Serdica J. Computing **7** (2013), No 2, 115–134

Serdica
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

PROGRAMMING AND TESTING A TWO-TREE ALGORITHM

Tzvetalin S. Vassilev, Joanna Ammerlaan

Abstract. Recently, Markov, Vassilev and Manev [2] proposed an algorithm for finding the longest path in 2-trees. In this paper, we describe an implementation of the algorithm. We briefly discuss the algorithm and present example that helps the reader grasp the main algorithmic ideas. Further, we discuss the important stages in the implementation of the algorithm and justify the decisions taken. Then, we present experimental results and discuss them in the light of the dependence on the platform and machine architecture. We present timing analysis of the implementation, as well as results on the average length of the longest path.

1. Introduction. A two-tree is a particular type of graph. The smallest possible two-tree consists of a single line segment (u, v) known as an edge. All larger two-trees are constructed by adding a single vertex w and two more line segments to an existing two-tree. One line will contain the endpoint a and the other the endpoint b of an existing edge, (a, b), on the two-tree. Both line segments will contain w as the other endpoint (Figures 1 and 2)[2].

ACM Computing Classification System (1998): G.2.2, F.2.2.
Key words: longest path, 2-trees, divide-and-conquer, experimental analysis.

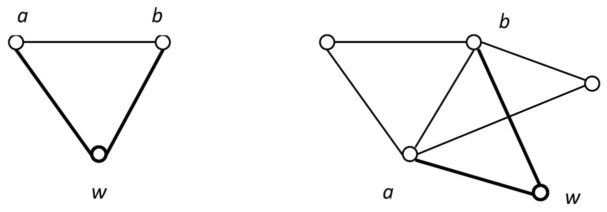


Fig. 1. A two-tree is created by adding a vertex w and two edges (w, a) and (w, b) to an edge (a, b) of an existing two-tree. The most trivial two-tree consists of a single edge

A path in a two-tree is described as a sequence of vertices connected by edges which contains no vertex twice. There exists an algorithm for finding the longest path in a two-tree, we are going to describe it briefly, and then will present an implementation of this algorithm. When the testing of the implementation is complete and the program runs with a certain degree of confidence, the timing of the implementation should be found. Though the timing of the algorithm is linear, it says little of the timing of the program which implements it as some designs will be more efficient than others.

2. The Algorithm. All non-trivial two-trees can be viewed as a graph of triangular faces attached along their edges, where each face is identified by it's three vertices (x, y, z) (Figure 2) [2]. Considering a two-tree in this fashion, all edges can be classified as either periphery or interior. A periphery edge is an edge that belongs to a single face and an interior edge is common to multiple faces [2]. From the two-tree in Figure 2, it is apparent that edge (w, v) is peripheral while edge (v, u) is interior.

In order to find the longest path in a given two-tree, the graph is recursively broken down into smaller independent subgraphs whose longest paths are easier to find. The simplest subtree is the trivial case where the tree consists of a single root edge. Careful consideration is needed to preserve the links between these subgraphs in order to find the correct longest path in the original graph. This demands care in the breaking down, or rather splitting, of the graph, a process which is carried out relative to the edges and faces of a graph.

Every non-trivial graph, G is split on it's root edge $e(u, v)$, $G \ominus e$. This means that each face, F which contains both vertices u and v is "peeled" off along with any structures that are built upon F. What is now present is a collection of smaller two-trees known as folios or the subtrees of G (G^1, G^2, G^3, $\dots$) and in each subgraph, whose root edge is also (u, v), the root edge is peripheral [2] (Figures 3 and 4). If the root edge of G is peripheral, $G \ominus$e, we will find a single

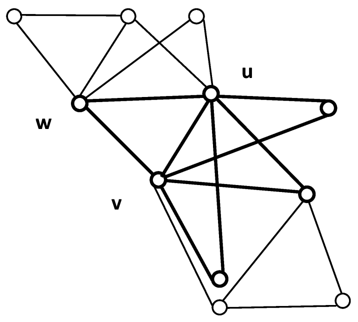


Fig. 2. An example of a two-tree graph G with root edge $e(u,v)$

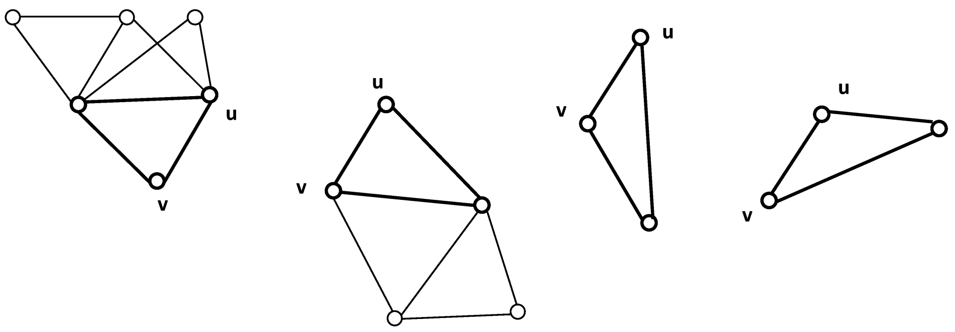


Fig. 3. The subgraphs resulting from $G \ominus e$

subgraph identical to G.

At this point trying to break down any of the subgraphs by this process of splitting on edge $G^i \ominus$e would be futile. Instead, the split is now done on the one face containing the root edge. If this root face contains the edges (u,v,w) with $e(u,v)$ being the root edge, the tree G^i can be split on face $(G^i - e)\ominus$w, into two subtrees G^{i1} and G^{i2}. G^{i1} will have root edge (u,w) and G^{i2} will have root edge (v,w) (Figure 4). It's impossible to tell whether or not either of these subtree root edges are peripheral or interior. The best way to split these trees would be to once again split on their root edges e_j, $G^{ij} \ominus e_j$.

