Genetic Optimization Applied to Via and Route Strategy

by Zachary J. Zumbo

[S.B., E.E.C.S. M.I.T., 2018]

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

February 2020

© 2020 Zachary J. Zumbo. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Author: __________________________________________
Department of Electrical Engineering and Computer Science
January 28, 2020

Certified by: __________________________________________
Jacob White, Thesis Supervisor
January 28, 2020

Certified by: __________________________________________
Taylor Hogan, Thesis Co-Supervisor
January 28, 2020

Accepted by: __________________________________________
Katrina LaCurts, Chair, Master of Engineering Thesis Committee
Genetic Optimization Applied to Via and Route Strategy
by Zachary Zumbo
Submitted to the Department of Electrical Engineering and Computer Science

January 28, 2020
In Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

To meet increased demand and higher PCB design expectations, research engineers have been tasked to develop models to automate PCB placement and routing procedures using machine learning and artificial intelligence techniques. Since placement and routing are still tedious manual processes which limit the design search space, these techniques allow engineers to quickly investigate better solutions. The Move37 team within Cadence Design Systems found via placement to be a crucial problem to be solved and integrated into their OrbitIO platform. We evaluated multiple non-gradient-based optimization strategies and compiled data of their performance. From these tests, a genetic algorithm-based strategy was sought due to its fast convergence and the ability to substitute cost functions. In this study, we converted the via placement problem to a simpler layer assignment problem by enforcing the location of vias and pins to be the same. We then determined an optimal layer assignment given a set of flylines, i.e. logical connections, using a genetic optimization library called DEAP. We coined this genetic optimization approach to via strategy GO VIA.
# Table of Contents

1. **Intro** .......................................................................................................................... 5  
2. **Project Background** ................................................................................................. 6  
3. **Literature Review** ....................................................................................................... 7  
4. **Objectives** .................................................................................................................. 12  
5. **Problem Formulation and Algorithm** ......................................................................... 14  
   a. Assumptions .................................................................................................................. 20  
   b. Drawbacks .................................................................................................................... 21  
6. **Project Timeline** ........................................................................................................ 23  
7. **Findings** ...................................................................................................................... 24  
8. **Future Work / Suggested Improvements** .................................................................... 26  
9. **Conclusion** .................................................................................................................. 27  
10. **References** ................................................................................................................ 28  
11. **Appendix** ................................................................................................................... 29
Table of Figures

Figure 1: Move37 design flow. Bottom left - unplaced board, Bottom center - placed and viaed, Bottom right - placed, viaed and routed .................................................................................................................. 7

Figure 2: Results of four selected algorithms, genetic optimization, MkCS, augmented MkCS, and simulated annealing. Results were taken over three designs that varied in complexity; their sums are plotted. .................................................................................................................................................. 10

Figure 3: BGA fan-out - Note the local symmetry of the outer yellow routes ................................................. 12

Figure 4: Example PCB prior to flyline layer assignment. Black lines are logical, i.e. they are not yet routed .................................................................................................................................................. 14

Figure 5: Top left - Example PCB with colored layer assignment (before routing), Top right - table mapping each line to the lines they cross (not double counted), Bottom - Chromosome that represents the layer information of each line .................................................................................................................. 15

Figure 6: GOVia flow chart ................................................................................................................................. 19

Figure 7: Example of GOVia layer assignment for a randomly generated set of flylines (actual simulation) .................................................................................................................................................. 20

Figure 8: Intel Interposer manually routed, vs automatically routed using auto placement, auto via, and auto routing .................................................................................................................................................. 24

Figure 9: We tested how the chromosome mutation probability affected the convergence of GOVia, optimal range - 0.5 to 0.7 .................................................................................................................................................. 25

Figure 10: We tested how the chromosome crossover probability affected the convergence of GOVia . 25

Figure 11: GenOpt cost as a function of the number of generations completed ................................................. 26

Figure 12: Data compiled from literature review algorithm survey ......................................................................... 29
1. Intro

The concept of via placement is not new in the field of Electronic Design Automation (EDA). PCB designers were tasked with routing traces and placing vias on multiple layers as far back as the late 1940s when the first double-sided PCB was produced. (Brown, 2013) A vertical interconnect access, otherwise known as a via, was necessary to ensure electrical connectivity between multiple layers. Originally punched through the PCB in ‘plated through hole’ fashion, vias were used to connect through-hole components to a board with solder. Today, vias come in multiple types, including through-hole, blind, and buried types. These are primarily designed to route traces from one layer to any other rather than mount components, as most modern components are mounted using Surface Mount Technology (SMT). Even on relatively simple designs, packages and boards incorporate hundreds to thousands of vias to route connections.

