Dynamic Software Testing

Software Quality and Dependability

Alexy Almeida	2019192123	adenis@student.dei.uc.pt
Edgar Duarte	2019216077	edgarduarte@student.dei.uc.pt
Rodrigo Ferreira	2019220060	rferreira@student.dei.uc.pt

PL1

May 2023

Contents

1 Introduction . . . 2
2 Software Risks and Issues . . . 2
3 Items and Features to be tested . . . 2
  3.1 Items . . . 2
  3.2 Features . . . 3
4 Items and Features not to be tested . . . 5
  4.1 Items . . . 5
  4.2 Features . . . 5
  4.3 Other . . . 5
5 Testing Approach . . . 5
  5.1 Control Flow Testing . . . 5
  5.2 Data Flow Testing . . . 8
  5.3 Black Box Testing . . . 16
6 Item Pass/Fail Criteria . . . 20
  6.1 Unit Testing Level . . . 20
  6.2 System Testing Level . . . 21
7 Test Deliverables . . . 21
8 Environmental Needs . . . 22
  8.1 Black Box Testing . . . 22
  8.2 GoogleTest . . . 22
9 Staffing Responsibilities . . . 22
10 Test Completion Report . . . 23
  10.1 Control Flow Testing . . . 23
  10.2 Data Flow Testing . . . 24
  10.3 Black Box Testing . . . 27

1 Introduction

This testing plan aims to perform a testing campaign for a strategy algorithm problem called "Analyzing a Data Pipeline".

The algorithm must design a data pipeline that loads a data set and performs a series of processing operations on it, until we get a final result that should be stored in a file or database. The data pipeline is represented as a network of operations where each operation depends on the former being completed first, in other words, one operation can only start once all its dependencies have finished.

The testing plan will be conducted as follows:

1. Choose the testing techniques;
2. For each testing technique, create test cases;
3. Based on the test cases created, define item pass/fail criteria;
4. Use the appropriate tools to evaluate each test case;
5. Report the expected output, output received and possible logs for each test case.

For the testing techniques, we decided to go with **2 White Box** techniques, Control Flow and Data Flow, as well as **Black Box** testing. For the Control Flow and Data Flow techniques, we chose **3 functions** from the source code that cover a good complexity and a good amount of work needed to be done for the software to execute, which helps us cover a wide variety of tests; then, for the Black Box testing, we created **32 test cases**, that will be explained in section 5, to still cover a good amount of variety without bringing too much redundancy into the tests.

The code functions we chose are as follows:

- *helper_function()* - an auxiliary function for *bottleneck()* function which helps to see if the node is a bottleneck or not;
- *recursion()* - an auxiliary function of the *minimum_amount_of_time()* function that will be responsible for calculating the maximum cost needed to reach the end node from each node.
- *minimum_amount_of_time_AND_sequence()* - displays the minimum amount of time time takes to end the pipeline.

2 Software Risks and Issues

For this concrete project, there are no critical sotware risks or issues, as it's an academic project to test the efficacy of an algorithm. The only issue we may encounter has to do with the algorithm's complexity or poor documentation, but even that isn't enough.

3 Items and Features to be tested

3.1 Items

As previously mentionned in section 1, for the White Box techniques we want to test the following functions and variables:

- *helper_function()*:
  - n;
  - pq (global);
  - visited_childs (global);
  - visited_parents (global);

    - stack_ (global);
    - degree (global);
    - childs (global);
    - parents (global);

- *recursion()*:
    - node;
    - leaf;
    - childs (global);
    - dp (global);
    - max_time;

- *minimum_amount_of_time_AND_sequence()*:
    - number_of_tasks (global);
    - degree (global);
    - visited (global);
    - time_ (global);
    - first (global);
    - childs (global);
    - pq (global);
    - parents (global);

3.2 Features

There are 4 features that need to be tested:

- Validity: Check if the pipeline is valid;
- Statistic 1: The minimum amount of time needed to run the pipeline and the feasible order to run the operations, assuming an operation can be performed one at a time;
- Statistic 2: The minimum amount of time needed to run the pipeline, assuming in infinite number of operations can be ran in parallel;
- Statistic 3: Identify the operations that are a bottleneck, which is an operation that cannot be run in parallel with others, independently of the considered scheduling.

3.2.1 Inputs

The program receives an input as follows:

- The first line contains a positive integer N that gives the number of operations;
- Operations are numbered from 1 to N;
- The next N lines describe the operations such as:
    - The first line describes operation 1, the second line operation 2, etc;
    - Each line starts with 2 positive integers, T and D;
    - The first integer corresponds to the amount of time needed to process the operation;
    - The second integer corresponds to the number of dependencies for this operation;
    - After that, D integers follow, denoting which operations need to be done before this one.
    - The last line of the input gives a integer 0, 1, 2 or 3 corresponding to the statistic that should be computed. In the case of 0, it should only check if the data pipeline is valid, otherwise it computes the corresponding statistic.

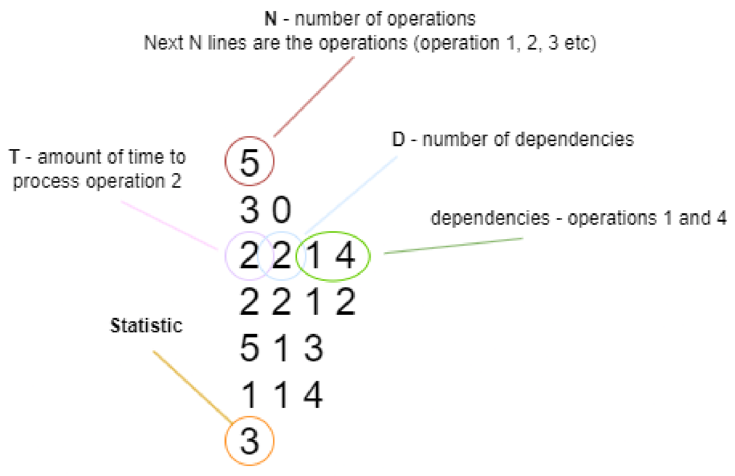

N - number of operations
Next N lines are the operations (operation 1, 2, 3 etc)
T - amount of time to process operation 2
D - number of dependencies
dependencies - operations 1 and 4
Statistic
5
3 0
2 2 1 4
2 2 1 2
5 1 3
1 1 4
3

Figure 1: Inputs explanation

3.2.2 Outputs

The program should output the following:

- If the pipeline is invalid, it should print INVALID;
- If the pipeline is valid, it should print VALID;
- For statistic 1, it should:
  - Print an integer denoting the minimum amount of time the data pipelines takes to process if only one operation can be processed at a time;
  - In the following N lines, print a feasible order for running the operations, one per line;
  - If there is more than one operation that can run at any given time, the numerically smallest should be executed, and consequently printed, first.
- For statistic 2, it should:
  - Output the minimum amount of time the pipeline takes to process if an infinite number of operations can be processed simultaneously.
- For statistic 3, it should:
  - Print the ids of the operations that are bottlenecks, one per line;
  - The ids should be printed in the order that they are processed in the data pipeline;
  - The initial and terminal operations are always bottlenecks.

3.2.3 Constraints

The program shouldn't be expected to work as intended if two variables, N and T, don't fulfill the following requirements:

- $3 \leq N \leq 1000$
- $1 \leq T \leq 100$

4 Items and Features not to be tested

4.1 Items

Based on the functions chosen for our testing plan, it's obvious to say that the following functions will not be tested:

- *check_if_one_initial_task()* - low complexity;
- *check_if_one_final_task()* - low complexity;
- *dfs(int v)* - doesn't cover a good amount of variables to be tested;
- *acyclic_and_connected_pipeline()* - doesn't cover a good amount of variables to be tested;
- *check_validity_of_pipeline()* - only 1 line of code;
- *print_childs(int n)* - the purpose is to only print values;
- *minimum_amount_of_time()* - only 3 lines of code;
- *call_childs()* - recursive function that only has 4 lines of code;
- *call_parents()* - recursive function that only has 4 lines of code;
- *am_I_a_bottleneck(int node)* - uses two recursive functions and doesn't cover a good amount of variables; also has low complexity;
- *bottleneck()* - could have been interesting to test if it wasn't for the low complexity; it also uses *helper_function(pq.top()*, which is a more interesting function to test.

