

PĀLI PLATFORM **2**

The Official Manual

J. R. Bhaddacak

Version 2.0

Copyright © 2023 J. R. Bhaddacak

This work is licensed under a Creative Commons
Attribution-NonCommercial-NoDerivatives 4.0 International License.
<https://creativecommons.org/licenses/by-nc-nd/4.0/>



Release History

Version	Built on	Description
2.0	10 Nov 2022	First release, bundled with the program

Preface

PĀLI PLATFORM 1 has been released since January 2020. It might be only me who used it substantially to produce Pāli learning books.¹ After I finished the books, I have spent many months (9 or so) to rewrite PĀLI PLATFORM, to make it modernized in look, more user-friendly, to add more features, and to fix many bugs.² Now its version 2 has been done. And I think it is the time to write a full-blown user's manual because the program is quite sophisticated now. So here we are.

After the release of PĀLI PLATFORM 2, the old version will be discarded. So, the users have to learn a new way for doing things, for example, methods of typing in Pāli characters. We will learn about all features in due course. However, I still cannot explain every bit of details. One reason for this is the graphic user interface of the program is intuitively easy to

¹*Pāli for New Learners*, Book I and II

²I moved from Java 8 to Java 11 and replaced Swing UI with JavaFX. I thought at first I would use Kotlin, but it is still more difficult to work with. Java is still the best friend I have. It is not the language itself is great, but the whole Java ecosystem makes programming with Java enjoyable: enormous available libraries, fast compile time, excellent API documentation, flexible and uncomplicated deployment scheme, etc. Moreover, as I work with JavaFX so far, this GUI is the best among all cross-platform GUI toolkits. It can run and look the same in all platforms without any adjustment. Some might say Java is slow, at least in comparison with C or C++. As you shall see by using the program, it just needs start-up time. After that the slowness is negligible.

learn, and I have already provided some helps in several places. Still, the users need to spend time playing with the program to gain familiarity.

I divide the manual into five parts. The first part, Essential Starter, is crucial. It is supposed to be read deliberately. The main purpose of this part is to help the users start the program successfully in various contexts, and to provide an initial guidance and troubleshooting.

The second part is all about grammatical tools. It is enough to just go through the part quickly and come back when necessary. Several chapters are short. The longest one is Chapter 9 (Prosody), which needs an elaborate treatment.

The third part is about the Pāli collection. You will learn how to find a document and open it, how to use the viewer, and how to deal with the overall term list. This part is more substantial than the previous one and each chapter is not long. So, it should be read carefully.

The fourth part is about advanced search tools. It has one big chapter about Lucene Finder. This chapter needs a careful read because of the complexity of its functions. Another chapter, about Tokenizer, is short, not because it is simple, but rather it requires the understanding of the previous chapter. The tool itself is quite complicated and needs an exploration by the users themselves.

The fifth part is the rest of the all above. You will learn various tools, like the program's text editor, batch script transformer, and more importantly the text Reader. The Reader, together with its companion tool, Sentence Manager, is an innovative tool that can help the learners read Pāli texts more conveniently. Furthermore, translations can be added to the texts at sentence level by this set of tools.

The last chapter is a short treatment of regular expression. Because I add this search function to the program in various places, some guidance is needed. For the topic itself is big and beyond our main concern, what I can do is just a survival introduction.

My target readers of this manual are those who want to make use of PĀLI PLATFORM to its full capacity, both for learning and researching purpose. Some basic knowledge of Pāli is helpful, particularly the terminology used in the field. For the fundamental of the language, see the books mentioned. Users

outside the sphere of computational technology may skip computer-related technical terms that appear unexplained, or else they can surely find an explanation in Wikipedia.

This manual is a little hastily written. It is a product of one and a half month of my full-time working.³ My intention is to bundle the manual with the program, and release all of them before the end of 2022. The program has been tested and debugged along the way when I have written the book. So, everything should look nearly complete. Yet, errors always lurk somewhere to show up, both in the books and in the program. If you find something, or many things, unusual, please kindly report it to me.⁴

³I usually work 5–6 hours a day, no weekend. I use only one day in a week to connect to the Internet, mostly for updating information and searching for needed materials. Yet, I still work 2–3 hours that day. I choose to make this manual with \LaTeX , despite its laborious process of writing, because it looks authoritative and it is easier and more pleasurable to read.

⁴`jakratep@gmail dot com`

Contents

Preface	iii
Contents	vii
List of Tables	x
List of Figures	xi
I. Essential Starter	1
1. Begin at the beginning	3
1.1. A history of PĀLI PLATFORM	3
1.2. Why I take Pāli seriously?	4
1.3. Features so far	6
1.4. How to run the program	7
1.4.1. Windows	9
1.4.2. GNU/Linux	10
1.4.3. macOS	11
1.5. When things go right	12
1.6. When things go wrong	13
1.7. Download links	14

2. Basic operations and settings	16
2.1. Main window	16
2.2. Common tool bar	17
2.3. Fonts and problems	18
2.4. Pāli input	19
2.5. Minor concerns	21
II. Grammatical Tools	23
3. Dictionaries	24
4. Letters	29
5. Declension table	31
5.1. Pronouns	31
5.2. Nouns/Adjectives	33
5.3. Numbers	34
6. Verbs	36
7. Conjugation table	38
8. Roots	41
9. Prosody	43
9.1. A survival introduction to Pāli prosody	44
9.2. Two types of prosodic patterns	45
9.3. Verse types of <i>mattāvutti</i>	48
9.4. Verse types of <i>vaññāvutti</i>	53
III. Pāli Collection	59
10. Browsing and bookmarking	60
11. Document Finder	64
12. Document viewer	67
13. Simple Lister	72

IV. Advanced Search Tools	78
14. Lucene Finder	79
14.1. Options for indexing	80
14.2. Description of fields	82
14.3. Lucene simple search	83
14.4. Lucene query syntax	85
14.4.1. Using wildcards	86
14.4.2. Using regular expression	86
14.4.3. Using fuzzy query	86
14.4.4. Using proximity	87
14.4.5. Using range	87
14.4.6. Using term boost	88
14.4.7. Using logical operators	89
14.5. Concluding remarks	90
15. Tokenizer	91
V. Miscellaneous Tools	94
16. Pāli Text Editor	95
17. Batch Script Transformer	97
18. Pāli Text Reader	99
19. Sentence Manager	105
20. Quick guide to regular expression	109
About the author	113
Colophon	114

List of Tables

5.1. Expansion of case abbreviations	32
7.1. Tenses and moods	40
9.1. Syllable vs. weight summation	46
9.2. Meter groups used in <i>mattāvutti</i>	46
9.3. Meter groups used in <i>vaññavutti</i>	47
9.4. Syllable vs. weight grouping	48
9.5. Symbols in verse formulas	48
9.6. Verse types of <i>mattāvutti</i>	49
9.7. Verse types of <i>vaññavutti</i>	54
12.1. Transformation rules of Thai script	71
14.1. Fields used in Lucene Finder	82
20.1. Some uses of regular expression	110

List of Figures

1.1. PaliPlatform2's folder structure	8
1.2. Main window of PĀLI PLATFORM 2	12
1.3. Information of runtime environment	13
1.4. An example of the JVM's error message	14
2.1. The common local tool bar	17
2.2. Settings for Pāli input	20
3.1. Menu Grammar	24
3.2. Dictionaries window	25
3.3. Default dictionary selection	26
3.4. Dictionaries' wildcard search	27
3.5. Dictionaries' meaning search	28
3.6. Dictionaries' result opened in an editor	28
4.1. Letters window	30
5.1. Declension table of a pronoun	32
5.2. Declension of <i>tumha</i> against term list	33
5.3. Declension of <i>vimutti</i> against term list	33
5.4. Declension of <i>sara</i>	35
5.5. Declension of 100250	35

6.1. Result of main verbs when search ‘vim’ and select <i>vimuccati</i>	36
6.2. Options of other verb forms	37
7.1. Aorist forms of <i>pacati</i>	39
7.2. Masculine <i>ta</i> forms of <i>pacati</i>	40
8.1. Roots window	42
9.1. Prosody window	43
9.2. Meter calculation in the program’s editor	45
9.3. Analysis of <i>Ariyā</i> verse type	51
9.4. Example of an over-required case	52
9.5. Analysis of <i>Tanumajjhā</i> verse type in edit mode	57
10.1. TOC Tree window	61
10.2. TOC Tree window at text level	62
10.3. Bookmarks window	63
11.1. Searching Dhammapada in Document Finder	65
11.2. Searching with a wildcard in Document Finder	65
11.3. Content searching in Document Finder	66
12.1. Document viewer in its full form	68
12.2. Document viewer with Quick Dictionary	69
12.3. Document viewer displaying in Myanmar script	70
12.4. General Settings of script transformation	70
13.1. Simple Lister window with term summary	73
13.2. Top longest terms in Simple Lister	75
13.3. Filter by meter ‘gggl’ in Simple Lister	76
13.4. Filter by meter ‘2221’ in Simple Lister	76
13.5. Grouping by the first letter in Simple Lister	77
13.6. Grouping by the last 2 letters in Simple Lister	77
14.1. Apache Lucene’s logo	79
14.2. Options for building Lucene index	80
14.3. Simple one-term search in Lucene Finder	83
14.4. Simple two-term search in Lucene Finder	84
14.5. Proximity search in Lucene Finder	87
14.6. Range search in Lucene Finder	88
14.7. Term boosting in Lucene Finder	89

15.1. Tokenizer window in full	92
16.1. Text-processing tools in Pāli Text Editor	96
17.1. Batch Script Transformer window	97
18.1. A sentence with translations in Pāli Text Reader	101
18.2. Detail mode in Pāli Text Reader	103
18.3. A use of edit in Pāli Text Reader	103
19.1. Sentences tab in Sentence Manager	106
19.2. Translation Variants tab in Sentence Manager .	107
19.3. Merger tab in Sentence Manager	108

Part I.

Essential Starter

Begin at the beginning

In this starting chapter, I will tell you a short story of the development of the program. At the end, we will learn how to make it run in your computer.

1.1. A history of PĀLI PLATFORM

Originally, I had an idea to make a program that can search words in the Pāli canon effectively, at least more effective than the programs we had at the time. That was around 2–3 decades ago, when I was interested in Buddhism after I finished my undergrad engineering program. At the time, it was too difficult to do with limited technology and data available, and I had many more pressing things to do. So, the idea just got lingered in my mind since.

After I found that we have a digital version of the whole Pāli collection, distributed by Vipassana Research Institute (VRI) via tipitaka.org, the possibility of the project began to light up. But I still had more important things to do. Once I quit my job, I had an opportunity to continue my studies. Then I finished a master program in Software Engineering. The subject of Information and Retrieval (IR) was the main concern of that study.¹

¹Meanwhile, I also studied Buddhism and Philosophy at another univer-

1. Begin at the beginning

Nothing still happened yet. My life changed. In 2010, I became a monk and have been enjoying the peaceful life. The idea has been neglected. At some point, I thought it is better for me to deepen my studies. Then in 2018, I finished a PhD program in Religious Studies. After that, I had time and nothing important to pursue. So, I started the PĀLI PLATFORM project, and finished its first version in January 2020, by one full year of development.

The first PĀLI PLATFORM was written in Java 8. The main reasons for using Java are: First, Java executables can run in all major operating systems, as its motto says, “Write once, run anywhere.” And second, *Apache Lucene*, an excellent IR system, is written in Java.

There was a limitation at the time. Because I often traveled from place to place, mostly by foot. I had to keep my belongings minimal, able to carry along in a bag or two. From that limitation, the first release of PĀLI PLATFORM was totally developed in Raspberry Pi 3 Model B, a credit-card-sized computer, with 5-inch LCD display. As a result, it forced me to use *Swing*, an old, a little outdated, Java user interface (UI) toolkit.²

After I made use of PĀLI PLATFORM 1 to write two Pāli learning books. I have rewritten the program again from the ground up, using Java 11 and JavaFX UI (around 10% of code was reused). Doing the same thing twice can get us some insight. The result of my effort this time is really rewarding. I just love it, and I hope students of Pāli would feel in the same way, and use the program as their study/research platform.

1.2. Why I take Pāli seriously?

Let me use this section to explain that why I spent my time and effort to do my Pāli related projects. As a matter of fact, I neither work in the software industry nor academia. So, I am not interested in producing programs to sell, nor pursuing academic positions. I just live my life peacefully, mostly in solitude (and a kind of destitution). That makes me have

sity, and finished it in the following semester.

²In fact, JavaFX was available. But for ARM devices, it was incomplete at the time. Another reason was the Swing UI is really well-documented, easy to learn and use.

plenty of time to do what I think it is worth spending the day to day hours. This means the projects is closely related to my religious life, but not in the way Buddhists expect.

I do not think mastering Pāli will lead anybody to enlightenment. Many liberated ones do not know a word of Pāli. Likewise, knowing everything stated in the Pāli canon cannot lead anyone to liberation. The all Pāli related matters are about intellectual enterprise, as well as political strategy. This may raise a big issue of debate, but this is not a good place to go into that. In my view, most of Pāli matters are power related and about thought control³, little to do with real liberation.

It is undeniable that the Pāli canon constitutes the bedrock of all Theravāda traditions and cultures. Most cultural values in Theravāda countries are derived from the canon. The only key to 'decipher' the meaningful messages from the text is Pāli language. The word 'decipher' is tricky. It sounds like there is the only way to decode the messages. In other words, you cannot read the text in whatever way you want. There is the only way, and the right way is maintained by the tradition. That is an illusion. The illusion that the tradition (read, those who gets the power) tries to implant in the Buddhists' mind.

I think, that is why learning Pāli in the traditional way is extremely (but unnecessarily) difficult. Those who can pass the laborious process are counted as 'authority.' Then they can determine what should be read from the canon. Outside of that scope, it is regarded as unorthodox, or even blasphemous. By such a system of learning, the ability of reasoning is impaired, if not destroyed altogether.

From my engineering background, I was trained to think that using a better tool brings a better result with less effort. With a good learning tool, now those who have some effort (not that much as the traditional students) and perseverance can become 'authority' themselves. They can investigate why the tradition gives certain explanations. And they can use reasons whether to believe or embrace those positions or not.

That can emancipate Buddhists from the yoke of surrepti-

³Some may argue it is also about knowledge. That is partly true. The main purpose of such knowledge is a kind of power, ability to control and maintain the superior position. We have to understand it nonetheless, to guard ourselves against that kind of manipulation. So, for me, understanding of Pāli leads to knowledge in this manner.

1. *Begin at the beginning*

tious manipulation of power through religion. That is what I have been trying to do. I have built a good tool, introduced an effective way to learn the language, and encouraged the use of good reasoning to assess whether certain beliefs should be taken or not.

That should be enough for this uneasy matter. I will never raise the issue again. For serious readers, please read further Part 1 of *Pāli for New Learners*, Book II.

1.3. Features so far

The program can do many things related to Pāli studies. Before we go into detail of these in our course, let us see the big picture. I divide the groups of features into three:

(1) Pāli text collection Most of the text compiled in the Chatṭha Saṅgāyanā⁴ is here. In addition, the users can add their own text to the collection, in both XML⁵ and plain text format, known as the *Extra*. The text can be accessed easily by a tree of grouped contents. I call this *TOC Tree*. Or a specific text can be searched by various tools provided, from very simple to mind-bogglingly complex.

(2) Essential grammatical tools New Pāli learners will find many helpful resources here. They have indispensable Pāli dictionaries to consult with, a bit old, but still useful. They can learn various Pāli scripts, including Roman, Devanagari, Sinhalese, Khmer, Myanmar, and Thai. They can learn inflections of nouns and verbs, by typical paradigms, as well as by experimenting with generic rules. The number generator can produce Pāli numerals up to 6 digits. We also have a table of Pāli roots as listed in *Saddanīti Dhātumālā*. Advanced students can find learning Pāli prosody is easy and fun with a powerful stanza analyzer. (We even have meter search in Simple Lister and Tokenizer to facilitate prosodic composition.)

⁴The collection is known as Chatṭha Saṅgāyana CD (CSCD), distributed by Vipassana Research Institute (VRI), tipitaka.org.

⁵The format is conformed to the collection's.

(3) Powerful search tools Searching is the primary goal of the program. So, we can see various approaches for finding things here. At basic level, texts in the collection can be searched by Document Finder, both by text name (as shown in TOC Tree) and by text content (brute full text search). For a more advanced search, Lucene Finder can be used with its full capacity. The users can build Lucene's index with options adjusted as needed. For a custom index builder, the users can use Tokenizer instead, the IR module developed by the programmer. This allows the users can add any number of documents, including the Extra, to the index. The search function of Tokenizer may be not so powerful as that of Lucene, but it is good enough for a specific search and it can list all terms in the index (not implemented in Lucene Finder). Or, if the users just want to see the list of all terms in the collection. The pre-built list can be found in Simple Lister. A specific term can be searched here.

(4) Unique and powerful accessories The program also incorporates a number of general tools, such as Pāli Text Editor (with several helpful functions), Pāli Text Reader with Sentence Manager (with ability to help text reading and to add translations), and batch script transformer (to convert a bunch of text files between Roman and other scripts). Furthermore, I introduce new ways of entering Roman Pāli character, which are more comfortable than the old methods.

1.4. How to run the program

Now we come to the technical instruction. If you have not got the program yet, go to paliplatform.blogspot.com⁶ and download it from the link provided. The file you get looks something like this: PaliPlatform2.0-?.zip

The ? stands for its subversion, e.g. RC1 (Release Candidate 1) or whatever. The file is packed in zip format. There are two options: if you use Windows, you should download the full version with JRE bundled (around 150 MB in size); if you use Linux or macOS (or Windows, for those who plan to install

⁶This is the only entry point I maintain. It can be changed in the future.

1. Begin at the beginning

or add JRE by themselves), you can choose to download the lesser version (around 90 MB) instead. Any common unpacker should be able to explode the file.

Once you unpack the program to a writable place, you will see a folder named PaliPlatform2 with the structure depicted in Figure 1.1 (What is really shown in your computer is likely different, but the content is the same).

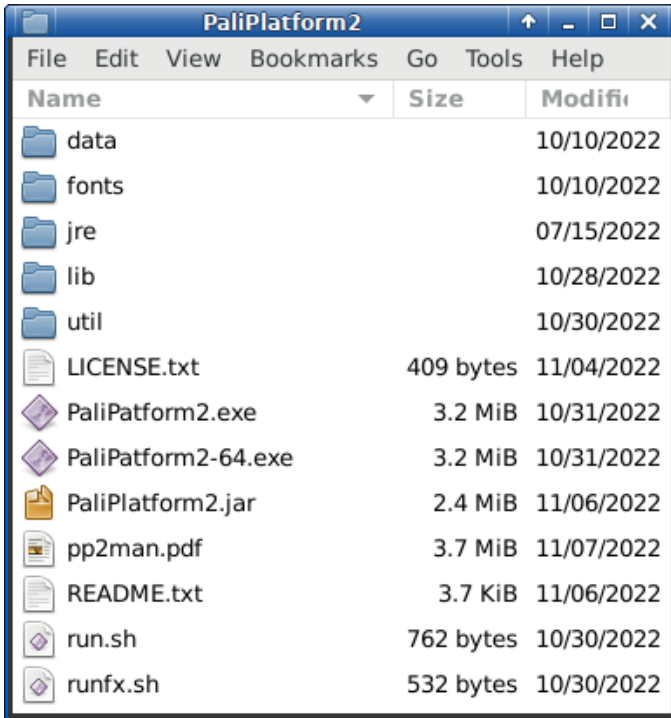


Figure 1.1.: PaliPlatform2's folder structure

Now you are ready to run the program. The principle is simple: The *Java Virtual Machine (JVM)*, version 11 or newer, will pick up the executable and run it. The JVM is a part of *Java Development Kit (JDK)*. For the users, you do not need the whole JDK, just the part called *Java Runtime Environment (JRE)* is enough to run the program. The process can be different from platform to platform as shown below. The

required computer spec is low, 2-GB RAM is safe, but more is better. I even use an old 32-bit laptop in the whole process of development.

1.4.1. Windows

Even though my developing environment is Linux, I make it easy to run in Windows (7 or newer). For Windows, I propose three use contexts:

(1) Using the bundled JRE This is the easiest way. The required JRE is already shipped with the program under `jre` folder. The only action you need is double-clicking `PaliPlatform2.exe` to make the program start. That is all. You also can make a desktop shortcut from this exe file.



NOTE: To ensure that every machine can run the program, I bundled only 32-bit JRE. For those who have a 64-bit computer, you may need to replace the existing JRE with 64-bit version (see download links below), then select `PaliPlatform2-64.exe` instead.

Replacing the existing JRE can also be done in the case that you want to use your own JRE or to update it to the latest version. To do so, delete the existing `jre` folder, unpack the JRE in the program's root, rename it to `jre`. Then run `PaliPlatform2.exe` or `PaliPlatform2-64.exe` for 64-bit JRE.

(2) Using locally installed JRE There is a good chance that your computer has a JRE installed. However, not every JRE is usable. You need a JRE with JavaFX included. There are some vendors who provide this (see links below). When you download the JRE, make sure you get a full JRE or JRE with JavaFX suitable to your machine. Once the JRE is installed into the system, you can run the program by double-clicking `PaliPlatform.jar`, or open it (by a right-click) with `OpenJDK platform` binary or the like. You may test this first if your existing JRE is usable.

1. *Begin at the beginning*

(3) Using manual method This approach does not use the exe file but use the bundled JRE, and you have to do it by hand. First open a console terminal (command prompt)⁷ at the program's root, then enter this:

```
» jre\bin\java -p .;lib -m paliplatform/paliplatform.PaliPlatform
```

If you have a full JRE installed, you can use java directly, hence:

```
» java -p .;lib -m paliplatform/paliplatform.PaliPlatform
```

1.4.2. GNU/Linux

This is my beloved operating system (OS). Although the number of Linux's users is less than Windows and macOS, Linux is very important OS nowadays. It liberates us. For Linux's users, mostly power users, it looks trivial to tell what to do to run a Java program. So, I will leave out some details.

(1) Using an installed JRE There are two ways to install a JRE into your system. First, you can install Java packages from your Linux distro's repository. For example, if you use Debian-based OS, enter this command:⁸

```
» apt-get install openjdk-11-jre openjfx
```


Second, you can install JRE with JavaFX from other providers (see download links below). Once, you have JRE and JavaFX in your system, open a console at the program's directory, and type this launcher script:

```
» ./run.sh
```

If a separate JavaFX is used, you have to edit the launcher script accordingly.

