A Generic, Reusable Diff Algorithm in C# - II

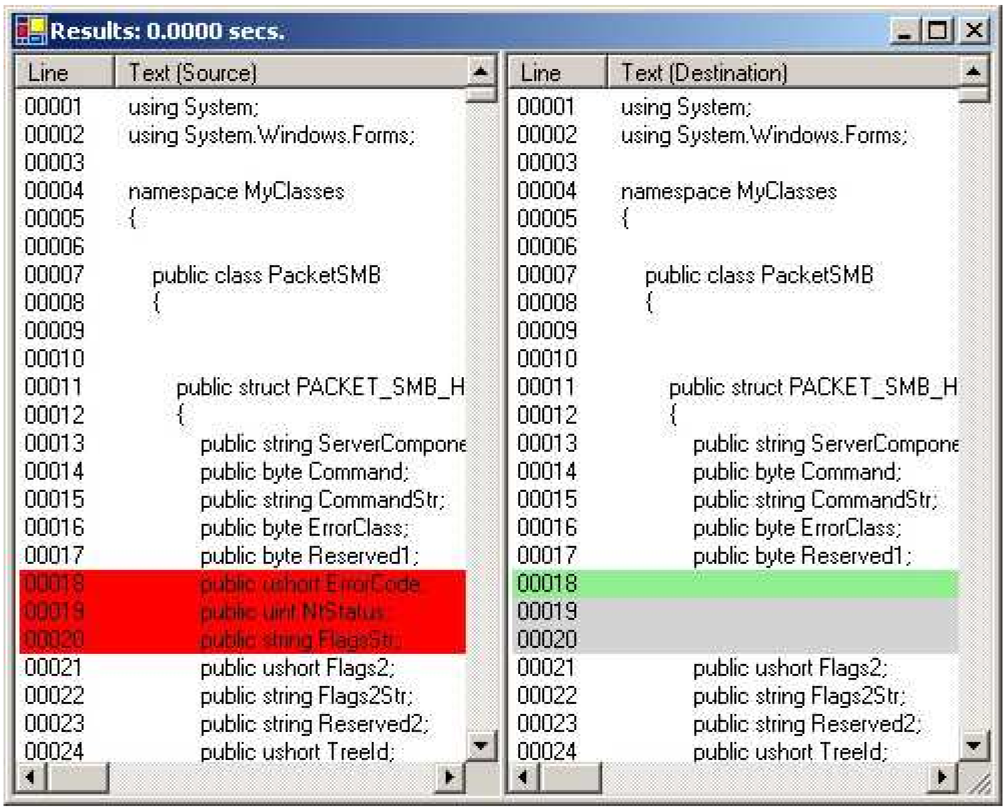
Michael Potter, 10 Jun 2004

★★★★★ 4.87 (124 votes)

A reusable difference engine written in C#.

Download source files - 24 Kb

Download demo project - 14 Kb



```
Results: 0.0000 secs.

Line   Text (Source)                    Line   Text (Destination)
00001  using System;                    00001  using System;
00002  using System.Windows.Forms;      00002  using System.Windows.Forms;
00003                                   00003
00004  namespace MyClasses              00004  namespace MyClasses
00005  {                                00005  {
00006                                   00006
00007      public class PacketSMB       00007      public class PacketSMB
00008      {                            00008      {
00009                                   00009
00010                                   00010
00011          public struct PACKET_SMB_H   00011          public struct PACKET_SMB_H
00012          {                        00012          {
00013              public string ServerCompone   00013              public string ServerCompone
00014              public byte Command;     00014              public byte Command;
00015              public string CommandStr;   00015              public string CommandStr;
00016              public byte ErrorClass;  00016              public byte ErrorClass;
00017              public byte Reserved1;   00017              public byte Reserved1;
00018              public ushort ErrorCode  00018
00019              public uint NtStatus     00019
00020              public string FlagsStr;  00020
00021              public ushort Flags2;    00021              public ushort Flags2;
00022              public string Flags2Str; 00022              public string Flags2Str;
00023              public string Reserved2; 00023              public string Reserved2;
00024              public ushort TreeId;    00024              public ushort TreeId;
```

Introduction

During one of my daily visits to CodeProject, I ran across an excellent article by Aprenot: A Generic - Reusable Diff Algorithm in C#. Aprenot's method of locating the Longest Common Sequence of two sequential sets of objects works extremely well on small sets. As the sets get larger, the algorithm begins experiencing the constraints of reality.

