Chair of
Software Engineering

TrucStudio - Automatic modeling of courses

Master Thesis

By: Adrian Roman Müller
Supervised by: Michela Pedroni
Prof. Bertrand Meyer

Student Number: 02-920-312

18.02. – 17.08.2008

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

Table of contents

Abstract 2
1 Introduction 3
1.1 Scope of the work 3
1.2 Implementation 3
1.3 Difficulties 3
2 Implementation 5
3 Results 7
3.1 Test conditions 7
3.2 Notioncover Manager 8
3.2.1 Time consumption 8
3.2.2 simple versus view horizon 9
3.2.3 Best size of viewhorizon 14
3.3 Keyword comparison 15
3.3.1 Time consumption 15
3.3.2 Accuracy of found Notioncovers 16
3.4 Conclusion 17
3.4.1 Recommended settings 17
4 Future work 19
5 Sources list 21
5.1 Educational material used for testing 21
5.2 Reading list 21
6 Appendix 23
6.1 Project plan 25
6.2 User Guide: Material Import 31
6.3 Developer Guide: Material Import 41

Abstract

Planning and executing quality courses and curricula are difficult tasks and require both a thorough knowledge of the domain and a systematic approach. Trucs (Testable Reusable Units of Cognition) and their finer grained equivalent Notions allow to capture the content of educational material (which consists of chunks of knowledge) and provide the basis for a well founded approach to course planning. TrucStudio is the software system that builds upon this methodology. At the current stage it provides instructors with domain modeling facilities and supports basic course management features.

The scope of the present master thesis was to provide functionality which allows automatically finding Notions taught in a lecture. This is achieved by a plain text scan of provided lecture material (slides, scripts, books). The user can decide between different algorithms which deal with aspects like the generation of suitable keywords describing a Notion or the detection of mistyped words.

1 Introduction

In the last couple of years, universities made an effort to harmonize their studies. So a model was introduced, which allows the students to gather credit points from a huge amount of different courses. These courses are often very specialized but can also intersect in some single lectures. This makes it important that the universities use course management software like TrucStudio.

TrucStudio has an own data model which contains amongst other things Notions, the smallest educational unit. If an educator knows for two courses which Notions they contain, he can compare them on an objective basis. This makes it easier to deal with exchange students and allows professors to help students finding the courses fitting best into their program. TrucStudio also helps find reusable parts from already existing courses and integrate them into a new course. So it supports an instructor in finding slides which already deal with the course's Notions.

Unfortunately, all the courses and their lectures must be collected into TrucStudio, to get the benefits.

1.1 Scope of the work

This project builds on top of the existing TrucStudio application, and extend the course management interface with an option to automatically parse teaching material (such as course slides) for the Notions that are provided in the domain model. The result of this thesis will greatly reduce the amount of work required from instructors that want to start using TrucStudio for existing courses.

1.2 Implementation

The implementation is done in theEiffel language and fits into the TrucStudio source code. No additional libraries are used for the program part developed in this master thesis.

1.3 Difficulties

TrucStudio is a platform independent program. Therefore all program parts must be the same. This leads to problems when including external libraries. Today there are some libraries included in TrucStudio, which need some special treatment when compiling for systems like Linux or Mac OS X. Learning from this fact, the goal was not to include libraries which are not well supported Eiffel libraries.

This made it impossible to find a library which reads a PDF file and provides the file's

content. Since writing a new PDF library was out of the scope of this master thesis, we rely on the user to provide the sources as plaintext files. The lecture import therefore contains a reader which accepts plaintext files. Implementation of a PDF reader is left to future work.

Because a plain text file contains only very little structural information, it was also not useful to implement a manager which deals with structural information in a source.

2 Implementation

The lecture extraction implemented in this thesis is written in the Eiffel language and integrates tightly into the TrucStudio source code. There are no additional libraries required to run it.

The implementation contains the following parts:

- A Material Import Manager controls the import.
- A view, the settings-dialog, gives the user the possibility to change the settings concerning the import.
- A model stores the data used for the settings-dialog.
- A File Reader extracts the words out of a source file.
- Text comparison methods deal with typing errors and plural forms of keywords.
- A Keyword Generator extracts keywords describing a Notion from its Summary and/or Notion name.
- For each Notion in the domain model there exists a Notioncover Manager. Here the Notions coverages are detected.
- Some additional parts, such as a tools class ready to support also other parts of TrucStudio.

For detailed implementation information, please refer to the developer guide, which is attached to this document.

3 Results

3.1 Test conditions

To make sure the tests in this chapter are comparable between each other, they have predefined settings, which are only changed where the test conditions required to. The test descriptions will mention all differences to the conditions listed below.

The test machine was a Lenovo T40p Laptop with 1.6 GHz Intel M processor and 1 GB physical memory. The tests were done under Windows XP SP2, using Microsoft C Compiler.

All used material was real educational material provided by the Chair of Software Engineering, ETH Zurich. It includes the slides of the "object" lecture held in Introduction to Programming course, the according chapter of the textbook "Touch of class", and the complete textbook "Touch of class".

The normally used parameters in TrucStudio were:

- Notions: 28 (oo.txm)
  - argument
  - argument declaration
  - argument passing
  - attribute
  - chains of qualified feature calls
  - class
  - class declaration
  - class invariant
  - contract
  - creation declaration
  - debugger
  - feature
  - feature body
  - feature call
  - feature declaration
  - feature signature
  - function
  - instance
  - object
  - postcondition
  - precondition
  - procedure
  - return type
  - return value
  - simple qualified feature call

  - step wise debugging
  - type
  - unqualified feature call
- Notioncover Manager: with viewhorizon
- Repeated Notions: no
- size of viewhorizon: 100
- comparison method: exact compare
- keyword generation: Notion name only

Section 3.2 and 3.3 show the test results for (1) the Notioncover Manager and (2) the keyword comparison. They both include time measurements where the various algorithms were applied to test data. In a second test, the accuracy of the results was measured. For time measurement tests, the profiler included into EiffelStudio was used. So the units of the times shown in the table are those of the EiffelStudio profiler.

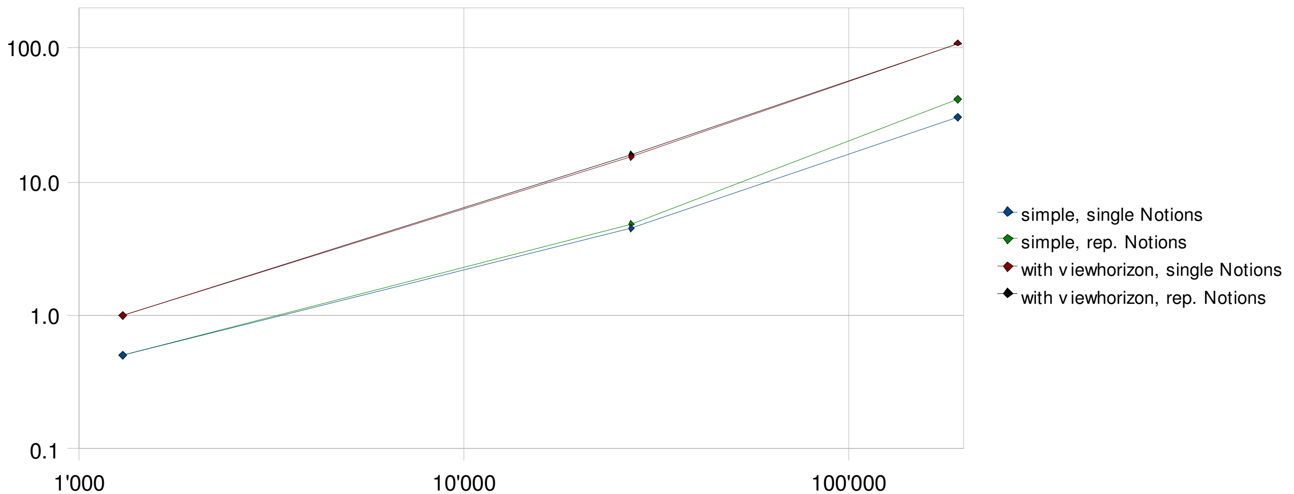
3.2 Notioncover Manager

3.2.1 Time consumption

The first test was to measure the time consumed by the algorithm when extracting the Notions covered in a text. This test allowed measuring the speed differences between the various Notioncover Managers. The measuring only considered the time for the import process, without the time needed to display the GUI, to wait for the user and to read the words from the import file.

	simple single Notions	simple repeated Notions	viewhorizon single Notions	viewhorizon repeated Notions
Slides: “Introduction to programming” course, lecture “objects” ⟶ 1'307 words	0.5 (1.0 x)	0.5 (1.0 x)	1.0 (2.0 x)	1.0 (2.0 x)
Book: “Touch of class” part 3 (chapter 16 + 17) ⟶ 27'302 words	4.5 (1.0 x)	4.9 (1.1 x)	15.4 (3.4 x)	16.0 (3.6 x)
Book: “Touch of class” complete book ⟶ 191'843 words	30.3 (1.0 x)	41.7 (1.4 x)	109.6 (3.6 x)	109.5 (3.6 x)

The test shows that the Notioncover Manager with viewhorizon takes about 3.5 times as long as the simple version and repeated Notions do not affect the result very much. Both algorithms do their job in reasonable time. As shown in the figure below, the consumed time rises linear to the number of words read.



X-axis: number of words (logarithmic); y-axis: time (logarithmic)

3.2.2 simple versus view horizon

In a second test, the results of the different Notioncover Managers were checked for their correctness. To do this, the Notions have been extracted by hand and used as reference. The following signs describe the accuracy of those algorithms.

