannot assume that three will be the better value if, say, the number of GA passes change. Nor can we make any claims about how the output is affected by values 4 through 159. We can conjecture that larger recombination frequencies will lead to a less effective GA, but claims without tests remain unsupported.

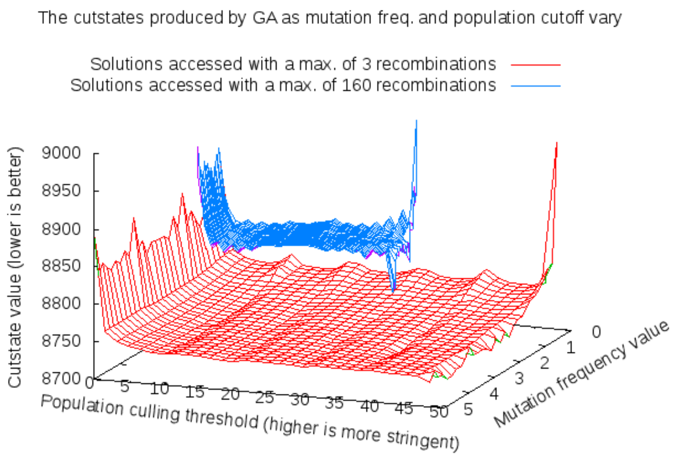

Figure 6: Hyperparameters can have a large impact on the output of the GA

For our simplified GA, we have to think about five parameters: population size, population culling threshold, mutation frequency, recombination frequency, and the number of GA passes. Population refers to the number of offspring solution, and the culling threshold determines how many solutions impact the next generation. The mutation frequency introduces point mutations to the array, changing bits from 0 to 1 and vise versa. The recombination frequency determines how closely an offspring resembles its parent solutions, and the number of GA passes is the number of generations produced. These parameters are not independent of one another.

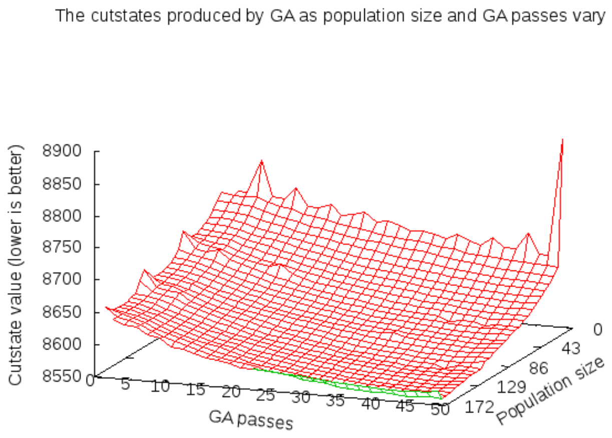

Figure 7: Increasing the number of generations and solutions in each generation leads to a lower cutstate

A rigorous approach, under the domain of hyperparameter optimization, would apply one of several algorithms[6] to the problem. Instead, the author had a bit of fun manually whittling down the parameter combinations with guesses and tests until further improvements to the GA output were negligible. One strong example of a trend is Figure 7, which shows that increasing both the number of solutions considered per generation (population size) and the number of generations (GA passes) decreases the final cutstate. Anectdotal evidence suggests that the cutstate can be indefinitely improved, at the expense of time. For a practical algorithm we will choose values high enough for good solutions, but low enough to be quick. In general, tradeoff of performance with time is common in genetic algorithms[5].

The results of all this effort are displayed in Figure 8 and Table 2. From the figure, we see that the GA does not completely mitigate the effects of an exhasted solution space. The rise and fall still occurs in late iterations, with some GA runs *increasing* in cutstate when compared to the FM repeat original. Most FM/GA output averages in the table are below that of the original, but within the standard deviation of both. From this evidence the author concludes the genetic addition to Fiduccia-Mattheyses as a qualified success.

