FACULTY OF SCIENCES

Optimising Continuous Integration using Test Case Prioritisation

Pieter De Clercq
Student number: 01503338

Supervisors: Prof. dr. Bruno Volckaert, Prof. dr. ir. Filip De Turck
Counsellors: Jasper Vaneessen, Dwight Kerkhove

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in de informatica

Academic year 2019-2020

GHENT UNIVERSITY

Pieter De Clercq – May 23, 2020.

Acknowledgements

Completing this thesis would not have been possible without the help and support of many people, some of which I want to thank personally.

First of all, I want to thank prof. dr. Bruno Volckaert and prof. dr. ir. Filip De Turck for allowing me to propose this subject and for their prompt and clear responses to every question I have asked. I especially want to thank you for permitting me to insert a two-week hiatus during the Easter break, so I could help out on the UGent Dodona project.

Secondly, I want to express my gratitude towards my counsellors Jasper Vaneessen and Dwight Kerkhove, for steering me into researching this topic, as well as their guidance, availability, and willingness to review every intermediary version of this thesis.

Furthermore, I want to thank my parents, my brother Stijn and my family for convincing me and giving me the possibility to study at the university, to support me throughout my entire academic career and to provide me with the opportunity to pursue my childhood dreams.

Last, but surely not least, I want to thank my amazing friends, a few of them in particular. My best friend Robbe, for always being there when I need him even when I least expect it. For both supporting my wildest dreams while protecting me against my often unrealistic ideas and ambition to excel. Helena for never leaving my side, for always making me laugh when I don't want to, and most importantly to remind me that I should relax from time to time. Jana for my daily dose of laughter, fun and inexhaustible positivity. Tobiah for the endless design discussions and for outperforming me in almost every school project, to encourage me to continuously raise the bar and to never give up. Finally, I want to thank Doortje and Freija for answering my mathematical questions, regularly asking about my thesis progression and thereby motivating me to persevere.

Thank you.

Pieter – Ghent, 2020

Summary

Summary in English will come here.

Samenvatting

Nederlandse samenvatting komt hier.

Optimising Continuous Integration using Test Case Prioritisation

Pieter De Clercq

Supervisor(s): Prof. dr. B. Volckaert, Prof. dr. ir. F. De Turck, J. Vaneessen, D Kerkhove

Abstract—**This abstract is very abstract.**

Keywords—**words, will, appear, here, soon**

I. Introductie

Things will appear here. [1]

References

[1] Michael Cusumano, Akindutire Michael, and Stanley Smith, “Beyond the waterfall : software development at microsoft,” 02 1995.

Optimaliseren van Continue Integratie door middel van Test Prioritering

Pieter De Clercq

Supervisor(s): Prof. dr. B. Volckaert, Prof. dr. ir. F. De Turck, J. Vaneessen, D Kerkhove

Abstract—**Dit abstract is super abstract.**

Trefwoorden—**woorden, komen, hier**

I. Introductie

Dingen komen hier. [1]

Referenties

[1] Michael Cusumano, Akindutire Michael, and Stanley Smith, “Beyond the waterfall : software development at microsoft,” 02 1995.

Lay summary

Lay summary will come here.

Contents

Chapter 1

Proposed framework: VeloCIty [TODO REVISE]

The implementation part of this thesis consists of a framework and a set of tools, tailored at optimising the test suite as well as providing accompanying metrics and insights. The framework was named *VeloCIty* to reflect its purpose of enhancing the speed at which Continuous Integration is practised. This paper will now proceed by describing the design goals of the framework. Afterwards, a high-level schematic overview of the implemented architecture will be provided, followed by a more in-depth explanation of every pipeline step. In the final section of this chapter, the *Alpha* algorithm will be presented.

1.1 Design goals

VeloCIty has been implemented with four design goals in mind:

1. **Extensibility:** It should be possible and straightforward to support additional Continuous Integration systems, programming languages and test frameworks. Subsequently, a clear interface should be provided to integrate additional prioritisation algorithms.