OK: Notion has been found correctly

+: Notion has been detected where no Notion should be detected (false positive)

–: A Notion coverage has not been found (false negative)

ord: Notion has been found but isn't at the correct place in the list (wrong order of list)

Algorithms with single Notions allowed only:

	reference	simple	with viewhorizon
Slides: “Introduction to programming” course, lecture 02 “objects” ⟶ 7 Notioncovers	class feature feature declaration feature body feature call object argument	class feature class declaration feature declaration feature body feature call object creation declaration procedure return value function argument argument declaration 7 OK / 6 +	class feature feature declaration class declaration feature body feature call object --------------------------- 6 OK / 1 + / 1 –
Book: “Touch of class” chapter 2 ⟶ 9 Notioncovers	class class declaration feature feature declaration object feature call return value argument argument passing	class feature *(see beneath)* class declaration *(see above)* feature declaration type object feature call creation declaration *(see beneath)* argument argument declaration function return value *(see above)* return type --------------------------- 6 OK / 5 + / 1 – / 2 ord	class feature *(see beneath)* class declaration *(see above)* feature declaration type object feature call --------------------------- argument --------------------------- 6 OK / 1 + / 2 – / 1 ord

Algorithms with repeated Notions allowed:

	reference	simple	with viewhorizon
Slides:	class	class	class
“Introduction to	feature	feature	feature
programming” course,		class declaration	class declaration
lecture 02 “objects”	feature declaration	feature declaration	feature declaration
⟶ 10 Notioncovers		class	
		feature	
	feature body	feature body	feature body
		class	class
		feature	feature
	feature call	feature call	feature call
		object	
		feature	
		class	class
		feature call	
		feature	feature
		class	
		feature call	feature call
	object	object	object
	feature	feature	feature
	feature call	feature call	feature call
		object	
		feature	
		class	
		creation declaration	
		procedure	
			feature body
	object	object	object
		feature	
		return value	
		object	
		feature	feature
		feature call	
		feature body	
		class	
		feature	
		function	
		argument declaration	
	argument	argument	--------------------------
		object	
		feature	
		feature call	
		object	
		10 OK / 31 +	9 OK / 8 + / 1 –

Book:	class	class	class
“Touch of class”		feature	feature
chapter 2	class declaration	class declaration	class declaration
→ 14 Notioncovers		feature declaration	
		feature	
		class	class
		feature declaration	
		class declaration	
	feature	feature	feature
		class	
		class declaration	
	feature declaration	feature declaration	feature declaration
		feature	
		class	
		type	type
		feature	feature
	object	object	object
		feature call	feature call
		feature	
		object	
		class	class
		feature	
		feature call	
		object	object
	feature	feature	feature
	feature call	feature call	feature call
		object	
		feature	
		feature call	
	object	object	object
		feature	feature
		creation declaration	
		class	class
		feature	
		feature call	feature call
		object	
	feature	feature	feature
		class	
		feature call	
		feature	
		argument	argument
		argument declaration	
		feature	
		function	
		argument	
		object	
		return type	

	return value	return value	--------------------------
	object	object	object
	feature	feature	feature
		argument	
		class	
		feature declaration	feature declaration
		class declaration	
		argument declaration	
		feature	
		feature call	feature call
		object	
		class	class
			class declaration
		feature	feature
	argument	argument	argument
	argument passing	--------------------------	--------------------------
		13 OK / 48 + / 1 –	12 OK / 16 + / 2 –

This second test shows that the results are more accurate for slides than text extracts. An explanation is that keywords are used in slides very often, whereas in a real text there is a much larger vocabulary used. The word “type” is used here also for the verb “to type”, which has a completely different meaning.

The keywords “class” and “function” are found very often. (Before improving the algorithms it was more extreme, but this behavior can still be observed.) This is due to the extensive use of source code examples in the inputs used for this test. In fact, the keywords “class” and “feature” in most cases exist at least once or twice per example. In order not to miss other keywords, the algorithm allows such false positives.

All four tests for the Notioncover Manager with viewhorizon show that the last covered Notion is never found. The analysis of this fact shows that this is just a coincidence. The keyword “argument” of the last Notion in the slides of the “objects” lecture is only once exactly equally typed in the source file. Because the Notioncover Manager with viewhorizon needs at least two occurrences (to prevent some false positives), the Notioncover is not found. The same goes for the chapter 2 in “Touch of class”. Here the keyword “passing” does not exist exactly equally typed in the source file – so the last Notioncover, “argument passing” can be found neither by the Notioncover Manager with viewhorizon nor by the simple one.

3.2.3 Best size of viewhorizon

To find determine the optimal viewhorizon size, the work included tests with various sizes and source types. The size of the viewhorizon takes only effect, when using the "repeated Notions" mode.

size of viewhorizon	**50**	**100**	**150**	**200**	**250**	**300**	**350**	**500**
Slides: "Introduction to programming" course, lecture 02 "objects" ⟶ 7 Notioncovers	7 OK 6 + 2 − 1 ord	9 OK 8 + 1 −	**9 OK** **3 +** **1 −**	8 OK 3 + 2 −	8 OK 4 + 2 −	8 OK 2 + 2 −	8 OK 2 + 2 −	6 OK 2 + 4 −
Book: "Touch of class" chapter 2	11 OK 14 + 2 − 1 ord	12 OK 16 + 2 −	11 OK 14 + 2 − 1 ord	**11 OK** **12 +** **2 −** **1 ord**	10 OK 12 + 2 − 2 ord	11 OK 14 + 2 − 1 ord	9 OK 10 + 2 − 3 ord	9 OK 7 + 2 − 3 ord

OK: Notion has been found correctly
+: Notion has been detected where no Notion should be detected (false positive)
−: A Notion coverage has not been found (false negative)
ord: Notion has been found but isn't at the correct place in the list (wrong order of list)

It shows that books need a larger size of viewhorizon than slides. For the tested slides we reached the best results when using a size of 150 words, whereas the extraction for the book chapter showed the best results for the size of 200 words.

3.3 Keyword comparison

3.3.1 Time consumption

This test measured how long the different algorithms took to compare the words found in the text with the keywords describing the Notions. The time listed in the table is the one used in comparison features. No other program parts are involved.

	exact	Levenshtein distance = 1	Levenshtein mem. opt. distance = 1	Damerau Levenshtein distance = 1	Ratcliff Obershelp distance = 2
Slides: “Introduction to programming” course, lecture “objects” ⟶ 64’451 comparisons	0.04 (1.0 x)	2.46 (61.5 x)	3.98 (99.5 x)	3.62 (90.5 x)	0.83 (20.8 x)
Book: “Touch of class” chapter 2 ⟶ 212’083 comparisons	0.12 (1.0 x)	7.12 (59.3 x)	12.19 (101.6 x)	10.49 (87.4 x)	2.14 (17.8 x)
Book: “Touch of class” complete book ⟶ 9’532’513 comparisons	5.34 (1.0 x)	332.93 (62.3 x)	579.44 (108.5 x)	490.26 (91.8 x)	105.43 (19.7 x)

When analyzing this test, it is easy to order the algorithms by their efficiency. The most efficient algorithm is obviously the exact comparison which does not deal with typing errors. Surprisingly, the Ratcliff Obershelp algorithm, which uses recursion, reaches second place. This shows that this algorithm can be implemented efficiently.

When looking at Levenshtein algorithms, it shows that the memory optimized version does use so much time, that it is normally a better choice to take the normal Levenshtein algorithm. Comparing Damerau Levenshtein algorithm with the normal Levenshtein algorithm shows, that Damerau Levenshtein algorithm is nearly 50% slower. This comes from the additional typing error correction which is more complex than the others.

3.3.2 Accuracy of found Notioncovers

To find the most accurate keyword comparison algorithm and its distance, there have been several settings for keyword comparison tested and compared. The memory optimized Levenshtein algorithm has not been tested because it returns, for a given setting, always the same result as the normal Levenshtein algorithm.

	exact	Levenshtein distance = 1	Levenshtein distance = 2	Damerau Levenshtein distance =1	Damerau Levenshtein distance = 1	RatcliffObershelp distance = 1	RatcliffObershelp distance = 2	RatcliffObershelp distance = 3
Slides: “Introduction to programming” course, lecture “objects” ⟶ 7 Notioncovers	6 OK 1 + 1 –	**6 OK** **1 +** **1 ord**	4 OK 3 + 3 ord	6 OK 1 + 1 – 1 ord	4 OK 3 + 3 ord	6 OK 1 + 1 –	5 OK 2 + 2 ord	5 OK 6 + 2 ord
Book: “Touch of class” chapter 2 ⟶ 9 Notioncovers	6 OK 1 + 2 – 1 ord	5 OK 2 + 2 – 2 ord	6 OK 4 + 2 – 1 ord	5 OK 2 + 2 – 2 ord	6 OK 4 + 2 – 1 ord	6 OK 2 + 2 – 1 ord	5 OK 3 + 2 – 2 ord	5 OK 6 + 1 – 3 ord

OK: Notion has been found correctly
+: Notion has been detected where no Notion should be detected (false positive)
–: A Notion coverage has not been found (false negative)
ord: Notion has been found but isn't at the correct place in the list (wrong order of list)

The test shows, that the simplest algorithm is not always the worst. The exact comparison algorithm gets nice results also because it does not get additional Notion coverages from the typing error correction. This goes with the fact, that the algorithms which use a distance value for typing correction have best results with a distance of 1.

3.4 Conclusion

The results show that extracting Notions from plaintext is tricky. There are different things to keep in mind:

- We don't know the meaning of a word if we just read it. For example the word "type", which has often be found in chapter 2 of the textbook "Toch of class", can have different meanings. It can refer to the verb "to type" or it can mean the noun "type" (which too has different meanings in different contexts).
- If, in the source text, there are only synonyms of the keywords describing a Notion, then the Notioncover cannot be found.
- Allowing typing errors comes with false positive Notion covers. Best results can still be achieved using the exact text comparison.
- The implemented algorithms have different complexity and therefore have completely different running times. This does not matter so much when taking a look at the consumed time. As long as the scan is not applied to a complete course book (over 1 million word comparisons), the needed time is reasonable.
- Using the option "repeated Notions" makes rarely sense. Normally a lecture covers a Notion only once. Disabling the option returns much better results. Moreover the option "Repeated Notions" lets the list of covered Notions grow extremely. This can lead to GUI problems which make the program very slow.