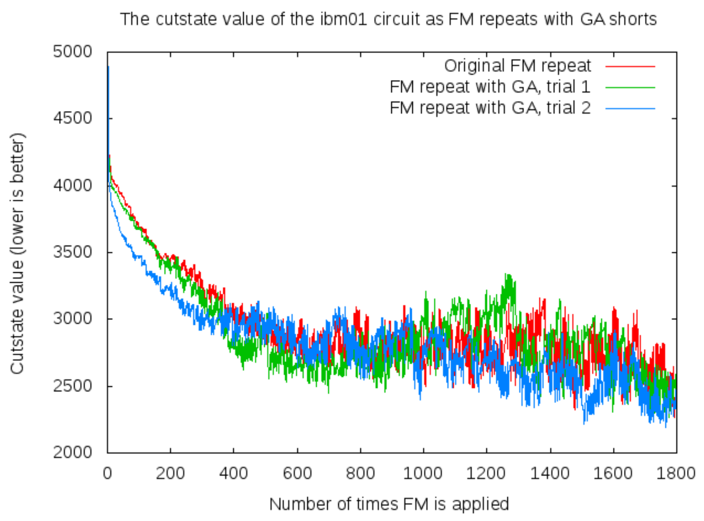

Figure 8: The GA has a small desirable effect on the final cutstate

## Closing remarks

Implementing the data structures in C took some time. The author feels justified in his choice to do so, as his FM implementation enjoys a fast run-time when compared to similar implementations in C++[7]. The project was a pleasure to work on and learn from.

## Acknowledgements

This report was typeset in latex. The figures were produced with gnuplot. Valgrind was used to detect memory leaks. I thank Prof. Alex Doboli for inspiring this work.

## References

[1] Kyu Lim, S. (n.d.). Partitioning summary, slide 7. Retrieved from https://limsk.ece.gatech.edu/course/ece6133/slides/partitioning.pdf

[2] C. M. Fiduccia and R. M. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions," 19th Design Automation Conference, Las Vegas, NV, USA, 1982, pp. 175-181, doi: 10.1109/DAC.1982.1585498.

[3] Alpert, Charles J. ISPD98 Circuit Benchmark. The ISPD98 Circuit Benchmark Suite, vlsicad.ucsd.edu/UCLAWeb/cheese/ispd98.html.

[4] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," in The Bell System Technical Journal, vol. 49, no. 2, pp. 291-307, Feb. 1970, doi: 10.1002/j.1538-7305.1970.tb01770.x.

[5] Haupt, R. L.; Haupt, S. E. (2004). Practical genetic algorithms. Hoboken, NJ: Wiley-Interscience.

[6] Yu, Tong, and Hong Zhu. Hyper-Parameter Optimization: A Review of Algorithms and Applications. ArXiv:2003.05689 [Cs, Stat], Mar. 2020. arXiv.org, http://arxiv.org/abs/2003.05689.

[7] Caldwell, A. E., Kahng, A. B., Markov, I. L. (n.d.). Design and Implementation of the Fiduccia-Mattheyses Heuristic for VLSI Netlist Partitioning (pp. 1-20, Rep.). Los Angeles, CA: UCLA Computer Science Dept. doi:https://vlsicad.ucsd.edu/Publications/Conferences/90/c90.pdf

| | Original FM | | Cutoff FM | |
|---|---|---|---|---|
| IBM Filename | Cutstate | Time (s) | Cutstate | Time (s) |
| 01 | 4971.82(53.28) | 0.058(0.0028) | 4979.74(64.5) | 0.043(0.0064) |
| 02 | 7472.4(507.79) | 0.11(0.027) | 7498.5(537.2) | 0.080(0.0035) |
| 03 | 9138.89(602.08) | 0.14(0.0074) | 9299.6(429.1) | 0.10(0.0052) |
| 04 | 11534.09(184.90) | 0.26(0.049) | 11589.63(173.5) | 0.16(0.032) |
| 05 | 10229.46(48.10) | 0.37(0.22) | 10229.1(45.8) | 0.36(0.28) |
| 06 | 12338.43(828.77) | 0.21(0.040) | 12511.85(756.7) | 0.15(0.010) |
| 07 | 17987.77(113.16) | 0.31(0.074) | 18027.12(132.4) | 0.23(0.037) |
| 08 | 17307.06(1488.19) | 0.79(0.17) | 17549.1(1240.0) | 0.39(0.095) |
| 09 | 23109.07(138.90) | 0.41(0.094) | 23118.22(135.9) | 0.30(0.060) |
| 10 | 29549.81(327.73) | 0.45(0.11) | 30054.38(1890.7) | 0.37(0.076) |
| 11 | 31338.89(191.56) | 0.41(0.013) | 31379.33(238.9) | 0.32(0.053) |
| 12 | 31670.03(289.04) | 0.56(0.020) | 31733.89(410.0) | 0.47(0.12) |
| 13 | 38409.32(172.69) | 0.60(0.10) | 38394.89(155.4) | 0.51(0.12) |
| 14 | 57687.72(218.41) | 1.27(0.24) | 64827.56(15557.6) | 0.82(0.15) |
| 15 | 74100.39(2270.62) | 1.69(0.38) | 75335.4(1835.8) | 0.93(0.15) |
| 16 | 76836.26(287.96) | 1.62(0.33) | 76738.75(202.4) | 1.18(0.24) |
| 17 | 81411.76(158.23) | 1.29(0.28) | 81389.39(141.3) | 1.03(0.25) |
| 18 | 77703.46(189.67) | 1.36(0.20) | 77702.95(326.0) | 1.17(0.27) |

