A Fast Algorithm to Locate Concepts in Execution Traces

Soumaya Medini[1], Philippe Galinier[1], Massimiliano Di Penta[2], Yann-Gaël Guéhéneuc[1], Giuliano Antoniol[1]

[1] DGIGL, École Polytechnique de Montréal, Canada
[2] RCOST, University of Sannio, Italy —
soumaya.medini@polymtl.ca, philippe.galinier@polymtl.ca, dipenta@unisannio.it, yann-gael.gueheneuc@polymtl.ca, antoniol@ieee.org

Abstract. The identification of cohesive segments in execution traces is a important step in concept location which, in turns, is of paramount importance for many program-comprehension activities. In this paper, we reformulate the trace segmentation problem as a dynamic programming problem. Differently to approaches based on genetic algorithms, dynamic programming can compute an exact solution with better performance than previous approaches, even on long traces. We describe the new problem formulation and the algorithmic details of our approach. We then compare the performances of dynamic programming with those of a genetic algorithm, showing that dynamic programming reduces dramatically the time required to segment traces, without sacrificing precision and recall; even slightly improving them.

Keywords: Concept identification, dynamic analysis, information retrieval, dynamic programming.

1 Introduction

Program comprehension is an important preliminary activity that may require half of the effort devoted to software maintenance and evolution. An important task during program comprehension is concept location, which aims at identifying concepts (*e.g.*, domain concepts, user-observable features) and locating them within code regions or, more generally, into software artifact chunks [14, 8]. The literature reports concept location approaches built upon static [1] and dynamic [24, 23] analyses; information retrieval (IR) [19]; and hybrid (static and dynamic) [3, 12, 5] techniques. Dynamic and hybrid approaches rely on execution traces.

A typical scenario in which concept location takes part is the following. Let us suppose that (1) a failure has been observed in a software system under certain execution conditions, (2) unfortunately, such execution conditions are hard to reproduce, but (3) one execution trace was saved during such a failure. Maintainers then face the difficult and demanding task of analyzing the one execution trace of the system to identify in the trace the set(s) of methods pertaining to the failure, *i.e.*, some unexpected sequence(s) of method invocations, and then to relate the invoked methods to some features producing the failure.

Inspired by the above scenario, a step of the (hybrid) concept location process has been recently defined as the *trace segmentation problem*, where the textual content of the methods contained in execution traces is used to split the traces into segments that likely participate in the implementation of some concepts related to some features [5, 4]. The underlying assumption of this step is that, if a specific feature is being executed within a complex scenario (*e.g.*, "Open a Web page from a browser" or "Save an image in a paint application"), then the set of methods being invoked is likely to be conceptually cohesive, decoupled from those of other features, and invoked in sequence. Unfortunately, despite the use of meta-heuristic techniques, *e.g.*, genetic algorithms [5] and their parallelization [4], this step yields to computationally intensive approaches that do not scale on traces of thousands of methods.

In this paper, we reformulate the trace segmentation problem as a dynamic programming (DP) problem. Differently to approaches based on meta-heuristic techniques, in particular genetic algorithms (GA), the DP approach can compute an exact solution to the trace segmentation problem with better performance that previous approaches, which would possibly make this approach more scalable. The DP approach relies on the same representation and fitness function as proposed for a previous approach based on a GA [4, 5], however, the trace segmentation problem is reformulated as an optimization problem taking advantage of (1) the order of the methods in the trace, (2) the additive property of the fitness function, and (3) the Bellman's Principle of Optimality [7].

Thus, the contributions of this paper are:

1. A novel reformulation of the trace segmentation problem as a DP problem. We describe the new problem formulation and its algorithmic details.
2. An empirical study comparing the DP approach with a previous GA approach [4, 5]. We show that the DP approach can segment traces in a few seconds, at most, while the GA approach takes several minutes/hours. Despite such a drastic improvement of performances, precision and recall do not decrease; they even slightly increase.

The remainder of the paper is organized as follows. Section 2 summarizes a previous trace-segmentation approach for the sake of completeness [4, 5]. Section 3 explains trace segmentation using GA and DP approaches. Section 4 describes the empirical study and reports and discusses the obtained results. Section 5 recalls related work. Section 6 concludes the paper with future work.

2 The Trace Segmentation Problem

This section summarizes essential details of a previous trace segmentation approach [4, 5], which problem we reformulate as a dynamic programming problem. Therefore, the five steps of the two approaches are identical, with the only difference that the trace segmentation was previously performed using a GA algorithm and that we describe the use of DP in Section 3.

2.1 Steps 1 and 2 – System Instrumentation and Trace Collection

First, a software system under study is instrumented using the *instrumentor* of MoDeC to collect traces of its execution under some scenarios. MoDeC is a tool to extract and model sequence diagrams from Java systems [22], implemented using the Apache BCEL bytecode transformation library[3]. The tool also allows to manually label parts of the traces during executions of the instrumented systems, which we did to produce our oracle.