(2) Using a custom JRE If you do not want to install Java into your system, you can just download an archive (zip or gz) version and unpack it to the program's directory. Rename it to jrefx. Then type this launcher script:

```
» ./runfx.sh
```

⁷In Windows 10, you can open a terminal at any place in File Explorer by using the File menu. In Windows 7, I find it more difficult. You have to search for command prompt or cmd and open it, or hit -R and enter cmd, then make your way to the target directory by dir command.

⁸If you already have a higher version of Java installed, just openjfx is enough.

If you name the directory otherwise, editing the script is needed. The script works only for full JRE with JavaFX included.

(3) Using manual method If you prefer doing things by hand, try this:

- If you have JRE with JavaFX included, type this at the program's root:

```
» jre_somewhere/bin/java -jar PaliPlatform2.jar &
```

- If you have JRE and JavaFX separately, use this instead:

```
» jre_somewhere/bin/java -p ".:lib:javafxdir/lib" -  
-m paliplatform/paliplatform.PaliPlatform &
```

Notes: It is important to run these commands from the program's root directory. And it is advisable to set `JAVA_HOME` and export it to `PATH`. Then, you just type "java"

(4) Creating desktop launcher In directory `util`, there are helper scripts to create a Linux desktop launcher for the program. There are two scripts that make use of `run.sh` and `runfx.sh`. For more information, please read the instruction there.

1.4.3. macOS

I have never owned or used any of Apple's devices. For the sake of testing, however, I spent many hours to have macOS Catalina (10.15.7) installed in my Linux's virtual machine. Here are successful methods.

(1) Using an installed JRE This is the easiest way. First, you have to download a full JRE (with JavaFX) installer from any provider (see download links below), and install it in the system.⁹ Then you can make the program run by double-clicking `PaliPlatform2.jar`, or open it (by a right-click) with Jar Launcher. Or, you can open a terminal in the program's directory and type this:

```
» ./run.sh
```

⁹The installer packages can be either of `.pkg` or `.dmg` type.

1. Begin at the beginning

(2) Using a custom JRE If you do not want to mess up with the system, you can download an archive package instead and unpack it in the program's directory. Rename it to `jrefx`, then type this:

```
» ./runfx.sh
```

(3) Using manual method Since macOS is a kind of UNIX-like OS, the Linux's methods described above can be applied in most cases. However, Linux desktop launcher does not work in macOS. You have to create an alias of `PaliPlatform2.jar` and move it to your Desktop.

1.5. When things go right

Once you succeed to make the program run in your computer. You will see the main window as shown in Figure 1.2.

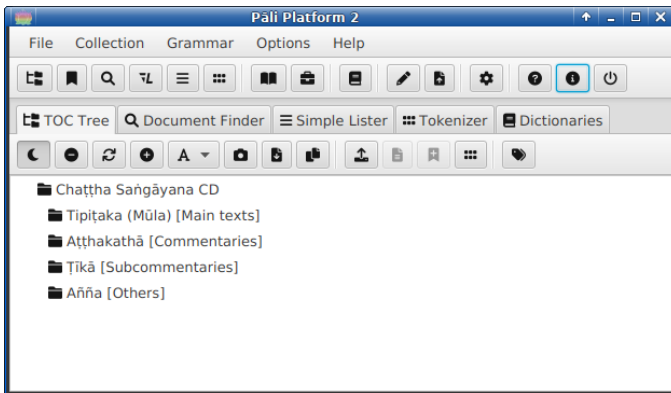


Figure 1.2.: Main window of PĀLI PLATFORM 2

You can check your runtime environment by choosing `Help>About` in the menu, or clicking **i** button. The system's information will show on the left of About window (see Figure 1.3).

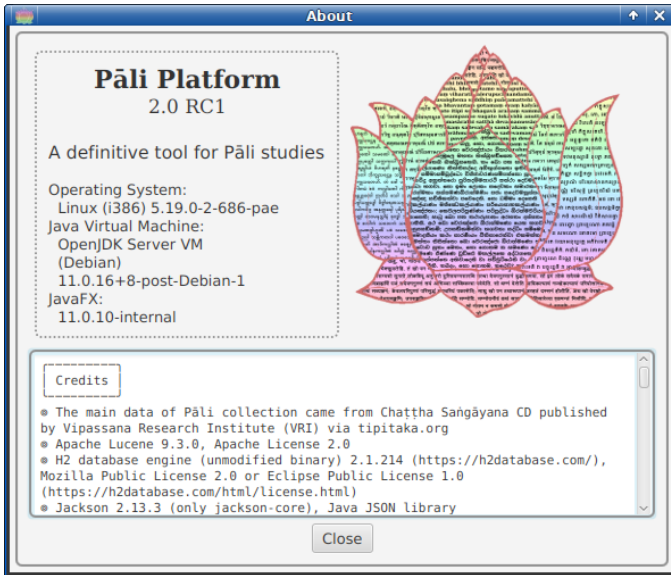


Figure 1.3.: Information of runtime environment

1.6. When things go wrong

Even though the program was well-tested, many bugs are still waiting. They can show up when encountering unexpected use cases. If the users find that the program behaves in an unusual way or even crashes, please report the bugs.

To see error messages fired by the JVM, you have to run the program with a console, i.e., you have to run the program manually and leave the terminal opened. Error messages may look unintelligible to the users, as shown in Figure 1.4, but they indeed give useful information to trace the bugs' causes.

When this occurs to you, please record the symptom and save the message, then send it to me.¹⁰ I will find the causes and fix the bugs.

¹⁰ jakratep at gmail dot com

1. *Begin at the beginning*

```
[java] Exception in Application init method
[java] Exception in thread "main" java.lang.reflect.InvocationTargetExcepti
on
[java]   at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke
e0(Native Method)
[java]   at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke
e(NativeMethodAccessorImpl.java:62)
[java]   at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.i
nvoke(DelegatingMethodAccessorImpl.java:43)
[java]   at java.base/java.lang.reflect.Method.invoke(Method.java:566)
[java]   at java.base/sun.launcher.LauncherHelper$FXHelper.main(LauncherH
elper.java:1071)
[java] Caused by: java.lang.RuntimeException: Exception in Application init
method
[java]   at javafx.graphics/com.sun.javafx.application.LauncherImpl.launc
hApplication1(LauncherImpl.java:895)
[java]   at javafx.graphics/com.sun.javafx.application.LauncherImpl.lambd
a$launchApplication$2(LauncherImpl.java:195)
[java]   at java.base/java.lang.Thread.run(Thread.java:829)
[java] Caused by: java.sql.SQLException: No suitable driver found for
[java]   at java.sql/java.sql.DriverManager.getConnection(DriverManager.j
ava:702)
[java]   at java.sql/java.sql.DriverManager.getConnection(DriverManager.j
ava:228)
```

Figure 1.4.: An example of the JVM's error message

1.7. Download links

Here are some useful links related to the program's execution.

(1) BellSoft Liberica JDK BellSoft provides installers of JRE with JavaFX for major platforms. This can be easy for new users who just want to install what is needed. When downloading, select JRE Full.

<https://bell-sw.com/pages/downloads/#/java-11-lts>

(2) Azul Zulu JDK Azul Systems also has JRE with JavaFX (32- and 64-bit) for many platforms, but few have installers. I used JRE from this vendor in the program's bundle. When downloading, select JRE FX.

<https://www.azul.com/downloads/?version=java-11-lts-&package=jre-fx>

(3) Eclipse Temurin JDK This provider does not provide JavaFX, just JRE. This may be suitable for a competent user who knows how to combine the two components together. (It is not that difficult, particularly when you use Linux. See the manual method described above.)

<https://adoptium.net/temurin/releases/?version=11>

(4) Gluon's JavaFX This is the official site of JavaFX. You rarely need this, except if you want the newest version of it (really unnecessary for us). When downloading, select SDK binary package, not jmods.

<https://gluonhq.com/products/javafx>

Basic operations and settings

Now, I suppose that the reader can run the program successfully and get the first screen. Unlike PĀLI PLATFORM 1 which integrates all working areas in one window, this version uses multiple windows approach. So, you will see many windows doing various jobs. There are two kinds of window: singleton and multi-instance. The former has only one instance in the program's lifetime, while the later can have multiple instances. Let us start with the main window.

2.1. Main window

The main window is a mandatory singleton. You only have one main window when you run the program. You can start the program multiple times, however, to create multiple running instances. But, it is unnecessary, and not recommended, to do so. Therefore, you are supposed to have only one main window.

There are three parts in the main window: the menu bar, the main tool bar, and tabs of some working areas (see Figure 1.2). The first two parts are familiar to most computer users. Every button in the tool bar has its counterpart in the menu. They are selected to show here because they may be used a lot. We will see the use of each command in the menu in due course.

The third part is the big area below those two. There are

fixed tabs of the most-used 5 modules: TOC Tree, Document Finder, Simple Lister, Tokenizer, and Dictionaries. These tabs are always there, no more no less, but you can reorder them as you want. Each tab can be opened as new multiple windows using the menu or the tool bar. When a new window is created, it works separately from the main tab area. For referencing, therefore, we will call these 5 tabs as the main TOC Tree, the main Document Finder, and so on.

2.2. Common tool bar

Every working tab, also every opened window, has its own tool bar that has a part in common (see Figure 2.1).

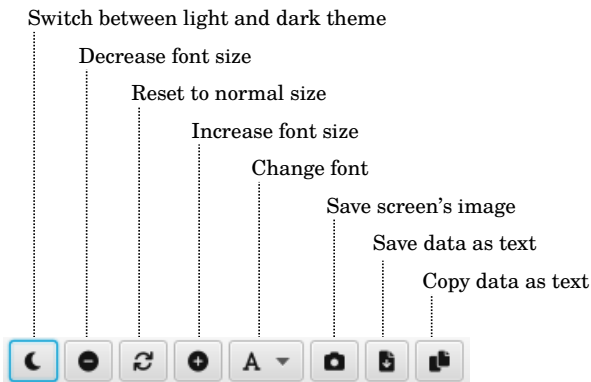


Figure 2.1.: The common local tool bar

When you change theme by the tool bar, it changes only that instance. If you click this button in the main tabs, the whole main window is changed. This change is not persistent. If you want permanent theme change, use menu `Options>Global theme` instead. Other buttons are intuitively understandable, but read more about font in its section below. Another common button you can see here and there is help (🔍) button. When clicked, the related help of that part will show up.

2.3. Fonts and problems

The users will not feel the significance of fonts and their problems until they read or edit a Pāli text. However, you should know this before you find out yourself that things do not always go as you expect.

Fonts used in the program have two kinds: embedded and external. The former has nothing to do with the users. You may be able to select some of them, but they cannot be changed or removed. This includes *DejaVu*¹ font family (Serif, Sans Serif, and Mono space) and icon fonts. So, when applicable, you can always choose either *DejaVu Serif* or *DejaVu Sans* or *DejaVu Sans Mono*. These fonts should have no problem in most running environment.

Problems mostly come from other scripts than Roman. That is why external fonts is needed. The idea is simple: put your font files (TTF) in folder `fonts` at the program's root and restart the program. You should add two files: one for regular face, another for bold face. If only regular face is added, you will see only regular text, no bold, in Pāli documents.

Not every font is usable, however. For displaying Pāli Roman, you must have a Unicode font with *Latin Extended Additional* block (not every Unicode font has it). I added *Times Ext Roman*² as an example.

For other scripts, I added *Arundina*³ font for Thai, and *Noto*⁴ fonts for the others. Here is a warning: Not all fonts work as expected. I always have problems with Myanmar fonts. The provided Noto Myanmar works fine with my current Linux but not the former one. When I tested with Windows, the Myanmar font did not work in Windows 7, but worked without problem in Windows 10. Fonts of other scripts may look weird in some machines.

Here are some solutions when problems related to external fonts occur:

(1) Just delete the external fonts If your system has proper fonts installed, you can delete fonts provided in the font folder (or

¹<https://dejavu-fonts.github.io>

²This font, copyright of Monotype Corporation, was shipped in CSCD.

³<https://github.com/tlwg/fonts-arundina>


⁴<https://github.com/notofonts/noto-fonts>


move them away). Then, the program will use a system font instead as a fallback, currently set to *Sans*.


(2) Try different font versions For example, some versions of Noto fonts work in some systems, some do not, or behave oddly. Try another version of them may help.


(3) Use alternative fonts I found that Padauk, Myanmar3, and MyMyanmar font work fine in Windows 7. KhmerOS fonts also work in various contexts. LKLUG (Sinhala) font also works well in most Linux.

2.4. Pāli input

Typing in Pāli characters is used throughout the program. So, it is essential that the users have to learn the input method. It is easy, but needs some familiarity. In text fields and text areas that expect Pāli Roman characters, there are 3 modes of input indicated by their symbol: Regular mode (a►a), Unused-characters mode (x►ā), and Composite mode (ā►ā). These three modes can be switched circularly by clicking the symbol button or pressing Ctrl-Space. Pāli input also has detailed settings in Settings window, invoked by menu Options>Settings or  button in the main tool bar.

 *Regular mode* is the normal state of your computer. As the symbol says, when you type ‘a’ you simply get ‘a’. You can type in English or other languages supported by your system in this mode.

 *Unused-characters mode* is a unique and innovative method. This mode utilizes unused English characters for some Pāli letters with diacritics. Here are the program’s default mapping: x = ā, X = ī, w = ṃ, W = ū, F = ñ, f = ñ̄, q = ṭ, Q = ḍ, z = ṇ, Z = ḷ. That is why the symbol says thus: you get ‘ā’ when you hit ‘x’, for example. This mode includes two special keys for making a character under cursor uppercase (<) and lowercase (>). All these keys can be set in Settings (see Figure 2.2).

 *Composite mode* is the most intuitive way to enter a diacritic mark. Here are the program’s presets: ~ = tilde,

2. Basic operations and settings

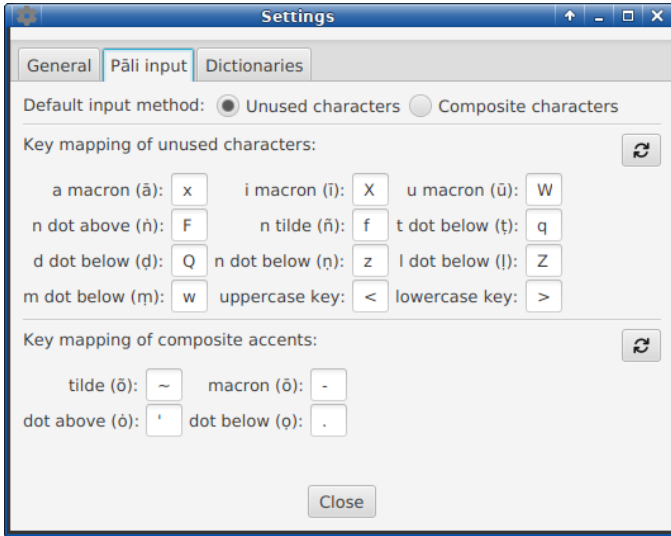


Figure 2.2.: Settings for Pāli input

- = macron, ' = dot above, . = dot below (see also Figure 2.2). For example, when you need 'ā' you type 'a' followed by '-' immediately. The mapping can be set otherwise in the program's settings. This mode is a little more expensive than the previous one. You need two strokes rather than one, but you do not need to remember the character mapping.

The users can set their preferred method as the default in the program's settings (see Figure 2.2).



NOTE: All user's preferences, such as the global theme, bookmarks, and those in Settings, are saved in an external property file, named PaliPlatform2.properties. This file may exist in the user's home directory or the program's root or somewhere else depending on how the program is invoked. If you mess up with the settings, you can return to the original state by just deleting the property file and restart the program.

2.5. Minor concerns

There are some miscellaneous points that are worth mentioning here, before you make a real use of the program.


(1) Drag-and-drop The program supports the drag-and-drop action in various places. In some tasks, it is mandatory, such as adding files into a Tokenizer window. In some, it is just a convenient tool, for example, a file can be dragged from the system's explorer and drop into the program's editor, or a text portion can be dragged from a system's editor and drop into a Dictionaries window, or a text portion in the Dictionaries' result can be dragged to somewhere else.

There are many possibilities of drag-and-drop. Some look obvious. Some seem likely applicable, but it cannot be done (because of a technical limitation). It will be too fussy to list all of them here. Learning by using is better. If any drag-and-drop action does not work, using copy-and-paste, if applicable, can achieve the same goal in most cases.

(2) Context menus Context menus are menus that pop up when you right-click (or left-click in some places) at a certain object. These menus are context-dependent. You have to test and learn by yourself. In many cases, context menus provide more solid functions, than drag-and-drop does.

(3) Sortable tables There are a number of tabular results produced by the program. Most of those tables are sortable. It can be done by clicking on the table header of the target columns. The mode of sorting is changed circularly: ascending, descending, and unsorted. A triangle in the header will show up accordingly. Some tables, to which sorting makes no sense, cannot be sorted, though. There are some things to keep in mind. First, most tables are already sorted by a pre-selected criterion. Using a custom sort can disrupt the preset order, in the way that mode of sorting may be shown incorrectly. And second, the sorting is done only on the existing data in the tables. No new retrieval or calculation is applied.

2. Basic operations and settings

(4) Tooltips When you are stuck in the using, try  button first. If the help is not available, try reading tooltips by hovering the mouse over an unknown button. Tooltips can be seen in other places as well, such as in a list or table that has truncated data entries.

Part II.

Grammatical Tools

Dictionaries

We will start with grammatical tools, because they are not complex and rather easy to use and understand. All of these tools can be invoked by menu Grammar (Figure 3.1). Only Dictionaries has its button (📖) in the main tool bar.

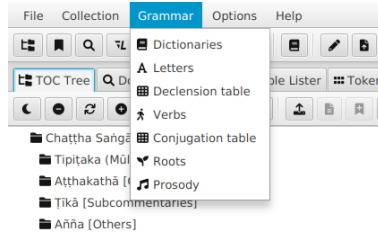


Figure 3.1.: Menu Grammar

Dictionaries is the most used tool. It occupies one tab in the main window, and it can be opened as separate windows, as many as you want. A Dictionaries window is shown in Figure 3.2.

There are four dictionaries you can search in this module.

(1) Concise Pāli-English Dictionary (CPED)¹ This dictionary holds a significant position in the program. It is the

¹By Ven. A. P. Buddhadatta, available online at <https://www.budsas.org/ebud/dict-pe/index.htm>

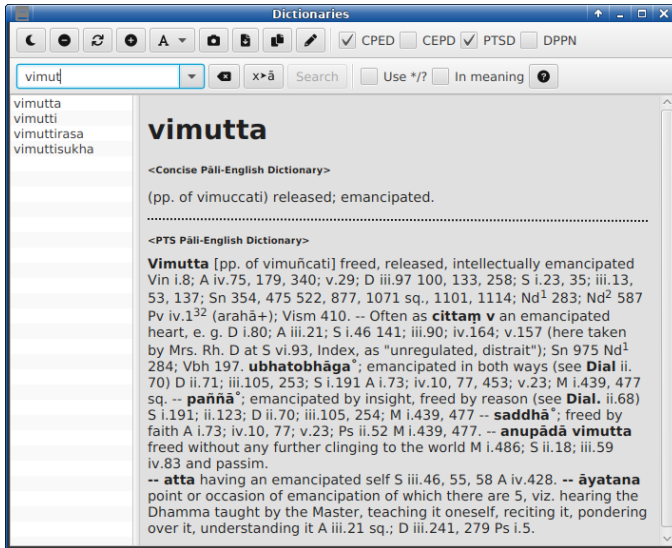


Figure 3.2.: Dictionaries window

only one that is structured and used widely in various contexts. I also made a few corrections and added some entries.

(2) **Concise English-Pāli Dictionary (CEPD)**² This may be less used. Sometimes it can give you a word that you do not know it in Pāli, but you know its close English words.

(3) **The Pali Text Society's Pali-English Dictionary (PTSD)**³ A bit old, this dictionary is still indispensable and often gives some insight to the words we study.

(4) **Dictionary of Pāli Proper Names (DPPN)**⁴ You may seldom use this one. It gives detailed information about Pāli names, not directly the explanation of the words.

The inclusion of these can be set by their check boxes in the tool bar. For persistent setting, consider using Settings window as shown in Figure 3.3.

²By Ven. A. P. Buddhadatta, available online at <https://www.budsas.org/ebud/dict-ep/index.htm>

³By T. W. Rhys Davids and W. Stede (1921–25), available online at <https://dsal.uchicago.edu/dictionaries/pali>

⁴By Gunapala Piyasena Malalasekera, available online at http://www.palikanon.com/english/pali_names/dic_idx.html

3. Dictionaries

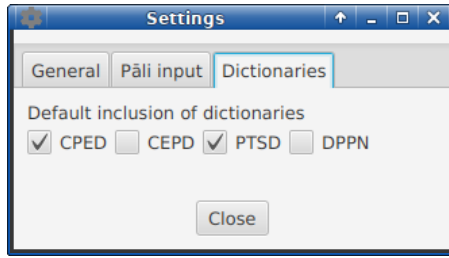


Figure 3.3.: Default dictionary selection

Searching in Dictionaries can be of three use cases as follows:

1. Simple search

Simple search is fast and simple. You enter some characters, the relevant result will be listed immediately, and the topmost entry is shown in the result area. Figure 3.2 shows the result of a simple search.

2. Wildcard search

Sometimes you have a vague idea what the word looks like. You can use wildcard mode by checking **Use */?**. In this mode you can use **?** to stand for any one character and ***** to stand for any characters (including none). Figure 3.4 shows the result of searching ***mutt?**. Remember that in wildcard mode the result does not come up immediately. You have to press Search button or hit the Enter key.

As shown in the picture, ***** can mean zero character, and **?** always represents one character. So, we also see *mutta*, *muttā*, and *mutti* in the result list. In the result of CPED, if the selected term is declinable (either noun or adj.), we will see **Show declension**. When we click on this box-button, a declension window of that term will be opened.

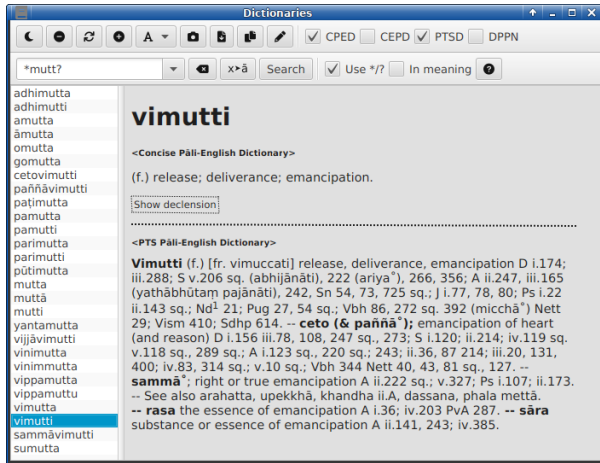



Figure 3.4.: Dictionaries' wildcard search

3. Meaning search

Yet sometimes you have no idea what the word should be. In this case, searching in dictionaries' meanings can help. Figure 3.5 shows the result of searching 'emancipate.' In this mode, using wildcards is not allowed, and do not forget to press Search or hit Enter to submit the query.

