

Contents

1	Basic Test Results	2
2	TreeAnalyzer.c	3

1 Basic Test Results

```
1  Running...
2  Opening tar file
3  TreeAnalyzer.c
4  OK
5  Tar extracted O.K.
6  Checking files...
7  OK
8  Making sure files are not empty...
9  OK
10 Compilation check...
11 Compiling...
12 OK
13 Compilation seems OK! Check if you got warnings!
14
15 =====
16   Public test cases
17   =====
18
19 =====
20 Running test...
21 OK
22 Running test...
23 OK
24 Test good_root_last__4__3 Succeeded.
25 Info: valid tree with nodes 4 and 3.
26 =====
27
28 =====
29 Running test...
30 OK
31 Running test...
32 OK
33 Test invalid_test__4__3 Succeeded.
34 Info: incorrect number of nodes.
35 =====
36
37
38 =====
39 = Checking coding style =
40 =====
41 ** Total Violated Rules      : 0
42 ** Total Errors Occurs      : 0
43 ** Total Violated Files Count: 0
```

2 TreeAnalyzer.c

```
1  /**
2   * @file TreeAnalyzer.c
3   * @author Brahan Wassan <brahan>
4   * @version 1.0
5   * @date 27 Nov 2019
6   *
7   * @brief Program that build a tree from a given txt file
8   *
9   * @section DESCRIPTION
10  * The program builds a tree from txt file
11  * Input : txt file with integer that represent the tree nodes
12  * Process: checks if the user input is valid, and then print the Tree root, Vertices
13  * count, num of Edges in the tree, length of the minimal, and maximal branch in the tree, the tree Diameter, and
14  * the shortest path between 2 given nodes
15  * Output : print all the above
16  */
17
18 #include <stdio.h>
19 #include <stdlib.h>
20 #include <string.h>
21 #include <stdbool.h>
22 #include "queue.h"
23
24 #define MAX_CLI_ARG 4
25 #define MAX_LINE 1024
26 #define FILE_IDX 1
27 #define FIRST_NODE 2
28 #define SECOND_NODE 3
29 #define READ "r"
30 #define KEY_FACTOR 2
31 #define NUMBER_BASE 10
32 #define USAGE_ERR "Usage: TreeAnalyzer <Graph File Path> <First Vertex> <Second Vertex>\n"
33 #define INPUT_ERR "Invalid input\n"
34 #define ROOT_MSG "Root Vertex:"
35 #define NODE_COUNT "Vertices Count:"
36 #define EDGE_COUNT "Edges Count:"
37 #define MIN_BRANCH_LEN "Length of Minimal Branch:"
38 #define MAX_BRANCH_LEN "Length of Maximal Branch:"
39 #define DIAMETER_LEN "Diameter Length:"
40 #define SHORTEST_PATH_MSG "Shortest Path Between %d and %d: "
41 #define UNDEFINED_SIZE -1
42 #define MIN_TREE_SIZE 1
43 #define IS_LEAF -2
44 #define LEAF 1
45 #define FIRST_LINE 1
46 #define SEPARATOR " \t"
47 #define EQUAL 0
48 #define VISITED 1
49 #define EMPTY_QUEUE 1
50 #define SPACE_ASCII 32
51 #define NEW_LINE '\n'
52 #define LINE_WIN '\r'
53 #define NEW_LINE_WIN "\r\n"
54 #define LEAF_INDICATOR "-"
55 #define INT_LOW 48
56 #define INT_HI 57
57 #define LEAF_WIN "-\r\n"
58 #define LEAF_LIN "-\n"
59
```

```

60  /**
61   * The struct define one Vertex in the Graph
62   */
63  typedef struct Vertex
64  {
65      int parent;
66      int *sons;
67      int isLeaf;
68      int childrenCount;
69      int dist;
70      int prev;
71      int visit;
72  } Vertex;
73
74  /**
75   * the function initiate a tree with default values
76   * @param tree our tree
77   * @param size the size of the tree
78   */
79  void initTree(Vertex **tree, int size)
80  {
81      for (int i = 0; i < size; ++i)
82      {
83          (*tree)[i].childrenCount = 0;
84          (*tree)[i].sons = NULL;
85          (*tree)[i].parent = -1;
86          (*tree)[i].prev = -1;
87          (*tree)[i].dist = -1;
88          (*tree)[i].visit = -1;
89          (*tree)[i].isLeaf = -1;
90      }
91  }
92
93  /**
94   * the function free all the allocated memory
95   * @param tree the tree we build
96   * @param treeSize the tree Size
97   */
98  void freeEverything(Vertex **tree, int treeSize)
99  {
100     int i = 0;
101     if (*tree != NULL)
102     {
103         while (i < treeSize)
104         {
105             if ((*tree)[i].sons != NULL)
106             {
107                 free((*tree)[i].sons);
108             }
109             ++i;
110         }
111         free(*tree);
112         *tree = NULL;
113     }
114 }
115
116 /**
117  * the function prints the needed err msg
118  * @param isUsage flag that indicate if its USAGE err
119  */
120 void errMsg(bool isUsage)
121 {
122     if (isUsage)
123     {
124         fprintf(stderr, "%s", USAGE_ERR);
125     }
126     else
127     {

