Using a Genetic Algorithm to Optimize Configurations in a Data-driven Application

Urjoshi Sinha[1], Mikaela Cashman[1,2], and Myra B. Cohen[1]

[1] Iowa State University, Ames, IA 50011, USA
[2] Oak Ridge National Laboratory, Oak Ridge, TN 37830, USA
{urjoshi,mcohen}@iastate.edu,mcashman.isu@gmail.com

Abstract. Users of highly-configurable software systems often want to optimize a particular objective such as improving a functional outcome or increasing system performance. One approach is to use an evolutionary algorithm. However, many applications today are data-driven, meaning they depend on inputs or data which can be complex and varied. Hence, a search needs to be run (and re-run) for all inputs, making optimization a heavy-weight and potentially impractical process. In this paper, we explore this issue on a data-driven highly-configurable scientific application. We build an exhaustive database containing 3,000 configurations and 10,000 inputs, leading to almost 100 million records as our oracle, and then run a genetic algorithm individually on each of the 10,000 inputs. We ask if (1) a genetic algorithm can find configurations to improve functional objectives; (2) whether patterns of best configurations over all input data emerge; and (3) if we can we use sampling to approximate the results. We find that the original (default) configuration is best only 34% of the time, while clear patterns emerge of other best configurations. Out of 3,000 possible configurations, only 112 distinct configurations achieve the optimal result at least once across all 10,000 inputs, suggesting the potential for lighter weight optimization approaches. We show that sampling of the input data finds similar patterns at a lower cost.

Keywords: Genetic Algorithm · Data-driven · SSBSE

1 Introduction

Many scientific applications are heavily data-driven, meaning their function (or behavior) is dependent on the specific data used to run the application and the data is often complex and varied. At the same time, these systems are often highly-configurable; the end user can modify a myriad of configuration options that control how the system behaves. The options may induce simple changes such as controlling how output formatting is handled, or they can change underlying algorithms and the algorithm's parameters, returning entirely different results [2]. Some options may also change system performance, causing the application to run faster, or to utilize fewer resources such as bandwidth and energy. There has been a lot of focus on highly-configurable software with the goal of optimizing or finding options to improve quality [7, 8, 20].

To assist with configuration selection, some state of the art techniques use prediction [20], building models of behavior of the configuration space. An alternative approach is to use optimization, or search-based techniques [5, 17, 21]. Instead of modeling the configuration space to ask how a particular configuration will behave, search-based methodologies work to find an optimal (or near-optimal) configuration for a particular input data set.

However, as we demonstrate in this paper, and something that has been eluded to by Nair et al. [16], optimization in data-driven systems carries additional challenges. In these systems, different configuration options may be optimal for different data in the input data set, making optimization a heavy-weight process – it needs to be run each time new data is utilized and this may be impractical in practice. In our case study, optimization on a single input takes minutes. Although an individual run over a single configuration takes milliseconds, this is magnified if optimization needs to be run for each input in the dataset. Instead, we want to re-optimize as little as possible. Furthermore, as configuration spaces grow, the time for optimization will continue to be amplified. Research on *transfer learning* for system performance in highly-configurable software has suggested that some models provide information which can be re-used as workloads change [7, 8, 10]. However, workloads define the load on a system; they are not system inputs, and they impact performance, not functionality. As far as we know, there has been little focus on the problem of optimization for a data-driven application, when the optimization goal is to improve system functionality.

In this paper we explore the problem of functional optimization for data-driven applications. We ask how different inputs from a data set change the functional results of configuration optimization in a scientific application. In order to establish ground truth to ask our questions, we design an experimental testbed that contains exhaustive data of 3,000 configurations (all possible configurations from our model) and 10,000 different input sets. While this is a small configuration space, it provides us with a rich set of data – an exhaustive set of data of almost 100 million records.[3] We use this database to first evaluate the quality of our genetic algorithm. We then ask if there is a pattern of *best configurations*. We see a reduction from 3,000 to 112 configurations (or only 3.7%) appearing as the best configurations, therefore we ask if we can sample the input data and achieve similar results. Our study finds that samples as small as 10 inputs can produce similar results, suggesting a light-weight optimization approach is possible.

The contributions of this work are:

1. A study on using a genetic algorithm to optimize configurations to improve the functional behavior of a popular data-driven scientific software application;
2. An exhaustive database containing almost 100 million records for a set of 10,000 different inputs;

[3] Each run of the application can return multiple answers leading to many more records than 3,000 × 10,000.

3. Results on sampling input data, suggesting that the observed patterns hold over even small random samples.

In the next section we present some motivation. In Section 3 we present an overview of our experimental database framework. We follow that with a case study in Sections 4 and 5. We then present some related work (Section 6) and end in Section 7 with conclusions and future work.

