Optimizing Product Orders Using Graph Algorithms for Improving Incremental Product-Line Analysis

Sascha Lity
TU Braunschweig,
Braunschweig, Germany

Mustafa Al-Hajjaji
University of Magdeburg,
Magdeburg, Germany

Thomas Thüm, and
Ina Schaefer
TU Braunschweig,
Braunschweig, Germany

ABSTRACT

The individual analysis of each product of a software product line (SPL) leads to redundant analysis steps due to the inherent commonality. Therefore, incremental SPL analyses exploit commonalities and focus on the differences between products to reduce the analysis effort. However, existing techniques are influenced by the order in which products are analyzed. The more similar subsequently analyzed products are, the greater is the potential reduction of the overall analysis effort as similar products imply less differences to be analyzed. Hence, an order of products, where the total number of differences is minimized, facilitates incremental SPL analyses. In this paper, we apply graph algorithms to determine optimized product orders. We capture products as nodes in a graph, where solution-space information defines edge weights between product nodes. We adopt existing heuristics for finding an optimal solution of the traveling salesperson problem to determine a path in the product graph with minimal costs. A path represents an optimized product order w.r.t. minimized differences between all products. We realize a prototype of our approach and evaluate its applicability and performance showing a significant optimization compared to standard and random orders.

CCS Concepts

•**Software and its engineering** → **Software product lines;** •**Mathematics of computing** → Graph algorithms;

Keywords

Delta-Oriented Software Product Lines; Product Orders; Graph Algorithms

1. INTRODUCTION

The analysis of software product lines (SPL) [28] is a challenging task [32]. An SPL represents a family of similar software systems of a certain application domain, where their commonality and variability are explicitly specified by means of features. Due to the combination of features resulting in a huge set of products, the individual analysis of each product is often infeasible. Furthermore, based on the inherent commonality between products, the analysis leads to redundant results and analysis steps. To cope with these issues, incremental SPL analysis techniques [9, 14, 20, 24, 22, 32] focus on the differences between products and exploit the commonalities to provide their results more efficiently. One factor influencing incremental SPL analyses, can be the order in which the products are analyzed. The more similar subsequently analyzed products are, the greater is the potential reduction of the overall analysis effort as similar products imply less differences to be analyzed [22].

Existing approaches for computing product orders [1, 3, 10, 15, 11, 27] mainly apply coverage-driven product prioritization to find an order in which software faults are detected earlier. By always selecting the most dissimilar product [1, 3, 15, 11] as next product to be tested, the early coverage of feature interactions is increased. However, for incremental strategies the dissimilarity of subsequent products results in an increasing analysis effort and may have a negative impact on approaches such as SPL regression testing exploiting the analysis results [22]. Hence, a product order is required, where subsequent products are more similar to each other.

In this paper, we encode the problem of finding such a product order as traveling salesperson problem (TSP) [29, 4], where a tour through all nodes of a graph starting and ending in the same node has to be identified with minimal costs (i.e., distances). Based on the definition of a product graph, where nodes denote products and edges represent the subsequent analysis of products, TSP solvers [8] and graph algorithms, i.e., heuristics [29] are applicable to find optimized product orders with a minimal overall number of differences between products. Different measurements are reasonable to define edge weights required for instantiating a TSP. In contrast to product-prioritization techniques [15, 3], where the distance between feature configurations is used, we incorporate solution-space information for the weight computation. Based on delta modeling [6], where the commonality and differences between products are explicitly specified by means of deltas, we directly determine the number of differences between products by deriving regression deltas. A regression delta captures change operations to transform a product into another one and, thus, defines the differences between two arbitrary products [23] used to define the respective edge weight. Based on the product graph, we adopt existing heuristics proposed for solving a TSP to find an optimized path, i.e., product order, where the

VaMoS '17, February 01 - 03, 2017, Eindhoven, Netherlands

DOI: http://dx.doi.org/10.1145/3023956.3023961

overall number of differences gets minimized. We prototypically implemented our approach to evaluate its applicability and performance by means of four existing delta-oriented SPLs and artificial data. In this paper, we focus only on the ordering problem and abstract from concrete analysis strategies which can benefit from optimized orders and leave this investigation for future work. In summary, we contribute the following:

- Problem encoding as TSP incorporating the definition of product graphs and the adoption of existing heuristics for computing optimized product orders
- Tool support and evaluation of applicability and performance

The remainder of the paper is structured as follows. In Sect. 2, we describe our foundations. We propose our novel product order concept in Sect. 3. In Sect. 4, we present our evaluation and its results. We discuss related work in Sect. 5 and conclude the paper in Sect. 6.

2. DELTA-ORIENTED PRODUCT LINES

For SPL development, various variability modeling techniques exist [31] like the transformational approach delta modeling [6]. Transformational approaches realize an SPL by specifying transformations, e.g., the addition of an element, w.r.t. a designated core model. To generate a product-specific model, a set of transformations is stepwise applied to the core. We exploit this information for computing an optimized product order on which incremental SPL analysis techniques can be applied.