2. **Minimally invasive:** Integrating VeloCIty into an existing test suite should not require drastic changes to any of the test cases.

3. **Language agnosticism:** This design goal is related to the framework being extensible. The implemented tools should not need to be aware of the programming language of the source code, nor the used test framework.

4. **Self-improvement:** The prioritisation framework supports all of the algorithms presented in **??**. It is possible that the performance of a given algorithm is strongly dependent on the nature of the project it is being applied to. In order to facilitate this behaviour, the framework should be able to measure the performance of every algorithm and “learn” which algorithm offers the best prediction, given a set of source code.

1.2 Architecture

The architecture of the VeloCIty framework consists of seven steps that are performed sequentially in a pipeline fashion, as illustrated in the sequence diagram (Figure 1.1). Every step is executed by one of three individual components, which will now be introduced briefly.

1.2.1 Agent

The first component that will be discussed is the agent. This is the only component that depends actively on both the programming language, as well as the used test framework, since it must interact directly with the source code and test suite. For every programming language or test framework that needs to be supported, a different implementation of an agent must be provided. This implementations are however strongly related, so much code can be reused or even shared. In this thesis, an agent was implemented in Java, more specifically as a plugin for the widely used Gradle and JUnit test framework. This combination was previously described in **??**. This plugin is responsible for running the test suite in a certain prioritised order, which is obtained by communicating with the controller (subsection 1.2.2). After the test cases have been executed, the plugin sends a feedback report to the controller, where it is analysed.

1.2.2 Controller

The second component is the core of the framework, acting as an intermediary between the agent on the left side and the predictor (subsection 1.2.3) on the right side. In order to satisfy the second design goal and allow language agnosticism, the agent communicates with the controller using the `HTTP` protocol by exposing a *REST*-interface. Representational State Transfer `[REST]` is a software architecture used by modern web applications that allows standardised communication using existing HTTP methods. On the right side, the controller does not communicate directly with the predictor, but rather stores prediction requests in a shared database which is periodically polled by the predictor. Besides routing prediction requests from the agent to the predictor, the controller will also update the meta predictor by evaluating the accuracy of earlier predictions of this project.

1.2.3 Predictor and Metrics

The final component is twofold. Its main responsibility is to apply the prioritisation algorithms and predict an order in which the test cases should be executed. This order is calculated by first executing ten algorithms and subsequently picking the algorithm

