Echoes from Space: Grouping Commands with Large-Scale Telemetry Data

Alexander Lattas
Department of Computing
Imperial College London
London, United Kingdom
alexandros.lattas17@imperial.ac.uk

Diomidis Spinellis
Department of Management Science and Technology
Athens University of Economics and Business
Athens, Greece
dds@aueb.gr

ABSTRACT

Background: As evolving desktop applications continuously accrue new features and grow more complex with denser user interfaces and deeply-nested commands, it becomes inefficient to use simple heuristic processes for grouping GUI commands in multi-level menus. Existing search-based software engineering studies on user performance prediction and command grouping optimization lack evidence-based answers on choosing a systematic grouping method.

Research Questions: We investigate the scope of command grouping optimization methods to reduce a user's average task completion time and improve their relative performance, as well as the benefit of using detailed interaction logs compared to sampling.

Method: We introduce seven grouping methods and compare their performance based on extensive telemetry data, collected from program runs of a CAD application.

Results: We find that methods using global frequencies, user-specific frequencies, deterministic and stochastic optimization, and clustering perform the best.

Conclusions: We reduce the average user task completion time by more than 17%, by running a Knapsack Problem algorithm on clustered users, training only on a small sample of the available data. We show that with most methods using just a 1% sample of the data is enough to obtain nearly the same results as those obtained from all the data. Additionally, we map the methods to specific problems and applications where they would perform better. Overall, we provide a guide on how practitioners can use search-based software engineering techniques when grouping commands in menus and interfaces, to maximize users' task execution efficiency.

CCS CONCEPTS

• **Human-centered computing** → **Interaction design process and methods**; • **Software and its engineering** → **Software evolution**; *Search-based software engineering*;

KEYWORDS

Command grouping, menu layout, GUI optimization, telemetry, sampling

ACM Reference Format:
Alexander Lattas and Diomidis Spinellis. 2018. Echoes from Space: Grouping Commands with Large-Scale Telemetry Data. In *ICSE-SEIP '18: 40th International Conference on Software Engineering: Software Engineering in Practice Track, May 27-June 3 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3183519.3183545

1 INTRODUCTION

Computer applications aimed at professional users are deemed to be at least as complex as the problem they aim to solve. Computer aided design applications, image and video editors, simulators and enterprise resource planners are just a few examples of programs whose interfaces have become unyieldingly complex. As developers struggle to publish frequent updates that introduce more and more integrated commands, tools and extensions, their graphical user interfaces (GUIs) become packed with icons impossible to memorize. Moreover, new, modern but niche features are placed at the center of a user's attention in order to justify the increasing costs of an update, while well-known and frequently used commands get buried in multiple nested hierarchical menus.

In the meantime, parallel efforts to improve the user's experience often are in vain, as they focus on the aesthetic aspect, or derive conclusions based on heuristics and small-scale experiments. Common user experience (UX) experiments, involving heuristics-based testing tools focus on specific scenarios that the developers think important. However, lacking a user-centric approach, the majority of the users are likely to face mental overhead and require more time when executing common tasks that involve deep-nested commands. Moreover, new users that are introduced to such complex applications will need much time to become comfortable in using them resulting, for example, in longer profitless training sessions.

We propose, evaluate and compare seven methods that exploit easily accessible program telemetry data to reorganize an application's command tree structure based on actual evidence. These methods involve a combination of command frequencies, domain-based heuristics, continuous training, and stochastic optimization. To train the algorithms, as well as to evaluate them, we use a large data set of telemetry data, created by real users of a fairly complex professional application. Additionally, we use experimental data we produced, to understand the data set and to eliminate the noise from the data.

The application studied is a CAD suite for architects and civil engineers. The architectural design functionality (TEKTON) supports

ICSE-SEIP '18, May 27-June 3 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5659-6/18/05…$15.00
https://doi.org/10.1145/3183519.3183545

2D and 3D modeling, as well as photo-realistic rendering. On the civil engineering front (FESPA) the system supports the analysis and design of steel, concrete, and masonry structures, and also the evaluation, strenghtening and repair of existing buildings under a variety of structural design standards, such as the Eurocodes and the corresponding national annexes.

Figure 1 is a screen dump depicting the application's salient characteristics and features. The top two windows show an architectural drawing floor plan and its photo-realistic rendering, while the bottom two windows show a civil engineering wood mould plan and the corresponding beam and column model, which is used for finite element analysis. The user interaction is based around entities, such as walls, windows, beam, slabs, columns, text, hatches, and stairs. Each entity has associated parameters (e.g. dimensions and material) and commands (e.g. add new, delete, extend, change properties). The current version of the application supports 13 general-purpose entities (e.g. spline or cross-section), 15 entities supporting architects (e.g. balustrade or roof), and 18 entities supporting civil engineers (e.g. column or footing). A few so-called entities relate to groupings of related commands and properties, without being associated with concrete elements appearing on a plan. Examples of these are the groupings of commands used for rendering and for global manipulations. In total, 48 entities are associated with 627 commands and 3735 properties.

The large number of available commands and properties is managed by having users interact with the application by first selecting the entity they want to manipulate. The corresponding entity icons appear in the top toolbars of Figure 1. Once an entity is selected, a toolbar with the commands associated with it appears on the left, while a separate dialog (not shown) provides access to the corresponding properties. For example, the toolbar on the left side of the window shown in Figure 1 contains the commands associated with the beam entity.

While the organization of commands and properties around entities provides a way to navigate through their large number, it also imposes a switching overhead. For instance, an architect wishing to design a house, might first employ the grid entity to draw the lines along which the house's elements will be aligned, then switch between the wall and the opening entity to add walls and windows, and then switch to the roof entity to add a tiled covering. Further adjustments to the drawing's elements (e.g. to change a window's size), would have the architect switch again to the corresponding entity.

Recently, staff dealing with the application's user experience (UX) asked us to explore alternative command arrangements that might enhance user productivity by reducing the cost of entity switching. The main idea was to make some commonly-used commands always accessible on screen. The design of a new arrangement proved to be controversial. Proposals based on the intuitive understanding of users' interactions were criticized as lacking empirical backing. On the other hand, proposed arrangements based on command frequency counts were considered unrealistic, because they ignored the sequence in which commands were issued.