In this manner any two-tree can be broken down continuously until all that remains is a single face $F(a,b,c)$, which is split into the two trivial subtrees (a,c) and (b,c), assuming a root edge of (a,b). The longest path for any trivial tree is clearly of length one. However, it becomes increasingly difficult to keep track of the longest path as trees become much larger, so each tree's root edge,

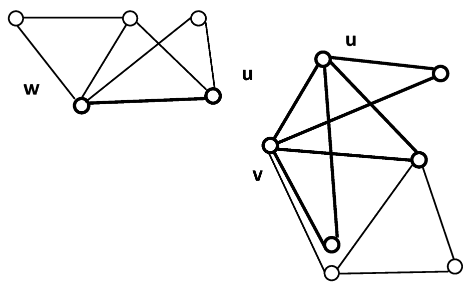


Fig. 4. The subgraphs resulting from $(G - (w, v)) \ominus u$

e, is associated with a label $\widetilde{\lambda}(e)$, which systematically keeps track of the tree's longest path and the lengths of the longest paths with starting or ending vertices at the roots of the tree [2]. This later information is needed for finding the longest path of a tree G from the labels of its subtrees G^1, G^2, G^3, ...

Suppose G is a graph with root edge $e(u, v)$. There are exactly 6 ways in which a path may be formed relative to e. By finding the longest path for each possible route, and from that the longest path for G, an ordered septuple:

$$\widetilde{\lambda}(e) = (\lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5, \lambda_6, \lambda_7)$$

known as a label for e may be found, where λ_i is described by the following [2]:

$\lambda_1 = \{p|p$ is a max path in $G\}$

$\lambda_2 = \{p|p$ is a max path in G with endpoints u and $v\}$

$\lambda_3 = \{p|p$ is a max path in G with endpoint u and internal vertex $v\}$

$\lambda_4 = \{p|p$ is a max path in G with endpoint u and not containing $v\}$

$\lambda_5 = \{p|p$ is a max path in G with endpoint v and internal vertex $u\}$

$\lambda_6 = \{p|p$ is a max path in G with endpoint v and not containing $u\}$

$\lambda_7 = \{< p, q > \mid < p, q >$ are max-sum $< u, v >$ paths in $G\}$

Now, rather than reconstructing every subgraph beginning from the trivial trees, the labels of the trivial trees can be used to acquire the labels of the subtrees which they are derived from [2]. In this way a label can be found for every subtree's root edge working backwards up until finally a label is found for the original tree, G's, root edge (u, v).

As two methods are needed to break down a tree, the algorithm consists of two mutually recursive functions; *computeLabel* and *computeSimple*.

ComputeLabel will recognize a graph of a two-tree G as trivial and assign it the appropriate label, or it will split G on it root edge creating a listing of all the subtrees G^i. For each of these subtrees the function *computeSimple* is called to further subdivide each of G^i into two more trees. Continuing to decompose the original tree G, *computeSimple* then calls the function *computeLabel* for each of these two subtrees G^{ij}. Together the two functions work to decompose the graph.

When all subtrees have been broken down to trivial edges whose labels are known to be $(1, 1, 0, 0, 0, 0, 0)$, a second part to the above functions, *computeLabel* and *computeSimple*, works to calculate the longest path. This requires the use of two private internal functions: *combineOnEdge* and *combineOnFace*. As their names would suggest, the *combineOnEdge* function is called by *computeLabel* as it splits a tree on its edge and the *combineOnFace* function is called by *computeSimple* as it splits a tree on its face. Both *combine* functions take all the subtrees created in the reciprocal split and use the subtrees' root edge labels to calculate the label of their root tree's root edge and thus the longest path in G.

For example, consider the tree T shown in Figure 5 with root edge (a, b). Following the algorithm, in order to find the longest path in the tree, (a, b) and T are passed to the function *computeLabel*.

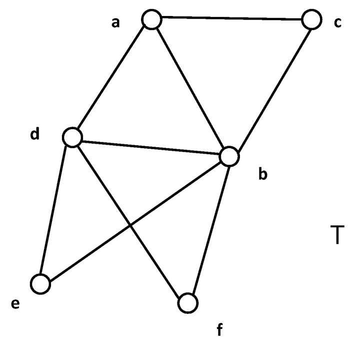


Fig. 5. An example of a two-tree T with root edge (a,b)

Since T is not trivial, the function will split the graph on the edge (a, b), $T \ominus (a, b)$, creating two subgraphs T^1 and T^2 in which their root edges (a, b) are peripheral (Figure 6).

Both subgraphs are held by the function *computeLabel*$((a, b), T)$ and for each graph the function *computeSimple* is called. *ComputeSimple*$((a, b), T^1)$ will be called first. From the peripheral root edge (a, b) the function will locate the vertex d and split T^1 on the root face (a, b, d), producing two more subgraphs,

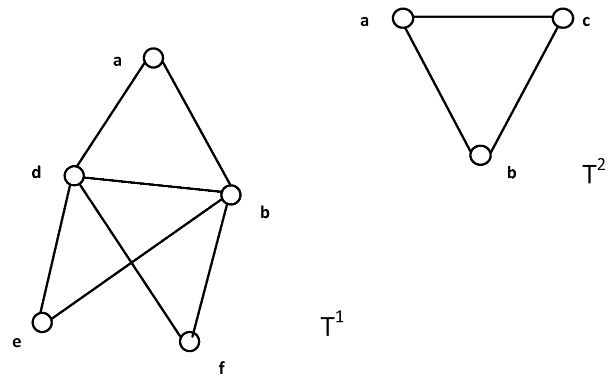


Fig. 6. The two-trees T^1 and T^2 resulting from $T\ominus$(a,b)

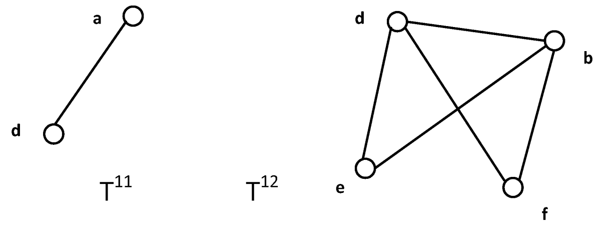


Fig. 7. $(T^1 - (a,b))\ominus$d results in two more subgraphs, T^{11} with root edge (a,d) and T^{12} with root edge (d,b)

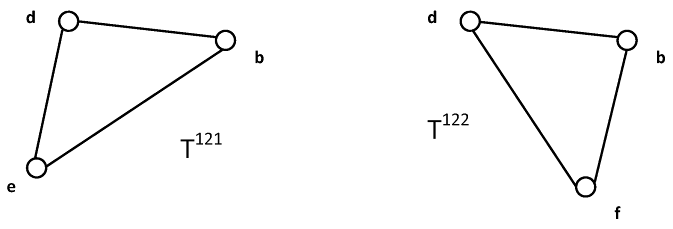


Fig. 8. Subtrees T^{121} and T^{122} resulting from $T^{12}\ominus$(d,b)

T^{11} and T^{12} (Figure 7).

