Programmer's Manual

By: Boris Pismenny

This part is meant to be read by someone who intends to do some changes/improvements to the source code. For start I would suggest you read the article:

"An Optimal Decomposition Algorithm for Tree Edit Distance". Erik Demaine, Shay Mozes, Benjamin Rossman , Oren Weimann. *ACM Transactions on Algorithms*

Especially parts 1,2,3,5. (you may skip parts about complexity analysis)

Note: You should not skip parts about the Decomposition Strategy – part 2.3

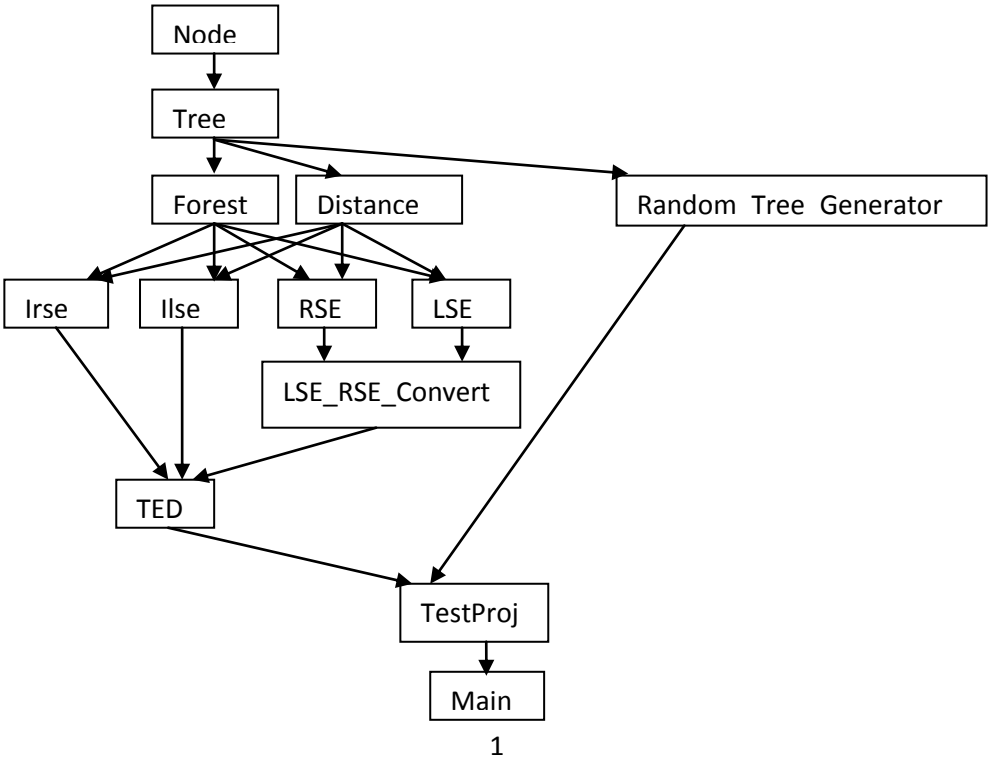
Now that you have read the article we speak in the same language. To understand the program fully you need to understand 2 things: Algorithms and Accessories. I will explain each algorithm in turn, going in the following order: Shasha & Zhang , Klein, DMRW (the one that turns general trees to binary trees, and also the one that deals with general trees directly).Before these I will explain the accessories used to make it all work. But I will start with the compilation instructions and Windows vs non-Windows versions

compilation instructions and Windows vs non-Windows

First I will explain the make file and the dependencies after wards I will compare the main from the windows version with the main from the non-windows version.

make.txt:

For explaining the dependencies I will use a dependency graph:



Windows vs. Non-Windows Versions

The first thing you notice is that the windows version has GUI while the non-windows version does not. If you would like to add GUI for other OS other than windows change the main program and add the resources you need. You could look at my main from the windows version, all that it does is add the GUI functionality to the program.