2.2 Step 3 – Pruning and Compressing Traces

Usually, execution traces contain methods invoked in most scenarios, *e.g.*, methods related to logging or GUI events. Yet, it is unlikely that such invocations are related to any particular concept, *i.e.*, they are utility methods. We prune out methods having an invocation frequency greater than $Q3 + 2 \times IQR$, where $Q3$ is the third quartile (75% percentile) of the distribution and IQR is the interquartile range because these methods do not provide useful information when segmenting traces and locating concepts.

Finally, we compress the traces using a Run Length Encoding (RLE) algorithm to remove repetitions of method invocations. We introduced this compression to address scalability issues of the GA approach [4, 5]. We still apply the RLE compression to compare segments obtained with the DP approach with those obtained using the GA approach when segmenting the same traces.

2.3 Step 4 – Textual Analysis of Method Source Code

Trace segmentation aims at grouping together subsequent method invocations that form conceptually cohesive groups. The conceptual cohesion among method is computed using the Conceptual Cohesion metric defined by Marcus *et al.* [15].

We first extract terms from source code, split compound identifiers separated by camel case (*e.g.*, `getBook` is split into `get` and `book`), remove programming language keywords and English stop words, and perform stemming [18]. We then index the obtained terms using the *tf-idf* indexing mechanisms [6]. We obtain a term–document matrix, and finally, we apply Latent Semantic Indexing (LSI) [11] to reduce the term–document matrix into a concept–document[4] matrix, choosing, as in previous work, a LSI subspace size equal to 50.

2.4 Step 5 – Trace Splitting through Optimization Techniques

The final step consists of applying some optimization techniques to segment the obtained trace. Applying an optimization technique requires a representation of the trace and of a trace segmentation and a means to evaluate the quality of a

[3] http://jakarta.apache.org/bcel/

[4] In LSI "concept" refers to orthonormal dimensions of the LSI space, while in the rest of the paper "concept" means some abstraction relevant to developers.

trace segmentation, *i.e.*, a fitness function. In the following paragraphs, we reuse where possible previous notations and definitions [5] for the sake of simplicity.

We represent a problem solution, *i.e.*, a trace segmentation, as a bit-string as long as the execution trace in number of method invocations. Each method invocation is represented as a "0", except the last method invocation in a segment, which is represented as a "1". For example, the bit-string $\underbrace{00010010001}_{11}$ represents a trace containing 11 method invocations and split into three segments: the first four method invocations, the next three, and the last four.

The fitness function drives the optimization technique to produce a (near) optimal segmentation of a trace into segments likely to relate to some concepts. It relies on the software design principles of cohesion and coupling, already adopted in the past to identify modules in software systems [17], although we use conceptual (*i.e.*, textual) cohesion and coupling measures [15, 20], rather than structural cohesion and coupling measures.

Segment cohesion (COH) is the average (textual) similarity between the source code any pair of methods invoked in a given segment l. It is computed using the formulas in Equation 1 where $begin(l)$ is the position of the first method invocation of the l^{th} segment and $end(l)$ the position of the last method invocation in that segment. The similarity σ between methods m_i and m_j is computed using the cosine similarity measure over the LSI matrix from the previous step. COH is the average of the similarity [15, 20] of all pairs of methods in a segment.

Segment coupling (COU) is the average similarity between a segment l and all other segments in the trace, computed using Equation 2, where N is the trace length. It represents, for a given segment, the average similarity between methods in that segment and those in different ones.

Thus, we compute the quality of the segmentation of a trace split into K segments using the fitness function (fit) defined in Equation 3, which balances segment cohesion and their coupling with other segments in the split trace.

$$COH_l = \frac{\sum_{i=begin(l)}^{end(l)-1} \sum_{j=i+1}^{end(l)} \sigma(m_i, m_j)}{(end(l) - begin(l) + 1) \cdot (end(l) - begin(l))/2} \tag{1}$$

$$COU_l = \frac{\sum_{i=begin(l)}^{end(l)} \sum_{j=1, j<begin(l) \text{ or } j>end(l)}^{l} \sigma(m_i, m_j)}{(N - (end(l) - begin(l) + 1)) \cdot (end(l) - begin(l) + 1)} \tag{2}$$

$$fit(segmentation) = \frac{1}{K} \cdot \sum_{i=1}^{K} \frac{COH_i}{COU_i + 1} \tag{3}$$

3 Segmenting Traces using a Genetic Algorithm and Dynamic Programming

We now use previous notations and definitions to describe the use of a GA algorithm to segment traces and the reformulation of the trace segmentation problem as a dynamic programming problem.