As for the variables, there aren't that much that will not be covered, as most of them are used in the 3 functions that we initially selected and the probability of them being global is very high.

4.2 Features

Since the goal of software is to only print a valid or invalid output for the four specific statistics, there are many features that are irrelevant for our testing plan, such as:

- The complexity of the code;
- The time it takes for the software to run;
- Other irrelevant statistics, such as printing the number of parents or children for bottlenecks;

4.3 Other

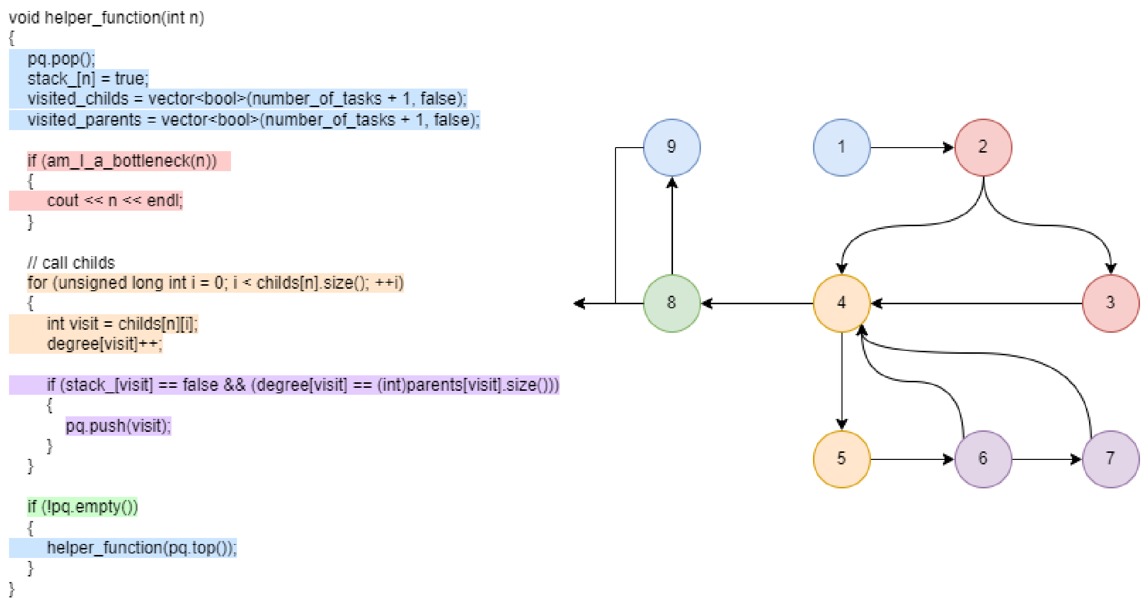
The complete list of requirements and needs of the software can be found inside the document **project_of_code**, described in section 7.

5 Testing Approach

5.1 Control Flow Testing

We used to Control Flow Testing as a testing strategy as to use the program control flow as a model. With this, we can select a set of test paths through the program that we want to cover for our testing completeness. To do this, three functions were selected: **helper_function()**, **recursion()** and **minimum_amount_of_time_AND_sequence()**.

5.1.1 helper_function()



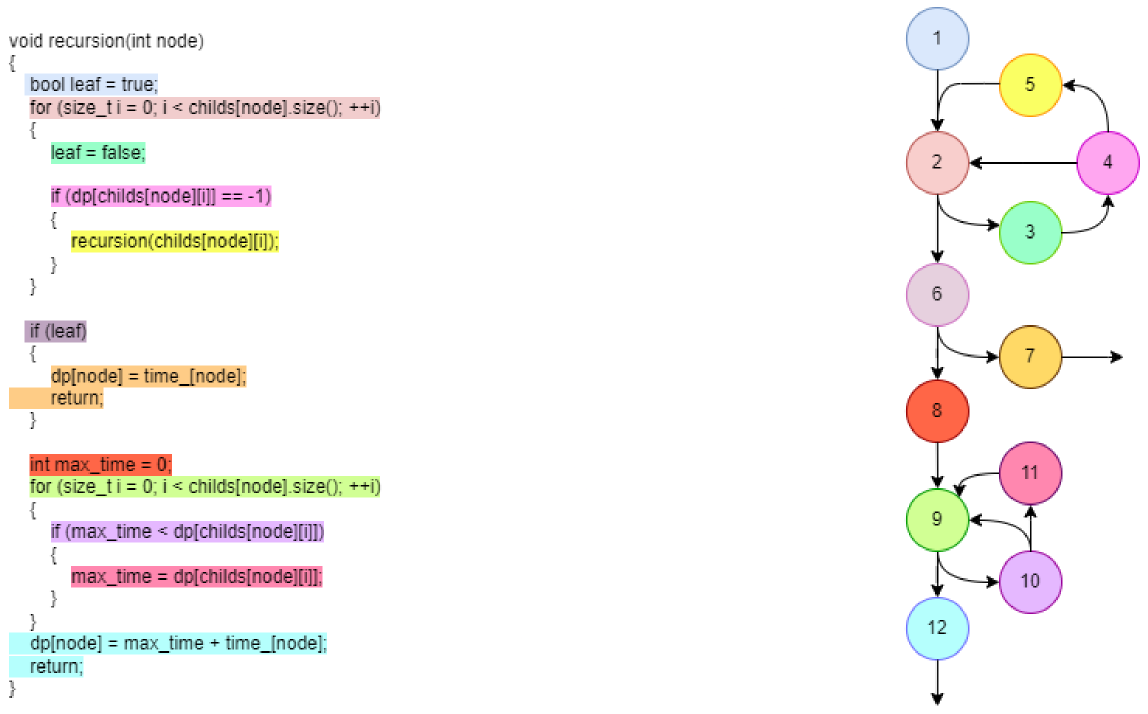
```
void helper_function(int n)
{
    pq.pop();
    stack_[n] = true;
    visited_childs = vector<bool>(number_of_tasks + 1, false);
    visited_parents = vector<bool>(number_of_tasks + 1, false);

    if (am_I_a_bottleneck(n))
    {
        cout << n << endl;
    }

    // call childs
    for (unsigned long int i = 0; i < childs[n].size(); ++i)
    {
        int visit = childs[n][i];
        degree[visit]++;

        if (stack_[visit] == false && (degree[visit] == (int)parents[visit].size()))
        {
            pq.push(visit);
        }
    }

    if (!pq.empty())
    {
        helper_function(pq.top());
    }
}
```

Figure 2: CFG of helper_function()

The cyclomatic complexity for this function is $V(G) = P + 1 = 4 + 1 = 5$, which means that there are **5 independent paths**.

Test Case	Path	pq	n	degree	childs	stack_	Feasible	Comments
TC1	1,2,3,4,8	[4]	4	[0,0,1,1,2]	[[],[2,3],[4],[4],[]]	[f, t, t, t, f]	🟩	
TC2	1,2,4,8	[3]	3	[0,0,1,1]	[[],[2,3],[],[]]	[f, t, t, f]	🟩	
TC3	1,2,4,8,9	-	-	-	-	-	🟥	Initially, pq needs to only have 1 element that gets removed in node 1 making the pq priority queue empty. As such, node 9 is unreachable.
TC4	1,2,4,5,6,4,8,9	[3,4]	3	[0,0,1,1,1]	[[],[2],[3,4],[4],[]]	[f, t, t, f,f]	🟩	
TC5	1,2,4,5,6,7,4,8,9	[2,3]	2	[0,0,1,1,0]	[[],[2,3],[4],[4],[]]	[f, t, f, f,f]	🟩	
TC6	Bypass the loop	[3]	3	[0,0,1,1,1]	[[],[2],[3],[]]	[f, t, t, f]	🟩	
TC7	Enter loop once	[2]	2	[0,0,1,0]	[[],[2],[3],[]]	[f, t, f, f]	🟩	
TC8	Enter loop twice	[1]	1	[0,0,0,0,0]	[[],[2,3],[4],[4],[]]	[f, f, f, f, f]	🟩	
TC9	Enter loop max times	[1]	1	[0,0,0,0,0]	[[], [2,3,4,5],[5],[5],[5],[]]	[f, f, f, f, f]	🟩	