3.4.1 Recommended settings

To improve user's first results, the knowledge acquired through these tests has been integrated into the preset settings. So the following recommended settings will be preset automatically:

- Notioncover Manager: with viewhorizon
- Repeated Notions: no
- Size of horizon: 150 (for slides – for textbooks use a size of 200)
- Text comparison method: exact comparison
- Distance (for other comparison methods): 1
- Keyword source: Notion name only

4 Future work

This master thesis included investigating a possibility to **directly read PDF-Files**. PDF files are well known and are used very often for educational material. There exists an Eiffel library that deals with PDF files, the EPDF library. Unfortunately, EPDF only supports writing of PDF files – and for our purpose we needed a reading support. Of course there are other libraries, like the C library zlib, which allowed reading a PDF file, but it was a goal of the master thesis, not to introduce additional dependencies to such non-Eiffel libraries. If in the future EPDF supports reading PDF files, a reader which takes PDF files instead of TXT files should be implemented to improve the tool.

Having such a reader could also allow a **new type of Notioncover Manager**. This new type could read the text not only word by word but also take the document structure into account.

A third field of future development could be the introduction of **new keyword comparison algorithms**. These could find also words which are synonyms of the Notion's keywords. Additionally there could be some combination of already existing keyword comparison algorithms.

To improve the usability, the dialogs could be extended with buttons which allowed **adding or removing selected source files**. This could also solve the problem that multiple source files can only be selected if they are located in the same directory.

5 Sources list

5.1 Educational material used for testing

Slides of the course “Introduction to Programming”; Prof. Dr. B. Meyer; ETH Zurich; fall 2007
http://se.inf.ethz.ch/teaching/2007-F/eprog-0001/english_index.html

Textbook “Touch of class”, Draft 17.14, Prof. Dr. B. Meyer, ETH Zurich, 2007-09-12

5.2 Reading list

Testable, reusable units of cognition. IEEE Computer, 39(4):20-24; Prof. Dr. B. Meyer; ETH Zurich; 2006
http://ieeexplore.ieee.org/iel5/2/33950/01620989.pdf

TrucStudio – A Prototype; Masterthesis; L. Widmer; ETH Zurich; April 2007
http://se.inf.ethz.ch/projects/leo_widmer/report.pdf

Efficent Streaming Text Clustering; S. Zhong; Florida Atlantic University; August 2005
http://www.cse.fau.edu/~zhong/papers/nn05_Zhong.pdf

6 Appendix

Here you find the Wiki-pages added to TrucStudio Wiki. These can be found in the Documentation part of TrucStudio page (http://trucstudio.origo.ethz.ch/wiki/doc).

- Project plan
- User Guide: Material Import
- Developer Guide: Material Import

Home

Project plan Adrian Mueller

MATIMPORT | project plan

Table of Contents

- Information
- Project description
  - Overview
  - Scope of the work
  - Intended results
- Background material
  - Reading list
- Project management
  - Objectives and priorities
  - Criteria for success
  - Method of work
  - Quality management
    - Documentation
    - Validation steps
- Plan with milestones
  - Project steps
  - Deadline
  - Tentative schedule
- References

edit

Information

Master project
Project period: **02/2008 - 08/2008**
Student name: **Adrian Roman Müller**
Status: **diploma semester**
Email address: **muellead[at]student.ethz.ch**
Supervisor name: **Michela Pedroni**

edit

Project description

edit

Overview

Planning and executing quality courses and curricula are difficult tasks and require both a thorough knowledge of the domain and a systematic approach. Trucs (Testable Reusable Units of Cognition) and their finer grained equivalent Notions allow to capture the content of educational material (which consists of chunks of knowledge) and provide the basis for a well founded approach to course planning. TrucStudio is the software system that builds upon this methodology. At the current stage it provides instructors with domain modeling facilities and supports basic course management features.

edit

Scope of the work

TrucStudio - Automatic modeling of courses
This project builds on top of the existing TrucStudio application, extending the course management interface with an option to automatically parse teaching material (such as course slides) for notions that are provided in the domain model. The result of this thesis will greatly reduce the amount of work required from instructors that want to start using TrucStudio for

existing courses.

edit

Intended results

The program part should be able to extract the notions covered in some educational material. The program should not force the user to give additional information about the notions. It has to deal with the existing data (like the notions name or summary).

edit

Background material

edit

Reading list

[3] **Bertrand Meyer: Testable, reusable units of cognition. IEEE Computer, 39(4):20-24, 2006**
[4] **Leo Widmer: TrucStudio – A Prototype; Master Thesis, April 2007**
[6] **Shi Zhong: Efficient Streaming Text Clustering; Florida Atlantic University; August 2005**

edit

Project management

edit

Objectives and priorities

- First of all, it has to be checked whether and how it is possible to read PDF-Files in Eiffel
- Very high priority has the implementation of a simple algorithm, which can import some educational material into a new Lecture. This material should be stored in a *.txt file and the Lecture shall contain the Notions which are covered in the file. This first implementation should find the coverages of a Notion by searching for keywords.
- Further on there have to be implemented some functionalities to check for keywords with regards to typing errors.
- Also some mechanism to find suitable keywords for a Notion have to be found. A simple one could be to take the words which are used as the Notions name. Keywords with no sense like 'a' should be filtered out.
- After all this, some more intelligent algorithms for coverage detection should be found and implemented
- If possible, also some files like *.pdf, *.ps, *.doc or *.ppt should be importable.

edit

Criteria for success

- The new program part can extract a possible coverage of Notions out of some educational material which is provided as plain text file.
- The implementation follows oo-programming standards and is well documented.
- The documentation is done into the wiki for this project. It should cover the changes and a how-to for the new features.

edit

Method of work

This program part is done iteratively, which means that it is implemented stepwise, beginning with few functionality and extending it step by step. All these iteration steps contain a design, implementation and testing phase.

edit

Quality management

edit

Documentation

The documentation will be done in this wiki, which does not mean that the source code does not contain enough comments. But it is essential that people which will maintain the program do not have to study the whole source code. The wiki is the main source for users and developers. So also the other project members document their code in this wiki.

edit

Validation steps

After each iteration the testing will be forced, but it is done also during the implementation process. To test this project part with some real material, the material of the course 'Introduction to Programming' [5] (which is held by Prof. B. Meyer) has been chosen.
We have planned weekly meetings to discuss projects progress and future work.

edit

Plan with milestones

edit

Project steps

- Search for papers and libraries
- Plan the program part
- Implement first program parts (reduced functionality)
- Test and improve program parts
- Bugfixing
- Measure correctness, time consumption, etc. and find good standard parameter values
- Write report and online documentation (wiki)
- Prepare presentation

edit

Deadline

Sunday, 17. August 2008

edit

Tentative schedule

Documentation is an ongoing process and will not be mentioned in the project plan

18.02.2008

- Start master thesis

18.02. - 22.02.

- Set up laptop which is provided by ETH
- Set up workplace

25.02. - 28.02.

- Get familiar with TrucStudio and its sources
- Discover the Internet for possibilities to read PDF files

03.03. - 04.04.

- First version of material import (text reader, word parser, ...)
- Simple GUI interface
- Testing

07.04. - 18.04

- Bugfixing
- Extending model for the new possibility to add a Notion more than once in a Lecture

21.04. - 02.05.

- Extending and improving GUI
- New separate model for Matimport dialog
- Uncouple Matimport Dialog from Class-Lecture Tree (new: start it in Tools menu)
- Fix Issue #39 (remove icons from TS buttons)

03.05. - 23.05.

- Develop a cleverer Notioncover Manager

26.05. - 14.06.

- Improve GUI (including the usage of an hourglass while importing)
- Fix bugs in Matimport
- First profiling of the different algorithms
- Implement a version which allows to import multiple files at once
- Tests for release 0.2

16.06. - 20.06.

- General tests for release 0.2 (on Windows XP and SuSE Linux 10.3)

23.06. - 16.07.

- Add possibility to select the size of viewhorizon for some notioncover managers
- Documentation

17.07. - 30.07.

- finding results
- analysing the tools qualities

31.07. - 05.08.

- slides for presentation
- continuing to measure program

06.08.2008

- presentation

07.08. - 13.08.

- finish report

14.08. - 17.08.

- reserve and printing

17.08.2008

- End master thesis

References

edit

[1] **Chair of Software Engineering: Semester-/Diplomarbeiten**
[2] Bertrand Meyer: Object-Oriented Software Construction, 2nd edition, Prentice Hall, 1997.
[3] Bertrand Meyer: Testable, reusable units of cognition. IEEE Computer, 39(4):20-24, 2006.
[4] **Leo Widmer: TrucStudio – A Prototype, Master Thesis, April 2007.**

[5] **Introduction to Programming course (educational material used for testing)**
[6] **Shi Zhong: Efficient Streaming Text Clustering, Florida Atlantic University, August 2005**

Home

User Guide: Material Import

import | MATIMPORT | user guide

User guide home

Table of Contents

- Introduction
- File open dialog
- Lecture Creation dialogs
  - Single lecture creation dialog
    - Name
    - Course
    - Cover Manager
    - Repeated Notions
    - Size of horizon
    - Comparison
    - Distance
    - Keywords
    - Create
  - Multiple lectures creation dialog
    - Course
    - Cover Manager
    - Repeated Notions
    - Size of horizon
    - Comparison
    - Distance
    - Keywords
    - Create
- Notioncover Managers
  - Notioncover Manager with view horizon
  - Simple Notioncover Manager
- Text comparison Methods
  - exact comparison
  - Levenshtein Algorithm
  - Levenshtein Algorithm (memory optimized)
  - Damerau Levenshtein Algorithm
  - Ratcliff / Obershelp Algorithm
- Keyword selection methods
- Converting some common file types into plain text files (TXT)
  - Converting PDF into TXT
    - Using Adobe Reader (freeware)
    - Using GSview (freeware)
  - Converting presentation files (PPT, PPS, ODP, ...) into TXT
    - Using OpenOffice Impress (freeware)
    - Using Microsoft PowerPoint (commercial)
  - Converting text documents (RTF, DOC, ODT, ...) into TXT
    - Using OpenOffice Writer (freeware)
    - Using Microsoft Word (commercial)