$ComputeSimple((a,b),T^1)$ next calls for both newly created subgraphs and their root edges to be passed to *computeLabel*. T^{11} and its root edge are passed first. $ComputeLabel((a,d),T^{11})$ will recognize the tree as trivial. Rather than attempting to break it down any further, $computeLabel((a,d),T^{11})$ will assign values to the edge's associated label, $\widetilde{\lambda} = (1,1,0,0,0,0,0)$. T^{12} and the root edge (b, d) are now passed to *computeLabel*. As this graph is not trivial $computeLabel((b, d), T^{12})$ will split the subtree on its root edge resulting in further subtrees T^{121} and T^{122} (Figure 8).

Control will pass to $computeSimple((b, d), T^{121})$. The function will split the T^{121} on face into two trivial trees T^{1211} and T^{1212} (Figure 9). $ComputeLabel$ $((d, e), T^{1211})$ will then recognize T^{1211} as trivial and assign it the appropriate label, and $ComputeLabel((b, e), T^{1212})$ will find T^{1212} to be trivial and so also assign its label $(1, 1, 0, 0, 0, 0, 0)$.

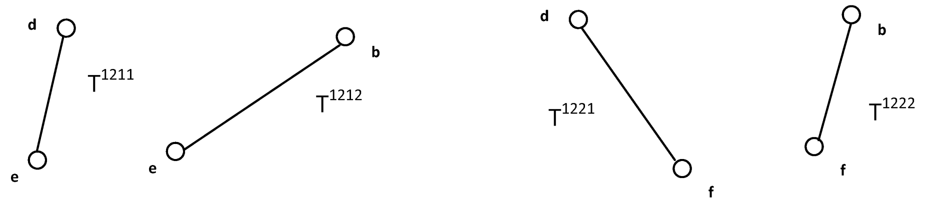


Fig. 9. T^{121} split on face, $(T^{121} - (b, d)) \ominus e$ creates two trivial graphs T^{1211} and T^{1212} and $(T^{122} - (b, d)) \ominus f$ creates two trivial subtrees T^{1221} and T^{1222}

As both subtrees T^{1211} and T^{1212} resulting from $(T^{121} - (b, d)) \ominus e$ now have associated labels, $computeSimple((b, d), T^{121})$–the function which created these trees–can find the label of the root edge of the subtree T^{121} which it was passed. To do so it calls the function *combineOnFace* with the labels of the two subtrees as the arguments. This function will return the label $(2, 2, 2, 1, 2, 1, 1)$ associated with the root edge of the tree T^{121}. This completes $computeSimple((b, d), T^{121})$.

Control will return to $computeLabel((b, d), T^{12})$ which will now call *computeSimple* for the other subtree, T^{122}. $ComputeSimple((b, d), T^{122})$ will split T^{122} on the face (b, d, f) producing the pair of subtrees T^{1221} and T^{1222} (Figure 9). The function call $computeLabel((d, f), T^{1221})$ is made and the trivial tree (d, f) is given the label $(1, 1, 0, 0, 0, 0, 0)$. Similarly, the same label is attached to the trivial two-tree (b, f) after the function call $computeLabel((b, f), T^{1222})$. Returning to $computeSimple((b, d), T^{122})$, *combineOnFace* is now called to combine the two trivial labels and returns the label $(2, 2, 2, 1, 2, 1, 1)$ which is associated with the edge (b, d) of the subtree T^{122}.

All subtrees, that is to say T^{121} and T^{122}, resulting from $T^{12} \ominus (d, b)$ of the function call $computeLabel((b, d), T^{12})$ (Figure 7) have at this point been passed to *computeSimple* and now have labels. The final step in $computeLabel((b, d), T^{12})$ is to combine these labels by calling the method *combineOnEdge* using the labels of the subtrees as arguments. This will result in the label $(3, 2, 3, 1, 3, 1, 2)$ and complete $computeLabel((b, d), T^{12})$.

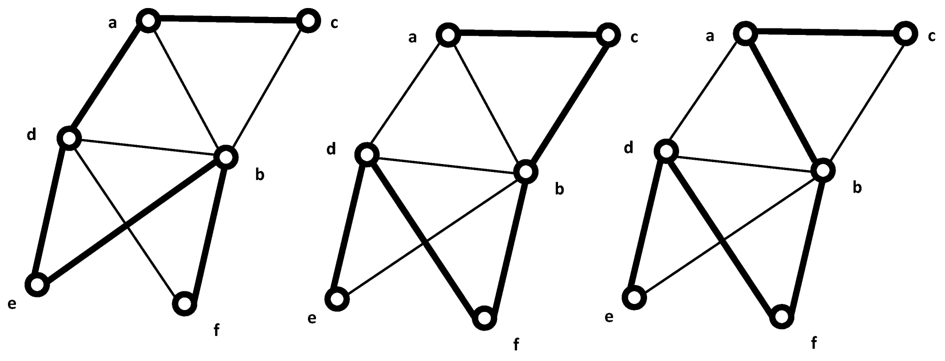


Fig. 10. Possible longest paths in T

Now both subtrees T^{11} and T^{12}, which are results within *computeSimple* $((a,b),T^1)$, have labels associated with their root edges. The final command in *computeSimple*$((a,b),T^1)$ is the function call *combineOnFace*. It takes the two labels, $(1,1,0,0,0,0,0)$ and $(3,2,3,1,3,1,2)$, combines them on the root face of T^1 and produces the label $(4,3,4,2,3,3,3)$.