5.1.2 recursion()



```
void recursion(int node)
{
   bool leaf = true;
   for (size_t i = 0; i < childs[node].size(); ++i)
   {
      leaf = false;

      if (dp[childs[node][i]] == -1)
      {
         recursion(childs[node][i]);
      }
   }

   if (leaf)
   {
      dp[node] = time_[node];
      return;
   }

   int max_time = 0;
   for (size_t i = 0; i < childs[node].size(); ++i)
   {
      if (max_time < dp[childs[node][i]])
      {
         max_time = dp[childs[node][i]];
      }
   }
   dp[node] = max_time + time_[node];
   return;
}
```

Figure 3: CFG of recursion()

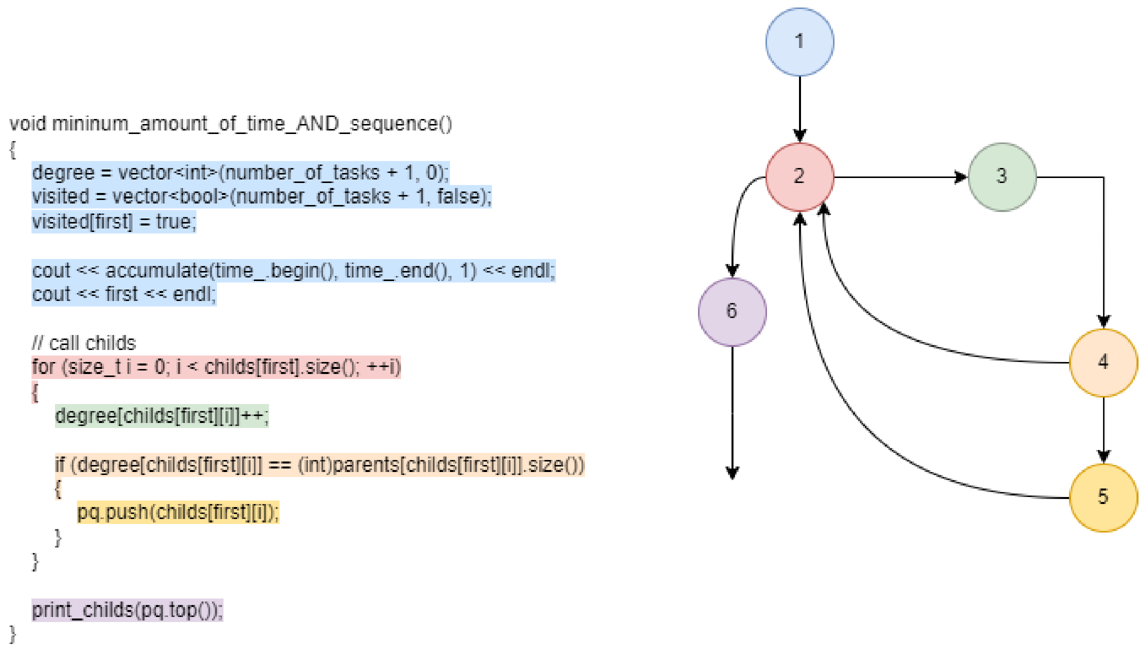
The cyclomatic complexity for this function is $V(G) = P + 1 = 5 + 1 = 6$. Due to the function having 2 exit points, there are more than 6 independent paths. We identified **8 independent paths** in total:

Test Case	Path	childs	node	dp	time_	Feasible	Comments
TC1	1,2,6,7	[[], [2,3], [4], [4], [ ]]	4	[-1,-1,-1,-1,-1]	[0,1,1,3,4]	🟩	A value of node must be chosen where the corresponding index in the childs array corresponds to an empty array
TC2	1,2,3,4,2,6,7	-	-	-	-	🟥	By passing node 3, the Boolean leaf gets set to False, never having another chance to change its value to True. In node 6, the value of the leaf variable is checked. To reach node 7, in node 6, the leaf variable needs to be True.
TC3	1,2,3,4,5,2,6,7	-	-	-	-	🟥	Same reasons as TC2.
TC4	1,2,6,8,9,12	-	-	-	-	🟥	To reach node 8, the condition on node 6 needs to be False (the variable leaf needs to be False). Seeing that node 2 turns leaf's value to True, node 8 will not be reached.
TC5	1,2,6,8,9,10,9,12	-	-	-	-	🟥	Same reason as TC4.
TC6	1,2,6,8,9,10,11,9,12	-	-	-	-	🟥	The leaf's value is forced to be True.
TC7	1,2,3,4,2,6,8,9,10,9,12	[[], [2],[3,4],[5],[5],[]]	3	[-1,-1,-1,-1,-1,-1]	[0,1,1,3,4]	🟥	We need a test case where the value of childs[node] is not empty and dp[childs[node][i]] is less than max_time = 0. Seeing that time is an array of positive numbers, and dp array gets updated by dfs - meaning that the leaf values get updated first (endpoint of node 7), when node 10 is reached - all of the children of a node already get a value associated with them, which is higher than 0. As such, unless we give the time array negative numbers, it will never be possible to have the condition in node 10 as False.
TC8	1,2,3,4,5,2,6,8,9,10,11,9,12	[[], [2], [3,4],[5],[5],[]]	3	[-1,-1,-1,-1,-1,-1]	[0, 1, 3, 2, 1, 4]	🟩	
TC9	Bypass the loop	[[], [2,3], [4], [4], [5], []]	5	[0,1,4,5,3,5]	[-1,-1,-1,-1,-1,-1]	🟩	
TC10	Enter loop once	[[], [2,3], [4], [4], [5], []]	4	[-1,-1,-1,-1,-1,-1]	[0,1,4,5,3,5]	🟩	
TC11	Enter loop twice	[[],[2],[3,4],[4],[]]	2	[-1,-1,-1,-1,-1]	[0,1,4,5,3]	🟩	
TC12	Enter loop max times	[[], [2,3,4,5],[5],[5],[5],[]]	1	[0,1,4,5,3,5]	[-1,-1,-1,-1,-1,-1]	🟩	

Loop Structure Testing

In recursion(), there are 2 loops but seeing that they have the same conditions, we only need to generate one set of test cases to test both loops.

5.1.3 mininum_amount_of_time_AND_sequence()



```
void mininum_amount_of_time_AND_sequence()
{
    degree = vector<int>(number_of_tasks + 1, 0);
    visited = vector<bool>(number_of_tasks + 1, false);
    visited[first] = true;

    cout << accumulate(time_.begin(), time_.end(), 1) << endl;
    cout << first << endl;

    // call childs
    for (size_t i = 0; i < childs[first].size(); ++i)
    {
        degree[childs[first][i]]++;

        if (degree[childs[first][i]] == (int)parents[childs[first][i]].size())
        {
            pq.push(childs[first][i]);
        }
    }

    print_childs(pq.top());
}
```

Figure 4: CFG of minimum_amount_of_time_AND_sequence()

The cyclomatic complexity for this function is $V(G) = P + 1 = 2 + 1 = 3$. The function has 1 exit point and there are **3 independent** paths.

Test Case	Path	childs	number_of_tasks	first	time_	Feasible	Comments
TC1	1,2,6	-	-	-	-	🟥	Since the minimum number of nodes is 3 and to avoid the for loop we needed exactly one node.
TC2	1,2,3,4,2,6	-	-	-	-	🟥	The loop in step 2 marks and prepares the children of the root node passed in the input and, since the input was previously evaluated in order to determine if the sequence is possible, there will be at least a child node that fulfills the condition in step 4 (the only dependency is the root node). This requires to pass in the step 5 at least once in the for loop.
TC3	1,2,3,4,5,2,6	[[], [2, 3], [4], [4], [5], []]	5	1	[0, 1, 4, 2, 2, 1]	🟩	
TC4	Bypass the loop	-	-	-	-	🟥	Minimum number of nodes is 3 and to skip the for loop there should only exist one node
TC5	Enter loop once	-	-	-	-	🟥	Minimum number of nodes is 3 and to skip the for loop there should only exist one node
TC6	Enter loop twice	[[], [3], [1, 3], []]	3	2	[0, 3, 3, 2]	🟩	The number of iterations of the for loop is related to the number os nodes specified -1
TC7	Enter loop max times	[[], [2], [3, 4], [5], [5], []]	5	1	[0, 1, 3, 2, 1, 4]	🟩	

5.2 Data Flow Testing

We used Data Flow Testing to define tests that cause the execution of a path from the point of assignment to a point where the value is used, with the goal to uncover anomalies in the flow of data in program variables.