As multi-layer PCBs have become increasingly common and as EDA tools have necessarily become more aware of complex designs, the problem of routing under stricter conditions has pivoted. Engineers must be aware of more specific design considerations such as signal and power integrity, EMI issues, high speed constraints, low latency, cost, etc. while EDA software must also understand these multifaceted and often related issues. Historically, PCB design engineers have decried automatic placement and routing tools as unprofessional, often faulty, and a crutch for those without industry knowledge. (Jones, 2004)

Currently, PCB EDA tools are still not well equipped to understand how component placement informs via placement, and consequently how via placement informs routing, unlike in similar IC place-and-route systems. However, improving each stage independently and providing a tool the ability to feedback design information using ML techniques would significantly increase the efficiency of design workflow. Ideally, whole designs would be synthesized with minimal human interaction in hours rather than weeks or months. This has widespread implications in the production of military and consumer electronics and could potentially change the timescales for product releases from quarters/years to weeks/months.
2. **Project Background**

“Cadence Design Systems, Inc. announced (on June 24th, 2018) that it was selected by the Defense Advanced Research Projects Agency (DARPA) to support the Intelligent Design of Electronic Assets (IDEA) program, one of six new programs within DARPA’s Electronics Resurgence Initiative (ERI) to use advanced machine learning techniques to develop a unified platform for a fully integrated, intelligent design flow for systems on chip (SoCs), systems in package (SiPs) and printed circuit boards (PCBs). The ERI investments are the next steps in creating a more automated electronics design capability that will benefit the aerospace/defense ecosystem and the electronic industry’s commercial needs. To fulfill the program charter over the four-year term of the contract, Cadence created the Machine learning-driven Automatic Generation of Electronic Systems Through Intelligent Collaboration (MAGESTIC) research and development program.” (Cadence Newsroom, 2018).

MAGESTIC aims to create an end-to-end solution for the evaluation and generation of systems, packages and boards. This flow would begin with specifying design rules, constraints and components and end with a fully routed and electrically feasible board, all completely automated. At the time of writing, Cadence has designed effective placement and routing algorithms for use in this design flow, but many complex designs fail to complete the routing stage due to clearance issues and high trace density. The idea to separate placement, via, and routing strategies materialized as a step to modularize the design process to offload the work of the auto-router to subprocesses and use the new architecture to parallelize layer routing. Move37’s design of GO VIA evolved because of this idea.

As a researcher through MIT’s 6A program, I was placed under the guidance of one of Cadence’s MAGESTIC team leads, Taylor Hogan, to evaluate various AI/ML algorithms, particularly in the via strategy domain. He heads the project known internally as Move37 (fittingly named after the 37th move in AlphaGo’s second game against Lee Sedol), an R&D group that specializes in the intelligent synthesis of
packages and boards (as opposed to Analog or VLSI design). My focus within Move37 was to research applicable optimization methods for via strategy / layer assignment to maximize routability, develop a mapping between the real-world problem and the optimization, implement said mapping in Python, and report test results to the team at large. This module would then be integrated between the component placement and routing stages.

3. Literature Review

The abstract problem we were attempting to solve is the minimization of the number of line segment intersections in 3D, where each segment (two endpoints, each with x, y and z values) is restricted to lie on the plane $z = C$ where $C$ is an integer we can choose, and the x-y endpoint coordinates are fixed. Essentially, this is the mathematical representation of logical (not physical) connections on a PCB, where line segments represent the unrouted device-to-device connections and the z-plane the line lies on represents the effective layer. For the purposes of this paper, we will call these logical connections flylines.
to differentiate from routed connections. We seek to distribute these flylines and crossings uniformly to ease congestion. We explain how we came to select this representation as an acceptable proxy in Sections 4 and 5. This problem at its core is NP-Hard given that it requires super-polynomial time to enumerate the possible solutions and/or verify correctness using a deterministic algorithm. As a result, the approaches we evaluated were primarily non-deterministic, i.e. pseudorandom in some internal process. Our reasoning was that many pseudorandom and non-gradient based approaches have fast convergence times, and even though many of such algorithms find local rather than global minima, local minima are often ‘good enough’ for our purposes. To further motivate this decision, let’s analyze the dimensionality of the design search space for an average board.