2 Motivating Example

We present a small motivating example based on our case study application in this section. BLAST stands for the Basic Local Alignment Search Tool [1], and is a widely used data-driven application. Bioinformatics users utilize BLAST to ask questions about sequences of deoxyribonucleic acid (DNA) fragments that they observe in nature or in the lab. When using BLAST, the user selects a database of known sequences and then inputs their unknown sequence. The application returns *hits* which are a set of matches found in the database. Note, that most of these sequences are not exact matches, but are partial matches of the sequence. There are some key quality objectives that many users rely on. The *e-value* (expected value) is a number that describes how many times you would expect a match by chance in a database of that size. The lower the value is, the more significant the match. In many cases users filter for only e-values of zero (optimal). Another key metric called *percent identity* is a number that describes how similar the query sequence is to the target sequence (how many characters in each sequence are identical). The higher the percent identity is, the more significant the match. Users sometimes filter by a value of 100 or 99% (100% is optimal).

One common use case of a BLAST search is to obtain as many *quality hits* as possible so that the user can explore those and find out what is known about their input sequence. Underneath the covers BLAST uses a dynamic programming algorithm for matching, and this algorithm has been highly tuned over the years. The current version of BLAST [1] has over 50 configuration options that a user can modify. We only explore a part of this in our case study.

Suppose we want to optimize a BLAST query with the goal of obtaining the best e-value and percentage identity, while at the same time increasing the number of hits. An obvious first choice is to just run the default configuration. If we choose for instance, the second input sequence from this study, the default BLAST configuration returns 1 hit and has the max percentage identity and smallest e-value; it is the best we can do. If we optimize with a genetic algorithm, we will be unable to improve further; the default configuration is the best.

However, if we instead optimize using input sequence number 84, we see different behavior. The default configuration returns 3 hits with a median e-value of 0 and percentage identity of 100. But if we change 4 of the configuration options (away from the default) we can improve the number of hits to 8 while maintaining the minimal median e-value and maximum median percent identity possible. We have increased our quality hits by 5, therefore a user might want to

consider using the new parameters. In our case study, only one-third of 10,000 input sequences returns the default configuration as the optimal result (using our objective value), meaning the other two-thirds have an opportunity to be improved.

After running these two experiments, we have a conundrum. If we use the results from the genetic algorithm of input sequence number 2, we won't optimize correctly for input sequence number 84, and vice versa. We may also want to consider other user preferences such as the distance from the default configuration. As we can see, tuning this configuration space is complex and may be data dependent.

As we started building a genetic algorithm for optimization of this program, we were left with many open questions. For instance: (1) How can we confirm a genetic algorithm is working, given the range of behavior for different input data? (2) Can we find patterns of genetic algorithm behavior across sequences? (3) Do we need to run our genetic algorithm for all different inputs? As we iterated to tune the fitness and landscape that can guide a genetic algorithm for this program, the data dependency made this time consuming and ad-hoc. We learned that data-driven applications are difficult to tune and reason about when building a search algorithm, and asked if instead, we can find a better approach. We hypothesized the need for an exhaustive data set that can be easily queried. We present and use such a framework in the following section.

3 A Framework for Data-driven Exploration

We present our design of a Framework for a Data-driven Exploration in a search environment (FRDDE) that will help us understand the quality of our genetic algorithm, and the variability of optimizations within our dataset. Figure 1 shows an overview of FRDDE. We note that we only implemented this for a single data set in our case study, but the approach is general and we plan to build additional data sets for other data-driven systems as future work.

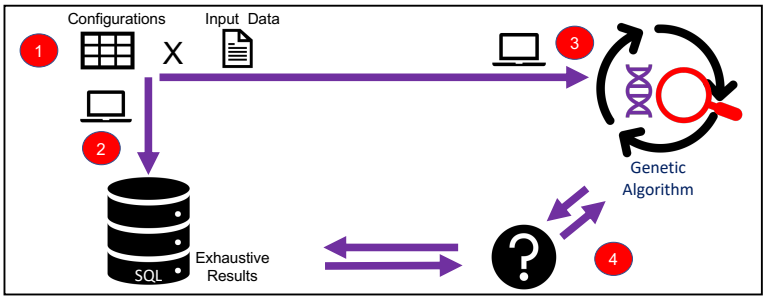

Fig. 1. Overview of the FRDDE framework. It starts with a set of configurations and input data (#1) and populates the database (#2). A genetic algorithm can then be run #3 and the database can be used to answer questions #4.

