QCS TP2
Dynamic Software Testing

Gonçalo São Marcos
2018284198
Universidade de Coimbra
gmarcos@student.dei.uc.pt

José Esperança
2018278596
Universidade de Coimbra
esperanca@student.dei.uc.pt

May 30, 2022

1 Introduction

The main goal of this practical assignment is to develop a software test plan and perform the testing necessary for a software code that we picked out from the internet. The software in particular that we selected was first proposed to the professor João Fernandes among other pieces of software. The teacher suggested that we went ahead and tested this one since it revealed a good level of challenge and possibilities to explore and learn more about software testing. The program was selected from the following GeeksForGeeks Link and its purpose is to find the maximum size sub matrix with all 1s in a binary matrix. It contains a good balance of *for* and *if* statements and it is also coded in a way that we can divide it into separate functions that each have their own specific task within the global objective and can be studied in separate.

We are testing this code with the objective of looking for bugs that will cause the program to output wrong results or even input states that will outright make the program not work (i.e. crash). Our plan is to follow a dynamic approach to the unit testing and, for that, we first started out, as we mentioned, by dividing the code into three separate functions, each of them with its own goal inside the global objective of the whole code. For each function we then generated test cases for each kind of test: black-box and white-box, as well as control flow and data flow testing for the latter.

To finish off the testing suite, we analysed the results obtained and elaborated a conclusion to the testing that was performed.

2 Software risk issues

Straight away one of the risks we identified was the fact that the whole code as a whole would be too complex to test since it involved many nested *for* loops in a row (a normal programming strategy to observe when dealing with matrices). As a solution to that issue we started out by dividing the whole code into three separate functions that would execute sequentially (i.e. one after the other) and, in that way, we simplified the process of testing the code by looking at smaller distinct snippets, each with its own objective. Another risk we identified was the nature of the functions we had just created: they all run sequentially; this means that the output of the first function directly affects the input of the second function and so on. Well, if the first function outputs the wrong result it will affect how the second function behaves and might even make it crash if the output of the previous function somehow does not match the input criteria of the next one. In the testing that we will perform, we always assume that the previous function outputs a result that is correct and that allows us to independently test each of the functions without having to worry about if the previous function returned an invalid parameter. It would be possible to test the software considering the dependency of the functions but we believe that it is not in our best interest to test all those combinations since the scope of this project is to learn the most we can about dynamic software testing, not get stuck in a small nuance of this particular code.

We should always consider the impact of imperfect testing in the results of our work, that is, it is likely that an error in the function's code will lead to a wrong final result but that risk itself is very low and that behaviour will never cause a crash, it will only output a wrong result. The only way we identified that crashed could occur is if there were errors when handling the number of lines and columns of the matrices which is directly associated with the standard function *len()* of Python.

3 Items and features to be tested

Like we mentioned in **Sections 1** and **2**, we divided the code into three sections and we will analyse all of them except for the else du-paths in the data-flow of the *"createAuxMatrix"* function as suggested by professor João since it would lead to a very high complexity (16 paths) and it would not add much to our learning on the subject. In the following list we detail the separate functions that will be tested and their task in the global objective of the code.

Function	**Task**
CreateAuxMatrix()	Creates the Auxiliary Matrix, it is initialized with the first row and first column equal to the input matrix
FillAux()	Fills the Auxiliary Matrix, adding to each cell as it finds possible corners to Sub-Matrixes of 1s
FindMax()	Searches for the maximum value and its coordinates in the Auxiliary Matrix

Table 1: Functions to be tested

4 Items and features not to be tested

For the items and features not to be tested we already discussed in **Section 3** one of the items that we will not test: the else du-paths in the data-flow of the *"createAuxMatrix"* function as suggested by professor João since it would lead to a very high complexity (16 paths) and it would not add much to our learning on the subject. We also discarded the testing of the *printMatrix()* function whose purpose was just to print the final Sub-Matrix that the program had found since this is a low risk, low complexity and low interest part of the code.

5 Testing Approach

As mentioned in **Section 2**, the functions are ran in a sequential way so we always assume that the previous output is valid and correct. We will **not** test incoherent values and we will assume that the default suite of Python functions is implemented correctly (i.e. functions like *len()* return the correct value for the input provided).

