

Dokumentation: DuplikateFinder / DuplicateFinder

Aufgabe gestellt von: Peter Rill

Aufgabe bearbeitet durch: Jan Mothes

Aufgabenstellung:

Doppelte Dateien suchen (Name/Inhalt/Größe),
Aktionen mit gefundenen Duplikaten durchführen

Inhalt:

1. Ausformulierte Aufgabenstellung
2. Implementation Description (Englisch)
3. Manual (Englisch)
4. Discussion (Englisch)

1. Ausformulierte Aufgabenstellung

Es sollen in einem vom User asugewählten Ordner Duplikate gesucht werden. Der Benutzer soll einstellen können, wie die Suche nach Duplikaten durchgeführt wird und bestimmen können, nach welchen Gesichtspunkten das Programm 2 Dateien als Duplikate erkennt.

Das Programm soll nicht darauf beschränkt sein, nur Duplikat-Paare anzuzeigen, sondern auch in der Lage sein, mehr als 2 Dateien als Duplikate zu behandeln und gruppiert anzuzeigen.

Verschiedenen Methoden sollen es ermöglichen, schnell und eher ungenau oder auch langsam und dafür genauer nach Duplikaten zu suchen, je nach Anwendungsfall.

Beispiele für Anwendungsfälle:

- Der User möchte Dateien als Duplikate identifizieren, die sowohl den gleichen Namen als auch den gleichen Inhalt haben. Er möchte nicht Dateien als Duplikate identifizieren, die zwar den gleichen Namen aber nicht den gleichen Inhalt haben und umgekehrt.
- Der User möchte Dateien als Duplikate identifizieren, die den gleichen Inhalt haben, unabhängig vom Namen.
- Der User möchte Dateien als Duplikate identifizieren, die den gleichen Namen haben, möchte aber nicht den Inhalt prüfen.
- Der User möchte Dateien als Duplikate identifizieren, die einen ähnlichen Inhalt haben, unabhängig vom Namen.

Der User möchte in der Lage sein, eine zu lange dauernde Suche abbrechen zu können. Außerdem möchte er Informationen wie Dateipfad und Dateiname für gefundene Duplikate einsehen können und die Duplikate löschen können.

2. Implementation Description

2.1 Finding Duplicates

The process of finding duplicates can be split into multiple stages:

2.1.1 Identifying candidates

Candidates are the files, that will be checked for duplicates in the later stages. The user is given some options to restrict the amount of files to analyze in this stage, before the program even tries to find duplicates. This stage is needed for

- the user to restrict the amount of results to what he is interested in and for
- achieving good performance

Settings in the GUI which are related to this stage are:

- Root search directory: Only files below this directory will be candidates
- Do not search subdirectories: Only files directly inside the root search dir will be candidates
- Filter by file pattern: Only files that match the given pattern will be candidates

2.1.2 Determine which candidates are duplicates

In this stage, the candidates are grouped into groups/buckets based on what the user defined as „being a duplicate“. The grouping process is split into two conceptually different stages (again):

For the first stage („MatcherTypes“), only a single criteria from a provided set can be chosen by the user to determine equality of files. The reason for that is that these criteria exclude each other and/or build on each other. This stage can be bypassed.

In the second stage („MatcherConstraints“), which is optional unless the first was bypassed, multiple criteria can be chosen from a provided set, and a file must be equal for every selected criteria for it to be considered a duplicate of another one (effectively an AND operation).

The criteria, that this program implements for the first stage, are:

- Length
- Hash
- Content
- Similarity Threshold

Performance of these methods varies widely. To be able to compare performance, let $O(n)$ be the time complexity for reading the content of n files.

2.1.2.1 Length

When this comparison method is selected, only the file length will be used to determine equality between two files. Since the files do not need to be read, this method is very fast ($O(1)$), but error-prone due to the possibility of declaring two very different files as equals. If this method is combined with name matching (see second stage) though, the errors can be reduced and this method becomes very useful.

2.1.2.2 Hash

This method uses the length method first to reduce the number of candidates (filtering out all files that have a unique file size). It then computes the MD5 hash of each leftover candidate file from its

byte content, and considers files with the same hash to be equal. While an equal hash does not guarantee equal content, hash collisions are rare unless maliciously provoked. With a time complexity of $O(n)$ from reading every file once, it is still performant enough for most purposes and thus the recommended method.

2.1.2.3 Content

This method first uses the Hash method to group files by hash. For every group that was found by the hash method, it compares every file of that group with every other file of that group pair-wise by checking their content using a simple byte-by-byte comparison. This is much faster than comparing each candidate with each other candidate ($O(n!)$), because only files that are very likely to be duplicates are content-checked. Its worst-case time complexity is $O(n + (m * k))$, where m is the number of found hash groups and k is the number of files inside the biggest hash group. This method guarantees to have a correct result concerning file content equality.