Delta modeling [6] is a flexible and modular variability modeling approach already adapted for various artifact types such as finite state machines [23] and software architectures [23]. Based on a *core model* of a *core product* $p_{core} \in P_{SPL}$, a set of deltas Δ_{SPL} is defined to transform the core into the models of the remaining products $p_i \in P_{SPL} \backslash \{p_{core}\}$. By P_{SPL}, we refer to the set of all products of an SPL, whereas Δ_{SPL} comprises all valid deltas to generate product-specific models. A *delta* $\delta = \{op_1, \ldots, op_m\} \subseteq \mathcal{OP}$ encapsulates *change operations*, i.e., the addition or removal of model elements to or from the core, where $\mathcal{OP}$ comprises for each model element of the SPL an addition or removal operation, to transform the core into a product p_i. In this paper, we abstract from modifications of model elements as we are able to convert a modification by removing the old element version and adding the new version of the element to the model. In addition to the specification of deltas between the core and a certain product, we are able to derive so called regression deltas δ_{p_i,p_j} defining the differences between two arbitrary products $p_i, p_j \in P_{SPL}$ by incorporating their deltas δ_{p_i} and δ_{p_j} using an adapted version of the symmetric difference operator [23].

EXAMPLE 1. *Consider the state machine shown at the top of Fig.* 1 *used as core* p_{core}*. By applying* δ_{p_1} *and, thus, by adding transition* t_3*,* t_4*, and state* S_3 *to* p_{core}*, we obtain* p_1 *on the left hand side. For* p_2*, we apply* δ_{p_2} *to* p_{core}*. To transform* p_1 *into* p_2*, we derive the regression delta* δ_{p_1,p_2}*.*

3. GRAPH-BASED PRODUCT ORDERING

In this section, we propose the computation of optimized product orders for improving incremental SPL analysis. We first introduce a weighted product graph incorporating solution-space information to define edge weights. Based on this graph, we secondly describe how to instantiate the TSP to find an optimized product order with a minimal overall number of differences between all products. Finally, we shortly discuss some open issues concerning the computation of optimal product orders to be addressed in future work.

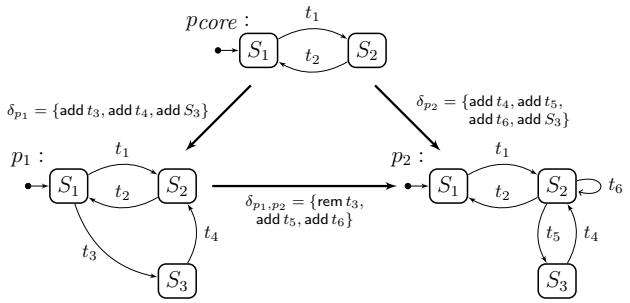
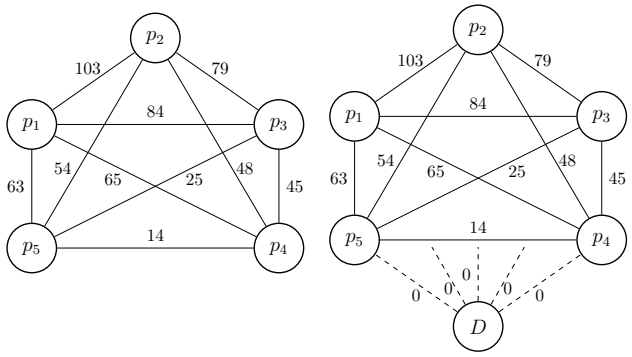


Figure 1: Sample Delta-Oriented State Machines

Define a Weighted Product Graph. To find an optimal product order facilitating incremental SPL analysis [22], we exploit in this paper the well-known TSP [29, 4]. For a successful problem encoding, we require a graph and its properties to build the basis for instantiating and solving a TSP. Thus, we define a *product graph* $G_{SPL} = (P_{SPL}, E)$ comprising a finite set of nodes, i.e., the set of all products P_{SPL} of an SPL, and a finite set of edges E, where each edge $e_{p_i,p_j} \in E$ denotes the potential step from p_i to p_j, e.g., during incremental analysis. Each edge e_{p_i,p_j} has further a weight specified by a weighting function $\omega : E \to \mathbb{N}$ to allow for the reasoning about distances between products. In literature [15, 3], promising candidates for ω are the Hamming or Jaccard distance measurements incorporating the feature configurations of products to derive the similarity used as edge weight. As we focus on delta modeling [6], we specify edge weights based on the derivation of the regression delta δ_{p_i,p_j} between products $p_i, p_j \in P_{SPL}$. Hence, the sum of captured operations in δ_{p_i,p_j} define the edge weight as follows $\omega(e_{p_i,p_j}) = |\delta_{p_i,p_j}| = |\{op_1, \ldots, op_m\}|$. We interpret the addition and removal of elements equally such that the corresponding operations have the same impact to the model and, therefore, count each operation with 1. This interpretation allows for the definition of our product graph as *undirected* graph. In contrast, by incorporating additions and removals differently, i.e., the corresponding operations differ in their impact to the model, we have to adapt our product graph to be directed, i.e., each edge has a concrete direction as the corresponding regression deltas δ_{p_i,p_j} and δ_{p_j,p_i} will result in different weights.

A product graph G_{SPL} needs to satisfy three additional properties. First, the graph is *complete*, i.e., for every pair $p_i, p_j \in P_{SPL}$ there exists exactly one edge $e_{p_i,p_j} \in E$, as we allow a potential step between two arbitrary products during incremental analysis. The second property directly follows from the completeness, i.e., the graph is *connected* such that for every pair $p_i, p_j \in P_{SPL}$ a corresponding path within the graph exists. A *path* within a product graph representing a certain product order is defined as a sequence of products with distinct start and end products, whereas a *tour* denoting the result of a TSP solution is specified by a sequence starting and ending in the same product. Third, the graph fulfills the *triangle inequality*, i.e., for all distinct products $p_i, p_j, p_k \in P_{SPL}$ holds $\omega(e_{p_i,p_j}) \leq \omega(e_{p_i,p_k}) + \omega(e_{p_k,p_j})$. If



(a) Product Graph of P_{SPL} (b) Extended Product Graph

Figure 2: Sample Graph for Finding Product Orders

we consider the triangle of products in Fig. 1 again, we can see, that both product-specific deltas share two additions. The regression delta δ_{p_1,p_2} between products p_1 and p_2 exploits this information and, therefore, only captures operations corresponding to the operations which differ between the product-specific deltas. Hence, the regression delta comprises less operations than the combination of both product-specific deltas. In case, that both product-specific deltas are disjoint, the regression delta will comprise an equal number of operations as both disjoint product-specific deltas. In both scenarios, the triangle inequality holds.