There is a technical limitation in highlighting the search result, when you search in meanings. Also, you cannot search further in the display result. To do this, you have to open the result in a text editor, and do the search there. This can be done by hitting  button in the tool bar. Then the result will be opened in the program's editor. Then the users can use search function in the editor, as shown in Figure 3.6.

3. Dictionaries

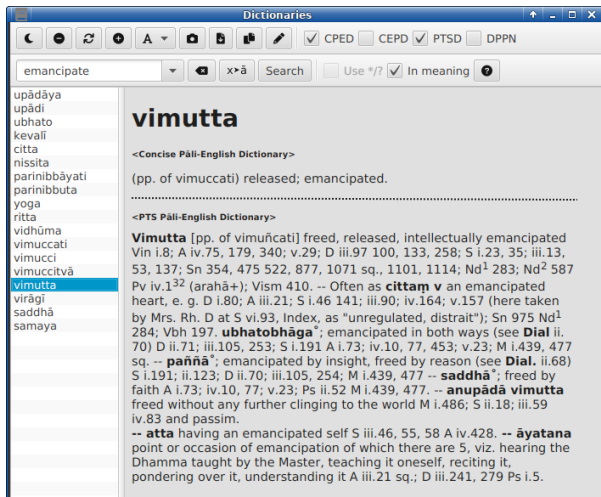


Figure 3.5.: Dictionaries' meaning search

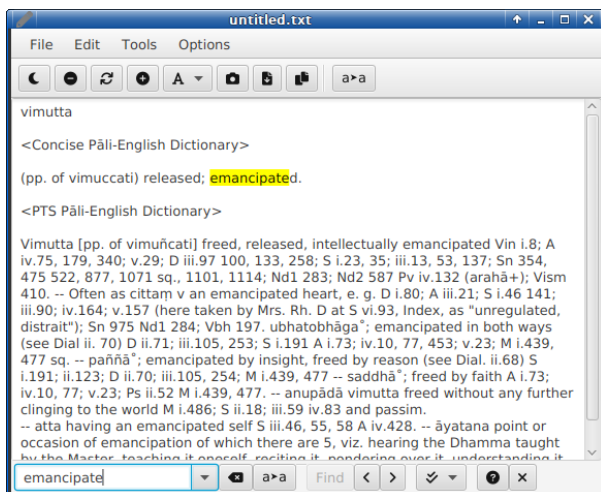





Figure 3.6.: Dictionaries' result opened in an editor

4

Letters

With a good tool, learning Pāli alphabet can be an enjoyable experience. The Letters window can be opened by selecting menu Grammar>Letters. In its tool bar, there are additional buttons:

- E/P** Switch between English and Pāli technical terms
-  Clear highlights and selections
-  Select a script language
-  Open/close typing test area

The display of script languages other than Roman, i.e., Devanagari, Khmer, Myanmar, Sinhala, and Thai, depends on external fonts loaded when the program started. If any problem occurs, read the instruction in Section 2.3. Other operations look obvious. So, I leave the detail to the user to find out.

4. Letters

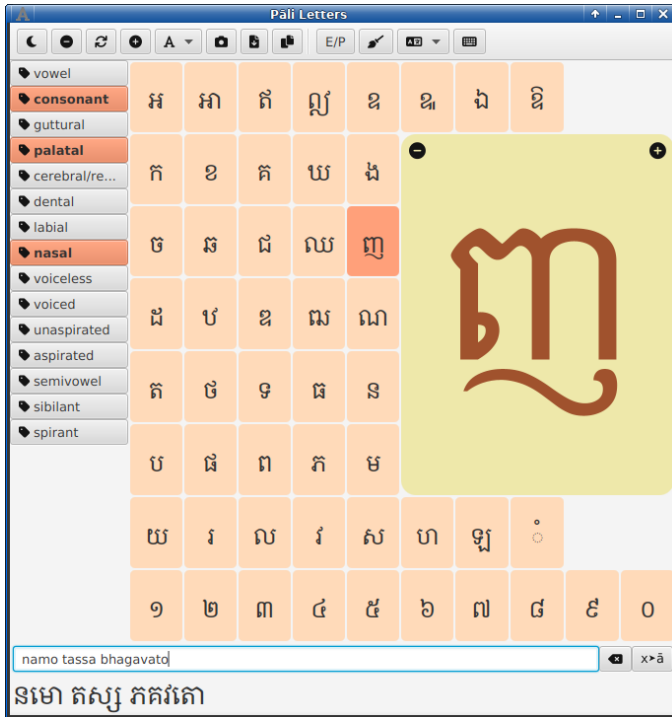


Figure 4.1.: Letters window

Declension table

This is one of the most useful tool for new learners. Traditional students have to remember many of these tables. While remembering some key paradigms is still important in learning process, this tool can enhance your ability to search and experiment in just a few clicks. The Declension window can be opened by menu Grammar>Declension table. The results of Declension table can be grouped into three: nouns and adjectives, pronouns, and numbers.

5.1. Pronouns

If you are new to this window, I suggest you select Pronouns first because the pronoun list is finite. You will see something like Figure 5.1. The result of declension tables will show one gender at a time. If there are multiple genders to be declined, you can choose whether m (masculine), f (feminine), or nt (neuter). Short meaning of each pronoun can also be added when this option is selected.

For new students, the expansion of abbreviations used in the table, as shown in Table 5.1, may do some help.

An amazing feature of this tool is that you can check the declined terms against term list in the collection. When ☰ button is hit, you will see the right pane opened (see Figure 5.2). This may take a little time in the first load. In the list, terms

5. Declension table

	Case	Singular	Plural
1	nom.	tvam, tuvam	tumhe, vo
2	acc.	tavam, tam, tvam, tuvam	tumhakaṃ, tumhe, vo
3	ins.	tayā, tvayā, te	tumhehi, tumhebbhi, vo
4	dat.	tumhaṃ, tava, tuyhaṃ, te	tumhaṃ, tumhakaṃ, vo
5	abl.	tayā	tumhehi, tumhebbhi
6	gen.	tumhaṃ, tava, tuyhaṃ, te	tumhaṃ, tumhakaṃ, vo
7	loc.	tayi, tvayi	tumhesu
ā	voc.		

Figure 5.1.: Declension table of a pronoun

Table 5.1.: Expansion of case abbreviations

1	paṭhamā	nominative case
2	dutiya	accusative case
3	tatiya	instrumental case
4	catutthī	dative case
5	pañcamī	ablative case
6	chaṭṭhī	genitive case
7	sattamī	locative case
ā	ālapana	vocative case

in the table will be sorted by their frequency. Then, you can know which forms are actually used in the texts. Remember that one declined form of a Pāli word can come from different bases, for example *tam* and *te* can come from *tumha* or *ta*.

There are two options for the source of term list used here (selected by button above the list): The first is from the over-all list, every terms in the whole collection. And the second is from the list produced by the main Tokenizer, the Tokenizer tab in the main window. If you have not yet used the Tokenizer, the list should be empty by this option.

	Case	Singular	Plural	
tumha				tam (65363)
amha				te (38819)
sabba	1 nom.	tvam, tuvam	tumhe, vo	tvam (8757)
katara	2 acc.	tavam, tam, tvam, tuvam	tumhākam, tumhe, vo	tumhe (3508)
ubbhaya	3 ins.	tayā, tvayā, te	tumhehi, tumhebbhi, vo	vo (2918)
añña	4 dat.	tumham, tava, tuyham, te	tumham, tumhākam, vo	tava (2797)
aññatara	5 abl.	tayā	tumhehi, tumhebbhi	tumhākam (2384)
aññatama	6 gen.	tumham, tava, tuyham, te	tumham, tumhākam, vo	tayā (1799)
pubba	7 loc.	tayi, tvayi	tumhesu	tuyham (1224)
para	ā voc.			tumhehi (777)
apara				tuvam (709)
dakkhiṇa				
uttara				
adhara				
ya				
ta				

Figure 5.2.: Declension of *tumha* against term list

5.2. Nouns/Adjectives

Nouns and adjectives in Pāli undergo the same inflectional rules, and sometimes a word takes dual position. So, I group them together. Since terms in this group are numerous, you have to select what you want by entering some query. Only simple search is available here. The source of terms is the *Concise Pāli-English Dictionary* (CPED). An example of noun's declension is shown in Figure 5.3.

	Case	Singular	Plural	
vimutti				vimutti (353)
vimuttirasa	1 nom.	vimutti	vimuttī, vimuttiyo	vimuttiyā (218)
vimuttisukha	2 acc.	vimuttiṃ	vimuttī, vimuttiyo	vimuttiṃ (157)
	3 ins.	vimuttiyā	vimuttihi, vimuttibhi, vimuttihi, vimuttibhi	vimutti (74)
	4 dat.	vimuttiyā	vimuttiṇam, vimuttiṇam	vimuttiyo (31)
	5 abl.	vimuttiyā	vimuttihi, vimuttibhi, vimuttihi, vimuttibhi	vimuttihi (14)
	6 gen.	vimuttiyā	vimuttiṇam, vimuttiṇam	vimuttiyaṃ (9)
	7 loc.	vimuttiyā, vimuttiyaṃ	vimuttisu, vimuttisu	vimuttiṇam (7)
	ā voc.	vimutti	vimuttī, vimuttiyo	vimuttisu (2)

Figure 5.3.: Declension of *vimutti* against term list

Results of an adjective's declension look exactly the same, but adjectives can decline into all three genders.

5. Declension table

What about the Compute button then? It has two uses. First, when a term has both regular and irregular declensions. And second, just when you want to play around with experiments.

I will give an example of the first case. For the second, the users can do by themselves. There are many words that use irregular paradigms, and I put a lot of effort to incorporate them here seamlessly.¹

A term I give here is *sara*. When it means ‘pond’ it declines irregularly as *mano-gaṇa* (m. only). This is what is shown when you enter ‘*sara*’ at the search box. But when it means ‘sound’ or ‘arrow’ it declines like normal nouns (m. and nt.). To reach the latter case you have to hit Compute after you enter ‘*sara*’. See the comparison in Figure 5.4.

For those who want to experiment with generic paradigms, you simply input a word with a proper ending and hit Compute. The proper endings are *a, i, ī, u, ū* for masculine words, *ā, ī, ī, u, ū* for feminine words, and *a, i, u* for neuter words. There are also two exceptions which use their special paradigms: *-ant* and *-ar*.

5.3. Numbers

Numeral declension is one of the most difficult parts of the program to implement.² All output in this part uses calculation. That means you can enter any number up to 6 digits, and it will be converted to its Pāli numeral³ with declension tables, both cardinal and ordinal. To help the beginners, however, there are pre-built lists that can be easily selected. Figure 5.5 shows the result of 100250.

I will skip other details because everything is self-explained and it is better to learn by playing.

¹If you find any irregular word that declines wrongly, according to the traditional textbooks, please report me right away.

²This part is not long but complicated. Other difficult and quite big modules are *Tokenizer*, *Pāli Text Reader & Sentence Manager*, and the viewer of Pāli documents including script transformers.

³The result list is not exhaustive, because one complex number can be rendered in many ways. The results here show only some compound units that can be calculated by the computer. Try 150, 250, 350, and so on; you will see why the conversion is so difficult.

Case	Singular	Plural
1 nom.	saro	sarā
2 acc.	saraṃ	sare
3 ins.	sarasā, sarena	sarehi, sarebhi
4 dat.	saraso, sarassa, sarāya, saratthaṃ	sarānaṃ
5 abl.	sarā, saramhā, sarasmā	sarehi, sarebhi
6 gen.	saraso, sarassa	sarānaṃ
7 loc.	sarasi, sare, saramhi, sarasmim̐	saresu
ā voc.	sara, sarā	sarā

(a) Declined as mano-gaṇa

Case	Singular	Plural
1 nom.	saro	sarā
2 acc.	saraṃ	sare
3 ins.	sarena	sarehi, sarebhi
4 dat.	sarassa, sarāya, saratthaṃ	sarānaṃ
5 abl.	sarā, saramhā, sarasmā	sarehi, sarebhi
6 gen.	sarassa	sarānaṃ
7 loc.	sare, saramhi, sarasmim̐	saresu
ā voc.	sara, sarā	sarā

(b) Computed declension

Figure 5.4.: Declension of *sara*

Case	Singular	Plural
1 nom.	paññāsuttaradvisatādhikāsatasahassaṃ	paññāsuttaradvisatādhikāsatasahasāni, paññāsuttaradvisatādhikāsatasahasā
2 acc.	paññāsuttaradvisatādhikāsatasahassaṃ	paññāsuttaradvisatādhikāsatasahasāni, paññāsuttaradvisatādhikāsatasahasse
3 ins.	paññāsuttaradvisatādhikāsatasahasena	paññāsuttaradvisatādhikāsatasahasēhi, paññāsuttaradvisatādhikāsatasahasēbhi
4 dat.	paññāsuttaradvisatādhikāsatasahasassa	paññāsuttaradvisatādhikāsatasahasānaṃ
5 abl.	paññāsuttaradvisatādhikāsatasahasā, paññāsuttaradvisatādhikāsatasahasamhā, paññāsuttaradvisatādhikāsatasahasasmā	paññāsuttaradvisatādhikāsatasahasēhi, paññāsuttaradvisatādhikāsatasahasēbhi
6 gen.	paññāsuttaradvisatādhikāsatasahasassa	paññāsuttaradvisatādhikāsatasahasānaṃ
7 loc.	paññāsuttaradvisatādhikāsatasahasē, paññāsuttaradvisatādhikāsatasahasamhi, paññāsuttaradvisatādhikāsatasahasasmim̐	paññāsuttaradvisatādhikāsatasahasēsu
ā voc.	paññāsuttaradvisatādhikāsatasahasā	paññāsuttaradvisatādhikāsatasahasāni

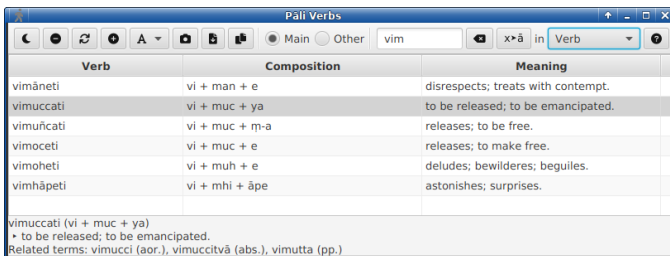
Figure 5.5.: Declension of 100250

6

Verbs

Verbs listed here come from CPED. Fortunately, the dictionary has already provided us the composition of verbs and their related forms. New students can learn a lot just by going through these lists.

We have only two main options: to see main (canonical) verb form (singular, present, third-person, active), typically those ending with *-ti*, and the rest. In main verbs, you can also filter the list by the verbs themselves, or their root and prefix, or their ending component (*paccaya*)¹, or their meaning. The use is straightforward. Figure 6.1 shows a search result.



The screenshot shows a web browser window titled "Pali Verbs". The search bar contains "vim" and the dropdown menu is set to "Verb". The results are displayed in a table with three columns: Verb, Composition, and Meaning.

Verb	Composition	Meaning
vimāneti	vi + man + e	disrespects; treats with contempt.
vimuccati	vi + muc + ya	to be released; to be emancipated.
vimuñcati	vi + muc + m-a	releases; to be free.
vimoceti	vi + muc + e	releases; to make free.
vimoheti	vi + muh + e	deludes; bewilderes; beguiles.
vimhāpeti	vi + mhi + āpe	astonishes; surprises.

vimuccati (vi + muc + ya)
* to be released; to be emancipated.
Related terms: vimucci (aor.), vimuccitvā (abs.), vimutta (pp.)

Figure 6.1.: Result of main verbs when search ‘vim’ and select *vimuccati*

¹The name of roots and *paccayas* can be slightly different from Kaccāyana/Saddanīti’s convention.

In other verb forms, you have to choose the group you want (see Figure 6.2). The names of verb forms here come directly from the dictionary. So, some name may look unfamiliar, for example, *Potential Participle* is called commonly by other textbooks as *Future Passive Participle*.

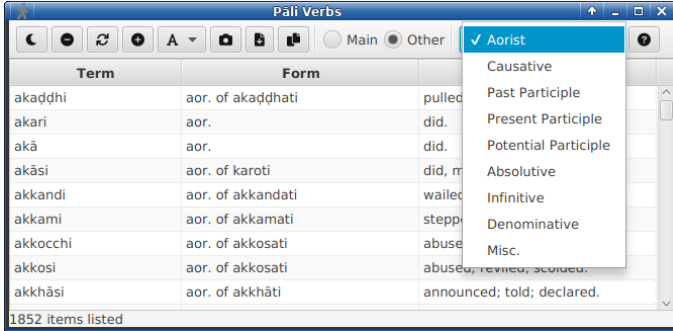


Figure 6.2.: Options of other verb forms

Conjugation table

Rendering verb forms is a formidable task for new students because there are many things to remember, as well as to know how to put them together. Having a good tool that can provide a good number of examples can speed up the learning process. Unlike a declension table that can be created upon any declinable word, a conjugation table cannot be created for any root because of the lack of complete uniformity.

With this variety of verb formation, we can do, at best, only by providing some typical verb samples, as shown in the left-sided list. And only a handful of verbs that have wide range of forms covering most of tenses and moods, for example, *pacati* (*paca*), *gacchati* (*gamu*), and *bhavati* (*bhū*). That is why the program starts with *pacati*.¹

There are two main modes in this module: main and derivation. Main verb formation is about *ākhyāta*. So, conjugation makes sense only for main verb forms. Derivative verbs have no conjugation, because they undergo different rules resulting in different verb forms. Most of derivative verbs are declinable. So, we will see declension tables as a result of derivation mode.

Like declension tables, we can check the words rendered against an internal term list by opening the right pane (☰)

¹Rendering *paca* is straightforward and clean, having no weird variation. It may be the most used example for teaching verb formation.

button). Opening the right pane can make rendering a new verb sluggish, so use it when necessary.

Rendering a conjugation table has additional two options (✓ button): (1) whether *Attanopada* (middle voice) will be shown, (2) whether Augment (*a-*)² will be added. Figure 7.1 shows aorist forms of *pacati* in full options.

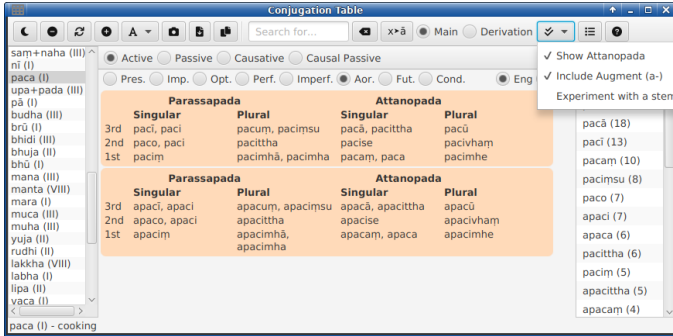


Figure 7.1.: Aorist forms of *pacati*

To help new learners, I add Table 7.1 to show technical terms used by Pāli textbooks, as abbreviated in the result area. However, for those who are new to Pāli grammar and not familiar with verb formation, just playing around this tool can do little help. So, you should read *Pāli for New Learners*, Book I, to understand the main idea about Pāli verb formation first.

In Derivation mode, major derivative verb forms can be studied here in various combinations. The result is shown as declension tables, except for *tvā* verbs. Figure 7.2 show masculine *ta* forms (past participle) of *pacati*. Derivative verbs have substantial details in Pāli grammar, so please see further in the book mentioned.

Like Declension table window, we can experiment with any verb stem here (see the option in ✓ button). In experiment mode, the result will be rendered against the generic rules of the verb form selected. This will be useful when the right pane is also opened. In this way, the users can check whether

²Augment is added only to some of Imperfect verbs and all of Aorist, and Conditional verbs. The augment has no specific meaning, so it is a redundant part. However, it is fashionable to be used with some verbs.

7. Conjugation table

Table 7.1.: Tenses and moods

English	Pāli
Present tense	<i>Vattamānā</i>
Imperative mood	<i>Pañcamī</i>
Optative mood	<i>Sattamī</i>
Perfect tense	<i>Parokkhā</i>
Imperfect tense	<i>Hiyyattanī</i>
Aorist tense	<i>Ajjatanī</i>
Future tense	<i>Bhavissanti</i>
Conditional mood	<i>Kālātipatti</i>

Case	Singular	Plural
1 nom.	pacito	pacitā
2 acc.	pacitaṃ	pacite
3 ins.	pacitena	pacitehi, pacitebhi
4 dat.	pacitassa, pacitāya, pacitattham	pacitānam
5 abl.	pacitā, pacitamhā, pacitasmā	pacitehi, pacitebhi
6 gen.	pacitassa	pacitānam
7 loc.	pacite, pacitamhi, pacitasmim	pacitesu
ā voc.	pacita, pacitā	pacitā

Figure 7.2.: Masculine *ta* forms of *pacati*

a particular stem has its use in the texts.

In the traditional way of learning, to know a verb means to know its root. So, it is really important to identify a root when we encounter a verb. Learning Pāli roots, unfortunately, is not an easy task because different schools name roots differently. Furthermore, some roots can behave strangely in certain contexts. So, in modern way of learning Pāli, knowing roots is less pressing. But knowing them still brings great benefits.

Roots listed here come from the compilation of Ven. U Silananda in *Pali Roots in Saddanīti Dhātu-Mālā compared with Pāṇinīya-Dhātupāṭha*.¹ So, the roots are familiar to those who follow Kaccāyana/Saddanīti school.

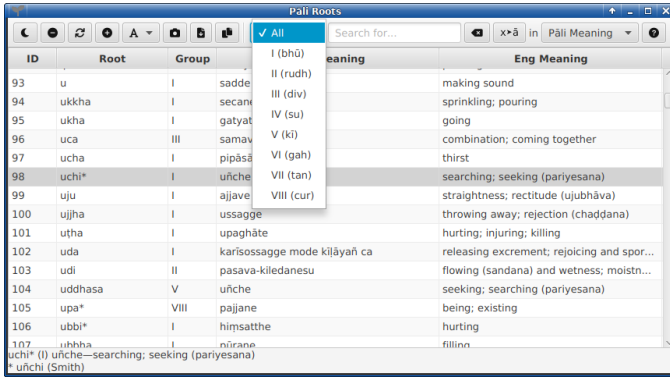
Things the users can do with the roots are: you can select only a specific verb group, you can search in the roots' name, in Pāli meaning, and in English meaning. Figure 8.1 show Roots window with an option opened.