Thankfully, in order to help debugging and to improve the users' experience, recent application versions can log the commands a user issues in a centralized database. It was obvious that these

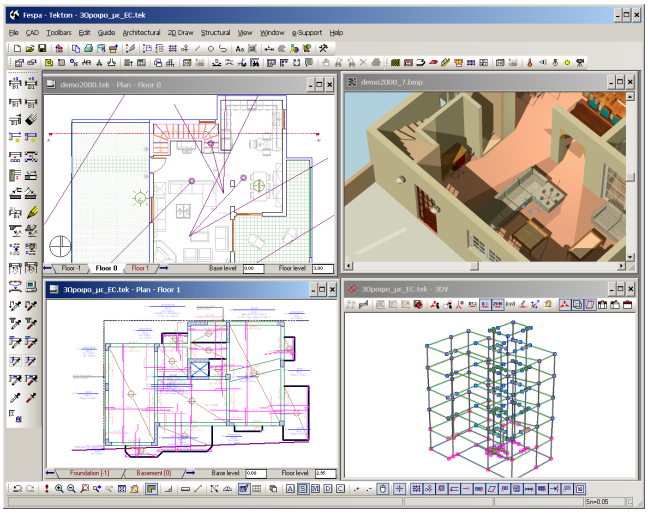

Figure 1: The CAD application in action.

data could be used to create realistic command grouping optimization proposals, and also to evaluate the performance improvement associated with these proposals.

Within this context we sought to answer the following research questions.

(1) What is the scope for increasing the CAD's user productivity through the optimization of command grouping?
(2) What is the relative performance of diverse command grouping optimization methods in terms of user productivity?
(3) What, if any, is the benefit of using comprehensive and detailed interaction logs, instead of sampling a few users or obtaining simple command invocation frequencies in terms of achievable command grouping performance optimization?

The two main contributions of this study are 1) the evaluation of seven command grouping optimization methods based on detailed actual interaction data, and, 2) results regarding the effect of data sampling as part of the evaluation. Our findings can guide software developers and UX designers on how to use telemetry data to optimize their applications. We therefore devise specific design suggestions for CAD applications that can be directly applied to improve user performance.

We begin this study by analyzing the preexisting quantitative and empirical approaches to command grouping optimization, as well as the relevant literature on which we are building upon (Section 2). Then, in Section 3, we outline how we obtained the data we used and describe the command optimization methods we propose in terms of their purpose and the algorithms used for training and evaluation. In Section 4 we discuss our results, comparing the methods and mapping them to specific use cases, while also offering advice regarding sampling and interface design techniques. Section 5 concludes the paper with an overview of our findings and pointers to future work.

2 COMMAND GROUPING OPTIMIZATION

With *command grouping* optimization, we aim to find evidence-based answers on choosing a systematic method for grouping multiple commands into multi-level menus. The study belongs to the general area of *search-based software engineering* [11], because it utilizes search-based optimization algorithms to guide software engineering decisions.

The usability of an application depends on the effectiveness of the users' actions, their efficiency and the satisfaction users receive upon the completion of a task [8]. Usability inspection is the method of evaluating interfaces and can be carried by automatically computed measures, involving real users, models and formulas or rules of thumb. A combination of these has been found to be the most efficient [21].

The usability analysis method used in this study falls into the category of action analysis, or keystroke-level analysis [14], which involves monitoring user's action sequences and timing. The Keystroke-Level Model (KLM) was introduced in 1980 [2], proposing that the time T_X needed to execute a task consists of the task acquisition time and the task execution time. As shown in Equation 1, this model takes six variables into account: the time, K, required for pressing a key or a button; the time, P, required to point on a displayed item with a mouse; the time, H, required to move hands between the mouse and the keyboard; the time, D, required to draw (the task involved drawing straight lines); the time, M, required for mental preparation, and the time, R, required by the system to respond.

$$T_X = T_K + T_P + T_H + T_M + T_R \tag{1}$$

A later study [15] found that keystroke entry (K) is relatively fast compared to moving hands between keyboard and mouse (H) and very fast compared to the mental preparation (M), visual searches and mouse moves (P). A study on AutoCAD [3] calculated that $P = 1.10s$, $M = 1.35s$ and $0.08 \leq K \leq 1.20$, as it depends a lot on the user's expertise level. Additionally, it argued that task completion time consisted of the task acquisition time, the task execution time and the error time as well. Moreover, the study's authors found that as the model becomes more fine its predictive power weakens, because it becomes more prone to errors.

There are numerous other studies in the literature using KLM. A study that showcased a KLM for mobile phone applications [13] highlighted the important difference in time between novices and professionals. Another study on mobile phones, in common with this research, deviated from the classical KLM by using the K and M factors to measure task of keyboard typing [20]. A third one [17] found that on handheld devices, KLM can predict task execution time with less than 8% error.

Many studies find a trade-off between M and K. Lane et al. [16] argued that when dealing with several menu choices for a task, a user goes through a single rather than a multiple mental preparation task (M). Its time varies by experience and by making menus deeper and shorter or shallower and wider. However, researchers agree that breadth is preferable to depth [1, 18, 19].

To address the menu depth issue we described, Microsoft introduced in its Office products the Ribbon User Interface, a tabbed toolbar menu that promotes the work efficiency of document readers and creators [5]. Also, as a more radical approach, experts suggest CommandMaps [22], an interface technique that flattens command hierarchies to exploit human spatial memory. It can achieve greater performance, because spatially remembering a task is easier than remembering its category[22]. Additionally, a similar approach to CommandMaps, called Ephemeral Adaptation, lists all commands and highlights those that are predicted as important [6].

