Using data-mining techniques to improve combinatorial optimization algorithms

Peter Jamieson¹, Farnaz Gharibian², Lesley Shannon³ and Steve Wilton⁴

Abstract
In this work, we show how data-mining can be used to cluster algorithmic generated data and use that data to improve algorithms that solve combinatorial optimization problems for a real-world application—the field-programmable gate array placement problem. Our methodology is a means for other algorithm engineers to improve their own algorithms for specific real-world problems that are hard to improve. In our case, the placement algorithms are difficult to improve, and to find better heuristics we analyze the results of placement solutions to find clustered information which can then be used to improve the algorithms. Specifically, we show a technique for gathering cluster information about placement, we create a new simulated annealing algorithm and a new genetic algorithm that can deal with a mixed granularity of placement objects on a virtual field-programmable gate array, and we show that these algorithms either execute faster or improve the overall quality of solution compared to their basic algorithm without this clustering data and improved heuristics. For our improved simulated annealing placer we improve the algorithms run-time by 17% across a range of benchmarks, and our genetic algorithm improves placement metrics—critical path by 10% and channel-width by 4%.

Keywords
Clustering, heuristic, genetic algorithm, simulated annealing, field-programmable gate array placement

Introduction
Combinatorial optimization problems, the most commonly known is the Traveling Salesman Problem, are used to model real-world problems and meta-heuristic algorithms are used to find solutions to these problems in a reasonable amount of time.¹ For example, in this research, we examine the 2D placement problem and its direct application to field-programmable gate array (FPGA) design mapping as a real-world combinatorial optimization problem. One of the key challenges with these problems, their engineering application, and the algorithms we use to heuristically find solutions is that these algorithms are hard to improve. Typically, algorithmic improvements are achieved by finding domain-specific knowledge related to the problem that a heuristic can exploit and improve the base algorithms. In this work, we take a different approach and show how techniques in data-mining can be used for improving these algorithms.

Specifically, we show how both a genetic algorithm (GA) and the simulated annealing (SA) algorithm can use clustering data to improve the run-time or quality of solution of these algorithms for the FPGA placement problem. The main idea is that for many of these types of problems there are certain highly related low-level objects that exist in the problem instance that can be considered as a group (or cluster) when solving the global problem, and these objects can be expanded and dealt with at the early algorithmic steps. To find these highly related low-level objects we use clustering techniques, and the data we mine is generated from the algorithms in a post-algorithm data generation process.

The clustered objects found from data-mining, however, cause a problem for traditional algorithms such as GA and

¹Miami University, USA
²Xilinx Inc., USA
³Simon Fraser, Canada
⁴University of British Columbia, Canada

Corresponding author:
Peter Jamieson, Miami University, Department of Electrical and Computer Engineering, Oxford, OH 45056, USA.
Email: jamiespa@miamioh.edu

Creative Commons Non Commercial CC BY-NC. This article is distributed under the terms of the Creative Commons Attribution-NonCommercial 4.0 License (https://creativecommons.org/licenses/by-nc/4.0) which permits non-commercial use, reproduction and distribution of the work without further permission provided the original work is attributed as specified on the SAGE and Open Access page (https://us.sagepub.com/en-us/nam/open-access-at-sage).
SA since these algorithms for the 2D placement problem operate on a consistent granularity size of placement object. Therefore, we modify both GA and SA algorithms to deal with a mixed distribution of objects, creating new algorithms in this space.

The main contributions of this work are:

- A methodology to improve combinatorial algorithms for real engineering problems.
- A definition of the importance of granularity to certain algorithms.
- A description of Singularity SA and Supergene GA for mixed granularity FPGA placement.

The remainder of this paper is organized as follows: We start by providing a background on the FPGA placement problem and data-mining techniques. Next, we describe our methodology for generating our data and clustering it, including both a post-data and pre-data method to create clusters. Next, we describe our two modified placement algorithms that use our clustered data and provide results from FPGA placement. Finally, we discuss and conclude this work providing some ideas for future work and describing caveats with this work.

Background

In this section, we briefly review a simplified version of what an FPGA is and how the FPGA placement problem relates to mapping designs to these chips. We, also, briefly describe what data-mining is and how clustering algorithms are one of the main tools in data-mining and are used in this work.

FPGA placement

The FPGA is a programmable hardware substrate that can implement any digital design that fits on the chip. Typically, a design is created in a Hardware Description Language (HDL), either Verilog or VHDL, and is then converted to programmable bits with a synthesizing computer-aided design flow, which includes a step called placement.

Figure 1 shows the basic FPGA architecture where logic blocks are connected with programmable routing such that a design is first mapped into logic blocks, logic blocks are placed onto the FPGA, and routing paths are selected to connect the logic blocks to each other and I/O pins. The placement problem, specifically, attempts to place logic blocks in a graph format called a netlist onto the FPGA to minimize the connections between them in a metric called wirelength and to minimize the worst-case delay path in a metric called critical path.