In some entries, you can see an asterisk (*) mark in the root's name. In some, you can also see double asterisks (**) in the Pāli meaning. These marks tell us that there is a comment in the entry. You can see the comment by clicking the row in the table. Comments are directly taken from the original work. So, some of them can be difficult to understand exactly.²

¹Edited by U Nandisena, available at <https://archive.org/details/ThePaliRootsInSaddaniti>

²*Smith* mentioned by comments is the publication of Saddanīti edited by H. Smith (1928–66), 6 volumes. In this context it means the second volume

8. Roots



ID	Root	Group	Pali Meaning	Eng Meaning
93	u	I	sadde	making sound
94	ukkha	I	secanā	sprinkling; pouring
95	ukha	I	gatyat	going
96	uca	III	samav	combination; coming together
97	ucha	I	pipāsā	thirst
98	uchi*	I	uñche	searching; seeking (pariyesana)
99	uju	I	ajjave	straightness; rectitude (ujubhāva)
100	ujjha	I	ussagge	throwing away; rejection (chaddana)
101	utha	I	upaghāte	hurting; injuring; killing
102	uda	I	karissagge mode kilāyañ ca	releasing excrement; rejoicing and spor...
103	udi	II	pasava-kiledanesu	flowing (sandana) and wetness; moistn...
104	uddhasa	V	uñche	seeking; searching (pariyesana)
105	upa*	VIII	pajjane	being; existing
106	ubbi*	I	himsatthe	hurting
107	upphā	I	nirāna	filling

* uñchi (Smith)

Figure 8.1.: Roots window

The best way to study these roots, I suggest, is to read *Saddanīti Dhātumālā* (available in the collection under *Byākaraṇa gantha-saṅgaho* in the *Añña* group) directly on the root you are interested in. A quick way to access to the desired point is to search the root and its Pāli meaning in that text group. Learn more about search in its part below.

(*Dhātumālā*) of that work. The 5 volumes of the first edition can be found at <https://archive.org/details/SaddanitiAggavamsasPaliGrammar01> (to 05).

Prosody

Pāli prosody, or *chanda*, is a big, difficult topic to learn. I am not keen on this subject, nor I am a poetry enthusiast. I see it from an engineering point of view that if we have a good tool we can learn this hard topic easier, or even with fun. There are several things to learn before you can use this tool effectively. When it is first opened, it shows just a list of prosodic types. When you click on a row, you just see a bizarre formula at the bottom border (see Figure 9.1).

The screenshot shows a window titled "Prosody" with a toolbar containing icons for navigation and analysis. Below the toolbar is a table with the following data:

ID	Name	Type	Ref.	Score
0	Ariyā	Mattāvutti, Ariyā	Vut.17	0.0000
1	Capalā (full)	Mattāvutti, Ariyā	Vut.22	0.0000
2	Capalā (1st half)	Mattāvutti, Ariyā	Vut.22	0.0000
3	Capalā (2nd half)	Mattāvutti, Ariyā	Vut.22	0.0000
4	Mukhacapalā	Mattāvutti, Ariyā	Vut.23	0.0000
5	Jaghanacapalā	Mattāvutti, Ariyā	Vut.24	0.0000
6	Gīti (1st half of Ariyā)	Mattāvutti, Gīti	Vut.25	0.0000
7	Upagīti (2nd half of Ariyā)	Mattāvutti, Gīti	Vut.26	0.0000
8	Uggīti (Reversed Ariyā)	Mattāvutti, Gīti	Vut.27	0.0000
9	Ariyāgīti	Mattāvutti, Gīti	Vut.28	0.0000
10	Vetāliya	Mattāvutti, Vetāliya	Vut.29	0.0000

At the bottom of the window, a formula is displayed: `!j-4-!j-4-!j-j|n-!j-g;!j-4-!j-4-!j-l-!j-g (57)`

Figure 9.1.: Prosody window

For I have never mentioned Pāli prosody in my previous books, I will do it in this chapter just enough to understand how the program works. The main, in fact only, work that we refer to is *Vuttodaya*.¹

9.1. A survival introduction to Pāli prosody

As a subject, prosody is “The study of the metrical structure of verse.”² For our concern, it is the way to put words into an organized structure ruled by groups of syllables. At the fundamental level, a Pāli syllable can be either ‘light’ (*lahu*) or ‘heavy’ (*garu*). To measure this ‘weight’ we consider only vowels in a word.

Lahu is short open vowels When *a*, *i*, or *u* sits by itself alone or follows a consonant, and **not** followed by a double-consonant or *m*, it is counted as ‘light.’ *Lahu* weighs 1 unit (measure). So, we use symbol **1** and **l** here.³

Garu is everything else When you can identify *lahu*, *garu* is simply the rest of that. *Garu* weighs 2 units. We use symbol **2** and **g** here.⁴

¹The author is Saṅgharakkhita of Udumbaragiri, around the 12th-13th century AD, in Sri Lanka. In the collection, it is *Vuttodayapāṭha*, under *Byākaraṇa gantha-saṅgaho*. The text is so terse that we can make little sense out of it. Fortunately, I have *Vuttodayamañjarī* by Gandhasārābhivamsa, printed by Mahachulalongkorn Buddhist University, Nakhon Pathom, Thailand (2001). This commentary on *Vuttodaya* is written in Thai. It is comprehensive and well-organized. I think this is what a commentary should be. Writing new commentaries with modern languages is preferable nowadays because we can make them very clear to the readers, as exemplified by this work. If you cannot read them, certainly someone can. Writing a new text with Pāli makes things worse, and does little help to the textual understanding. For English readers, I find a translation by Anandajoti Bhikkhu useful, but not beginner-friendly, see <https://www.ancient-buddhist-texts.net/Textual-Studies/Vuttodaya/index.htm>.

²The American Heritage Dictionary, <https://www.ahdictionary.com/word/search.html?q=prosody>

³Many textbooks on the subject use \sim for *lahu* and $-$ for *garu*, or vice versa (see *Vut.7*). I find this very confusing, so I avoid using these symbols.

⁴The symbol **2** has double meaning. It means *garu* in the analyzed result (as we shall see shortly). And if the symbol appears in a formula, it means any combination that give 2 measures long, i.e., both ll (1+1) and g (2) can be

9.2. Two types of prosodic patterns

Let us see an example to make sure we get the same picture. In fact, I make a tool that can calculate the meter of any text. It is a part of the program's text editor. A result of meter calculation is shown in Figure 9.2. For more information, see Chapter 16.

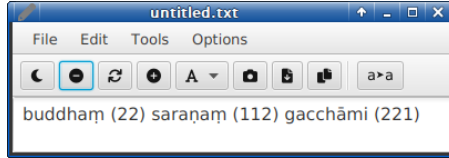


Figure 9.2.: Meter calculation in the program's editor

In the result, *lahu* and *garu* are denoted by 1 and 2 respectively. First, *buddham* gets 22 because the *u* is followed by *ddh* (double consonants), and the *a* is followed by *m*. Second, *saraṇam* gets 112 because the first two *a* are an open short vowel, whereas the last one is followed by *m*. And third, *gacchāmi* gets 221 because the *a* is followed by *cch* (double consonants), the *ā* is a long vowel, and the *i* is an open short vowel.⁵

Now we can see that when we put words together by a constraint of *lahu-garu* patterns, we are composing a verse. The whole business of Pāli prosody is to recognize patterns and to put words into patterns.

9.2. Two types of prosodic patterns

Broadly speaking, there are two ways to put syllables into pattern: (1) by counting the syllables, and (2) by counting the weight or measure. Understanding the distinction between the two is important, so let make it clear first. We can illustrate the two types of counting by using the example above in

counted as 2. This may make some confused, but I insist on using 1 and 2 because they look better and clearer.

⁵Some may wonder “What about a short vowel followed by a single consonant (which does not belong to the next syllable)?” Put it bluntly, there is no such a thing. You may see *tad* here and there in Pāli books, but not in the traditional textbooks. We will not see *tad* stands alone. It is *tam* with *d* substitution as a result of word-joining (*Sandhi*). So, that syllable may be either of *lahu* or *garu* type depending on whether the *d* is doubled or not.

Table 9.1.

Table 9.1.: Syllable vs. weight summation

Word	Meter	Syllable sum	Weight sum
<i>buddham</i>	22	2	4
<i>saraṇam</i>	112	3	4
<i>gacchāmi</i>	221	3	5

By those ways of counting, *Vuttodaya* categorizes prosodic patterns into two, namely *mattāvutti* and *vaññavutti*. The former uses weight-counting, mainly with 4-measure groups (*gaṇa*). The latter uses syllable-counting, mainly with 3-syllable groups. Table 9.2 summarizes the *mattāvutti* groups used in the program's prosodic formulas, whereas Table 9.3 summarizes the *vaññavutti* groups.

Table 9.2.: Meter groups used in *mattāvutti*

Symbol	Group	Notes
n	1111	<i>Na gaṇa</i>
s	112	<i>Sa gaṇa</i>
j	121	<i>Ja gaṇa</i>
b	211	<i>Bha gaṇa</i>
m	22	<i>Ma gaṇa</i>
4	any	any of the above five
6	any	any of the above five plus g or ll
8	any	any combination of the above five
l	1	<i>lahu</i>
g	2	<i>garu</i>
2	any	g (2) or ll (11)

In the tables, we can see that in *mattāvutti* there are 5 main groups (*gaṇa*), all have 4-unit weight but variable syllables. The names of *gaṇa* are traditional, but I use lowercase here to distinguish them from *vaññavutti* types. In the latter case, there are 8 main groups, all have 3 syllables but variable weight.

Some names are common, some are new. Note carefully on *Na gaṇa* and *Ma gaṇa*, which share the name but not the content. I use uppercase in the latter. The 14-*lahu* group (L) is used only in the program, not in any textbooks. Some of *mattāvutti* verse types use the symbols of *vaññavutti* in their formula. For example, Vetāliya (Vut.29) uses R (212), Opacchandāsaka (Vut.30) uses R and Y (122), and Acaladhiti (Vut.38) uses L.⁶

Table 9.3.: Meter groups used in *vaññavutti*

Symbol	Group	Notes
N	111	<i>Na gaṇa</i>
S	112	<i>Sa gaṇa</i>
J	121	<i>Ja gaṇa</i>
Y	122	<i>Ya gaṇa</i>
B	211	<i>Bha gaṇa</i>
R	212	<i>Ra gaṇa</i>
T	221	<i>Ta gaṇa</i>
M	222	<i>Ma gaṇa</i>
l	1	<i>lahu</i>
g	2	<i>garu</i>
L	14 × l	used only in the program

As described above, it is possible that one passage can be analyzed by both methods, resulting in different patterns. Let us see the analysis of our example above in Table 9.4. There are many possibilities how to put syllables into a group even when using the same method, as shown in the table.⁷ This variety gives us a number of predefined patterns presented in *Vuttodaya*, as we shall see in the next sections.

Before we are going to see the verse list, let me explain some symbols used in verse formulas. I translated the descriptive rules into computable formulas. So, they look weird to non-programmer users. But the formulas are not difficult to read if you try to understand the symbols. Beside of meter-group symbols described above, there are also special symbols, as

⁶To make it clear, using symbols in the formulas has nothing to do with the tradition. It is a convenient way for computation in programming.

⁷8/13 = syllable sum/weight sum

Table 9.4.: Syllable vs. weight grouping

Passage	<i>buddham saraṇaṃ gacchāmi</i>
Raw meter	22 112 221 (8/13)
Mattāvutti grouping 1	22-112-22-1 = m-s-m-l
Mattāvutti grouping 2	2-211-22-21 = g-b-m-gl
Vaññavutti grouping 1	22-112-221 = gg-S-T
Vaññavutti grouping 2	221-122-21 = T-Y-gl

shown in Table 9.5.

Table 9.5.: Symbols in verse formulas

Symbol	Meaning
Vertical bar ()	logical OR
Exclamation mark (!)	logical NOT
Semicolon (;)	line separator
Hyphen (-)	mere separator

We will see how logical OR and NOT work when I put a real formula into explanation. The line separator (;), if present, tells us that the pattern needs 2 lines of verse to make the analysis complete. If the input passage is not long enough, or does not have 2 lines as required, incomplete will be shown in the status bar. But, if the analyzed text has more syllables than the formula requires, ‘over-required’ will be shown instead. Hyphen (-) is just a separator, having no meaning. It only makes formulas easier to read.

9.3. Verse types of *mattāvutti*

Vuttodaya starts with definitions and general rules, then verse types of *mattāvutti* are listed. We can divide further into 4 subgroups, namely, Ariyā, Gīti, Vetāliya, and Mattāsamaka. All verse types (*gāthā*) in each subgroup are listed in Table 9.6. Not all of these can be analyzed by the program, as noted

below.

Table 9.6.: Verse types of *mattāvutti*

ID	Name	WS ⁸	Parent group	Ref.
0	Ariyā	57	Ariyā	Vut.17
–	Paṭhayā	57	Ariyā	Vut.20
–	Vipulā	57	Ariyā	Vut.21
1	Capalā (full)	57	Ariyā	Vut.22
2	Capalā (1st half)	30	Ariyā	Vut.22
3	Capalā (2nd half)	27	Ariyā	Vut.22
4	Mukhacapalā	57	Ariyā	Vut.23
5	Jaghanacapalā	57	Ariyā	Vut.24
6	Gīti (1st half of Ariyā)	30	Gīti	Vut.25
7	Upagīti (2nd half of Ariyā)	27	Gīti	Vut.26
8	Uggīti (Reversed Ariyā)	57	Gīti	Vut.27
9	Ariyāgīti	32	Gīti	Vut.28
10	Vetālīya	30	Vetālīya	Vut.29
11	Opacchandāsaka	34	Vetālīya	Vut.30
12	Āpātalikā	30	Vetālīya	Vut.31
13	Dakkhiṇantikā	30	Vetālīya	Vut.32
14	Udiccavutti	30	Vetālīya	Vut.33
15	Paccavutti	30	Vetālīya	Vut.34
16	Pavattaka	30	Vetālīya	Vut.35
17	Aparantikā (1)	32	Vetālīya	Vut.36
18	Aparantikā (2)	32	Vetālīya	Vut.36
19	Cāruhāsini (1)	28	Vetālīya	Vut.37
20	Cāruhāsini (2)	28	Vetālīya	Vut.37
21	Acaladhiti	16	Mattāsamaka	Vut.38
22	Mattāsamaka	16	Mattāsamaka	Vut.39
23	Visiloka	16	Mattāsamaka	Vut.40
24	Vānavāsikā	16	Mattāsamaka	Vut.41
25	Citrā	16	Mattāsamaka	Vut.42
26	Upacitrā	16	Mattāsamaka	Vut.43
–	Pādākulaka		Mattāsamaka	Vut.44

Verse types in this category have complicated formulas. So, let us go slowly. Normally, one stanza consists of 4 feet (quar-

⁸Weight Sum, the total weight required

9. Prosody

ter, *pāda*) or 2 lines. Some verse types need 2-line pattern, some need only 1-line to complete the analysis. This sounds a bit tricky. Normally, we do not compose a single-line verse. At least, we need two lines. However, many patterns have repeated formula for the first and second line. As a result, checking only one line for such patterns is enough. A suggestion here is “Always analyze two lines of verse.” Let us see the metrical pattern of Ariyā:

!j-4-!j-4-!j-j|n-!j-g; (30)

!j-4-!j-4-!j-l-!j-g (27)

The pattern needs 2 lines (see ;). The first line is read NOT j, 4, NOT j, 4, NOT j, j OR n, NOT j, g. The second is read NOT j, 4, NOT j, 4, NOT j, l, NOT j, g. As shown in Table 9.2, 4 means any of meter groups is applicable, NOT j means any meter group but j, and j OR n means either j or n is valid.

Let us see a real example. This stanza is the first one of Ganthārambhakathā in Dīghanikāya-aṭṭhakathā.

Karunāsītalahadayaṃ, paññāpajjotavihatamohatamaṃ;

112-211-112, 22-22-1111-211-2;

s-b-s, m-m-n-b-g;

Sanarāmaralokagaruṃ, vande sugataṃ gativimuttaṃ.

112-112-112, 22 112 1-112-2.

s-s-s, m s l-s-g.

If you carefully analyze the verse as shown above, you will see that it conforms to the pattern described perfectly. To analyze the verse by the program, first open the text mentioned by TOC Tree (see Chapter 10), select the stanza and copy. Then open Prosody window and hit Analyze button. The most relevant results will show at the top of the table. The analyzed result against Ariyā pattern is shown in Figure 9.3.

In the result table produced by the program, *Score* is the calculation of percentage of matching between the text and the formula. So, 1.0 means 100% matched. There are possibilities that we get multiple 1.0 results. But all of them are not always ‘correct’ answers. Some portion of text may be totally matched

ID	Name	Type	Ref.	Score
0	Ariyā	Mattāvutti, Ariyā	Vut.17	1.0000
22	Mattāsamaka	Mattāvutti, Mattāsamaka	Vut.39	1.0000
103	Setavavipulā	Vaññāvutti, Visamavutti	Vut.123	1.0000
7	Upagīti (2nd half of Ariyā)	Mattāvutti, Gīti	Vut.26	0.9259
6	Gīti (1st half of Ariyā)	Mattāvutti, Gīti	Vut.25	0.9000
9	Ariyāgīti	Mattāvutti, Gīti	Vut.28	0.8438

!j-4-!j-4-!j-j|n-!j-g;!j-4-!j-4-!j-l-!j-g (57)

Figure 9.3.: Analysis of *Ariyā* verse type

against the formula, but the text is over-required as shown in Figure 9.4.

Another noteworthy point, even though a result is not 100% matched, it can be a correct answer. Most often cases are the unmatched last syllable of each line: it can be *g* despite the required *l*, or vice versa. And as you may find out yourselves later, exactly matched verses in the real text are quite rare, except newly composed ones that are crafted in that way. The real old verses are recalcitrant to the rules somehow. That is why many exceptions are mentioned in detailed textbooks on prosody.¹⁰

Now you can see how the program is really helpful in metrical analysis. Doing it by hand is tedious, laborious, and error-prone. Computer has its limitation, however. There are certain things unable to be analyzed, at least in a simple way, by the program. Metrical pause (caesura), or *yati* in Pāli, is one

¹⁰An attempt to shoehorn a verse into a recognized pattern creates exceptions, or extra considerations, to the rules. I do not use that approach in the program. We just find the closest candidates and regard the verse as it is. I think this is the normal way how poetry is composed. To achieve certain artistic result, breaking some rules seems natural. Of course, there can be cases of half-baked poetry that just ignores or plays with rules. And, finally, there can be many cases that rules are held so firmly that the meter trumps the clarity of the words used—words are oddly changed to conform with rules.

9. Prosody

ID	Name	Type	Ref.	Score
0	Ariyā	Mattāvutti, Ariyā	Vut.17	1.0000
22	Mattāsamaka	Mattāvutti, Mattāsamaka	Vut.39	1.0000
103	Setavavipulā	Vaññavutti, Visamavutti	Vut.123	1.0000
7	Upagīti (2nd half of Ariyā)	Mattāvutti, Gīti	Vut.26	0.9259
6	Gīti (1st half of Ariyā)	Mattāvutti, Gīti	Vut.25	0.9000
9	Ariyāgīti	Mattāvutti, Gīti	Vut.28	0.8438

m|s|b|n-m|s|b-s-m|s (16) over-required

Figure 9.4.: Example of an over-required case

of those.

Generally, metrical pause has two kinds: First, the pause after each foot ends. This is common in recitation and writing. When you read a verse, you are supposed to pause at points to make a sense of rhythm. And it is natural to pause at the end of a foot, probably a small pause after the odd feet, a bigger one after the first line (first even foot), and the biggest one after the second line (last even foot). The second kind of pause is rarer, the pause inside the feet. Some verse types have rules about pauses, particularly the Ariyā group. Pauses at the endings is easy to see because punctuation marks can be used in text. But in-between pauses cannot be shown in the program, because this kind of pause makes sense only when the verse is recited.¹¹

Another limitation is the program cannot detect the spilled-over effect: a word is cut into two and both parts are split across feet. This aspect differentiates Vipulā (having a spilled-over) from Paṭhayā (no spilled-over).¹² As a result, the pro-

¹¹The rules concerning metrical pause are not easy to understand, so I skip the explanation.

¹²From Vut.21, “*Yattha gaṇattaya mullaṅghi, Yo’bhayathā’dimo bhava vipulā.*” According to the formula, the word *mullaṅghi* has to be split so that the *-ghi* part belongs to the latter foot.

gram can not tell the two verse types apart. They are grouped together under Ariyā.¹³

It should be noted that Vetāliya (Vut.29) has a particular rule: “No successive 6 *lahus* are allowed in the even feet.” This means in its formula, 6-R-l-g-8-R-l-g, the 8 in the second part does not allow neither 111112 nor 211111. This rule has been already implemented in the latest program.

Finally in this category, Pādākulaka (Vut.44) is an arbitrary combination of Mattāsamaka group. It has no fixed pattern to detect, so it is left out from analysis (no ID).¹⁴

9.4. Verse types of *vaññavutti*

There are 3 main subgroups in this category: Samavutti (symmetry), Aḍḍhasamavutti (semi-symmetry), and Visamavutti (asymmetry). The sense of ‘symmetry’ here means whether each foot (quarter) of a verse is equal in length. It can be that all 4 feet are equal (symmetry), or odd feet and even feet are equal (semi-symmetry), or arbitrary (asymmetry). All verse types in this category, plus some more, are listed in Table 9.7.

In Vut.13, twenty-six types of (symmetric) prosody (*chanda*) are mentioned. That is to say, in theory, the metrical patterns can run from 1 syllable to 26 syllables. *Vuttodaya* itself, however, lists only seventeen types: from 6-syllable (Gāyattī) to 22-syllable (Ākati) pattern (see other names in footnotes).

¹³There is also a confusion of this. According to the text, in Vut.19 Ariyāsāmañña is mentioned, and Paṭṭhayā is mentioned in Vut.20, Vipulā in Vut.21. So, it is supposed that Ariyāsāmañña and Paṭṭhayā are different kinds. But it is not very clear how they differ to each other. In a way, we can interpret that they are the same. By this concern, I drop Ariyāsāmañña from the list.