3.1 Trace Segmentation using a Genetic Algorithm

Section 3 described the representations of a trace and its segmentation and a fitness function. We now define the mutation, crossover, and selection operators, used by a GA to segment traces [4, 5].

The mutation operator randomly chooses one bit in the trace representation and flips it over. Flipping a “0” into a “1” means splitting an existing segment into two segments, while flipping a “1” into a “0” means merging two consecutive segments. The crossover operator is the standard 2-points crossover. Given two individuals, two random positions x, y, with $x < y$, are chosen in one individual’s bit-string and the bits from x to y are swapped between the two individuals to create a new offspring. The selection operator is the roulette-wheel selection. We use a simple GA with no elitism, *i.e.*, it does not guarantee to retain best individuals across subsequent generations.

3.2 Trace Segmentation using Dynamic Programming

Dynamic Programming (DP) is a technique to solve search and optimization problems with overlapping sub-problems and an optimal substructure. It is based on the divide-and-conquer strategy where a problem is divided into sub-problems, recursively solved, and where the solution of the original problem is obtained by combining the solutions of the sub-problems [7, 10].

Sub-problems are overlapping if the solving of a (sub-)problem depends on the solutions of two or more other sub-problems, *e.g.*, the computation of the Fibonachi numbers. The original problem must have a particular structure. First, it must be possible to recursively break it down into sub-problems up to some elementary problem easily solved; second, it must be possible to express the solution of the original problem in term of the solutions of the sub-problems; and, third, the Bellman’s principle of optimality must be applicable. For our trace segmentation problem, we interpret this principle as follows. When computing a trace segmentation, at a given intermediate method invocation in the trace and for a given number of segments ending with that invocation, only the best among those possible partial splits, will be, possibly, part of the final optimal solution. Thus, we must record only the best fitness for any segmentation and we must expand only the corresponding best segment to include more method invocation, possibly including the entire trace.

Using the previous notations and definitions, we thus reformulate the trace segmentation problem as a problem of dividing a string of characters, which can be solved efficiently using DP and formalized as follows. Let $\mathcal{A} = \{1, 2, \ldots, n\}$ be an alphabet of n symbols, *i.e.*, method invocations, and $T[1 \ldots N]$ be an array of method invocations of $\mathcal{A}$, *i.e.*, an execution trace. Given an interval $T[p \ldots q]$ $(1 \leq p \leq q \leq N)$ of $T[1 \ldots N]$, as explained Section 2, we compute COH as the average similarity between the elements of $T[p \ldots q]$ and the interval coupling, COU, as the average similarity between any element of $T[p \ldots q]$ (methods between p and q) and any element of $T[1 \ldots N] - T[p \ldots q]$. We compute the *score of an interval* as COH/COU.

A segmentation S of $T[1 \dots L](L \leq N)$ is a partition S of $T[1 \dots L]$ in k_S intervals: $S = \{T[1 \dots a_1], T[a_1 + 1 \dots a_2] \dots T[a_{k-1} + 1 \dots a_k = N]\}$. We denote such a segmentation by $(a_0 = 0, a_1, \dots, a_{k_S} = L)$. We then define the segmentation score (*e.g.*, fitness) of an array as the average score of its intervals. Therefore, the *trace segmentation problem* consists to find a segmentation of $T[1 \dots N]$ maximizing the score fit, as defined in 2.

We introduce the definitions D1–D4 to explain our DP approach:

(D1) $A(p,q) = \Sigma_{i=p}^{q-1} \Sigma_{j=i+1}^{q} \sigma(i,j)$
(D2) $B(p,q) = \Sigma_{i=p}^{q} \Sigma_{j=1 \dots N(j \notin [p,q])} \sigma(i,j)$
(D3) $f(p,q) = \frac{2 \times (N-(q-p+1))}{(q-p)} \times \frac{A(p,q)}{B(p,q)}$
(D4) $fit(k,L) = max_{\{(a_i)_{i=0..k} : a_0 = 0, a_i < a_{i+1}, a_k = L\}} \Sigma_{i=1..k} f(a_{i-1}+1, a_i)$

We notice that the COH and COU of an interval $T[p \dots q]$ correspond to $\frac{2 \times A(p,q)}{(q-p) \times (q-p+1)}$ and $\frac{B(p,q)}{(N-(q-p+1)) \times (q-p+1)}$, respectively. Thus $f(p,q)$ represents the score of the interval $T[p \dots q]$. It also represents the contribution of the interval to a solution and $fit(k,L)$ corresponds to the maximum score of a (k,L)-segmentation, *i.e.*, a segmentation of $T[1 \dots L]$ in k intervals. Therefore, the optimum segmentation score is $max_{k=1}^{N/2} \frac{fit(k,N)}{k}$.