A variety of placement algorithms have been studied over the years. The software we use in this paper, VTR, uses SA as the placement algorithm. In addition to SA for placement, researchers have used genetic algorithms, force-directed placement, and partitioning based placement.

Placement improvement. An improvement to the placement algorithm can be realized either through improved run-time or improved quality of results. This work focuses on either improving quality while maintaining run-time or maintaining quality and improving algorithmic run-time. A number of placement papers focus on improving the quality of results. A seminal paper by Cong et al. showed how placement algorithms are 10%–18% worse in critical path measurements with respect to a known optimal value. This result showed the research community that there is still work to be done in placement, and examples such as Lin and Wawrzynek show new techniques in improving quality of these algorithms.

Based on the “no free lunch” theorem, the optimization algorithms will, on average, result in the same quality. However, a common technique to improve an optimization algorithm is to use domain-specific knowledge. In this paper, we explore another method of improving the optimization algorithm via data-mining techniques, which may lead to a better understanding of the domain-specific knowledge that is not obvious to a human given the complexity of the problem.

Techniques in run-time improvement can be approached either with a single-threaded approach or via parallel approaches. Parallel approaches, such as Ludwin and Betz’s approach, are orthogonal performance optimizations compared to single-threaded run-time improvements including the one described in this work, and we note our approach is orthogonal to these parallel improvements.

Placement terminology. FPGA placement, specifically, places a netlist that represents the digital design onto a
2D grid, which represents the FPGA. The nodes in the netlist are, unfortunately for this work, named clusters. Therefore, the data-mining approach we use on post-placement data generates what we call gems where a gem is a cluster of FPGA clusters that have a high affinity to each other. We will interchangeably describe FPGA clusters as blocks to clear up some of this confusing terminology.

Another concept that we need to understand with placement is that, generally, FPGA placement deals with fine-grain netlists where each of the elements in a netlist is a block. We define three types of netlist block granularity:

- Fine-grain blocks: contain only one FPGA cluster/block.
- Medium-grain blocks: multi-blocks containing between 2 and 20 FPGA clusters/blocks.
- Course-grain blocks: contain more than 20 FPGA clusters/blocks.

Note that “gems” are medium-grain blocks.

We use the terminology {fine-grain, multi-block} to mean a netlist that contains a mixture of fine-grain blocks and multi-blocks. It should be noted that the mixture of {fine-grain, multi-block} does not describe the distribution of these blocks within the design. For example, a benchmark might have 40% fine-grain blocks and 60% multi-blocks or 90% fine-grain blocks and 10% multi-blocks.

Placement algorithms can be classified based on the distribution of granularity block types that they take as inputs; for example:

- {Fine-grain} placer: placement method for individual blocks. Traditional placement algorithms such as SA are in this category.
- {Course-grain-block} placer: placement methods for placing of multi-blocks, such as placement methods for placing large pre-placed IPs (RapidWright) and floorplanning.
- {Fine-grained, Multi-block} placer: placement methods for a mixture of blocks and multi-blocks, such as mix-size placement presented in Roy et al. or the singularity placer presented in this work which places a mixture of blocks and multi-blocks.

This work, to our knowledge, presents the first {Fine-grained, Multi-block} placement algorithms.

**Data-mining techniques**

The goal of data-mining is to extract information from data that may not be obvious from just looking at the information. Formally, data-mining involves executing three key steps:

1. feature selection—determining what you are trying to find,
2. discovery—finding if the feature is in the data and how often,
3. interpretation—determining what the result means depending on how common the feature.

A major algorithmic tool used in data-mining is clustering, where generally, clustering involves the grouping of objects based on some similarity of metrics. Therefore, clustering is a tool used in step 2 of data-mining for the discovery of features. For example, imagine a map of a country. Cities (the objects under question) might be clustered together if they are geographically (a measurement metric of distance) closer to one another. The most basic clustering technique is k-means clustering, but many other clustering algorithms have been proposed to perform clustering. In this work, we look at how data-mining the algorithmic data generated by multiple runs of placement algorithms can be clustered to find low-level highly connected logic blocks in the original design. We will describe this methodology in the following section.

**Methodology for placement algorithmic data-mining**

In this section, we describe how we use data-mining clustering techniques to find low-level highly connected objects into medium-grain blocks to improve placement. We describe a post-placement methodology and discuss how these mined gems are important in understanding the nature of FPGA blocks as related to the placement problem.

**Post-placement clustering methodology**

The details of this methodology were first described in Gharibian et al., and that paper includes more details of our methodology for finding gems. This methodology was invented because we were seeking ways to find gems in FPGA designs, but our high-level methods such as using affinity clustering and HDL language structures to cluster were not effective in finding useful medium-grain blocks.