The steps to perform this test are as follows:

- Draw a data flow graph from the program;
- Select one or more data flow testing criteria. In this case, we're only considering ADUP;
- For each variable, identify paths in the data flow graph satisfying the selection criteria (all du paths for each variable);
- Identify all complete paths needed to execute all du paths identified for all variables;
- Define test cases to execute all the complete paths identified in point 4.

5.2.1 helper_function()

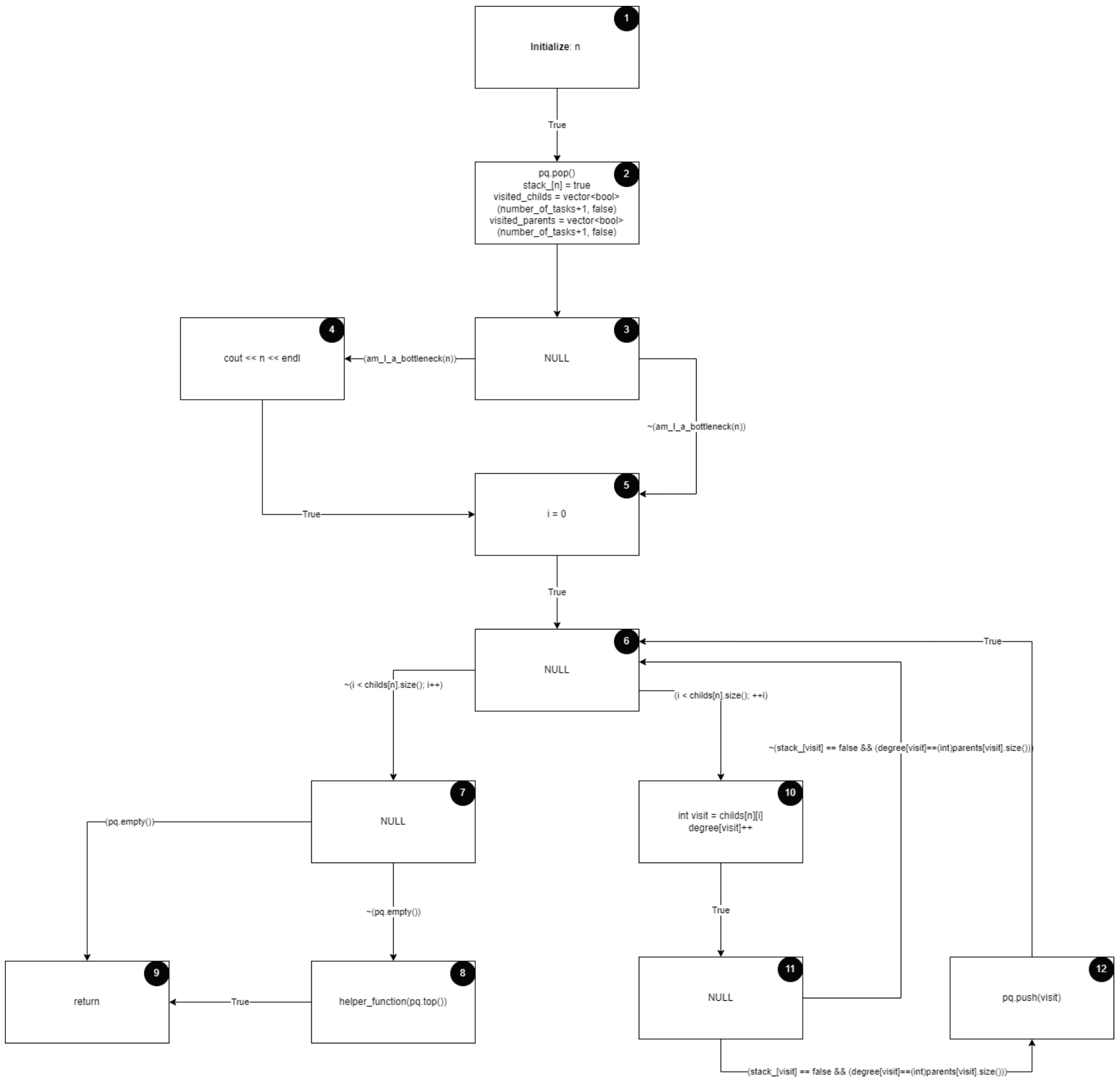

1
Initialize: n
True
2
pq.pop()
stack_[n] = true
visited_childs = vector<bool>
(number_of_tasks+1, false)
visited_parents = vector<bool>
(number_of_tasks+1, false)
3
NULL
(am_I_a_bottleneck(n))
4
cout << n << endl
~(am_I_a_bottleneck(n))
True
5
i = 0
True
6
NULL
~(i < childs[n].size(); i++)
(i < childs[n].size(); ++i)
~(stack_[visit] == false && (degree[visit]==(int)parents[visit].size()))
True
7
NULL
(pq.empty())
~(pq.empty())
10
int visit = childs[n][i]
degree[visit]++
True
9
return
True
8
helper_function(pq.top())
11
NULL
12
pq.push(visit)
(stack_[visit] == false && (degree[visit]==(int)parents[visit].size()))

Figure 5: Data flow of helper_function()

5.2.1.1 Paths

n

- ADUP1 - (1, 2, 3)
- ADUP2 - (1, 2, 3, 4)
- ADUP3 - (1, 2, 3, 4, 5, 6, 10)
- ADUP4 - (1, 2, 3, 4, 5, 6)
- ADUP5 - (1, 2, 3, 5, 6, 10)
- Complete - (1, 2, 3, 4, 5, 6, 10)

pq (global variable)

- ADUP1 - (1, 2)
- ADUP2 - (2, 3, 5, 6, 10, 11, 12)
- ADUP2 - (2, 3, 4, 5, 6, 10, 11, 12)
- ADUP3 - (12, 6, 7)
- ADUP4 - (12, 6, 7, 8)
- Complete - (1, 2, 3, 4, 5, 6, 10, 11, 12, 6, 7, 8)

visited_childs (global variable) The variable is used inside the *am_I_a_bottleneck(n)*.

- ADUP1 - (1, 2)
- ADUP2 - (2, 3)
- Complete - (1, 2, 3)

visited_parents (global variable) The variable is used inside the *am_I_a_bottleneck(n)*.

- ADUP1 - (1, 2)
- ADUP2 - (2, 3)
- Complete - (1, 2, 3)

stack_ (global variable)

- ADUP1 - (1, 2)
- ADUP2 - (2, 3, 5, 6, 10, 11)
- ADUP3 - (2, 3, 4, 5, 6, 10, 11)
- Complete - (1, 2, 3, 5, 6, 10, 11)

degree (global variable)

- ADUP1 - (1, 2, 3, 4, 5, 6, 10, 11)
- ADUP2 - (1, 2, 3, 5, 6, 10, 11)
- Complete - (1, 2, 3, 4, 5, 6, 10, 11)

childs (global variable)

- ADUP1 - (1, 2, 3, 5, 6)
- ADUP2 - (1, 2, 3, 5, 6, 10)
- Complete - (1, 2, 3, 5, 6, 10)

parents (global variable)

- ADUP1 - (1, 2, 3, 5, 6, 10, 11)
- Complete - (1, 2, 3, 5, 6, 10, 11)

Variable	Test Case	Complete Path	pq	n	degree	childs	stack_
n	**TC1**	(1, 2, 3, 4, 5, 6, 10)	[2]	2	[0,0,1,0]	[[],[2],[3],[]]	[f, t, f, f]
pq	**TC2**	(1, 2, 3, 4, 5, 6, 10, 11, 12, 6, 7, 8)	[1]	1	[0,0,0,0,0]	[[],[2,3],[4],[4],[]	[f, f, f, f, f]
visited_childs	**TC3**	(1, 2, 3)	[3]	3	[0,0,1,2]	[[],[2,3],[3],[]]	[f,t,t,f]
visited_parents	**TC4**	(1, 2, 3)	[3]	3	[0,0,1,2]	[[],[2,3],[3],[]]	[f,t,t,f]
stack_	**TC5**	(1, 2, 3, 5, 6, 10, 11)	[2,3]	2	[0,0,1,1,0]	[[],[2,3],[4],[4],[]]	[f,t,f,f,f]
degree	**TC6**	(1, 2, 3, 4, 5, 6, 10, 11)	[1]	1	[0,0,0,0,0]	[[],[2,3],[4],[4],[]]	[f,f,f,f,f]
childs	**TC7**	(1, 2, 3, 5, 6, 10)	[2,3]	2	[0,0,1,1,0]	[[],[2,3],[4],[4],[]]	[f,t,f,f,f]
parents	**TC8**	(1, 2, 3, 5, 6, 10, 11)	[1]	1	[0,0,0,0,0]	[[],[2,3],[4],[4],[]]	[f,f,f,f,f]