¹⁴ID numbers used here have nothing to do with *Vuttodaya*. They are used internally in the program. The users can use the numbers for sorting the table, by clicking the column headers. To refer to a particular verse type, using a reference to *Vuttodaya*, if any, is more reliable.

Table 9.7.: Verse types of *vaññavutti*

ID	Name	SS ¹⁵	Parent group	Ref.
27	Tanumajjhā	6 ¹⁷	Samavutti	Vut.46
28	Kumāralalitā	7 ¹⁸	Samavutti	Vut.47
29	Citrapadā	8 ¹⁹	Samavutti	Vut.48
30	Vijjumālā	8	Samavutti	Vut.49
31	Māṇavaka	8	Samavutti	Vut.50
32	Samānikā	8	Samavutti	Vut.51
33	Pamāṇikā	8	Samavutti	Vut.52
34	Halamukhī	9 ²⁰	Samavutti	Vut.53
35	Bhujagasususatā	9	Samavutti	Vut.54
36	Suddhvirājita	10 ²¹	Samavutti	Vut.55
37	Paṇava	10	Samavutti	Vut.56
38	Rummavatī	10	Samavutti	Vut.57
39	Mattā	10	Samavutti	Vut.58
40	Campakamālā	10	Samavutti	Vut.59
41	Manoramā	10	Samavutti	Vut.60
42	Ubbhāsaka	10	Samavutti	Vut.61
43	Upaṭṭhitā	10	Samavutti	Vut.62
44	Indavajira	11 ²²	Samavutti	Vut.63
45	Upendavajira	11	Samavutti	Vut.64
–	Upjāti	11	Samavutti	Vut.65
46	Sumukhī	11	Samavutti	Vut.66
47	Dodhaka	11	Samavutti	Vut.67
48	Sālinī	11	Samavutti	Vut.68
49	Vātommi	11	Samavutti	Vut.69
50	Sirī	11	Samavutti	Vut.70
51	Rathoddhatā	11	Samavutti	Vut.71
52	Svāgatā	11	Samavutti	Vut.72
53	Bhaddikā	11	Samavutti	Vut.73
54	Vamsaṭṭha	12 ²³	Samavutti	Vut.74

Continued on the next page...

¹⁶Syllable Sum, the total syllables required¹⁷Gāyattī¹⁸Uṇhikā¹⁹Anuṭṭhubhā²⁰Barahatī²¹Panti²²Tiṭṭhubhā²³Jagatī

Table 9.7: Verse types of *vaññavutti* (contd...)

ID	Name	SS ¹⁶	Parent group	Ref.
55	Indavaṃsā	12	Samavutti	Vut.75
56	Toṭaka	12	Samavutti	Vut.76
57	Dutavilambita	12	Samavutti	Vut.77
58	Puṭa	12	Samavutti	Vut.78
59	Kusumavicittā	12	Samavutti	Vut.79
60	Bhujāṅgappayāta	12	Samavutti	Vut.80
61	Piyaṃvadā	12	Samavutti	Vut.81
62	Lalitā	12	Samavutti	Vut.82
63	Pamitakkharā	12	Samavutti	Vut.83
64	Ujjalā	12	Samavutti	Vut.84
65	Vessadevī	12	Samavutti	Vut.85
66	Tāmarasa	12	Samavutti	Vut.86
67	Kamalā	12	Samavutti	Vut.87
68	Pahassiṇī	13 ²⁴	Samavutti	Vut.88
69	Rucirā	13	Samavutti	Vut.89
70	Aparājitā	14 ²⁵	Samavutti	Vut.90
71	Paharaṅkalikā	14	Samavutti	Vut.91
72	Vasantatilakā	14	Samavutti	Vut.92
73	Sasikalā	15 ²⁶	Samavutti	Vut.93
74	Maṇiḡuṅanikara	15	Samavutti	Vut.94
75	Mālinī	15	Samavutti	Vut.95
76	Pabhaddaka	15	Samavutti	Vut.96
77	Vāṇinī	16 ²⁷	Samavutti	Vut.97
78	Sikharāṇī	17 ²⁸	Samavutti	Vut.98
79	Hariṇī	17	Samavutti	Vut.99
80	Mandakkantā	17	Samavutti	Vut.100
81	Kusumitalatā	18 ²⁹	Samavutti	Vut.101
82	Meghavipphujjitā	19 ³⁰	Samavutti	Vut.102
83	Saddūlavikkīḷita	19	Samavutti	Vut.103
84	Vutta	20 ³¹	Samavutti	Vut.104

Continued on the next page...

²⁴Atijagatī²⁵Sakkari²⁶Atisakkari²⁷Aṭṭhi²⁸Accaṭṭhi²⁹Dhiti³⁰Atidhiti³¹Kati

Table 9.7: Verse types of *vaññavutti* (contd...)

ID	Name	SS ¹⁶	Parent group	Ref.
85	Sandharā	21 ³²	Samavutti	Vut.105
86	Bhaddaka	22 ³³	Samavutti	Vut.106
87	Upacitta	22	Aḍḍhasamavutti	Vut.107
88	Dutamajjhā	23	Aḍḍhasamavutti	Vut.108
89	Vegavatī	21	Aḍḍhasamavutti	Vut.109
90	Bhaddavirāja	21	Aḍḍhasamavutti	Vut.110
91	Ketumatī	21	Aḍḍhasamavutti	Vut.111
92	Ākhyānakī	22	Aḍḍhasamavutti	Vut.112
93	Viparītākhyānakī	22	Aḍḍhasamavutti	Vut.113
94	Hariṇaplutā	23	Aḍḍhasamavutti	Vut.114
95	Aparavatta	23	Aḍḍhasamavutti	Vut.115
96	Pupphitaggā	25	Aḍḍhasamavutti	Vut.116
97	Yavamatī	25	Aḍḍhasamavutti	Vut.117
98	Vatta	16	Visamavutti	Vut.118
99	Pathyāvatta	16	Visamavutti	Vut.119
100	Vīparītapathyāvatta	16	Visamavutti	Vut.120
101	Capalāvatta	16	Visamavutti	Vut.121
102	Piṅgalavipulā	16	Visamavutti	Vut.122
103	Setavavipulā	16	Visamavutti	Vut.123
104	Bhakarāvipulā	16	Visamavutti	Vut.124
105	Pathamabhakarāvipulā	32	Visamavutti	Vut.124
106	Tatīyabhakarāvipulā	32	Visamavutti	Vut.124
107	Rakarāvipulā	16	Visamavutti	Vut.125
108	Pathamarakarāvipulā	32	Visamavutti	Vut.125
109	Tatīyarakarāvipulā	32	Visamavutti	Vut.125
110	Nakarāvipulā	16	Visamavutti	Vut.126
111	Pathamanakarāvipulā	32	Visamavutti	Vut.126
112	Tatīyanakarāvipulā	32	Visamavutti	Vut.126
113	Takarāvipulā	16	Visamavutti	
114	Pathamatakārāvipulā	32	Visamavutti	
115	Tatīyatakārāvipulā	32	Visamavutti	
116	Makarāvipulā	16	Visamavutti	
117	Pathamamakārāvipulā	32	Visamavutti	
118	Tatīyamakarāvipulā	32	Visamavutti	
119	Sakarāvipulā	16	Visamavutti	

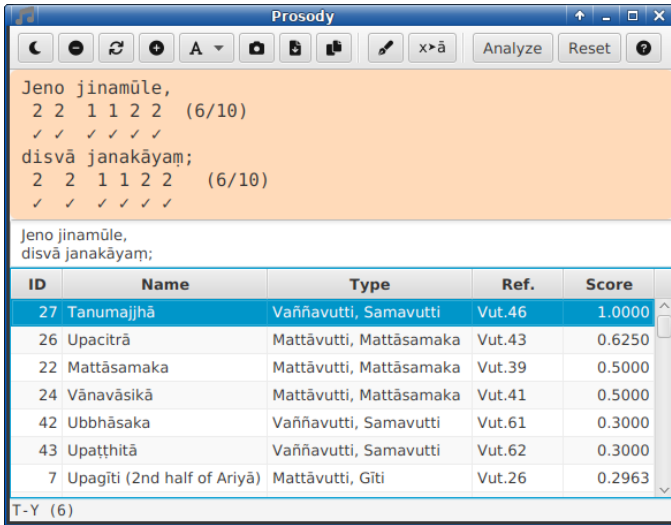
Continued on the next page...

³²Pakati³³Ākati

Table 9.7: Verse types of *vaññavutti* (contd...)

ID	Name	SS ¹⁶	Parent group	Ref.
120	Pathamasakāravipulā	32	Visamavutti	
121	Tatīyasakāravipulā	32	Visamavutti	
122	Jakāravipulā	16	Visamavutti	
123	Pathamajakāravipulā	32	Visamavutti	
124	Tatīyajakāravipulā	32	Visamavutti	

Vaññavutti verse types are relatively easier to understand and analyze, but there is something to consider particularly about symmetric types. Formulas given in *Samavutti* are only for one foot, not one line or two feet (or full two lines or four feet) as we have seen in the other group. As a result, you have to edit the input passage by cutting one line of the verse into two, otherwise ‘over- required’ will be the case. It is better to see a real example, as shown in Figure 9.5.

Figure 9.5.: Analysis of *Tanumajjhā* verse type in edit mode

In the example, a line of *Tanumajjhā* (6-syllable T-Y pattern) type is shown.³⁴ In the text, we have stanzas in full form. If

³⁴The sample is taken from the 6th stanza of *Mahāpaṇāmapāṭha*, under

you copy the whole line and analyze it, ‘over-required’ will show. You have to cut the line by using edit mode (✂ button), then you will get the correct result. In semi-symmetry and asymmetry groups, the formulas are given for one whole line, so you do not have to do likewise in these groups.

There are some other notes the users should know. First, Upajāti (Vut.65) is a mixed-up of Indavajira (Vut.63) and Upendavajira (Vut.64), or sometimes from other types. So, it has no fixed pattern to detect, and left out from analysis (no ID).

Second, Sasikalā (Vut.93) and Maṇigūṇanikara (Vut.94) use the same formula (14 *lahus* plus 1 *garu*) but the latter has two pauses, after the 8th syllable and the last one. These two are not distinguishable in the program, even if they are different.

And the last note, verse types ID 113 (Takāravipulā) to 124 (Tatīyajakāravipulā) have no references in *Vuttodaya*. These types follow the logic of the previous Bhakāravipulā (Vut.124), Rakāravipulā (Vut.125), and Nakāravipulā (Vut.126).³⁵ I add them here because they are easy to implement the analysis.

There are minor things I have not mentioned. The users should play around and find out how things work. Once you know this kind of tool exists, you can do research in Pāli prosody easier than before. For metrical composition, this tool can be your test bench. With edit mode mentioned above, you can compose a short verse in real time. Furthermore, the program also provides *search by meter* to help the users find a matched word to a pattern required. We will talk about this in the related modules.

Buddha-vandanā gantha-saṅgaho in the Añña.

³⁵These are added in Vuttodayamañjarī by Ven. Gandhasārābhivaṃsa.

Part III.

Pāli Collection


10

Browsing and bookmarking

Before we learn how to access to a Pāli document in our collection, it is better to be familiar with the collection first.

There are two kinds of documents used in the program: CSCD and the Extra. The former are those bundled with the program, whereas the latter should be empty at the first run.

The Chaṭṭha Saṅgāyana CD (CSCD)¹ is the best and most complete collection of Pāli literature nowadays.² This is the main corpus we use in Pāli studies. That is to say, Pāli Platform is a one-stop package. You have everything essential for Pāli learning in one place.

The Extra is a collection outside CSCD. It is just a directory that can be set by the user (in General Settings).³ Two formats are recognized as a document: XML (conformed to CSCD), and plain text (with .txt extension). Once you have documents in the Extra, you can open them in the program's viewer, and you can analyze them with Tokenizer. When documents are added to the Extra while the program is running,  button in TOC Tree (see below) has to be pressed to make them visible.

A document in the collection can be accessed directly by TOC

¹Distributed by Vipassana Research Institute (VRI), tipitaka.org

²In our application, Roman script is used as the base text. And its encoding is changed from UTF-16 to UTF-8. All documents are structured in XML. All files, including TOC files, are packed into one zip file, named `romn_utf8.zip` in directory `data/collection`.

³By default, it is set to `data/collection/extra`.

Tree, either its tab in the main window or a newly opened TOC Tree window (using the Collection menu or the main tool bar), as shown in Figure 10.1.

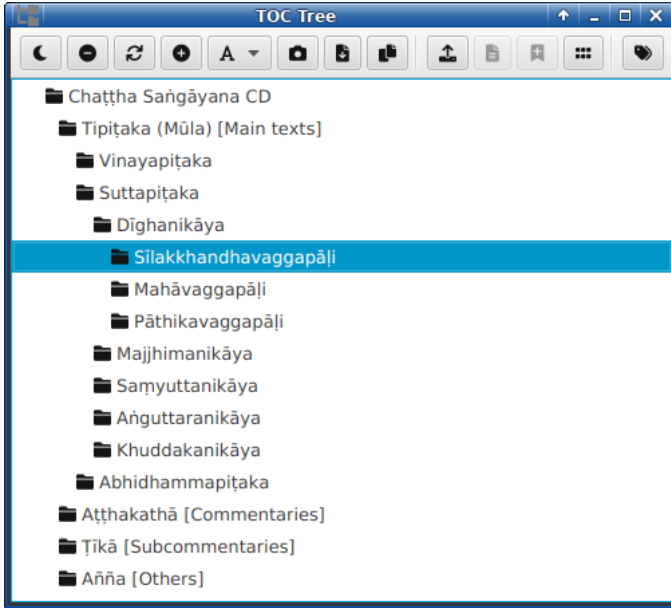


Figure 10.1.: TOC Tree window

The tree can be navigated by clicking its nodes. There are four main groups of texts: *Mūla* (the main texts), *Aṭṭhakathā* (commentaries), *Ṭīkā* (subcommentaries), and *Añña* (others). For the first three groups, if you are familiar with the structure of the Pāli canon, it will be easy to find what you want just by expanding the relevant nodes. Until you reach the text level, you will get a context menu (see Figure 10.2). From this menu, you can open the selected document either in the HTML Viewer (see Chapter 12) or in bare text.

You have two options for opening a document in bare text, with or without *notes*. The notes here are editorial insertion, enclosed with square brackets. They are not a part of the text. In certain situations, notes can disrupt the reading. So, they should be filtered out when you need to analyze just the text. Extracting bare text from a document in this way is useful

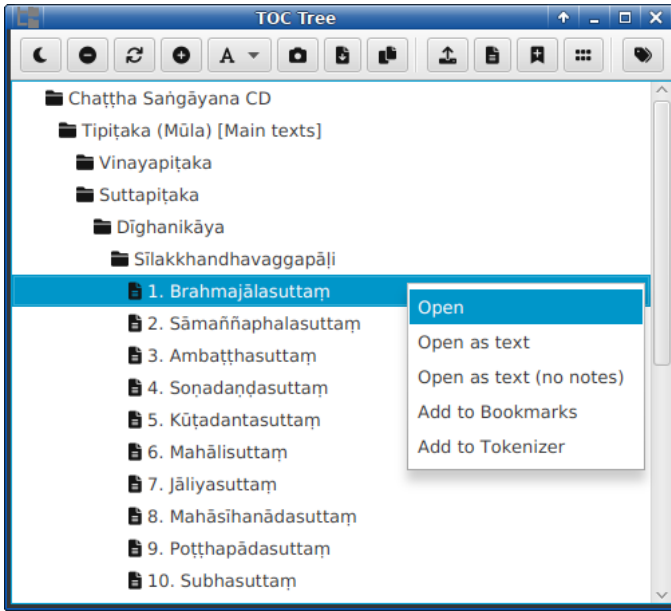




Figure 10.2.: TOC Tree window at text level

when we prepare text for Pāli Text Reader (see Chapter 18).

From the context menu, you can also bookmark the document as well as add it to Tokenizer. These two actions can also be done by drag-and-drop. There are some buttons provided in the tool bar corresponding to the functions mentioned above. Please check these by yourselves, as well as those unmentioned.

Adding documents to Tokenizer can be done with a whole bunch of texts in a tree node by using  button in the tool bar, because the context menu is available only at text level not the higher levels.

In the main window, when  button is pressed, or it is selected by the menu, Bookmarks window will show (see Figure 10.3). This window is a singleton. All documents you bookmark in TOC Tree, or somewhere else, will appear here. Actions available, both by the context menu or the tool bar, are similar to those of TOC Tree.

Bookmarks window is simple, so I will leave it to the users

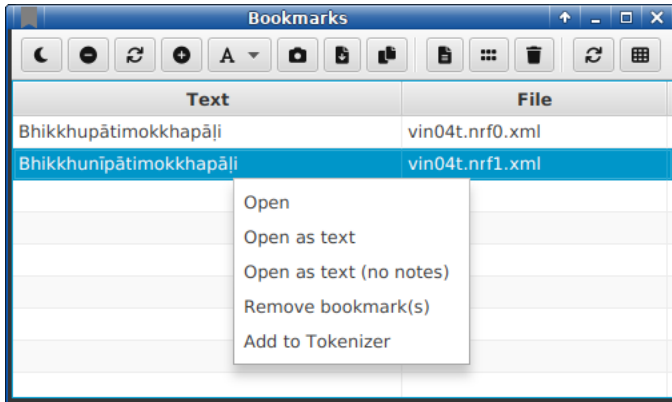


Figure 10.3.: Bookmarks window

to find out what else it can do. There are some entries that are preset in the bookmarks, two important documents difficult to find by new learners. If you lose the preset, you can bring it back by deleting the program's property file (PaliPlatform2.property) or move it away and restart the program.

11

Document Finder

Most new students of Pāli or Buddhism are not familiar with the structure of the Pāli canon. Finding a text by navigating TOC Tree can be difficult. That is why Document Finder comes in. With this tool, you can find relevant documents by entering a query. If you know a certain name or a text portion, you can find it more quickly than using TOC Tree. The tool is a part of the main tabs, and you can open it as many as you wish by clicking **Q** button in the main tool bar, or selecting it in the Collection menu.

What you should know first is there are two main kinds of search here: heading search and content search. In heading search, there are three fields available for searching: text name, book name, and group name. What is counted as text, or book, or group, is not exactly systematic. Those names come from the organization of CSCD. They correspond with the entries shown in TOC Tree. So, you may not find the document you need by entering your familiar text name.

If heading search fails, you can resort to content search. In this mode, full-text search will be applied, and the documents containing the query will be listed.

The search modes mentioned can be selected in one place, **✔** button. Figure 11.1 shows an attempt to find any document in Dhammapada by searching in book name.

In heading search, an asterisk (*) can be used as a wildcard. It can be used at any position (adding it to the last position is

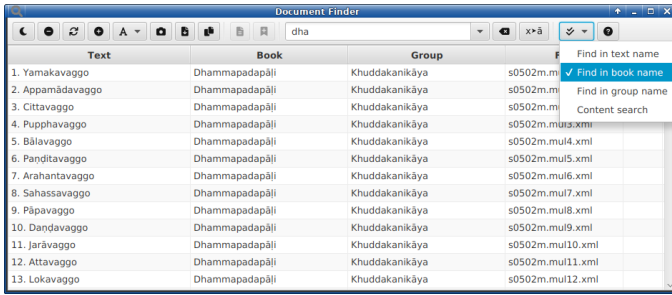


Figure 11.1.: Searching Dhammapada in Document Finder

unnecessary). This can be helpful when you cannot remember certain part of the name. Figure 11.2 shows a use of the wildcard in group name searching to find Visuddhimagga.

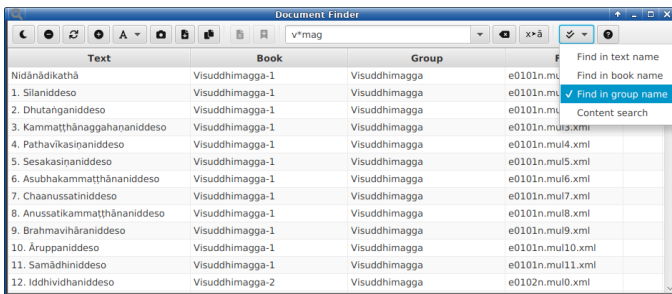


Figure 11.2.: Searching with a wildcard in Document Finder

Unlike heading search, in content search the query is always treated as regular expression, and you have to hit the Enter key to trigger the search, except when switching from other modes. In this mode, the last column of the result shows the number of hits in each document. Figure 11.3 shows the result of an attempt to find *Dhammacakkappavattanasutta* with the pattern `\b[Dd]hammacakkappa.*\b` (see Chapter 20 for more information).

Remember that this full-text search is case sensitive (know your search query can narrow down the result significantly), and brute (it searches directly in all files one by one, and you have to wait until it finishes). Since complex patterns take

11. Document Finder

more time to process, it is better to use exact query terms rather than meta-characters if you have those in mind. If the search fails, consider using a more sophisticated tool, such as Lucene Finder.

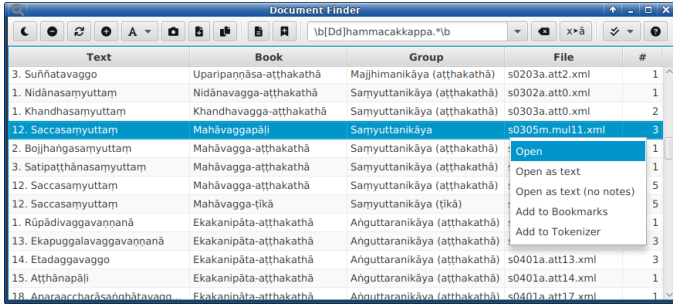



Figure 11.3.: Content searching in Document Finder

As shown in the picture, Document Finder has a context menu similar to that of TOC Tree. So, you can access to the document or do other actions in the same way.

12

Document viewer

After we know how to access to a document, in this chapter we will learn about the viewer. As mentioned earlier, we can view a document in two formats, XML or text. Here we focus only on the viewing of XML documents. I call this tool *Pāli HTML Viewer*.¹

When a document is opened, either by a context menu or  button in a tool bar, the viewer will show up. In its full form, the viewer looks like Figure 12.1.

There are main three panes: (1) the center, always present, displaying the text, (2) the right pane, opened by default, showing the text's information and related documents, and (3) the left pane, hidden by default, used for navigation. The right and left pane can be turned on and off by the three buttons in the tool bar.