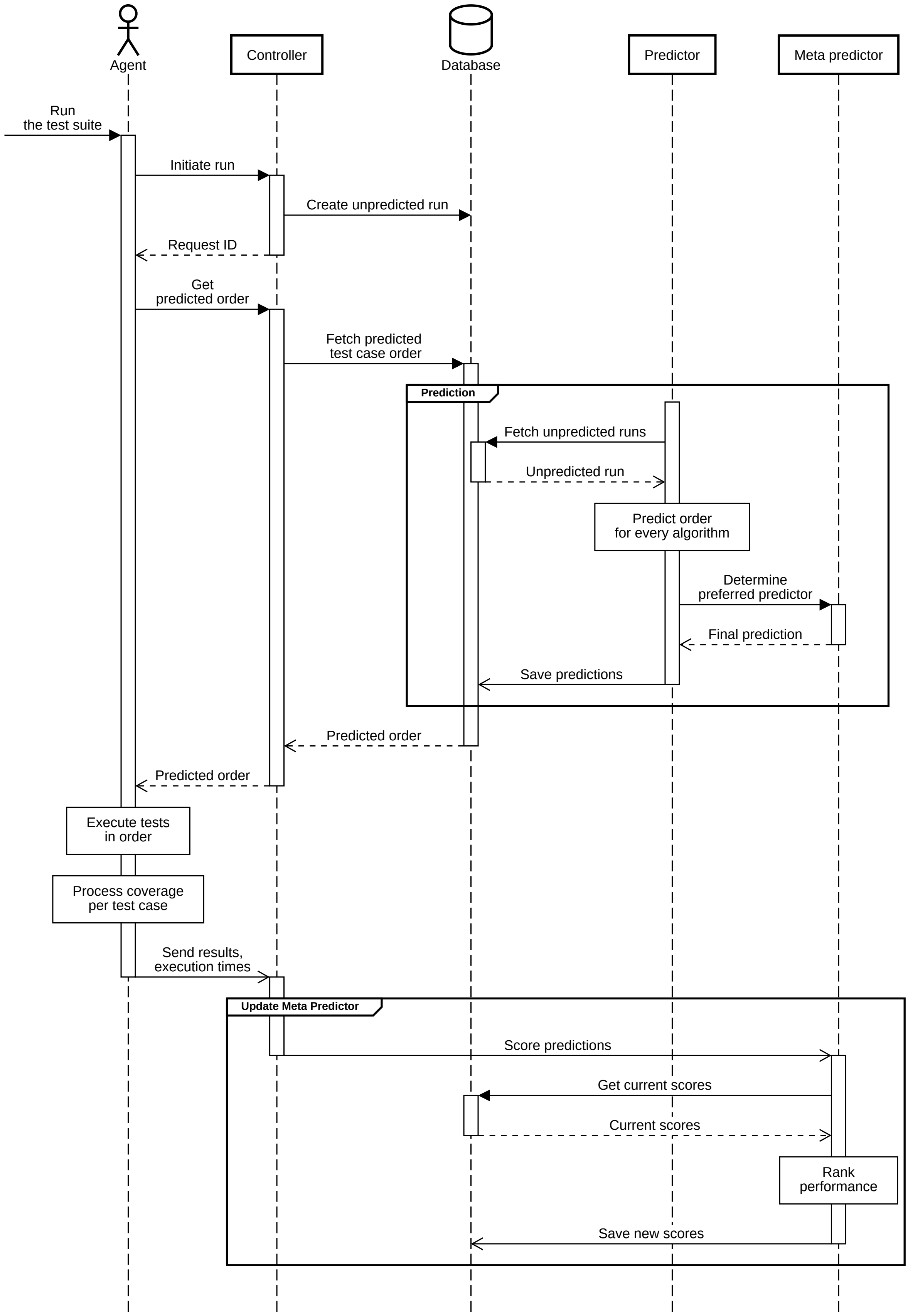

Figure 1.1: Sequence diagram of VeloCIty

that has been preferred by the meta predictor. Additionally, this component is able to provide metrics about the test suite, such as identifying superfluous test cases by applying Test Suite Minimisation. More specifically, this redundancy is obtained using the greedy algorithm (**??**). Both of these scripts have been implemented in Python, because of its simplicity and existing libraries for many common operations, such as numerical calculations (NumPy[1]) and machine learning (TensorFlow[2]).

1.3 Pipeline

This section will elaborate on the individual steps of the pipeline. The steps will be discussed by manually executing the pipeline that has hypothetically been implemented on a Java project. For the sake of simplicity, this explanation will assume a steady-state situation, ensuring the existence of at least one completed run of this project in the database at the controller side.

1.3.1 Initialisation

As was explained before, the provided Java implementation of the agent was designed to be used in conjunction with Gradle. In order to integrate VeloCIty into a Gradle project, the build script (`build.gradle`) should be modified in two places. The first change is to include and apply the plugin in the header of the file. Afterwards, the plugin requires three properties to be configured:

- `base` the path to the Java source files, relative to the location of the build script. This will typically resemble `src/main/java`.
- `repository` the url to the git repository at which the project is hosted. This is required in subsequent steps of the pipeline, to detect which code lines have been changed in the commit currently being analysed.
- `server` the url at which the controller can be reached.