FrDdE begins (#1) with a model of the configurations and a set of inputs. The Cartesian product of these are generated as pairs (configurations, inputs) and then run against the application (#2). It is not necessary to define an exact fitness function yet, but to capture key measurements that can be used to build different fitness functions for this application. The relational data model is shown in Figure 2. We have a table of the configuration models where the primary key (Pk) is the configuration number and the other fields are the individual configuration options and their possible choices. Table 2 (outputs) contains the results of running the configurations and inputs. In this table we have a three-way primary key which we see as inputNo, configNo, and outputNo. This three-way key allows for applications which can return more than one output per run (many data-driven applications such as the one we use, do not return a single result). This table has a foreign key (Fk) which can be joined with Table 1. The other fields contain detailed information relevant to the use case and fitness which can be used during exploration. The last table is an aggregate table built from Table 2. It has one record per inputNo and configNo and uses the configNo as a foreign key. We can use this to build different fitness values and to explore different parts of the search space.

The next phase (#3) of FrDdE is to run a genetic algorithm (or other evolutionary algorithm) using the sample application and configuration model. This can be run on all inputs or a sample of inputs. In the last phase (#4), we can interactively ask questions and compare results from the database against the runs of the genetic algorithm.

Once FrDdE is complete, the goal is to use what we have learned to expand the configuration space and or data set, where an exhaustive analysis is no longer possible, with information learned from the exploration. We present an example exploration with FrDdE in the next section. We leave the expansion of our configuration space for future work.

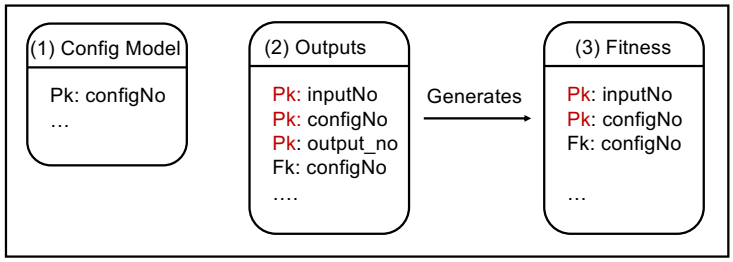

Fig. 2. FrDdE schema showing only the primary keys (Pk) and foreign keys (Fk). Table 1 contains the configuration model. It has one row for each of the configurations in the model. Table 2 is raw data from the application. Each returned hit has a record. It has a composite primary key, consisting of the configuration, input sequence and the number of the returned output (generated). The last table (Table 3) is a generated fitness table containing a fitness value along with the individual components of the fitness for each configuration and each input.

4 Case Study

We present a case study leveraging FrDdE.[4] While the long-term goal is to understand how useful FrDdE is in a data-driven search based environment, we perform this case study as an exemplar of its use. The questions we ask are:
RQ1: How effective and efficient is a genetic algorithm across all input data?
RQ2: What patterns, if any, emerge across all input data?
RQ3: How well does sampling capture patterns seen in the exhaustive data?

Table 1. Configuration Model for BLAST from [2]. The names of options are followed by the number of choices and their values. Default values are shown in bold.

Option Name	Abbr.	No. of Choices	Configuration Values
dust	dust	3	yes, no, **'20 64 1'**
lcase_masking	lcase	2	true, **false**
soft_masking	soft	2	**true**, false
ungapped	ungap	2	true **false**
xdrop_gap	xdgap	5	0, 0.1, 0.5, **30**, 100
xdrop_gap_final	xdgfin	5	0, 0.1, 0.5, 10, **100**
xdrop_ungap	xdugap	5	0, 0.1, 0.5, **20**, 100

4.1 Object of Study

As our software subject we use the popular bioinformatics tool, BLAST. We use the nucleotide sequencer (BLASTn), version 2.6.0. This is the same subject studied by Cashman et al. [2] and was sensitive to changing configurations. We use their configuration model (shown in Table 1). The BLAST model has 7 configuration options. Three are binary, one has 3 choices, and the other three have 5 choices. This leads to an exhaustive configuration space of 3,000 configurations. This is not the complete configuration space for BLAST, but one that is tractable to study exhaustively. Their input data set contains 10,000 input sequences from the yeast genome. We use these as our input data set.

4.2 Genetic Algorithm Implementation

Our chromosome has seven genes (one for each configuration option). The alleles are configuration choices.
Fitness. The fitness for this case study is based on the scientific use case presented earlier. However, we can theoretically use alternative fitness functions within FrDdE. When using BLAST, a biologist may want to increase the number of *quality hits* they match from the database. Often, users restrict two quality values, e-value and the percentage identity. Instead of filtering these values, we

[4] supplementary data website: https://github.com/LavaOps/ssbse-2020-FrDdE

use them for optimization, since a value that is close to the optimal may be of interest. We also include another component in our fitness, the distance from the default configuration. It has been documented that users prefer not to change too many settings [4], hence our goal is to stay close to the original configuration.