If we consider a solution ending at p (sub-trace $T[1 \dots p]$) and made up by k segments, then its score is $fit(k,p)$ and we have multiple optimum segmentations: one for each possible k in $1 < k < p/2$. When we extend the sub-trace to q, $T[1 \dots p \dots q]$ and given a solution made up of k segments ending in p, we seek the solution $fit(k+1,q)$ into $max_{p=k \dots q}(fit(k,p) + f(p+1,q))$, where $1 \leq k < q \leq N$. If we pre-compute and store $fit(k,p)$ in a table, we do not need to recompute the expensive COH and COU every time to evaluate $fit(k+1,q)$. However, we still must compute $f(p+1,q)$ for every sub-problems and we perform this computation efficiently using the following definitions:

(D5) $\Delta(p,q) = \Sigma_{i=p}^{q-1} \sigma(T[i], T[q])$
(D6) $\Theta(p) = \Sigma_{i=1..N(i \neq p)} \sigma(T[i], T[p])$

It can be proved that $\Delta(p,q) = \Delta(p+1,q) + \sigma(T[p], T[q])$ and, thus, $A(p,q) = A(p,q-1) + \Delta(p,q)$ and $B(p,q+1) = B(p,q) + \Theta(q+1) - 2 \times \Delta(p,q+1)$ and thus we can recursively update $A(p,q)$ and $B(p,q+1)$. We choose $q = p+1$, which means that we extend the current solution one method at the time from left-to-right and that $A(p,q)$ becomes $A(p,p+1)$ and $B(p,q+1)$ becomes $B(p,p+2)$, which we can pre-compute (from previous values) and stored into two arrays.

To conclude, we can compute $fit(k+1,p+1)$ using $fit(k,i)$ and the sum of the values of $f(i+1,p+1)$, which we can compute by dividing $A(i+1,p+1)$ by $B(i+1,p+1)$, both already pre-computed. The DP approach is thus fast because it goes left-to-right and reuses as much as possible of previous computation.

We show below the pseudo-code of (a basic version of) the algorithm at the core of the DP approach.

Algorithm DP split
Input:
integers n and N, matrix of similarities $Sim[1..n][1..n]$, array $T[1..N]$
Output: matrix of fitnesses $fit[1..N][1..N]$

```
1.  For L=1..N do
2.      Theta := comp_theta(L)
3.      Delta := 0
4.      A[L] := 0
5.      B[L] := Theta
6.      For p=L-1..1 do
7.          Delta := Delta + Sim[T[p]][T[L]]
8.          A[p] := A[p-1] + Delta
9.          B[p] := B[p-1] + Theta − 2 × Delta
10. For L=1..N do
11.     fit[1][L] := comp_f(1,L)
12.     For k=2..L do
13.         F_max := 0
14.         For p=k..L-1 do
15.             F_max:=max(F_max, fit[k-1][p] + comp_f(p+1))
16.         fit[k][L] := F_max
17. Return fit
```

where the input matrices $Sim[1..n][1..n]$ and $T[1..N]$ contain the similarities between methods and the trace encoding, respectively. The function *comp_f()* computes the value of f based on definition $D3$ and *comp_theta* recursively evaluates $\Theta(p)$. The most expensive part of the algorithm are the nested loops at lines 10, 12, and 14. The algorithm, in this basic formulation, has a complexity of $\mathcal{O}(N^3)$, which is also the (worst case) complexity of the evaluation of the GA fitness function as both COH and COU have worst case complexity of $\mathcal{O}(N^2)$ and in the worst case must be evaluated for $N/2$ segments. Thus, a single step of the GA approach equates the entire calculation of the DP approach.

4 Empirical Study

This section reports an empirical study comparing the GA approach proposed by Asadi *et al.* [5] with our novel DP approach. The *goal* of this study is to analyze the performances of the trace segmentation approaches based on GA and DP with the *purpose* of evaluating their capability to identify meaningful concepts in traces. The *quality focus* is the accuracy and completeness of the identified concepts. The *perspective* is that of researchers who want to evaluate which of the two techniques (GA or DP) better solves the trace segmentation problem. The *context* consists of two trace segmentation approaches, one based on GA and one on DP, and of the same execution traces used in previous work [5] and extracted from two open-source systems, ArgoUML and JHotDraw.

Table 1. Data of the empirical study.

(a) Statistics of the two systems.

Systems	**NOC**	**KLOC**	**Release Dates**
ArgoUML v0.18.1	1,267	203	30/04/05
JHotDraw v5.4b2	413	45	1/02/04

(b) Statistics of the collected traces.