Windows Main.cpp:

The main function creates a windows whose procedure handles 3 operations RUN, TEST, ABOUT. The RUN and TEST operations create a dialog box each with a specific functionality. Then there is a function that loads files from a directory, a function that saves into a directory, and last a function to run TED. This is all well documented so just look at the source and there should not be a problem.

Non-Windows Main.cpp:

This a console application thus the main is quite simple it presents a series of questions the requite y/n answers and the number of test cases to run. Afterwards it just used TestProj.h to run the test and save the results in the required directory.

Accessories

In this part I will describe the classes which are included with almost every algorithm. They are: Node.h Tree.h , Forest.h , Ilse.h/Irse.h , Lse.h/Rse.h , LSE_RSE_Convert , Distance.h , Random_Tree_Generator.h , TestProj.h

Node.h:

This class describes all the information that is included in a tree node not including children. (Soon it will be clearer)

The parameters are:

```
bool heavy; // is in heavy path
int index;  // index in the Post Order traversal of the tree
```

This index is used mainly to identify each node as a tree root. We implement this transformation using the GetTree[][] array(which is going to be explained in Tree.h). Also we use this index in the delta[][] array(delta like in the article about DMRW) again to index each node as a rootless tree. In Shasha & Zhang and also in Klein it is an index to the dynamic programming table.

```
int LTRP;   // Left-To-Right Preorder traversal value
int RTLP;   // Right-To-Left Preorder traversal value
```

Both of this parameters are needed to convert between the left sub forest enumeration to the right sub forest enumeration and backwards.

```
int DC;     // Delete-Cost of the Fk forest which is represented by that node
```

This parameter is needed to save time when DMRW needs to know the cost of deleting some Fk forest(it is called NTDF in projects that don't include the Distance class).

Note: the case of a general tree will be discussed when we get to the algorithms part.

```
int DT;     //cost to delete the tree rooted at this node
```
This parameter is used in Rse.h and Lse.h in Ilse.h and Irse.h and numerous times in the DMRW algorithm.

The methods of this class are all trivial Get/Set.

Tree.h:

This class describes a tree node

The Parameters are:

```
int Size;               // Size of sub tree (0 = leave) without the root
Node* data;             // Node data
deque<Tree*> children;  //all of trees children
Tree* heavyChild;       // pointer to the heavy child
static vector<Tree*> GetTree[2];
//a mapping between the nodes indexes in the postorder
//left to right and the roots of the trees(0 stands for an empty tree)
```

GetTree[0] is a mapping between postorder left to right indexes of nodes in the F tree and GetTree[1] is a symmetrical for the G tree. It is static because there is only one array for each tree, not for each tree node. It is used mainly to update delta in DMRW.

The methods are:

```
Tree();
Tree(Node* data1);
~Tree();
void AddLeftTree(Tree* T);
void AddRightTree(Tree* T);
int getSize() const {return Size;}
int getChildrenCount() const {return children.size();}
Node* getData() const {return data;};
Tree* getLeft() const;
Tree* getRight() const;
Tree* getHeavy()const {return heavyChild;}
Tree* getChild(int i) const {return children.at(i);}

void ClearHeavyPath();  //mark the heavy field as false for the heavy path
void FindHeavyPath();   //mark the heaviest nodes & fill HeavyPath[n] &
TopLight[n]
void SelectRightHeavyPath();//marks the rightmost path as heavy
int Postorder();        //postorder traversal which updates the Size parameter
void PostOrderUpdateGetTree(int treeIndex,bool tempCount=true);//this
function is used to fill GetTree[treeIndex]


void Save(string filename) const;
void SaveRec(ofstream& myfile) const;
```

About Load/Save all I can say is that the format of a tree to be loaded and the format of a tree to be saved is (root-label(child1)(child2)(child3)…)

Forest.h:

This class describes a forest which is a deque of trees.
The parameters are:

```
deque<Tree*> Frst;//front=left back=right
```