The information shown in the right pane is obvious. You can also open further the related documents, if any, by using context menu (right click). The related documents are the texts that hierarchically related to the opened text. For example, if you open a Mūla (main) text, the related texts in Aṭṭhakathā (commentaries) will show. If you open a commentary, you will also see links to the related texts in Tīkā (subcommentaries)

¹Technically speaking, the document is converted from XML to HTML using SGML transformation, then opened in the HTML viewer. The transformation has strict rules. So, non-compliant XML files cannot be viewed correctly by the program.

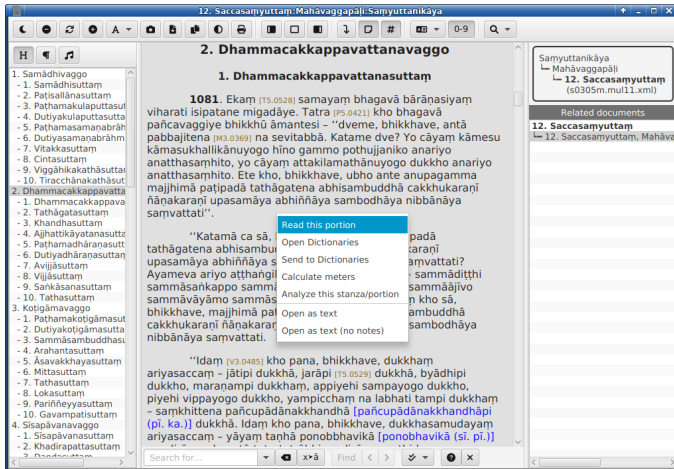


Figure 12.1.: Document viewer in its full form



level.

The navigator in the left pane is quite useful. You can jump to points in the document according to three criteria: heading jump (**H**), paragraph-number jump (**¶**), and stanza jump (**♪**). The last point of jumping is saved. You can return there by using the last jump button. To understand these you have to experiment with various set of texts.

When you search a string by pressing **Ctrl-F**, the search widget will show at the bottom of the window. There are three options for searching: case sensitivity, whole word search, and using regular expression. By default, the search is case-insensitive and not whole-word. The options can be changed by **✓** button. For a more advanced search, you can use regular expression (see more in Chapter 20).

When you right-click at a portion, or a selection, of text, a context menu will show up, as shown at the center of the picture. This allows us to do certain operations upon the selected text. The users should explore these by themselves. One explanation, though, **Send to Dictionaries** means the selected portion will be copied to the Dictionaries tab in the main window.

There are some buttons in the tool bar needed an explana-

tion. When  button is selected, the editorial notes embedded in the text will show. The notes can be seen in blue text enclosed with square brackets. When  button is selected, the reference points to other publications will show.²

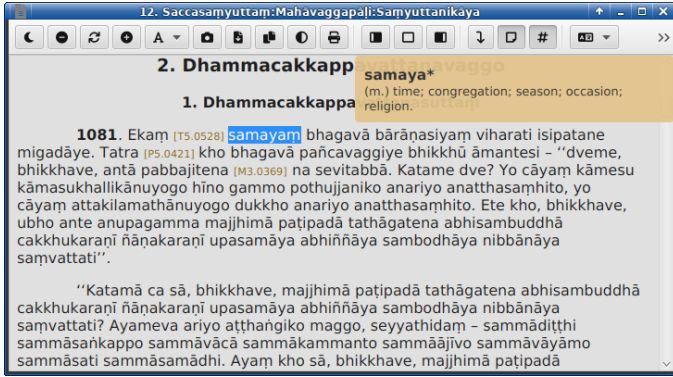



Figure 12.2.: Document viewer with Quick Dictionary

One useful feature of the viewer, is Quick Dictionary (using CPED). When a word is selected, it will be searched in CPED and the result is shown in a pop-up (see Figure 12.2). If the exact word is not found, the nearest result is shown instead with an asterisk (*) mark (see the picture).

Another exciting feature of the viewer is it can display text in various scripts, other than Roman, namely Devanagari, Khmer, Myanmar, Sinhala, and Thai. This can be done by  button with an additional option, whether numbers are also converted or not (0-9 button). An example in Myanmar script is shown in Figure 12.3.³

There are some considerations and limitations concerning script transformation. First, only Pāli characters are acceptable. Converting Sanskrit characters produces unexpected result. Second, you can transform Roman script to any other scripts, and vice versa. But you cannot convert a non-Roman

²The publications are V (the VRI edition), P (the PTS edition), M (the Myanmar edition), and T (the Thai edition). I have no idea what the exact editions these refer to, and what the numbers are represented.

³It is likely that you may encounter certain problem when displaying non-Roman scripts. See the solution in Section 2.3

12. Document viewer

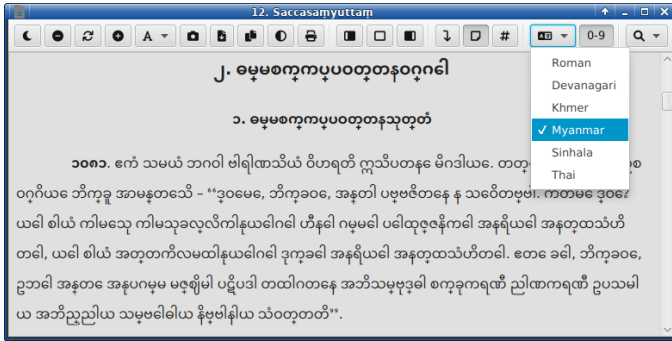


Figure 12.3.: Document viewer displaying in Myanmar script to other non-Roman.

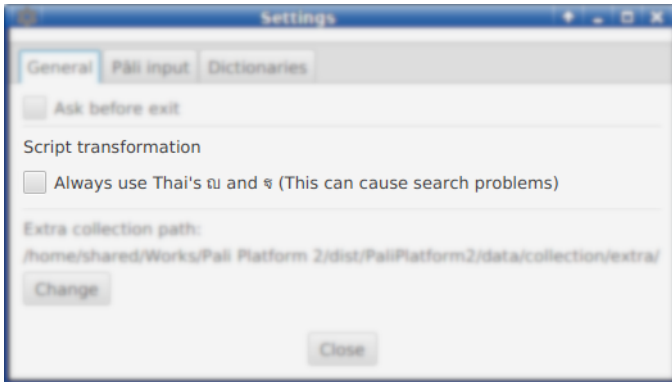


Figure 12.4.: General Settings of script transformation

Third, for to-Thai transformation, the users have an option in the General Settings (see Figure 12.4) whether only the special characters will be used or not. In Thai, \tilde{n} and th have two forms, with and without their lower part. The stripped forms are displayed automatically in their specific contexts, otherwise their full form (which looks less Pāli) will be used. If the users want to use these special letters exclusively, they can select the option. Remember that this can cause search problems in the native language. When converting from Thai to

Roman, this matter is not to be concerned because the program can read both forms.

And finally also about the transformation of Thai vowel *e* and *o*, as shown in Table 12.1, when Roman script is converted into Thai, only one form is produced. On the other side, converting from Thai to Roman can tolerate the variation of input.

Table 12.1.: Transformation rules of Thai script

Input	Output
tve	ตฺเว
vho	ว्हโห
ตฺเว	tve
เตว	tve
ว्हโห	vho
โวห	vho

13

Simple Lister




As we have learned so far, we can locate a document, open it, read it with a help from Quick Dictionary, or transform it to another script. Together with various grammar tools mentioned earlier, it is quite enough for learning Pāli. It is in fact incredible to traditional students, who may undergo more hardship without using such tools.

From this chapter on, we will see the true capacity of computer-aided learning. We will realize that data processing can give us insight about the subject in various dimensions. In this chapter I will introduce a tool that can show you all terms in the collection. And here is one of indispensable tools in modern Pāli learning.

I call this tool *Simple Lister*. It is ‘simple’ because it just lists terms and does something upon the list. However, because the tool is really powerful, you have to understand some strange options to get most of its capacity.

What is counted as ‘term’ here is simply a token. It is not a Pāli word in strict sense. A token, in the domain of computational Pāli, is a chunk of character string without in-between separators. So, a portion with punctuation marks will be split with those marks resulting in standalone tokens. That explains why *nti* is counted as a term here. All terms or tokens were prepared and put into the program’s database. All of them are extracted, or technically called ‘tokenized,’ from the

CSCD with their number of occurrences (frequency). All tokens are also normalized into lowercase letters.

Simple Lister is one of the tabs in the main window. It can be opened as a separate window by  button, or by the Collection menu. At first, it will show the top-most frequent terms. We can see the summary by using  button. The result is shown in Figure 13.1. With  button, you can choose between term and document summary. In the picture, term summary is shown.

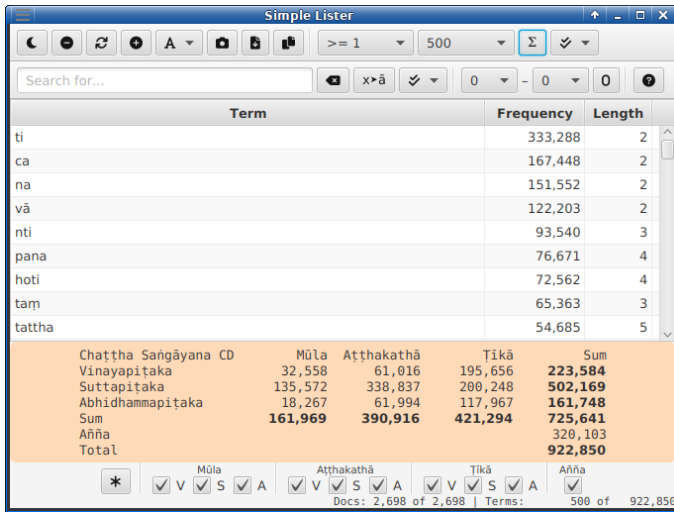


Figure 13.1.: Simple Lister window with term summary

In the result table, each row shows the term, its frequency, and length. The length is Pāli-sensitive, meaning *th*, for example, has 1 character long. So, you see *tattha* in the list has length = 5.

By default, the result is retrieved only 500 rows, ordered by top frequency. This makes the starting time is very fast. You can set the number of maximum rows by a drop-down option in the tool bar, ranging from 50 to 1,000,000 rows. The less you select the faster you get the result.

The table can be sorted by a specific column. You just click on the column header you want. But remember that the sorting is done only on the existing data limited by the maximum

13. *Simple Lister*


rows chosen. It does not retrieve new data.

At the bottom, there are text group selectors (V = Vinaya, S = Suttanta, A = Abhidhamma). You can include text groups so that you see only terms in these groups. Result from selecting all groups is fast, as well as an individual group. A combination of groups, not all of them, may take a little more time to process. When all groups are selected, and the maximum-rows is set to 1000000, the result is the list of all terms in the collection.

Another useful option is frequency range, the drop-down before the maximum-rows option. By default it is set to ≥ 1 (greater or equal to 1), meaning all in the range. You can select other criteria to see only what you are interested in. For example, by selecting =1 (equal to 1) you will see only terms that appear only once. These are more than 500,000 terms in the whole collection. That explains why 550000 is in the maximum-rows option, not shear 500000. Playing around with this option can bring you some insight about statistics of the textual data.

Now you can apply all the options mentioned to find the longest term in the collection. First, select all text groups. Second, select 1000000 maximum rows. When the result is shown, click the header of column Length twice. Figure 13.2 shows the result of what I have done myself.

If you choose the full frequency range (≥ 1) like me, you have to wait a little long.¹ It will be faster if you choose = 1 instead (very long terms are supposed to appear once), because less data will be processed.

Simple Lister also incorporates a powerful search tool. The search result comes up immediately after you enter the first character, or you can drag a word from other places and drop it here. You have four search modes: (1) Simple filter, (2) Using ? and *, (3) Regular expression, and (4) Filter by meter. These options can be selected by  button behind the search input. In the first three modes, the searching is done directly in the database. So, you do not worry that whether your selected maximum-rows is sufficient. In contrast, the meter search is done upon the retrieved data. You have to choose a suitable

¹It is processed under 10 seconds each step in my computer, pretty fast in fact. It is long because the result does not come up right away.

Term	Frequency	Length
avippavāsasammutisathanthasammutibhattuddesakasenāsanaggāhāp...	1	206
bhattuddesakasenāsanaggāhāpakabhaṇḍāgārikacivarapaṭiggāhakaci...	1	195
bhattuddesakasenāsanapaññāpakabhaṇḍāgārikacivarapaṭiggāhakaci...	1	193
bhattuddesakasenāsanapaññāpakabhaṇḍāgārikacivarapaṭiggāhakaci...	1	191
bhedānūvattakadubbacakuladūsakapaṭhamadutiyaṭiyakathinaabhi...	1	187
bhattuddesakasenāsanaggāhāpakabhaṇḍāgārikacivarapaṭiggāhakaci...	1	186
bhattuddesakasenāsanaggāhāpakabhaṇḍāgārikacivarapaṭiggāhakaci...	1	184
āsavavippayuttasāsavasamyojanavippayuttasamyojanīyaganthavipp...	1	182
sukkavisatthimūsvāvādomasavādapesuññabhūtagāmaaññāvādakaujjh...	1	163
āyatanadhātusaccaindriyapaccayākārasatipatṭhānasammappadhānai...	1	150
saṅghaganapaṇḍakatheyayasamvāsakatitthiyapakantakātiracchānag...	1	143
samyojanākiḷesamicchattalokadhammamacchariyavipallāsaganthaag...	1	142

Figure 13.2.: Top longest terms in Simple Lister

number of maximum rows.

The input acceptable in filter-by-meter mode is (case-sensitive): 1, 2, 4, l, g, n, s, j, b, m, N, S, J, Y, B, R, T, M, and L (see Chapter 9 for more information).² Be careful with 2 here. It means either g or ll. Figure 13.3 shows the result of ggg1 with 500 terms retrieved, comparing to Figure 13.4, which uses the input 2221 instead. So, a suggestion when you use the meter search is always use g (not 2) when you search with raw meter (1 and l bring the same result).

The last feature I will talk about is *grouping*. Grouping is an action upon the retrieved data. It groups terms by similarity. For example, you can see that how many terms starting with each letter by setting the start number to 1 (see Figure 13.5). There are some limitations to be aware of. First, characters here are not Pāli-sensitive. This means under the group of 'p-', for example, terms starting with 'ph-' are included. And Second, the operation is done upon the existing data, so you have to select a proper number of maximum rows.

Similarly, you can see that how many terms ending with the

²The weight 6 and 8 are excluded from the process because they are difficult to implement. You have to be specific. For example, using 24 or 42 for 6 and use 44 for 8.

13. Simple Lister

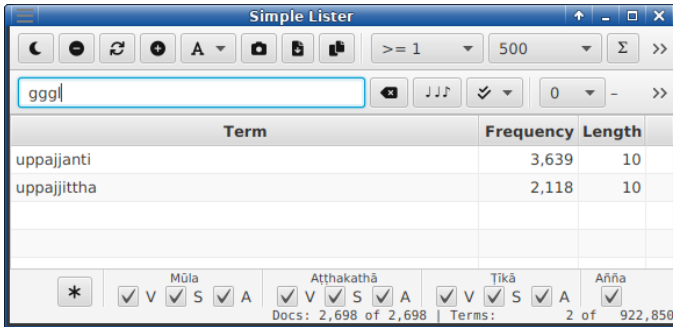


Figure 13.3.: Filter by meter 'gggl' in Simple Lister

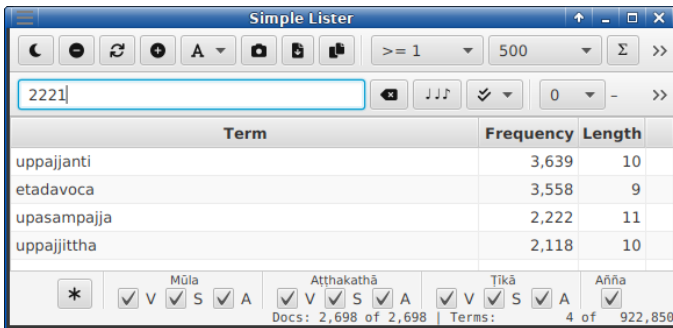


Figure 13.4.: Filter by meter '2221' in Simple Lister

same letters by setting the end number. Figure 13.6 shows the result of two-letter endings. Remember that the hyphen (-) includes zero or any number of characters. So, '-ti' means 'ti' itself and everything ending with 'ti'. You can certainly set both the start and end number to see a particular effect. Please experiment.

Term	Frequency	Length
p-	1,096,115	
s-	1,080,709	
t-	1,037,021	
a-	982,383	
v-	790,829	
n-	733,428	
k-	649,464	
d-	516,812	

Search for... x>ã 1 - 0 0

Mūla Atthakathā Tikā Anña

Docs: 2,698 of 2,698 | Terms: 31 of 922,850

Figure 13.5.: Grouping by the first letter in Simple Lister

Term	Frequency	Length
-am	1,840,064	
-ti	1,195,510	
-na	601,512	
-vā	415,869	
-to	352,545	
-sa	343,795	
-ca	281,668	
-ha	260,480	

Search for... x>ã 0 - 2 0

Mūla Atthakathā Tikā Anña

Docs: 2,698 of 2,698 | Terms: 252 of 922,850

Figure 13.6.: Grouping by the last 2 letters in Simple Lister

Part IV.

Advanced Search Tools

In Chapter 11, Document Finder, a tool that can locate required documents, is described. It is simple and handy, and it works fine in common situations. If the tool is not powerful enough so that you still cannot find what you want. Lucene Finder is the final answer to all sophisticated search. It is really easy to use and powerful, but you have to learn some new expressions of the search query.



Figure 14.1.: Apache Lucene's logo

This tool utilizes Apache Lucene 9 as the search engine. You cannot use it yet in the first run. You have to build a Lucene index first. Lucene index is a group of files generated by Lucene with options specified by the users. The files reside in a directory (data/index/main by default). The users can build indices with a variety of options as many as they need, but only one index is selected to use at a time.

14.1. Options for indexing

Lucene Finder can be opened by its button in the tool bar, or by the Collection menu. At the title bar, the name of index's directory is shown, together with number of documents indexed. So, at the first open, you should see [main:0]. It is better to learn the options before you build the index (see Figure 14.2).

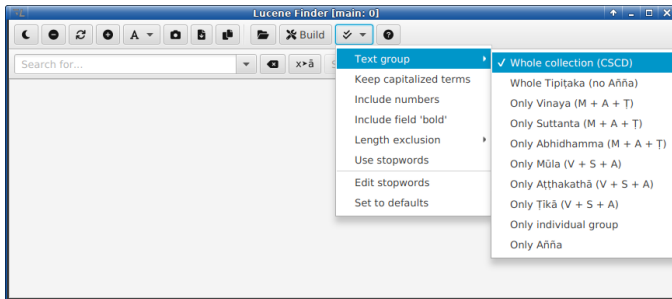


Figure 14.2.: Options for building Lucene index

(1) Text group The first option you should consider is which group of text you will work with. There are several options available, running from a single set of text to the whole collection. In general, indexing the whole collection is suitable for most use cases. If you have a particular purpose to work with a set of text, you can make the index for it and save in a different directory.

(2) Keep capitalized terms This option allows capitalized terms to be indexed and searched. Normally, we lowercase all tokens when indexing. That can save space and reduce search complexity. With English corpora, it is a common practice. But in Pāli, particularly in the collection we have, capitalized terms are those starting sentences. So, they are embedded with additional meaning. Keeping capitalized terms, therefore, allows us to do a more refined search. By default, this option is not set.

(3) Include numbers In the body of text, there is no use of numbers in Arabic form, except paragraph numbers. Numbers can also be found in editorial notes. When this option is set, you can search numbers in the text. This has little use in most contexts, so this option is turned off by default.

(4) Include field 'bold' If you have ever opened a document in the program's viewer, you will see that most of them has portions of text running in boldface. I call this field 'bold' (learn more about fields in the next section). The bold text signals that there are explanations on that part in the text's commentaries/subcommentaries. Should we include this? A typical answer is 'No.' Because the bold part of text is already collected in other fields. You lose nothing when 'bold' is excluded. However, when you really need to search text in 'bold,' this field has to be included in your index.

(5) Length exclusion You can exclude terms by their length. We have three options here: `==1`, `<=2`, and `<=3`. The first option is the default, meaning all single letters are excluded. If you choose the second, all terms with 2 letters and 1 letter long will be excluded. So, you cannot find 'ti' or 'ca' or 'vā' in this case. Cautiously use this option, or else you will lose a good number of words.

(6) Use stopwords Finally, you can exclude specific terms by adding them into a list. Technically, we call such terms *stopwords*. The stopword file is hardcodedly set to `data/rules/stopwords.text`. You can edit it by the menu provided, or using an external editor. By default, this option is turned off.

Once you finish your option choosing, then you hit the **Build** button to create the index. The program will ask you for the output directory. You can create a new one at this step. After that the program will do its job, and you need to wait. Indexing is heavily resources-consuming, so you should not do other things meanwhile. It will not take long.¹ For the whole collec-

¹In my old dual core 32-bit laptop, it takes only 2 minutes for the whole collection with default settings.

tion, if you see 2698 (the number of all documents) in the title bar, the index is successfully built.

14.2. Description of fields

Thanks to XML format of files in the corpus, we can search by selecting relevant fields. When the texts are indexed, not the whole bunch of them is processed, texts are pigeonholed into fields. Fields were pre-organized by the collector of CSCD. Table 14.1 show all fields used in Lucene Finder. Selection of fields can be done by ☰ button in the second tool bar.

Table 14.1.: Fields used in Lucene Finder

Field	Group	Description
bodytext	body	the main body of text
center		text with center alignment
indent		text with an indent, outside bodytext
unindented		text without indent, outside bodytext
nikaya	headings	e.g. Vinayapiṭake
book		e.g. Pārājikapāli
chapter		e.g. Pārājikakaṇḍaṃ
title		e.g. Paṭhamapārājikaṃ
subhead		e.g. Vinītavatthu
subsubhead		rare, arbitrary, e.g. Nidānagāthā
gatha1	gatha	the first line of a stanza
gatha2		the second line of a stanza
gatha3		the third line of a stanza
gathalast		the last line of a stanza ²
note	note	editorial notes, text in [...]
bold	bold	text in boldface

²When a stanza has only 2 or 3 lines, ‘gathalast’ is always the last. The unused lines are skipped. That is to say, ‘gatha1’ is always present, ‘gathalast’ is almost if the stanza has more than one line, and ‘gatha2’ and ‘gatha3’ appear only in long verses.