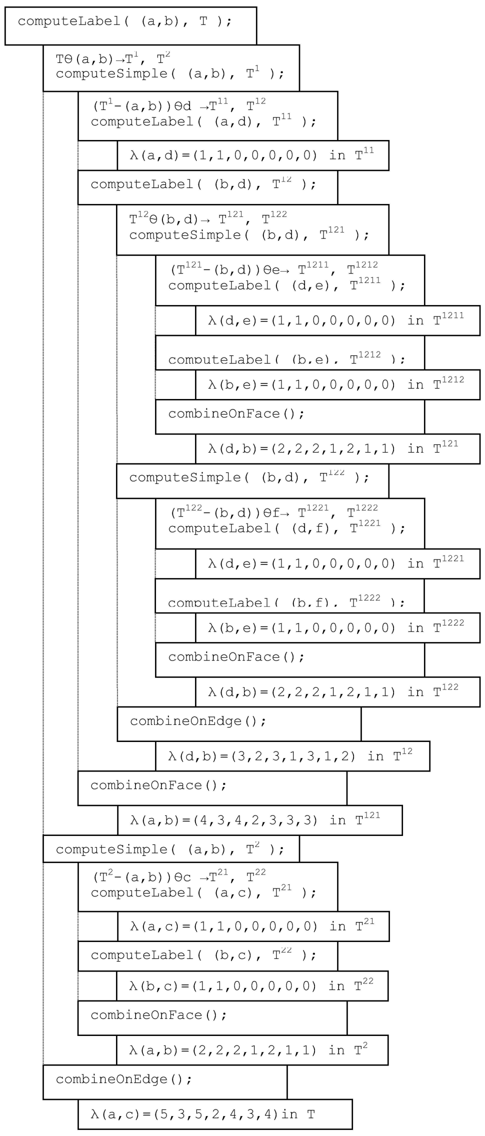
The program will move onto computing *computeSimple*$((a,b),T^2)$. Splitting the single face will result in two trivial trees $T^{21} = (a,c)$ and $T^{22} = (b,c)$. Both will immediately be assigned the trivial label when passed to *computeLabel* and both labels will be passed to *combineOnFace*. The returned label will be $(2,2,2,1,2,1,1)$ completing *computeSimple*$((a,b),T^2)$.

The final step in competing the algorithm is to *combineonEdge* the labels of the first subtrees created; that of T^1, $(4,3,4,2,3,3,3)$, and that of T^2, $(2,2,2,1,2,1,1)$. The function will return the label $(5,3,5,2,4,3,4)$. The first term in this label represents the length of the longest path, so the longest path possible in the two-tree T is 5 as shown in Figure 10.

3. Programming the Algorithm.

It was apparent from the modular design of the algorithm that an object oriented approach would work out neatly. It also had the advantage of allowing for relatively easy modification of the program as needed and the partitioning involved would make testing and debugging easier. With the idea of using an adjacency matrix for input, there was no obvious need for any pointers for the programming of the algorithm, so Java was chosen as the language. Specifically, Eclipse was used as the Java programming environment.

Much of the program structure is provided in algorithm itself but to bridge the gap between algorithm and program a few new functions had to be added



```
computeLabel( (a,b), T );
    Tθ(a,b)→T^1, T^2
    computeSimple( (a,b), T^1 );
        (T^1-(a,b))θd →T^11, T^12
        computeLabel( (a,d), T^11 );
            λ(a,d)=(1,1,0,0,0,0,0) in T^11
        computeLabel( (b,d), T^12 );
            T^12θ(b,d)→ T^121, T^122
            computeSimple( (b,d), T^121 );
                (T^121-(b,d))θe→ T^1211, T^1212
                computeLabel( (d,e), T^1211 );
                    λ(d,e)=(1,1,0,0,0,0,0) in T^1211
                computeLabel( (b,e), T^1212 );
                    λ(b,e)=(1,1,0,0,0,0,0) in T^1212
                combineOnFace();
                    λ(d,b)=(2,2,2,1,2,1,1) in T^121
            computeSimple( (b,d), T^122 );
                (T^122-(b,d))θf→ T^1221, T^1222
                computeLabel( (d,f), T^1221 );
                    λ(d,e)=(1,1,0,0,0,0,0) in T^1221
                computeLabel( (b,f), T^1222 );
                    λ(b,e)=(1,1,0,0,0,0,0) in T^1222
                combineOnFace();
                    λ(d,b)=(2,2,2,1,2,1,1) in T^122
            combineOnEdge();
                λ(d,b)=(3,2,3,1,3,1,2) in T^12
        combineOnFace();
            λ(a,b)=(4,3,4,2,3,3,3) in T^121
    computeSimple( (a,b), T^2 );
        (T^2-(a,b))θc →T^21, T^22
        computeLabel( (a,c), T^21 );
            λ(a,c)=(1,1,0,0,0,0,0) in T^21
        computeLabel( (b,c), T^22 );
            λ(b,c)=(1,1,0,0,0,0,0) in T^22
        combineOnFace();
            λ(a,b)=(2,2,2,1,2,1,1) in T^2
    combineOnEdge();
        λ(a,c)=(5,3,5,2,4,3,4)in T
```

Fig. 11. The complete trace of the algorithm for finding the longest path in the tree T

and, using an object-oriented approach, classes and subclasses had to be decided upon (Figure 12).

The first class to consider is the *TwoTree* class. As two-trees are represented as adjacency matrices in the program, the class contains the member

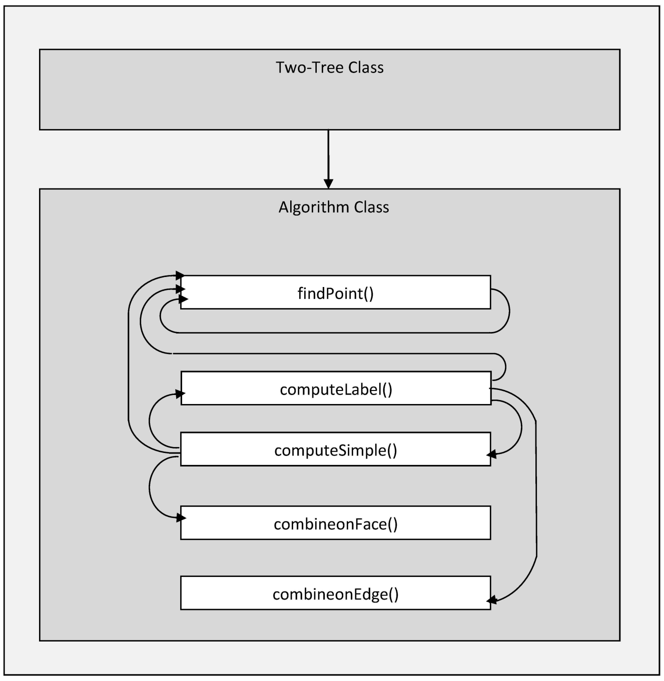


Fig. 12. A general outline of the main methods in the program. Class $Two - Tree$ is the parent to the class *Algorithm*. In the *Algorithm* class *computeLabel*() calls the recursive function $findPoint()$ to aid in the creation of the subclasses. Once they are created *computeLabel*() calls *computeSimple*() for each subtree. This function also calls $findPoint()$ to make it's two subtrees. Each of these subtrees calls the *computeLabel*() commencing the mutual recursion. After reaching it base case *computeSimple*() calls the function $combineOnFace()$. Likewise, *compluteLabel*() calls *conbinOnEdge*()