Additionally, a more generalized model for multilevel menus was suggested by Cockburn et al. [4]. This calculates the decision, search (M) and pointing time (P), as well as the steering cost (sc) of navigating between menu layers as shown in Equation 2. For a total number of $\mathbb{L}$ hierarchical menus, T_M, T_P and sc are summed for each level of the menus and only T_M and T_P for the final one. With this model, they found that efficiency degrades as menus get lengthier, linearly for novices and logarithmically for experts.

$$T_X = \sum_{j=1}^{l-1,\, l\in\mathbb{L}} (T_{M_j} + T_{P_j} + sc_j) + T_{M_l} + T_{P_l} \tag{2}$$

Addressing the menu optimization problem from a different perspective, numerous research studies employ stochastic algorithms to find the optimal menu structure. Matsui and Yamada introduced a genetic algorithm [18, 19] and a simulated annealing algorithm [19] that minimized average selection time of menu items by considering movement (P) and decision time (M). A different approach by Troiano et al. [23, 24] optimized menu structures by applying genetic algorithms on user accessibility and preference and suggested similar approaches for the optimization of other GUI aspects [23].

This study examines a broad command optimization toolset using actual data. In contrast to this research, the majority of the aforementioned studies used experiments where novice or professional users were given prespecified tasks to complete. We build a cost function based on the Keystroke-Level Model, which is used to assess several GUI optimization techniques, that have or have not been connected with KLM before. These cover stochastic optimization, spatially wide options, such as CommandMaps, and user-optimized solution, such as the Microsoft Ribbon, but also add naive optimization on frequency, continuously adjusted user optimization, clustering on user types and heuristic groupings. In the end, these provide an insight into which menu optimization methods can be the most effective, as well as which experimental techniques provide the best results.

3 MODELS AND EVALUATION

Our study involved obtaining command usage patterns from telemetry data, performing a controlled experiment to obtain the users' navigation overhead, and examining seven command grouping optimization approaches.

3.1 Telemetry Data

We created and evaluated models of user interactions based on a sample of telemetry data associated with actual program executions. These data contain the name of each executed command, which provides an accurate picture regarding the precise order of command invocations. We sampled program runs over three years, selected a 14 GB subset comprising 182 thousand anonymized program runs

that contained sequences of 32 million command invocations. During that period, the GUI changes were limited and we can safely assume these do not affect the cohesion of our data. To protect the users' privacy, these were used in an anonymized format that included the command initiation time, and opaque identifiers of the session, the user, the entity, the command, and the command's class (e.g. move or delete). From these we extracted a number of program runs and their executed commands, as well as the required execution time.

Table 1: Data Overview

	Telemetry	Experiment
Users	1,784	11
Sessions	182,273	11
Commands executed	32,514,217	534
Distinct commands	627	35
Distinct entities	44	16
Commands per session (avg)	178	48
Session execution time (avg; s)	6213	342
Command execution time (avg; s)	34.8	9.0

Each one of the 627 commands was associated with one of the program's 48 entities. This allowed us to identify all points in the command sequences where a different entity would have to be selected under the current command grouping scheme. Moreover, each one of 162 commands was associated with one of seven groups of commands with common functionality. We used this association to investigate functional command grouping as an optimization option. A description of the data is shown on Table 1.

The series of executed commands were used to obtain command use scenarios, which we then used to train and evaluate the optimization methods described in Section 3.4.

3.2 Navigation Overhead

We conducted a controlled experiment in order to calculate the average decision and pointing time ($M + P$) for novice and expert users. The novice users were experiencing the application for the first time and the expert users had years of professional experience with it. We measured the overhead associated with switching between entities by having users execute a sequence of commands from diverse entities. In order to obtain data from realistic transitions from one command to another, we selected the commands among *digrams* (pairs) of commands actually executed. Specifically, we obtained from the command log data 32 million digrams (27 thousand unique ones) and ordered them by the frequency of their execution. We then selected one set of digrams coming from commands associated with the same entity (e.g. move wall, delete wall), and one set of digrams coming from commands associated with different entities (e.g. add line, edit hatch). We then selected for each of the two sets, four commands from the most frequently executed digrams, four commands from the digrams executed with median frequency, and four commands from the least frequently executed digrams. This gave us in total 48 commands. We also selected another ten commands at random as a training set for users to try out before the experiment. The metrics associated with the experiment are also listed in the Table 1.

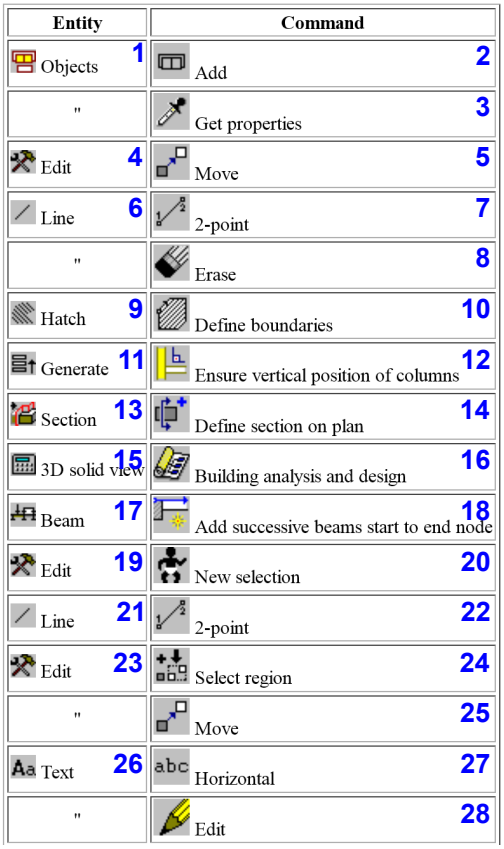

Figure 2: Part of the interaction scenario.

Figure 2 illustrates part of the scenario given to the users. The (blue) top-right numbers in each box indicate the sequence of the scenario's steps. (The numbers were not given to the users, but users were instructed on the process to execute commands.) Note that commands applied to the same entity (e.g. 2, 3) can be executed in sequence without requiring the specification of another entity, whereas those coming from different entities (e.g. 5, 7) require in between the selection of the another entity or command group (e.g. 4 — Edit).

We also instrumented the CAD program to record the time of each command invocation with millisecond accuracy. In addition we displayed after each command execution a prompt, which the user had to acknowledge by pressing OK. This forced the mouse cursor to move away from the command selection area and into the drawing area, as is also the case when the commands are executed in practice.