2.1.2.4 Similarity

This is a more sophisticated method, which uses the Levenshtein distance algorithm to calculate a score from two given files. This score is the minimum number of operations (insert byte, remove byte, substitute byte) that, when applied to one of the given files, will change that file so that its byte content is equal to the other file. The score can be used to easily calculate „similarity“ of two files to each other, and thus allows for determining two files as equal, if their similarity exceeds a given threshold value.

The time this algorithm needs to calculate similarity between two files is roughly $O(i * j)$, where i and j are the lengths of the files respectively (in Bytes). Since the similarity is always specific to only two compared files, the algorithm needs to compare each file with each other file and can only group 2 files into the same group. As a result, the time complexity for this method is $O((i*j) * n!)$, which can be extremely slow for big and/or many files.

To be able to use this method without unreasonable slowdown/waiting, this method ignores any files bigger than 20 KB.

The second stage provides the following criteria:

- Name
- Creation date
- Modified date

These can be mixed with each other and the first stage criterium. If the first stage is bypassed, at least one of these must be selected. All criteria must be fulfilled between two files for them to be considered equal.

Selecting a criterium from the first stage and all criteria of the second stage and will likely result in no files being considered equal and thus will not yield any results.

2.2 Presenting Found Duplicates

After finding the duplicate groups, they must be presented. Two information types are displayed for each found group:

2.2.1 Criteria information

The values of the selected criteria, that caused the files in a group to be considered equal to each other. This is useful information (could show hash for each group when using Hash method, or file length etc.), that only needs to be displayed once per group since it is exactly equal for every file in the group. It also illustrates to the user, how all criteria are combined (AND operation), since the

user can see the group „key“ consisting of the values from all criteria he selected.

2.2.2 General file information

For each file in each group, the most relevant file information is shown, namely its path relative to the chosen root search directory and its name.

2.3 Dealing With Found Duplicates

Since the user most likely wants to select some of the duplicates in each group and delete them, he can do so. The deleted files will be moved to the Recycle Bin in case a deletion was done by mistake.

2.4 Asynchronous Execution

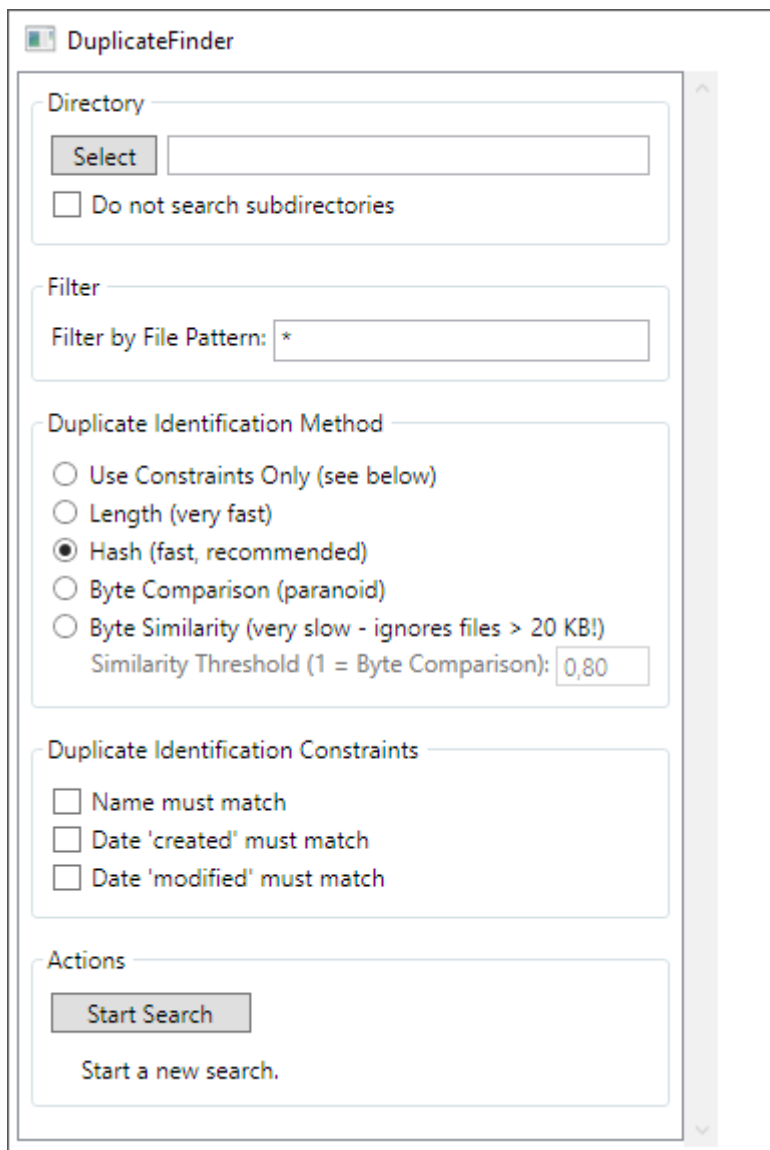
The search for duplicates executes asynchronously. This allows the user to cancel the execution of the search while it is still running, because his GUI is usable during the search. Every part of the search checks in between reasonable time intervals if a cancellation was requested and will abort the search if it was.