5.2.2 recursion()

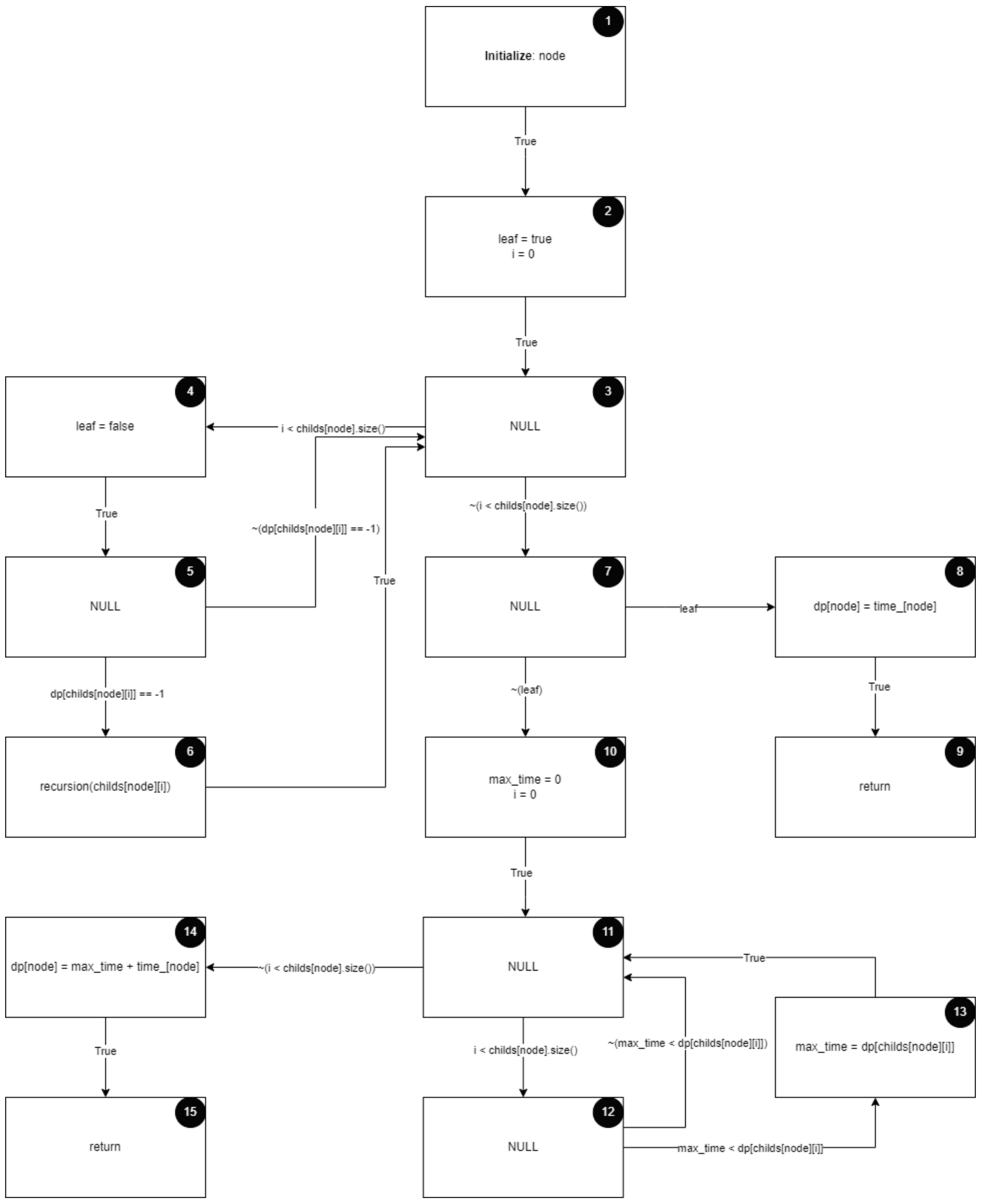

1
Initialize: node
True
2
leaf = true
i = 0
True
3
NULL
i < childs[node].size()
4
leaf = false
True
5
NULL
~(dp[childs[node][i]] == -1)
dp[childs[node][i]] == -1
6
recursion(childs[node][i])
True
~(i < childs[node].size())
7
NULL
leaf
8
dp[node] = time_[node]
True
9
return
~(leaf)
10
max_time = 0
i = 0
True
11
NULL
~(i < childs[node].size())
14
dp[node] = max_time + time_[node]
True
15
return
i < childs[node].size()
12
NULL
~(max_time < dp[childs[node][i]])
max_time < dp[childs[node][i]]
13
max_time = dp[childs[node][i]]
True

Figure 6: Data flow of recursion()

5.2.2.1 Paths

node

- ADUP1 - (1, 2, 3)
- ADUP2 - (1, 2, 3, 4, 5)

- ADUP3 - (1, 2, 3, 4, 5, 6)
- ADUP4 - (1, 2, 3, 7, 8)
- ADUP5 - (1, 2, 3, 7, 10, 11)
- ADUP6 - (1, 2, 3, 7, 10, 11, 12)
- ADUP7 - (1, 2, 3, 7, 10, 11, 12, 13)
- ADUP8 - (1, 2, 3, 7, 10, 11, 14)

leaf

- ADUP1 - (2, 3, 7)
- ADUP2 - (2, 3, 4)
- ADUP3 - (4, 5, 3, 7)
- Complete - (2, 3, 4, 5, 3, 7)

childs (global variable)

- ADUP1 - (1, 2, 3)
- ADUP2 - (1, 2, 3, 4, 5)
- ADUP3 - (1, 2, 3, 4, 5, 6)
- ADUP4 - (1, 2, 3, 7, 10, 11)
- ADUP5 - (1, 2, 3, 7, 10, 11, 12)
- ADUP6 - (1, 2, 3, 7, 10, 11, 12, 13)
- Complete - (1, 2, 3, 4, 5, 6, 3, 7, 10, 11, 12, 13)

dp (global variable)

- ADUP1 - (1, 2, 3, 4, 5)
- ADUP2 - (1, 2, 3, 4, 5, 6)
- ADUP3 - (1, 2, 3, 7, 8)
- ADUP4 - (1, 2, 3, 7, 10, 11, 12, 13)
- ADUP5 - (1, 2, 3, 7, 10, 11, 14)
- Complete - (1, 2, 3, 4, 5, 6, 3, 7, 10, 11, 12, 13, 14) and (1, 2, 3, 7, 8).

max_time

- ADUP1 - (10, 11, 12)
- ADUP2 - (10, 11, 12, 13)
- ADUP3 - (10, 14)
- Complete - (10, 11, 12, 13, 10, 14)

Variable	Test Case	childs	node	dp	time_
node	**TC1**	[[ ],[2,3],[4],[4],[]]	1	[-1,-1,-1,-1,-1]	[0,2,7,4,3]
node	**TC2**	[[ ],[2,3],[4],[4],[]]	4	[-1,-1,-1,-1,-1]	[0,2,7,4,3]
leaf	**TC3**	[[ ],[2,3],[4],[4],[]]	3	[1,-1,10,-1,5]	[0,1,5,3,5]
childs	**TC4**	[[ ],[2,3],[3,4],[4],[]]	2	[-1,-1,-1,-1,-1]	[0,1,5,3,5]
dp	**TC5**	[[],[2,3],[4],[4],[]]	2	[-1,-1,-1,-1,-1]	[0, 1, 3, 4, 3]
dp	**TC6**	[[],[2,3],[4],[4],[]]	4	[0, 1, 3, 4, 3]	[-1,-1,-1,-1,-1]
max_time	**TC7**	[[], [2,3],[4],[4],[]]	1	[-1,-1,-1,-1,-1]	[0,1,3,4,3]

5.2.3 mininum_amount_of_time_AND_sequence()