Sometimes it is hard to tell, if we do not look into the XML files, one field from another. For example, field ‘center’ looks similar to those in the headings group, but it normally contains a longer portion of text, whereas text in headings is typically short, if not only one compound word. Knowing all these can help you search more effectively. You can, for instance, search text only in verse form, or more specifically in any line of verse form.

Only ‘bold,’ if included in the index, forces exclusively one-field search. Other fields can be selected in combination. To ease the user, fields are grouped so that you can select multiple fields at once. So, you see two modes of field list: simple and detailed mode.

14.3. Lucene simple search

After you build the index and know how fields work, then you can use the search function. Only thing you have to do is enter some word and hit the Enter key, or Search button. I will show you an easy case first. If you know exactly what you want to find, and know which fields it resides, you should enter the full word and select the fields accordingly. This leads you to the result immediately. Figure 14.3 shows the result of finding ‘*dhammacakkappavattanasuttam.*’

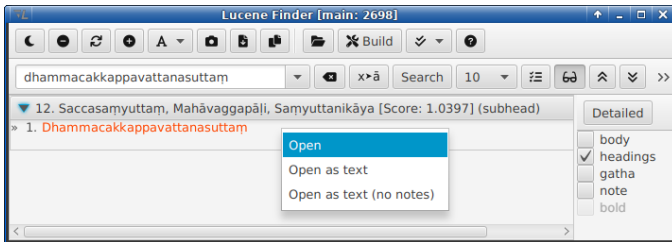


Figure 14.3.: Simple one-term search in Lucene Finder

Options concerning search result you should know are the number of maximum results (10–100), the field selector (☰ button), and the text fragments (📄 button). It is really unnecessary to use a high number of search results, except when

14. Lucene Finder

you want to see a lot of them. The main reason is the results are ranked by scoring, and the most relevant result is shown first. The method of scoring is the Lucene's default and mathematically complex. You do not need an explanation. Just keep in mind, a higher score means more relevant. Sometimes, the rank of scores does not make sense to you, or even to me, because we do not know its internal conditions. The output, however, is mostly reliable.

The text fragments are parts of the result that match the query, showing with highlight and their context. This is a useful feature, but unfortunately buggy. If you enter full words, you are likely to see the fragments. If you use wildcards (see below), it is less likely you will see them.³

Once you have search results, you can open the documents shown by using context menu (right-click), as shown in the picture.

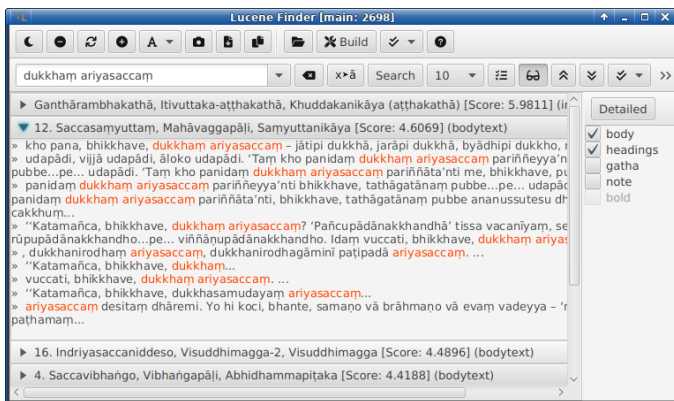





Figure 14.4.: Simple two-term search in Lucene Finder

A search query is not limited to one word. You can enter multiple words as many as you want. The result in this case will be the documents that contain all the words you enter (see also logical operators below).⁴ Figure 14.4 show the result

³Lucene has its highlighter module, and we use it. But sometimes the highlighter misses the result completely. I compensate this with my implementation. It works fine, but not perfect. That is why you still see nothing in text fragments sometimes, even if the query is accurately matched.

⁴In other words, all query words will be joined with AND operator. This is

when searching ‘*dukkham ariyasaccam*’ (you might know that the Dhammacak has this phrase). This time you see several documents. You have to check the fragments whether that document is what you need. Note that these two words may not sit together, as shown in some of the fragments. By using  and  button, you can collapse and expand all fragments quickly. Moreover, if you feel that the fragments are too short, you can set ‘Show whole lines’ option by  button in the second tool bar.⁵

In Chapter 11 we talk about content searching to find a document with Document Finder. Using that tool is similar to simple search here in some respect. Once the index is built, searching with Lucene Finder is faster and more flexible. Also, you can see inside the documents if they are really what you want. Moreover, if you do not have an exact word to find, in Document Finder you are in a hardship, but in Lucene Finder there are ways to deal with a vague query, as we shall learn in the next section.

The best practice in finding unfamiliar words is checking them with Simple Lister first, then you can take the words from the Lister to search here, or in Document Finder, by drag-and-drop. Keep in mind that entering full-word query always bring better search result.

14.4. Lucene query syntax

To use Lucene effectively, you have to know its syntax. And this is the true power of Lucene. Some parts of the explanation below may look difficult or too complicated for most students of Humanities who use this program. I will try my best to keep them easy to understand. Planting with a hoe is fine, but today we also have to learn to drive a tractor.

the default setting. If you want OR joining, you have to use the OR operator explicitly.

⁵Showing whole lines in the fragments does not work in all cases. It may work with wildcards and regular expression, but fails with other complex search schemes.

14.4.1. Using wildcards

Here we meet our old friends. Lucene accepts two wildcard characters: question mark (?) and asterisk (*). The former stands for a single character, the latter multiple characters. We also use these in other parts of the program.

Here are some examples: Entering ‘dhamm?’ can match ‘dhammo’ or ‘dhammā’ or ‘dhamme’ but not ‘dhammam.’ Entering ‘dhammacak*’ can match ‘dhammacakkhu’ or ‘dhammacakkam’ or ‘dhammacak-whatsoever.’

One caveat, Lucene does not allow wildcards in the first position. You cannot find ‘?hammo’ or ‘*ammo’ here. But there is no such limitation in Simple Lister (Chapter 13) and Tokenizer (Chapter 15). If you really want to search in this way, use Simple Lister to find the exact word first and bring it here, or use Tokenizer to make a custom index and use its search function instead.

14.4.2. Using regular expression

Regular expression enhances the use of wildcards substantially, like you replace a slingshot with a machine gun. Other parts of the program allow using regular expression as well. The downside of this is it takes a steep learning curve. The syntax also looks bizarre, if not terrible, to non-technical users. Teaching how to use regular expression is not an easy task either. However, in Chapter 20 I have a brief treatment of this topic.

To use regular expression in a query, just enclose it with slashes /.../. For example, entering ‘/dhamm[oāe]/’ can search either ‘dhammo’ or ‘dhammā’ or ‘dhamme.’

14.4.3. Using fuzzy query

As explained in the Lucene’s API document, fuzzy search utilizes *Damerau-Levenshtein Distance* algorithm. We can use this mode by adding a tilde (~) to the end of a search term.⁶

⁶Optionally you can put a number after the tilde, like ‘dhammo~1.’ The number specifies “the maximum number of edits allowed.” The value can be either 0, 1 or 2 (default). I cannot give you any clearer explanation of this. Just try it yourselves.

This will find terms that look similar to the query.

For example, in my search of ‘dhammo~’ I get these in return: dhammo, dhammā, dhamma, dhamme, dhammaṃ, adhamma, adhammo, dammi, dhajo, etc.

14.4.4. Using proximity

Normally when we enter multiple search terms, we do not use quotation marks (“...”). If the proximity of the terms is taken into account, we use quotation marks. That is to say, if you want to search exactly ‘dukkhaṃ ariyasaccaṃ,’ enclose the phrase with double quotes (hence “dukkhaṃ ariyasaccaṃ”).⁷

In addition, you can specify the distance by adding a tilde (~) at the end plus a number. For example, entering “pana bhikkhū”~2 can search ‘pana’ and ‘bhikkhū’ within 2 words apart. This can match either ‘pana bhikkhū’ or ‘pana something bhikkhū’ or ‘pana something something bhikkhū’ (see Figure 14.5).

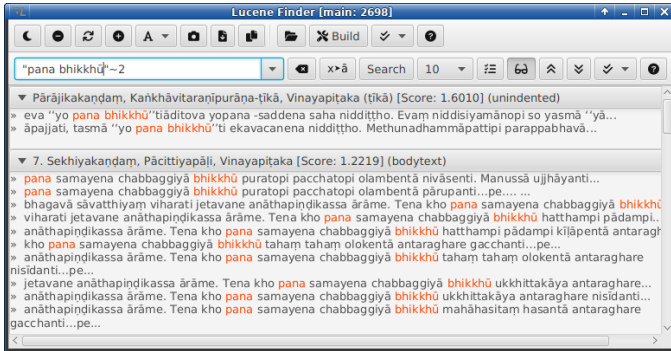


Figure 14.5.: Proximity search in Lucene Finder

14.4.5. Using range

This mode can be useful when searching a range of numbers. We normally do not use this because we hardly search for num-

⁷This is equivalent to “dukkhaṃ ariyasaccaṃ”~0.

14. Lucene Finder

bers in the texts, except you want to find certain paragraph numbers by building the index with numbers included.⁸

To search a range, use TO operator (all caps) and enclose the phrase with square brackets, for example, '[120 TO 250].'

Range is not limited to numbers. It can be used with terms as well. But it still has little use here, because range is not Pāli-sensitive. You have to think it in English. Figure 14.6 shows some results of a search for '[dhamma TO dhamme].'

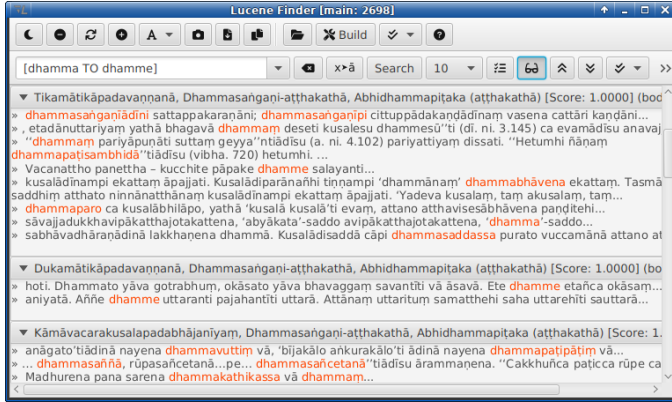


Figure 14.6.: Range search in Lucene Finder

As you see in the picture, every term between 'dhamma' and 'dhamme' is found, but not 'dhammo' because *o* comes after *e*. But if you search '[dhamma TO dhammā]' instead you will also see 'dhammo.' That is not we expect.

14.4.6. Using term boost

When you search multiple terms or phrases, you can put unequal weight to each term to make the heavier term has higher degree of importance or relevance. This is called *boosting*. It is can be done by adding a caret (^) symbol after the term. For example, in searching 'dukkhaṃ ariyasaccaṃ' if you put more

⁸Paragraph numbers in the collection are not congruous. Some are put as a range, some are missing. Some documents do not have numbers at all. This warns you that not every number is searchable, even if it is seemingly to be there.

weight to ‘*dukkham*,’ say, 10 times, you can enter ‘*dukkham^10 ariyasaccam*’ (see the result in Figure 14.7, compare it to Figure 14.4).

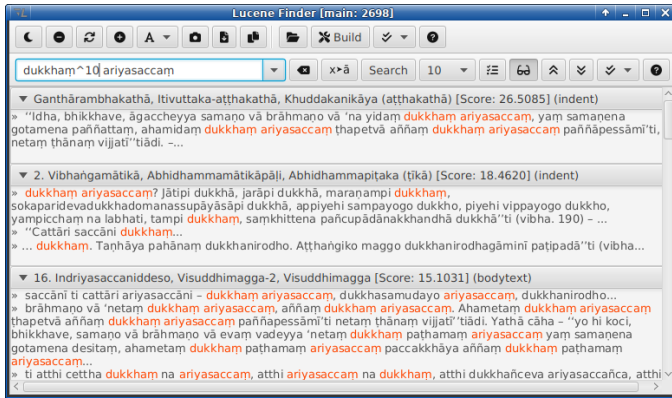


Figure 14.7.: Term boosting in Lucene Finder

14.4.7. Using logical operators

To refine our search, we normally use multiple-term queries. When multiple terms are present together, they have a logical relation to each other. This relation is denoted by operators, which we have five of them here: AND, OR, NOT, plus (+), and minus (-). The first three must be in uppercase. By default, or by the absence of any operator, terms are related with logical AND, as I show you in Section 14.3 above. If you want to use other relations, you have to explicitly use the operators.

Plus (+) means always present, minus (-) means always absent, and no symbol means optional. These two symbols work in a similar way as AND, OR, and NOT, but have their own meaning and use. Normally, we use the two sets of operators exclusively, but they can be mixed.

Here are some typical uses of these operators.

(1) *dukkham* NOT *ariyasaccam* This finds documents containing *dukkham* but not *ariyasaccam*.

(2) “**dukkhaṃ ariyasaccaṃ**” AND (**vijjā** OR **āloko**) This finds the phrase “*dukkhaṃ ariyasaccaṃ*” with *vijjā* or *āloko*.

(3) (**dukkhaṃ** OR **dukkhasamu***) AND **ariyasaccaṃ** This finds *dukkhaṃ* and *ariyasaccaṃ* or *dukkhasamu** and *ariyasaccaṃ*. Using a wildcard can save some typing but also can bring irrelevant results.

(4) **-dukkhaṃ aniccaṃ +anattā** The result of this must contain *anattā* but not *dukkhaṃ*; *aniccaṃ* is optional.

(5) **-“dukkhaṃ ariyasaccaṃ” +vijjā** This finds documents having *vijjā* but not “*dukkhaṃ ariyasaccaṃ*.”

14.5. Concluding remarks

Lucene is really a powerful search tool. To use it at full capacity, you have to learn how to build index with options that agree with your specific needs. You have to know about data, i.e. the fields that are used to structure the documents. And most importantly, you have to know its search syntax. I use a lot of space for this chapter because non-technical users are usually frightened by complicated tools, so they need a friendly guide.

What we have learned so far is just one part of Lucene’s comprehensive features. Lucene has many functions that can be used in Natural Language Processing (NLP), beside the Information Retrieval (IR) system that we utilize. Even so, we just touch the surface of its IR capacity. The software can do much more than I have shown you here. However, I think what we know here is enough for Pāli learning and research. Now you should have ideas to play around to deepen your understanding both of the tool and of the language.

This is the most difficult module in the program to write. I spent more than a month to implement this alone. In a nutshell, Tokenizer is a mini version of Lucene. Why we need this if Simple Lister and Lucene Finder can do the same job?¹ If you find that both modules mentioned are enough for your uses, it is not necessary to use Tokenizer. However, there are scenarios that this module is needed or helpful:

(1) You can index custom documents in the Extra with Tokenizer If you have your own collection and want to search or make its term list, you have to use this. By putting your documents in the Extra and add them to Tokenizer, you can get the term list and you can search for a document you are looking for.² Even though the search function in Tokenizer is not so powerful as Lucene Finder, it is enough to get the job done.

(2) You can add any document in the collection to Tokenizer As we have seen in Lucene Finder's options for indexing, we cannot

¹As a matter of fact, I wrote Tokenizer before Simple Lister and Lucene Finder in the hope that I could get rid of Lucene from the program's library and reduce the database side. After I finished the module, I realized that I had made a wrong decision. I thought that after all Simple Lister is needed for its simplicity and speed and Lucene Finder is needed for its superb search function.

²For plain text documents, all tokens are put into the bodytext field.

15. Tokenizer

select documents arbitrarily. There are preset groups of texts to be indexed. In Tokenizer, in contrast, we can select documents freely. Then we can search or explore the term list only in the texts we are interested in.

(3) You can create a custom term list in Tokenizer This is a consequence of the previous item. When you select only a text group that interests you, and make the term list out of it. You can use this list in Declension Table and Conjugation Table to check against the terms produced, alternatively to the whole term list.

(4) Capitalized terms are analyzed statistically in Tokenizer If you take capitalized terms seriously, this can give you more detail on this matter than Simple Lister or Lucene Finder. It calculates the percentage of each term found as capitalized. This may look trivial to other languages, but in Pāli it is informative. You can know which terms are normally used as sentence starter. In Tokenizer, you have no option for normalizing capitalized terms, because it is always kept as such.³

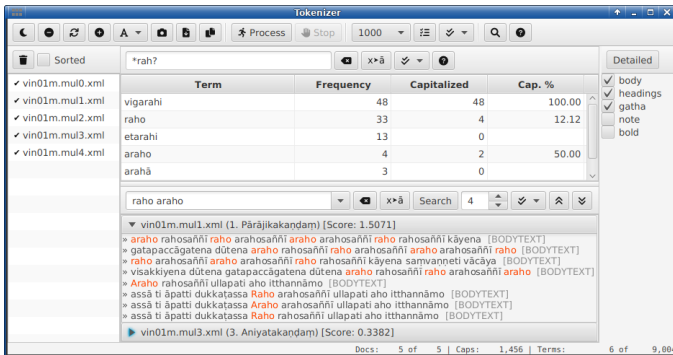


Figure 15.1.: Tokenizer window in full

Now I will explain briefly how to use Tokenizer and what else you should be aware of. First, the module is one of the tabs in the main window, which is known by other parts of the

³As a result of this implementation, I drop the calculation of terms' length from this module. For that information, use Simple Lister instead.

program as the ‘main Tokenizer.’ We can open it as separate windows as many as we want, but only the main Tokenizer is the target of the addition by context menus or tool bars.

This leads us to two ways of adding documents into Tokenizer: by context menus (or tool bars) and by drag-and-drop. There are three places that provide you a document list: TOC Tree, Bookmarks, and Document Finder. If you use the context menu (right-click) from these to add documents, the addition will be done in the main window’s Tokenizer tab. But you can drag-and-drop freely from these three modules to any Tokenizer opened.

Term filtering has the same function as that in Simple Lister, so as the field selector in Lucene Finder. Search pane is not visible by default. You have to click **Q** button. You can drag-and-drop a term in the list to the search text field. A multiple-term query uses logical OR relation (unlike Lucene Finder which uses AND by default). You cannot use Lucene syntax here. Only terms in full form are accept as valid query. No special symbols are used. The search result is similar to that of Lucene Finder, with slightly different display and options.⁴ Figure 15.1 shows Tokenizer window in its full form.

There are many things I have not talked about. I leave them to the users. You should explore by yourselves what else you can do. Try right-clicking here and there, and set various options to see their effect.



⁴Score calculation in Tokenizer and Lucene are different, so you should not expect the same ranking in both. For Lucene, I do not know exactly. In Tokenizer, I just simply use logarithmic TF-IDF (see Wikipedia for more information).

Part V.

Miscellaneous Tools

16

Pāli Text Editor

In this part we will examine general tools that can be used for various purposes. The first I introduce here is Pāli Text Editor. It can be opened by  button in the main tool bar. To open the editor with an existing text file, use  button instead. The function of these two buttons can be invoked also by the File menu. This is a simple text editor with basic functions, but supports to Pāli language are added, e.g. we can type in Pāli text easily with built-in input methods (see Section 2.4).

I will not explain the basic functions of the editor, which are familiar to most computer users and easy to learn. What makes the editor useful to Pāli learners is text processing tools provided. You can see these in menu Tools (see Figure 16.1).

By using tools, you can convert text from Roman script to other scripts or vice versa. You can remove diacritic marks from Roman text, decompose them, or re-compose them. You can calculate raw meter, send a portion of text to prosodic analyzer (see Chapter 9), or Pāli Text Reader (see Chapter 18). You can change the text to lowercase or uppercase. You can sort a list of terms by Pāli order. Finally, you can change Pāli characters into T_EX format or vice versa. I added the last function for my own use because I use a lot of such conversion in my book writing.

The best way to understand all these is to play with them. One caveat, some operations are undoable, particularly when you operate on a selected portion of text, but some are not

16. Pāli Text Editor

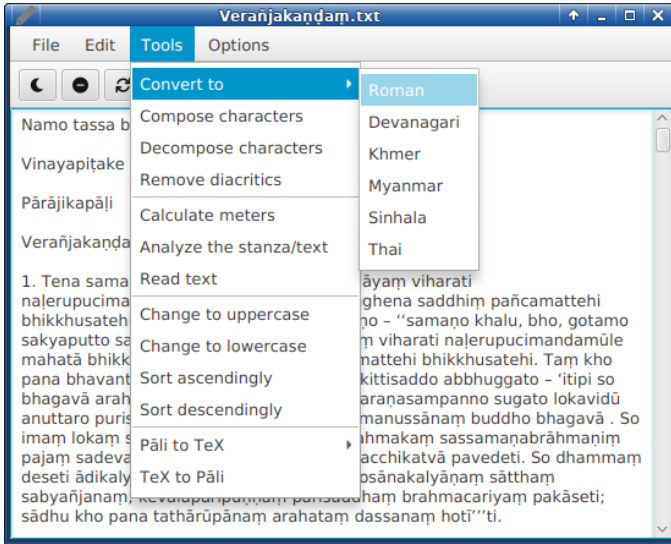


Figure 16.1.: Text-processing tools in Pāli Text Editor

(even with selected text). So, it is advisable to save your document before you use any tool, unless you are familiar well with the behavior of the actions. Some critical operations do not make change to the original text. They create new text in a new editor instead. Script conversion is a marked example.

Batch Script Transformer

This tool helps you do script transformation for multiple files at a time. This can save time if you have many Pāli text files needed the conversion. The tool is simple and straightforward. It works only with text files. You can open the transformer by menu File>Batch Script Transformer.

First, you have to add files by hitting the button provided, select the output script you want, then hit the Convert button. That is all. Figure 17.1 shows the result from my playing.

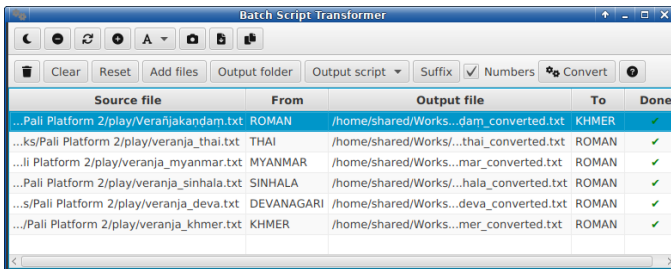


Figure 17.1.: Batch Script Transformer window

There are some other things you should be aware of. You can convert from Roman script to any of other scripts, but other scripts can be converted only to Roman. You can set the output folder by the button provided, otherwise you have to use



17. Batch Script Transformer

suffix added to the output files' name. The default suffix is '_converted.' You can change this by its button. You can choose whether numbers are included in the conversion or not. This option works only in the transformation from Roman to other scripts. The Reset button is used after a transformation is successfully done so that you can do it again, possibly with a different output script.