On this basis, we instantiate a (symmetric) TSP [29] as follows: Given a product graph G_{SPL} with edge weights $\omega(e_{p_i,p_j})$, find the shortest tour. By shortest tour, we refer to a tour that has the minimal overall number of differences between all products in the tour. The overall number is defined by the sum of edge weights such that $\omega(E') = \sum_{e_{p_i,p_j} \in E'} \omega(e_{p_i,p_j})$ holds, where $E' \subseteq E$ denote the subset of edges contained in the shortest tour.

Example 2. *In Fig. 2(a), a product graph is shown containing five nodes w.r.t. a sample SPL* $P_{SPL} = \{p_1, p_2, p_3, p_4, p_5\}$. *Each edge between nodes* p_i *and* p_j *is weighted by the number of change operations captured in the regression delta* δ_{p_i,p_j}. *For instance, the edge* e_{p_1,p_4} *gets* 65 *as weight.*

Computing an Optimized Product Order. We exploit the product graph definition to instantiate the TSP [29, 4] for finding an optimal product order, where the overall number of differences between products is minimized. Solving the TSP for a given product graph will result in a tour, where all products are visited once. For a product order, we solely need a path within the graph, where we still visit all products only once. Therefore, we extend our product graph by adding a dummy node D. In addition, we connect the dummy node to all other product nodes $p_i \in P_{SPL}$ with respective new edges e_{D,p_i} comprising the weight 0. Based on this graph extension, we are able to solve a TSP, where the tour starts and ends in the dummy node D. To obtain an optimal path from the computed optimal tour, we remove the dummy node and its two connected edges from the tour.

As the TSP is NP-complete [29, 4] finding the optimal solution by applying a TSP solver is dependent from the number of products to be analyzed. To cope with larger product sets, we adopt existing incremental heuristics for solving a given TSP by approximating from the optimal solution [29].

The first heuristic we adopt is the *nearest neighbor heuristic* [29] which is a pure *Greedy* algorithm. By using this heuristic, we determine the next product to be added to the order by selecting the nearest neighbor, i.e., the most similar product w.r.t. minimal differences, of the last added product. Due to the triangle inequality satisfied by the product graph, we find the next product by examining the adjacent product nodes and the respective weighted edges. For the path computation, we can choose any product as starting point. Based on delta modeling [6], we start with the core p_{core} as it builds the basis for all defined deltas specifying the differences between products used for the edge weights. In addition, we do not determine a tour as the original nearest neighbor heuristic, but rather a path through the product graph such that we ignore the addition of the returning edge from the last added product node to the starting product node. In summary, to find a product order with the nearest neighbor heuristic [29], we apply the following steps:

1. Select p_{core} as first product
2. Find the edge with the minimal differences compared to the last added product
3. Add target product at the end of the path
4. Repeat Step 2 and 3 until all products are added

As the product graph is complete, the selection of the nearest neighbor always results in an approximation of the optimal solution, where the complexity is $O(n^2)$ in the number of product nodes. A potential drawback of the heuristic is that in the first steps we add very similar products to the order, but we may add very dissimilar ones in the end as we have no insights about the remaining products to be added reducing the quality of the approximation.

We improve this situation by extending the nearest neighbor heuristic [29] such that we examine the next product to be added from the begin and the end of the already computed path. In this way, we integrate a *look up* to find the more suitable product to be added enhancing the resulting approximation, where the complexity is still $O(n^2)$ in the number of product nodes. For this purpose, we split up Step 2 from the nearest neighbor heuristic into three new steps 2*a*, 2*b*, and 2*c*. In Step 2*a*, we determine the most similar adjacent product compared to the first product in the current path. Conversely, we find the most similar adjacent product w.r.t. the last product in Step 2*b*. In Step 2*c*, we select one of the recommended products which has the minimum of differences to either the first or last product of the current path, i.e., product order. We also update Step 3 such that we add the next product to the begin or end of the path, respectively. In the end, the core may not be the first product to be analyzed as we add products to both ends of the path. While the improved nearest neighbor heuristic provides a better approximation of the optimal solution, it does not prevent completely from the addition of dissimilar products in the last steps of the heuristic.

To cope with this issue, products have to be added to the current path not only at the begin or end, but rather at any position in the path. Thus, we apply insertion heuristics [29] to compute a good approximation w.r.t. the optimal product order. Based on an initial tour comprising at least two products, we always select and insert the next product based on a specific criterion, e.g., the product with minimal distances to another product from the current tour. Hence, we again extend our product graph with a dummy node and respective zero-weighted edges to facilitate the adoption of

those insertion heuristics to find an approximately optimal product order. For both greedy algorithms, the extension was not necessary as we directly computed a path through the product graph. In this paper, we apply the (1) **near**est **in**sertion (NEARIN), and (2) **far**thest **in**sertion (FARIN), as those heuristics provide good results when solving a given TSP further enhanceable by applying improvement heuristics such as k-opt heuristics [29].