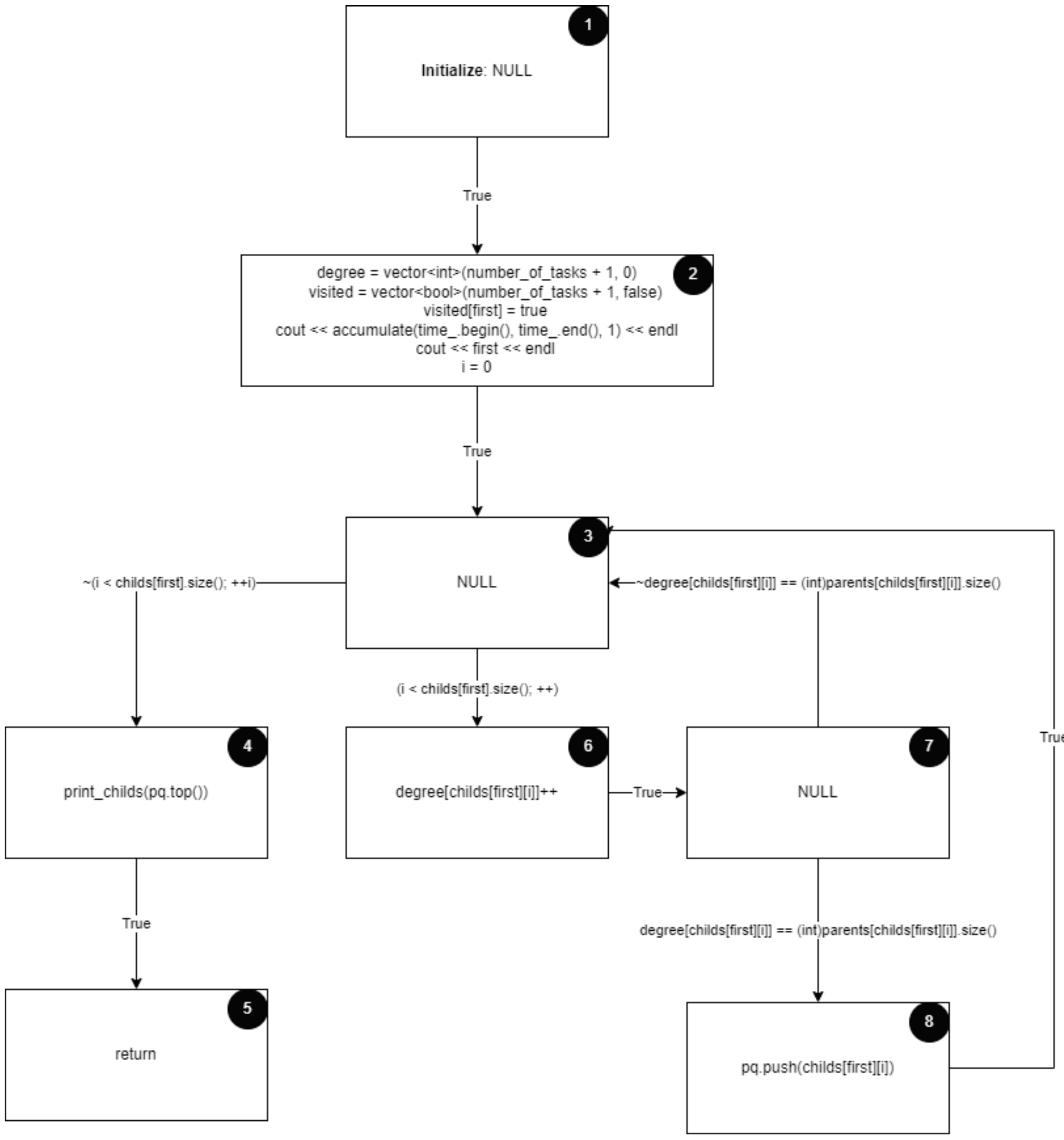

1
Initialize: NULL
True
2
degree = vector<int>(number_of_tasks + 1, 0)
visited = vector<bool>(number_of_tasks + 1, false)
visited[first] = true
cout << accumulate(time_.begin(), time_.end(), 1) << endl
cout << first << endl
i = 0
True
3
NULL
~(i < childs[first].size(); ++i)
~degree[childs[first][i]] == (int)parents[childs[first][i]].size()
(i < childs[first].size(); ++)
True
4
print_childs(pq.top())
6
degree[childs[first][i]]++
True
7
NULL
True
degree[childs[first][i]] == (int)parents[childs[first][i]].size()
5
return
8
pq.push(childs[first][i])

Figure 7: Data flow of mininum_amount_of_time_AND_sequence()

5.2.3.1 Paths

number_of_tasks (global variable)

- ADUP1 - (1, 2)
- Complete - (1, 2)

degree (global variable)

- ADUP1 - (1, 2)
- ADUP2 - (2, 3, 4)
- ADUP3 - (2, 3, 6)
- ADUP4 - (6, 7)
- Complete - (1, 2, 3, 6, 7, 8, 3, 4)

visited (global variable)

- ADUP1 - (1, 2)
- ADUP2 - (2, 3, 4)
- Complete - (1, 2, 3, 4)

time_ (global variable)

- ADUP1 - (1, 2)
- Complete - (1, 2)

first (global variable)

- ADUP1 - (1, 2, 3)
- ADUP2 - (1, 2, 3, 6, 7)
- ADUP3 - (1, 2, 3, 6, 7, 8)
- Complete - (1, 2, 3, 6, 7, 8)

childs (global variable)

- ADUP1 - (1, 2, 3)
- ADUP2 - (1, 2, 3, 6, 7)
- ADUP3 - (1, 2, 3, 6, 7, 8)
- Complete - (1, 2, 3, 6, 7, 8)

pq (global variable)

- ADUP1 - (1, 2, 3, 4)
- ADUP2 - (1, 2, 3, 6, 7, 8)
- Complete - (1, 2, 3, 6, 7, 8, 3, 4)

parents (global variable)

- ADUP1 - (1, 2, 3, 4)
- ADUP1 - (1, 2, 3, 6, 7)
- Complete - (1, 2, 3, 6, 7, 8, 3, 4)

Variable	Test Case	Complete Path	childs	first	number_of_tasks	time_
number_of_tasks	**TC1**	(1, 2)	[[],[4,5],[1,3,4],[5],[],[4]]	2	5	[0,2,7,4,3,2]
degree	**TC2**	(1, 2, 3, 6, 7, 8, 3, 4)	[[],[2,3],[4],[4,6],[5],[],[5]]	1	6	[0,2,5,2,8,1,3]
visited	**TC3**	(1, 2, 3, 4)	[[],[2,4],[3],[4],[]]	1	4	[0,4,2,1,1]
time_	**TC4**	(1, 2)	[[],[2],[3],[],[1,2]]	4	4	[0,5,3,4,1]
first	**TC5**	(1, 2, 3, 6, 7, 8)	[[],[2],[],[5],[2],[1,2,4]]	3	5	[0,2,3,4,1,5]
childs	**TC6**	(1, 2, 3, 6, 7, 8)	[[],[5],[1,5],[2],[3],[]]	4	5	[0,6,5,4,6,1]
pq	**TC7**	(1, 2, 3, 6, 7, 8, 3, 4)	[[],[2],[4],[1,2,4,5],[],[4]]	3	5	[0,4,2,3,5,8]
parents	**TC8**	(1, 2, 3, 6, 7, 8, 3, 4)	[[],[2,3],[5,6],[4],[6],[6],[]]	1	6	[0,3,3,2,2,5,2]

5.3 Black Box Testing

In order to perform an efficient black box testing, we grouped the test cases into the following categories:

	D	**T**	**N**	**stat0**	**stat1**	**stat2**	**stat3**	**dependencies**
valid	TC1, TC2	TC3, TC4	TC5, TC6	TC7, TC8	TC9, TC10	TC11, TC12	TC13, TC14	TC15, TC16
invalid	TC17, TC18	TC19, TC20	TC21, TC22	TC23, TC24	TC25, TC26	TC27, TC28	TC29, TC30	TC31, TC32

First, we separate tests that should print a valid output from tests that should print an invalid output. Then, for each type of valid or invalid output, we want to test different categories of input that are as follows:

- D - test if the number of dependencies is accurate. e.g: D = 2 should be followed by 2 operations.
- T - test if the amount of time to process operation x is valid. e.g: T = 2000 is a too high number for the test to be valid.
- N - test if the number of operations corresponds the number of N lines. e.g: N = 10 should have 10 lines, each representing an opeation.
- stat0 - test if the pipeline is valid. e.g: if the pipeline is invalid, it should print INVALID.
- stat1 - test if the statistic1 is valid. e.g: if the statistic1 is valid, it prints several lines of integers (explained in section 3)
- stat2 - test if the statistic2 is valid. e.g: if the statistic2 is valid, it outputs an integer.
- stat3 - test if the statistic3 is valid. e.g: if the statistic3 is valid, it outputs several lines of integers (explained in section 3)
- dependencies - test if the dependencies are correct. e.g: if a node 4 has parent 3 and 5, it the operation 4 should have operations 3 and 5 as dependencies.