edit

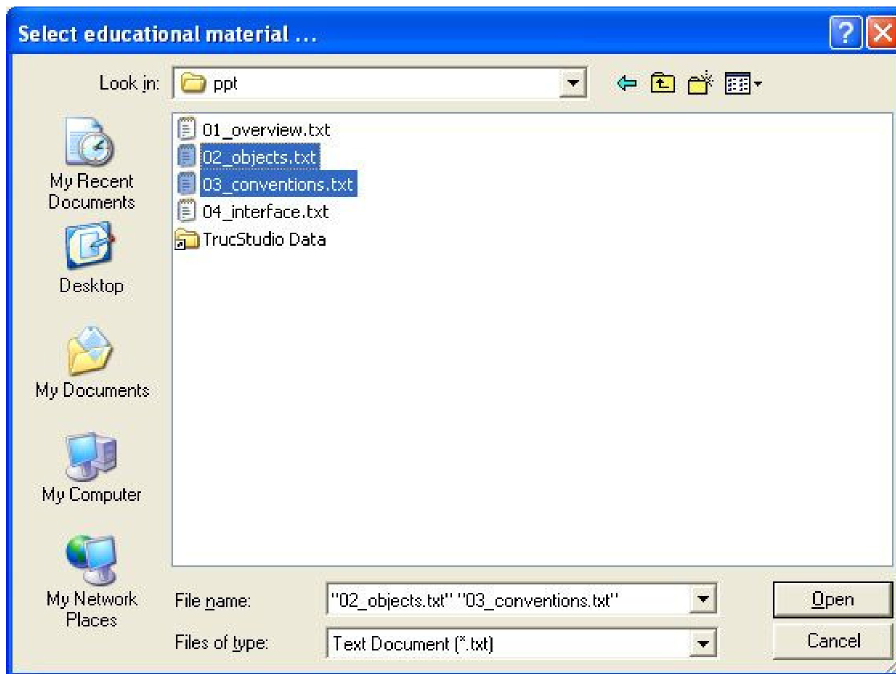
Introduction

If you've got one or several plaintext files (ending with .txt), containing the contents of your lecture(s), you can easily extract the Notions out of each file. To do this, chose ***Tools/Extract from TXT...*** and follow the dialogues.

If the educational material is not available as txt file, then you have to do the conversion first.
(see: 'Converting some common file types into plain text files').

edit

File open dialog

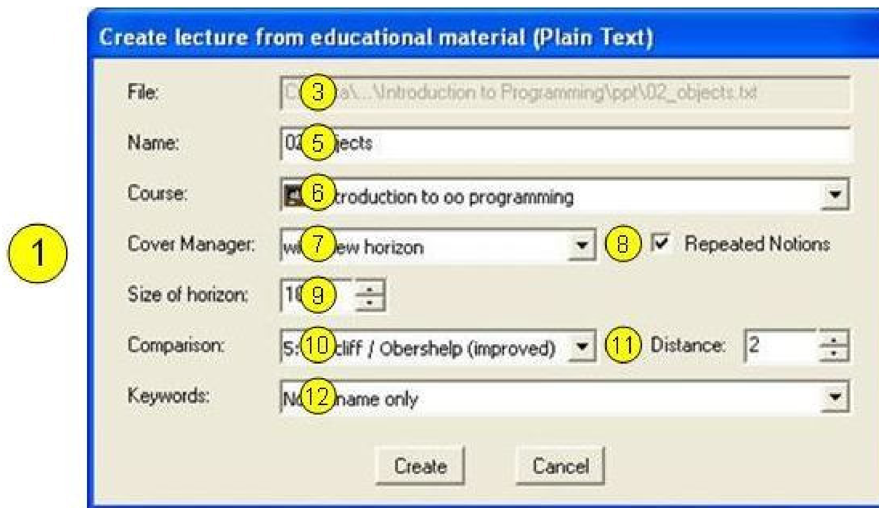


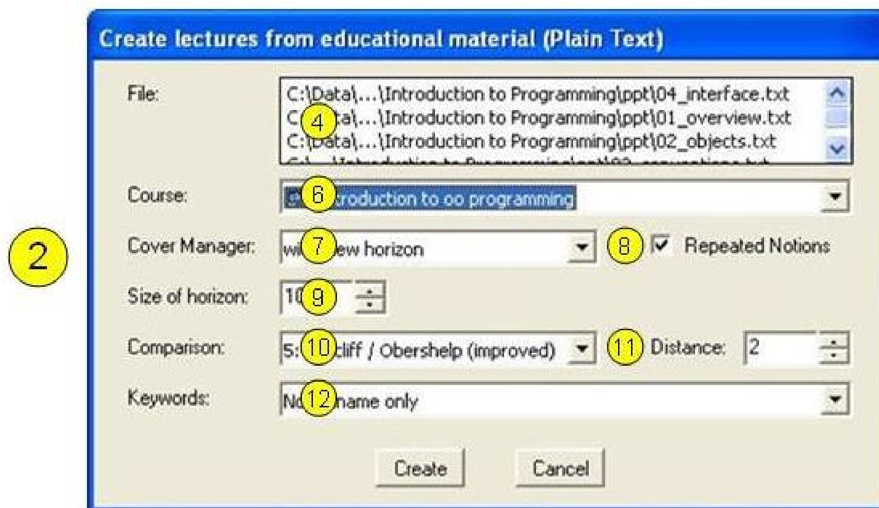
When selecting 'Extract from TXT...' a file open dialog as shown in the figure above opens. Select one or more txt-files to import and click on 'Open'.

Lecture Creation dialogs

edit

After you have opened one or more source files, you will see one of the following dialogs. They differ in the fact, that you can choose the created lectures name only if you have chosen exactly one source file to import. Otherwise the created lectures will be named after the source file names.



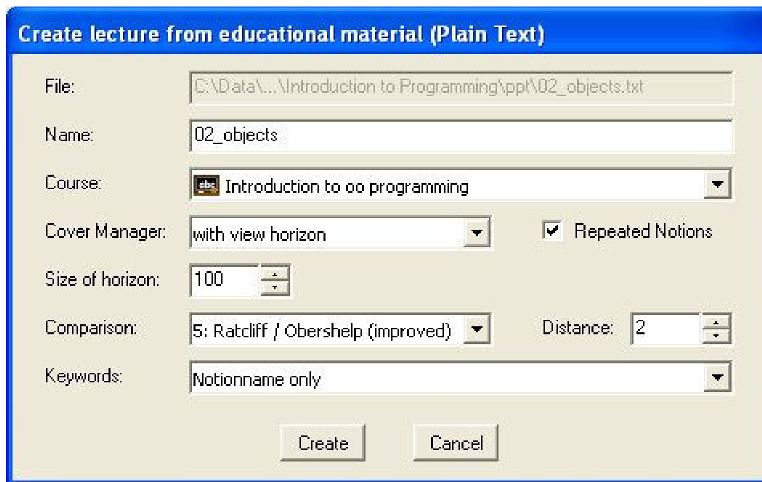


The numbers in the picture stand for the following parts:

1. Single lecture creation dialog: Dialog shown if only one source file has been selected
2. Multiple lecture creation dialog: Dialog shown if at least two source files have been selected
3. Single only: The file name of the single source file selected to import
4. Multiple only: A list containing all file names of the source files selected to import
5. Single only: The name of the single lecture which will be created during this import
6. The course to which the lecture(s) will be added
7. The Notioncover Manager type used for this import.
8. Defines whether it is allowed to have a Notion more than once in each created lecture
9. Defines the size of horizon used for Notion search *(only available for Notioncover Manager type "with view horizon")*
10. Defines which method is used to compare the words. This can be used to also detect words which contain typing errors *(only available for Notioncover Manager type "with view horizon")*
11. Typing distance, which is used for some word comparison types
12. Defines where TrucStudio should take the keywords from

Single lecture creation dialog

edit



You see this dialog only if you selected exactly one file to import. This is the only case where you can define the new created lectures name in the dialog.

Name

edit

The name, which is automatically filled for you, is the lectures name guessed from the filename. In most cases this is the filename without the ending '.txt'. Therefore you often don't need to change it.

Course

edit

You have to select a course where you want to create the lecture in. If you don't have any course in your project, you will receive an error message, which indicates you to create one.

Cover Manager

edit

The type of Notioncover Manager used for this extraction.
See 'Notioncover Managers' for further details.

Repeated Notions

edit

You can choose, whether a Notion can be covered at most once or multiple (repeated) times in every lecture.

Size of horizon

edit

This option is only available if you choose 'with view horizon' as Cover Manager. Here you can specify the size of the horizon used.
See 'Link|Notioncover Manager with view horizon' for further details.

Comparison

edit

The algorithm used to compare the words of the source text with the Notion keywords.
See 'Text comparison Methods' for further details.

Distance

edit

The distance is a measure of equality between two words. If a word in the plain text and a keyword have a distance of at most the selected value, they are considered "identical". The higher the distance, the more typing errors are accepted – but also the more wrong results you get.

Keywords

edit

You can choose how TrucStudio selects the keywords which describe the Notions. You can choose between 'Notion name only', 'Summary only' and 'Summary and Notion name'. The first option takes all words of the Notion name (at least 2 characters, no pure number sequences). The second does the same for the Notion summary, and the third uses both - Notion name and summary.

Attention: If a Notion does not contain a single useful keyword which could describe it, then no Notioncover Manager (and therefore no Notioncover) will ever exist for this Notion. This means that this Notion will never be detected automatically in the educational material.

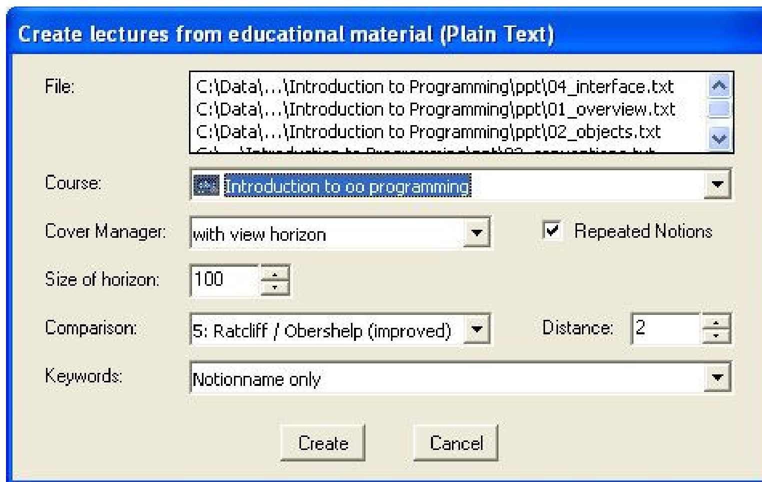
Create

edit

When clicking 'Create' the extraction starts. For larger files this can take some seconds up to about a minute.