If we focus strictly on layer-assignment, we can characterize the problem using the number of available signal layers $N$ and the number of flylines $K$. The total number of possible layer placement combinations would then be $N^K$. Putting this in perspective of a physical design, the open source BeagleBone PCB has on the order of 1100 flylines on a total of four signal layers. For this relatively simple example, there would then be $4^{1100} \approx 1.8 \times 10^{662}$ possible layer assignment combinations, which is significantly more than the number of atoms in the observable universe. Using a brute-force method to enumerate all combinations at a rate of 10 GHz (assuming we could calculate the number of intersections in a single cycle, which is a vast underestimate) would take about $5.85 \times 10^{644}$ years. For reference, the universe is $1.38 \times 10^{10}$ years old. This illustrates why a brute-force approach is not feasible. For such a large search space, non-deterministic or greedy algorithms make sense.

Although other papers have explored the layer assignment of traces in VLSI design (Chang & Du, 1988), we found little past work concerning the layer assignment of flylines on a PCB. Still, the VLSI approach can be applied to our PCB problem given we can make a few guiding assumptions, such as enforcing vias
to lie under the pins / pads of the components on the PCB. We found it necessary to research a wide array of approaches, including each of the following:

- Genetic Optimization
- Simulated Annealing
- Particle Swarm
- Nelder-Mead (Simplex)
- Combinatorial Optimization
- Graph Coloring
- Adapted Maximum k-Colorable Subgraph (MkCS)
- Integer Linear Programming
- Reinforcement Learning

We used small synthetically generated designs as a benchmark for the via problem, so that we could quickly evaluate potential approaches. Of the previously enumerated, we began to evaluate genetic optimization, simulated annealing, graph coloring and adapted MkCS using our own implementations in Python 3.6.

For both graph-based approaches, we used the python library NetworkX. Nodes would represent individual flylines, and an edge from one node to another would represent flylines that cross. A node’s color represents the layer the associated flyline is assigned to. The goal for a successful algorithm would be to minimize the number of directly connected nodes that have the same color (violations, which would mean the number of true intersections in 3D), as well as to uniformly distribute the number of violations and flylines over the layers. After initial prototyping with the built-in NetworkX greedy coloring approach, we determined that this would not be a suitable method because greedy coloring does not allow a user to specify a constraint on the maximum number of colors to use. This would effectively mean we could
not specify a maximum number of layers the PCB can use to route flylines. Adapting the graph coloring problem to require a constrained number of colors and coloring to minimize violations is structurally similar to the maximum k-colorable subgraph problem (MkCS).

Our greedy implementation of a MkCS algorithm works as follows: sort the nodes of the graph by degree, highest degree first. Then, loop through the nodes, assigning a color that would give zero violations if it exists, or none if there is no zero-violation coloring. Accumulate the unassigned nodes, then repeat the color assignment, this time allowing one violation. Repeat this, incrementing the number of allowable violations each loop, until all nodes have been assigned a color. This achieved favorable results for the total number of crossings but tended to poorly distribute the flylines and crossings due to its greedy nature to assign to the first color that fits the violation criteria. To mitigate this issue, we augmented the MkCS implementation to pick the color that had been used the least thus far and that met the violation criteria. This traded off the total number of crossings for a better distribution of flylines.

Simulated annealing and genetic optimization were both set up in similar ways – we initialized a state vector (simulated annealing) / initial population of individuals (GenOpt) with randomized layer information and defined a function to evaluate the resulting state / individuals. For GenOpt, we defined

![Algorithm Performance](image_url)

*Figure 2: Results of four selected algorithms, genetic optimization, MkCS, augmented MkCS, and simulated annealing. Results were taken over three designs that varied in complexity; their sums are plotted.*
mutation, crossover, fitness and selection for our problem as described in Section 5. For simulated annealing, we defined movement similarly to GenOpt’s mutation function, and energy to be equal to the evaluation function. We ran these for ten thousand steps / generations each and recorded the results for these plus both MkCS algorithms in the appendix, which are also plotted in Figure 2.

We found that the two MkCS algorithms did particularly well when evaluating the solution using the cost function provided to GenOpt, which was a weighted combination of the max(crossings per layer), max(flylines per layer), and total crossings. The main issue with using this method was its extensibility; adding an additional metric forced a manual change in the implementation, whereas using a multi-objective optimization strategy would allow for quicker deployment in the event of change. In addition, introducing multiple objectives hampers a greedy algorithm’s ability to optimize in a single direction. We expect that using the solutions of these graph algorithms as a starting point for GenOpt and simulated annealing would allow us to find an even better solution, as GenOpt and simulated annealing converge slowly from random initialization but are eventually able to find solutions that rival the MkCS algorithms’ solutions. In contrast, the graph approaches were sensitive to initial conditions, deterministic, and don’t improve over time.