Listing 1.1 contains a minimal integration of the agent in a Gradle build script, applied to a library for generating random numbers[3]. The controller is hosted at the same host as the agent and is accessible at port `8080`.

```
buildscript {
    dependencies {
        classpath ’io.github.thepieterdc.velocity:velocity-junit:0.0.1-SNAPSHOT’
```

[1] https://numpy.org/

[2] https://www.tensorflow.org/

[3] https://github.com/thepieterdc/random-java

```
    }
}

plugins {
    id ’java’
}

apply plugin: ’velocity-junit’

velocity {
    base ’src/main/java/’
    repository ’https://github.com/thepieterdc/random-java’
    server ’http://localhost:8080’
}
```

Listing 1.1: Minimal Gradle buildscript

After the project has been configured, the test suite must be executed. For the Gradle agent, this involves executing the built-in `test` task. This task requires an additional argument to be passed, which is the commit hash of the changeset to prioritise. In every discussed Continuous Integration system, this commit hash is available as an environment variable.

The first step is for the agent to initiate a new test run in the controller. This is accomplished by sending a `POST`-request to the `/runs` endpoint of the controller, which will reply with an identifier. On the controller side, this request will result in a new prioritisation request being enqueued in the database that will asynchronously be processed by the predictor daemon in the next step.

1.3.2 Prediction

The prediction of the test execution order is performed by the predictor daemon. This daemon continuously polls the database to fetch new test runs that need to be predicted. When a new test run is detected, the predictor executes every available prediction algorithm in order to obtain multiple prioritised test sequences. The following algorithms are available:

AllInOrder The first algorithm will simply prioritise every test case alphabetically and will be used for for benchmarking purposes in **??**.

AllRandom The second algorithm has also been implemented for benchmarking purposes. This algorithm will "prioritise" every test case arbitrarily.

AffectedRandom This algorithm will only consider the test cases that cover source code lines which have been modified in the current commit. These test cases will be ordered randomly, followed by the other test cases in the test suite in no particular order.

GreedyCoverAll The first of three implementations of the Greedy algorithm (**??**) will execute the algorithm to prioritise the entire test suite.

GreedyCoverAffected As opposed to the previous greedy algorithm, the second Greedy algorithm will only consider test cases covering changed source code lines to be prioritised. After these test cases, the remaining test cases in the test suite will be ordered randomly.

GreedyTimeAll Instead of greedily attempting to cover as many lines of the source code using as few tests as possible, this implementation will attempt to execute as many tests as possible, as soon as possible. In other words, this algorithm will prioritise test cases based on their average execution time.

HGSAll This algorithm is an implementation of the algorithm presented by Harrold, Gupta and Soffa (**??**). It is executed for every test case in the test suite.

HGSAffected Similar to the *GreedyAffected* algorithm, this algorithm is identical to the previous *HGSAll* algorithm besides that it will only prioritise test cases covering changed source code lines.

ROCKET The penultimate algorithm is a straightforward implementation of the pseudocode provided in **??**.

Alpha The final algorithm has been inspired by the other implemented algorithms. section 1.4 will further elaborate on the details.

Subsequently, the final prioritisation order is determined by applying the meta predictor. Essentially, the meta predictor can be seen as a table which assigns a score to every algorithm, indicating its performance on this codebase. subsection 1.3.4 will explain later how this score is updated. The predicted order by the algorithm with the highest score is eventually elected by the meta predictor as the final prioritisation order, and saved to the database.

1.3.3 Test case execution

Regarding the agent, the identifier obtained in subsection 1.3.1 is used to poll the controller by sending a `GET` request to `/runs/id`, which will reply with the test execution order if this has already been determined. One of the discussed features of Gradle in **??** was the possibility to execute test cases in a chosen order by adding annotations. However, this feature cannot be used to implement the Java agent, since it only supports ordering test cases within the same test class. In order to facilitate complete control over the order of execution, a custom `TestProcessor` and `TestListener` have been implemented.

The `TestProcessor` is responsible for processing every test class in the classpath and forward it along with configurable options to a delegate processor. The final processor in this chain will eventually perform the actual execution of the test class. Since the delegate processors that are built into Gradle will by default execute every method in the test class, the custom processor needs to work differently. The implemented agent will first store every received test class into a list and load the class to obtain all test cases in the class using reflection. After all classes have been processed, the processor will iterate over the prioritised order. For every test case t in the order, the delegate processor is called with a tuple of the corresponding test class and an options array which excludes every test case except t. This will effectively forward the same test class multiple times to the delegate processor, but each time with an option that restricts test execution to the prioritised test case, resulting in the desired behaviour.