Multiple lectures creation dialog

edit



You see this dialog only if you selected multiple files to import. This dialog does not allow you to specify the names of the created lectures. The lectures are automatically named by the file names.

Course

edit

You have to select a course where you want to create the lecture in. If you don't have any course in your project, you will receive an error message, which indicates you to create one.

Cover Manager

edit

The type of Notioncover Manager used for this extraction.
See 'Notioncover Managers' for further details.

Repeated Notions

edit

You can choose, whether a Notion can be covered at most once or multiple (repeated) times in every lecture.

Size of horizon

edit

This option is only available if you choose 'with view horizon' as Cover Manager. Here you can specify the size of the horizon used.
See 'Link|Notioncover Manager with view horizon' for further details.

Comparison

edit

The algorithm used to compare the words of the source text with the Notion keywords.
See 'Text comparison Methods' for further details.

Distance

edit

The distance is a measure of equality between two words. If a word in the plain text and a keyword have a distance of at most the selected value, they are considered "identical". The higher the distance, the more typing errors are accepted – but also the more wrong results you get.

Keywords

edit

You can choose how TrucStudio selects the keywords which describe the Notions. You can choose between 'Notion name only', 'Summary only' and 'Summary and Notion name'. The first option takes all words of the Notion name (at least 2 characters, no pure number sequences). The second does the same for the Notion summary, and the third uses both - Notion name and summary.

Attention: If a Notion does not contain a single useful keyword which could describe it, then no Notioncover Manager (and therefore no Notioncover) will ever exist for this Notion. This means that this Notion will never be detected automatically in the educational material.

Create

edit

When clicking 'Create' the extraction starts. For larger files this can take some seconds up to about a minute.

Notioncover Managers

edit

You can choose between two Notioncover Manager types. A Notioncover Manager tries to find a Notion's coverages in the read text. It may be impossible to change some settings in the dialog according to the Matimport Manager type you selected. We recommend you to select the Notioncover Manager type "with view horizon".

Notioncover Manager with view horizon

edit

This Notioncover Manager uses a window, which slides over the source text. Words which are outside this window are not used to decide whether a Notion is covered or not. This allows to use larger files, where it is possible that one part of a Notions keywords is used in one context at the beginning of the text and the others in a second context near the end of the

same text.

viewhorizon
... to do... type ... for a ... feature ..... and it ...
class ... class ... found. If ... call ... type ... get a ...
1

... to do... type ... for a ... feature ..... and it ...
class ... class ... found. If ... call ... type ... get a ...
2

... to do... type ... for a ... feature ..... and it ...
class ... class ... found. If ... call ... type ... get a ...
3

... to do... type ... for a ... feature ..... and it ...
class ... class ... found. If ... call ... type ... get a ...
4

In the visualization above you can see the following:

1. The blue part in this text is the viewhorizon. In this state, the Notion 'type' is covered.
2. When advancing to the state 2, the Notions 'feature' and class are covered. The Notion 'class' is only covered once, because the viewhorizon is large enough to contain both occurences of 'class'. On the other side, the occurrence of 'type' falls out of the window. So 'type' can be covered later again.
3. In the next state, 'feature' falls out of the viewhorizon and 'call' is not yet read. So the Notioncover 'feature call' is not added. This is only the case because a viewhorizon is used. Otherwise a wrong Notioncover would be added.
4. In the last state, the Notion 'type' is covered for a second time.

The size of this window can be set. Internal test showed best results using a window size of 150 for slides and a size of 200 for scripts.

Simple Notioncover Manager

edit

This Notioncover Manager is very simple and therefore extremely fast. It scans the source file for the Notions keywords and as soon as every keyword has been found, it assumes that the Notion has been covered. So if a Notion has only the keyword 'feature' it will be covered every time the word 'feature' occurs in the read text. Of course there is some filtering which tries to avoid some simple repetitions of Notion coverage, but nevertheless this can result in many incorrect coverages.

Text comparison Methods

edit

To deal with plural forms and typing errors in the source files (slides, scripts, etc.) you have the possibility to accept words, which differ slightly from the searched keywords.

You have the choice between six different text comparison methods:

1. exact comparison
2. Levenshtein Algorithm
3. Levenshtein Algorithm (memory optimized)
4. Damerau Levenshtein Algorithm
5. Ratcliff / Obershelp

exact comparison

edit

If you selected this option, there are no typing errors detected. If the words are not written exactly like the keywords (as you typed it in the Notion's properties), then the algorithms will not detect the corresponding word. On the other side, you can be sure that only those words will be accepted which are typed exactly the same.

Levenshtein Algorithm

edit

If you selected Levenshtein Algorithm as comparison method, then words with the following typing errors are also accepted to be describing the keyword (here 'monday'):

typing error	mistyped word	correct word
insertion	monday**s**	monday
deletion	moday	mo**n**day
substitution	monda**z**	monda**y**

The distance is here the number of typing errors.

Recommended distance : 1

Levenshtein Algorithm (memory optimized)

edit

This Algorithm is useful if you have very large words to compare and are low of memory. It does the same as the normal Levenshtein Algorithm and is memory optimised.

As in the normal Levenshtein Algorithm, the distance is here the number of typing errors.

Recommended distance : 1

Damerau Levenshtein Algorithm

edit

This is an extended Levenshtein Algorithm which finds some more typing errors. Because of this fact it is a little bit slower than the original Algorithm.

If you selected Damerau Levenshtein Algorithm as comparison method, then words with the following typing errors are also accepted to be describing the keyword (here 'monday'):

typing error	mistyped word	correct word
insertion	monday**s**	monday
deletion	moday	mo**n**day
substitution	monda**z**	monda**y**
transposition	mo**dn**ay	mo**nd**ay

As in the original Levenshtein Algorithm, the distance is here the number of typing errors.

Recommended distance : 1

Ratcliff / Obershelp Algorithm

edit

The idea behind Ratcliff / Obershelp Algorithm is different from the others. When using this Algorithm, it is not checked how many typing differences exist between two words, it is compared how long the equally typed parts are (longest substrings). This respects the fact, that short words can have a completely different meaning when doing one simple typing error while longer words often need multiple typing errors to change their meaning.

For this algorithm we can't really speak of a distance. The value interpreted as distance is a value between 0 and 20, where 0 means that the words are exactly equal and 20 means that they do not even share a single character.

Recommended distance : 1

Keyword selection methods

edit

To find a Notion in the text, the algorithms use keywords. These keywords can be found in the Notion's fields 'Name' and 'Summary'. This gives you the following choices to select the keywords which describe a Notion:

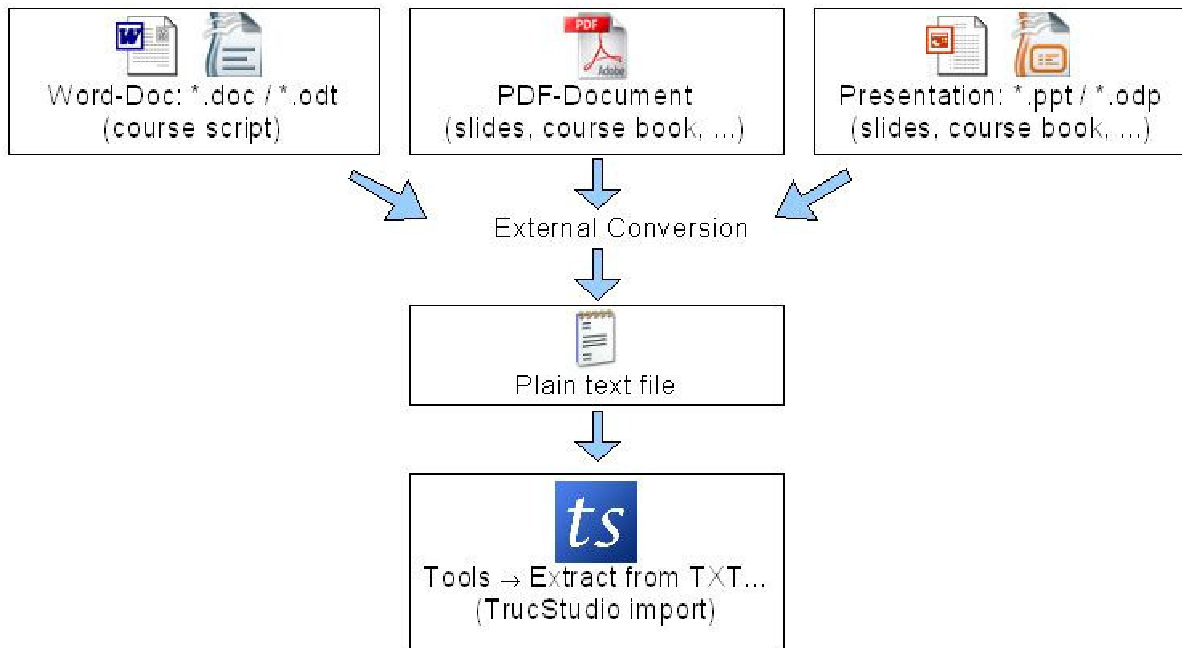
- Notion name only
- Summary + Notion name
- Summary only

It is important to know that each word read from those fields will be used as keyword. This can lead to some undesired effects when including summaries with full sentences. If, for any reason, no keyword can be generated for a Notion (e.g. missing summary when using option 'summary only'), then the algorithm will discover no coverages for the specific Notion. So be sure to name the Notions correctly and use the options which include the summary only if qualitatively good summaries are available for all Notions.

Converting some common file types into plain text files (TXT)

edit

To import any educational material (like presentation slides, scripts or chapters out of a book) the contained text must be available as plaintext (the filename should end with '.txt'). Therefore it might be necessary to extract the plaintext out of a file with different file format. This chapter describes how to do this for some common file types and applications.