Both heuristics differ in the way the next product to be inserted is selected. For NEARIN, we always select the product which has the minimal distance by means of differences to any product already contained in the current tour. In contrast, for FARIN, we select the product which has the maximal distance. Afterwards, we determine for the selected product in both heuristics equally the best fitting position in the current tour. Thus, we examine between which products in the tour the insertion would result in the minimal increase of the overall number of differences. For the insertion, we remove the old connecting edge between those products and add the selected product to the tour by also adding the connecting edges. As we focus on a path instead of a tour, we remove the dummy node as last step to obtain a product order as result. Again, the core may not be first product of the computed optimized order. In summary, to find a product order with the NEARIN/FARIN heuristic [29], we apply the following steps, where the complexity is $O(n^2)$ in the number of product nodes:

1. Build an initial tour with p_{core} and the dummy node
2. Find the edge with the minimal (NEARIN)/maximal (FARIN) distance w.r.t. a product in the tour
3. Find the position within the tour for the cheapest insertion of the target product
4. Remove obsolete edges from the current tour and reconnect tour with the selected product
5. Repeat Step 2 to 4 until all products are inserted
6. Remove the dummy node to obtain a path

After obtaining an optimized order, we are able to exploit the difference reduction when applying incremental analysis techniques, e.g., incremental model slicing as change impact analysis for SPL regression testing [22]. Due to the symmetry of the product graph, we can either apply the analysis techniques from the begin or the end of the path as starting from both sides results in the same number of overall differences. Furthermore, although we start the computation from a fixed product node, i.e., the core p_{core}, the optimized order may specify another product to be analyzed first.

EXAMPLE 3. *To find an optimal product order in the graph from Ex. 2 shown in Fig. 2(a), we extend it with a dummy node D. By solving the respective TSP, we obtain the optimized order* p_2, p_4, p_5, p_3, *and* p_1 *with* 171 *as overall number of differences. For the look up greedy heuristic, the order is given as* p_1, p_5, p_4, p_3, *and* p_2 *with* 201 *differences. The NEARIN heuristic find the same order, but solely reversed. In this example, we obtain the same product order w.r.t. the optimal solution by applying the FARIN heuristic.*

Open Issues for Optimized Product Orders. For our presented technique, some open issues exist. First, the completeness property of a product graph can be violated based on potential *restrictions*. A restriction denotes the scenario that we are not able to step from one product to a certain one due to engineering decisions or domain knowledge. For example, the hardware change over time would be too expensive. For a restriction, the respective edge does not exist in the product graph. The reduction of the set of edges and, thus, the reduction of potential paths within the graph will influence the results of the adopted heuristics. The investigation of restrictions and their impacts is left to future work. Second, we start all heuristics with an initial product denoting the first product of the order which may change during the application of the heuristics. In contrast, when we consider for instance the Linux kernel, there is a fixed starting point of an order for applying, e.g, SPL testing, namely the *allyesconfig* [3, 12]. To cope with such requirements and, therefore, to handle real-world SPLs, we have to examine our technique and provide an extension in the future. Third, we currently interpret additions and removals encapsulated in regression deltas equally. By incorporating additions and removals differently, the product graph will comprise directed edges and, thus, we have to instantiate an asymmetric TSP [29] to allow for the computation of optimal product orders.

4. EVALUATION

In this section, we describe the evaluation of graph-based product ordering. For the evaluation, we derive three research questions to be answered:

RQ1 Does graph-based product ordering *reduce* the overall number of differences between products?

RQ2 How good is the *approximation* with the proposed heuristics compared to finding the optimal solution?

RQ3 How good is the *performance* of the heuristics compared to finding the optimal solution?

The research questions are suitable, as we (1) investigate by RQ1 whether product orders with a differing number of overall differences exist, (2) validate by RQ2 whether the results obtained by the heuristics are satisfactory approximations, and (3) check by RQ3 whether the heuristics provide their solution faster than the TSP solver.

Prototype. We prototypically implemented our approach as ECLIPSE[1] plug-ins facilitating (1) its extendability, e.g., by means of further heuristics, and (2) a lightweight integration in existing frameworks, e.g., for incremental SPL analysis [22]. We use the ECLIPSE modeling framework[2] for meta modeling to capture the product graph and resulting product orders. Based on a given delta model and a product set, we derive a respective product graph. The graph is then used as input for a given heuristic. The resulting order is stored to be used in incremental SPL analysis tools.

Subject Systems and Artificial Data. For the evaluation, we apply our approach to two types of data. First, we focus on four existing SPLs, namely *Wiper*, *Mine Pump*, *Body Comfort System*, and *Vending Machine*, which already served as benchmarks in the literature [7, 21]. Wiper [7] comprises variable qualities of rain sensors and wipers resulting in 8 products. Mine Pump [7] defines 16 products realizing a pump control system with variable water level handling and an optional methane detection facility. Body Comfort System (BCS) [21] describes a comfort system of a car comprising optional components as for instance a remote control key, an alarm system, or a central locking system. We already used BCS in prior work for evaluation purposes [23], where we applied our techniques to a pairwise

[1] https://eclipse.org/

[2] https://eclipse.org/modeling/emf/

Table 1: Overall Differences Resulting from Product Order Optimization for Existing Software Product Lines

SPL ($\|P_{SPL}\|$)	Standard	Random (∅)	Optimal	Greedy	Look Up	NEARIN	FARIN
Wiper (8)	43	61	25	25	25	25	27
Mine Pump (16)	164	328	84	84	84	84	91
Body Comfort System (18)	1760	1520	518	607	568	592	596
Vending Machine (28)	194	327	80	80	80	80	84

product sample comprising 18 products [26, 21]. In this paper, we also rely on this pairwise sample for our evaluation. Vending Machine [7] describes a variable selection of various beverages. We chose those systems as we used them already in previous work on incremental SPL testing [22], where we identified the influence of product orders in our evaluation.