Genetic optimization required selecting many hyperparameters (probabilities, population size, tournament size, etc.), but our experimentation showed that the default settings gave performant results for a wide set of inputs. It also allowed for natural use of parallelism due to the independent nature of the individuals in the population, which is a benefit for tests with large populations and expensive fitness calculations. We knew that having an adjustable objective function was desirable for trying new metrics, and since routability is our true measure of success, we plan to later replace the fitness function with a neural network trained using layer assignment as input and routability as the loss. We also found an abundance of documentation and research done using genetic algorithms for optimization, hyperparameter tuning and neural net architecture search, which would be useful during our initial
implementation phase. We then selected genetic optimization as our strategy, and we began to research libraries to reduce prototyping time. We found a genetic optimization package that would allow us to create an environment for the via problem, called Distributed Evolutionary Algorithms in Python, also known as DEAP.

4. Objectives

The main objectives for GO VIA are to provide a new level of abstraction to exploit, to reduce the local density of traces on a design, to allow parallelizable auto-routing functionality, and ultimately provide for 100% routability of a design (meaning every flyline on a design has been fully physically routed). This method circumvents the auto-router’s relatively weak ability to place vias by precomputing via layers and locations before the routing stage. The main issue with auto-routing in a typical setting is that traces are laid out in a greedy fashion, often unnecessarily congesting areas of the board and compounding routing issues that could have been solved with a bit of preparation and organization. The router tends to place vias haphazardly to escape areas of the board that could have been planned around. In addition, traditional auto-routing suffers from its inability to capitalize on local symmetry. This is especially evident when attempting to fan out routes from a BGA. It can be seen from the image above that a valid solution
is to route the outer pins out first (shown with outer yellow traces), but a greedy approach would be very unlikely to find this solution. It would instead tend to fill channels to their max capacity and ignore that other traces will require such space later. By instead placing vias first (i.e. assigning a flyline to a layer), we can restrict the search space of the routing algorithm by eliminating the router’s need to place vias on-the-fly, and thus requiring a flyline to be fully routed on a single layer. This in turn means each signal layer can be treated independently, and thus routing can be parallelized, potentially reducing routing time by an order of magnitude.

The question now arises of how to effectively predict ahead of time which layers each trace primarily belongs. A simplification we arrived upon was to use the flyline as an approximation for future routing. The straight-line path between two pins is generally a suitable heuristic for the final path a trace will be routed in the future, especially since the routing search algorithms commonly use the Euclidean distance to the goal as a live heuristic. We use the number of crossings of these flylines / line segments on each layer, as well as the total crossing distribution across layers, as a proxy metric / fitness for the congestion of a design, and consequently the ability to route said design. By using flyline crossings as a substitute for trace density, we lower the dimensionality of the search space while simultaneously decreasing the total time needed to complete a design. By using this generalized GenOpt framework, we also gain the added benefit of being able to later swap the fitness function for a different evaluation strategy, such as a convolutional neural net trained directly on routability.

The final objective is for the GO VIA algorithm to be a shim between the placement and routing operations, thus making the end-to-end automatic placement to route process feasible. Ultimately the success of the routing stage is dependent on GO VIA’s solution, meaning if GO VIA provides an effective enough solution, then the automatic router should be able to fully complete a design. On the other hand, since GO VIA only roughly approximates true congestion, there is no guarantee that the translation from GO VIA to the auto-router will be a successful one, i.e. there may still be unroutable flylines. The
algorithm’s process, the assumptions made, and their associated shortcomings are enumerated in the next section.

5. Problem Formulation and Algorithm

As explained in Section 3, the abstract problem involves minimizing the number of flyline intersections in 3D by optimally assigning each flyline to a layer. When mapping this problem to genetic optimization, we must choose a representation that works within the language of chromosomes, mutation, crossover, fitness and selection such that the evolution process is well defined. In this case, we have chosen to define a chromosome such that the element at index $i$ represents the layer where flyline $i$ resides. We have provided an example in Figures 4 and 5 to show how a PCB would be represented within this framework.

![PCB Diagram](image)

Figure 4: Example PCB prior to flyline layer assignment. Black lines are logical, i.e. they are not yet routed.

We can also precompute a table of intersections which will allow for easy iteration during the fitness calculation process.