Converting PDF into TXT

edit

Most programs which are used to display PDF files have some functionality to extract the written text out of the file. Unfortunately the text is sometimes stored as a picture (contained text is neither selectable nor findable using the integrated text search tool). In this case, no simple solution exists to extract the text into a TXT file and you should look for other sources. In the other case, you should be able to extract the text following one of the instructions below.

Using Adobe Reader (freeware)

edit

1. Open the PDF document with Adobe Reader.
2. Select 'File' -> 'Save as Text...'
3. Select the destination to store the file in.
4. Give it an expressive file name (will be used during import as standard lecture name).
5. Make sure that 'Text (Accessible) (*.txt)' is selected as save type.
6. Push button 'Save'.
7. The program should now extract the text and save it.

Adobe Reader is available at: **http://www.adobe.com**

Using GSview (freeware)

edit

1. Open the PDF document with GSview.
2. Select 'Edit' -> 'Text Extract...'
3. Select the numbers of the pages from which you want to extract the text
4. Select the destination to store the file in.
5. Give it an expressive file name (will be used during import as standard lecture name).
6. Don't forget to add the ending '.txt' to the file name.
7. Make sure that 'Text Files (*.txt)' is selected as save type.
8. Push button 'Save'.
9. The program should now extract the text and save it.

GSview is available at: **http://www.cs.wisc.edu/~ghost/gsview/**

Converting presentation files (PPT, PPS, ODP, ...) into TXT

edit

Using OpenOffice Impress (freeware)

edit

1. Open the document with OpenOffice Impress.
2. Select 'File' -> 'Export as PDF...'
3. Push button 'Export'
4. Select the destination to store the file in.
5. Give it an expressive file name.
6. Make sure that 'PDF - Portable Document Format (.pdf)' is selected as save type.
7. Push button 'Save'.
8. The program should now generate a PDF file.
9. Go ahead and extract the text from the PDF-File as described in the chapter 'Converting PDF into TXT'.

OpenOffice is available at: **http://www.openoffice.org/**

Using Microsoft PowerPoint (commercial)

edit

1. Open the document with Microsoft PowerPoint.
2. Select 'File' -> 'Save as...'
3. Select the destination to store the file in.
4. Give it an expressive file name.
5. Select 'RTF (.rtf)' as save type.
6. Push button 'Save'.
7. The program should now generate a RTF file.
8. Go ahead and extract the text from the RTF-File as described in the chapter 'Converting text documents (RTF, DOC, ODT, ...) into TXT'.

Converting text documents (RTF, DOC, ODT, ...) into TXT

edit

Using OpenOffice Writer (freeware)

edit

1. Open the document with OpenOffice Writer.
2. Select 'File' -> 'Save as...'
3. Select the destination to store the file in.
4. Give it an expressive file name (will be used during import as standard lecture name).
5. Select 'Text (.txt)' as save type.
6. Push button 'Save'.
7. The program should now create a TXT file.

OpenOffice is available at: **http://www.openoffice.org/**

Using Microsoft Word (commercial)

edit

1. Open the document with Microsoft Word.
2. Select 'File' -> 'Save as...'
3. Select the destination to store the file in.
4. Give it an expressive file name (will be used during import as standard lecture name).
5. Select 'Only Text (.txt)' as save type.
6. Push button 'Save'.
7. The program should now create a TXT file.

Home

Developer Guide: Material Import

developer guide | import | MATIMPORT

Developer guide home
User guide: Material Import

Table of Contents

- Introduction
- Material Import Manager
- Model
- Settings-Dialogs
- Notioncover Manager
  - Simple Notioncover Manager
  - Simple Notioncover Manager II
  - Notioncover Manager with viewhorizon
- Notioncover
- File Reader
  - Plain text file reader
- Text Comparison Methods
  - exact comparison
  - Levenshtein algorithm
    - Example
  - Memory optimised Levenshtein algorithm
    - Example
  - Damerau Levenshtein algorithm
    - Example
  - Ratcliff / Obershelp algorithms
    - Example
    - Differences between the three Ratcliff / Obershelp algorithms
- Keyword Generation
- Tools

edit

Introduction

Material Import, or short Matimport, is the program part developed during the master thesis of Adrian Roman Müller. It provides TrucStudio with an automation facility to extract (already known) Notions from existing educational material – a task which required a lot of manually work before.

The Material Import Tool consists of the following parts:

- The Material Import Manager is the class controlling the import.
- A view, the settings-dialog, gives the user the possibility to change the settings concerning the import.
- A model stores the data used for the settings-dialog.
- For each Notion in the domain model there exists a Notioncover Manager. Here the Notions coverages are detected.
- A Notioncover is a coverage of a Notion in the text file, detected by a Notioncover Manager.
- A File Reader extracts the words out of a source file.
- Text comparison methods deal with typing errors and plural forms of keywords.
- A Keyword Generator extracts keywords describing a Notion from its Summary and/or Notion name.
- Some general tools support TrucStudio with additional functionalities.

This section of the TrucStudio Wiki describes all information needed to maintain and / or extend this program part. The folder '/tstudio/import/material_import' contains the classes created for this program part prefixed with 'TS_MATIMPORT_'. If a class should not be located in the

mentioned folder, then the chapter describing this class will mention this.

Material Import Manager

edit

Classes:

- TS_MATIMPORT_MANAGER

The Material Import Manager is the controller used for Matimport and controls the entire importing process. Therefore it is the entry point for the launch of the extraction process. On creation time, it will create also the fitting model and a first dialog. At this time it's not clear what the exact contents of the dialog will be, but to fulfill the invariant, there must be created a view nonetheless. The dialog window that is presented to the user will be created each time the dialog is shown (earlier used dialog parts will be restored).

A call to `process_import_dialogs` launches the extraction process. This feature first creates and displays a file-open dialog of type `EV_FILE_OPEN_DIALOG`. In this dialog, the user can select one or more source files to import. According to the number of files selected, the features `execute_single_file_dialog` or `execute_batch_dialog` will be called. These features display a dialog of the type `TS_D_MATIMPORT_ADD_LECTURE`, in which the user can specify the extraction settings. With this information the features prepare the Notioncover Managers and start for each source file the feature `start_import` with a new lecture for each source file.

The feature `start_import(l:TS_LECTURE)` handles the main aspects of Notion extraction. At this point, all settings have been done and the input file is open. In a first step, the feature uses a file reader (descendant of `TS_MATIMPORT_FILE_READER`) and reads the source file word by word. As soon as a word is read, it is registered in all available Notioncover Managers by using their feature `scan_word`. This makes sure that each Notioncover Manager is now up to date. In a second step, the feature reads the coverages from each of the Notioncover Managers, sorts them by their appearance in the text and filters out Notioncovers which belong to the same Notion and are at most two covers away. Based on the filtered list of Notioncovers, the corresponding Notions are added in the right order to the lecture's covered Notions.

Model

edit

Classes:

- TS_M_MATIMPORT_ADD_LECTURE

This model stores some data used by the settings-dialogs. So also complete dialog parts can be saved here for reuse.

Not only complete dialog items but also their content is stored here. So the whole context (without file names, lectures names and available courses) can be accessed from a dialog.

Settings-Dialogs

edit

Classes:
The following classes can be found in the cluster `/tstudio/view/dialogs`

- TS_D_MATIMPORT_ADD_LECTURE *(deferred)*
- TS_D_MATIMPORT_ADD_MULTIPLE_LECTURES
- TS_D_MATIMPORT_ADD_SINGLE_LECTURE

There exist two different dialogs where the user can specify the extraction settings. The dialog `TS_D_MATIMPORT_ADD_SINGLE_LECTURE` is called if the user wishes to import only one single file, and the dialog `TS_D_MATIMPORT_ADD_MULTIPLE_LECTURES` is called when the user wishes to import two or more files. The only difference between these two dialogs are the fields 'File' and 'Name'. As you can see in the pictures below, there is no field 'Name' in the dialog `TS_D_MATIMPORT_ADD_MULTIPLE_LECTURES`. Instead of that, the field 'File' is extended to a

scrollable list.

TS_D_MATIMPORT_ADD_SINGLE_LECTURE

TS_D_MATIMPORT_ADD_MULTIPLE_LECTURES

Because of the fact that a big part of these dialogs is identical, there exists the deferred class TS_D_MATIMPORT_ADD_LECTURE. Every feature needed to read the dialog for use in the Material Import Manager is located here.

There are many different options you can choose in a settings-dialog. These are often connected somehow (e.g. there could be value restrictions in other fields). This makes it necessary that there is some checking and correction algorithm launched when such an option has been changed. These can be found in the 'Implementation' part on the bottom of the source code and are called e.g. 'on_select_comparison'.

A controller can easily check whether the options have been correctly entered and the dialog has been closed by pressing the 'Create' button. The attribute 'do_import' provides this information, which is set to true if the import can be started.

Notioncover Manager

edit

Classes:

- `TS_MATIMPORT_NCOVER_MANAGER` *(deferred)*
- `TS_MATIMPORT_NCOVER_MANAGER_VIEWHORIZON`
- `TS_MATIMPORT_SIMPLE_NCOVER_MANAGER`
- `TS_MATIMPORT_SIMPLE_NCOVER_MGR_II`

A Notioncover Manager implements the relation between a Notion and the Material Import manager. So it contains the information about which keywords must be in some course material to make sure it covers the Notion, gets informed about each word read from the current source file and checks whether (and when) a Notion has been covered by the read educational material.

The deferred class `TS_MATIMPORT_NCOVER_MANAGER` contains the features which have to be in each implementation of a Notioncover Manager. The most important one is the feature `scan_word`. Through this feature, the Notioncover Manager knows about a newly read word and its position in the text, so that it can compare the word with its keywords and decide whether a coverage has been found.

The feature `add_coverages` takes a list and adds for each found coverage a Notioncover to it. This feature is used in the Matimport Manager to get all Notioncovers of a lecture. If the flag 'Repeated Notions' in the settings-dialog is selected, then there will be multiple coverages per Notion possible (multiple covers mode), otherwise there will only be the first coverage returned (single cover mode).