Since we are attempting to maximize the number of hits (and expect more than one), the e-value and percentage identity are aggregate values. We chose the median for this (we have also experimented with the mean and observe similar results). Putting these together our fitness is shown next. We are maximizing. The various weights for the components were determined heuristically.

$$w_{\text{e-val}} = \begin{cases} 10, & \text{if median e-value} = 0 \\ \dfrac{1}{\text{median e-value}}, & \text{otherwise} \end{cases} \tag{1}$$

$$w_{\text{distance}} = \begin{cases} 10, & \text{if distance} = 0 \\ \dfrac{7}{\text{distance}}, & \text{otherwise} \end{cases} \tag{2}$$

$$\textit{fitness} = \text{hits} \times 10 + w_{\text{e-val}} + \text{median(percent identity)}/2 + w_{\text{distance}} \tag{3}$$

Equation 1 shows the weighted e-value, (or $w_{\text{e-val}}$) which takes a value of 10 when the median e-value is 0. Otherwise it is the inverse. Equation 2 is the calculation of the weighted distance (w_{distance}). Under the default configuration this value is set to 10, otherwise we normalize by the number of configuration options (7). Last, Equation 3 weights these together. Hits are weighted by 10 to make them as important as $w_{\text{e-val}}$ and distance. The percent identity is reduced by one half. Although not part of this study, we made use of the database to help tune the weights.

Genetic Algorithm Parameters. We heuristically tuned our algorithm. We ended with a population size of 16. When we seed the initial random population we generate all but one chromosome randomly. The last chromosome is made up of default values for the program to ensure its existence in each population. We use one point crossover selecting a random point (within the middle two thirds of the chromosome for each pair). We pair even and odd numbered chromosomes in rank order (e.g. the best with the third best and second best with the fourth best) to maintain some diversity.

We use a rank selection with elitism. This has an important effect for our use case. If the default configuration is the best fitness, this will propagate to be the best overall at the end. We tuned the mutation rate and settled on a value of 0.5 for this program. We randomly select 50 percent of the genes in the population and change the value randomly (with replacement). We do not mutate genes of the two fittest (elite) chromosomes. We experimented with different numbers of generations. We usually see convergence relatively quickly (within 20-30 generations), therefore we chose 50 which will balance the need to solver harder cases, with being tractable with respect to runtime.

4.3 Experimental Setup

All experiments are run on a parallel computing RedHat Enterprise Linux 7 Server with Intel(R) Xeon(R) Gold 6244 CPU 2.60GHz nodes. We collected the exhaustive data by running BLASTn on 10,000 input sequences against the yeast database. The experiments were run such that each input sequence was tested under all 3,000 possible configuration options. These led to a total of 30 million BLASTn calls. 100 of these jobs were executed in parallel to make this scalable. Each job was allocated 1 node with 5GB memory. The following outputs were collected post every BLASTn call: the sequence number, its corresponding configuration number, the blast sequence id, total hits generated, the percentage identity and e-value observed, and the start and end positions of the subject sequence against which the match was observed.

Next, we used a MariaDB fork of MYSQL, version 5.5.56 in a singularity container version 2.6.0. We constructed a database having three primary tables, which are concrete instantiations of the FRDDE tables. Additional temporary and derived tables were used for intermediate computations and to fetch results from the database.

In total we have 96,875,728 records in FRDDE. This is because we have a record for each hit. We also recorded the combination of configuration numbers and the sequences against which no hits were reported by BLASTn. All such sequence numbers along with their configurations were stored in a separate table. Next, we constructed the fitness table which is derived from the output table. We store the count of hits returned per configuration for a specific input sequence. We compute the median percentage identity and e-value over all hits recorded for a sequence under a given configuration number, the distance from the default configuration, and the computed fitness value. We annotated each possible distinct configuration set across its 3,000 possibilities with an integer number for ease of reference and access across tables. These annotations along with the combination of 7 individual configuration options such as dust="20 64 1", specific to a configuration number, was also stored in the configuration table.

4.4 Threats to Validity

With respect to generalization we used only a single software subject, a single type of input file, and a single fitness for this study. However, we use 10,000 inputs and this is a common scientific application that is widely used and has been studied in other papers on configurable software. We used the same configuration space and input files as another paper to avoid bias due to our selection. With respect to threats to internal validity, we wrote programs to perform many of the parts of this experiment and acknowledge there could be errors. We did extensive checking of the results and compared the database against the genetic algorithm. We also make the data and artifacts from these experiments publicly available on our supplementary website so the community can validate and reuse our results.

5 Results

In this section we present results for each of our research questions in order.