Our post-placement methodology uses the placement algorithms, themselves, with random seeds to generate a dataset that we then mine to find gems from. The following describes our steps:

1. Run placement algorithm \( N \) times to generate the dataset of placed circuits.
2. Using the results create a proximity graph consisting of each of the FPGA clusters and edges between these nodes if they are considered close in terms of distance.
3. Use a “Disconnected” Max-Clique algorithm to find gems.

In the first step, we generate the dataset by running our SA algorithm on a benchmark \( N \) times. As this is a post-
placement methodology this can take considerable time, but our main goal is to find gems, which we can later analyze to find faster up-front clustering methods.

Step 2 of our methodology looks to find blocks in the 2D FPGA placement that are consistently close to one another in a proximity graph. This graph consists of nodes that are each of the blocks in the original netlist. The distance between blocks is a Manhattan distance measurement in terms of a block \(i\) and block \(j\) such that:

\[
|X_{\text{block}(i)} - X_{\text{block}(j)}| + |Y_{\text{block}(i)} - Y_{\text{block}(j)}|
\]

(1)

Then we define a minimum threshold, called \textit{minimal\_distance}, that must be met in a percentage of placement solutions—called recurrence. Therefore, an edge will be included between nodes \(i\) and \(j\) if the \text{recurrence\_percent} is met or surpassed for all the benchmark placements such that the Manhattan distance is less than the \textit{minimal\_distance}. The proximity graph captures which blocks are closely related to the next step of our methodology.

In step 3, the proximity graph is analyzed with a disconnected max-clique algorithm (described by Gharibian \textit{et al.}) and creates gems. The max-clique algorithm, such as the one described by Bron and Kerboscht \textsuperscript{22} may include a block in more than one clique, and therefore, we have extended the max-clique algorithm and call it the disconnected max-clique algorithm such that each FPGA block is included in only one clique. The resulting output of this algorithm is a set of clustered FPGA blocks that are consistently placed close to one another, and therefore are gems.

The nature of gems

Applying data-mining to algorithmic results, as described above, provides us with a set of gems. Gems are clusters within the initial digital design that we could not find using upfront clustering techniques with theoretical relationships on the design netlists, \textsuperscript{23} but by finding gems, which are localized placement knowledge extracted from repeated runs of the placement algorithm, we can do two things. First, we can use these gems as medium-grain blocks mixed with the unclustered blocks (fine-grain blocks) of the original design to examine new algorithms for placing this distribution of blocks. We will examine this in the following section.

Additionally, by using this method of discovering gems, we can look into the nature of a gem and determine if there are ways of finding them without needing to perform the computationally expensive post-placement data-mining methodology. Indeed, we have used this information to create a pre-clustering technique for placement, \textsuperscript{24} but do not present these results here since our goal is to show how data-mining techniques can be used in combinatorial optimization algorithms to improve the run-time.

Also, note that the parameters \(N, \text{minimal\_distance}\), and \text{recurrence\_percent} can be changed to find different gem distributions. For our results, we look at two parameter sets in our two experiments. For GA experimentation, we use \(N = 20, \text{minimal\_distance} = 2,\) and \text{recurrence\_percent} \(= 100\%\), and for SA experiments we use \(N = 20, \text{minimal\_distance} = 6,\) and \text{recurrence\_percent} \(= 80\%\). Note, however, that our algorithms could handle different parameter settings.

Mixed granularity placement algorithms

Using our post-placement data-mining technique we have a collection of gems as inputs to the placement algorithms. Based on the assumption that gems are high-probability consistently placed close together blocks, then in the early stages of a placement algorithm, they can be kept together saving time and effort of the placement algorithms. This, however, is not trivially implementable in existing GA and SA placement algorithms as these two algorithms are designed to deal with a \{Fine-grain\} distribution based netlist. In this section, we provide details on how we modified the GA to improve the quality of results with an input of the circuit including data-mined gems. Additionally, we provide details on how we modified SA to improve algorithmic run-time with both the circuit and gems.

With the inclusion of gems, we have a \{Fine-grain, Medium-grain\} netlist and the algorithms need to be modified to deal with the mixture of two granularity types. We describe these two algorithms called Supergene Genetic Algorithms and Singularity Simulated and Annealing below.

Supergene GAs

The idea of supergenes \textsuperscript{25} in GAs is that the phenotype (traits of an individual) emerge based on the interaction between multiple encodings within the genome. Our major contribution in this algorithm is using existing combinatorial remapping techniques to give priority to supergenes and remap fine-grain blocks via these techniques to create a legal placement.