Systems	**Scenarios**	**Original Size**	**Cleaned Sizes**	**Compressed Sizes**
ArgoUML	Start, Create note, Stop	34,746	821	588
	Start, Create class, Create note, Stop	64,947	1,066	764
JHotDraw	Start, Draw rectangle, Stop	6,668	447	240
	Start, Add text, Draw rectangle, Stop	13,841	753	361
	Start, Draw rectangle, Cut rectangle, Stop	11,215	1,206	414
	Start, Spawn window, Draw circle, Stop	16,366	670	433

ArgoUML[5] is an open-source UML modelling tool with advanced features, such as reverse engineering and code generation. The ArgoUML project started in September 2000 and is still active. We analyzed release 0.19.8. *JHotDraw*[6] is a Java framework for drawing 2D graphics. JHotDraw started in October 2000 with the main purpose of illustrating the use of design patterns in a real context. We analyzed release 5.1. Table 1(a) summarizes the systems statistics. We generated traces by exercising various scenarios in the two systems. Table 1(b) summarizes the scenarios and shows that the generated traces include from 6,000 to almost 65,000 method invocations. The compressed traces include from 240 up to more than 750 method invocations.

This study aims at answering the three following research questions:

- **RQ1.** *How do the performances of the GA and DP approaches compare in terms of fitness values, convergence times, and numbers of segments?*
- **RQ2.** *How do the GA and DP approaches perform in terms of overlaps between the automatic segmentation and the manually-built oracle, i.e., recall?*
- **RQ3.** *How do the precision values of the GA and DP approaches compare when splitting execution traces?*

4.1 Study Settings and Analysis Method

The GA approach was implemented using the *Java GA Lib*[7] library. We use a simple GA with no elitism, *i.e.*, it does not guarantee to retain best individuals across subsequent generations. We set the population size to 200 individuals

[5] http://argouml.tigris.org
[6] http://www.jhotdraw.org
[7] http://sourceforge.net/projects/javagalib/

and a number of generations of 2,000 for shorter traces (those of JHotDraw) and 3,000 for longer ones (those of ArgoUML). The crossover probability was set to 70% and the mutation to 5%, which are values used in many GA applications.

The DP approach scans the trace from left-to-right building the exact solution and in its current formulation does not have any configuration parameter.

In previous work, the results of the GA approach were reported for for multiple runs of the algorithm to account for the nondeterministic nature of the technique. We only report the results of the DP approach for one of its run per traces because it is by nature deterministic and multiple runs would produce exactly the same results.

To address **RQ1**, we compare the value of the fitness function reached by the GA approach with the value of the segmentation score obtained by the DP approach. The values of the fitness function and segmentation score *per se* do not say anything about the quality of the obtained solutions. Yet, we compare these values to assess, given a representation and a fitness function/segmentation score, which of the GA or DP approach obtain the best value. We also compare the execution times of the GA and DP approaches. We finally report the number of segments that the two approaches create for each execution trace.

For **RQ2**, we compare the overlap between a manually-built oracle and segments identified by the GA and DP approaches. We build an oracle by manually assigning a concept to trace segments—using the tagging feature of the instrumentor tool—while executing the instrumented systems. Given the segments determined by the tags in the oracle and given the segments obtained by an execution of either of the approaches, we compute the overlap between each manually-tagged segment in the oracle and the closest, most similar segment obtained automatically. Let us consider a (compressed) trace composed of N method invocations $T \equiv m_1, \ldots m_N$ and partitioned in k segments $s_1 \ldots s_k$. For each segment s_x, we compute the maximum overlap between s_x and the manually-tagged segments so_y as follows:

$$max(Jaccard(s_x, so_y)), y \in \{1 \ldots k\}$$

where:

$$Jaccard(s_x, so_y) = \frac{|s_x \cap so_y|}{|s_x \cup s_y|}$$

and where union and intersection are computed considering method invocations occurring at a given position in the trace.

For **RQ3**, we evaluate (and compare) the precision of both the GA and DP approaches in terms of precision, which is defined as follows:

$$Precision(s_x, so_y) = \frac{|s_x \cap so_y|}{|s_y|}$$

where s_x is a segment obtained by an automatic approach (GA or DP) and so_y is a segment in the corresponding trace of the oracle.

For **RQ1**, **RQ2**, and **RQ3**, we statistically compare results obtained with the GA and DP approaches using the non-parametric, paired Wilcoxon test. We also compute the magnitude of the differences using the non-parametric effect-size Cliff's δ measure [13], which, for dependent samples, as in our study, is defined as the probability that a randomly-selected member of one sample DP has a higher response than a randomly-selected member of the second sample GA, minus the reverse probability:

$$\delta = \frac{\left|\mathrm{DP}^i > \mathrm{GA}^j\right| - \left|\mathrm{GA}^j > \mathrm{DP}^i\right|}{|\mathrm{DP}|\,|\mathrm{GA}|}$$

The effect size δ is considered small for $0.148 \leq \delta < 0.33$, medium for $0.33 \leq \delta < 0.474$ and large for $\delta \geq 0.474$ [13].