```

```

128     fprintf(stderr, "%s", INPUT_ERR);
129 }
130 }
131
132 /**
133  * closes the file and return fail
134  * @param file the file we want to close
135  * @return 1
136  */
137 int fileErrorHandling(FILE *file)
138 {
139     fclose(file);
140     return EXIT_FAILURE;
141 }
142
143 /**
144  * validate the line and parse it into integer arr
145  * @param line a line from the given file
146  * @return the size of vertex
147  */
148 int checkFirstLine(char line[MAX_LINE])
149 {
150     int i = 0;
151     for (; i < (int) strlen(line); ++i)
152     {
153         if (!(line[i] >= INT_LOW && line[i] <= INT_HI) && line[i] != NEW_LINE && line[i] != LINE_WIN)
154         {
155             return UNDEFINED_SIZE;
156         }
157     }
158     char *candidate = strtok(line, SEPARATOR);
159     int num = (int) strtol(candidate, NULL, NUMBER_BASE);
160     if (num < MIN_TREE_SIZE)
161     {
162         return UNDEFINED_SIZE;
163     }
164     return num;
165 }
166
167 /**
168  * validate single line in the file,
169  * check if their no double node (the same node) , no char , the vertices < size
170  * @param line input line from the input file
171  * @return the num of children if the line is valid, otherwise return -1
172  */
173 int validateLine(char line[MAX_LINE], int curSonsArr[MAX_LINE], int treeSize) //TODO WORKS
174 {
175     int childrens = 0;
176     if ((strcmp(line, LEAF_LIN) == EQUAL) || (strcmp(line, LEAF_WIN) == EQUAL) ||
177         (strcmp(line, LEAF_INDICATOR) == EQUAL))
178     {
179         return IS_LEAF;
180     }
181     if ((strcmp(line, "\n") == EQUAL) || (strcmp(line, NEW_LINE_WIN) == EQUAL))
182     {
183         return UNDEFINED_SIZE;
184     }
185     for (int i = 0; i < (int) strlen(line); ++i)
186     {
187         bool isNum = (!(line[i] >= INT_LOW && line[i] <= INT_HI));
188         if (isNum && (line[i] != SPACE_ASCII) && line[i] != NEW_LINE && line[i] != LINE_WIN)
189         {
190             return UNDEFINED_SIZE;
191         }
192     }
193     char *p = strtok(line, SEPARATOR);
194     while (p != NULL)
195     {