Supergenes have research similarities to messy GAs, the idea of gene expression, and canonical genes. Messy genetic algorithms (MGAs) \textsuperscript{26} were introduced in 1989 by Goldberg \textit{et al.} and introduce the idea of variation in both the information length and the rigidity of the genome. Proportional GAs (arguably a subset of MGAs and similar to Canonical GAs \textsuperscript{27}), as introduced by Wu and Garibay, \textsuperscript{28} presents the idea of the phenotype of an individual as an expression of the existence or non-existence of genes within the genome. Gene expression has been studied by researchers in the area of simulation of biological evolution. For example, Eggenberger \textsuperscript{29} looks at gene
expression for the evolution of 3D organisms. Our first presentation of the supergene algorithm was in Jamieson et al.\textsuperscript{3} Our supergene GA for FPGA placement consists of several base GA parameters and features that have been developed over a number of years and includes the following features:

- **Inbreeding Avoidance:** Our GA avoids crossbreeding similar individuals by maintaining a history of ancestors five generations back and not breeding with shared ancestors.\textsuperscript{30,31} For combinatorial optimization problems, this is maintained via recording ancestors and is derived from ideas in Tabu search\textsuperscript{32} and CHC.\textsuperscript{33}

- **PMX or CSR:** The crossbreeding operator is one of PMX\textsuperscript{34} or CSR\textsuperscript{35} and the ideas of these crossover operators are explained with graphs and algorithms in Collier et al.\textsuperscript{36}

- **New Generations:** Parents and children combined together in competition for new generations where 50% of the parents are maintained.\textsuperscript{35}

Beyond these characteristics, the population size for our experiments is 500 individuals per generation, 50% of the genes are randomly mutated, and 90% of the new generation of individuals are created from crossbreeding and mutation (meaning 10% of the population comes from the best of the previous generation). The selection of parents for crossbreeding is random within the 10% of the best individuals.

**FPGA genome including supergenes.** The FPGA genome for the placement of fine-grain blocks includes the location of each block as described as a location on the FPGA, and this allows for easy mapping to the FPGA fabric to evaluate the cost function for a placement solution.

**Algorithm 1 Genetic Algorithm**

```
1: function Genetic_Algorithm()
2:    pop = initialize_population(genome; pop size)
3:    exit condition() == FALSE do
4:        rank = evaluate_and_rank_population(pop)
5:    pop = create_new_population(mutation; crossover; selection(rank))
6:    end while
7: end function
```

Algorithm 1 shows the basic pseudocode of a GA. The idea of supergenes is just a change to the genome that represents an individual in the population (also known as a solution to the problem). To include the medium-grain blocks or gems in the genome, we add supergenes with the following properties:

- **Size:** The number of fine-grain genes contained in a supergene.
- **Internal Genes:** A collection of genes in the supergene and properties of these genes.
- **Dominance factor:** A function defining if the supergene will express itself over other genes.

In the case of the FPGA placement problem, the supergene has a size relative to the size of a gem instance and the genes in the supergene are the blocks contained in the gene with additional details of each genes relative x and y coordinates in relation to the x and y locations of a supergene. For a specific individual, if a supergene expresses itself the locations of the clusters in the supergene will take precedence over the information contained in the fine-grain genome. Any genomes in the fine-grain genome that would map to the locations now occupied by the supergene are remapped in the same method as in the PMX or CSR crossover operator.

Figure 2 shows an example of a genome with fine-grain blocks in the genome and two supergenes that have expressed themselves (note that the bottom left corner corresponds to the coordinates 0, 0 and the upper right is 4, 4). The placement points are colored white for fine-grain clusters, gray for super-clusters, and black if the location is not used. In particular, note how FPGA clusters 11 and eight are remapped to a different location because of the supergene precedence. To achieve this remapping, the PMX or CSR approach is used where a conflict is detected and the already placed supergene cluster’s coordinates are used for the remapping. For example, in Figure 2, cluster 16 is already at coordinates (1,2), which cluster 11 is currently assigned in the fine-grain genome. We remap 11 to coordinates (1,1) as this is the coordinates of 16 in the fine-grain genome. Multiple conflicts will iterate through this remapping procedure guaranteeing that each cluster will have a unique location and maintaining a valid individual.

The following is the experimental aspects of supergenes that can be set:

- **Gene expression:** This is the different methods for a supergene to express itself. We use a binary dominance factor.
- **Supergene mutation:** What and how to mutate the supergene including the dominance factor, the global relative information of the supergene, and the internal details of the supergene.
- **Supergene crossbreeding:** What and how to crossbreed supergenes in different individuals. We do not consider this in this work.
- **Expression conflict remapping:** How to deal with conflicts between expressed supergenes and the genome to remap the collisions. We use a CSR remapping technique in this work.
In addition to the supergene parameters, the GA has a population size of 500, keeps 10% of the best solutions to the next generation, cross breeds the new generation by using either PMX or CSR, and for each new individual allows a 10% mutation rate based on placement swaps.