Second, we use artificial data to obtain more significant results as the four SPLs are rather small compared to real-world SPLs. Therefore, we generate different data sets using Ruby which are varying in their sizes w.r.t. the number of products they should contain. For the data sizes, we consider the number of 100, 500, 1,000, 5,000, and 10,000 products, where for each size 50 different data sets are created. Those data sizes are sufficient for the evaluation compared to real-world SPLs as for instance a version of the Linux kernel with about 6,000 features has a pairwise feature sampling comprising only 480 products [16]. For the generation of one data set, we first determine the maximal number of elements, products can comprise. As the number of elements or lines of codes often increases with the number of products by a specific percentage factor [19], we apply a factor of 15 to the number of products to specify the maximal number. On this basis, we compute for each product of a data set a binary element vector randomly comprising a 1 if an element is contained or a 0, otherwise. To determine the number of differences between two products used for the edge weights, we apply the hamming distance. In the end, each data set is stored in a distance matrix transformable into a product graph to be used as input in our prototype.

Experimental Setting. We compare the results of the heuristics with 100 *random* orders and an *optimal* solution determined by a TSP solver. In addition, for the existing SPLs, we use the standard order in which configurations are derived by FeatureIDE [2] (i.e., by exploiting the hierarchical structure of the feature model). As TSP solver, we apply Concorde[3] with QSopt LP Solver[4] as underlying linear programming solver required by Concorde to solve a given TSP. For the TSP solver and the heuristics, we apply a timeout of $12h$ to restrict the computation time. In case of a timeout, we use the last optimal solution the solver found. This solution can already be the optimal solution as the solver may take the remaining time to verify that there is no other option. We performed our evaluation on a machine with an Intel Xeon $E5-2687W$ processor with 3.1Ghz and provide 20GB for the Java VM.

We use the Mann-Whitney U Test in R v3.2.3[5] to explore whether differences between each heuristic and random as well as optimal orders are significant. The Mann-Whitney U Test is a non-parametric statistical test used to compare differences between two independent samples. From this test, we obtain a value representing the probability that both samples are equal, called p-value. If a p-value is lower than 0.05, we assume the difference is significant.

Table 2: Results of Significant Measurements with p-Values for $|P_{SPL}| = 10,000$ Artificial Data Sets

	Greedy	Look Up	NEARIN	FARIN
Random	$8.77e^{-09}$	$5.36e^{-09}$	$6.88e^{-14}$	$2.26e^{-14}$
Optimal	$1.26e^{-11}$	$1.60e^{-11}$	$5.82e^{-06}$	$2.81e^{-05}$

Results for Existing SPLs. We summarize the results of finding an optimized product order for the existing SPLs in Tab. 1. As we can see, there is a distinction between the standard and the random order. Except for BCS, standard orders (i.e., orders derived by FeatureIDE) are on average 40.1% better than the random orders. For BCS, the standard order was derived by pairwise sampling [26] and is 13.6% worse than the random orders. This is in line with previous work [3], which showed that sampling algorithms typically choose most different products first.

By finding the optimal product order, we can achieve a reduction of 55.0% to the standard and 68.7% to the random orders on average. Furthermore, we can also observe that there is a distinction between the optimal solution and the heuristics. As Wiper, Mine Pump and Vending Machine [7] are rather small (i.e., few and small products), we achieve equal or very similar results compared to the optimal solution with all heuristics. While the pairwise sample of BCS [21] also contains only few products, the products are larger and, thus, product regression deltas are larger, too. This seems to be the cause that we obtain distinct results for the different heuristics which are on average 12.3% worse than the optimal solution, but 61.2% better than random orders.

The results indicate that graph-based product ordering can indeed reduce the overall number of differences (RQ1). For each subject system, the optimal solution as well as all four heuristics were better than random orders. Also, the heuristics lead to similar or even identical overall differences than the optimal solution (RQ2). Optimal solutions and approximations were each computed within 700 ms. However, as the existing subject systems are too few to apply significance tests and to generalize our results, we also considered artificial data.

Results for Artificial Data. We applied all heuristics to each of the 50 artificial data sets for each SPL size to obtain more reliable evaluation results. We compared all heuristics with random orders to answer RQ1 and with optimal solutions to answer RQ2. In Tab. 2, we show the p-values for the data sets containing 10,000 products. The

[3]http://www.math.uwaterloo.ca/tsp/concorde.html
[4]http://www.math.uwaterloo.ca/ bico/qsopt/
[5]http://www.r-project.org/