At the heart of the algorithm is a table that stores the comparison results of every item in the first set to every item in the second set. Although this method always produces a perfect difference solution, it can require an enormous amount of CPU and memory. The author solved these issues by only comparing small portions of the two data sets at a time. However, this solution makes the assumption that the changes between the data sets are very close together, and reports inefficient results when the data sets are large with dispersed changes.

This article will present an algorithm based on the one presented by aprenot. The goals of the algorithm are as follows:

1. Maintain the original concepts of Generic and Reusable.
2. Correctly handle large data sets.
3. Greatly lower the number of comparisons necessary.
4. Greatly lower the memory requirements.

The Problem Defined

Given a data set of 100,000 items, with the need to compare it to a data set of similar size, you can quickly see the problem with using a table to hold the results. If each element in the result table was 1 bit wide, you would need over a Gigabyte of memory. To fill the table, you would need to execute 10 billion comparisons.

Some Terminology

There are always two sets of items to compare. To help differentiate between the two sets, they will be called Source and Destination. The question we are usually asking is: What do we need to do to the Source list to make it look like the Destination list?

Generic and Reusable

In order to maintain the generic aspects of the original algorithm, a generic structure is needed. I chose to make use of C#'s `interface`.

```
public interface IDiffList
{
     int Count();
     IComparable GetByIndex(int index);
}
```

Both the Destination list and Source list must inherit `IDiffList`. It is assumed that the list is indexed from 0 to `Count()`-1. Just like the original article, the `IComparable` interface is used to compare the items between the lists.

Included in the source code are two structures that make use of this interface; `DiffList_TextFile` and `DiffList_BinaryFile`. They both know how to load their respective files into memory and return their individual items as `IComparable` structures. The source code examples for these objects should be more than adequate for expanding the system to other object types. For example, the system can be easily expanded to compare rows between `DataSet`s or directory structures between drives.

The Overall Solution

The problem presented earlier is very similar to the differences in sorting algorithms. A shell sort is very quick to code but highly inefficient when the data set gets large. A quick sort is much more efficient on a large dataset. It breaks up the data into very smaller chunks using recursion.

The approach this algorithm takes is similar to that of a quick sort. It breaks up the data into very smaller chunks and processes those smaller chunks through recursion. The steps the algorithm takes are as follows:

1. Find the current Longest Matching Sequence **(LMS)** of items.
2. Store this **LMS** in a results pile.
3. Process all data left above the **LMS** using recursion.
4. Process all data left below the **LMS** using recursion.

These steps recursively repeat until there is no more data to process (or no more matches are found). At first glance, you should be able to easily understand the recursion logic. What needs further explanation is Step 1. How do we find the **LMS** without comparing everything to everything? Since this process is called recursively, won't we end up re-comparing some items?

Finding the Longest Matching Sequence

First, we need to define where to look for the **LMS**. The system needs to maintain some boundaries within the Source and Destination lists. This is done using simple integer indexes called `destStart`, `destEnd`, `sourceStart` and `sourceEnd`. At first, these will encompass the entire bounds of each list. Recursion will shrink their ranges with each call.