We only mutate supergenes, and when a new individual is created from crossbreeding of two fine-grain parents, we randomly pick one of the parents to pass on their supergenes to the new individual.

Mutating the supergene encompasses a number of options as described above. In particular, we can mutate the global supergene location (called a global mutation), the relative gene locations in the supergene (called a local mutation), and the dominance factor. Our dominance factor is a binary decision that is set to either on or off. Experimentally, we have found that mutating the global and relative positions produces faster improvements in results, but the dominance factor is best set to on. Because the super-clusters are gems in the experiments and these gems are guaranteed to benefit the final placement when placed in proximity. There is no benefit to mutating this self-expression off unless there is a higher chance that the supergene will not benefit the final placement.

The result of adding the supergene is we now have a GA that can deal with the \{Fine-grain, Medium-grain\} netlist.

**Figure 2.** A genome that includes expressed supergenes and appropriate remapping.
The overarching idea is the PMX or CSR remapping technique is used to allow supergenes to be placed with priority and the remapping then makes a legal placement.

**Singularity simulated annealing**

Singularity placement uses SA placement such that inputted gems are considered single blocks in early stages of global swaps (warm temperature) and then are expanded in the second stage and optimized for at more local swaps (cooler temperature).

In general, SA works by swapping blocks randomly and keeping swaps that benefit the cost function depending on a cooling schedule. In the scheduling of SA, warmer temperatures allow swaps that may not improve the cost function to be accepted. As the temperature cools only swaps that improve the cost function are accepted. Also, in FPGA placement, there is a distance of swap operator that determines the maximum Manhattan distance between blocks that will be swapped, and the scheduler decreases this distance value as another aspect of scheduling. Therefore, global swaps and local swaps change depending on the schedule.

**Algorithm 2** Singularity Placement

```plaintext
1: function Singularity_Placement(Netlist, Multi-Blocks)
2: Create Singularity Blocks
3: Initial Random Placement of singularity blocks
4: Phase1: Run SA until rlim < rlim - const
5: Multi-block expansion
6: Phase2: Continue SA
7: end function
```

Algorithm 2 shows two phases of SA where in the first phase we place gems, but treat each gem as a single FPGA block. This is where the term, singularity, comes from and means a densely packed block of blocks to allow a singularity block to be swapped seamlessly with the fine-grain blocks and other singularities. Once the scheduling of the algorithm shifts from global swaps to local swaps, we shift to phase 2 by first expanding the singularities into normal fine-grain blocks and then we run SA at a low temperature to find good local swaps.

Details of how scheduling and expansion are described by Gharibian et al.2 and Gharibian.24 The main idea is that the singularity allows us to use the existing SA algorithm to mix the placement of (Fine-grain, Medium-grain) netlists by splitting up the global singularity step and local refinement step into the two described phases. The Singularity SA has a number of internal scheduling control parameters which are beyond the scope of this paper, and for the interested reader we suggest using24 and the included code appendix to understand the algorithmic parameters.

**Experimental results**

In this section, we present results from two experiments that use the two algorithms above for the {Fine-grain, Medium-grain} netlist distribution. In the first experiment, we show how the supergene GA using extracted gems improves the cost function results compared to a GA for placement. This shows how the data-mined gems can be used to improve the quality of results. In the second experiment, Singularity Simulated Annealing is used with extracted gems to show how the data-mined results can speed up this algorithm compared to SA.

**Supergenes genetic algorithm to improve quality of results**

For the experiments and results that we will present in the next section, there are a number of items we describe as related to the methodology for FPGA CAD experiments. We describe these details so that FPGA focused researchers can understand the environment under which the experiments were run, and GA focused researchers understand why the experimental methodology might differ from their expected methodology. In either case, the goal of the experiments is to demonstrate that supergenes can be used to implement an algorithm for medium and fine-grain FPGA placement, this implementation is well suited for GAs, and the supergene concept improves on the current state-of-the-art GAs for FPGA placement. We remind the reader that even though this approach improves the algorithm, the GA for FPGA placement is significantly worse than, for example, SA for FPGA placement. This disparity is, likely, due to the lack of a good crossover operator in this domain.36

For this experimental setup, the details that we describe are:

1. The software framework that the algorithm is implemented in and the benchmarks circuits used.
2. The architectural parameters describing the FPGA we are placing the benchmarks on.
3. The computation system and conditions under which the algorithms are executed.