5.1 RQ1: Effectiveness

To evaluate the effectiveness of the genetic algorithm we compare the configurations reported in the final population of the genetic algorithm to the best configurations found in the database. Of the 10,000 input sequences, 19 of them never find any hits in any configuration. We remove these from our analysis, leaving 9,981 input sequences. We then analyze for 9,981 input sequences if the fitness of the best configuration returned by the genetic algorithm is the same as the best fitness from the database. Results over two runs of the genetic algorithm can be seen in Table 2. The first result column provides the number of cases where we match the best fitness (9,721 and 9,713). This is also shown as a percentage (97.4% and 97.3%). For 3,382 of the inputs (34%), the default is the best fitness. We discuss this implication in RQ2. For the remaining sequences where a best configuration was not found by the genetic algorithm, we ask if the returned configuration has a higher fitness than the default. In all cases (260 and 268) the answer is yes. While we were not able to find the optimal, we still improved over the default. Hence, we conclude that our genetic algorithm is working well on this data set; finding an optimal solution for more than 97% of the inputs.

To confirm the genetic algorithm is useful even in this small configuration space, we randomly selected 500 inputs and randomly generated five sets of 1,000 chromosomes for each input. Our rationale is that each run of our genetic algorithm would evaluate a maximum of 800 chromosomes after 50 generations with our population size of 16 (we usually converge by generation 20-30). One of the chromosomes is seeded as the default configuration. 67.5% of the time, we find the best configuration option across all runs for all 500 inputs. There is also little variation (standard deviation of 8.7 for the number of times the maximum is found). However, for 185 inputs (or 37%), the default already is the best fitness. If we remove those inputs from our data, random finds the best fitness 48.38% of the time. However, in almost all cases, when the best fitness is not found, we can improve on the default (99.3%). We conclude that while random does relatively well and almost always finds the best or a better fitness than random, the genetic algorithm performs better overall. As the configuration size grows, we expect the gap in performance to increase.

To evaluate the efficiency we observe the runtime (in minutes) of the genetic algorithm. On average it takes approximately 1 minute for the genetic algorithm to run on a single input sequence, with a maximum of 2.8 minutes. If we sum the runtimes across all of the genetic algorithms (these were run in parallel) it took 8.11 days for the first run and 7.84 days for the second run. Note, these are sequential times. The actual clock time was faster due to parallelization.

Summary RQ1: We conclude that the genetic algorithm is effective. While it is relatively efficient for a single input, it takes days to run the entire data set.

Table 2. Comparison of matches of maximum fitness of the genetic algorithm on 10,000 sequences compared against database (known maximum). 19 returned no hits in any configuration and are removed. Remaining # > default are runs which did not match the best fitness, but still improve on default. Times is shown for the genetic algorithm in minutes.

	No. of Best (of 9981)	Percent	Remaining # > Default	Avg. runtime	Std deviation	Min runtime	Max runtime
Run 1	9721	97.40	260	1.17	0.76	0.26	2.78
Run 2	9713	97.31	268	1.13	0.66	0.23	2.71

5.2 RQ2: Patterns

We utilize the database information to ask if there are patterns in the data. Of the possible 3,000 configurations, only 112 unique configurations achieve the best fitness for at least one of the 9,981 input sequences. This indicates that only 3.7% of the configurations are considered optimal for this fitness and set of inputs. We sorted these configurations by the times they occur. Since a maximal fitness for a particular input sequence may have more than one possible configuration, we normalize the counts so that each input sequence is accounted for once. For example, if the sequence has two configurations with the optimal fitness, we weight each of them by 1/2. Table 3 shows the top configurations, sorted in descending order by count of occurrences. These make up 92.4% of the best configurations. The first configuration (#1724) is the default. This occurs 3,382 or 34% of the time. 66% of the time, another configuration is optimal. We highlight the changes from the default in each of the other commonly seen configurations.

Table 3. Most frequent configurations (top 9 out of 112) appearing as the best configuration. The left most column is the configuration number, followed by the normalized times the configuration is seen. The rest of the columns indicate the configuration options selected. Red indicates a change from the default configuration (#1724)

Default	Config No.	Count	dust	lcase	soft	ungap	xdgap	xdgfin	xdugap
YES	1724	3382	20 64 1	FALSE	TRUE	FALSE	30	100	20
–	1597	998	20 64 1	FALSE	TRUE	**TRUE**	30	100	**0.1**
–	1598	998	20 64 1	FALSE	TRUE	**TRUE**	30	100	**0.5**
–	1847	828	20 64 1	FALSE	**FALSE**	**TRUE**	30	100	**0.1**
–	1848	828	20 64 1	FALSE	**FALSE**	**TRUE**	30	100	**0.5**
–	1974	820	20 64 1	FALSE	**FALSE**	FALSE	30	100	20
–	1599	784	20 64 1	FALSE	TRUE	**TRUE**	30	100	20
–	2724	300	**no**	FALSE	TRUE	FALSE	30	100	20
–	1849	227	20 64 1	FALSE	**FALSE**	**TRUE**	30	100	20

We can identify certain configuration option values that are important, such as ungap. Combining ungap (TRUE) with xdugap (0.1 or 0.5) account for the

two next best configurations overall. Interestingly, neither of these options are set to these values in the default. We also see options like lcase which do not appear to have a strong effect. To analyze this further we look at the distribution of each configuration option's values in Figure 3. We can see that in most cases the default values will work well for any given input (the black bars). This matches what we see in the best configurations as the default is the best overall. We see that options such as ungap, soft, and xdugap seem to have influence, which others (like lcase) have no influence.