A graphical summary of the results we obtained is provided in Figure 3. As can be seen, the variation in execution times between commands of the same type was relatively small. As expected, switching commands (left in the Figure) requires more time than using commands of the same entity (right in the Figure). Furthermore, results varied depending on the users' expertise. Expert users could execute commands 32% faster than novices.

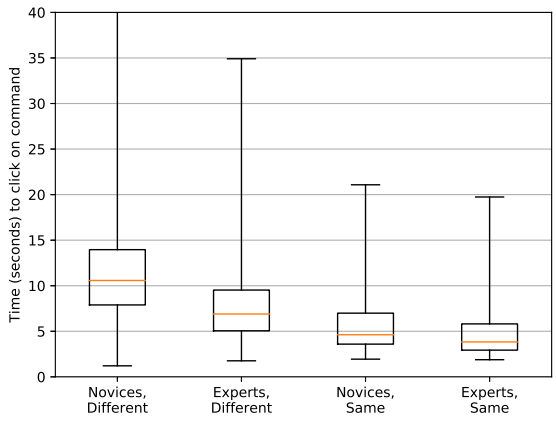

Figure 3: Command execution time of novice and expert users, executing commands from different or same entities

Table 2: Command Navigation Overhead (s)

	Novices	Experts
Same Entity (T_S)	6.1	4.7
Different Entity (T_D)	12.1	8.0

Based on the experimental data, we derived the average KLM's pointing and mental preparation time, $P + M$, for the two types of commands and two types of users listed in Table 2.

3.3 Assumptions

In order to evaluate the different command group optimization methods, we make the following assumptions for our implementation of KLM.

(1) The time needed to point and click a command (P) and the time spent on cognitive processes before clicking a command (M) are equal for all commands, regardless of their characteristics.
(2) The only variables that affect the time $P + M$ are the user's expertise and whether the command belongs to the same entity as the preceding one, or not.
(3) A given scenario of commands will always require the same time to be executed, when the expertise of users and the groups of commands remain the same.
(4) Commands that are always displayed on the screen require $P+M$ time for their execution, which is the same as executing a command without switching between entities.
(5) Normally, the complexity of the executed commands affects the execution time. However, in our experiments we use a modified version of the CAD application where calling a command only tracks the time it was called. Therefore, the complexity of the commands does not need to be considered when comparing the methods.

3.4 Training and Evaluation

To test the optimization methods under consideration, we first implement each method's algorithm, then train it with command execution scenarios, and finally evaluate its performance based on other execution scenarios. In order to determine the effects of sampling, we train the algorithms both with a large and a small data sample. To avoid over-fitting, for the training we use 50% of the scenarios when training the algorithm with extensive data, and only 1% of the scenarios when training on a sample of the data. For the evaluation of the algorithms, we use the other 50% of the telemetry data. In this way, we can also investigate the extend to which a small sample can produce results equivalent to those that can be obtained by using the whole dataset and thus allow resources to be conserved.

We grade the performance of each algorithm based on the total task execution time, T_T, required to execute a series of commands whose grouping has been optimized with it using KLM [2] We calculate T_T as follows. Let T_S be the time required to click a command in the same (current) entity, T_D the time required to click a command in a different entity, N_S the number of commands executed while in the same entity, and N_D the number of commands executed when switching entities, and N_A the number of commands always available. Then the total task execution time T_T used to evaluate a command grouping algorithm's performance is given by:

$$T_T = T_S \times (N_S + N_A) + T_D \times N_D \quad (3)$$

Based on previous work, on our own experience, and on the needs of our study we propose seven optimization approaches, which are described in the following sections. For each command grouping optimization method we describe its underlying algorithm and the specific way or parameters we used to evaluate T_T.

3.4.1 *Method: CURRENT.*

Algorithm. CURRENT is introduced as our baseline and not as an optimization method. It reflects the application's current command grouping.

Evaluation. We evaluate the current grouping with the Equation 3, counting those commands that are executed after changing an entity as N_D and the rest as N_S. No commands are always available in the current version, so we set $N_A = 0$.

3.4.2 *Method: ALL.*

Algorithm. ALL establishes the theoretical optimum that we will attempt to achieve with the following grouping algorithms. We minimize command group switching time by simulating a GUI setup where all the commands are available on an (unrealistically large) screen. Approximating this might be possible in workstations with an extra wide monitor used by expert users who can move rapidly their mouse from command to command, without having to decide in which entity the desired command is nested.

Evaluation. If every command is directly available on the screen, then the time needed for each one will be the same and equal to T_S. Therefore, we are using the Equation 3, assigning all commands C to be always available ($N_A = |C|$). Moreover, ALL involves no training time and therefore the relevant field in table 3 is omitted.

3.4.3 Method: NAIVE.

Algorithm. NAIVE makes the most frequently executed commands across all users always available to the user, in the form of an additional toolbar. To do this, we simply calculate command frequencies, once with the sample and once with the whole data set. We assume that the CAD's executed command frequency remains stable across time, Therefore, recalculating using new data should not improve the method's efficiency.

Evaluation. NAIVE is also evaluated with Equation 3. However, in this case we assign the N most frequently executed commands to be always available ($N_A = N$). Additionally, we run the algorithm with a variable number of commands, N, to investigate the point where adding commands does not lower significantly the execution time.

3.4.4 Method: GROUP.

Algorithm. GROUP, which was initially proposed by the application's designers, brings together most commonly used commands (e.g. delete or move) under a single command button. This could be applied on any object on the CAD's screen, automatically recognize the object's entity, and then call the actual command.

This method requires no training on data. The groups were determined heuristically and decided jointly by the authors and expert users. The groups decided were seven in total and 22% of the commands executed belonged to these groups.

Evaluation. For the evaluation we use Equation 3, while assigning all the grouped commands G to be always available ($N_A = G$).

3.4.5 Method: MRU-B.

Algorithm. Given the wide variety of users that use complex applications, we improve NAIVE with MRU-B, which stands for *Most Recently Used — Batch*. To train this grouping algorithm, we calculate the command frequencies for every user participating in a training period, and through these we determine the frequently used commands to always display in a personalized separate toolbar.