Subsequently, the `TestListener` is a method that is called before and after every invocation of a test case. This listener allows the agent to calculate the duration of every test case, as well as collect the intermediary coverage and save this on a per-test case basis.

1.3.4 Post-processing and analysis

The final step of the pipeline is to provide feedback to the controller, to evaluate the accuracy of the predictions and thereby implementing the fourth design goal of self-improvement. After executing all test cases, the agent sends the test case results, the execution time and the coverage per test case to the controller by issuing a `POST` request to `/runs/id/test-results` and `/runs/id/coverage`.

Upon receiving this data, the controller will update the meta predictor using the following procedure. The meta predictor is only updated if at least one of the test cases has failed, since the objective of Test Case Prioritisation is to detect failures as fast as possible, thus every prioritised order is equally good if there are no failures at all. If however a test case did fail, the predicted orders are inspected to calculate the duration until the first failed test case for every order. Subsequently, the average of all these durations is calculated. Finally, the score of every algorithm that predicted a below average duration until the first failure is increased, otherwise it is decreased. This will eventually lead to the most accurate algorithms being preferred in subsequent test runs.

1.4 Alpha algorithm

Besides the algorithms which have been presented in **??**, an additional algorithm has been implemented: the *Alpha* algorithm. This was constructed by examining the philosophy behind every discussed algorithm and subsequently combining the best ideas into a novel prioritisation algorithm. The specification below will assume the same naming convention as described in **??**. The pseudocode is provided in Algorithm 1

The algorithm consumes the following inputs:

- the set of all n test cases: $TS = \{T_1, \dots, T_n\}$
- the set of m *affected* test cases: $AS = \{T_1, \dots, T_m\} \subseteq TS$. A test case t is considered "affected" if any source code line which is covered by t has been modified or removed in the commit that is being predicted.
- C: the set of all lines in the application source code, for which a test case $t \in TS$ exists that covers this line and that has not yet been prioritised. Initially, this set contains every covered source code line.
- the failure status of every test case, for every past execution out of k executions of that test case: $F = \{F_1, \dots, F_n\}$, where $F_i = \{f_1, \dots, f_k\}$. $F_{tj} = 1$ implies that test case t has failed in execution $current - j$.
- the execution time of test case $t \in TS$ for run $r \in [1 \dots k]$, in milliseconds: D_{tr}.
- for every test case $t \in TS$, the set TL_t is composed of all source code lines that are covered by test case t.

The first step of the algorithm is to determine the execution time E_t of every test case t. This execution time is calculated as the average of the durations of every successful (i.e.) execution of t, since a test case will be prematurely aborted upon the first failed assertion, which introduces bias in the duration timings. In case t has never been executed successfully, the average is computed over every execution of t.

$$E_t = \begin{cases} \overline{\{D_{ti} | i \in [1 \dots k], F_{ti} = 0\}} & \exists j \in [1 \dots k], F_{tj} = 0 \\ \overline{\{D_{ti} | i \in [1 \dots k]\}} & \text{otherwise} \end{cases}$$

Next, the algorithm executes every affected test case that has also failed at least once in its three previous executions. This reflects the behaviour of a developer attempting to resolve the bug that caused the test case to fail. Specifically executing *affected* failing test cases first is required in case multiple test cases are failing and the developer is resolving these one by one, an idea which was extracted from the ROCKET algorithm (**??**). In case there are multiple affected failing test cases, the test cases are prioritised by increasing execution time. After every selected test case, C is updated by subtracting the code lines that have been covered by at least one of these test cases.

Afterwards, the same operation is repeated for every failed but unaffected test case, likewise ordered by increasing execution time. Where the previous step helps developers to get fast feedback about whether or not the specific failing test case they were working on has been resolved, this step ensures that other failing test cases are not forgotten and are executed early in the run as well. Similar to the previous step, C is again updated after every prioritised test case.

