Final Project

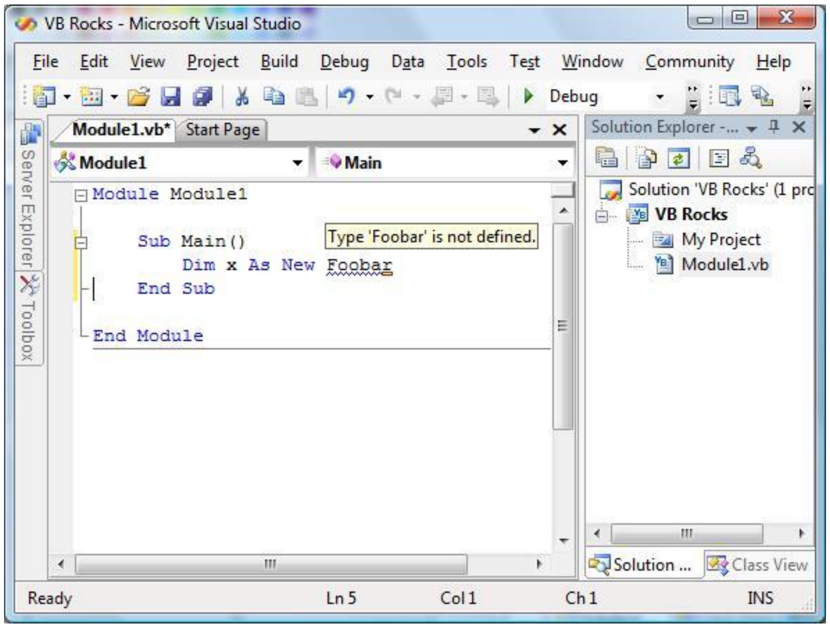
Applied Algorithms

CSEP 521

Chris Smith and Meher Malakapalli

Introduction

In order to provide a better development experience for Visual Studio .Net, a Visual Basic .Net background compiler was created in order to provide real time feedback on syntax errors. Very similar to Microsoft Word as soon as a syntax error is introduced in the program code the background compiler would detect the error and provide feedback to the programmer.



```
Module Module1

    Sub Main()
        Dim x As New Foobar
    End Sub

End Module
```

Type 'Foobar' is not defined.

Figure 1 The VB.Net background compiler in action

One of the challenges in developing a background compiler is performance. Once the project is initially compiled, the naïve solution would be to compile the current file being edited every time a change was made however that would lead to poor performance. An optimization was implemented such that context was taken into account for changes so rather than the entire file being recompiled only the method or class body would be updated based on the context of the change.

The Problem

While the context sensitive recompilation sped up editing times considerably, it performs poorly against generated code. (One of the hallmarks of the Visual Studio .Net development experience.) Mundane code such as typed datases, UI control layout, settings and resource accessors, etc. are all generated by custom tools. The problem however, is that the entire code file is replaced. So without a way to diff to the two files, there is no way to determine the locale of the changes and thus are forced to always recompile the entire file.