graph[ ][ ] which will hold the matrix representation of the tree; it holds the graph itself. Since the longest path in a two-tree is invariant to the choice of a root edge, and as the whole point of the label in the algorithm is to find the longest path, a single label is attached to a graph rather than an edge of the graph. Thus, another member of the *Graph* class is *label*. Along with these two members come various getter, setter, and manipulator functions: *getAij*, and *setAij* for *graph*[ ][ ]; and *setLabels* for *label*. There is also a function *colCompare*(*Integer firstCol*, *Integer secondCol*, *Graph g*) which compares two columns in *graph*[ ][ ] and returns a vector of the row numbers for which both columns contain a one. In a two-tree this corresponds to finding the set of all vertices that are connected to a given edge (w, x), described by the first two parameters of the *colCompare*. This effectively determines the set of all faces incident to the edge (w, x), as illustrated in Figure 13. Another function, *Label* simply returns the label of the *Graph* instance which called it. To read a tree-from a file the method $getGraphFile(Stringfile)$ was created. The only private function in the class is *assertGraph*, which stands as a safety net and double checks a matrix for a line of symmetry over the diagonal which must be all zeros.

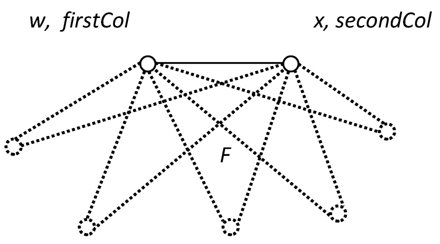


Fig. 13. *colCompare* function finds all faces containing the edge (w, x)

Using simple inheritance then, the class *Algorithm* publicly extends *TwoTree*. As the name describes, the class carries out the requests of the algorithm including its functions and all secondary methods needed to carry out these functions. First off, the class contains a constructor function to allow for the immediate creation of an instance of *TwoTree*, whose adjacency matrix is contained in the file `sGraph.txt`. The only public function in the class is $longestPath(TwoTree\ g,\ Integer\ u,\ Integer\ v)$ which is passed a two-tree g and the vertices of its root edge $e(u, v)$. The function does little other than set the values for g's associated label by calling the public method *setLabels* from the class *TwoTree* with the returned label from the function *computeLabel* : $g.setLabels(computeLabel(g, u, v));$. *LongestPath* then returns the first term in the label; the longest path in the two-tree g.

As described in the Algorithm (see Section 2) *computeLabel* splits a graph on its edge and *computeSimple* splits a graph on a periphery face. Programming this process relative to a matrix is a little more complicated than simply understanding it relative to a two-tree diagram. However, once a root edge has been found for a subtree, the process of building the subtree relative to the adjacency matrix representation of the parent tree is the same regardless of the type of split allowing a single function to be reused. The recursive method $findPoint(Integer\ u,\ Integer\ v,\ Integer\ x,\ TwoTree\ graph,\ Vector < Integer > \ subGraphVert)$ has been designed to find the third vertex of every face in the subtree (which is equivalent to all vertices less those of the root edge) given a singe root face (u, v, x), the parent two-tree, and a vector to store the vertices in. Please refer to Figure 14.

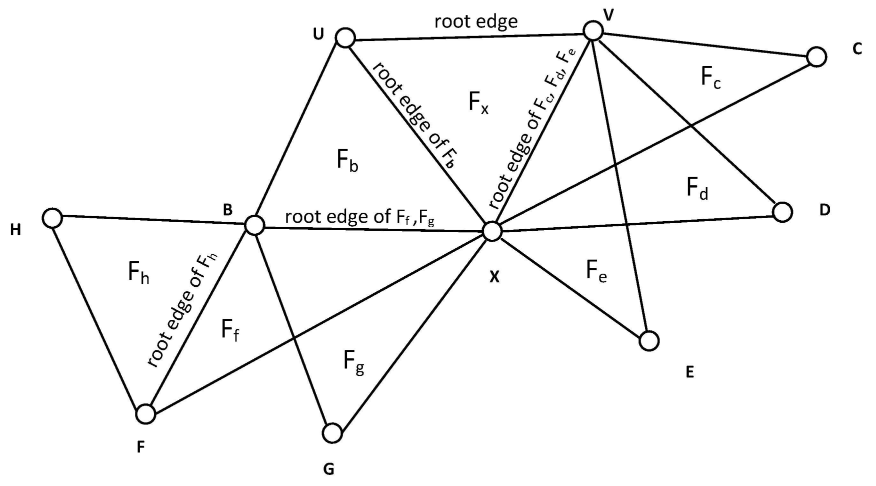


Fig. 14. An example of a two-tree with root face F_x. The method $findPoint()$ would recursively locate vertices B through H given F_x

The $findPoint$method works by considering each face as containing a root edge (u, v) and a vertex x. Now every other vertex can be found simply by comparing the root edge columns in $graph[][]$. This is done by calling the function $colCompare(Integer\ firstCol,\ Integer\ secondCol,\ Graph\ g)$ which produces the vertex list. This list is then appended to $subGraphVert$, the final argument in the method. $FindPoint$ considers two root edges; the left (u, x) and the right (v, x). Each root edge is examined by *colcompare* to find the final vertex in any further existing faces. The recursive call stops when the resulting vector is found to be empty. For each vertex that is found in the vector, two new left and right roots are created and $findPoint$ is called recursively. The end result is a vector list $subGraphVert$ containing every vertex in the subgraph. This list is then to be sorted, and a new adjacency matrix is created by copying terms from the original.