The methods are:

```
Forest() {iter = Frst.begin();};
Forest(const Forest& Fcopy);  // c'ctor
~Forest() {Frst.~deque();};
void AddRightTree(Tree* T);
Tree* GetRightTree() const;
Forest GetRightRootlessTree() const;
void RemoveRightTree();
void RemoveRightRoot();
void AddLeftTree(Tree* T);
Tree* GetLeftTree() const;
Forest GetLeftRootlessTree() const;
void RemoveLeftTree();
void RemoveLeftRoot();
int ForestCount() const {return Frst.size();}; // number of roots in the
Forest
int ForestSize() const; //how many nodes are in the forest?
Tree* getTree (int i) const {return Frst[i];};
```

Ilse.h/Irse.h:

This class describes the intermediate left/right sub forest enumeration. It is implemented as described in the article. Also it updates the DC parameter of each node. Both parameters and methods are self explanatory after reading the article.

Drawback: The only non-trivial part which is not described in the article is how is the intermediate left/right sub forest enumerations are accomplished when we use general trees and not binary. Well it stays the same except the DC part which is going to be described in the Algorithms setion.

Lse.h/Rse.h:

This class describes the left/right sub forest enumeration. It is implemented as described in the article.

The parameters are: (the same as in Rse)

```
Tree* inTree;//the initial G trees
Forest W;//parameter that is used to update the Table array
vector<Tree*>* Table;//Table[i][j] - the leftmost node of the Gij Forest
vector<int>* ForestSize;//ForestSize[i][j] - the cost of deleting the Gij
forest
int currSize;//parameter that is used to update the ForestSize array
Distance* distance;
```

The methods are: (the methods for Rse are symmetrical)

Apart from the basic Left Sub forest enumeration methods which are implemented straight forward as described in the article, there is the Convert method which was not described in the article, and I will describe it now. Convert is basically an operation which we need to do every time we change direction on the same heavy path between ComputePeriods. As you remember the table T described in the article is indexed by I and j from Gij, but Gij is different for RSE and

for LSE. Take for example the full binary tree consisted of 7 nodes, G3,1 in LSE is different from G3,1 in RSE. So to transform T[i][j] from LSE to RSE indexes we need to save at each node the values RTLP and LTRP (for reference see Node.h). So for Gij we look at the left most root and check its LTRP it is equal to the new-I value. Now all we have to do is calculate the new-j value according to the formula new-j = i + j – new-I. The transformation from RSE to LSE is symmetrical just replace right with left and left with right.

```
Lse(Tree* G, Distance* distance1);
~Lse();
void Right_To_Left_Preorder_Traversal(Tree* G, bool restart);//sets the value
of RTLP field of each node in G
void find_Lse(Tree* G);//computes the LSE of the tree G
void DeleteRightVertex(int i,Tree* G);//updates W for i right deletions
void DeleteLeftVertex(int i,int j);//updates the Ttable for j left deletions
from W after i right deletions have been made
Tree* GetT(){return inTree;}

Tree* GetForest(int i,int j);//return the left most node of Gij
int get_JofI(int i);
int getForestSize(int i,int j);//return the delete cost of the forest
Gij
```

LSE_RSE_Convert.h:

This class describes the conversion of LSE class to RSE class and vice versa. This is very important, because every time we change the direction of our strategy in the algorithm the Table T needs to be converted to the appropriate format.

```
void ConvertToRse(vector<int>* T , Lse &lse , Rse &rse) ;//converts the Table T from LSE to RSE

void ConvertToLse(vector<int>* T , Lse &lse , Rse& rse);//converts the Table T from RSE to LSE
```

Distance.h:

This class describes the distance metric which could be easily adapted to your needs.

The data types defined here are:

```
enum DistanceType
```

is the distance metric type currently only NORMAL is defined but if you want to add a general distance metric say match everithing with ‘A’ costs 100 then you should add a DistanceType.(see example at the users manual)

```
struct Match

struct Delete
```

Both are used to define special cases of delete and match which the user may define.