At the moment there exist three different implementations of Notioncover Managers. These are:

- Simple Notioncover Manager (only single cover mode)
- Simple Notioncover Manager II (only multiple covers mode)
- Notioncover Manager with viewhorizon (single and multiple covers modes)

Simple Notioncover Manager

edit

only single cover mode

This is a very simple Notioncover Manager. So it does not support many of the setting options. When scanning a word, it is compared exactly with the Notions keywords. As soon as it has found all keywords at least once, it reports a coverage. The Hash-Table `first_keyword_occurrences` stores each keyword's first occurrences. So the position of a found coverage can easily be calculated by searching the maximum position stored here.

Simple Notioncover Manager II

edit

only multiple covers mode

This is a enhancement of the Simple Notioncover Manager. Because it is a bit slower than the Simple Notioncover Manager, it does only provide the multiple covers mode which is not covered by the Simple Notioncover Manager.

When scanning a word, it is compared exactly with the Notions keywords. The Hash-Table 'keyword_occurences' stores its position in a list which contains all occurrences of this keyword. To calculate the coverages, the feature `calculate_notion_covers` gets at first all minimal positions of those lists. This determines the position of the first Notioncover (maximum of the results). For the next Notioncovers it always takes those entries in the lists which are the next higher ones (from the last Notion's position) and calculates from the results the position of the next Notioncover. This can be done until a keywords list contains no more positions larger than the one of the last found coverage.

Notioncover Manager with viewhorizon

edit

simple and multiple covers modes

This is the most complex of the three Notioncover Managers and is located in the class `TS_MATIMPORT_NCOVER_MANAGER_VIEWHORIZON`. When scanning the words, it works with a sliding

window, the viewhorizon, in which the keyword-ids of the scanned words are inserted. Of course the viewhorizon size must be corrected as soon as it exceeds the limit chosen by the user. This is reached by throwing away the oldest entries.

If a keyword has been found, the value called activation_level and the keyword's counter in found_coverages are increased by one, therefore they must be decreased by one when the same keyword falls out of the sliding window.

After the viewhorizon has been updated, the Notioncover Manager checks for new covers. A cover can be found if the activation_level has reached a minimum value (coverage_enabling_value), all keywords have been found at least once within the viewhorizon (found_coverages) and there is no older coverage active (cover_is_active). If all these conditions hold, the new Notioncover is created with the current position and cover_is_active is set to true.

The flag cover_is_active prevents multiple entries per coverage because it blocks new coverage detections until is set to false when the activation_level gets under a given limit (coverage_disabling_value). This is a mechanism which can also be found in neural networks.

The values of coverage_enabling_value and coverage_disabling_value depend on how many keywords describe a Notion and are calculated set within the feature recalc_coverage_limits

Notioncover

edit

Classes:

- TS_MATIMPORT_NCOVER

A Notioncover represents a single coverage of a Notion in a text. It consists of the position, where it appeared in the text, and of the Notion which is covered. To keep the implementation simple, the class inherits from DS_PAIR[INTEGER,TS_NOTION]. This provides the class with all needed features to store and recover two attributes.

Because it has to be possible to sort a list of Notioncovers by their positions, the class also inherits from COMPAREABLE. So the feature infix "<" is the only one coded in this class.

File Reader

edit

Classes:

- TS_MATIMPORT_FILE_READER *(deferred)*
- TS_MATIMPORT_TXT_READER

The deferred class TS_MATIMPORT_FILE_READER defines the functionality which is provided by a Matimport file reader. These are more or less those of a normal file reader but has some special functionalities. The contained text is read word by word. When calling the feature forth the input file is read until a valid word could be found or the end of the text is reached.

To decide whether a word is good, the feature is_bad_string can be used. For the moment, this feature tells whether the given string is a pure number sequence or consists of less than two characters. If one of this is the case, the result would be true and therefore the given string no good word.

For every word there is also its text position provided. How this position is being calculated depends on the implemented file reader.

At the very moment there only exists an implementation for plain text files (*.txt). But the data structure is extensible and therefore prepared to support additional file readers supporting other file types.

Plain text file reader

edit

This file reader builds on a KL_TEXT_INPUT_FILE to read from the source, which allows to read the source file line by line. So there exists a feature read_next_line which reads a line from the source, cleans it from undesired characters, extracts all possible words from this line and fills them into a queue. This feature is always called when the queue is empty and forth is called. The feature forth takes words from the queue until a valid word could be found and prepares it for access. Therefore it is possible that read_next_line is called multiple times during the same forth call.

The text position is the word count (good and bad words are counted).

Text Comparison Methods

edit

Classes:

- TS_MATIMPORT_STRINGCOMPARE

There are different methods to deal with typing errors and plural forms in a source file. These methods are located in the class TS_MATIMPORT_STRINGCOMPARE. Notioncovers Manager use them while a new word is scanned. The comparison method can be set using the feature set_standard_comparison_method. Having done this, two strings can be compared by equal_string which automatically calls the corresponding feature.

There are seven different algorithms implemented, but only five of them can be selected in TrucStudio. The others have been removed because they could be replaced by faster versions of the same principle. The following types are implemented:

- Exact comparison
- Levenshtein algorithm
- Memory optimised Levenshtein algorithm
- Damerau Levenshtein algorithm
- Ratcliff / Obershelp algorithm *(removed)*
- Improved Ratcliff / Obershelp algorithm
- Slow Ratcliff / Obershelp algorithm *(removed)*

exact comparison

edit

When choosing the type type_exact_compare, then an exact comparison will be done using the standard comparison algorithm of the String class (is_equal).

Levenshtein algorithm

edit

The feature levenshtein_equal, which is called if the standard comparison algorithm is set to type_levenshtein_compare, executes the normal Levenshtein algorithm using a table (ARRAY2[INTEGER]). In the first part of the algorithm, the first row and column are filled with ascending values. In a second step the values of all other fields are filled from left to right and top down.

If the last value (down on the right) is smaller or equal to the maximum allowed distance, then the words are considered levenshtein-equal within the given distance. The distance means here the number of insertions, deletions and substitutions.

Example

edit

Assuming we have the two words "Levnshtien" and "Levenshtein", which we want to compare:

We build a table and fill it with the starting values.

	.	**L**	**E**	**V**	**E**	**N**	**S**	**H**	**T**	**E**	**I**	**N**
.	0	1	2	3	4	5	6	7	8	9	10	11

L	1	?	?	?	?	?	?	?	?	?	?	?
E	2	?	?	?	?	?	?	?	?	?	?	?
V	3	?	?	?	?	?	?	?	?	?	?	?
N	4	?	?	?	?	?	?	?	?	?	?	?
S	5	?	?	?	?	?	?	?	?	?	?	?
H	6	?	?	?	?	?	?	?	?	?	?	?
T	7	?	?	?	?	?	?	?	?	?	?	?
I	8	?	?	?	?	?	?	?	?	?	?	?
E	9	?	?	?	?	?	?	?	?	?	?	?
N	10	?	?	?	?	?	?	?	?	?	?	?

Now we can use for each cell the following rules:

If the two characters are equal, then take the minimum of:

- the value one up and one left (keep the character)
- the value one up, added by 1 (deletion)
- the value one left, added by 1 (insertion)

	.	**L**
.	0	1
L	1	?

min(0,2,2) = 0

If the two characters are unequal, then take the minimum of:

- the value one up and one left, added by 1 (substitution)
- the value one up, added by 1 (deletion)
- the value one left, added by 1 (insertion)

	L	**E**
.	1	2
L	0	?

min(2,3,1) = 1

When using this rules, the complete table can be filled like that:

	.	**L**	**E**	**V**	**E**	**N**	**S**	**H**	**T**	**E**	**I**	**N**
.	0	1	2	3	4	5	6	7	8	9	10	11
L	1	0	1	2	3	4	5	6	7	8	9	10
E	2	1	0	1	2	3	4	5	6	7	8	9
V	3	2	1	0	1	2	3	4	5	6	7	8
N	4	3	2	1	1	1	2	3	4	5	6	7
S	5	4	3	2	2	2	1	2	3	4	5	6
H	6	5	4	3	3	3	2	1	2	3	4	5
T	7	6	5	4	4	4	3	2	1	2	3	4
I	8	7	6	5	5	5	4	3	2	2	2	3
E	9	8	7	6	5	6	5	4	3	2	3	3
N	10	9	8	7	6	5	6	5	4	3	3	**3**

The value on the bottom right shows us now the typing distance, which is here 3. With this distance, the mistyped word "Levnshtien" will be accepted only if a distance of 3 and more has been chosen.

Memory optimised Levenshtein algorithm

edit

The feature `levenshtein_equal_mem_opt`, which is called if the standard comparison algorithm is set to `type_levenshtein_compare_memory_optimized`, does nearly the same as if you chose to run the normal Levenshtein algorithm. The only difference is that this implementation does not use a table. It deals with a one dimensional array, which has the length of a single line in the standard implementation's table. This is reached by writing a computed value not directly to the

array but storing it during the calculation in a integer variable called `cur` and saving it after the calculation into the integer variable `last` while the old `last` is written to the array. This allows to access the value which would have been written to the cell one up and left from the current cell when using a table. This value is still in the array and is accessed for the substitution calculation or if both current characters are the same.

So the only difference between the normal Levenshtein algorithm is that this algorithm reduces memory consumtion. The result will always be the same – so if the last value in the array is smaller or equal the maximum allowed distance, then the words are considered levenshtein-equal within the given distance. The distance means here the number of insertions, deletions and substitutions.

Example

edit

Assuming we have the two words "Levnshtien" and "Levenshtein", which we want to compare:

We build the array and fill it with the starting values.

	.	L	E	V	E	N	S	H	T	E	I	N
.	0	1	2	3	4	5	6	7	8	9	10	11
-												

Additionally, we create the variables 'cur' and 'last' which will store the values calculated at the current respectively last step.