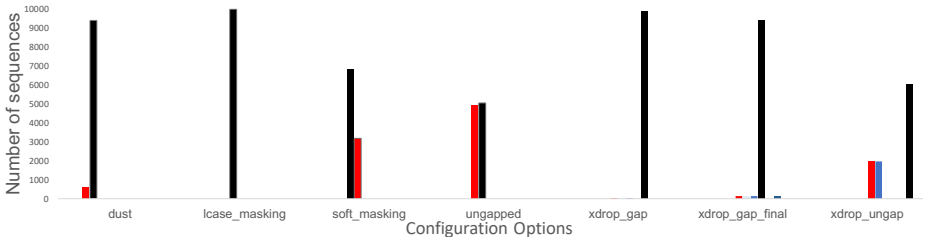

Fig. 3. Number of times each value for the seven configuration options appear (normalized) in the configuration with the highest fitness. Black solid bars indicate the default value for that configuration option. Other colors are alternative choices. Spaces indicate that the value does not appear.

Summary RQ2: We observe a set of patterns for the input data, demonstrating that it may be possible to sample the inputs. No single configuration works best across all input sequences, however we have reduced our configuration space from 3,000 to 112 (by 96.3%).

5.3 RQ3: Sampling

We now examine the effectiveness of sampling, i.e. can identify the same pattern of best configurations as we did in RQ2? We use random sample sizes of 10, 100 and 1,000. While the genetic algorithm's search space is not reduced, we can potentially avoid running it against all 10,000 inputs, which may provide a significant time savings. We repeat each sample 5 times. Table 4 reports how many times a sample run was able to identify any of the best (or top) 9 configurations seen in Table 3. Ideally, the majority of configurations reported by the sampled GAs would be from that set. In a sample size of 10 we see between 70-100% of reported configurations appearing from the top 9 configurations. As we move to a sample size of 100, 88-92% are from the top 9. We see slightly less (88.8%) in a sample size of 1,000. This may be due to more opportunities for noise; a sequence leading to a less common configuration is more likely to be selected.

Table 5 shows another view of this data. The first two columns show the average and standard deviation of the percent of times the default configuration is selected. Remember, the default appeared in 34% of the exhaustive data in the

Table 4. Counts of the occurrences of the top configurations in the database. For each of the best configurations from Table 3 we show the number of times found in the five runs (R1-R5) for different sample sizes (10,100,1000). The percentage of the best configurations found by the GA sample relative to the ground truth best configurations found are displayed in the last row.

	DB	Sample Size 10					Sample Size 100					Sample Size 1,000				
ConfigID	**Occur.**	**R1**	**R2**	**R3**	**R4**	**R5**	**R1**	**R2**	**R3**	**R4**	**R5**	**R1**	**R2**	**R3**	**R4**	**R5**
1724	3382	3	4	5	4	4	34	24	35	30	46	336	328	348	352	340
1597	998	2	0	1	0	2	8	10	13	7	10	94	101	88	117	111
1598	998	1	1	1	0	2	12	15	7	8	6	117	92	100	86	92
1847	828	0	1	1	1	1	2	11	7	7	8	77	88	85	89	65
1848	828	0	0	1	1	1	11	9	7	9	5	71	85	83	75	90
1974	820	1	0	1	1	0	11	7	8	9	7	78	75	68	76	74
1599	784	1	1	0	0	0	5	14	9	12	5	78	58	69	43	74
2724	300	0	0	0	0	0	4	2	3	4	0	28	35	29	20	26
1849	227	0	0	0	2	0	2	0	1	3	1	10	21	15	22	17
% of Best		**80**	**70**	**100**	**90**	**100**	**89**	**92**	**90**	**89**	**88**	**88.9**	**88.3**	**88.5**	**88**	**88.9**

Table 5. Average and standard deviation across five runs of each sample size. We report the percentage of occurrences of the default configuration. Uniqueness is measured as the percentage of unique configurations. We report the total runtime in hours.