```

```

196         if ((*p == NEW_LINE) || (strcmp(line, LEAF_WIN) == EQUAL))
197         {
198             break;
199         }
200         int candidate = (int) strtol(p, NULL, NUMBER_BASE);
201         if (candidate > treeSize - 1)
202         {
203             return UNDEFINED_SIZE;
204         }
205         curSonsArr[childrens] = candidate;
206         childrens++;
207         p = strtok(NULL, SEPARATOR);
208     }
209     return childrens;
210 }
211
212 /**
213  * parse the file data to a tree
214  * @param fileName the given file name
215  * @param tree the tree we want to build
216  * @param treeSize the tree size
217  * @return 1 if failed, 0 otherwise
218  */
219 int parseFile(const char *fileName, Vertex **tree, int *treeSize)
220 {
221     FILE *file;
222     char line[MAX_LINE];
223     int lineNum = FIRST_LINE;
224     file = fopen(fileName, READ);
225     if (file == NULL)
226     {
227         return fileErrorHandling(file);
228     }
229     int sizeOfChildren = 0;
230     int curVertex = 0;
231     int curSonsArr[MAX_LINE];
232     while ((fgets(line, MAX_LINE, file) != NULL))
233     {
234         if (lineNum == FIRST_LINE)
235         {
236             *treeSize = checkFirstLine(line);
237             if (*treeSize == UNDEFINED_SIZE)
238             {
239                 return fileErrorHandling(file);
240             }
241             *tree = (Vertex *) malloc((*treeSize) * sizeof(Vertex));
242             if (*tree == NULL)
243             {
244                 return fileErrorHandling(file);
245             }
246             initTree(tree, *treeSize);
247             lineNum++;
248             continue;
249         }
250         curVertex = lineNum - KEY_FACTOR;
251         sizeOfChildren = validateLine(line, curSonsArr, *treeSize);
252         if (sizeOfChildren == UNDEFINED_SIZE)
253         {
254             return fileErrorHandling(file);
255         }
256         if (sizeOfChildren == IS_LEAF)
257         {
258             (*tree)[curVertex].isLeaf = LEAF;
259             (*tree)[curVertex].childrenCount = 0;
260             ++lineNum;
261             continue;
262         }
263         if (curVertex >= *treeSize)

```

```

264     {
265         return fileErrorHandling(file);
266     }
267     else
268     {
269         (*tree)[curVertex].childrenCount = sizeofChildren;
270         (*tree)[curVertex].sons = (int *) malloc((sizeofChildren) * sizeof(int));
271         for (int i = 0; i < sizeofChildren; ++i)
272         {
273             if ((*tree)[curSonsArr[i]].visit != VISITED)
274             {
275                 (*tree)[curVertex].sons[i] = curSonsArr[i];
276                 int i1 = curSonsArr[i];
277                 (*tree)[i1].visit = VISITED;
278             }
279             else
280             {
281                 return fileErrorHandling(file);
282             }
283         }
284         ++lineNum;
285     }
286 }
287 if ((*treeSize != (lineNum - KEY_FACTOR)))
288 {
289     fclose(file);
290     return EXIT_FAILURE;
291 }
292 fclose(file);
293 return EXIT_SUCCESS;
294 }
295
296 /**
297  * iterate thru the tree nodes and connect between a node and its parent
298  * @param tree the tree
299  * @param treeSize the tree size
300  */
301 void setParent(Vertex **tree, int treeSize)
302 {
303     for (int k = 0; k < treeSize; ++k)
304     {
305         if ((*tree)[k].isLeaf != LEAF)
306         {
307             for (int l = 0; l < (*tree)[k].childrenCount; ++l)
308             {
309                 int sonIdx = (*tree)[k].sons[l];
310                 (*tree)[sonIdx].parent = k;
311             }
312         }
313     }
314 }
315
316 /**
317  * the function find the tree root
318  * @param tree the tree
319  * @param treeSize the tree size
320  * @return the tree root, -1 if not found
321  */
322 int getRoot(Vertex *tree, int treeSize)
323 {
324     int i = 0;
325     for (i = 0; i < treeSize; i++)
326     {
327         if (tree[i].parent == UNDEFINED_SIZE)
328         {
329             return i;
330         }
331     }