Below are the inputs for each test case:

5.3.1 TC1

```
5
2 1 2
3 0
2 1 2
1 1 5
5 2 1 3
3
```

5.3.2 TC2

```
6
21 0
56 2 5 3
23 1 4
84 1 1
6 1 1
84 5 1 4 5 3 2
1
```

5.3.3 TC3

```
11
10 0
25 1 1
20 1 5
80 1 5
10 1 2
10 2 4 3
5 1 1
40 1 7
3 2 8 10
17 1 11
30 1 6
1
```

5.3.4 TC4

```
5
80 0
2 1 1
50 1 1
5 2 2 3
1 1 4
1
```

5.3.5 TC5

```
3
1 0
1 1 1
1 2 2 1
3
```

5.3.6 TC6

```
11
10 0
30 1 1
20 1 5
40 1 5
10 1 2
10 2 4 3
20 1 1
40 1 7
20 2 8 10
20 1 11
30 1 6
1
```

5.3.7 TC7

```
5
3 0
2 1 1
2 1 1
5 2 3 2
1 1 4
0
```

5.3.8 TC8

```
5
3 0
2 1 1
2 1 1
5 2 2 3
1 1 4
0
```

5.3.9 TC9

```
6
21 0
56 2 5 3
23 1 4
84 1 1
6 1 1
84 5 1 4 5 3 2
1
```

5.3.10 TC10

```
6
21 0
56 2 5 3
23 1 4
63 1 1
6 1 1
63 5 1 4 5 3 2
1
```

5.3.11 TC11

```
5
3 0
2 1 1
2 1 1
5 2 2 3
1 1 4
2
```

5.3.12 TC12

```
13
1 0
1 1 1
1 1 2
1 1 2
1 1 3
4 1 4
1 1 5
1 1 5
1 2 7 8
1 2 9 6
1 1 10
1 1 10
1 2 11 12
2
```

5.3.13 TC13

```
6
1 1 2
1 1 6
1 1 1
1 1 5
1 1 3
1 0
3
```

5.3.14 TC14

```
5
2 1 2
3 0
2 1 2
1 1 5
5 2 1 3
3
```

5.3.15 TC15

```
6
10 0
10 1 1
10 1 2
10 1 3
10 2 2 4
10 1 5
3
```

5.3.16 TC16

```
21
30 1 3
20 1 3
10 0
40 2 3 5
42 1 10
20 1 21
10 1 2
10 1 12
18 1 5
15 2 6 1
30 1 2
20 1 2
20 3 7 8 11
13 1 5
40 2 5 4
15 4 21 9 14 15
17 1 15
40 1 20
100 2 18 20
20 2 16 17
10 1 13
3
```

5.3.17 TC17

```
5
2 1 2
3 0
2 1 2 3 5
1 1 5
5 2 1 3
3
```

5.3.18 TC18

```
6
21 0
56 2 5 3
23 1 4
84 1 1
6 1 1
84 9 1 4 5 3 2
1
```

5.3.19 TC19

```
11
10 0
25 1 1
20 1 5
80 1 5
10 1 2
10 2 4 3
5 1 1
119 1 7
3 2 8 10
17 1 11
30 1 6
1
```

5.3.20 TC20

```
13
1 0
1 1 1
1 1 2
1 1 2
1 1 3
200 1 4
1 1 5
1 1 5
1 2 7 8
1 2 9 6
1 1 10
1 1 10
1 2 11 12
2
```

5.3.21 TC21

```
20
10 0
30 1 1
20 1 5
40 1 5
10 1 2
10 2 4 3
20 1 1
40 1 7
20 2 8 10
20 1 11
30 1 6
1
```

5.3.22 TC22

```
2
1 0
1 1 1
0
```

5.3.23 TC23

```
5
3 0
2 1 3
4 1 2
3 0
0
```

5.3.24 TC24

```
4
3 0
2 1 1
4 1 2
3 0
0
```

5.3.25 TC25

```
5
3 0
2 1 1
2 1 1
5 2 2 3
1 1 4
1
```

5.3.26 TC26

```
6
21 0
56 2 5 3
23 1 3
63 1 1
6 1 1
63 5 1 4 5 3 2
1
```

5.3.27 TC27

```
6
3 0
2 1 1
2 1 1
5 2 2 3
1 1 4
2 1 4
2
```

5.3.28 TC28

```
13
1 0
1 1 1
1 1 2
1 1 2
1 1 1
4 1 4
1 1 5
1 1 2
1 2 7 8
1 2 9 6
1 1 10
1 1 10
1 2 11 12
2
```

5.3.29 TC21

```
20
10 0
30 1 1
20 1 5
40 1 5
10 1 2
10 2 4 3
20 1 1
40 1 7
20 2 8 10
20 1 11
30 1 6
1
```

5.3.30 TC22

```
2
1 0
1 1 1
0
```

5.3.31 TC23

```
5
3 0
2 1 3
4 1 2
3 0
0
```

5.3.32 TC24

```
4
3 0
2 1 1
4 1 2
3 0
0
```

5.3.33 TC29

```
5
3 0
2 2 1 4
2 2 1 2
5 1 3
1 1 4
3
```

5.3.34 TC30

```
5
3 0
2 2 1 4
2 2 1 3
5 1 3
1 1 4
3
```

5.3.35 TC31

```
21
30 1 3
20 1 3
10 0
40 2 3 5
42 1 10
20 1 14
10 1 2
10 1 12
18 1 5
15 2 6 1
30 1 2
20 1 2
20 3 7 8 13
13 1 5
40 2 5 4
15 4 21 9 14 18
17 1 15
40 1 20
100 2 18 20
20 2 16 17
10 1 13
3
```

5.3.36 TC32

```
6
10 0
10 1 1
10 1 5
10 1 3
10 2 2 4
10 1 5
3
```

6 Item Pass/Fail Criteria

6.1 Unit Testing Level

For this level, a test can be considered successful if:

- The output corresponds to the expected output;

If, and only if, all of the tests are successful, we can say that the software works as intended for that specific purpose.

6.2 System Testing Level

For this level, a test can be considered successful if:

- The output corresponds to the expected output;
- There is a log or comment that justifies why the output doesn't correspond to the expected output, which can go from a variety of reasons, such as the nature of the project, some specifications not mentioned in the project or other.

If there is at least 90% of successful test cases, we can say that the software works as intended for that specific purpose.

7 Test Deliverables

During this report, the following documents were created inside the project repository:

- Assignment
  - file **project_of_code** that explains the origin of the code problem;
  - file **proposal** that briefly explain the link between the **project_of_code** file and this assignment.
- Testing Techniques
  - folder *black_box* containing the test cases for that techniques, which are the valid/invalid inputs for each category and the produced outputs;
  - folder *diagrams* that contains the diagrams manually produced for the control flow and data flow testing techniques;
- Code
  - file **howtorun** that explains how to compile and run the code for the black box testing;
  - file **code.cpp** that contains the source code of the software we tested;
  - file **code.exe**, which allows to execute the software;
  - file **test.cpp** that contains the setup and tests of the software using the gtest library.
  - file **test.exe** is an executable file used to run the tests.
- Report
  - Test plan;
  - Test cases;
  - Environmental needs for black box and white box testing;
  - Inputs;
  - Test criterias;
  - Test completion report: outputs expected, outputs produced and logs.