An overview of the testing plan is shown in **Table 2**. With the testing plan set it is important to state that every section should be tested with more than one test case, trying to vary the cases as much as much as possible to cover the most interesting ones. Regarding

Whitebox	**Data Flow**	**Blackbox**
• Compute complexity • Draw Graphs • Look for paths • Exclude non-feasable paths • Create test cases to cover all paths	• Choose variables to analyse • Draw Graphs • Look for paths • Exclude non-feasable paths • Create test cases to cover all paths	• Check input/output of each function • Check inputs to test • Compute the domain, equivalence classes and boundry values • Generate test cases

Table 2: Testing Plan

coverage requirements, the whitebox tests seek to completely cover all the branches while in the blackbox tests we try to include different size and format matrices.

The tool we will use for the testing purposes is Python's module ”unittest” which implements some very useful features that we will benefit from in our work. For the case tests we will generate matrices using a script and store them in a file that will later be imported and used in the tests. In some cases it could be useful to measure and consider the time elapsed during the tests, for example, if the program was to be used in an online service environment where latency is an important factor. In our case we only consider the amount of tests passed or failed.

6 Item Pass/Fail Criteria

We will consider Major Defects as cases where the system crashes like when an out of bound value occurs and it causes the program to stop abruptly. Minor Defects in our case are considered to be wrong outputs that the program returns instead of the correct answer to the input provided. Since we are conducting dynamic unit tests of an algorithm, we are looking for: 0 major and 0 minor defects on the entirety of the test cases.

7 Test Deliverables

As a result of this planned testing, we shall submit the following materials as part of the development and results of the tests:

- Testing Plan
- Test cases, organized by folders and correctly labeled
- Code for the test cases
- Output of the test cases
- Error log
- Test Completion Report
- Test measures with appropriate visual representations

It is important to note that some of the items listed above are directly included in this report in the corresponding sections. All other deliverables will be attached and submited alongside this report.

8 Environmental Needs

We did not identify or define any specific environmental needs for the process of this practical assignment of software testing.

9 Staffing and responsibilities

The tests should be carried out by someone that did not directly create the code that is to be tested and they should have at least some experience in the field in order to properly design the blackbox and white box tests. The tests that are created should be adequate and span the totality of the code, especially the white box tests. The black box tests should be carefully designed to cover all boundary cases as well as the most common ones.

10 Test Completion Report

Before rushing for the results, we find it important to document and explain the steps taken along the journey. This includes the definition of the multiple types of tests and the reason behind some decisions that were made. We should also mention the variables' names that are present on the code and what they mean:

- **R** - Number of rows
- **C** - Number of columns
- **M** - Input matrix
- **S** - Sizes matrix

10.1 Blackbox Testing

We shall start by the Blackbox tests. By definition we should follow the guide: (1) defining domain, (2) partition the test cases into valid and invalid equivalence classes, (3) test all the combination of classes, (4) check for boundary cases in the classes, (5) test all the combination of boundary cases. We decided that variables **R** and **C** were not to be messed with since they are directly dependent on the size of the input matrix, **M**, and not something that can be defined separately. These two variables are calculated using Python's *len()* function to calculate the number of rows and number of columns. We will also not test incorrect **S** matrices, we assume that the previous function returned a valid output matrix.

```
Input: R, C, S, M

Inputs to test: M (R, C are correct because they depend on M, S empty)

Output: S with first column and row equal to M, all other cells are 0s

Domain: M - [0x0, 999999x999999]

Valid equivalence classes: M - 0x0
             [1x1, 9x9]
             [10x10, 999999x999999]

Invalid equivalence classes: M - Non-binary matrix

Boundary cases: M - 0x0, 1x1
             9x9, 10x10,
             999998x999998,
          999999x999999,
             1000000x1000000 (aka, biggest possible matrix)
```

Listing 1: *CreateAuxMatrix()* Function - Black Box Testing

```
Input: R, C, S, M

Inputs to test: M (R, C and S are correct because they depend on M)

Output: S filled with the sizes

Domain: M - [0x0, 999999x999999]

Valid equivalence classes: M - 0x0
             [1x1, 9x9]
             [10x10, 999999x999999]

Invalid equivalence classes: M - Non-binary matrix

Boundary cases: M - 0x0, 1x1
             9x9, 10x10,
             999998x999998,
             999999x999999,
             1000000x1000000 (aka, biggest possible matrix)
```