4.2 Results

This section reports the results of the empirical study. Data sets are available for replication on-line[8].

Table 2. Numbers of segments, values of fitness function/segmentation score, and times required by the GA and DP approaches.

System	Scenario	# of Segments		Fitness		Time (s)	
		GA	DP	GA	DP	GA	DP
ArgoUML	(1)	24	13	0.54	0.58	7,080	2.13
	(2)	73	19	0.52	0.60	10,800	4.33
JHotDraw	(1)	17	21	0.39	0.67	2,040	0.13
	(2)	21	21	0.38	0.69	1,260	0.64
	(3)	56	20	0.46	0.72	1,200	0.86
	(4)	63	26	0.34	0.69	240	1.00

Regarding **RQ1**, Table 2 summarizes the obtained results using both the GA and DP approaches, in terms of (1) numbers of segments in which the traces were split, (2) achieved values of fitness function/segmentation score, and (3) times needed to complete the segmentations (in seconds). The DP approach tends to segment the trace in less segments than the GA one, with the exception of Scenario (1) of JHotDraw, composed of one feature only and for which the number of segments is 21 for both approaches, and of Scenario 3 of JHotDraw, for which the DP approach creates 21 segments whereas GA creates only 17 segments. The difference of the numbers of segments is not statistically significant (p-value=0.10), although Cliff's δ effect size is high (1.16) and in favor of the GA approach. Looking at the values of the fitness function/segmentation score, the DP approach always produces better values than the GA one. The Wilxocon test indicates that the difference is statistically significant (p-value=0.03) and the Cliff's δ effect size is high (0.76): the DP approach performs significantly

[8] http://web.soccerlab.polymtl.ca/ser-repos/public/dp_sp.tar.gz

better than the GA approach, given the representations described in Section 2. Finally, the convergence times of the GA approach are by far higher than that of the DP one: from several minutes or hours (for ArgoUML) to seconds. The difference between the GA and DP approaches is statistically significant (p-value=0.03) and the effect size high (1.05). We thus answer **RQ1** by stating that *in terms of fitness values, convergence time, and numbers of segments, the DP approach out-performs the GA approach.*

Table 3. Jaccard overlaps and precision values between segments identified by the GA and DP approaches.

System	Scenario	Feature	Jaccard		Precision	
			GA	DP	GA	DP
ArgoUML	(1)	Create Note	0.33	0.87	1.00	0.99
	(2)	Create Class	0.26	0.53	1.00	1.00
	(2)	Create Note	0.34	0.56	1.00	1.00
JHotDraw	(1)	Draw Rectangle	0.90	0.75	0.90	1.00
	(2)	Add Text	0.31	0.33	0.36	0.39
	(2)	Draw Rectangle	0.62	0.52	0.62	1.00
	(3)	Draw Rectangle	0.74	0.24	0.79	0.24
	(3)	Cut Rectangle	0.22	0.31	1.00	1.00
	(4)	Draw Circle	0.82	0.82	0.82	1.00
	(4)	Spawn window	0.42	0.44	1.00	1.00

To address **RQ2**, we evaluate the Jaccard overlap between the manually-identified segments corresponding to each feature of the execution scenarios and the segments obtained using the GA and DP approaches. Columns 4 and 5 of Table 3 report the results. Jaccard scores are always higher for the GA approach than for the DP one, with the only exception of the *Draw Rectangle* feature in JHotDraw, for which the Wilcoxon paired test indicates that there is no significant difference between Jaccard scores (p-value=0.56). The obtained Cliff's δ (0.11) is small, although slightly in favor of the DP approach. We thus answer **RQ2** by stating that *in terms of overlap, segments obtained with the GA and DP approaches do not significantly differ and the DP approach has thus a recall similar to that of the GA one.*

Regarding **RQ3**, Columns 6 and 7 of Table 3 compare the precision values obtained using the GA and DP approaches. Consistently with results reported in previous work [5], precision is almost always higher than 80%, with some exceptions, in particular the *Add Text* and *Draw Rectangle* features of JHotDraw. There is only one case for which the DP approach exhibits a lower precision than the GA one: for the *Draw Rectangle* feature of JHotDraw (Scenario 3) where the DP approach has a precision of 0.24 whereas the GA one has a precision of 0.79. Yet, in general, the Wilcoxon paired test indicates no significant differences between the GA and DP approaches (p-value=0.52) and the Cliff's δ (0.04) indicates a negligible difference between the two approaches. In conclusion, we answer **RQ3** by stating that *the precision obtained using the DP approach does not significantly differ from the one obtained using the GA approach.*

4.3 Threats to Validity

We now discuss the threats to the validity of our empirical study.