Other additional methods include *max* and *positive*. Max has been overloaded to find and return to the caller the maximum integer in an array: $max(Integer[]list)$ or the maximum of two integers: $max(Integer\ x,\ Integer\ y)$. The method *positive*(*Integer otherwise*, *Integer isZero*) returns *otherwise* if *iszero* is not equivalent to zero and zero if it is.

As described in the algorithm if the two-tree passed to the function *computeLabel*($TwoTree\ g$, *Integer u*, *Integer v*) is trivial, then the trivial label is immediately returned. Otherwise, the two-tree graph g is split on its root edge [2]. To split a graph on an edge $e(u, v)$ all the folios are separated to form the subtrees. Each of these folios contains a root face with the periphery edge e. The first step is to find all faces containing e. These will be the root faces of the folios. The number of faces found will then be equivalent to the number of subtrees as the trees are "built" upon these faces. Again, the *colcompare* function is used. Now, for each vertex in the resulting vector the $findPoint$ method is called and a subtree is created. The function *computeSimple* is called for each of the subtrees returning a label for each. These labels are then combined with the function *combineOnEdge* which returns a single label for g.

The method $computeSimple(TwoTree\ g,\ Integer\ u,\ Integer\ v)$ is passed a subtree g with peripheral root edge $e(u, v)$ and calls for the splitting of g on its face. By calling the *colcompare* function for the root edge, the root face $F(u, v, w)$ is found. Repeating this action with the edge (u, w) will result in a list $subMatrixChildren$, of folios' root faces—more precisely a list of the third vertex z in each face; the other two vertices are u and w. After each face $F'(u, w, z)$ and the vector $subGraphVert$ has been passed to the method $findPoint$, $subGraphVert$, which was originally empty, contains a list of all the vertices in the folios formed by the (u, w) edge of F. This list is then used to form a subtree in the same way as those subtrees are formed in the function *computeLabel*. Similarly, the subtree formed by the folio with root edge (v, w) is found completing $(g - e) \ominus w$. *ComputeSimple* begins the recursion by calling the method *computeLabel* for each of the two subtrees just formed. The two labels that result are then combined by the method $combineOnFace$.

With the functions *max* and *positive* defined, programming the method $combineOnFace(Integer[\,]\ labelG1,\ Integer[\,]\ labelG2)$ was relativel straight forward. The code follows the algorithm nearly line for line. The functions is passed the labels of the two subgraphs, $labelG1$ and $labelG2$, and combines them to find each term in the label of the root graph. These terms are stored in the array *combinedLabel* which is returned as the root graph label when the function completes execution.

Completing *computeSimple*, the execution returns to *computeLabel*. When each of the subgraphs g^i created by $g \ominus e$ has obtained its label λ_i, these labels are passed to the function $combineOnEdge(Integer[\,][\,]\ subGraphLabels)$ in the form of a $n \times 7$ integer matrix. The method *combineOnEdge* was not as simple to program as $combineOnFace$. Due to the restrictions of the Java compiler, a single command in the algorithm such as $y = \max\{\lambda_4^i + \lambda_6^j | 1 \leq i \leq k, 1 \leq j \leq k, i \neq j\}$ cannot be written as a single line of code. Rather the set

$\{\lambda_4^i + \lambda_6^j | 1 \leq i \leq k, 1 \leq j \leq k, i \neq j\}$ must be created and stored as a list, known as $L4L6$ in the program, before the maximum of the list may be found. To construct $L4L6$ from the parameter *subGraphLabels* an embedded for loop is used as it allows every combination possible to be found. Similarly, for the list $L2L4L6$ to be created, an embedded for loop is embedded in another for loop. With the list constructed the maximum can be found by passing the list to the function *max*. Once all the sets have been created the function follows the designed algorithm once again nearly line for line and returns a single label for the graph g.

Having just completed the *combineOnEdge* method, the function *computeLabel* returns the same label to its caller *longestPath*. The driver function receives the length of the longest path from *longestPath* and prints it.

4. Random Two-Tree Generation and Timing Trials. To test the timing of the program, it was decided to run 1000 trials of random two-trees for each of the sizes: 100, 200, 400, 800, 1600, 3200, 6400, and 12800 vertices. In order to do this, a random two-tree generator had to be created. As two-trees are represented as adjacency matrices, this was merely a matter of creating random adjacency matrices of the desired size which accurately represents a unique two-tree. The first set of trials were run on a 32-bit machine with 3GB of RAM.

In order to get the running time for the implemented algorithm, several lines of code were added to the driver, one before the function *longestPath* was called:

```
long startTime = System.currentTimeMillis();
```

and two after.

```
long endTime = System.currentTimeMillis();
String aString = Integer.toString((int) (endTime-startTime));
```

Simply stated, the time in milliseconds is recorded both before and after the function call. The difference in this time is the time needed to carry out the demands of the function, which will be the time it takes for the longest path algorithm to complete. To make recording easier, the process of creating of a graph, finding it's longest path, and recording the path length in a file is looped in the driver function.

A small public class, *Generator*, was designed to create two-tree graphs. It was added to the program so it could be called with the same driver as the longest path algorithm. In this way only one program has to be run to create and find the longest path of a two-tree rather than having to run a separate program to create the tree. The original design consists of for loop that creates a new matrix with every iteration. An original matrix of size two represents the most basic two-tree possible. A vector, *edges*, is used to keep track of every edge in the tree. Each term in the vector represents an edge, and thus contains the two vertex integers which describe the edge. Every time a new vertex is added to the tree it creates two new edges which are added to the vector. The original vector has only one term as there is only one edge to begin with. In order to create random two-trees, new vertices are added to random edges. A counter is used to keep track of the number of edges.

The loop runs until the desired graph size is acquired. First it asks for a random number between 0 and the counter value be chosen. The number chosen will correspond to an edge in the vector. It is to this edge that the next vertex is added. A new matrix, one size larger than the current matrix, is created and the contents of the current matrix are copied to the new one. The final row and column of the new matrix, which represents the new vertex, are determined by the random edge which was just picked; the edge the new vertex was attached to. Lastly, the current matrix is discarded and the new matrix becomes the current matrix. This cycle will continue until the end of the loop.

Though the design worked well for the fist few matrix sizes (100, 200, and 400 vertices), when the graph size increased to 800, the time needed to create a single graph–the timing of the generator class–increased dramatically. Further, when the graph size was increased again to 3200, the generator class would cause the program to fail every 50 or so trials having used up all available heap space.