**Table 1.** Details for the benchmarks including extracted gems (recurrence = 100%, num-run = 20, and minimal-distance=2).

| Benchmark | # of Blocks | # of IOs | # of Gems |
|-----------|-------------|---------|----------|
| cfc8      | 110         | 51      | 5        |
| dconvert  | 396         | 258     | 39       |
| desa      | 229         | 190     | 1        |
| dsystemC  | 393         | 162     | 22       |
| iir1      | 118         | 58      | 13       |
| pajf      | 88          | 103     | 1        |
| rsd1      | 171         | 20      | 5        |
| rsd2      | 370         | 32      | 8        |
| sv3       | 46          | 40      | 4        |
| synth_14  | 521         | 544     | 11       |
Experimental setup and benchmarks for quality of results experiment. In this work, we use VTR, an open-source academic tool that allows researchers to experiment with both FPGA architecture and CAD.4 This experiment is run using 10 benchmarks, where these benchmarks have been converted to a netlist of clusters using an academic CAD flow. Table 1 shows a summary of the details of these benchmarks. Column 1 lists the benchmark name. Columns 2 and 3 show the grid size of the FPGA, the number of blocks, and the number of I/Os. Column 4 shows the number of super-clusters in the benchmark where the maximum number of blocks in a super-cluster is 3.

Each benchmark is passed into VPR for the same FPGA architecture as described below. VPR uses one of the placement algorithms and then routes the design. The output of VPR is the size of the FPGA (the same for each benchmark regardless of placement algorithm), the speed of the circuit in terms of the time delay for the critical path, and the channel width needed to route the circuit increased by 20% (as a relaxation on the circuit modeling real-world device usage).

The FPGA architecture parameters that describe both the FPGA are shown in Table 2. For a more detailed explanation of these parameters please consult,37 but for the sake of space, we do not describe these parameters in detail here.

These experiments are run on a Linux OS running on Intel Xeon 2.4 GHz cores. For the placement portion of the algorithm, there is no contention in the system. For the routing portion of the circuit, which includes a binary search to find the minimum channel width, all four cores run in parallel to speed up this computationally heavy process. This has no impact on the results reported.

Finally, in combinatorial optimization experiments, the approaches are impacted by the initial random seed. To eliminate some of the noise in these experiments we use a number of runs to average out the results. In the experiments, we use 10 random seeds for each case and average these using the geometric mean, which is a normal procedure in this type of CAD experiment. Any summary results across a range of benchmarks include a geometric mean.

Quality of results. Figures 3 and 4 show the results of these benchmarks run on three separate designs. The base comparison point is a GA with only mutations. Both _csr_ and coarse_iicsr represent the GA with a CSR crossover and the CSR crossover with supergenes, respectively. The values generated by these two algorithms are divided by the base values to create a ratio that when less than one indicates that the GAs with crossover are performing better than the base algorithm.

In both, channel-width (W) and critical path results, shown in Figures 3 and 4, it is clear that adding crossover capabilities to the GA improves the result. More importantly, the supergenes further improve the results by approximately 10% for critical path and 4% for channel-width.
through 4 lists the number of Blocks, the number of IOs, and the number of gems for each benchmark.

The experimental details including FPGA parameters and methodology are mostly the same for the run-time result experiments. The only notable difference is that for run-time experiments we need to guarantee that other than the operating system processes, the algorithms under test have dedicated access to the processor, which is the case.

**Singularity simulated annealing to improve run-time results**

Figure 5 shows the normalized results of the benchmarks where for each benchmark we compare the singularity SA against VPR’s basic SA algorithm. In this figure, the x-axis lists the benchmarks and the bars show the normalized results of the singularity placer compared to VPR for placement run-time, critical path delay, minimum channel width and wirelength. When a value is less than one, it indicates that singularity placer has a better performance in that category compared to VPR. The last two groups (Average Real and Average Synthetic) show average placement run-time, critical path delay, minimum channel width and wirelength for real benchmarks and synthetic benchmarks.

The results show that on average our placement run-time has been decreased by more than 17% compared to VPR over these seventeen benchmarks while maintaining the quality of placement for the critical path delay on average.

Wirelength and channel width slightly increase (2% and 6.3%, respectively) in the case of the real benchmarks. We claim that the quality of results is roughly equal to SA with a large run-time benefit. We hypothesize that the increase in cost is part of the multi-objective algorithm maintaining the critical path delay for our proposed approach as the singularity placer will tend to result in slightly more compact designs than the traditional VPR SA leading to greater routing congestion.

**Discussion**

The above results provide compelling evidence that data-mining algorithmic results can be used to improve the results of applied algorithms when no domain-specific knowledge can be leveraged. However, the reader may notice some inconsistencies and concerns in these experiments, which we briefly discuss.

First, the above two experiments used different sets of benchmarks and gem creation parameters. The reason for this was the time when the experiments were performed in the software development lifecycle. The GA experiments were performed with an earlier version of VPR, and for the sake of this publication, we believe the idea is sufficiently proven with the old experiments without a need to re-implement and redo the experiments on a newer version.
of the software. Particularly, since GAs for placement are worse than SAs this additional work has little value.