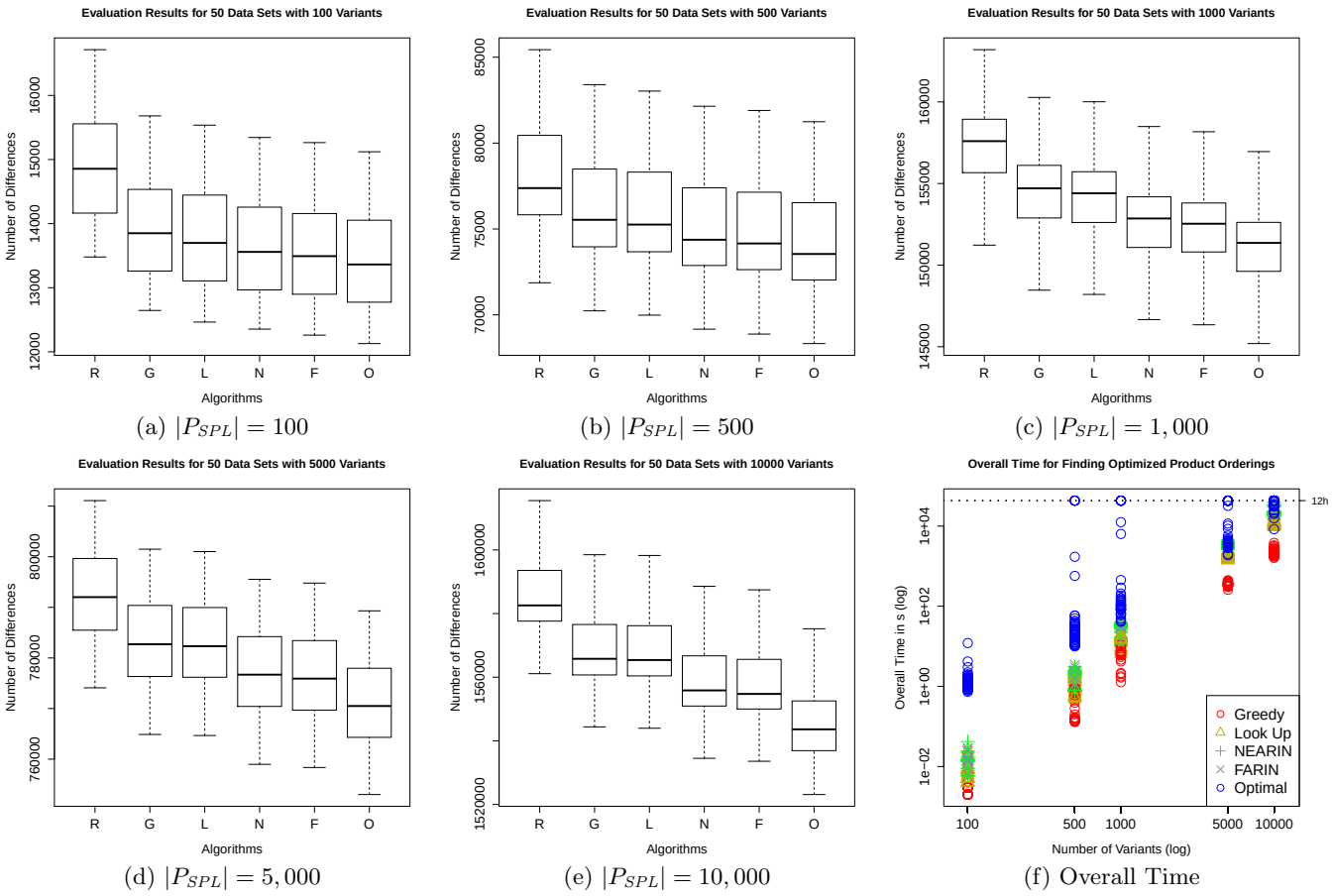


Figure 3: Results of Product Order Optimization for Different Artificial Data Sets and Algorithms (R = Random, G = Greedy, L = Look Up, N = NEARIN, F = FARIN, O = Optimal)

values indicate that all heuristics reduce the overall differences compared to random orders. Similarly, for all other data sets (i.e., 100, 500, 1,000, 5,000 products) the p-values were smaller than 0.004. Hence, we can confirm our results with the existing SPLs that graph-based product ordering reduces the overall differences (RQ1).

In contrast, the differences are significant for most, but not for all data sets when comparing the heuristics with the optimal solutions. In particular, the optimal solution is not significantly better than NEARIN ($p = 0.087$ and $p = 0.145$) and FARIN ($p = 0.292$ and $p = 0.260$) for 100 and 500 products. That is, NEARIN and FARIN are good approximations for few products. To assess the goodness of all heuristics, we present the data as boxplots in Fig. 3(a) to Fig. 3(e). In all 5 boxplots, the values of Greedy and Look Up as well as the values of NEARIN and FARIN are similar. For $10,000$ products, Random orders, Greedy, Look Up, NEARIN, and FARIN lead to 0.045 %, 0.034 %, 0.034 %, 0.028 %, and 0.027 % more differences than the Concorde TSP solver, respectively.

In the larger data sets, the situation may occur that for the selection of the next product to be added to the order several candidates exist which potentially influence the complete result differently. To cope with this scenario, we want to apply improvement heuristics such as 2-opt heuristics [29] in the future to further enhance our obtained approximations.

To answer RQ3, we investigate the performance of the algorithms applied to the artificial data sets. In Fig. 3(f), the computation time of the heuristics and the TSP solver is shown in logarithmic scale on both axes. The data suggests that heuristics with better results also consume more time. In other words, Greedy is the fastest heuristic, followed by Look Up, NEARIN, and FARIN. Furthermore, we can see that with an increasing number of products, all algorithms require an increasing computation time. Already for 500 products, the TSP solver reached the timeout after 12 h, whereas all heuristics finished within 10 s. For 10,000 products, the TSP solver was interrupted in 38 out of 50 measurements. In summary, all adopted heuristics achieve a good performance compared to the TSP solver. However, the better run-time comes with the trade-off in terms of an approximated solution.