To find the **LMS**, we use brute force looping with some intelligent short circuits. I chose to loop through the Destination items and compare them to the available Source items. The pseudo code looks something like this:

```
For each Destination Item in Destination Range
     Jump out if we can not mathematically find a longer match
     Find the LMS for this destination item in the Source Range
     If there is a match sequence
          If it's the longest one so far - store the result.
          Jump over the Destination Items that are included in this sequence.
     End if
Next For

If we have a good best match sequence
     Store the match in a final match result list
     If there is space left above the match in both
     the Destination and Source Ranges
          Recursively call function with the upper ranges
     If there is space left above the match in both
     The Destination and Source Ranges
          Recursively call function with upper ranges
Else
     There are no matches in this range so just drop out
End if
```

The Jumps are what gives this algorithm a lot of its speed. The first mathematical jump looks at the result of some simple math:

```
maxPossibleDestLength = (destEnd - destIndex) + 1;
if (maxPossibleDestLength <= curBestLength) break;
```

This formula calculates the theoretical best possible match the current Destination item can produce. If it is less than (or equal too) the current best match length then there is no reason to continue in the loop through the rest of the Destination range.

The second jump is more of a leap of faith. We ignore overlapping matching sequences by jumping over the destination indexes that are internal to a current match sequence, and therefore, cut way down on the number of comparisons. This is a really big speed enhancement. If the lists we are testing contain a lot of repetitive data, we may not come up with the perfect solution, but we will find a valid solution fast. I found in testing that I had to manually create a set of files to demonstrate the imperfect solution. In practice, it should be very rare. You can comment out this jump in the source code and run your own tests. I have to warn you that it will greatly increase the calculation time on large data sets.

There is a very good chance during recursion that the same Destination Item will need to be tested again. The only difference in the test will be the width of the Source Range. Since the goal of the algorithm is to lower the number of comparisons, we need to store the previous match results. `DiffState` is the structure that stores the result. It contains an index of the first matching item in the source range and a length so that we know how far the match goes for. DiffStateList stores the DiffStates by destination index. The loop simple requests the DiffState for a particular destination index, and DiffStateList either returns a pre-calculated one or a new uncalculated one. There is a simple test performed to see if the DiffState needs to be recalculated given the current Destination and source ranges. DiffState will also store a status of 'No Match' when appropriate.

If a DiffState needs to be recalculated, an algorithm similar to the one above is called.

```
For each Source Item in Source Range
     Jump out if we can not mathematically find a longer match
     Find the match length for the Destination Item
              on the particular Source Index
     If there is a match
          If it’s the longest one so far – store the result
          Jump over the Source Items that are included in the match
     End if
End for

Store the longest sequence or mark as 'No Match'
```

The Jumps are again giving us more speed by avoiding unnecessary item comparisons. I found that the second jump cuts the run time speed by 2/3rds on large data sets. Finding the match length for a particular destination item at a particular source index is just the result of comparing the item lists in sequence at those points and returning the number of sequential matches.

Gathering the Results

After the algorithm has run its course, you will be left with an ArrayList valid match objects. I use an object called DiffResultSpan to store these matches. A quick sort of these objects will put them in the necessary sequential order. DiffResultSpan can also store the necessary delete, addition and replace states within the comparison.

To build the final result ArrayList of ordered DiffResultSpans, we just loop through the matches filling in the blank (unmatched) indexes in between. We return the result as an ordered list of DiffResultSpans that each contains a DiffResultSpanStatus.

```
public enum DiffResultSpanStatus
{
     NoChange, //matched
     Replace,
     DeleteSource,
     AddDestination
}

public class DiffResultSpan : IComparable
{
     private int _destIndex;
     private int _sourceIndex;
     private int _length;
     private DiffResultSpanStatus _status;

     public int DestIndex {get{return _destIndex;}}
     public int SourceIndex {get{return _sourceIndex;}}
     public int Length {get{return _length;}}
     public DiffResultSpanStatus Status {get{return _status;}}

     //.... other code removed for brevity
}
```

You can now process the ArrayList as is necessary for your application.

Algorithm Weaknesses

When using the algorithm described above, and when the data sets are completely different, it will compare every item in both sets before it finds out that there are no matches. This can be a very time consuming process on large datasets. On the other hand, if the data sets are equivalent, it will find this out in one iteration of the main loop.