The PCB graphic in Figure 5 shows the first PCB after an initial layer assignment has been completed. This is a non-optimal layer assignment due to the intersection between flylines L3 and L6 on the red layer. A true intersection means that two flylines cross each other from a top-down view and are on the same
layer. In this framework, this means even though flylines L2 and L4 cross from a birds-eye view, they do not contribute to a true intersection because they are on different layers (red and blue respectively). Moving either L3 or L6 to the yellow layer would reduce the total number of crossings for this board to zero (a globally optimal solution in this case). This is what we expect GOVia to do automatically using its genetic operators. Note how the layer information is encoded in the chromosome as a color, which is also shown on the board. This is just for abstraction purposes -- in GOVia all layers are defined by an integer which is mapped to the correct layer name outside the GenOpt process. Now we will describe how the genetic operators (mutation, crossover, fitness, selection) are defined on our chromosome.

Since genetic optimization is a general framework, we must implement each genetic operator with a knowledge of how such operators will affect the higher-level abstraction. For mutation, this representation is simple. We use two hyperparameters to define the mutation: the probability of mutating the chromosome (mutpb), and the independent probability of mutating a single gene, or element, within
the chromosome \((\text{indpb})\). In addition, we use another hyperparameter to specify the number of individuals / chromosomes within our population \((\text{pop\_size})\). During the mutation process, we iterate through the population of individuals using a random number generator compared to \(\text{mutpb}\) to decide if we mutate the current chromosome. If we do, we then iterate through each element of the chromosome, using another random number generator compared to \(\text{indpb}\) to decide if we mutate the current gene. If we do, we choose a new layer for this flyline, which is determined by the set of possible signal layers.

For crossover, we define a new hyperparameter \(\text{cxpb}\) which determines if the current chromosome is crossed with another randomly chosen individual (these two individuals are called parents). Crossover in this case was defined as a single-point, meaning the splice point was at a single element. Since all chromosomes are represented as simple arrays in code, this is as simple as taking slices of those two parent individuals (from a randomly selected index until the last element) and swapping these subarrays. This results in two new individuals (offspring) with genetic information from each of the parents which are then placed back into the population.

The fitness assignment stage determines a score for every individual in the population, given based on a user-defined metric. We define the fitness function to be the sum of intersecting flylines on each layer of the board. This means we must calculate all intersections on the board, and then check for every unique intersecting pair (i.e. \(L1-L2\) and \(L2-L1\) are the same pair) if both flylines are on the same layer. It makes sense in our case to precompute all intersections first, which we keep track of in the previously mentioned crossing map. To find the number of true intersections, we can then iterate through the crossing map (which is essentially a table of intersecting pairs), checking if both flylines are on the same layer using the associated information in the chromosome. If they are, we increment the fitness. We also consider the distribution of the crossings on our board (layer with the maximum number of crossings). The idea is that we would rather distribute the total number of crossings uniformly to increase the likelihood that the
auto-router will be successful. The two metrics are then averaged using weights to create the final fitness score. In our case, a higher fitness score represents a worse design.

Selection is the mechanism that determines which individuals live on to the next generation. We used a tournament selection method which randomly selects the best (lowest fitness) individual from \text{tourn\_size} individuals. This process is done \text{pop\_size} times to create the population for the next generation.

Once the GenOpt process has finished, either due to timeout or reaching a specified fitness, control is turned over to Cadence’s system planner. Integration between the Python GenOpt system and the real PCB system planner is orchestrated through an import/export IO system. This process first assumes we have both component and connection information supplied before the process of layer assignment begins. After component placement has been completed, Cadence’s OrbitIO design planner imports a design file containing the physical connectivity information, i.e. the net list, each component’s footprint, port and net information, location, metadata, etc. Orbit then automatically places the components using a heuristic to minimize copper usage (the total length of traces). This placement information is then exported into JSON format so that GO VIA can interpret the location of each component’s pin. From there, control is turned over to Python where the following steps occur:
GO VIA Pseudocode:

