LABORATORY ASSIGNMENT 2
Problem 2

Draghici Bogdan
The Faculty of ACE, Craiova
First Year, Calculatoare romana
CR 1.2.A

May 19, 2019

Contents

1 Problem statement

N children play the well-known ala-bala-portocala game. They are placed on a circle and start counting from 1 to k. The k-th child is removed from the game. The process continues with the rest of the children, each time counting from 1 up to k, until all the children are removed from the game.
Implement an algorithm for printing out the order in which the children are removed from the game using a circular list.

Input: Read the number N of chidren and the number k, then the n values, one for each child from the game.

Output: Write in order the children's values that are removed.

2 Pseudo-code description of the algorithms used

2.1 Creating and adding the edges of graph

In order to create the graph, I used three structures: structure graph having the number of vertexes V and array of pointers to the structure AdjList that is memorizing for each node the beginning of the list of neighbors, having as parameter the structure AdjListNode that is having as parameters the value of adjacent node and a pointer to the next adjacent one. In order to create the graph, I used the next functions: **Function** createGraph(no-nodes)

```
1. V of graph = no-nodes
2. for i=1, no-nodes, execute
3. head of array[i] of graph= NULL
4. return graph
```

Function addEdge(graph, src, dest)

```
1. adress of newNode = newAdjListNode(dest)
2. next of newNode = head of array[src] of graph
3. head of array[src] of graph = newNode
4. adress of newNode = newAdlListNode(src)
```

```
5. next of newNode = head of array[dest] of graph
6. head of array[dest] of graph = newNode
```

Function AdjListNode(info)

```
1. dest of newNode = info
2. next of newNode = NULL
3. textbfreturn node
```

The function *AdjListNode* is adding the info of a new node to the adjacency list of a node. The function *addEdge* is using the previous function to point to it's values' adress and att it to the array of the graph. The function *createGraph* just creates the graph, memorize the number of nodes of the graph and initialize de adjacency list with NULL.

2.2 Creating the Queue

In order to solve the problem I also needed to use a structure queue that has as parameters info and a pointer to the next structure of type queue, and also the following functions:
Function queue-pop(Q)

```
1. adress of poped-element = next of Q
2. aux = info of poped-element
3. info of Q = info of Q + 1
4. next of Q = next of poped-element
5. remove poped-element
6. return aux
```

Function queue-push(Q, new-value)

```
1. adress of iterator = Q
2. while next iterator != NULL
3.     iterator = next of iterator
4. last-element = iterator
```

5. next of last element = new-element
6. info of new-element = new-value
7. next of new-element = NULL

The function *queue-pop(Q)* is used to remove the first element of the queue and return its value, and the function *queue-push(Q,new-value)* adds a new element at the end of the queue that has *new-value* as info.

2.3 The main algorithm

The main algorithm used to solve the problem is contained in the next function:
Function towers(G)

```
1.  ok=1
2.  for i=0, V of G -1, execute
3.      tower-type[i]=-1
4.  next of Q=NULL
5.  for i=0, V of G -1, execute
6.      if towertype[i]=-1 do
7.          towertype[i]=1
8.      queue-push(Q,i)
9.      while next of Q!=NULL and ok!=0 do
10.         u = queue-pop(Q)
11.         v = head of array[u] of G
12.         while v!=NULL do
13.             if towertype[dest of v] = -1
14.                 towertype[dest of v] = 1 - towertype[u]
15.                 queue-push(Q,dest of v)
16.             else if towertype[dest of v] = towertype[u]
17.                 ok=0
18.             v = next of v
19. if ok=1
20.     write The towers of attack and defense are successfully created
21. else
22.     write The towers can not be created with the given tunnels
```

In order to assign the towers as attack or defense I had to create an array initialized with -1(lines 2 and 3), in which to put 1 on position i if the tower i is for defense tower, and 0 otherwise for attack tower. I also create the queue Q initialize the pointer to the next element in it to NULL(line 4).

Next I use an repetitive structure to search in all nodes their adjacent nodes(line 5). On lines 6 and 7 I assign the nodes that have not been visited yet as defence. After this, add the current node to the queue.
The while will continue to be executed while the queue is not empty and the counter ok (that tells us if we can or can not create the towers by the given rules) is not 0. The element u takes the first element in the queue, and v will take the first value of adjacency list of node u (lines 10 and 11).
Till we pass the last element from the adjacency list (line 12), we will compare if the type of tower of node from the current position in the adjacency list has not been assigned, in which condition I assign it as defense and add it to the queue to be verified also(lines 13, 14 and 15). If it has already been assigned, we compare if the types of u and current element in the adjacency list of u are the same. If so, the counter ok becomes 0(lines 16 and 17). At the end of the while, we mode to the next node stored in the adjacency list of node u (line 18).
In the end, we see if the counter remained 1, in this condition writing that the towers can be assigned by the given rules as attack and defense, and in contrary, we write that the given tunnels do not let us to do so.

3 Algorithms' memory and computational complexity

In order to find if we can or can not assign the towers as attack and defense, we have to search in adjacency lists to see how the tunnels are connecting the towers, so we have to verify for each node it's neighbors. There are n steps in order to search in all adjacency lists, and the graph is not oriented, in all the adjacency lists are 2*r elements. Each verification has O(1) complexity, so in the end we have O(n+2*r) complexity.
The memory allocated for the adjacency list is 2*r+n+n, because we have 2*r elements memorized, n NULL parameters, and n pointers for the lists For the whole graph is 1 more (2*(n+r)+1), because it also have a parameter that stores the number of node. For the queue we have n+1, because each node is verified only once and the last position is NULL. There is also the vector that stores the types of nodes *tower-type* that has n memory spaces allocated. So the memory allocated is 2*n+2*r+1+n+1+n=4*n+2*r+2.

