CEditDist Abstract Template Class for Edit Distance Calculation on Generic Data Types

Zvika Ben-Haim
zvikabh@aluf.technion.ac.il
http://www.technion.ac.il/~zvikabh/software
June 1999

Purpose

The `CEditDist` class performs edit distance calculations on abstract data types. The *edit distance* is defined as the minimum cost required to convert one string into another, where the conversion can include the following three operations:

- changing one character to another
- deleting one character
- inserting one character

This algorithm is useful for finding the degree of similarily between strings, when character omissions and duplications may occur. The term 'character' is used loosely here, since the template mechanism allows you to implement the class for any data type, including user-defined types. The *cost* of each of these operations is an integer defined by the user and can depend on the characters being changed.

The operation performs in $O(nm)$ space and time, where n and m are the lengths of the two compared strings. Specifically, an array of n by m integers is dynamically allocated for the operation, and deallocated automatically.

System Requirements

`CEditDist` was tested with Visual C++ 5.0, but contains only standard C++ commands, and thus should functions correctly under other standard C++ compilers as well.

Please note that under Visual C++ 5.0, an apparent bug causes warning messages when explicitly instantiating the base class. These warning messages can be safely ignored.

Overview

`CEditDist` is an abstract template class. To define a working edit-distance calculation class, you will first instantiate a `CEditDist` class with a data type equal to the type of 'character' you will be working with. This is the single unit of the two strings which are to be compared. Thus, if you want to compare two arrays of `char`, you will instantiate `CEditDist<char>`.

Next, you will define a derivation of this class, e.g.

```
class CMyEditDist : public CEditDist<char> {...};
```

This class will contain three virtual member functions: `DeleteCost`, `InsertCost` and `ChangeCost`. These functions define the costs for single edit operations.

Finally, to use the class, you will define an object of type `CMyEditDist`, and call its inherited member function `EditDistance`. The return value is the total cost of the least expensive path.

Instantiating a Class

You must add an explicit instantiaion in the EditDist.cpp file. This ensures that all member functions are instantiated, even if they are not called from within EditDist.cpp. In the last few lines of the file, add the statement `template class CEditDist<CMyType>;`

This statement may generate warnings 4660 and 4661, which can be safely ignored. `CMyType` should, of course, be whatever type you want to instantiate. `CMyType` can be any internal or user-defined data type which supports the following:

- Default constructor available
- `CMyType::operator=(const CMyType&)` defined

Deriving from CEditDist<CMyType>

Deriving is straightforward. Just remember to derive from the instantiated class, i.e. use the format

```
class CMyEditDist : public CEditDist<CMyType> {...};
```

After deriving, you need to define the following three virtual member functions, which are abstract functions in the base class.

All of these functions receive the parameters `x` and `y`, which specify the position in the source and destination strings, respectively. These can be used for additional flexibility in determining costs.

The functions can also use the protected data members `m_xmax` and `m_ymax`. (These variables may not have valid data when not calling cost functions.) These values specify the maximum index in the source and destination strings, respectively. In other words, they are equal to one less than the length of the source and destination strings.

These members can be used, for example, to create cost functions in which the insertion cost is 0 if we have already reached the end of the source string. This is useful when we are searching for the beginning of a string, and do not care if the destination continues past the end of the source string.

- `int DeleteCost(const CMyType& deleted, int x, int y);`
  This function returns the cost for deleting the item `deleted`.
- `int InsertCost(const CMyType& inserted, int x, int y);`
  This function returns the cost for inserting the item `inserted`.
- `int ChangeCost(const CMyType& from, const CMyType& to, int x, int y);`
  This function returns the cost for changing the item `from` to the item `to`.

Do not override the inherited function `EditDistance`.

Using the Class

To use the class, define an object of type CMyEditDist. Then, call the object's EditDistance function.

```
int EditDistance(CMyType* ar1, int len1, CMyType* ar2, int len2);
```

- `ar1`: First array of items
- `len1`: Length of ar1
- `ar2`: Second array of items
- `len2`: Length of ar2
- Return value: Minimal total cost for converting `ar1` into `ar2`

Example: Integer Array Edit Distance

The following example shows a simple imlpementation of the edit distance function in which the cost is 5 for deletions and insertions, 3 for changes between different integers and 0 for changes of identical integers.

The class `CIntEditDist` is defined as follows:

```
class CIntEditDist : public CEditDist<int>
{
public:
	int DeleteCost(const int& deleted, int x, int y) {return 5;};
	int InsertCost(const int& inserted, int x, int y) {return 5;};
	int ChangeCost(const int& from, const int& to, int x, int y)
		{ return (from==to?0:3);};
};
```

The class is instantiated using the following declaration:

```
template class CEditDist<int>;
```

To test the functionality of this class, use the following code:

```
	CIntEditDist ed;
	int a[] = {1,1,2,3,1,2,2};
	int b[] = {1,2,2,3,1,1};
	int c = ed.EditDistance(a,7,b,6);
	CString str;
	str.Format("Distance: %d",c);
	AfxMessageBox(str);
```

The distance displayed should be 11. The least expensive edit replaces the second character in a from 1 to 2, replaces the sixth character in a from 2 to 1, and deletes the last character in a, for a total of two changes and one deletion, with a cost of 3+3+5=11.