Second, a common argument with the run-time results is that the algorithmic run-time needed to extract the gems is not included in the reported results. As described earlier, we have come up with an algorithm (using what we learned from post-placement data-mining) to create an algorithm that can discover this data pre-placement. One also might argue, the nature of the placement algorithm is that it is industrially used in a CAD flow that is similar to compilation in software development where the synthesis of hardware is done repeatedly while further development is also being performed. We argue, that this development time can be used to extract gems from portions of the design that are in a state of near-completion in what is called incremental hardware design.

Third, the question remains are these results significant. We believe our results are significant because we are applying these techniques to a well studied, real-world problem—FPGA placement. As earlier referenced, Cong et al. seminal work showed that FPGA placers have room for improvement, but over the last 10 years there have been few gains other than in the space of parallelizing these algorithms for run-time gains.

The most compelling argument against our approach is that the data-mined cluster results come from SA results, and therefore, in the case of the GA, which is not as good as SA for FPGA placement, the clustered data improves the quality of results as the clusters are passing useful information from a good algorithm to a weaker algorithm. Similarly, in the case of singularity SA the clusters improve run-time since the information is passed to the algorithm in a “hindsight” like method, and will, of course, improve run-time. While this is true, the bigger questions are can these “gems” be found without data-mining (yes, as we have already described) and can a heterogeneous mix of components be effectively used in the placement algorithm. We believe that we have provided evidence that answers these two more important questions and leave the techniques of data-mining algorithms as future work.

Conclusion

In this work, we have shown how we can improve either the algorithmic quality of results or run-time by clustering and using these clusters in modified placement algorithms (that handle a mix of fine and medium-grained objects). In particular, we have focused on the FPGA placement problem and created two new algorithms, GA with supergenes and singularity SA, and shown how these algorithms improve quality of results and run-time, respectively. What is so exciting about these results is it provides a method to improve optimization algorithms even when domain-specific knowledge cannot be leveraged, and this provides another means for engineers to improve optimization algorithms when improved efficiency is needed.

There are several potential future work paths that emerge from this research. First, what happens if several meta-heuristic algorithms are used to create the data set that is then mined. For example, if GA and SA are used to create the data set does this degrade the performance because GA introduces worse results into the set. Second, an exploration of the clustering parameters might provide us with better knowledge of how much can they be relaxed and still get good results. Finally, for each of the two proposed algorithms, there are several questions in terms of how can these algorithms be tuned and improved.

Declaration of conflicting interests

The authors declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The authors received no financial support for the research, authorship, and/or publication of this article.
References