18

Pāli Text Reader

To address this question, “What is the best way to make Pāli text reading easier?” I have developed this module. In the document viewer, as we have seen in Chapter 12, we have an embedded dictionary that can give us meaning of a word at a time. This can be helpful when you scan the text and get stuck with some words, but it is too cumbersome for deliberate reading.

Here is how the reader works. First, you open the reader window by  button in the main tool bar, or in the Collection menu. Then, you select a portion of text, either in the HTML viewer or the text editor or even an external editor, copy it and paste into the reader. In the HTML viewer and text editor, you can open the reader directly by their context menu. To paste the copied text to the reader, you can use  button in its tool bar, or press `Ctrl-V`.

Now I will explain the conceptual idea behind the module. Please read the following carefully, especially when key terms are introduced.

Once the reader gets the raw text, the reader splits the text into sentences. What is counted as a sentence here is any section that starts with a capitalized term. It can be very short (only one word), or very long (hundreds of words). That is to say, sentences are determined by the compilers of the text, not options able to set by the program.

As we have immutable text in the collection, when we paste

portions that have the same arrangement into the reader, we will always get the same individual sentences. By this way, each sentence is unique by its components. Even if two sentences are the same in grammatical sense, they can be different to each other if their arrangement is different (punctuation symbols do not count).

When sentences are strung together, they constitute a sequence. So, when you paste a portion of text, you work with a sequence that have several sentences. That is to say, the reader has no concept of paragraph or passage or discourse. Its fundamental element is *sentence*; a collection of those elements is *sequence*.

Now I hope you understand our technical terms here. If not, you may not fully understand how the reader, also Sentence Manager, works. Once you have a sequence in the reader (it must have at least one sentence), you can save it, along with its sentences, and load it afterwards. Since sentence is the basic unit, sequences can share the same sentences. When several sequences are saved into a same place, if some sentences already exist, you will be asked to replace them or not. That is because the program save only one instance of a sentence. The sentence's identifier is call *hash*.¹

By the infrastructure described above, we can also add translations, and/or explanations, to a sentence. By this model, sentences are separated from their contexts. We have the sentence pool. When we read a sequence, it picks its sentences, which may have translations embedded, from the pool. You may see this as a drawback or limitation. However, this treatment is suitable for the repetitive nature of the Pāli text, I think.

Now you know we can add translations to a sentence. The last key word you have to know is *variant*. A variant is a variation of translation identified by its author or related information. You can see a variant simply as an author of translations. But one author can also be represented by several variants, or seen as versions. By the use of variant, we can add translations, including any native language², as many as we need to a

¹Hash is a string of hexadecimal numbers. It is calculated from its corresponding sentence. Each sentence has its own hash. A duplication is possible but very very unlikely. Technically, I use MD5 digest calculation here.

²By native language, I mean the language or locale of the users recognized

single sentence. This can demolish the illusion of one correct translation.

In the program bundle, I have already added many sentences and several sequences. With these examples, you can work further by your own. Figure 18.1 shows a sentence in the reader with translations.

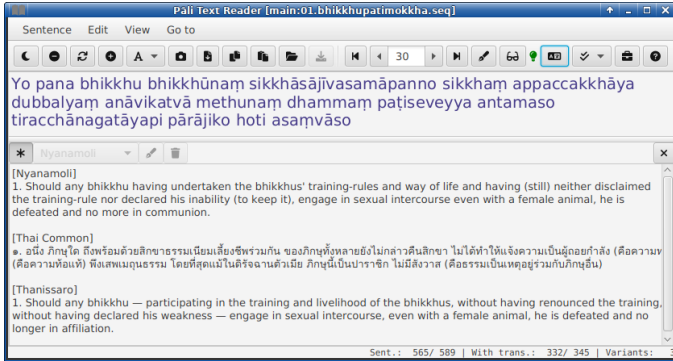


Figure 18.1.: A sentence with translations in Pāli Text Reader


I will not explain in detail about how to use particular commands in the menu or tool bar. You can learn those by yourselves. But some points are worth mentioning. When you see the light bulb turns green, it signals that the sentence has a translation. You can open the translation pane by **A** button, or hit **Ctrl-T**. The translations can be shown one variant at a time, or all of them at once (use ***** button).

On the status bar, a summary is shown. This can tell us how many sentences in this sequence. For example, 565/589 means we have 565 unique sentences and overall 589 sentences (i.e. some are repeated). The number of sentences having translations is also shown in the same manner, e.g. 332/345. And the number of variants used in this sequence is shown in the last part.

Translations can be edited and added here by **✎** button (only when only one variant is shown at a time). If you want to add a new variant, you have to add the variant in Sentence Manager first (see Chapter 19), and then use menu **Sentence>Update**


by their system. So, fonts for that language are always available.

variant list. For a massive addition of translations, using Sentence Manager is more convenient.

Having translations of the text is helpful to text reading, but it is not the best way to learn the language. So, I always encourage learners to read the text directly. And here there are tools to facilitate the reading. These include meaning lookup for each term (using CPED), custom dictionary lookup³, reconstructing *iti*, automatic sandhi cut (according to pre-defined rules), and recognition of declensions of pronouns, numerals, and some irregular nouns/adjectives (see the options in  button).

One option worth talking about here is the reconstruction of *iti*. When *iti* is marked out in the text, for example, *vilapi'nti*, the sandhi term will be tokenized into two words, i.e. *vilapi* and *nti*. When the option of Reconstruct *iti* is turned on, we will get *vilapim* and *iti*.

Another example, *passāmī'ti* yields *passāmī* and *ti*. And if the option of Shorten vowel before *iti* is also turned on, *passāmī* will become *passāmi*, and *ti* becomes *iti*. That is what we expect when we learn in the class. But things do not go that easy. In some instances, shortening the vowel does not make the right word, and the program does not know that. So, you have to make a decision whether you should use the option or not. Either way is not perfect. For the terms that seem to have *iti* but it is not marked out, e.g. *abhinanduntī*, you can cut the *it* (or *nti* in this case) out manually by editing the sentence (see below).

You can edit the custom dictionary and the sandhi list by the Edit menu.⁴ To enable detail mode, you have to press  button, or hit Ctrl-I (use Ctrl-U to return to simple mode). Figure 18.2 shows a sentence analyzed in detail.

As you see in the picture, 'Yo' is a pronoun in nominative case, so the reader recognizes as such. The reader also recognizes 'rājāno' as an irregular noun. Other terms marked with '[CPED]' has a definition in the concise dictionary. For those

³The custom dictionary has more priority than CPED. We can add new words or replace the old ones with it. However, there is no option to turn the custom dictionary off. You have to comment out the entries when you edit the file.

⁴The two files are hardcodedly located at `data/rules/dict.txt` and `data/rules/sandhi.txt` respectively.

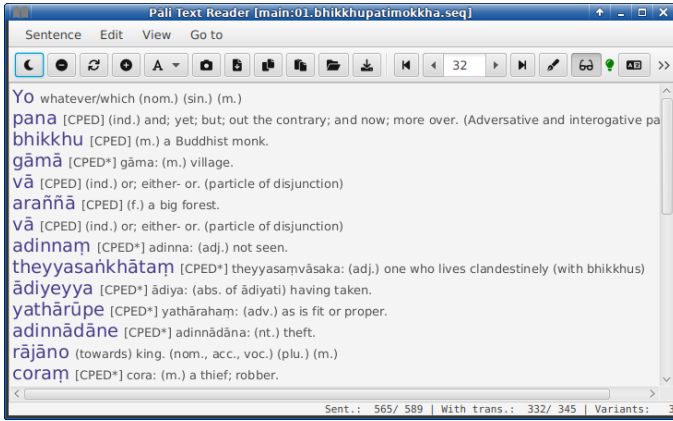



Figure 18.2.: Detail mode in Pāli Text Reader

with '[CPED*]', the exact definition is not found, so the nearest is shown instead. As you see in *theyyasāṅkhātāṃ*, the meaning is not quite relevant. We can make it better by editing the sentence (use  button in the tool bar or Ctrl-E), as shown in Figure 18.3.

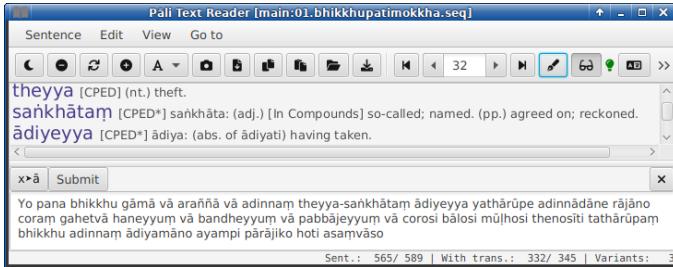


Figure 18.3.: A use of edit in Pāli Text Reader

In the example, I open the edit pane and insert a hyphen (-) in between the word, so we get *theyya-saṅkhātāṃ*. After hitting the Submit button, you will see the result as shown. So, you can get a better definition by editing the sentence, either splitting a word, adding a new one, deleting some, or inserting some symbols (adding two hyphens produces a dash).

An explanation about edit is needed. Edit is another field

which is saved together with the original text. When you edit a sentence, you just edit this mutable field, not the original. This means what is shown in the reader is the edit, whether it is changed or not. You can edit the sentence any way you want and save it, the original version is intact, so you can restore the original whenever you need. But remember that, each sentence has only one instance when saved. So, each sentence has only one edit. When you edit a sentence that has been edited before, the old edit is gone, the new one is saved. If you work on edit extensively, it is advisable to save the sentences/sequence in different directories. You can do this by menu Sentence>Save this sequence as. Once you make an edition, you have to save it first, otherwise the menu is disabled.

There are many things concerning Pāli grammar that cannot be implemented here, for example, verb form recognition. We mostly rely on CPED in this matter. It has a good coverage, I think. This can pave the way for future research on computational Pāli.⁵ As we have gone so far, it is already amazing.


I think that is enough to know to make a smooth start for using this tool. It is really helpful, despite its complexity. You have to learn by playing with it: make your own edits and translations. Remember that you can ruin the given data easily. But this is not to fear. The original data is always available, either in the software's package or in the website.


⁵I make no promise whatsoever here. I have no academic interest or motivation to do a comprehensive study on the subject. However, If I come up with a way to improve the language learning, I will add it to the program. Pāli machine translator is an impractical goal, but the best Pāli learning tool is a viable target.

19

Sentence Manager

The creation of this tool is a consequence of Pāli Text Reader. When operations upon sentences grow complex, a suitable tool is needed. Before you read this chapter, you must understand how the reader works (see Chapter 18), and know what I mean by *sentence*, *sequence*, *hash*, *edit*, and *variant*.

Sentence Manager can be opened by  button or by the Collection menu. It can also be opened from the reader. The manager is the most complex tool of all, in terms of its functions and components. I cannot tell and show you everything here. I will explain only the basic ideas that you should know when you explore the tool by yourselves. Figure 19.1 shows the manager on the first open.¹

There are three main tabs in the manager: Sentences, Translation Variants, and Merger. The first two tabs work for one directory at a time. This means there must be one working directory, the default is ‘main.’ You can change the working directory by  button. The directory contains many sentence files, several sequence files, and one variant info file.²

When the manager opens a directory, it reads all sentences and lists them in the table, and show the number of trans-

¹Because there are many sentences to load in the first run, starting Sentence Manager takes time initially. If you open the reader first, the slowness will be of the reader instead.

²In the implementation, I use JSON format for sentence and info files. Sequence files are just plain text with .seq extension.

19. Sentence Manager

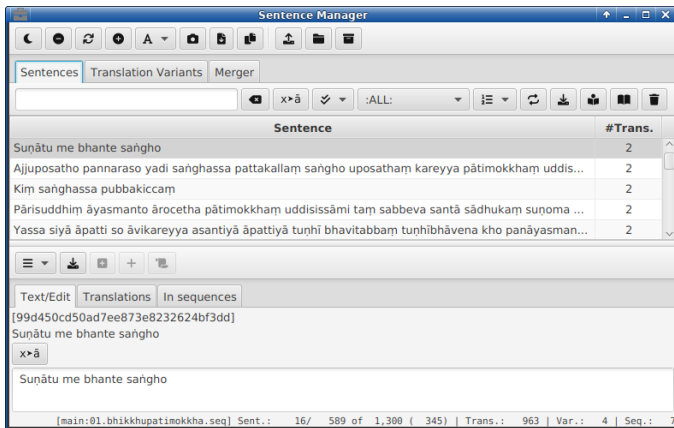










Figure 19.1.: Sentences tab in Sentence Manager

lations of each sentence, if any. When some sentences are changed outside the manager, edited by the reader, for example,  button has to be pressed to update the sentence list. You can save the whole directory into a zip file by hitting  button.

On the status bar, the directory's summary is shown. This tell us the directory's name, the sequence's name (if selected), the number of sentences, translations, variants, and sequences. The number in the parentheses is the number of sentences that have translations. When a sequence is selected, the number of the selected sentence is also shown.

You can search for sentences by three options: in (original) text, in edit, and in translations (see  button). You can choose to see all sentences or the sentences ordered by a sequence by using  button.

Once a sequence is selected, you can save it as a new sequence by  button. You can open the sequence in the reader by using . To open only one sentence, use  instead.

When you mess up the table order by clicking its header, to restore the order you can use the button provided. You can delete the selected sentence from the directory by using  button. (Be careful, if you play with the main directory bundled.)

In the lower part of the Sentences tab, there are another

three tabs. In these, you can edit the sentence’s text, add/edit translations, and see the sentence-sequence statistical relation. I will not go into the details of these functions.

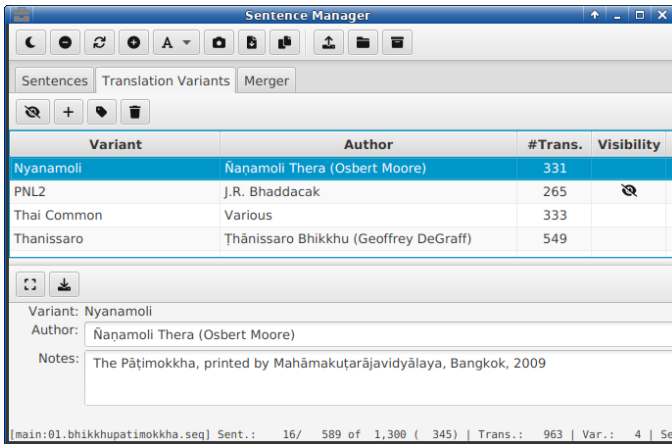


Figure 19.2.: Translation Variants tab in Sentence Manager

The Variants tab is far more simpler (see Figure 19.2). All variants and their information are shown here. You can add a new variant (+ button) and edit its details. You can also hide variants you do not want to see in the translations with ☒ button. This can make the display less clustered.

If you want to delete all translations under a variant name, you can delete that variant by ☒ button. This causes all sentences containing the variant to get updated. Be careful, you cannot undo this action and a lot of data can be deleted. Make sure you have a backup, and you are sober enough at the moment. If you just want to rename the variant, use ✎ button instead. I leave other minor things in this tab to the users to find out by themselves.

The last tab of the manager (Figure 19.3) is really useful practically. It can merge two directories into one with simple steps.

First, you have to create more than one sentence directories. You can test this by selected a sequence and save it in a new directory. Do it again with another sequence. Then you load these two directories into the Merger, left and right. Now you

19. Sentence Manager

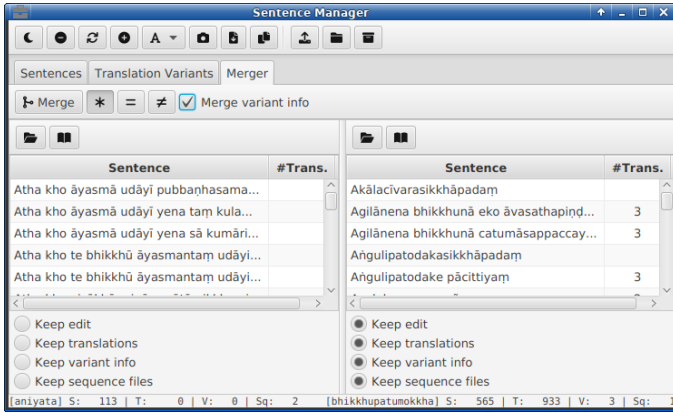


Figure 19.3.: Merger tab in Sentence Manager

have to think whether you should merge variant information or not. And think further, if a crash occurs (two directories contain the same sentence), which one you want to keep its data. The options to set lie in the lower part. Once you finish the settings, hit the Merge button. It will ask you the output directory. You should create a new one or select an empty one. When done, you will see the magic.

To understand all the things we talk about here, you have to roll the sleeves up and get your hand dirty. I have prepared enough data you can play with. Have fun.

Quick guide to regular expression

It is probable that most users of this program know nothing about regular expression. If you are not in a computer science department, or leaning about formal language theory, it is unlikely that you come across the term. But if you use computer a lot, particular when you use programs that have advanced search functions, the chance is good that you meet it.

What is it, then? Regular expression is a technical term, so it will be distracting if you ask for its meaning. To put it down to earth, regular expression is an enhancement of wildcard pattern that can make string matching more effectively.¹ Learning about regular expression is not easy. Many books about it have several hundred pages. So, the topic is really too big to discuss here.

However, I think we do not need to know all of its functions. That is the reason I add this chapter to introduce the users this powerful technique. Once you realize its capacity, you may need to learn it more. So, in this chapter I will show you some uses of regular expression in searching. I select only easy techniques that can be applied to our Pāli search (see Table 20.1). Many things not included here may or may not be used in the program.² You have to test them yourselves.

¹This is my definition to make it relevant here. Applications of regular expression are vast in computer science, and lesser in linguistics. To dig further, see https://en.wikipedia.org/wiki/Regular_expression.

²Not every technique can be used here because regular expression itself

20. Quick guide to regular expression

To understand regular expression (regex, from now on), you have to know how to use wildcards first. Basically, we use question mark (?) and asterisk (*) for this function. Technically, we call these *meta-characters*—characters that represent other characters. Here, ? means any one character, and * means any characters including none. (These two meanings are not used in regex.) For example, p??? can match *pana* or *puna* or *pitā* or whatever that starts with *p* and have totally four characters long. Whereas pi* can match *pitā* or *pitaro* or anything starts with *pi* including *pi* itself. We can see that even a simple use of wildcards can ease you search significantly. But regex has much more than this to offer. First, we have to forget about ? and * here because in regex they have different meaning.

Table 20.1.: Some uses of regular expression

Pattern	Meaning	Example	Result
.	any character	p...	<i>pana, puna, pitā</i> , and so on ³
\d ⁴	a digit	\d\d	00, 01, ...99
\D	a non-digit	\D	anything but a digit
\s	a whitespace	kho\s pana	<i>kho pana</i>
\b	a word boundary	\biti\b	<i>iti</i> as a whole word ⁵
\B	a non-word boundary	\Bti\b	anything ends with <i>ti</i>

Continued on the next page...

has several implementations. Even in our program, four parser engines are used: Java (in Tokenizer, the Editor), JavaScript (in the HTML Viewer), H2 database (in Simple Lister), and Lucene (in Lucene Finder). So, a pattern used in one place may not work in others, or may need an adjustment.

³A dot in regex is equivalent to ? in the wildcards.

⁴In the Text Editor, you can use this verbatim. But in the HTML Viewer, you have to double the backslash, so use \\d instead. This is true to all patterns that use backslashes.

⁵The word boundary is not Pāli sensitive. So, a non-English character is also counted as word boundary. For example, you may also find *āiti* in this case, if there is such a word.

Table 20.1: Some uses of regular expression (contd...)

Pattern	Meaning	Example	Result
\S	a non-whitespace	eta\Savoca ⁶	<i>etadavoca</i>
[...]	any in the class	[Tt]ena dinn[oā]	<i>Tena</i> or <i>tena</i> <i>dinno</i> or <i>dinnā</i>
[^...]	not in the class	dinn[^oā]	<i>dinne</i>
?	once or not at all	i?ti	<i>it</i> or <i>iti</i>
*	zero or more times	i*ti	<i>ti</i> or <i>iti</i> or even <i>iiti</i> , etc.
+	one or more times	manas+a	<i>manasa</i> or <i>manassa</i>
{n}	exactly n times	manas{2}a	<i>manassa</i>
{n, }	at least n times	manas{1, }a	<i>manasa</i> or <i>manassa</i> or even <i>manasssa</i>
{n, m}	at least n but not more than m times	manas{1, 2}a	<i>manasa</i> or <i>manassa</i>
(...)	grouping	man(as)?o	<i>mano</i> or <i>manaso</i>
(... ...)	any in grouping	manas(o sa)	<i>manaso</i> or <i>manassa</i>
^	at the beginning	^attha.*	any word start- ing with <i>attha</i> ⁷
\$	at the end	.*attha\$	any word end- ing with <i>attha</i>

What you have seen in the table is just a small part of regex that I think it can be applied to out search. Many other things seem difficult to use with Pāli, at least in an easy way. For those who know regex well, you may try group capturing and back references, for example, using '(puna)p\1m' to find 'punapunam.' I find little use of this, but it can be useful in some situations.

⁶You may use eta.avoca to yield the same result, but the meaning is different. Using \S stresses that there should be no space in between.

⁷The symbol ^ and \$ should be used only in term searching, like in Simple Lister, because when searching in full text, the meaning of 'start' and 'end' is context-dependent.

About the author

J. R. Bhaddacak holds a PhD in Religious Studies and has professional background of computer science and engineering. Nowadays he is an independent researcher, working alone outside any academic milieu. His main field of study is on religion, particularly Theravāda Buddhism as a cultural product. Recently he has started investigating into Pāli language with three goals in mind: first, to make Pāli more accessible by making it easier to learn; second, to make Pāli studies more critical by also taking modern literary theory and its kin into account; and third, to research into computational Pāli and produce effective Pāli learning tools. He is also the maker of Pāli Platform, a comprehensive program for Pāli studies. By the days of writing this manual, he lives as a mendicant somewhere in a rural area of Thailand.

Colophon

This document is produced by \LaTeX typesetting system using \TeX Live 2022/Debian on GNU/Linux. Devuan Daedalus/Ceres (testing branch) is used to date. Main fonts used are \TeX Gyre Schola (Serif), \TeX Gyre Heros (Sans) and DejaVu Sans Mono. The fontawesome5 package facilitates the use of graphic icons. To make the final PDF unicode-searchable, Lua \LaTeX is used as the engine. Neovim is the main editor. The working machine is 32-bit Dell Inspiron N4030 (2011).