8 Environmental Needs

8.1 Black Box Testing

To compile the project code for the black box testing, we first need to install **cygwin** with some packages, which are as follows:

- gcc-core
- gcc-g++
- make
- gdb

```
g++ -o code code.cpp
```

Finally, to test any test case, we add one following lines in the bash command line, for either a valid or invalid input:

```
./code.exe < black_box/test_cases/inputs/valid/test.input > test_cases/
    outputs/test.output
./code.exe < black_box/test_cases/inputs/invalid/test.input > test_cases/
    outputs/test.output
```

The ”test.input” file contains the input of the test case and the ”test.out” file will give us the output of that same test case.

8.2 GoogleTest

Since the code includes the **gtest** library from Google, we need to download the code from `https://github.com/google/googletest/tree/release-1.10.0` and add it to the include folder on gcc. In order to test the *code.cpp* file we created an auxiliary file (*test.cpp)* that will perform the tests on the code. To compile the test file, use the following command:

```
g++ -o test.exe test.cpp -lgtest -lgtest_main -pthread
```

To execute the program, we just call the executable file and the tests will be performed, giving important feedback to the user.

9 Staffing Responsibilities

This software testing project work was divided between the three members mentionned in the cover of this report. The work was divided as follows:

- Choose the code to be tested, while paying attention to possible constraints that could be encountered;
- Choose the testing techniques and tools to be used;
- For each testing technique, divide the diagram creation between each member, for one function of the code each:
  - Member 1: does CFG and data flow for *helper_function()*
  - Member 2: does CFG and data flow for *recursion()*
  - Member 3: does CFG and data flow for *minimum_amount_of_time_AND_sequence()*
- For each CFG, divide once again the creation of paths and test cases:
  - Member 1: calculate V(G), create independent paths, and create test cases for *helper_function()*

  - Member 2: calculate V(G), create independent paths, and create test cases for *recursion()*
  - Member 3: calculate V(G), create independent paths, and create test cases for *minimum_amount_of_time_AND_sequence()*

- For each data flow, divide once again the creation of paths and test cases:
  - Member 1: create ADUP, complete paths, and create test cases for *helper_function()*
  - Member 2: create ADUP, complete paths, and create test cases for *recursion()*
  - Member 3: create ADUP, complete paths, and create test cases for *minimum_amount_of_time_AND_sequence()*
- One of the members creates the test cases for the **black box** testing;
- The item pass/fail criteria are defined based on the test cases created;
- All of the members write the expected outputs for the test cases they created;
- Each member tests each test case with the specific tools;
- All of the members create multiple condensed tables for all the test cases produced with their respective expected outputs and true outputs.

10 Test Completion Report

10.1 Control Flow Testing

10.1.1 helper_function()

Test Case	Expected output	Output	Logs/Comments	Pass/Fail
TC1	4	4	OK	🟩
TC2	No output	No output	OK	🟩
TC4	2 4	2 3 4	FAIL	🟥
TC5	4	4	OK	🟩
TC6	3	3	OK	🟩
TC7	2 3	2 3	OK	🟩
TC8	1 4	1 4	OK	🟩
TC9	1 5	1 5	OK	🟩

According to section 6, the software **fails** the criteria.

10.1.2 recursion()

Test Case	Expected output	Output	Logs/Comments	Pass/Fail
TC1	[-1,-1,-1,-1,4]	[-1,-1,-1,-1,4]	OK	🟩
TC8	[-1,-1,-1,6,-1,4]	[-1,-1,-1,6,-1,4]	OK	🟩
TC9	[-1,-1,-1,-1,-1,5]	[-1,-1,-1,-1,-1,5]	OK	🟩
TC10	[-1,-1,-1,-1,8,5]	[-1,-1,-1,-1,8,5]	OK	🟩
TC11	[-1,-1,12,8,3]	[-1,-1,12,8,3]	OK	🟩
TC12	[-1,11,9,10,8,5]	[-1,11,9,10,8,5]	OK	🟩

According to section 6, the software **passes** the criteria.

10.1.3 minimum_amount_of_time_AND_sequence()

Test Case	Expected output	Output	Logs/Comments	Pass/Fail
TC3	11 1 2 3 4 5	11 1 2 3 4 5	OK	🟩
TC6	9 2 1 3	9 2 1 3	OK	🟩
TC7	12 1 2 3 4 5	12 1 2 3 4 5	OK	🟩

According to section 6, the software **passes** the criteria.

10.2 Data Flow Testing

10.2.1 helper_function()

Test Case	Expected output	Output	Logs/Comments	Pass/Fail
TC1	2 3	2 3	OK	🟩
TC2	1 4	1 4	OK	🟩
TC3	3	3	OK	🟩
TC4	3	3	OK	🟩
TC5	4	4	OK	🟩
TC6	1 4	1 4	OK	🟩
TC7	4	4	OK	🟩
TC8	1 4	1 4	OK	🟩

According to section 6, the software **passes** the criteria.

10.2.2 recursion()

Test Case	Expected output	Output	Logs/Comments	Pass/Fail
TC1	[-1,12,10,7,3]	[-1,12,10,7,3]	OK	🟩
TC2	[-1,-1,-1,-1,3]	[-1,-1,-1,-1,3]	OK	🟩
TC3	[-1,-1,10,8,5]	[-1,-1,10,8,5]	OK	🟩
TC4	[-1,-1,13,8,5]	[-1,-1,13,8,5]	OK	🟩
TC5	[-1,-1,6,-1,3]	[-1,-1,6,-1,3]	OK	🟩
TC6	[-1,-1,-1,-1,3]	[-1,-1,-1,-1,3]	OK	🟩
TC7	[-1,8,6,7,3]	[-1,8,6,7,3]	OK	🟩

According to section 6, the software **passes** the criteria.

10.2.3 minimum_amount_of_time()

Test Case	Expected output	Output	Logs/Comments	Pass/Fail
TC1	19 2 1 3 5 4	19 2 1 3 5 4	OK	🟩
TC2	22 1 2 3 4 6 5	22 1 2 3 4 6 5	OK	🟩
TC3	9 1 2 3 4	9 1 2 3 4	OK	🟩
TC4	14 4 1 2 3	14 4 1 2 3	OK	🟩
TC5	14 4 1 2 3	14 4 1 2 3	OK	🟩
TC6	23 4 3 2 1 5	23 4 3 2 1 5	OK	🟩
TC7	23 3 1 2 5 4	23 3 1 2 5 4	OK	🟩
TC8	18 1 2 3 4 5 6	18 1 2 3 4 5 6	OK	🟩

According to section 6, the software **passes** the criteria.

10.3 Black Box Testing

10.3.1 Valid Inputs

Test Case	Expected output	Output	Logs/Comments	Pass/Fail
TC1	List of integers	List of integers	None	🟩
TC2	List of integers	List of integers	None	🟩
TC3	List of integers	List of integers	None	🟩
TC4	List of integers	List of integers	None	🟩
TC5	List of integers	List of integers	None	🟩
TC6	List of integers	List of integers	None	🟩
TC7	VALID	VALID	None	🟩
TC8	VALID	VALID	None	🟩
TC9	List of integers	List of integers	None	🟩
TC10	List of integers	List of integers	None	🟩
TC11	One integer	One integer	None	🟩
TC12	One integer	One integer	None	🟩
TC13	List of integers	List of integers	None	🟩
TC14	List of integers	List of integers	None	🟩
TC15	List of integers	List of integers	None	🟩
TC16	List of integers	List of integers	None	🟩

According to section 6, the software **passes** the criteria.

10.3.2 Invalid Inputs

Test Case	Expected output	Output	Logs/Comments	Pass/Fail
TC17	INVALID	Empty file	None	🟥
TC18	INVALID	Empty file	None	🟥
TC19	INVALID	List of integers	None	🟥
TC20	INVALID	One integer	None	🟥
TC21	INVALID	INVALID	None	🟩
TC22	INVALID	VALID	None	🟥
TC23	INVALID	INVALID	None	🟩
TC24	INVALID	INVALID	None	🟩
TC25	INVALID	List of integers	None	🟥
TC26	INVALID	INVALID	None	🟩
TC27	INVALID	INVALID	None	🟩
TC28	INVALID	INVALID	None	🟩
TC29	INVALID	INVALID	None	🟩
TC30	INVALID	INVALID	None	🟩
TC31	INVALID	List of integers	None	🟥
TC32	INVALID	One integer	None	🟥

According to section 6, the software **fails** the criteria.