1. Read pin and net information from OrbitIO’s exported JSON file.
2. Run Prim’s algorithm to find minimum spanning tree for all pins on same net. Repeat for all nets.
3. Collect pairs of connected pins, represent them as pin-pair objects without layer info
4. Pre-compute intersection of all line segments as if all flylines were on the same layer, call this the crossing map (key: single flyline, value: all flylines that intersect the keyed flyline)
5. Randomize initial population of individuals (assigns random layer info to each flyline)
6. Evaluate the fitness of every individual based on a combined weighted metric:
   a. The number of crossings of flylines (want to minimize)
   b. The distribution of crossings (want to uniformly spread crossings)
7. Select the best individuals using a tournament-based approach
8. Crossover then mutate the individuals using the given hyperparameters
9. Repeat steps 6-8 until desired fitness is achieved, or until time expires
10. Pick the best individual / design in the population
11. Export pin & layer data of this design to a CSV, to be imported by OrbitIO
Once Orbit receives the exported information from GO VIA, Orbit instantiates vias in the system design according to the layer each flyline was assigned. For example, let’s say flyline_1 represents pins P1 and P2. P1 and P2 belong to different components on the TOP layer. GO VIA calculates that flyline_1 should
belong to the BOTTOM layer, thus Orbit creates a via under P1 from TOP to BOTTOM, and another via under P2 from BOTTOM to TOP. This process only places vias directly under pins, a limitation chosen for reasons explained in the next section.

![30 flylines on 6 layers](image)

**Figure 7**: Example of GOVia layer assignment for a randomly generated set of flylines (actual simulation)

### a. Assumptions

GO VIA has been formulated under three main assumptions to make the design and implementation easier, the first being a via-under-pin implementation. Because via placement is typically defined by both a position and a set of layers, the time-complexity of the full problem is on the order of component placement, if not worse. Thus, we made a vast simplification to only place vias directly under pins, therefore reducing the degrees of freedom to a single discrete domain (i.e. the layer to place a trace on).
This allows the genetic fitness evaluation function to run quickly for an entire generation (milliseconds) as opposed to a full route (minutes / hours).

The second assumption is an extension of the first. Since vias are placed directly on top of pins, traces can be viewed logically as a straight line from pin to pin. These logical traces, known as flylines or rats, are the minimum spanning tree representation that connect all pins on the same net. These flylines also have zero width, meaning the thickness of a wire is not considered. By viewing the traces as line segments in 2D space, we further reduce time-complexity by trading off for model simplicity.

Finally, differential pairs are assumed to be single traces in this model, which is a close approximation since most diff pairs travel relatively close to each other and terminate at similar locations.

b. Drawbacks

Since flylines are only a proxy for the density of a design, there is not a guarantee that having a perfect or near-perfect via solution (i.e. one with no crossings or few crossings) implies the ability to fully route a design. Also, having a poor via solution also does not necessarily imply a design cannot be routed. The first case may occur if traces are wide, or if components prevent the placements of traces. The second could occur if there were components arranged in a star pattern. Traces can realistically navigate around the outside of the components, but our simplified model would show many crossings.

Pin escape and BGA fanout are common practices used by design engineers that are not implemented in GO VIA, but OrbitIO has support for these scenarios. When fully integrated with Orbit, BGAs could be pre-processed to produce escaped pins that could be passed to GO VIA.

Differential pair routing can be difficult near endpoints since both diff pair traces are constrained to be the same length. Often when finishing routing, one of the two traces in the diff pair needs to be elongated to match the longer. This results in congestion near the pin, reducing the ability of a complete route. GO VIA has no way to predict this type of congestion due to differential pairs.
Given these drawbacks, it can still be shown from our findings that the simplifying assumptions are a suitable trade-off for fast algorithmic performance and ultimately for fully routing a design.
6. **Project Timeline**

1. **Problem Statement** (June – July 2018)
   a. Verify problem, what is the end goal?
   b. Model reduction, problem simplification
   c. Draft high-level plan of attack

2. **Research phase** (July 2018)
   a. Evaluation of different approaches
   b. Literature Review

3. **Design Phase** (July – August 2018)
   a. VCS / Environment Setup
   b. Define API
   c. Evaluate external libraries

4. **Implementation Phase** (August – December 2018)
   a. Write code
   b. Test cases
   c. Debug
   d. Final integration

5. **Algorithm Meta-analysis** (December 2018 – January 2019)
   a. Profiling and Optimization
   b. Parameter testing / Sensitivity analysis

6. **Analysis of Results and Conclusion**
7. Findings

Thus far, we have made great improvements over current auto-routing capabilities by introducing a rudimentary via strategy component to the design process. Previously, even simple designs could not be routed, usually due to congestion that occurs in the last 5 to 10 percent of the routing procedure. Now, using GO VIA has allowed for multiple successful automated design runs on simple PCBs as well as more complex interposer designs. Even with 5 minutes of evolution time using GO VIA has shown major improvements over typical automatic routing scheme. In the best case, we’ve shown that a package that would normally take 120 hours from beginning to finish was successfully reduced to roughly 30 minutes using the GOVia strategy.