1. Blum C and Roli A. Metaheuristics in combinatorial optimization: overview and conceptual comparison. ACM Comput Surv 2003; 35: 268–308.
2. Gharibian F, Shannon L and Jamieson P. Identifying and placing heterogeneously-sized cluster groupings based on FPGA placement data. In Field Programmable Logic and Applications (FPL), 2014 24th International Conference on. IEEE, pp. 1–6.
3. Jamieson P, Gharibian F and Shannon L. Supergenes in a genetic algorithm for heterogeneous FPGA placement. In 2013 IEEE Congress on Evolutionary Computation. IEEE, pp. 253–260.
4. Rose J, Luu J, Yu C et al. The vtr project: architecture and cad for FPGAs from verilog to routing. In Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays. pp. 77–86. http://dl.acm.org/citation.cfm?id=2145708.
5. Kirkpatrick S, Gelatt CD and Vecchi MP. Optimization by simulated annealing. Science 1983; 220: 671–680.
6. Jamieson P. Revisiting genetic algorithms for the FPGA placement problem. In GEM. pp. 16–22. http://www.users.muohio.edu/jamiespa/html/papers/gem’10.pdf.
7. Vorwerk K, Kennings A and Vannelli A. Engineering details of a stable force-directed placer. In Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design. IEEE Computer Society, pp. 573–580.
8. Maidee P, Ababei C and Bazargan K. Timing-driven partitioning-based placement for island style FPGAs. IEEE Trans Comput-Aided Des Integr Circuits Syst 2005; 24: 395–406.
9. Cong J, Romesis M and Xie M. Optimality and stability study of timing-driven placement algorithms. In Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design. IEEE Computer Society, p. 472.
10. Lin M and Wawrzynek J. Improving FPGA placement with dynamically adaptive stochastic tunneling. IEEE Trans Comput-Aided Des Integr Circuits Syst 2010; 29: 1858–1869.
11. Wölpert DH, Macready WG et al. No free lunch theorems for search. Technical report, Technical Report SFI-TR-95-02-010, Santa Fe Institute, 1995.
12. Sulyok C, Harte C and Bodó Z. On the impact of domain-specific knowledge in evolutionary music composition. In Proceedings of the Genetic and Evolutionary Computation Conference. ACM, pp. 188–197.
13. Ludwin A, Betz V and Padalia K. High-quality, deterministic parallel placement for FPGAs on commodity hardware. In Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays. ACM, pp. 14–23.
14. Luu J, Kuon I, Jamieson P et al. VPR 5.0: FPGA CAD and Architecture exploration Tools with Single-Driven Routing, Heterogeneity and Process Scaling. In ACM/SIGDA International Symposium on FPGA. http://doi.acm.org/10.1145/1508128.1508150.
15. Lavin C and Kaviani A. RapidWright: enabling Custom Crafted Implementations for FPGAs. In 26th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM) 2018.
16. Roy JA, Adya SN, Papa DA et al. Min-cut floorplanning. IEEE Trans Comput-Aided Des Integr Circuits Syst 2006; 25: 1313–1326.
17. Biswas G, Weinberg JB and Fisher DH. Iterate: a conceptual clustering algorithm for data mining. IEEE Trans Syst, Man, Cybernics, Part C (Applications and Reviews) 1998; 28: 219–230.
18. MacQueen JB. Some methods for classification and analysis of multivariate observations. In Cam LML and Neyman J (eds.) Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability, volume 1. University of California Press, pp. 281–297.
19. Berkhin P. A survey of clustering data mining techniques. In Grouping multidimensional data. Springer, 2006. pp. 25–71.
20. Frey BJ and Dueck D. Clustering by passing messages between data points. Science 2007; 315: 972–976.
21. Gharibian F, Shannon L, Jamieson P et al. Analyzing system-level information’s correlation to FPGA placement. ACM Trans Reconfigurable Technol Syst 2013; 6: 15:1–15:21.
22. Bron C and Kerbosch J. Finding all cliques of an undirected graph [h]. Commun ACM 1976; 19: 575–580.
23. Gharibian F, Shannon L and Jamieson P. Finding system-level information and analyzing its correlation to FPGA placement. In 20th International Conference on Field Programmable Logic and Applications. www.users.muohio.edu/jamiespa/html/papers/FPL2010.pdf.
24. Gharibian F. Multi-phase placement approach for field programmable gate arrays. PhD Thesis, Applied Sciences: School of Engineering Science, 2018.
25. Joron M, Papa R, Beltrán M et al. A conserved supergene locus controls colour pattern diversity in heliconius butterflies. PLoS Biol 2006; 4: e303.
26. Goldberg D, Korb B and Deb K. Messy genetic algorithms: motivation, analysis, and first results. Complex Syst 1989; 3: 493–530.
27. Holland J. Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence. Cambridge, MA, USA: MIT press, 1992.
28. Wu A and Garibay I. The proportional genetic algorithm: gene expression in a genetic algorithm. Genet Program Evol Mach 2002; 3: 157–192.
29. Eggenberger P. Evolving morphologies of simulated 3d organisms based on differential gene expression. In Proceedings of the Fourth European Conference on Artificial Life. pp. 205–213.
30. Wibowo A and Jamieson P. Using simple ancestry to deter inbreeding for persistent genetic algorithm search. In GEM. pp. 23–29. http://www.users.muohio.edu/jamiespa/html/papers/gem’12.pdf.
31. Collier R and Wineberg M. The problems with counting ancestors in a simple genetic algorithm. In Proceedings of the 9th European conference on Advances in artificial life. ECAL’07, pp. 855–864. http://dl.acm.org/citation.cfm?id=1771390.1771489.
32. Glover F. Future paths for integer programming and links to artificial intelligence. Comput Oper Res 1986; 13: 533–549.
33. Eshelman LJ. The CHC adaptive search algorithm: how to have safe search when engaging in nontraditional genetic recombination. In FOGA. pp. 265–283.
34. Goldberg DE. *Genetic Algorithms in Search, Optimization, and Machine Learning*. 1st ed. Boston, MA, USA: Addison-Wesley Professional, 1989. http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20.

35. Collier R, Fobel C, Grewal G et al. Depictions of genotypic space for evaluating the suitability of different recombination operators. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*. pp. 609–616. http://dl.acm.org/citation.cfm?id=2330250.

36. Collier R, Fobel C, Pattison R et al. Advancing genetic algorithm approaches to field programmable gate array placement with enhanced recombination operators. *Evol Intell* 2014; 7: 183–200.

37. Betz V, Rose J and Marquardt A. *Architecture and CAD for Deep-Submicron FPGAs*. Amsterdam, NL: Kluwer Academic Publishers, 1999.

38. Ludwin A, Betz V and Padalia K. High-quality, deterministic parallel placement for FPGAs on commodity hardware. In *FPGA ’08: Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*. pp. 14–23. http://doi.acm.org/10.1145/1344671.1344676.