Evaluation. For the evaluation of MRU-B we used Equation 3, assigning for each user, their N most frequently executed commands, to be always available ($N_A = N$). Again, we evaluate the algorithm on various numbers of always available commands to find at which number the benefit becomes insignificant and whether this number is the same as the one found by NAIVE.

Algorithm 1 MRU-O

```
T_T = 0
for every user session and command do
    if not same entity or command in MRU set then
        T'_T = T_T + T_D
    else
        T'_T = T_T + T_S
    end if
    update displayed MRU set
end for
```

3.4.6 Method: MRU-O.

Algorithm MRU-O stands for *Most Recently Used — Online*, and differs from MRU-B in that there is no prior training period. Instead, the most frequently used commands are calculated on the fly and updated continuously. We assume that the cost of not having a well-established set of frequent commands in the beginning is less than the benefit of continuously adjusting the set according to the user behavior. Additionally, this approach does not require a training set.

Evaluation. For the algorithm's evaluation we use a mixed training-evaluation approach shown in Algorithm 1. MRU-O differs from MRU-B in that the training happens incrementally. Therefore, we do not differentiate between a sample and the whole dataset and use only the evaluation dataset of 50%.

3.4.7 Method: OPT(KS).

Algorithm. With OPT(KS) we use an optimization algorithm to select the commands to be made always available. This differs from NAIVE in that the selection of these commands takes into account the command execution order and the overhead of switching between entities. To do this in a deterministic way, we implement an algorithm, which is based on the infamous Knapsack Problem. For every command we evaluate the task execution time using Equation 3, as if this command was the only one always available on screen. Based on this we obtain the set of commands that are the highest contributors to the task execution time as the ones to be made always available on the screen. This training part is computed with the sample and whole data sets.

Evaluation. The evaluation follows the Equation 3, having the V most valuable commands always available ($N_A = V$).

Algorithm 2 OPT(GA)

```
population = Generate_random_population
for every iteration do
    fitness = evaluate(population)
    roulette_wheel = get_roulette_wheel(fitness)
    new_population = crossover(population, roulette_wheel)
    new_population = mutate(population)
    population = new_population + get_elites(population)
end for
Evaluate with get_elites(population)
```

3.4.8 Method: OPT(GA).

Algorithm. OPT(GA) is a stochastic optimization method that uses a genetic algorithm [7, 9, 12] to find the optimal set of commands to be always available to users. To do so, as seen in Algorithm 2 we create a population of random commands sets, and iterate using a process of keeping the best, generating new sets with a crossover function, while mutating the sets. The above are governed by a fitness function that evaluates each member of the population, as if it was the optimal solution of the NAIVE algorithm.

We chose not to train this method on the whole data set, but only on the sample, as we have already seen that the sample and the whole produce very similar results and because this is our most resource-intensive method.

As the algorithm's parameters, based on existing literature [10], we used a population of 50 members, a crossover rate, CR, of 0.6, a generation gap of 1, no scaling window, an elitist selection strategy that retains $1 - CR$ organisms, and a mutation probability of 0.001. We stop the algorithm after 1500 iterations, because through consecutive runs we found that at after about 1000 iterations the population stopped improving.

Evaluation. The evaluation uses Equation 3 on the organism with the best fitness, having its F commands to be always available ($N_A = F$). The algorithm's execution time was considerable. Specifically, it runs in $O(g \cdot n \cdot m)$ time, where g denotes the number of generations (1500), n the population size (50), and m the size of each member (7). It required nine times more execution time than all the other methods combined. Consequently, due to resource constraints, we only evaluated OPT(GA) with the sample data.

3.4.9 *Method: CLUSTER.*

Algorithm. The CLUSTER algorithms attempts to improve the optimization methods by first clustering the users and then applying the optimization separately. It is based on the insight that the CAD application is used by three types of users:

- architects, who mainly deal with architectural entities, such as walls, floors, and openings,
- civil engineers, who mainly deal with structural entities, such as pillars, beams, and slabs, and
- professionals who combine both roles in their work.

First, users are clustered with a common K-means algorithm and those who were not present in the training data are integrated in the most populous cluster. Then, we apply to each cluster the Knapsack Problem optimization algorithm, which we found be dramatically more efficient than the genetic algorithm. For the clustering, as well as for the training part of the optimization, we used both the small sample and the whole (50%) data set.

Evaluation. The evaluation of every cluster takes place separately. We evaluate every cluster using Equation 3, having the cluster's optimal set of commands C to be always available ($N_A = |C|$). Then the sub-scores of the clusters are added together to produce the overall score.

3.5 Threats to Validity

The model we used raises internal validity threats, while the single-sourced telemetry data limits our study's generalizability and thus its external validity.

3.5.1 *Internal Validity.*

Given our restricted access to information regarding KLM's variables, our simplification of it may have produced results that will not match fully the data of actual applications. Our telemetry data comes from years of user interaction logs in a production context with limited details, thus restricting our ability to obtain from it all KLM variables. To address this problem we ran the controlled experiment described in Section 3.2. During the experiment, we assumed that our changes in the GUI affect the decision and pointing time $M + P$ together, and that the keystroke or clicking time K, the homing time H, and the drawing time D remain constant. We controlled H and D by having the controlled experiment's subjects focused on the task and by eliminating the drawing time factor, simulating it with a uniform GUI dialog confirmation action. However, we could not isolate P and therefore we combine it with M.

This is a departure from what happens in practice. For example, we assume that putting all the commands on the screen (ALL) is the theoretical optimum, because it minimizes M, but at the same time it is likely to substantially raise P. We address this issue by not considering ALL as a suggested method, and by making the number of commands available on the screen the same for all the other methods.

We also examined the trustworthiness of our model and method as follows. Using the telemetry data we acquired, we extracted real-life scenarios and their real execution time. We then calculated the CURRENT time, using the controlled experiment score, as if it was another method. We found that the actual times were 6.4 times longer and had a mean with an additional 10.75 seconds. Moreover, the variance score was bellow 0.01.