Threats to *construct validity* concern the relation between theory and observation. In this study, they are mainly due to measurement errors. To compare the GA and DP approaches, other than considering the achieved fitness function values and the computation times, we used precision and Jaccard overlap, already used in a previous work [5] as well as in the past [21]. While in this paper, due to the lack of space, we cannot report a qualitative analysis of the obtained segments, previous work [5] already showed that a segmentation with high overlap and precision produces meaningful segments. Finally, we cannot compare the times required by the GA and DP approaches to achieve the a same fitness value/segmentation score because the DP approach always reaches, by construction, the global optimum while the GA approach does not. Moreover, even if the achieved fitness values and segmentation scores are different, we showed that the DP approach is able to reach a better score in a shorter time.

Threats to *internal validity* concern confounding factors that could affect our results. These could be due to the presence, in the execution traces, of extra method invocations related to GUI events or other system events. The frequency-based pruning explained in Step 3 of Section 2 mitigates this threat.

Threats to *conclusion validity* concern the relationship between treatment and outcome. We statistically compared the performances of the GA and DP approaches using the non-parametric Wilcoxon paired test and used the non-parametric Cliff's δ effect size measure.

Threats to *external validity* concern the possibility to generalize our results. Although we compared the GA and DP approaches on traces from two different systems, further studies on larger traces and more complex systems are needed, especially to better demonstrate the scalability of the DP approach. Indeed, we showed that the DP approach out-performs the GA one in terms of computation times to segment traces but did not show that, differently from the GA approach, its computation time does not exponentially increase with trace size.

5 Related Work

As sketched in the introduction, concept location approaches can be divided into static, dynamic, and hybrids approaches.

Static approaches relies on information statically collected from the program under analysis. Anquetil and Lethbridge [1] proposed techniques to extract concepts from a very simple source of information, *i.e.*, file names. Chen and Rajilich [9] developed an approach to locate concepts using only Abstract System-Dependency Graph (ASDG). The ASDG is constructed using a subset of the information of a system-dependency graph (SDG). Finally, Marcus *et al.* [16] performed concept location using an approach based on information retrieval. As Marcus *et al.*, our approach strongly relies on the textual content of the program source code.

Dynamic approaches use one or more execution traces to locate concepts in the source code. In their seminal work, Wilde and Scully [24] used test cases to produce execution traces; concepts location was performed by comparing different traces: one in which the concept is executed and another without concept. Similarly, Poshyvanyk *et al.* [19] used multiple traces from multiple scenarios.

Hybrid approaches have been introduced to overcome the limitations of dynamic and static approaches. Static approaches often fail to properly capture a system behavior, while dynamic approaches are sensitive to the chosen execution traces. Antoniol and Guéhéneuc [2] presented a hybrid approach to concept location and reported results for real-life large object-oriented multi-threaded systems. They used knowledge filtering and probabilistic ranking to overcome the difficulties of collecting uninteresting events. This work was improved [3] by using the notion of epidemiology of diseases in locating the concepts.

We share with previous works the general idea of concept location and with hybrid approaches the idea of using both static and dynamic data. This work extends our previous work [5, 4] by reformulating the trace segmentation problem as a DP problem and comparing the previous results with the new ones.

6 Conclusions and future work

Execution trace segmentation is a step in the conception location process. It consists in splitting an execution trace into segments most likely to correspond to some concepts. Previous work showed that it is possible to split execution traces into cohesive segments using a genetic algorithm (GA) [4, 5].

In this paper, we reformulate the trace segmentation problem as a dynamic programming (DP) problem and, specifically, as a particular case of the string splitting problem. We showed that we can benefit from the overlapping subproblems and an optimal substructure of the string splitting problem to reuse computed scores of intervals and segmentation scores and, thus, to obtain dramatic gains in performances without loss in precision and recall. Indeed, differently from the GA approach, the DP approach reuses pre-computed cohesion and coupling values among subsequent segments of an execution trace, which is not possible using genetic algorithms, due to their very nature. We believe that other problems, such as segmenting composed identifiers into component terms, could be modelled in a similar way and, thus, that we, as a community, should be careful when analyzing a problem: a different, possibly non-orthodox, problem formulation may lead to surprisingly good performances.

We empirically compared the DP and GA approaches, using the same data set from previous work [4, 5]. Our empirical study consisted in the execution traces from ArgoUML and JHotDraw, which were previously used to validate the GA approach. Results indicated that the DP approach can achieve results similar to the GA approach in terms of precision and recall when its segmentation is compared with a manually-built oracle. They also show that the DP approach has significantly better results in terms of the optimum segmentation score vs. fitness function. More importantly, results showed that the DP approach significantly out-performed the GA approach in terms of the times required to produce the segmentations: where the GA approach would take several minutes, even hours; the DP approach just takes a few seconds.