Threats to Validity. We discuss some threats to validity concerning our approach and its evaluation. First, the existing SPLs are rather small and may not reflect real-world SPLs. However, they are already used as benchmarks in the literature [7, 21, 22] and, therefore, are exploitable to perform first experiments. In addition, we used artificial data which is also a possible threat. To cope with this threat, the generation of our data sets ensures that the characteristics of the existing SPLs are fulfilled to simulate valid SPLs. The current timeout of $12h$ for the TSP solver is also a potential

threat as it influences our results. Besides the timeout, the choice of the distance measurement may be a threat. We applied the number of change operations captured in regression deltas as edge weights, but other metrics should also be applied to investigate their impacts on optimized product orders. Therefore, we will perform a more elaborated evaluation in the future to confirm our positive results taking those potential threats into account. In this evaluation, we further investigate the real effect of optimized product orders on incremental SPL analyses as we currently abstract from a concrete analysis approach solely focusing on the product ordering problem. The memory required for storing the graph during order computation is also a potential threat to our approach. We will improve our implementation to reduce the required memory in the future by exploiting more suitable data structures.

5. RELATED WORK

We discuss related work w.r.t. the prioritization of products for SPL analysis as well as SPL testing. In the literature, several approaches for product prioritization exists (1) to increase feature interaction coverage [3, 15, 11, 25], (2) incorporating domain knowledge [10, 17, 13], or (3) by applying multi-objective optimization [5, 30, 15, 27].

Al-Hajjaji et al. [3] propose an approach to prioritize products based on the dissimilarity of feature configurations. They extend their work to facilitate the generation and dissimilarity-based prioritization of pairwise samples of products [1]. Lopez-Herrejon et al. [25] also propose a technique for prioritizing pairwise samples of products by applying evolutionary algorithms. Henard et al. [15] exploit the product dissimilarity in a search-based approach to sample and prioritize products further incorporating the minimization of the test-suite size and the costs. Devroey et al. [11] employ the dissimilarity between products and their behavior to select test cases and products in a certain order. In contrast to our approach, those techniques focus on the dissimilarity between products to prioritize, whereas our approach determine product orders exploiting the product similarity w.r.t. solution-space information, i.e., regression deltas.

Devroey et al. [10] prioritize products incorporating feature models, feature transition systems representing the SPL behavior, and usage models capturing the probability of executing relevant behavior. The product with a higher execution probability has a higher priority to be tested first. Johansen et al. [17] combine sampling and prioritization by weighting products based on domain information extracted from the market. Ensan et al. [13] prioritize products to be tested first comprising the most desirable features specified by the domain experts. However, domain knowledge is often not sufficiently available. Hence, we use differences between products by means of regression deltas to order products.

Parejo et al. [27] apply search-based techniques to prioritize products based on functional, e.g., product dissimilarity, and nonfunctional properties, e.g., feature size. Sánchez et al. [30] propose the comparison of several criteria w.r.t. the feature selections to order products. They show that considering different orders of products can affect the fault detection rates. Baller et al. [5] present multi-objective test suite optimization for incremental SPL testing by incorporating costs and profits of test artifacts, where they obtain as a side-effect an order of products the optimized test suite is to be applied. Our approach only focuses on the product similarity to find an optimized order. However, optimizing the product order further incorporating other criteria may enhance our results to be investigated in future work.

In addition to product prioritization, Lachmann et al. [18] propose a delta-oriented test case prioritization approach for incremental SPL integration testing. Based on architectural and state machine deltas, reusable test cases represented as message sequence charts are ordered to increase an early coverage of changes to be retested, where further the test case dissimilarity is exploited. In contrast, our approach focuses on product prioritization. However, combining the prioritization on problem as well as solution-space level may enhance incremental SPL testing.

6. CONCLUSION

In this paper, we proposed the computation of optimized product orders, where the overall number of differences between all products gets minimized. Therefore, we first define a product graph capturing products as nodes and solution-space information in terms of differences between two products as weights w.r.t. the number of change operations encapsulated in a corresponding regression delta. Based on this graph, we instantiate a TSP to find an optimal order, where we adopt various heuristics. We prototypically implemented our approach and evaluate its applicability and performance by means of four existing SPLs as well as artificial data showing positive results.

As future work, we want to improve our approach by optimizing the incremental heuristics, e.g., based on k-ops heuristics [29]. We further improve our prototype by enhancing the underlying data structures resulting in a reduction of the computation time of the heuristics. We plan to perform more elaborated experiments with real-world systems to validate our positive results and also to evaluate the influence of the computation effort as well as the improvement of incremental SPL analyses based on the computed optimized product orders. A comparison to product prioritization techniques [1, 3, 10, 15, 11, 27] would be reasonable in this context. We will also investigate the three open issues to further enhance our presented approach. Furthermore, we want to abstract from the requirement of linear product orders such that we are able to apply minimal spanning trees to find optimal solutions in linear complexity.

7. ACKNOWLEDGMENTS

This work was partially supported by the German Research Foundation under the Priority Programme SPP 1593: Design For Future – Managed Software Evolution. We thank Frank-Michael Quedenfeld and Phillip Keldenich for the discussions about graph algorithms and their applications.

8. REFERENCES

[1] M. Al-Hajjaji, S. Krieter, T. Thüm, M. Lochau, and G. Saake. IncLing: Efficient Product-Line Testing Using Incremental Pairwise Sampling. In *GPCE'16*, pages 144–155. ACM, 2016.

[2] M. Al-Hajjaji, J. Meinicke, S. Krieter, R. Schröter, T. Thüm, T. Leich, and G. Saake. Tool Demo: Testing Configurable Systems with FeatureIDE. In *GPCE'16*, pages 173–177. ACM, 2016.