Listing 2: *FillAux()* Function - Black Box Testing

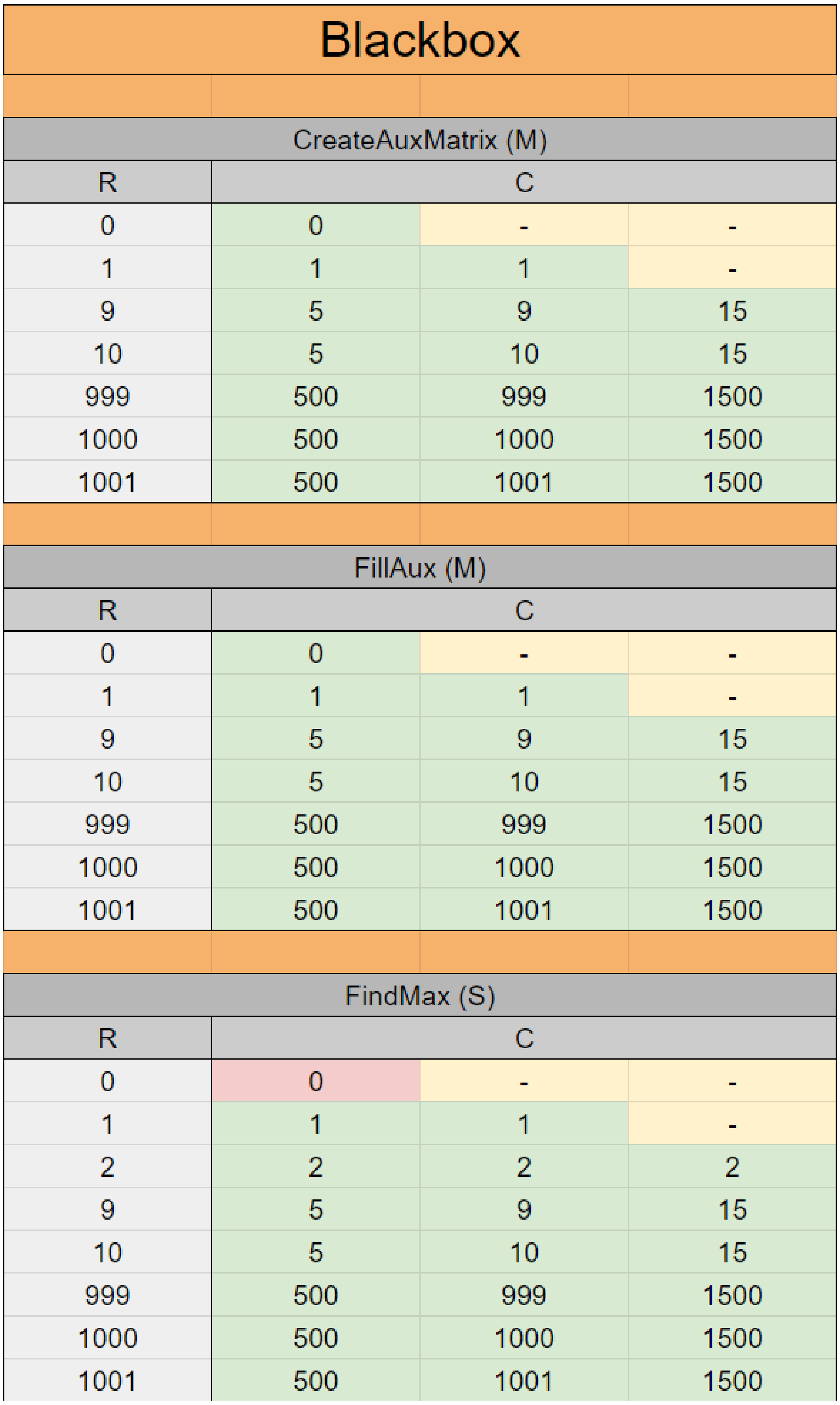
```
Input: R, C, S

Inputs to test: S

Output: Maximum value of S and its coordinates on the matrix

Domain: M - [0x0, 999999x999999]

Valid equivalence classes: M - 0x0
                [1x1, 9x9]
                [10x10, 999999x999999]

Invalid equivalence classes: S - Matrix not matching C or R

Boundary cases: M - 0x0, 1x1, 2x2
                9x9, 10x10,
                999998x999998,
                999999x999999,
                1000000x1000000 (aka, biggest possible matrix)
```

Listing 3: *FindMax()* Function - Black Box Testing

For the Blackbox testing we defined the parameters that are present in the **Listings 1**, **2** and **3**, each corresponding to one of the functions we are testing. Since all functions accept a matrix as their input, the valid equivalence classes are just the possible matrices that we can send to the function, that is, every matrix from 0x0 (null matrix) to the biggest matrix that is supported by the system. This upper limit is quite a tricky one since we very easily get to a point were the size of the matrix is too large, for example, a 10 thousand by 10 thousand matrix has one hundred million cells which will take an immense time to go through using this algorithm. For this reason it is unrealistic to test the absolute upper limits of these variables since the deadline for the assignment is already set and we don't have decades to test these values. The maximum size matrix that was tested was 1001x1500 during the Blackbox tests.