	# Default		**Uniqueness**		**Runtime (hrs)**	
	AVG	STD	AVG	STD	AVG	STD
Sample Size 10	40.00	7.07	60.00	7.07	0.06	0.00
Sample Size 100	33.80	8.07	16.60	1.67	3.39	0.10
Sample Size 1,000	34.08	0.95	4.38	0.27	15.94	2.91

database. The next two columns, represents a metric we call Uniqueness. This shows convergence of a configuration pattern (lower is better). It is the percent of unique configurations observed as the best fitness. For instance, if we have a sample of 10 inputs, the maximum number of best configurations returned is 10 (all are unique). If we see 6 unique configurations, then uniqueness is 60%. As we move from samples of 10 to 1,000 uniqueness drops from 60% to 4.38%. The uniqueness from the database (RQ2) is 112/3,000 or 3.73%. The last two columns show the total sequential runtime to complete the sample, when we re-ran the genetic algorithm on each input in the sample. A sample of 10 takes on average about 3 minutes, 100 about 3.4 hours, and a sample of 1,000 takes just under 16 hours. To compute the complete data set on 10,000 inputs as in RQ1 takes approximately 8 days. Sampling can reduce the runtime to find the best configuration patterns by over 90% while still identifying over 80% of the best configurations reducing uniqueness.

Summary RQ3: Sampling input data provides similar patterns to the exhaustive data set while compressing the percentage of unique results and runtime significantly.

6 Related Work

In the closest work to ours, Nair et al. define the notion of data-driven search-based software engineering (DSE) [16]. They call for combining data mining and search-based software engineering and present a set of challenges and research in this domain. While they discuss techniques such as parameter tuning and transfer learning, they do not explicitly examine implications for using search under specific data inputs while tuning configurations as we do in this work.

The largest body of work using search algorithms for configurable software falls into two main categories. The first is in testing of software configurations where search is used to find samples [3, 6, 9] that are representative of configurations which are likely to find faults. The general idea of configurability in software which can lead to different faults is also well studied (e.g. [14, 15, 19, 22]). There has also been extensive research on optimizing configurations that satisfy sets of constraints [5, 17, 18, 21]. While all of this research uses search-based algorithms, none that we know of focus on the problem of changing input data.

Another line of research is for system performance which involves building prediction or influence models for configurable systems [20]. The goal is to build a model that describes the configuration space. This work differs in that it focuses primarily on quality attributes (not functional as in our work) and describes the configuration space for prediction, rather than optimization. Some research in this domain examines the notion of transfer learning [7, 8], which attempts to use knowledge from one workflow for a different workflow. This has a similar flavor to our work, but the focus is on performance and we do not use learning techniques. Nair et al. proposed the idea of bellwether configurations; those which are indicative of good or bad performance across all workflows [10, 11, 17]. Interestingly, we see similar results where there are specific configurations that seem to matter across all input data, although our objective goals differ. With respect to functionality, Garvin et al. [4] documented the locality of failures within a configuration space, however they do not explore faults in a data-driven environment. Cashman et al. [2] presented problems related to configurability in bioinformatics tools. The work suggests the need for optimization, but does not go as far as to provide an optimization technique. We use their software subject (BLAST) and model in this work.

Finally, Landgon et al. have proposed the use of program data for evaluation as a fitness function in genetic improvement [12, 13]. The difference with that work is that the data is the fitness function, where as we use a traditional fitness function, but are looking at results across different input data.

7 Conclusions and Future Work

In this paper we presented FrDdE, an approach for optimizing functional objectives in a highly-configurable data-driven application. We built a database of almost 100 million records using an exhaustive configuration space which is a part of the BLAST bioinformatics tool, and studied its behavior on 10,000

different input sequences. We find that our genetic algorithm is effective, and see patterns emerge across input sequences. While we don't find a single configuration that is best, the configuration space is reduced from 3,000 to 112 and only 9 configurations account for over 92% of the results. We demonstrate that sampling inputs can find similar patterns to the full data set which leads us to believe we can develop a light-weight technique for data-driven search based configuration optimization. Our ultimate goal is to use FrDdE to tune optimization techniques and to explore data sets before moving to larger configuration spaces where exhaustive enumeration is not possible.

In future work we plan to expand FrDdE further by (1) increasing the experimental configuration space to see if results learned still hold, (2) applying FrDdE to additional input data sets and (3) using FrDdE for different data-driven applications. We also plan to incorporate learning into FrDdE to see if we can learn more about the patterns of optimal configurations.

Acknowledgements

This work is supported in part by NSF Grant CCF-1901543 and by The Center for Bioenergy Innovation (CBI) which is supported by the Office of Biological and Environmental Research in the DOE Office of Science.