The parameters are:

```
DistanceType Dtype;
list<Match> MatchList;//user defined special match cases
list<Delete> DeleteList;//user defined special delete cases
```

The methods are:

```
void Load(string filename);
void Clear();//clear all user defined special cases
void SetDtype(DistanceType type);
bool IsEmpty() const;//are there none user defined special cases?
int DeleteNode(Tree* t1);
int DeleteTree(Tree* t1);//delete the tree rooted in t1
int MatchNodes(Tree* t1, Tree* t2);
void UpdateTreeDeleteCosts(Tree* T);//update the DT value of all nodes
in T
```

Random Tree Generator.h:

This class was built to answer the need for random general trees to test the algorithm and to compare between all TEDs versions. The method of creating a random tree is the following:

Create a new node with a random label.

If there is no root the define new node as root

Else choose a random node and connect the new node as a child of the chosen node.

The parameters are:

```
vector<Tree*> TreeVector;//every node is connected to some integer.
//it is from here we choose a random node to connect a new child to
Tree* Root;//the root of our randomly chosen node
int rangeLimit;//every random tree has at most rangeLimit nodes
```

The methods are:

```
//create a random tree with at most rangeLimit nodes
//labels are numbers so they start at 0 and the top limit is 32767
RTG(int rangeLimit1 = 100);
~RTG() {TreeVector.~vector(); }
void Generate();//generates a new random tree.
void Clear();//clear the previous random tree.
Tree* getRoot();//get the root of the current tree.
//int to string
string ItoS(int num);
```

TestProj.h:

This class is meant to carry tests on the algorithms and to compare them with each other.

Note: The trees created and the trees to be loaded are saved in filenames and are numbered, “TreeF0.txt” , “TreeG0.txt” , “TreeF1.txt” , “TreeG1.txt” , etc.

The parameters are:

```
ofstream result;// results from the run of TED should be all equal.
ofstream rtime;// the time each algorithm ran
string path; // a directory path to the test directory
bool ChooseAlgorithm[3];//indecates which algorithms to use
```

This last parameter exists only in the Final TED it allows to choose if to test all algorithms or just some of them.

The methods are:

```
TestProj(string results , bool DMRW , bool Klein , bool Shasha_Zhang);
//constructor - results is the file where to save all results and the path
//to which all other thing will be saved (for example the randomly generated
trees).
//All other parameters indicate which algorithms to test
void TestProjCreate(int rangeLimit, Distance* distance);
//Create a test which will run rangeLimit ranfom trees using the distance
metric
void TestProjLoad(int amount, Distance* distance);
//Loads amount test trees to be run.
//IMPORTENT: all trees should be labeld
//"Test0F.txt" , Test0G.txt", "Test1F.txt" , Test1G.txt", etc.
```

Summery

All the classes covered are presented as they are in their final version in project "TED Final". Other projects may contain some or all of those classes in less complete. Although they are all well documented and are similar in many ways to the ones I described here, I suggest you use the ones in the "TED Final" project.

Algorithms

Shasha & Zhang

This algorithm has a binary version which has been created as a base for the not binary version. We will discuss the not binary version.

The algorithm starts by computing a four dimensional table called distance[i][j][k][l]. This is our dynamic programming table. We use GetTree[][] defined at Tree.h as indexes to the table. The indexes are i – The left most index in F, j – the right ost index in F, k – the left most index in G, l – the right most index in G.

TED() – as a constructor it computes the Size parameter of each tree and also it sets the GetTree[][] array. It defines the distance[][][][] table and updates all of its values to indicate that all the cells are empty.

TED_pro(Forest ForestF, Forest ForestG) – computes TED use the dynamic programming method: (distance is equal to δ here)

$\delta(\emptyset, \emptyset) = 0$

$\delta(F, \emptyset) = \delta(F - r_F, \emptyset) + c_{delete}(r_F)$

$\delta(\emptyset, G) = \delta(\emptyset, G - r_G) + c_{delete}(r_G)$

$$\delta(F,G) = min \begin{cases} \delta(F - r_F, G) + c_{delete}(r_F) \\ \delta(\emptyset, G - r_G) + c_{delete}(r_G) \\ c_{matc\,h}(r_F, r_G) + \delta(R_F^{\circ}, R_G^{\circ}) + \delta(F - R_F, G - R_G) \end{cases}$$

Where:

- F and G are both Forests
- r_G and r_F are the right nodes of F and G accordingly
- c_{delete} is the cost to delete a node and $c_{matc\,h}(r_F, r_G)$ is the cost of matching
- R_F and R_G are the right trees of F and G accordingly
- R_F° and R_G° are the right rootless trees of F and G accordingly

Klein

This algorithm has a binary version which has been created as a base for the not binary version. We will discuss the not binary version. Klein is very similar to Shasha & Zhang.