Work in progress aims at further validating the scalability of the DP trace segmentation approach as well as at complementing the approach with segment labelling to make the produced segments better suitable for program-comprehension activities. Finally, we would like to explore sub-optimal solution of the DP problem with much lower bound complexities and evaluate the impact on the solution accuracy.

References

1. Anquetil, N., Lethbridge, T.: Extracting concepts from file names: a new file clustering criterion. In: Proceedings of the International Conference on Software Engineering. pp. 84–93. IEEE Computer Society Press (1998)
2. Antoniol, G., Guéhéneuc, Y.G.: Feature identification: a novel approach and a case study. In: Proceedings of the International Conference on Software Maintenance. pp. 357–366. IEEE Computer Society Press (2005)
3. Antoniol, G., Guéhéneuc, Y.G.: Feature identification: An epidemiological metaphor. IEEE Transactions on Software Engineering 32(9), 627–641 (2006)
4. Asadi, F., Antoniol, G., Guéhéneuc, Y.G.: Concept locations with genetic algorithms: A comparison of four distributed architectures. In: Proceedings of the International Symposium on Search Based Software Engineering. pp. 153–162. IEEE Computer Society Press (2010)
5. Asadi, F., Penta, M.D., Antoniol, G., Guéhéneuc, Y.G.: A heuristic-based approach to identify concepts in execution traces. In: Proceedings of the European Conference on Software Maintenance and Reengineering. pp. 31–40. IEEE Computer Society Press (2010)
6. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley (1999)
7. Bellman, R.E., Dreyfus, S.E.: Applied Dynamic Programming, vol. 1. Princeton University Press (1962)
8. Biggerstaff, T., Mitbander, B., Webster, D.: The concept assignment problem in program understanding. In: Proceedings of the International Conference on Software Engineering. pp. 482–498 (1993)
9. Chen, K., Rajlich, V.: Case study of feature location using dependence graph. In: Proceedings of the International Workshop on Program Comprehension. pp. 241–249. IEEE Computer Society Press (2000)
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introductions to Algorithms. MIT Press (1990)
11. Deerwester, S., Dumais, S.T., Furnas, G.W., Landauer, T.K., Harshman, R.: Indexing by latent semantic analysis. Journal of the American Society for Information Science 41(6), 391–407 (1990)

12. Eaddy, M., Aho, A.V., Antoniol, G., Guéhéneuc, Y.G.: Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In: Proceedings of the International Conference on Program Comprehension. pp. 53–62. IEEE Computer Society Press (2008)
13. Grissom, R.J., Kim, J.J.: Effect sizes for research: A broad practical approach. Lawrence Earlbaum Associates, 2nd edn. (2005)
14. Kozaczynski, V., Ning, J.Q., Engberts, A.: Program concept recognition and transformation. IEEE Transactions on Software Engineering 18(12), 1065–1075 (1992)
15. Marcus, A., Poshyvanyk, D., Ferenc, R.: Using the conceptual cohesion of classes for fault prediction in object-oriented systems. IEEE Transactions on Software Engineering 34(2), 287–300 (2008)
16. Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J.I.: An information retrieval approach to concept location in source code. In: Proceedings of the Working Conference on Reverse Engineering. pp. 214–223. IEEE Computer Society Press (2004)
17. Mitchell, B.S., Mancoridis, S.: On the automatic modularization of software systems using the bunch tool. IEEE Transactions on Software Engineering 32(3), 193–208 (2006)
18. Porter, M.F.: An algorithm for suffix stripping. Program 14(3), 130–137 (1980)
19. Poshyvanyk, D., Guéhéneuc, Y.G., Marcus, A., Antoniol, G., Rajlich, V.: Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. Transactions on Software Engineering 33(6), 420–432 (2007)
20. Poshyvanyk, D., Marcus, A.: The conceptual coupling metrics for object-oriented systems. In: Proceedings of the International Conference on Software Maintenance. pp. 469–478. IEEE Computer Society Press (2006)
21. Salah, M., Mancordis, S., Antoniol, G., Penta, M.D.: Towards employing use-cases and dynamic analysis to comprehend mozilla. In: Proceedings of the International Conference on Software Maintenance. pp. 639–642. IEEE Press (2005)
22. Janice Ka-Yee Ng, Guéhéneuc, Y.G., Antoniol, G.: Identification of behavioral and creational design motifs through dynamic analysis. Journal of Software Maintenance and Evolution: Research and Practice 22(8), 597–627 (2010)
23. Tonella, P., Ceccato, M.: Aspect mining through the formal concept analysis of execution traces. In: Proceedings of Working Conference on Reverse Engineering. pp. 112–121 (2004)
24. Wilde, N., Scully, M.C.: Software reconnaissance: Mapping program features to code. Journal of Software Maintenance - Research and Practice 7(1), 49–62 (1995)