References

1. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. Journal of Molecular Biology **215** (2018), https://blast.ncbi.nlm.nih.gov/
2. Cashman, M., Cohen, M.B., Ranjan, P., Cottingham, R.W.: Navigating the maze: The impact of configurability in bioinformatics software. In: International Conference on Automated Software Engineering. pp. 757–767. ASE (September 2018)
3. Garvin, B.J., Cohen, M.B., Dwyer, M.B.: Evaluating improvements to a meta-heuristic search for constrained interaction testing. In: Empirical Software Engineering (EMSE). vol. 16, pp. 61–102 (February 2010)
4. Garvin, B.J., Cohen, M.B., Dwyer, M.B.: Failure avoidance in configurable systems through feature locality. In: Assurances for Self-Adaptive Systems - Principles, Models, and Techniques, Lecture Notes in Computer Science, vol. 7740, pp. 266–296. Springer (2013)
5. Henard, C., Papadakis, M., Harman, M., Le Traon, Y.: Combining multi-objective search and constraint solving for configuring large software product lines. In: IEEE/ACM 37th IEEE International Conference on Software Engineering. vol. 1, pp. 517–528 (2015)
6. Henard, C., Papadakis, M., Perrouin, G., Klein, J., Heymans, P., Le Traon, Y.: Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. IEEE Transactions on Software Engineering **40**(7), 650–670 (2014)
7. Jamshidi, P., Siegmund, N., Velez, M., Kästner, C., Patel, A., Agarwal, Y.: Transfer learning for performance modeling of configurable systems: An exploratory analysis. In: International Conference on Automated Software Engineering (ASE). pp. 497–508 (November 2017)

8. Jamshidi, P., Velez, M., Kästner, C., Siegmund, N.: Learning to sample: Exploiting similarities across environments to learn performance models for configurable systems. In: Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 71–82. ESEC/FSE (2018)
9. Jia, Y., Cohen, M.B., Harman, M., Petke, J.: Learning combinatorial interaction test generation strategies using hyperheuristic search. In: IEEE/ACM 37th IEEE International Conference on Software Engineering. vol. 1, pp. 540–550 (2015)
10. Krishna, R., Menzies, T., Fu, W.: Too much automation? The bellwether effect and its implications for transfer learning. In: 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 122–131 (2016)
11. Krishna, R., Menzies, T.: Bellwethers: A baseline method for transfer learning. IEEE Transactions on Software Engineering **45**(11), 1081–1105 (2019)
12. Langdon, W.B.: Big data driven genetic improvement for maintenance of legacy software systems. SIGEVOlution newsletter of the ACM Special Interest Group on Genetic and Evolutionary Computation **12**(3), 6–9 (2019)
13. Langdon, W.B., Krauss, O.: Evolving sqrt into $1/x$ via software data maintenance. In: Coello, C.A.C. (ed.) GECCO '20: Genetic and Evolutionary Computation Conference, Companion Volume. pp. 1928–1936. ACM (July 2020)
14. Medeiros, F., Kästner, C., Ribeiro, M., Gheyi, R., Apel, S.: A comparison of 10 sampling algorithms for configurable systems. In: International Conference on Software Engineering (ICSE). pp. 643–654. ACM (May 2016)
15. Meinicke, J., Wong, C.P., Kästner, C., Thüm, T., Saake, G.: On essential configuration complexity: Measuring interactions in highly-configurable systems. In: International Conference on Automated Software Engineering (ASE). pp. 483–494. ACM (September 2016)
16. Nair, V., Agrawal, A., Chen, J., Fu, W., Mathew, G., Menzies, T., Minku, L., Wagner, M., Yu, Z.: Data-driven search-based software engineering. In: IEEE/ACM 15th International Conference on Mining Software Repositories (MSR). pp. 341–352 (2018)
17. Nair, V., Menzies, T., Siegmund, N., Apel, S.: Using bad learners to find good configurations. In: Joint Meeting on Foundations of Software Engineering. p. 257–267. ESEC/FSE (2017)
18. Oh, J., Batory, D., Myers, M., Siegmund, N.: Finding near-optimal configurations in product lines by random sampling. In: Joint Meeting on Foundations of Software Engineering. p. 61–71. ESEC/FSE (2017)
19. Qu, X., Cohen, M.B., Rothermel, G.: Configuration-aware regression testing: An empirical study of sampling and prioritization. In: International Symposium on Software Testing and Analysis. pp. 75–86. ISSTA, ACM (2008)
20. Siegmund, N., Grebhahn, A., Kästner, C., Apel, S.: Performance-influence models for highly configurable systems. In: European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE). pp. 284–294. ACM Press (August 2015)
21. Xiang, Y., Zhou, Y., Zheng, Z., Li, M.: Configuring software product lines by combining many-objective optimization and sat solvers. ACM Trans. Softw. Eng. Methodol. **26**(4) (Feburary 2018)
22. Yilmaz, C., Dumlu, E., Cohen, M.B., Porter, A.: Reducing masking effects in combinatorial interaction testing: A feedback driven adaptive approach. IEEE Transactions on Software Engineering **40**(1), 43–66 (January 2014)