```

```

332     return UNDEFINED_SIZE;
333 }
334
335 /**
336  * bfs according to the given psudo code
337  * @param tree the tree
338  * @param treeSize the tree size
339  * @param vertex the vertex we start from
340  */
341 void bfs(Vertex **tree, int treeSize, int vertex)
342 {
343     for (int i = 0; i < treeSize; ++i)
344     {
345         (*tree)[i].dist = UNDEFINED_SIZE;
346     }
347     (*tree)[vertex].dist = EQUAL;
348     (*tree)[vertex].prev = UNDEFINED_SIZE;
349     Queue *queue = allocQueue();
350     enqueue(queue, (vertex));
351     while (queueIsEmpty(queue) != EMPTY_QUEUE)
352     {
353         int curKey = (int) dequeue(queue);
354         int keyParent = (*tree)[curKey].parent;
355         if (keyParent != UNDEFINED_SIZE)
356         {
357             if ((*tree)[keyParent].dist == UNDEFINED_SIZE)
358             {
359                 (*tree)[keyParent].prev = (int) curKey;
360                 (*tree)[keyParent].dist = (*tree)[curKey].dist + 1;
361                 enqueue(queue, keyParent);
362             }
363         }
364         for (int i = 0; i < (*tree)[curKey].childrenCount; ++i)
365         {
366             int curSonIdx = (*tree)[curKey].sons[i];
367             if ((*tree)[curSonIdx].dist == UNDEFINED_SIZE)
368             {
369                 (*tree)[curSonIdx].prev = (int) curKey;
370                 (*tree)[curSonIdx].dist = (*tree)[curKey].dist + 1;
371                 enqueue(queue, curSonIdx);
372             }
373         }
374     }
375     freeQueue(&queue);
376 }
377
378 /**
379  * finds the minimum and maximum branches in the tree
380  * @param tree the tree
381  * @param treeSize the tree size
382  * @param root the root of the tree
383  * @param minVal the shortest branch
384  * @param maxVal the longest branch
385  * @return the maxVal idx
386  */
387 int findMinMaxBranch(Vertex *tree, int treeSize, int root, int *minVal, int *maxVal)
388 {
389     int curMin = treeSize + 1;
390     int curMax = 0, maxIdx = 0;
391     bfs(&tree, treeSize, root);
392     for (int i = 0; i < treeSize; ++i)
393     {
394         if (tree[i].dist > curMax)
395         {
396             curMax = tree[i].dist;
397             maxIdx = i;
398         }
399         if ((tree[i].dist != EQUAL) && (tree[i].dist < curMin) && (tree[i].isLeaf == LEAF))

```



```

400         {
401             curMin = tree[i].dist;
402         }
403     }
404     if (treeSize == MIN_TREE_SIZE)
405     {
406         curMin = 0;
407     }
408     *minVal = curMin;
409     *maxVal = curMax;
410     return maxIdx;
411 }
412
413 /**
414  * finds the diameter of the tree
415  * @param tree the tree
416  * @param treeSize the tree size
417  * @param maxIdx the vertex in the end of the longest branch
418  * @return the tree diameter
419  */
420 int findDiameter(Vertex *tree, int treeSize, int maxIdx)
421 {
422     int diameter = 0;
423     bfs(&tree, treeSize, maxIdx);
424     for (int i = 0; i < treeSize; ++i)
425     {
426         if (tree[i].dist > diameter)
427         {
428             diameter = tree[i].dist;
429         }
430     }
431     return diameter;
432 }
433
434 /**
435  * finds the path between two nodes
436  * @param tree the tree
437  * @param treeSize the tree size
438  * @param u the first node
439  * @param v the second node
440  */
441 void findPath(Vertex *tree, int treeSize, int u, int v)
442 {
443     fprintf(stdout, SHORTEST_PATH_MSG, u, v);
444     bfs(&tree, treeSize, v);
445     int curNode = u;
446     if (u == v)
447     {
448         fprintf(stdout, "%d\n", v);
449     }
450     else
451     {
452         fprintf(stdout, "%d ", u);
453         while (tree[curNode].prev != v && tree[curNode].prev != UNDEFINED_SIZE)
454         {
455             fprintf(stdout, "%d ", tree[curNode].prev);
456             curNode = tree[curNode].prev;
457         }
458         fprintf(stdout, "%d\n", v);
459     }
460 }
461
462 /**
463  * prints the output of the program
464  * @param tree the tree
465  * @param treeSize the tree size
466  * @param u the first node
467  * @param v the second node