A few minor changes were made to the program in an attempt to decrease the amount of memory being used. In this case, Java was limiting. Its use of the garbage collection disallows the programmer to allocate and deallocate memory with the same precision as is possible in C++. A few lines of code–requesting a garbage collection, deleting or resetting a matrix to null–could, however, be added to the program at various appropriate points to help conserve memory. All these changes were in vain though as the program continued to run out of memory with what appeared to be the same frequency as before. Thus the first set of trials, run off the 32 bit system were carried out only to a size of 1600.

Two changes were made in an effort to get the desired number of trials complete: the design of the generator program was improved and it was decided to use a larger machine on which to complete the trials. The main change in the design is that one single matrix is used. It is initialized at the beginning of the program setting all terms to zero. From there on it follows the same general pattern as its predecessor, using a for loop and choosing random edges to attach a new vertex. However, no longer is it ever the case that there is more than one matrix or graph being held in the memory at any time. In this manner the design is incredibly more efficient. Consider a graph of size 3200. At it's worst moment the previous design would have demanded that two matrices of size 3199 and 3200 be held simultaneously in the heap space memory. In the newer design this number is nearly halved. What is more in the previous design 3200 matrices were created and 3199 of them had to be copied. The new design also changed the dynamic edge, vector *edges*, to a static array initialized to the desired size of the graph. The second change was opting for a larger machine. All further trials were run on a 64–bit machine with 8G of RAM.

The running trials resumed with the changes in effect. To maintain accuracy and avoid any unnecessary discrepancies however, the trials had to be restarted from the initial size of 100 vertices. With the loop in the diver function set to 1000 iterations, the first few sets of trials were quickly attained. Even through sizes of 800 and 1600 there was no significant jump in the time needed to create graphs. Everything ran along smoothly until trials of size 3200 were being run. Once again it was noticed that not all trials could complete, some were running out of heap space memory. The matrix used was of type Integer but in reality an adjacency matrix uses nothing other than zeros and ones. It seemed obvious that in order to get more memory the matrix type should be replaced by another data type that is more efficient.

Using boolean sounds as the most obvious choice as there are only two options available, zero and one. A few modifications were created to the Integer version and the boolean version was created. Trials were once again run starting from scratch, but again memory became an issue in size 3200. It was as if no change had been made. A further look into java's built in boolean type found that in spite of representing only a single bit of information, its actual size cannot be precisely defined. The likely cause of the the memory problem lies in this vagueness [1].

After a few similar changes to the program, the boolean type was replaced with byte. In this case it is as the name suggests, an 8-bit integer. The type Integer must be at least 32-bit as it wraps the type int into an object [1].

Much better results were expected as there is an obvious place where memory will be saved. Once again trials were run from the start but once again, despite there being an improvement, there was not enough memory available to complete the trials to 12800.

Between the running of trials for boolean and byte types, having not yet lost hope in the Integer matrix version, a way to allocate more memory to

Eclipse was found. Using another identical machine, the trials from type Integer were continued. If the program (ever) run out of memory before all the trials of given size were completed, more memory would automatically be allocated to Eclipse. This continued, until it became apparent that enough memory could be reserved to complete all sets of trials including those graphs of size 12800. At times possibly as much as 7G of the 8G RAM were being used for Eclipse and the running of the program.

5. Experimental Results. With the necessary information at hand, the timing of the program could be analyzed. Though the timing of the algorithm is linear, the timing of the implementation, namely the program and the way it has been designed, was expected to be quadratic. This is exactly what was found. The running time results from the trials with Integer type matrices (Table 1) confirmed this hypothesis. The column T/N^2 is the most constant revealing that the algorithm's implementation is closest to quadratic in time, $O(N^2)$. The divergency of the column T/N shows how $O(N)$ is an underestimate for the timing. The increasing nature of columns $T/\sqrt{N}$ and $T/(N\log_2 N)$ show the same for estimates of $O(\sqrt{N})$ and $O(N\log_2 N)$ respectively. On the other hand, the column T/N^3 appears to be converging to zero confirming $O(N^3)$ is an overestimate [3].

This conclusion is only confirmed by the other trials.

As it was necessary to run trials to obtain the program's timing, it was decided to take a look at the lengths of the longest paths at the same time. This in no way interfered with finding the timing of the program as all that had to be done was requesting that the longest path length be recorded to a file, `Longest Path.txt`, when it was found. So for every trial completed, not only was a running time recorded in a file `Times.txt`, but the length of the longest path was recorded in `Longest Path.txt`.

The data collected was handled similar to that of the timing. An average

Table 1. Trials of type Integer

N	T (msec)	$T/\sqrt{N}$	T/N	T/N^2	T/N^3	$T/(N\log_2 N)$
100	2.48	0.002482	0.0248	0.000248	2.482×10^{-6}	0.00374
200	7.02	0.002482	0.0351	0.000176	8.775×10^{-7}	0.00459
400	25.48	0.003185	0.0637	0.000159	3.981×10^{-7}	0.00737
800	107.31	0.004743	0.1341	0.000168	2.096×10^{-7}	0.01391
1600	455.37	0.007115	0.2846	0.000178	1.112×10^{-7}	0.02674
3200	2061.11	0.011386	0.6441	0.000201	6.290×10^{-8}	0.05532
6400	9004.01	0.017586	1.4068	0.000220	3.435×10^{-8}	0.11127
12800	45095.34	0.031140	3.5231	0.000275	2.510×10^{-8}	0.25822

Table 2. Trials of Type Boolean

N	T (msec)	$T/\sqrt{N}$	T/N	T/N^2	T/N^3	$T/(N \log_2 N)$
100	2.891	0.00289	0.0289	0.000289	2.891×10^{-8}	0.00435
200	7.501	0.00265	0.0375	0.000188	9.377×10^{-7}	0.00491
400	25.229	0.00315	0.0631	0.000158	3.941×10^{-7}	0.00730
800	98.234	0.00434	0.1228	0.000153	1.919×10^{-7}	0.01273
1600	424.097	0.00662	0.2651	0.000165	1.035×10^{-7}	0.02490
3200	1966.671	0.01086	0.6146	0.000192	6.002×10^{-8}	0.05278
6400	8584.170	0.01677	1.3413	0.000210	3.274×10^{-8}	0.10608