Although it should always find a valid difference, there is a chance that the algorithm will not find the best answer. I have included a set of text files that demonstrate this weakness (*source.txt* and *dest.txt*). There is a sequence of 5 matches that is missed by the system because it is overlapped by a previous smaller sequence.

Algorithm Update 6/10/2004

To help address the 2nd weakness above, three levels of optimization were added to the Diff Engine. Tests identified that large, highly redundant data produced extremely poor difference results. The following `enum` was added:

```
public enum DiffEngineLevel
{
     FastImperfect, //original level
     Medium,
     SlowPerfect
}
```

This can be passed to an additional `ProcessDiff()` method. The engine is still fully backward compatible and will default to the original Fast method. Only tests can identify if the `Medium` or `SlowPerfect` levels are necessary for your applications. The speed differences between the settings are quite large.

The differences in these levels effect when or if we jump over sections of the data when we find existing match runs. If you are interested, you will find the changes in the `ProcessRange()` method. They start on line 107 of *Engine.cs*.

Included Files

The project *DiffCalc* is the simple front-end used to test the algorithm. It is capable of doing a text or binary *diff* between two files.

The DLL project *DifferenceEngine* is where all the work is done.

- *Engine.cs*: Contains the actual diff engine as described above.
- *Structures.cs*: Contains the structures used by the engine.
- *TextFile.cs*: Contains the structures designed to handle text files.
- *BinaryFile.cs*: Contains the structures designed to handle binary files.

Special Note

The first line in *Structures.cs* is commented out. If you uncomment this line, you will lower the memory needs of the algorithm by using a `HashTable` instead of an allocated array to store the intermediate results. It does slow down the speed by a percent or two. I believe the decrease is due to the reflection that becomes necessary. I am hoping that the future .NET Generics will solve this issue.

Summary

If you take a step back from the algorithm, you will see that it is essentially building the table described in aprenot's original article. It simply attempts to intelligently jump over large portions of the table. In fact, the more the two lists are the same, the quicker the algorithm will run. It also stores what it does calculate in a more efficient structure instead of rows & columns.

Hopefully, others will find good uses for this code. I am sure there are some more optimizations that will become apparent over time. Drop me a note when you find them.

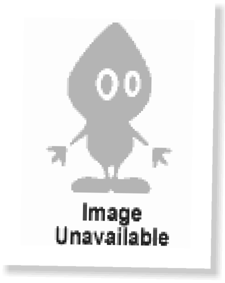
History

- **May 18, 2004** 'Off by One' error on line 106 in Engine.cs fixed. Thanks goes to Lutz Hanusch for identifying the bug.
- **May 27, 2004** Incrementor was missing on line 96 in Results.cs in the demo. Thanks goes to Cash Foley for identifying the bug.
- **June 10, 2004** Speed optimizations led to **very** poor diff results on redundant data. Two slower diff engine levels have been added to solve this problem. Thanks goes to Rick Morris for identifying this weakness.

License

Share

About the Author



Michael Potter

Chief Technology Officer
United States

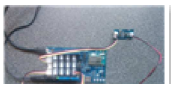
No Biography provided

You may also be interested in...

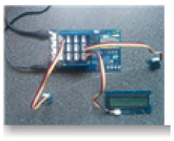
Visual COBOL New Release: Small point. Big deal

SAPrefs - Netscape-like Preferences Dialog

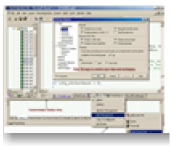
Equipment Activity Monitor in Python



Generate and add keyword variations using AdWords API



Doorbell in Python



Window Tabs (WndTabs) Add-In for DevStudio

Comments and Discussions

128 messages have been posted for this article Visit **https://www.codeproject.com/Articles/6943/A-Generic-Reusable-Diff-Algorithm-in-C-II** to post and view comments on this article, or click **here** to get a print view with messages.