```

```

468  */
469 void printOutput(Vertex *tree, int treeSize, int u, int v)
470 {
471     int root = getRoot(tree, treeSize);
472     fprintf(stdout, "%s %d\n", ROOT_MSG, root);
473     fprintf(stdout, "%s %d\n", NODE_COUNT, treeSize);
474     int edges = (int) (treeSize) - 1;
475     fprintf(stdout, "%s %d\n", EDGE_COUNT, edges);
476     int minVal, maxVal;
477     int maxIdx = findMinMaxBranch(tree, treeSize, root, &minVal, &maxVal);
478     fprintf(stdout, "%s %d\n", MIN_BRANCH_LEN, minVal);
479     fprintf(stdout, "%s %d\n", MAX_BRANCH_LEN, maxVal);
480     int diameter = findDiameter(tree, treeSize, maxIdx);
481     fprintf(stdout, "%s %d\n", DIAMETER_LEN, diameter);
482     findPath(tree, treeSize, u, v);
483 }
484
485 /**
486  * parse the two CLI given nodes into integers
487  * @param node the given cli node
488  * @param treeSize the tree size
489  * @return -1 if not valid, node value otherwise
490  */
491 int parseNodes(char *node, int treeSize)
492 {
493     for (int i = 0; i < (int) strlen(node); ++i)
494     {
495         bool isNum = (!(node[i] >= INT_LOW && node[i] <= INT_HI));
496         if (isNum)
497         {
498             return UNDEFINED_SIZE;
499         }
500     }
501     char *parsed = strtok(node, SEPARATOR);
502     int nodeValue = (int) strtol(parsed, NULL, NUMBER_BASE);
503     if (nodeValue == 0 && (strcmp(node, "0") != EQUAL))
504     {
505         return UNDEFINED_SIZE;
506     }
507     if (nodeValue > treeSize - 1 || nodeValue < 0)
508     {
509         return UNDEFINED_SIZE;
510     }
511     return nodeValue;
512 }
513
514 /**
515  * program main
516  * @param argc cli args
517  * @param argv cli args
518  * @return 0 if ok, 1 otherwise
519  */
520 int main(int argc, char *argv[])
521 {
522     Vertex *tree = NULL;
523     int treeSize = 0, flag = 0;
524     if (argc != MAX_CLI_ARG)
525     {
526         errMsg(true);
527         return EXIT_FAILURE;
528     }
529     flag = parseFile(argv[FILE_IDX], &tree, &treeSize);
530     if ((flag == EXIT_FAILURE) || tree == NULL)
531     {
532         errMsg(false);
533         freeEverything(&tree, treeSize);
534         return EXIT_FAILURE;
535     }

```

```

536     setParent(&tree, treeSize);
537     int firstNode = parseNodes(argv[FIRST_NODE], treeSize);
538     int secondNode = parseNodes(argv[SECOND_NODE], treeSize);
539     if (firstNode == UNDEFINED_SIZE || secondNode == UNDEFINED_SIZE)
540     {
541         errMsg(false);
542         freeEverything(&tree, treeSize);
543         return EXIT_FAILURE;
544     }
545     printOutput(tree, treeSize, firstNode, secondNode);
546     freeEverything(&tree, treeSize);
547     return EXIT_SUCCESS;
548 }

```