This is explained mostly by the fact that in real-life users had to manipulate drawing elements on the CAD screen, while in the controlled experiment they only had to acknowledge a dialog box. In addition, numerous distractions and setbacks that happen during a user's session, are not taken into account in our model. These include talking with colleagues, answering the telephone, thinking, browsing the internet, and leaving the computer while a session is running. This justifies our choice to use a lab-controlled experiment to acquire the data that predict the employed timings and using the telemetry data only to obtain the command execution scenarios.

3.5.2 *External Validity.*

Implementing our methods, in diverse environments, or using variations of the algorithms we propose, may produce task execution time optimization levels different from the ones we report. Our results give an overview and an estimation of the methods, as we aim to recommend the optimization techniques that are better for different application types. Therefore, we have not tried to optimize the execution of each specific method, as this is beyond the scope of our work. Consequently, tuning each algorithm we used can result in additional improvements. However, since we used sound and efficient versions of these algorithms, we assume that such changes will not materially affect our conclusions.

3.5.3 *Data Availability.*

Moreover, generalizing to applications other than CAD may alter the degree of achievable optimization, due to the varying nature of the available data. The described methods rely heavily on the given dataset, which is derived from a single application. Some programs may log users and their actions and others may lack the logging details we used, providing only aggregate command and session logs. Furthermore, some may use specific roles for users, which the algorithms can readily utilize.

4 PRACTICAL APPLICATION

Our research demonstrates that practitioners should use search-based techniques to improve users' task execution efficiency of GUI applications. We demonstrate the effectiveness of small samples of telemetry data, when used with the described optimization algorithms.

Table 3: Execution Time Improvement and Runtime of Command Grouping Algorithms

	182,273 scenarios			1822 scenarios			Normalized
Method	**Novices**	**Experts**	**Time** (s)	**Novices**	**Experts**	**Time** (s)	**Time**
ALL	18.56%	13.31%	—	18.56%	13.31%	—	—
NAIVE	13.22%	9.48%	7.79	13.22%	9.48%	0.15	0.00
GROUP	0.56%	0.48%	0.00	0.56%	0.48%	0.00	0.00
MRUB	10.37%	7.43%	74.21	6.72%	4.81%	51.9	0.24
MRUO	13.50%	9.63%	1.24	13.50%	9.63%	1.24	0.01
OPT(KS)	17.43%	12.52%	1946.63	17.38%	12.52%	49.81	0.23
OPT(GA)	—	—	—	17.40%	12.48%	19749.88	91.19
CLUSTER	17.43%	12.52%	2035.29	17.43%	12.52%	1804.87	8.33

4.1 Suggested Grouping Methods

Table 3 aggregates the results for all the discussed methods, for the whole and the sample training data, for novices and experts, as well as their absolute and normalized training times. The execution time percentages compared to the current situation show an improvement for both novices and experts, being between 0.00 and the theoretical optimum. The time columns show the time, in seconds, that is needed for training each algorithm. Finally, each algorithm's, r_m, *normalized time* is the proportion of time it requires for its execution, t_m, relative to that of all others:

$$r_m = \frac{t_m}{\sum_{n=1}^{7} t_n} \tag{4}$$

CLUSTER. We find CLUSTER to be the most efficient method, as it achieved the highest proximity to the theoretical optimum, while it had a medium-size execution time. It is relatively easy to implement, as there are various clustering libraries available and the Knapsack Problem algorithm is extensively discussed in the literature.

GROUP. On the contrary, the GROUP method, despite its promising idea and the careful preparation, proved to be useless in the examined dataset. The failure resulted from a simple fact: 22% of the commands belong in one of the seven groups, but 98% of them were executed right after a command from the same entity. In that way the time-reduction effect of GROUP was small. The success of such a method to optimize command groups depends on the groups made, as well as the type of the commands. In an application where the grouped commands are usually used after commands that belong to the same entities, this method will not be useful. However, in applications where the opposite is true and where the commands can be grouped intuitively, GROUP may result in a cheap and effective optimization.

NAIVE. It is a fact that NAIVE lacks in efficiency compared with the other methods. However, it is the cheapest one to implement and easiest one to train. It run very fast, in linear time. Additionally, it works regardless of the sample size. Training with 1% and 50% of the data, produced exactly the same improvement, rendering the use of the sample a clearly more efficient practice. It should therefore be a good candidate for applications that are still in development mode or developed and tested in an agile fashion. It can offer a quick deployment for optimization, in cases where few data and resources are available.

MRU-B. Batch training for every user, MRU-B, failed to satisfy our expectations. Intuitively, we expected MRU-B to be an improvement of NAIVE. It finds the most frequent commands, not globally, but for each user and therefore it should fit the data better. However, we found it to be about 3% worse than NAIVE, when training on the whole data set and a surprising 6% worse, when training on the sample. With further examination the cause is apparent. Our dataset spans years of program runs and new users are introduced throughout its span. New users have no command frequencies associated with them. Therefore, initially, they cannot benefit from the toolbar personalized with their most frequently executed commands, thus decreasing significantly the method's performance.

However, there are environments where users are fixed for a specific timespan. There, MRU-B can be more efficient and almost as cheap as NAIVE. Furthermore, given an organization whose employers carry specific tasks repetitively and work for specific roles, a successful MRU-B can be applied to the roles instead of the individuals.

For example, an enterprise resource planning (ERP) application which has numerous fixed roles, that are well documented, can benefit from such a method. ERP applications usually list hundreds of commands or *transactions* in multiple nested menus. Their users often struggle to find the desired transactions in the command tree or try to memorize alphanumerical commands that call the transactions. In such cases we've seen supervisors compose a set of "favorite" commands, with a heuristically determined combination and order, which their provide to their reports. Using an MRU-B method on their roles, could produce easily and with a low computational cost the most frequently used commands for each role, without relying on heuristics and personal judgment.