TED() – as a constructor it computes the Size parameter of each tree, afterwards it decides which tree is bigger and sets it to be F. Then it sets the GetTree[][] array. It defines the distance[][][][] table and updates all of its values to indicate that all the cells are empty.

TED_pro(Forest ForestF, Forest ForestG): computes TED use the dynamic programming method with a little twist: (distance is equal to δ here)

$\delta(\emptyset, \emptyset) = 0$

$\delta(F, \emptyset) = \delta(F - r_F, \emptyset) + c_{delete}(r_F)$

$\delta(\emptyset, G) = \delta(\emptyset, G - r_G) + c_{delete}(r_G)$

if($R_F \leq L_F$) then use:

$$\delta(F,G) = min \begin{cases} \delta(F - r_F, G) + c_{delete}(r_F) \\ \delta(\emptyset, G - r_G) + c_{delete}(r_G) \\ c_{matc\,h}(r_F, r_G) + \delta(R_F^{\circ}, R_G^{\circ}) + \delta(F - R_F, G - R_G) \end{cases}$$

else use:

$$\delta(F,G) = min \begin{cases} \delta(F - l_F, G) + c_{delete}(l_F) \\ \delta(\emptyset, G - l_G) + c_{delete}(l_G) \\ c_{matc\,h}(l_F, l_G) + \delta(L_F^{\circ}, L_G^{\circ}) + \delta(F - L_F, G - L_G) \end{cases}$$

Where:

- F and G are both Forests
- r_G and r_F are the right nodes of F and G accordingly
- l_F and l_G are the left nodes of F and G accordingly
- c_{delete} is the cost to delete a node and $c_{matc\,h}(r_F, r_G)$ is the cost of matching

- R_F and R_G are the right trees of F and G accordingly
- L_F and L_G are the left tress of F and G accordingly
- R_F° and R_G° are the right rootless trees of F and G accordingly
- L_F° and L_G° are the left rootless trees of F and G accordingly

DMRW

DMRW's implementation is a little more complex we will start with the binary version then discuss the not binary version the problems it creates and how I solved them. It is very important to notice that the this program is written much like in the article, many similar names for variables and the order of things done is **exactly** the same.

TED() – as a constructor it updates the size of each tree, computes the heavy path and updates the GetTree[][]array. It creates the delta[][] array (equal to Δ from the article) and computes it for all trivial cases – one or both of the trees are leafs then it calls TED_pro.

TED_pro(Tree* F1, Tree* G1, bool reversed) – if G1 is bigger than F1 then it runs TED(G1,F1,!reversed). (reversed tells if the trees are in reversed order from the original)

Then it runs TED_pro on every tree in TopLight(F1) with G1. Finally it runs ComputePeriodMain.

ComputePeriodMain(Tree* F1, Tree* G1, bool reversed) – this function initializes the needed parameters for ComptePeriod. Here we create T and initialize it for its first run, also it computes the LSE and RSE of G1. Finally we call ComputePeriodRec with the strategy we pick for the roots.

ComputePeriodRec(Tree* F1, Tree* G1, bool reversed, vector<int> *T, Strategy dir, Lse& lse, Rse& rse) – Here we go from top to bottom deciding the directions for each node on the heavy path. Afterwards we climb up the recursion from the bottom to the top and compute ComputePeriod.

ComputePeriod(Tree* Fvp, Tree* G1, bool reversed, vector<int> *T, Strategy dir, Lse& lse, Rse& rse) – now compute period is exactly the same as in the article besides a few extreme cases which were not discussed in the article:

1) if Fvp (the current root on the heavy path) has only one child the heavy child then Δ between the rootless Fvp and the rootless G1 is not computed. But it was already computed in T[1][0]. So just update Δ before running the dynamic program.