When all duplicate groups have been determined, the asynchronous task, which holds all the result, sends each file group to the UI thread separately with small pauses in between. This ensures that the UI thread does not get overwhelmed by a big amount of additional elements to display, and allows the GUI to remain responsive.

3. Manual

The GUI has two distinct areas: The left one is for setting up a search and executing it, the right one is for displaying the results of a search and executing actions on the results.

3.1 Search Settings



The screenshot shows the 'DuplicateFinder' application window. The 'Directory' section has a 'Select' button and a text field, with a checkbox for 'Do not search subdirectories'. The 'Filter' section has a 'Filter by File Pattern:' label and a text field containing '*'. The 'Duplicate Identification Method' section has five radio buttons: 'Use Constraints Only (see below)', 'Length (very fast)', 'Hash (fast, recommended)' (which is selected), 'Byte Comparison (paranoid)', and 'Byte Similarity (very slow - ignores files > 20 KB!)'. Below these is a 'Similarity Threshold (1 = Byte Comparison):' label and a text field with '0,80'. The 'Duplicate Identification Constraints' section has three checkboxes: 'Name must match', 'Date 'created' must match', and 'Date 'modified' must match'. The 'Actions' section has a 'Start Search' button and the text 'Start a new search.'

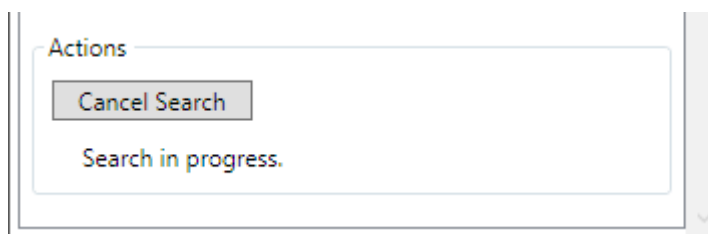
Select the root search directory. Use the option to not search subdirectories recursively.

Use simple file patterns like *.txt to reduce the number of files that will be analyzed.

Stage 1:
Choose a method for determining whether 2 files are „duplicates“. „Use Constraint only“ will bypass this stage.

Stage 2:
Choose additional constraints, that must be fulfilled for 2 files to be considered „duplicates“.

Start the search and use the status message below to determine when the search is complete.



The screenshot shows the 'Actions' section of the GUI. It contains a 'Cancel Search' button and the text 'Search in progress.'

A running search can be stopped as long as it hasn't started adding the found groups to the UI.

3.2 Search Results

When all groups have been found, they will be added to the result page on the right side. It may take a while until all groups have been added. The status message will indicate when all groups have been added.

Each group displays its key at the top, which shows the attributes which are the same for all files in a group, and which are the reason why they were detected as duplicates. Which attributes are used, depends on the used search settings.

Now it is possible to select duplicate files and delete them using the button „Delete Selected File“ (greyed out in this picture since no file is selected). The search can be restarted with different settings to yield a better result.

The screenshot shows the DuplicateFinder application window. On the left, the settings panel includes a 'Directory' section with a 'Select' button and a text field containing 'C:\Users\Jan\Dropbox', and a checkbox for 'Do not search subdirectories'. The 'Filter' section has a 'Filter by File Pattern:' label and a text field with an asterisk. The 'Duplicate Identification Method' section has four radio buttons: 'Use Constraints Only (see below)', 'Length (very fast)', 'Hash (fast, recommended)' (which is selected), and 'Byte Comparison (paranoid)'. Below these is a 'Byte Similarity (very slow - ignores files > 20 KB!)' section with a 'Similarity Threshold (1 = Byte Comparison):' label and a text field containing '0,80'. The 'Duplicate Identification Constraints' section has three checkboxes: 'Name must match', 'Date 'created' must match', and 'Date 'modified' must match'. The 'Actions' section has a 'Repeat Search' button and a status message 'Displaying all found groups of duplicates (810) !'.