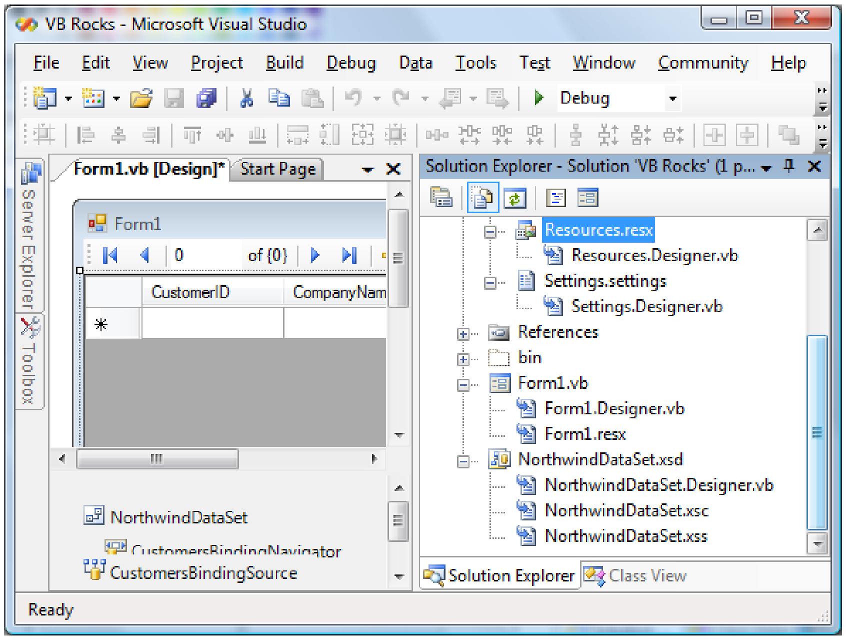


Figure 2 Generated code in a typical VB project

Formal Definition

Given two files locate the set of changes between them in order to target the recompilation needed for the code updates. In academic research this is known as the *longest common subsequence* or LCS. More formally, given two input strings X and Y find the string S of maximal length such that it is a subsequence of both X and Y.

If you consider lines in two files to be characters in the problem domain's alphabet, then from the LCS you can determine where the files are different by identifying characters which are not in the LCS.

The Least Common Subsequence problem

There has been much academic research on the LCS problem, with the basic solution being the following dynamic programming recurrence relation:

$$LCS(X_{1..i}, Y_{1..j}) = \begin{cases} 0 & if\ i = 0\ or\ j = 0 \\ LCS(X_{1..i-1}, Y_{1..j-1}) + x_i & if\ x_i = y_i \\ Max\left(LCS(X_{1..i}, Y_{1..j-1}), LCS(X_{1..i-1}, Y_{1..j})\right) & otherwise \end{cases}$$

Note that the baseline algorithm requires $O(nm)$ time and space, with n and m corresponding to the lengths of the input strings X and Y. Also, the function LCS only returns the maximal length of the longest common subsequence; however by simply backtracking in our resulting matrix we can find the actual characters corresponding to that sequence.

Improvements

A good summary of present research in LCS is presented in a paper by Bergroth et al[1] and the basis for most optimizations expands upon work by Hunt and Szymanski [2]. We will go over their improvement to the initial dynamic programming solution here.

For simplicity, assume the length of both strings is n. We will also introduce a key data structure which is an array of threshold values $T_{i,k}$ given by the following, with CS meaning common subsequence:

$T_{i,k}(0 \leq i, k \leq n) = Min\ (\ j\ \mid\ X[1..i]$ and $Y[1..j]$ contains a CS of length $k)$

For example, given the strings X = “abcbdda” and Y=”badbabd”, we get the following:

$T_{5,1} = 1$	$T_{5,2} = 3$	$T_{5,3} = 6$
$T_{5,4} = 7$	$T_{5,5} = undefined$	

Once the array of threshold values $T_{i,k}$ has been completely filled, the length of LCS(X,Y) is simply $Max($ k over $T_{n,k}$ $)$. Again, the actual string can be completed using a backtracking mechanism.

Lemma for all values $i > 1, k < n$

$T_{i,k} = Min(\ j \mid X[i] = Y[j]$ and $T_{i-1,k-1} < j \leq T_{i-1,k})$ OR $T_{i-1,k}$ if no such j exists

The above Lemma immediately gives us a linear space approach for filling the threshold T, since we can overwrite i^{th} row to get the $(i+1)^{\text{th}}$ row. (Opposed to keeping the full $n \times n$ matrix.)

The next step s is to search for character indices in Y that match $X[i]$. For this, we can have a list $Match[1..n] =< j_1, j_2, \dots, j_n >$ where $j_1 > j_2 > \cdots > j_n$. This is easy to do by sorting X and Y while keeping track of each character's original index position. Then, while merging, we can easily create the match list.

Once we have the match list, we just compute the threshold value $T_{i,k}$ using the lemma above. The critical part of this is to find a k that satisfies the lemma. For this, we can simply use a binary search on k, instead of trying all $k = 1..n$ possibilities. The largest k among the threshold values then gives us the LCS.

The time complexity of this algorithm then is $O(|M| \log n)$ where $|M|$ denotes the number of all pairs that have a match. In other words, $M = r + n$ where r is the total number of elements found for $Match[1 \dots n]$.