2) When we compute arg3 of the dynamic program in the third part $\delta(F - L_F, G - L_G)$ when j≠j(i). The basic idea is that we need to check if F_k is a tree and especially if it is Fvp. When it is F_{V_p} then compute the cost of $\delta(\emptyset, G - L_G)$. If it isn't F_{V_p} but still F_k is a tree, and we try to delete the leftmost tree of F_k from F_k getting F_0 (the previous node in the heavy path) for which we have the distance from $G - L_G$. Also if j=j(i) then we need still need to check that Fk is not F_{V_p} and if it is so then we get $\delta(\emptyset, \emptyset)$.

3) When updating Q and delta check if i == 0 because if it is so then j(0-1) is not defined.

4) When updating delta check if i+j(i-1) is even legal there are cases where i+j(i-1) is bigger or equal then the size of G.

DMRW not binary

When approaching the not binary case there are some issues that you need to cover, but before going to these there is a simple solution - turn all general trees to binary the method for doing it is defined at the article.

Now let's approach the general tree case. The main problem is caused by the need for a direction in each ComputePeriod when in a binary tree the direction is only one in a general tree there might be two strategies. How do we overcome this?

Well first of all we add to ComputePeriodRec the capability to identify the case when there are 2 strategies I call this strategy BOTH(both directions). Now when the BOTH strategy is defined the way to handle this case is to do a LEFT strategy on all the nodes left of the heavy path and then we do a RIGHT strategy. So why is it a problem? Well because what I just described doesn't yet solve the problem. When we do the LEFT strategy we should not update T for Fvp because that would be wrong thinking F_{V_p} has only left nodes of its heavy path. It also denies us the possibility of running the RIGHT strategy, because T will be deprecated. So instead we compute T for $F_{V_{p-1}}$ instead of F_{V_p} now after the LEFT strategy we have the distance between all that is left from the heavy path to every legal forest in G and the RIGHT strategy will compute correctly.

Now how do we do this programmatically:

1)We don't add a BOTH case to compute period.

2)We add a boolean parameter to ComputePeriod called FirstSideOf2 telling us if we are running a BOTH strategy for the first side – the left side, and if that is so then we act as mentioned above.

3)The parameter DC/NTDF (nodes to delete forest Fk which is defined for algorithms which didn't have the distance class), is defined to include only the cost of deleting nodes left of the heavy path when using ILSE, and to include all nodes when using IRSE.

TED Final

In the final version of the project I included all 3 algorithms(DMRW, Klein and Shasha&Zhang) to run in O(n^2) and there is an interface that allows to run one of the three on 2 trees or to run a test on a set of tree pairs chosen by the user. Finally there is an option to define a customizable distance metric.

The basic idea of implementing Shasha&Zhang using the method used before is to define the heavy path as the left most path and not to reverse the trees when F1 is smaller than G1.

To implement Klein we just don't reverse the trees when F1 is smaller than G1.

We add an enum called Algorithm which indicates the algorithm. Most changes are done to the constructor TED_pro and ComputePeriodMain.

TED() – we add a check for Klein to see if the original F tree is smaller than the original G tree and to swap them if it is so. Also we compute here the startLse and startRse parameters which are the LSE and RSE used by Klein and Shasha&Zhang during the whole run. (those two exist only in TED Final, they are meant to improve the run time of Klein and Shasha&Zhang).

TED_pro(Tree* F1, Tree* G1, bool reversed) – we check some parts that which one algorithm or more don't use. Also for Shasha&Zhang instead of using FindHeavyPath() we use SelectRightHeavyPath()

ComputePeriodMain(Tree* F1, Tree* G1, bool reversed) – define lse and rse to be startLse and startRse accordingly if the algorithm is Klein or Shasha&Zhang.

All the others are the same

Improvements to run time:

1)use startLse/startRse

2)save the delete cost of Gij for LSE/RSE

3)use DT to calculate 2 and at many other places