## Contents

1	Basic Test Results	2
2	TreeAnalyzer.c	3

## 1 Basic Test Results

```
Running...
    Opening tar file
   TreeAnalyzer.c
   OK
4
    Tar extracted O.K.
   Checking files...
8
   Making sure files are not empty...
9
   Compilation check...
    Compiling...
11
12
   Compilation seems OK! Check if you got warnings!
14
   15
   Public test cases
16
17
18
19
   Running test...
20
21
    OK
   Running test...
22
23
   OK
24
    Test good_root_last__4_3 Succeeded.
   Info: valid tree with nodes 4 and 3.
25
26
    _____
27
28
29
   Running test...
   OK
30
31
   Running test...
   Test invalid_test__4__3 Succeeded.
33
34
   Info: incorrect number of nodes.
35
36
37
38 ==========
   = Checking coding style =
39
    ** Total Violated Rules
41
   ** Total Errors Occurs
42
    ** Total Violated Files Count: 0
```

## 2 TreeAnalyzer.c

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

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

```
128
             fprintf(stderr, "%s", INPUT_ERR);
129
     }
130
131
132
      * closes the file and return fail
133
      * Oparam file the file we want to close
134
      * @return 1
135
136
     int fileErrorHandling(FILE *file)
137
138
139
          fclose(file);
         return EXIT_FAILURE;
140
     }
141
142
143
144
      * validate the line and parse it into integer arr
      * @param line a line from the given file
145
      * @return the size of vertex
146
147
     int checkFirstLine(char line[MAX_LINE])
148
149
     {
          int i = 0;
150
         for (; i < (int) strlen(line); ++i)</pre>
151
152
             if (!(line[i] >= INT_LOW && line[i] <= INT_HI) && line[i] != NEW_LINE && line[i] != LINE_WIN)
153
154
             {
155
                  return UNDEFINED_SIZE;
156
157
         }
158
          char *candidate = strtok(line, SEPARATOR);
         int num = (int) strtol(candidate, NULL, NUMBER_BASE);
159
         if (num < MIN_TREE_SIZE)</pre>
160
161
         {
             return UNDEFINED_SIZE;
162
163
         }
164
         return num;
     }
165
166
167
168
      * validate single line in the file,
      * check if their no double node (the same node) , no char , the vertices < size
169
      * @param line a line from the input file
170
171
      * Oreturn the num of children if the line is valid, otherwise return -1
172
     int validateLine(char line[MAX_LINE], int curSonsArr[MAX_LINE], int treeSize) //TODO WORKS
173
174
     {
          int childrens = 0;
175
176
         if ((strcmp(line, LEAF_LIN) == EQUAL) || (strcmp(line, LEAF_WIN) == EQUAL) ||
              (strcmp(line, LEAF_INDICATOR) == EQUAL))
177
178
179
             return IS_LEAF;
180
         }
         if ((strcmp(line, "\n") == EQUAL) || (strcmp(line, NEW_LINE_WIN) == EQUAL))
181
182
             return UNDEFINED_SIZE;
183
         7
184
         for (int i = 0; i < (int) strlen(line); ++i)</pre>
185
186
             bool isNum = (!(line[i] >= INT_LOW && line[i] <= INT_HI));</pre>
187
             if (isNum && (line[i] != SPACE_ASCII) && line[i] != NEW_LINE && line[i] != LINE_WIN)
188
189
             {
190
                  return UNDEFINED_SIZE;
             }
191
         }
192
         char *p = strtok(line, SEPARATOR);
193
         while (p != NULL)
194
195
```

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

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

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

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

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

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