![Image: Intel Interposer manually routed, vs automatically routed using auto placement, auto via, and auto routing](image)

Figure 8: Intel Interposer manually routed, vs automatically routed using auto placement, auto via, and auto routing

While testing the GOVia framework, we also completed a hyperparameter search on the crossover and mutation probabilities to further increase performance. The results of that search, completed on a simple internal design, are on the next page.
Mutation Probability Sensitivity Analysis

- Population size: 50
- Number of Generations: 300
- Crossover Probability: 0.3
- Mutation Probability: Swept from 0 to 1.0 in steps of 0.1, 100 samples per value

Figure 9: We tested how the chromosome mutation probability affected the convergence of GOVia, optimal range - 0.5 to 0.7

Crossover Probability Sensitivity Analysis

- Population size: 50
- Number of Generations: 300
- Crossover Probability: Swept from 0 to 1.0 in steps of 0.1, 100 samples per value
- Mutation Probability: 0.2

Figure 10: We tested how the chromosome crossover probability affected the convergence of GOVia
As we can see from the plots on the previous page, increasing the mutation probability allowed the GenOpt process to find better solutions faster, up until a value of about 0.7, where mutation occurs at such a high rate that elite individuals likely don’t stay in the gene pool long enough to have their effects felt. We also found that varying the crossover probability does little to increase or decrease the number of crossings given a fixed number of generations. This could be because the design has many flylines placed such that they don’t span the whole board, meaning an individual flyline only has influence on the flylines near it. If the chromosomes are assigned such that elements close to each other also represent flylines that are spatially nearby, then crossover between two individuals may not greatly influence the fitness of the resulting individuals. If this was true, then increasing crossover would not change the speed of convergence / the number of crossings for a fixed number of generations, which is what we observed above.

8. Future Work / Suggested Improvements

Much improvement to GO VIA’s initial design has already been made by pre-computing flyline intersections. This reduced the running time for large designs from hours to minutes. There are currently
plans to train a deep neural net on the same design data that we passed through GO VIA. Once implemented, GO VIA’s internal fitness function would be replaced with this deep net trained on true routability, greatly decreasing the time needed to do evaluation, and eliminating the pre-computing stage altogether.

In the grand scheme, several methods from machine learning will be evaluated to close the loop of intelligent design. For instance, when routing, much information is currently lost after failed designs that could be used to inform successful future designs. Data such as local trace density, route completion percentage, via strategy and much more could be fed back to the initial design stage to push the algorithm towards an optimal solution.

9. Conclusion

Previous attempts at solving auto-routing’s failure to produce complete designs have not made significant forward progress. Instead of focusing on improving the routing algorithm, we decided to abstract the idea to three major stages: Component placement, layer assignment, and trace routing. GO VIA emerged to fill the layer placement gap between typical placement and routing. Through testing and integration into our OrbitIO system planner, we found great success with our benchmark designs, allowing for an up to two order of magnitude speed increase over manual PCB place and route. While the application’s full potential has yet to be seen, our preliminary results show great promise. We have reason to believe the development of the GO VIA framework will be a major stepping stone in the complete automation of silicon, package and board design.
10. References