Now we iterate over the word "Levnshtien" and process for each character the whole line. We use for each cell the following rules:

If it's the first place of the line:
Take the first value of the array, add 1, and store it in 'cur'

Otherwise, if the current character of "Levnshtien" is equal to the character belonging to the current cell, store the minimum of the following results into 'cur':

- the value one left to the current array position (keep the character)
- the value of the current array position, added by 1 (deletion)
- the value in 'last', added by 1 (insertion)

Otherwise (the current character of "Levnshtien" is unequal to the character belonging to the current cell) store the minimum of the following results into 'cur':

- the value one left to the current array position, added by 1 (substitution)
- the value of the current array position, added by 1 (deletion)
- the value in 'last', added by 1 (insertion)

Before going to the next cell position, the value in 'last' is stored into the cell at the position one to the left. Afterwards the value in 'cur' overwrites the value in 'last'.

When using this rules, the array will look at the end like that, with a value of 3 stored in 'cur':

	.	L	E	V	E	N	S	H	T	E	I	N
N	10	9	8	7	6	5	6	5	4	3	3	3

This algorithm processes like the normal one the complete table but overwrites unneeded old values by newer ones. This is why it only needs a one dimensional array and not a complete table.

The value in 'cur' gives us now the typing distance, which is in this example 3. With this distance, the mistyped word "Levnshtien" will be accepted only if a distance of 3 and more has been chosen.

Damerau Levenshtein algorithm

edit

Setting standard comparison algorithm to `type_damerau_levenshtein_compare`, results in a call

to `damerau_levenshtein_equal`. This feature uses the Damerau Levenshtein comparison, which is a enhancement of the Levenshtein algorithm. The difference between these two algorithms is that the Damerau Levenshtein algorithm also handles transpositions of two neighbouring characters. This is achieved by looking at the current two characters and additionally at the past two characters. If the first word's current character is the same as the second word's past character and the first word's past character the same as the second word's current character then a transposition is found. In this case the value two up and two left in the table is taken and added by one, to calculate the distance. You can find the implementation on the bottom of the inner loop.

If the last value (down on the right) is smaller or equal the maximum allowed distance, then the words are considered damerau-levenshtein-equal within the given distance. The distance means here the number of insertions, deletions, substitutions, and transpositions.

Example

edit

Assuming we have the two words "Levnshtien" and "Levenshtein", which we want to compare:

We build a table and fill it with the starting values.

	.	L	E	V	E	N	S	H	T	E	I	N
.	0	1	2	3	4	5	6	7	8	9	10	11
L	1	?	?	?	?	?	?	?	?	?	?	?
E	2	?	?	?	?	?	?	?	?	?	?	?
V	3	?	?	?	?	?	?	?	?	?	?	?
N	4	?	?	?	?	?	?	?	?	?	?	?
S	5	?	?	?	?	?	?	?	?	?	?	?
H	6	?	?	?	?	?	?	?	?	?	?	?
T	7	?	?	?	?	?	?	?	?	?	?	?
I	8	?	?	?	?	?	?	?	?	?	?	?
E	9	?	?	?	?	?	?	?	?	?	?	?
N	10	?	?	?	?	?	?	?	?	?	?	?

Now we can use for each cell the following rules:

If the two characters are equal, then take the minimum of:

- the value one up and one left (keep the character)
- the value one up, added by 1 (deletion)
- the value one left, added by 1 (insertion)

	.	L
.	0	1
L	1	?

min(0,2,2) = 0

If the two characters are neither equal nor are both equal with each others parent, then take the minimum of:

- the value one up and one left, added by 1 (substitution)
- the value one up, added by 1 (deletion)
- the value one left, added by 1 (insertion)

	L	E	V
.	1	2	3
L	0	1	2
E	1	0	?

min(2,3,1) = 1

If the two characters are unequal, but both are equal with each others parent, then take the minimum of:

- the value one up and one left, added by 1 (substitution)
- the value one up, added by 1 (deletion)
- the value one left, added by 1 (insertion)
- the value two up and two left, added by 1 (transposition)

	T	**E**	**I**
T	1	2	3
I	2	2	2
E	3	2	?

min(3,3,3,2) = 2

When using this rules, the complete table can be filled like that:

	.	**L**	**E**	**V**	**E**	**N**	**S**	**H**	**T**	**E**	**I**	**N**
.	0	1	2	3	4	5	6	7	8	9	10	11
L	1	0	1	2	3	4	5	6	7	8	9	10
E	2	1	0	1	2	3	4	5	6	7	8	9
V	3	2	1	0	1	2	3	4	5	6	7	8
N	4	3	2	1	1	1	2	3	4	5	6	7
S	5	4	3	2	2	2	1	2	3	4	5	6
H	6	5	4	3	3	3	2	1	2	3	4	5
T	7	6	5	4	4	4	3	2	1	2	3	4
I	8	7	6	5	5	5	4	3	2	2	2	3
E	9	8	7	6	5	6	5	4	3	2	2	3
N	10	9	8	7	6	5	6	5	4	3	3	**2**

The value on the bottom right shows the typing distance, which here is 2. With this distance, the mistyped word "Levnshtien" will be accepted only if a distance of 2 and more has been chosen.

Ratcliff / Obershelp algorithms

edit

The main difference between the Ratcliff / Obershelp algorithms to the algorithms described above is, that not a typing distance is being calculated but a value for coverage. The algorithm calculates recursively the length of the the longest substrings found in both words. So it calculates in a first step the longest substrings of both words and adds recursively 0.9 times the result for the words resulting when those substrings are removed. When no substring could be found, the substring length of 0 will be returned.

Having calculated this recursive substring length, a rating of equality is being calculated where 0 means analogue to a distance that both words are identical. If the two words do not share a single character, then a rating of 20 will be calculated.

To calculate the rating, the recursive substring length will be divided by the average word length. Then the resulting coverage value must be subtracted from 1 to receive a value where 0 stands for exact equality. By multiplying with 20 we reach the final rating between 0 and 20.

The algorithm treats the word parts left and right to the found substring together. This allows to find more of the hidden substrings but leads to a deeper recursion.

If the computed rating is smaller or equal the maximum allowed distance, then the words are told to be ratcliff-obershelp-equal within the given ranking. As mentioned ahead, the distance does not mean here any typing distance but is meant as some rating of coverage with 0 as the best cover.

Example

edit

Assuming we have the mistyped word "REacliffft" and want to compare it with the keyword "Ratcliff":

recursion step	mistyped word	correct word	substring length
1	REA**CLIFF**T	RAT**CLIFF**	5
2	RE**AT**	R**AT**	2

3	**R**E	**R**	1
4	E	*(empty)*	0

recursive substring length = $5 + 0.9 * 2 + 0.9^2 * 1 = 7.61$
average word length = 8.5
rating = $20 * (1 - 7.61/8.5) = 20 * (1 - 0.895) = 20 * 0.105 = 2.1$

With a rating of 2.1, the mistyped word "REaclifft" will be accepted only if a distance of 3 and more has been chosen.

Differences between the three Ratcliff / Obershelp algorithms

edit

The three implemented Ratcliff / Obershelp algorithms only differ in how the recursive substring length is being calculated.

The slowest algorithm (`ratcliff_obershelp_equal_slow`) searches for the largest possible substring, beginning with the complete shorter word, and continuing with all possible substrings which are one character shorter, and so on. In worst case this algorithm has a runtime of O(n^3). You can easily check with the example using the words "ABCDEFG" and "GFEDCBA".

The normal algorithm (`ratcliff_obershelp_equal`) improves the slow one by taking into account that a substring is missingif no substring for some part of it is there. So the algorithm takes the first character of the smaller word and tries to find it in the larger word. If successful it enlarges the word by the next character and tries again. At some point the substring won't be in the larger word. In this case the algorithm lets the current substring size unchanged (we are now only interested in larger substrings than we already found) and shifts the start of the substring, so that the substring begins now at the character which was the old ones second character. At each program step gets a new character at the end of the searched substring. So the algorithm has for each recursion step exactly as many steps as the shorter word has characters. In worst case this algorithm has a runtime of O(n^2). You can check this again with the example "ABCDEFG" and "GFEDCBA".

The improved algorithm, which is the fastest, not only does the improvement done in the normal algorithm but also improves the recursion depth. When called, the algorithm knows what ranking he should reach. So the algorithm can calculate for each recursion step what recursive substring length has still to be reached to consider the words ratcliff-obershelp-equal (value of `border`). So the recursion is stopped if the value has already been reached (`border` is negative)or is no longer reacheable (the length of the smaller word is shorter than `border`).

Keyword Generation

edit

Classes:

- `TS_MATIMPORT_KEYWORD_GENERATOR`

When creating a Notion a user does not really want to provide the program with keywords describing it. So the Material Import has to deal with very little information given through the fields 'Name' and 'Summary', which can be found in a Notion's properties. Unfortunately, the Summary is often not filled and if, then it is filled with complete sentences.

So the approach used is that the program reads those fields and uses the contained words as keywords. To do this there exists a Keyword generation class. In the settings-dialog the user can choose between the following three options:

- Notion name only
- Summary only
- Summary and Notion name

According to the user's choice the features `generate_keywords_from_notionname`, `generate_keywords_from_summary` or both are processed when the feature `keywords` is called. The list returned contains only those keywords which contain of more than one character and are

no pure numbers. This filtering is done in the feature `clean_keywordlist`.

Attention: If a Notion does not contain a single useful keyword which could describe it, then no Notioncover Manager (and therefore no Notioncover) will ever exist for this Notion. This means that this Notion will never be detected automatically in the educational material.

Tools

edit

Classes:
The following class can be found in the cluster `/tstudio/model`

- `TS_TOOLS`

Some general functionalities which can be also used outside the Matimport part (but was introduced when implementing Matimport), are located in the class `TS_TOOLS`. At the moment this class is still very small. It contains the feature `string_clean` which replaces in a string characters which are known to make sometimes trouble. This feature can be called using the two levels 'normal clean' and 'full clean' where 'full clean' also does replacements of less dangerous characters. This feature is used for example inside a file reader.

The feature `shrink_path` can be used to shorten a files path to a length which can b displayed inside a GUI. In first priority this feature tries to write the whole file name, then the drive, and if it is still some space it adds some folders lying between. Where the path has been shorted, three points will be inserted.