MRU-O. Unlike MRU-B, MRU-O ran excellently on our dataset and produced very satisfying results. Exhibiting seven times lower training time than NAIVE, it resulted in a small increase in performance compared to it. Our calculations show that it exhibits the lowest training time. This however is not fully representative, as, in contrast with the other algorithms, it does not only need training, but is continuously recalculating the command invocation frequencies, work that is not included as a training time. Additionally, it manages to capture trends in command frequency, adapts to individual preferences, and adapts according to the users' habits. MRU-O would therefore be the preferable solution to applications with commonly

shifting modes of use, as well as those that invite new users frequently and on a continuous basis. It would be even more useful in cases where users are a mix of professionals and novices, do not follow specific roles, experiment with diverse commands, and apply the software to many fields and for multiple purposes. These are cases where little is known about the users, and the application is learning from them.

A typical example of such an application could be an image editor, such as Adobe Photoshop. Its users include novices, portrait picture editors photo-manipulation artists, 3D content creators, and professional animation creators. Currently, the Photoshop GUI offers a toolbar with more than 20 basic nested menus with icons, and more than 10 multiply nested menus that open additional command and control panels. Only the 3D and the animation environments offer an optimized GUI. The command structure complexity makes it difficult for novice users and for any user of multiple tools to learn the application and to recall the place of each tool. Given the application's vaguely defined users and the overly complicated GUI, such an application could benefit a lot by providing users a toolbar with their most frequently used commands.

OPT(KS). Exceptional results were found using combinatorial optimization algorithms. OPT(KS) along with CLUSTER achieved results closest to the optimal when trained on the whole dataset. When trained on the sample, OPT(KS)'s results were inferior to CLUSTER's by 0.05%.

Our solution runs in linear time, with the input being the product of the number of commands available and the length of scenarios. Therefore it takes about 300 times more time to finish than NAIVE but is more efficient than the other effective methods. The Knapsack Algorithm relies on heavy training for its operation. Therefore it needs many data to produce the same results, and these may not be available. In comparison with the combinatorial method, clustering on the users requires an additional *a priori* metric on them. This may be easily acquirable using the frequency of executed commands.

An example where CLUSTER is preferable is the aforementioned ERP system. Deploying it in a big organization easily provides a vast amount of structured data about users that can be used to cluster them. An example where OPT(KS) is preferable is Photoshop, as there is no information on users, but very specific task categories, such as photo retouch, 3D object rendering, animation creation, and photo collage creation. There the Knapsack Algorithm could find to display heavily used commands that are deeply nested in menus. The first example is user-centric and learns from the user while the second one is task-centric and optimizes for the task.

OPT(GA). The stochastic genetic optimization algorithm resulted in similar results as with the deterministic optimization of the Knapsack Problem. Clustering on frequencies results in a cluster with about 90% of users and two more with the rest. Optimizing them separately, results in small improvement over OPT(KS). Additionally, CLUSTER resulted in exactly the same improvement for the sample and the whole dataset, proving again our hypothesis that the sample is enough to improve the application.

However its execution time was 400 times higher than OPT(KS) and 10 times higher than CLUSTER. Its design and parameters make it more complicated. Specifically, it had the largest code size, and five parameters needed to be chosen: the population size, the crossover rate, the number of generations, the elitist selection strategy, and the mutation probability. These bring an unnecessary overhead to those who will implement it. Additionally, its execution time made it impossible for our resources to train on the whole dataset. Therefore, it is generally not suggested to be used for optimizing command groups.

Table 4: Method Guide

	Easy and Cheap	Optimal Solutions
Specific tasks	NAIVE	OPT(KS)
Specific users/roles	MRU-O	CLUSTER
No prior knowledge	NAIVE	OPT(GA)
Large dataset	NAIVE	MRU-B, MRU-O

Table 4 contains an overview of the most efficient methods for each scenario. NAIVE dominates those that provide an easy and cheap solution for its simplicity and speed. The Knapsack Problem–inspired optimization works better for applications with task-centric available data, such as an image editor. Moreover, clustering the users and then applying the optimization is recommended when there is adequate information for the users to be clustered. When there is not such information, a stochastic approach, such as a genetic algorithm, can be more successful. However, because of the algorithm's complexity, developers should choose a fitness function that executes quickly. In the end, user and command frequency mappings may not be as potent as the previous in reducing task execution time, but can be much quicker when the required dataset is very large in size.

4.2 Sample vs Whole Training Set

A second major conclusion we make in this study is that our hypothesis on the necessity of using a large dataset of program runs to compute the above methods failed.

Our large training dataset (50% of all data) has about seven million executed commands, during 90,000 sessions, involving more than a 1500 unique users. The 1% sample we acquired from it has about 140,000 executed commands, during 18,000 sessions, involving about 700 users.

Based on the results listed in Table 3, we conclude that the sample set is enough to achieve an almost-perfect command group optimization by any of the methods that we deemed efficient. OPT(KS) exhibited only a 0.05% improvement when computed with the whole dataset. Even more, NAIVE and the most efficient, CLUSTER, resulted in exactly the same level of optimization for both datasets, rendering the use of a much bigger dataset expensive and useless. The only method that contradicts this finding was MRU-B, as it exhibited a significant improvement when ran with the large dataset. It relied heavily on user data and the large dataset contained information for two times more users than the sample. On the other hand, we did not find MRU-O to depend on the dataset's size.

In conclusion, using a sample dataset of the above size is advisable and should be enough, except it there is a reason for an algorithm to fit the needs of a small subset of specific users.

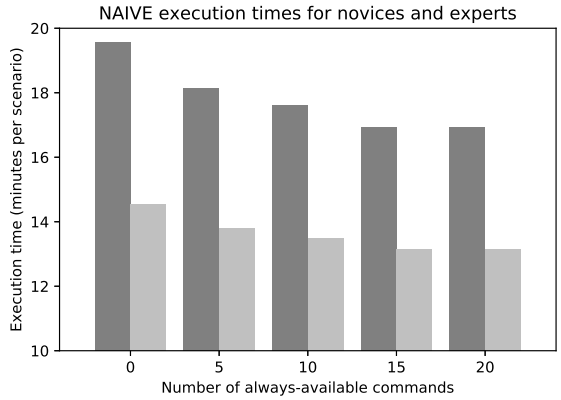

Figure 4: The effect of increasing the number of always-available commands provided by the NAIVE algorithm.