[3] M. Al-Hajjaji, T. Thüm, M. Lochau, J. Meinicke, and G. Saake. Effective Product-Line Testing Using

Similarity-Based Product Prioritization. *Software and System Modeling*, 2016. (To appear).

[4] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton university press, 2011.

[5] H. Baller, S. Lity, M. Lochau, and I. Schaefer. Multi-Objective Test Suite Optimization for Incremental Product Family Testing. In *ICST'14*, pages 303–312, 2014.

[6] D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract Delta Modelling. *Math. Struct. Comp. Sci.*, 25(3):482–527, 2015.

[7] A. Classen. Modelling with FTS: A Collection of Illustrative Examples. Technical Report P-CS-TR SPLMC-00000001, PReCISE Research Center, Univ. of Namur, 2010.

[8] W. J. Cook. *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*. Princeton University Press, 2014.

[9] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. Towards an Incremental Automata-Based Approach for Software Product-Line Model Checking. In *SPLC'12*, pages 74–81. ACM, 2012.

[10] X. Devroey, G. Perrouin, M. Cordy, P.-Y. Schobbens, A. Legay, and P. Heymans. Towards Statistical Prioritization for Software Product Lines Testing. In *VaMoS'14*, pages 10:1–10:7. ACM, 2014.

[11] X. Devroey, G. Perrouin, A. Legay, P.-Y. Schobbens, and P. Heymans. Search-Based Similarity-Driven Behavioural SPL Testing. In *VaMoS'16*, pages 89–96. ACM, 2016.

[12] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. Understanding Linux Feature Distribution. In *MISS'12*, pages 15–20. ACM, 2012.

[13] A. Ensan, E. Bagheri, M. Asadi, D. Gasevic, and Y. Biletskiy. Goal-Oriented Test Case Selection and Prioritization for Product Line Feature Models. In *ITNG'11*, pages 291–298. IEEE, 2011.

[14] J. Greenyer, C. Brenner, M. Cordy, P. Heymans, and E. Gressi. Incrementally Synthesizing Controllers from Scenario-Based Product Line Specifications. In *ESEC/FSE'13*, pages 433–443. ACM, 2013.

[15] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Le Traon. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *IEEE Trans. Soft. Eng.*, 40(7):650–670, 2014.

[16] M. F. Johansen, O. Haugen, and F. Fleurey. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *SPLC '12*, pages 46–55. ACM, 2012.

[17] M. F. Johansen, Ø. Haugen, F. Fleurey, A. G. Eldegard, and T. Syversen. Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines. In *MODELS'12*, pages 269–284. Springer, 2012.

[18] R. Lachmann, S. Lity, F. E. Fürchtegott, M. Al-Hajjaji, and I. Schaefer. Fine-Grained Test Case Prioritization for Integration Testing of Delta-Oriented Software Product Lines. In *FOSD'16*, pages 1–10. ACM, 2016.

[19] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *ICSE'10*, pages 105–114. ACM, 2010.

[20] S. Lity, H. Baller, and I. Schaefer. Towards Incremental Model Slicing for Delta-Oriented Software Product Lines. In *SANER'15*, pages 530–534, 2015.

[21] S. Lity, R. Lachmann, M. Lochau, and I. Schaefer. Delta-Oriented Software Product Line Test Models - The Body Comfort System Case Study. Technical Report 2012-07, Technische Universität Braunschweig, 2012.

[22] S. Lity, T. Morbach, T. Thüm, and I. Schaefer. Applying Incremental Model Slicing to Product-Line Regression Testing. In *ICSR'16*, pages 3–19. Springer, 2016.

[23] M. Lochau, S. Lity, R. Lachmann, I. Schaefer, and U. Goltz. Delta-Oriented Model-Based Integration Testing of Large-Scale Systems. *J. Sys. and Soft.*, 91:63–84, 2014.

[24] M. Lochau, S. Mennicke, H. Baller, and L. Ribbeck. Incremental Model Checking of Delta-Oriented Software Product Lines . *J. Logical and Algebraic Methods in Programming*, 85(1, Part 2):245 – 267, 2016. Formal Methods for Software Product Line Engineering.

[25] R. E. Lopez-Herrejon, J. Javier Ferrer, F. Chicano, E. N. Haslinger, A. Egyed, and E. Alba. A Parallel Evolutionary Algorithm for Prioritized Pairwise Testing of Software Product Lines. In *GECCO'14*, pages 1255–1262, 2014.

[26] S. Oster, M. Zink, M. Lochau, and M. Grechanik. Pairwise Feature-interaction Testing for SPLs: Potentials and Limitations. In *SPLC'11*, pages 6:1–6:8. ACM, 2011.

[27] J. A. Parejo, A. B. Sánchez, S. Segura, A. Ruiz-Cortés, R. E. Lopez-Herrejon, and A. Egyed. Multi-Objective Test Case Prioritization in Highly Configurable Systems: A Case Study. *J. Sys. and Soft.*, 122:287–310, 2016.

[28] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[29] G. Reinelt. *The Traveling Salesman - Computational Solutions for TSP Applications*, volume 840 of *LNCS*. Springer, 1994.

[30] A. B. Sánchez, S. Segura, and A. Ruiz-Cortés. A Comparison of Test Case Prioritization Criteria for Software Product Lines. In *ICST'14*, pages 41–50. IEEE, 2014.

[31] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, and K. Villela. Software Diversity: State of the Art and Perspectives. *Int. J. Softw. Tools Technol. Transf.*, 14(5):477–495, 2012.

[32] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *Computing Surveys*, 47(1):6:1–6:45, 2014.