Other Improvements

In fact, it is possible to improve the LCS algorithm even further beyond $O(|M| \log n)$ so that on average the algorithm behaves in accordance to different parameters. The following table summarizes current LCS algorithms:

Algorithm	Time Complexity
Hunt and Szymanski [2]	$O(\lvert M\rvert log\ n)$
Hsu and Du [4]	$O(rm\ log(n/r)\ +\ rm)$
Kuo and Cross [3]	$O(\lvert M\rvert\ +\ n(r\ +\ log\ n))$
Hirschberg [5]	$O(rn\ +\ n\ log\ n)$
Nakatsu [7]	$O(n(m-r))$
Miller and Myers [6]	$O(m(m-r))$

Figure 3 LCS algorithm performance

On whole the runtime of these algorithms are typically very good but can vary drastically when provided a wide range of input. Though for most large and non-uniform strings, *Kuo and Cross* performs the best [1].

The Solution

While there are many great text diffing algorithms, the one to be used for the next release of Visual Studio will take the following structure:

```
Given two files F and F', define F[i] to be the $i^{th}$ line in F

# Find the index of the first mismatch top down
for i = 1 to |F'|
   if F[i] = F'[i]  then continue
   if F[i] <> F'[i] then break
next

# Files are identical
if i = |F'| then return TextSpan(1,0)

# Find the index of the first mismatch bottom up
# Redefine F[j] to mean the $j^{th}$ character up from the bottom line
for j = |F'| to i
   if F[j] = F'[j]  then contine
   if F[j] <> F'[j] then break
next

if (j - i) > k then
   # Compile the entire file, they are too different
   return Textspan(1, |F'|)
else
   # The change occurred between i and j
   return Textspan(i, j)
end if
```

The first thing you will notice is that the algorithm returns at most one list of contiguous changes between the two files. And, it will only return the LCS IFF the changes are contiguous and less than some constant k.

The algorithm loops through each line in F' at most twice, and thus executes on order $O(n)$.

Results

While there are many possible LCS algorithms to choose from for diffing generated code, the next release of Visual Studio will go with a simpler approach. The main reason is to exploit an aspect of the generated files. Generated code has a well defined structure and when there is a change it is typically adding a new property or function which would be a single contiguous block of code.

While a more complex algorithm would be able to isolate the exact line changes and be able to better target the recompilation required, in many cases it is faster to simply recompile an entire file than to update the compiler’s internal data structures for specific code elements and their dependencies.

Currently it is unknown what sort of performance impact this will have out in the wild, but so far this algorithm, though primitive, has helped improve design time performance for Visual Basic application development.

References

1. L. Bergroth, H. Hakonen, and T. Raita. *A survey of longest common subsequence algorithms*. Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00), IEEE Computer Society, pp. 39-48, 2000.

2. J. W. Hunt and T. G. Szymanski. *A fast algorithm for computing longest common subsequences*. Comm. of the ACM, Vol. 20, No.5, pp. 350-353, 1977.

3. Kuo, S. and Cross, G.R. *An Improved Algorithm to Find the Length of the Longest Common Subsequence of Two Strings*. ACM SIGIR Forum, Vol. 23, No. 3-4 , pp.89-99, 1989.

4. Hsu, W.J. and Du, M.W. *New Algorithms for the LCS Problem*. Journal of Comp. and System Sc., Vol. 29, pp. 133-152, 1984.

5. Hirschberg, D.S. *Algorithms for the Longest Common Subsequence Problem*. Journal of the ACM, Vol. 24, No. 4, pp. 664-675, 1977. A Linear Space Algorithm for Computing Maximal Common Subsequences. Comm. of the ACM, Vol. 18, No. 6, pp. 341-343, 1975.

6. Miller, W. and Myers, E.W. *A File Comparison Program*. Software Practice and Experience, Vol. 15, No. 11, pp.1025-1040, 1985.

7. Nakatsu, N., Kambayashi, Y. and Yajima, S. *A Longest Common Subsequence Algorithm Suitable for Similar Texts, Acta Informatica*. Vol. 18, pp. 171-179, 1982.