4.3 Optimal Number of Commands on Screen

Another conclusion regards the optimal number of commands that a toolbar displaying frequently used commands should have. Here, we do not take into account the visual and aesthetic aspect of this choice, looking instead at the empirical quantitative derivation of this number.

Figure 4 shows that after displaying 15 commands, the benefit of having more commands on the screen, regardless of the aesthetics and the pointing time P, becomes insignificant. From 15 to 20 commands, for MRU-B the benefit of adding 33% more commands is bellow 0.1%. Moreover, for NAIVE it is less than 0.001%. However, the benefit is more than 0.5% for both algorithms when moving from 10 to 15 commands. Consequently, we consider a number between 15 and 20 to be the optimal.

5 CONCLUSIONS

Our study demonstrates how the problem of efficiently organizing the commands of a complex application's GUI can be solved using evidence and not intuition. By exploiting a large amount of telemetry data, obtained during years of program runs of a CAD application, we showed that diverse methods can reduce task execution times by up to 17.5%. Such an optimization, applied in practice, can easily reduce the mental effort, running time and therefore the cost of executing common scenarios in commercial applications.

Additionally, we find that using a large dataset is not important, because a sequence of about a hundred thousand commands can produce exactly the same results as a few millions.

Our findings suggest the following two avenues for further research. First, experiments on end users performing production work can demonstrate that our suggestions hold in practice, and that our assumptions do not significantly affect the outcome. Second, our methods can be applied to other applications in order to verify their generalizability.

Acknowledgements

We thank *LH Logismiki* for providing us access to the system's source code and the anonymized data used in the study. The project associated with this work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 732223. The first author is a recipient of the Hellenic Petroleum Group scholarship.

REFERENCES

[1] Evgeniy Abdulin. 2011. Using the keystroke-level model for designing user interface on middle-sized touch screens. In *CHI'11 Extended Abstracts on Human Factors in Computing Systems*. ACM, 673–686.
[2] Stuart K. Card, Thomas P. Moran, and Allen Newell. 1980. The keystroke-level model for user performance time with interactive systems. *Commun. ACM* 23, 7 (1980), 396–410.
[3] Chia-Fen Chi and Ku-Lun Chung. 1996. Task analysis for computer-aided design (CAD) at a keystroke level. *Applied ergonomics* 27, 4 (1996), 255–265.
[4] Andy Cockburn, Carl Gutwin, and Saul Greenberg. 2007. A predictive model of menu performance. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 627–636.
[5] Martin Dostal. 2010. User Acceptance of the Microsoft Ribbon User Interface. *Advances in Data Newtworks, Communications, Computers* (2010). OCLC: 751522773.
[6] Leah Findlater, Karyn Moffatt, Joanna McGrenere, and Jessica Dawson. 2009. Ephemeral adaptation: The use of gradual onset to improve menu selection performance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1655–1664.
[7] Stephanie Forrest. 1996. Genetic Algorithms. *Comput. Surveys* 28, 1 (March 1996), 77–83.
[8] Erik Frokjar, Morten Hertzum, and Kasper Hornb\a ek. 2000. Measuring usability: are effectiveness, efficiency, and satisfaction really correlated?. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*. ACM, 345–352.
[9] David E. Goldberg. 1989. *Genetic Algorithms: In Search of Optimization & Machine Learning*. Addison-Wesley.
[10] John J. Grefenstette. 1986. Optimization of Control Parameters for Genetic Algorithms. *IEEE Transactions on Systems, Man, and Cybernetics* 16, 1 (1986), 122–128.
[11] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *Comput. Surveys* 45, 1 (2012), 11.
[12] J. H. Holland. 1975. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Michigan.
[13] Paul Holleis, Friederike Otto, Heinrich Hussmann, and Albrecht Schmidt. 2007. Keystroke-level model for advanced mobile phone interaction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1505–1514.
[14] Andreas Holzinger. 2005. Usability Engineering Methods for Software Developers. *Commun. ACM* 48, 1 (Jan. 2005), 71–74. https://doi.org/10.1145/1039539.1039541
[15] David Kieras. 2001. Using the keystroke-level model to estimate execution times. *University of Michigan* 555 (2001).
[16] David M. Lane, H. Albert Napier, Richard R. Batsell, and John L. Naman. 1993. Predicting the Skilled Use of Hierarchical Menus with the Keystroke-level Model. *Hum.-Comput. Interact.* 8, 2 (June 1993), 185–192. https://doi.org/10.1207/s15327051hci0802_4
[17] Lu Luo and Bonnie E. John. 2005. Predicting Task Execution Time on Handheld Devices Using the Keystroke-level Model. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems (CHI EA '05)*. ACM, New York, NY, USA, 1605–1608. https://doi.org/10.1145/1056808.1056977
[18] Shouichi Matsui and Seiji Yamada. 2008. Genetic algorithm can optimize hierarchical menus. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 1385–1388.
[19] Shouichi Matsui and Seiji Yamada. 2008. Optimizing hierarchical menus by genetic algorithm and simulated annealing. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*. ACM, 1587–1594.
[20] Rohae Myung. 2004. Keystroke-level analysis of Korean text entry methods on mobile phones. *International Journal of Human-Computer Studies* 60, 5-6 (May 2004), 545–563. https://doi.org/10.1016/j.ijhcs.2003.10.002
[21] Jakob Nielsen. 1994. Usability inspection methods. In *Conference companion on Human factors in computing systems*. ACM, 413–414.
[22] Joey Scarr, Andy Cockburn, Carl Gutwin, and Andrea Bunt. 2012. Improving command selection with CommandMaps. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 257–266.
[23] Luigi Troiano and Cosimo Birtolo. 2014. Genetic algorithms supporting generative design of user interfaces: Examples. *Information Sciences* 259 (Feb. 2014), 433–451. https://doi.org/10.1016/j.ins.2012.01.006
[24] Luigi Troiano, Cosimo Birtolo, Roberto Armenise, and Gennaro Cirillo. 2008. Optimization of Menu Layouts by Means of Genetic Algorithms. In *Evolutionary Computation in Combinatorial Optimization*, Jano van Hemert and Carlos Cotta (Eds.), Vol. 4972. 242–253. DOI: 10.1007/978-3-540-78604-7_21.