Research (**??**) has indicated that on average, only a small fraction (10 $\% - 20\,\%$) of all test runs will contain failed tests, resulting in the previous two steps not being executed at all. Therefore, the most time should be dedicated to executing test cases that cover affected code lines. More specifically, the next step of the algorithm executes every affected test case, sorted by decreasing cardinality of the intersection between C and the lines which are covered by the test case. Conforming to the prior two steps, C is also updated to reflect the selected test case. As a consequence of these updates, the cardinalities of these intersections change after every update, which will ultimately lead to affected tests not strictly requiring to be executed. This idea has been adopted from the Greedy algorithm **??**.

In the penultimate step, the previous operation is repeated in an identical fashion for the remaining test cases, similarly ordered by the cardinality of the intersection with the remaining uncovered lines in C.

Finally, the algorithm selects every test case which had not yet been prioritised. Notice that these test cases do not contribute to the test coverage, as every test case that would incur additional coverage would have been prioritised already in the previous step. Subsequently, these test cases are actually redundant and are therefore candidates for removal by Test Suite Minimisation. However, since this is a prioritisation algorithm, these tests will still be executed and prioritised by increasing execution time.

Algorithm 1 Alpha algorithm for Test Case Prioritisation

```
**Input:** Set $TS = \{T_1, \ldots, T_n\}$ of all test cases,
        Set $AS = \{T_1, \ldots T_m\} \subseteq TS$ of affected test cases,
        Set $C$ of all source code lines that are covered by any $t \in TS$,
        Execution times $D_{tr}$ of every test case $t$, over all $k$ runs $r$ of that test case,
        Failure status $FS$ for each test case over the previous $m$ successive iterations,
        Sets $TL = \{TL_1, \ldots, TL_n\}$ of all source code lines that are covered by test case
    $t \in TS$.
**Output:** Ordered list $P$ of $n$ test cases and their priorities.
$P \leftarrow array[1 \ldots n]$                                   ▷ initially $0$
$i \leftarrow n$
$FTS \leftarrow \{t | t \in TS \wedge (F[t][1] = 1 \vee F[t][2] = 1 \vee F[t][3] = 1)\}$
$AFTS \leftarrow AS \cap FTS$
**for all** $t \in AFTS$ **do**               ▷ sorted by execution time in $E$ (ascending)
    $C \leftarrow C \setminus TL[t]$
    $P[t] \leftarrow i$
    $i \leftarrow i - 1$
$FTS \leftarrow FTS \setminus AFTS$
**for all** $t \in FTS$ **do**                ▷ sorted by execution time in $E$ (ascending)
    $C \leftarrow C \setminus TL[t]$
    $P[t] \leftarrow i$
    $i \leftarrow i - 1$
$AS \leftarrow AS \setminus FTS$
**while** $AS \neq \emptyset$ **do**
    $t\_max \leftarrow AS[1]$                                 ▷ any element from $AS$
    $tl\_max \leftarrow \emptyset$
    **for all** $t \in AS$ **do**
        $tl\_current \leftarrow C \cap TL_t$
        **if** $|tl\_current| > |tl\_max|$ **then**
            $t\_max \leftarrow t$
            $tl\_max \leftarrow tl\_current$
    $C \leftarrow C \setminus tl\_max$
    $P[t] \leftarrow i$
    $i \leftarrow i - 1$
$TS \leftarrow TS \setminus (AS \cup FTS)$
**while** $TS \neq \emptyset$ **do**
    $t\_max \leftarrow TS[1]$                                 ▷ any element from $TS$
    $tl\_max \leftarrow \emptyset$
    **for all** $t \in TS$ **do**
        $tl\_current \leftarrow C \cap TL_t$
        **if** $|tl\_current| > |tl\_max|$ **then**
            $t\_max \leftarrow t$
            $tl\_max \leftarrow tl\_current$
    $C \leftarrow C \setminus tl\_max$
    $P[t] \leftarrow i$
    $i \leftarrow i - 1$
**return** $P$
```