On the right, the results panel shows a list of duplicate groups. Each group has a 'Delete Selected File' button at the top. The first group has a hash 'd99970f5de7b5ae53f55290843e88003' and three entries in a table with columns 'Path' and 'Path'. The second group has a hash '23eeccc7fa2525a19b5d1a50223a2c96' and two entries. The third group has a hash '0a802c3530a61003c9591795c1fb6b9a' and two entries. The fourth group has a hash '898b85e504f5fac6f749077a7a36e725' and two entries. The fifth group has a hash 'ebbb37099940ea5b6de545cdf5ece6e' and two entries.

Hash	Path	Path
d99970f5de7b5ae53f55290843e88003	\dropbox.cache\2017-06-24\	Loesung_SS_2012 DBII_Zwe
	\Kellerkinder\Alte Klausuren\Datenbanken\Alte Klausuren	Loesung_SS_2012 DBII_Zwe
	\Kellerkinder\Semester IV\Datenbanken II\	Loesung_SS_2012 DBII_Zwe

Hash	Path	Path
23eeccc7fa2525a19b5d1a50223a2c96	\Kellerkinder\Alte Klausuren\algodat\göbel\	algorithmendatenstruktur
	\Kellerkinder\Semester IV\Formale Sprachen\Formale Spr	AlgorithmenDatenstrukture

Hash	Path	Path
0a802c3530a61003c9591795c1fb6b9a	\Kellerkinder\Alte Klausuren\algodat\göbel\	AlgorithmenDatenstrukture
	\Kellerkinder\Semester IV\Formale Sprachen\Formale Spr	AlgorithmenDatenstrukture

Hash	Path	Path
898b85e504f5fac6f749077a7a36e725	\Kellerkinder\Alte Klausuren\algodat\göbel\	AlgorithmenDatenstrukture
	\Kellerkinder\Semester IV\Formale Sprachen\Formale Spr	AlgorithmenDatenstrukture

Hash	Path	Path
ebbb37099940ea5b6de545cdf5ece6e	\Kellerkinder\Alte Klausuren\algodat\göbel\	AlgorithmenDatenstrukture
	\Kellerkinder\Semester IV\Formale Sprachen\Formale Spr	AlgorithmenDatenstrukture

4. Discussion

4.1 Positives

The application works and provides useful help for identifying duplicate files, even when a lot of files are involved. Its performance with most criteria methods is good. It gives the user a lot of power about how he wants to find duplicates while keeping the settings without too much clutter.

4.2 Problems

While MVVM worked pretty good after gaining more experience with it, the asynchronous implementation caused a lot of trouble. Initially, the Background searcher task/thread simply dumped all his results into the ViewModel class responsible for displaying the results. This resulted in the UI thread becoming completely unresponsive for minutes (even though it wasn't blocked), probably because of 2 reasons: The amount of interaction between the UI thread and the background thread as well as the amount of binding view needed to display all items in the GUI. The new solution circumvents this problem by sending the results piece by piece from an artificially slowed down background thread. This allows the user to see how the GUI fills with more and more results which feels much faster, while keeping the GUI responsive enough at all times to not freeze.

Another problem was caused by the Fody PropertyChanged plugin, which weaves PropertyChanged code into all properties (and even implements the needed interfaces when instructed). While a workaround exists and is implemented (see SearchVM.cs line 60), the problem itself is still quite mysterious and caused a very hard to debug problem.

4.3 Improvement: Similarity Matcher

The Levenshtein distance algorithm is too slow to be of much use. A better alternative would be a more modern algorithm described in the paper „An O(ND) Difference Algorithm and its Variations“ by Eugene Myers. Implementations of that algorithm exist for C#, the only problem is that all implementations assume textfiles, since they are mostly used for Diff-Tools. It could be possible to simply treat every (binary) file as if it was an ASCII encoded text file, and execute the algorithm on the decoded ASCII string. This should have the same effect as writing a binary implementation, as long as all ASCII bytes are treated equally (no special treatment for whitespace).

4.4 Idea: Automatic Duplicate Cleanup

Since a lot of information is known about the duplicates in the found groups, the user could be given the possibility to sort all the duplicate files (inside each group), for example by file path. This would not sort the groups themselves, just the ordering of a file inside its group. Then, a function that deletes all but the first/last file in each group could be useful, since the user could, before using the deletion function, sort the files in each group in such a way that the file in the correct place „survives“.

This would require careful sort functions though that do not behave unexpectedly.