Table 1: The results of 100 trials against each of the 18 testbench circuits. Lower cutstate is better.

We noticed that all FM results are roughly parabolic, so we implemented a "cutoff" that stops the algorithm when the results start to get worse. The higher cutstates in the cutoff results indicate that we set our cutoff criteria either too stringently (disallowing the algorithm to climb out of local minima on the way down) or too loosely (only cutting off after significant losses on the way up the parabola). An ideal cutoff mechanism would have identical cutstates to the original, with lower times.

Cutoff ends up saving between an eigth and a third of the algorithm's runtime.

The cutstate unit is nets, time is in seconds, and values indicate mean/standard deviation (n=100).

| | Repeated FM (250 runs) | | Repeated FM with GA (250 runs) | |
|---|---|---|---|---|
| IBM Filename | Cutstate | Time (s) | Cutstate | Time (s) |
| 01 | 3285.59(103.07) | 3.95(0.77) | 3275.30(119.03) | 4.53(0.21) |
| 02 | 5029.71(1286.44) | 8.44(1.16) | 4784.68(1368.64) | 8.35(0.35) |
| 03 | 5632.62(1008.06) | 11.67(1.66) | 5641.00(1032.30) | 10.38(0.59) |
| 04 | 7903.88(672.61) | 32.84(9.18) | 8089.55(493.11) | 13.13(0.85) |
| 05 | 8891.43(86.64) | 11.90(0.30) | 8871.79(79.19) | 13.52(1.52) |
| 06 | 8821.99(962.13) | 13.36(0.54) | 8437.63(1167.61) | 13.20(0.32) |
| 07 | 14019.27(455.29) | 21.08(1.07) | 14052.85(452.85) | 25.13(6.32) |
| 08 | 12124.57(3458.53) | 24.52(1.12) | 12669.01(2953.50) | 37.73(5.74) |
| 09 | 17445.12(442.74) | 24.68(0.53) | 17272.13(733.22) | 24.71(0.63) |
| 10 | 22739.74(2344.84) | 34.00(1.70) | 22843.01(2124.57) | 36.07(0.97) |
| 11 | 23412.67(509.69) | 53.52(5.58) | 23251.8(771.41) | 36.02(0.95) |
| 12 | 25838.83(1327.08) | 53.98(22.48) | 25487.83(1771.22) | 38.85(0.88) |
| 13 | 29244.89(572.49) | 54.67(3.73) | 29156.31(968.38) | 48.31(5.23) |
| 14 | 47256.36(531.65) | 83.69(9.38) | 47472.34(1249.39) | 93.92(10.69) |
| 15 | 55271.29(5118.60) | 100.14(17.25) | 54455.98(5308.67) | 85.89(2.40) |
| 16 | 63556.80(1529.57) | 105.44(8.71) | 63805.58(1319.20) | 139.05(62.47) |
| 17 | 71024.59(244.04) | 109.51(5.73) | 71063.07(213.37) | 272.67(3.31) |
| 18 | 64136.88(509.67) | 110.70(6.09) | 64265.88(423.81) | 200.35(87.45) |

Table 2: The results of 100 trials against each of the 18 testbench circuits. Lower cutstate is better.

The genetic algorithm addition only marginally improves the cutstate, with similar standard deviations to the original repeat. Using the GA on the larger chips nearly doubles the runtime of the algorithm, and produces roughly the same cutstate.

Some of the smaller chips seem at first glance to be quicker with GA, but one will notice that those instances are associated with abnormally high standard deviations in the original. The raw data shows one outlier in each instance with high runtime. Being unable to reproduce these outliers, the author conjectures that an unwitnessed computer suspension occured to skew the calculation.

The GA activated whenever the algorithm's current solution differed from the lowest recorded cutstate by five nets.

The cutstate unit is nets, time is in seconds, and values indicate mean/standard deviation (n=100).