3.1 Best case scenario

In the best case scenario, the third node found creates a cycle of odd length. In this instance, the function does n steps and verifies the first node with the second and third elements that create a cycle, then the second one in the queue, finding the third node and creating an odd cycle. In ths case, we have n+1+2+1+1=n+5 execution time.

3.2 Worst case scenario

The worse case scenario happens when all the nodes are verified and there is no odd cycle. In this case it searches in all n nodes, making in total n for adding in the queue, n for popping from the queue, and 2*r for verifying each node from adjacency list. In this case, we have 2*(n+r) execution time.

4 Experimental data

4.1 Data generation function

For the generation of the input data sets I used the rand() function from the library math.h, and also an function implemented by myself, that generates a random number from a given range.
Function random(min, max)

```
1. counter = max - min, number = 0
2. while counter != 0 do
3.     number = number * RAND_MAX + rand()
4.     counter = counter   RAND_MAX
5. return number % (max - min + 1) + min
```

RAND_MAX is a constant which represents the biggest value that can be generated using the function rand(), and it's value is 32767.
On line 2 I initialized counter with the length of the range and the variable in which the number will be created with 0. On line 3 is a repetitive structure that will be executed for log(RAND_MAX) counter times, in order to obtain a number in the range. On line 4, the number is multiplied by RAND_MAX and also added a new random value generated by the rand() function. On line 5 the counter is divided by the maximum value that can be taken by a number generated in rand().

So, when the number is multiplied by RAND_MAX, the counter is divided by RAND_MAX, in order to get to the given range as quickly as possible. After the repetitive structure ends, the number created might exceed the range length, so we must do the rest of the division with the range length+1, then add the first value of the range in order to obtain a number between min and max.

4.2 Data interpretation and graphics

Because the algorithm has complexity O(n+r), I used as abscise axis the sum between n and r.

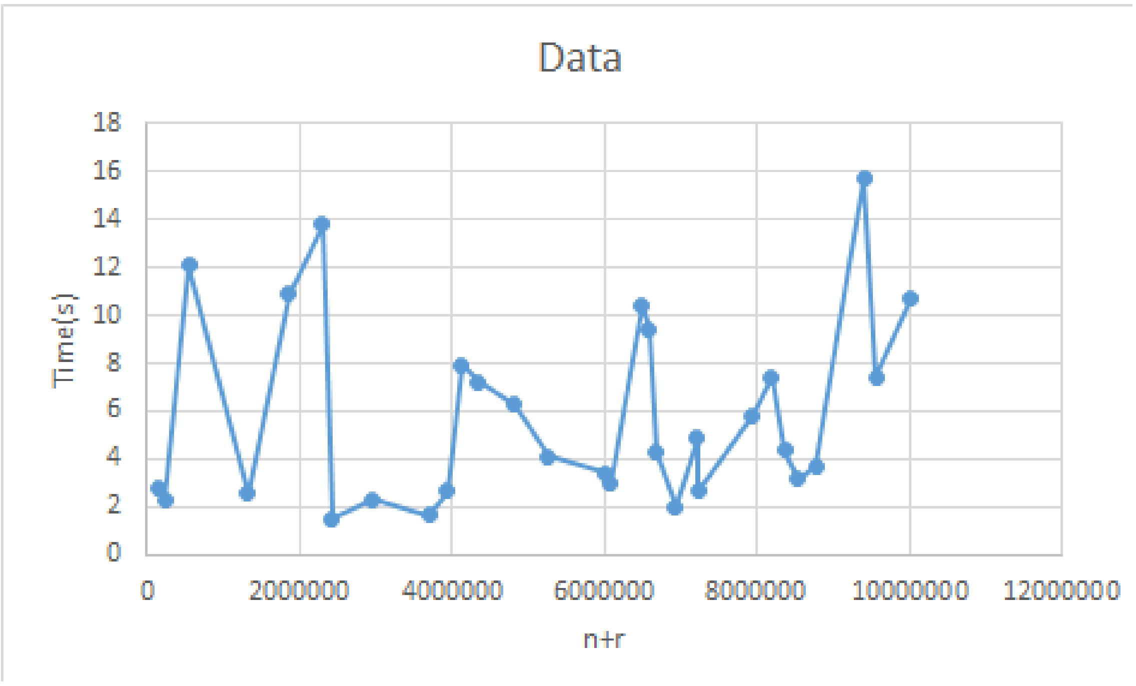

Figure 1: Data

As it can be seen, the algorithm is inconsequent, most probably because it ends when he finds an odd cycle.

5 Results and Conclusion

5.1 Summary of achievements

By making this problem, I learned how to create, read and implement a nonconex graph in a problem, using an adjacency matrix or an adjacency list, and also how to create a more compact queue.

5.2 Brief description of the most challenging achievement

The most challenging achievement was understand and implement an adjacency list for the graph I used in the algorithm. I know this will help me a lot with the next projects that will come in the future.

5.3 Future directions for extending the lab homework

I think I did the best I could at this moment and I am not intending to extend my lab homework, not until I extend my knowledge about graphs and adjacency lists.