11. Appendix

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Board</th>
<th>Xing L1</th>
<th>Xing L2</th>
<th>Xing L3</th>
<th>Xing L4</th>
<th>Max Xing</th>
<th>Mean Xing</th>
<th>Stdev Xing</th>
<th>Total Xing</th>
<th>Cxn L1</th>
<th>Cxn L2</th>
<th>Cxn L3</th>
<th>Cxn L4</th>
<th>Max Cxn</th>
<th>Stdev Cxn</th>
<th>Total Cxn</th>
<th>GenOpt Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenOpt</td>
<td>Test 1</td>
<td>106</td>
<td>104</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>104.75</td>
<td>0.9574271</td>
<td>419</td>
<td>98</td>
<td>100</td>
<td>97</td>
<td>99</td>
<td>100</td>
<td>1.290994</td>
<td>394</td>
<td>625</td>
</tr>
<tr>
<td>SimAnn</td>
<td>Test 1</td>
<td>112</td>
<td>111</td>
<td>111</td>
<td>81</td>
<td>112</td>
<td>103.75</td>
<td>15.173991</td>
<td>415</td>
<td>94</td>
<td>99</td>
<td>100</td>
<td>101</td>
<td>101</td>
<td>3.109126</td>
<td>394</td>
<td>628</td>
</tr>
<tr>
<td>MaxSub</td>
<td>Test 1</td>
<td>52</td>
<td>34</td>
<td>54</td>
<td>92</td>
<td>92</td>
<td>58</td>
<td>24.385788</td>
<td>232</td>
<td>154</td>
<td>104</td>
<td>77</td>
<td>59</td>
<td>154</td>
<td>41.36424</td>
<td>394</td>
<td>478</td>
</tr>
<tr>
<td>MaxSubAug</td>
<td>Test 1</td>
<td>68</td>
<td>53</td>
<td>108</td>
<td>64</td>
<td>108</td>
<td>73.25</td>
<td>24.01909</td>
<td>293</td>
<td>98</td>
<td>94</td>
<td>104</td>
<td>98</td>
<td>104</td>
<td>4.123106</td>
<td>394</td>
<td>505</td>
</tr>
<tr>
<td>GenOpt</td>
<td>Test 2</td>
<td>246</td>
<td>244</td>
<td>250</td>
<td>246</td>
<td>246.5</td>
<td>2.5166115</td>
<td>986</td>
<td>290</td>
<td>311</td>
<td>313</td>
<td>303</td>
<td>313</td>
<td>313</td>
<td>10.43631</td>
<td>1217</td>
<td>1549</td>
</tr>
<tr>
<td>SimAnn</td>
<td>Test 2</td>
<td>276</td>
<td>243</td>
<td>241</td>
<td>202</td>
<td>276</td>
<td>240.5</td>
<td>10.270998</td>
<td>962</td>
<td>294</td>
<td>308</td>
<td>310</td>
<td>305</td>
<td>310</td>
<td>7.131592</td>
<td>1217</td>
<td>1548</td>
</tr>
<tr>
<td>MaxSub</td>
<td>Test 2</td>
<td>117</td>
<td>75</td>
<td>93</td>
<td>109</td>
<td>117</td>
<td>98.5</td>
<td>18.574176</td>
<td>394</td>
<td>615</td>
<td>308</td>
<td>180</td>
<td>114</td>
<td>615</td>
<td>222.2707</td>
<td>1217</td>
<td>1126</td>
</tr>
<tr>
<td>MaxSubAug</td>
<td>Test 2</td>
<td>128</td>
<td>142</td>
<td>108</td>
<td>102</td>
<td>142</td>
<td>120</td>
<td>18.402898</td>
<td>480</td>
<td>306</td>
<td>307</td>
<td>302</td>
<td>302</td>
<td>307</td>
<td>2.629956</td>
<td>1217</td>
<td>929</td>
</tr>
<tr>
<td>GenOpt</td>
<td>Test 3</td>
<td>60</td>
<td>60</td>
<td>61</td>
<td>61</td>
<td>60.5</td>
<td>0.573503</td>
<td>242</td>
<td>48</td>
<td>49</td>
<td>49</td>
<td>49</td>
<td>49</td>
<td>49</td>
<td>0.57735</td>
<td>194</td>
<td>352</td>
</tr>
<tr>
<td>SimAnn</td>
<td>Test 3</td>
<td>47</td>
<td>72</td>
<td>70</td>
<td>77</td>
<td>77</td>
<td>66.5</td>
<td>13.329166</td>
<td>266</td>
<td>52</td>
<td>49</td>
<td>47</td>
<td>46</td>
<td>46</td>
<td>2.645751</td>
<td>194</td>
<td>395</td>
</tr>
<tr>
<td>MaxSub</td>
<td>Test 3</td>
<td>53</td>
<td>39</td>
<td>15</td>
<td>53</td>
<td>53</td>
<td>34.5</td>
<td>15.864005</td>
<td>138</td>
<td>71</td>
<td>44</td>
<td>45</td>
<td>32</td>
<td>73</td>
<td>17.36855</td>
<td>194</td>
<td>264</td>
</tr>
<tr>
<td>MaxSubAug</td>
<td>Test 3</td>
<td>81</td>
<td>87</td>
<td>44</td>
<td>31</td>
<td>87</td>
<td>60.75</td>
<td>27.475747</td>
<td>243</td>
<td>53</td>
<td>59</td>
<td>42</td>
<td>40</td>
<td>59</td>
<td>9.036961</td>
<td>194</td>
<td>389</td>
</tr>
</tbody>
</table>

Figure 12: Data compiled from literature review algorithm survey