Table 3. Trials of Type Byte

N	T (msec)	$T/\sqrt{N}$	T/N	T/N^2	T/N^3	$T/(N \log_2 N)$
100	2.639	0.00269	0.0269	0.000269	2.693×10^{-6}	0.00405
200	6.723	0.00238	0.0336	0.000168	8.404×10^{-7}	0.00440
400	23.449	0.00293	0.0586	0.000147	3.664×10^{-7}	0.00678
800	77.381	0.00342	0.0967	0.000121	1.511×10^{-7}	0.01003
1600	290.444	0.00454	0.1815	0.000113	7.091×10^{-8}	0.01705
3200	1355.711	0.00749	0.4237	0.000132	4.137×10^{-8}	0.03638
6400	6792.514	0.01327	1.0613	0.000166	2.591×10^{-8}	0.08394

Table 4. Trials from 32-bit Machine

N	T (msec)	$T/\sqrt{N}$	T/N	T/N^2	T/N^3	$T/(N \log_2 N)$
100	63.389	0.06339	0.6339	0.00634	6.334×10^{-5}	0.09541
200	142.772	0.09339	0.7139	0.00357	1.785×10^{-5}	0.09339
400	327.005	0.04088	0.8175	0.00204	5.109×10^{-6}	0.09458
800	939.713	0.04153	1.1746	0.00147	1.835×10^{-6}	0.12180
1600	3902.231	0.06097	2.4389	0.00152	9.527×10^{-7}	0.22914

longest path was found for each of the different two-tree sizes and the general growth factor can be seen by looking at the summarizing tables below.

From the tables it can be observed that the length of the average longest path in a two-tree relative to the size of the tree, N, decreases to zero. It is less than linear and $N^{3/4}$ but greater than both $\log_2 N$ and $\sqrt{N}$. This was not surprising given that the expected diameter of an Euclidean minimum spanning tree of a uniformly distributed set of N points is known to be $O(\sqrt{N})$. Here the problem concerns the longest path rather than the minimum or the diameter. Further, distance is not a concern. However, the expected diameter of a tree with N nodes is expected to be $O(\sqrt{N})$, and a two-tree can in fact be represented as a standard tree of N nodes, by representing the tree's faces as edges and the

Table 5. Trials of type Integer

N	Average Longest Path length	$T/(\log_2 N)$	$T/\sqrt{N}$	$T/(N^{3/4})$	T/N
100	51.746	7.7885	5.1746	1.6363	0.5174
200	82.739	10.8242	5.8505	1.5557	0.4137
400	130.787	15.1306	6.5393	1.4622	0.3270
800	204.257	21.1800	7.2215	1.3579	0.2553
1600	317.737	29.8517	7.9434	1.2560	0.1986
3200	497.496	42.7261	8.7946	1.1693	0.1554
6400	770.193	60.9144	9.6274	1.0763	0.1203
12800	1187.736	87.0528	10.4982	0.9867	0.0928

Table 6. Trials of type Boolean

N	Average Longest Path length	$T/(\log_2 N)$	$T/\sqrt{N}$	$T/(N^{3/4})$	T/N
100	51.590	7.7650	5.159	1.6314	0.1590
200	81.948	10.7208	5.7946	1.5409	0.4097
400	130.133	15.0550	6.5067	1.4549	0.3253
800	204.687	21.2246	7.2368	1.3607	0.2559
1600	320.834	30.1427	8.0209	1.2682	0.2005
3200	496.660	42.6543	8.7798	1.1673	0.1552
6400	770.193	60.7225	9.5971	1.0723	0.1200

Table 7. Trials of type Byte

N	Average Longest Path length	$T/(\log_2 N)$	$T/\sqrt{N}$	$T/(N^{3/4})$	T/N
100	51.619	7.7694	5.1619	1.6323	0.5162
200	82.132	10.7448	5.8076	1.5443	0.4107
400	130.206	15.0634	6.5103	1.4557	0.3255
800	203.984	21.1517	7.2119	1.3561	0.2550
1600	316.2	29.7072	7.9050	1.2499	0.1976
3200	497.756	42.7583	8.7991	1.1699	0.1555
6400	765.721	60.5607	9.5715	1.0701	0.1196

tree's edges as vertices. With this in mind it is believed that the longest path would increase at about the same rate, $\sqrt{N}$. Results show that this is actually an underestimate. It would be interesting whether the average length of the longest path in a 2-tree with N vertices can be tied to a function of N from general theoretical considerations.

Table 8. Trials from a 32-bit Machine

N	Average Longest Path length	$T/(\log_2 N)$	$T/\sqrt{N}$	$T/(N^{3/4})$	T/N
100	52.439	7.8928	5.2439	1.6582	0.5244
200	82.911	10.8467	5.8626	1.5590	0.4146
400	130.738	15.1250	6.5369	1.4617	0.3268
800	204.903	21.2470	7.2444	1.3622	0.2561
1600	318.751	29.9470	7.9688	1.2600	0.1992

6. Acknowledgments. Dr. Vassilev's research is supported by NSERC Discovery Grant. Joanna Ammerlaan's work on this project was funded partially by Nipissingu University Undergraduate Summer Research Awards (USRA) and partially by Dr. Vassilev's NSERC Discovery Grant. Both authors would like to acknowledge the friendly and collegial atmosphere of the 2nd Annual Workshop on Algorithmic Graph Theory held on May 16–20, 2011 at Nipissing University in North Bay, Ontario. In particular, we would like to thank Dr. Minko Markov for his suggestions, insights and discussion which helped us in carrying out this project.

REFERENCES

[1] Primitive data types (the java^{TM} tutorials learning the java language basics). `http://download.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html`, 1995.

[2] Markov M., T. Vassilev, K. Manev. A linear time algorithm for computing longest paths in 2-trees. *Ars Combinatoria*, **112** (2013), 329–351.

[3] Weiss M. A. Data Structures and Problem Solving Using C. Addison-Wesley, 2nd edition, 2000.

Department of Computer Science and Mathematics
Nipissing University
Box 5002, 100 College Drive
North Bay, Ontario P1B 8L7
Canada
e-mails: `tzvetalv@nipissingu.ca`,
`jammerlaan894@community.nipissingu.ca`

Received March 13, 2013
Final Accepted August 12, 2013