In **Figure 1** we can see the result of the blackbox tests that were carried out. Each table has the information about the tests for each separate function and inside each table there's the number of rows and number of columns of the matrix that was tested. Each cell inside the tables is colored and it adheres to the following color code:

- **Green**: Success
- **Red**: Failure
- Yellow: Non-applicable
- Blue: Non feasible



Blackbox

CreateAuxMatrix (M)			
R	C		
0	0	-	-
1	1	1	-
9	5	9	15
10	5	10	15
999	500	999	1500
1000	500	1000	1500
1001	500	1001	1500

FillAux (M)			
R	C		
0	0	-	-
1	1	1	-
9	5	9	15
10	5	10	15
999	500	999	1500
1000	500	1000	1500
1001	500	1001	1500

FindMax (S)			
R	C		
0	0	-	-
1	1	1	-
2	2	2	2
9	5	9	15
10	5	10	15
999	500	999	1500
1000	500	1000	1500
1001	500	1001	1500

Figure 1: Blackbox Tests

We can see that every applicable test was successful except for the case where the input for the function *FindMax()* is the null matrix which, in fact, can be the output of the previous function, *FillAux()*. We should also add that we tested not only square matrices (N*N) but also rectangular matrices (N*M) where N > M and N < M.

10.2 Control Flow Testing

Moving on from Blackbox testing we now turn our heads to Control Flow testing. Once again the variables are the same as the ones in Blackbox testing and we should always try to execute more than one test per path.

We calculated the McCabe's Cyclomatic Complexity, using the Control Flow Graphs, for the three separate functions and the values for each are: 5, 4, 4 for the functions *CreateAuxMatrix()*, *FillAux()* and *FindMax()*, respectively. This helped us confirm that we had found all possible paths when preparing the Control Flow Tests.

```
Input: R, C, S, M
Output: S with first column and row equal to M, all other cells are 0s

Path 1 -> R = 0
Path 2 -> R != 0, C = 0 #Non-executable because it is impossible for a
    ↪ matrix to have more than 0 rows and still have 0 columns

Path 3 -> R != 0, C != 0, (i = 0)
Path 4 -> R != 0, C != 0, (j = 0) Unfeasable
Path 5 -> R != 0, C != 0, (i != 0, j != 0) Unfeasable
```

Listing 4: *CreateAuxMatrix()* Function - Control Flow Testing

```
Input: R, C, S, M
Output: S filled with the sizes

Path 1 -> R = 0
Path 2 -> R != 0, C = 0 #Non-executable because it is impossible for a
    ↪ matrix to have more than 0 rows and still have 0 columns

Path 3 -> R != 0, C != 0, M[i,j] = 1
Path 4 -> R != 0, C != 0, M[i,j] != 1
```

Listing 5: *FillAux()* Function - Control Flow Testing

```
Input: R, C, S
Output: Maximum value of S and its coordinates on the matrix

Path 1 -> R = 0 #Function can't be executed with null matrices.
Path 2 -> R != 0, C = 0 #Non-executable because it is impossible for a
    ↪ matrix to have more than 0 rows and still have 0 columns

Path 3 -> R != 0, C != 0, S[i,j] > max_of_S (S[0,0])
Path 4 -> R != 0, C != 0, S[i,j] <= max_of_S (S[0,0])
```

Listing 6: *FindMax()* Function - Control Flow Testing

In the **Listings 4**, **5** and **6** we can see the Control Flow Testing planning for each function. In **Figures 2**, **3** and **4**, we can see the Control Flow Graph for the functions

CreateAuxMatrix(), *FillAux()* and *FindMax()*, respectively, with their nodes numbered so we can easily identify the paths in the CFG.

These Control Flow test tables in **Figure 5** follow the same coloring pattern defined for the Blackbox tests. Here we can see the different paths defined for each function as well as the results that were obtained, alongside some comments about each path. Just like in the Blackbox tests we can see that when the null matrix is provided as an input to the function *FindMax()*, it crashes because it does not support this input. Besides this fail, all of the other feasible and applicable tests passed.

It is important to explain all the paths that we identified as non feasible. From top to bottom on the Control Flow test tables, **Figure 5**, all of the blue lines are non feasible paths. Starting with the first one, it is not possible because we can't create a matrix with more than 0 rows and exactly 0 columns. The next path is not possible because the "*i=0*" statement has priority over the "*else if j=0*". Same thing goes for the next, and last, unfeasible path of the *CreateAuxMatrix()* function, the "*i=0*" statement has priority over the "*else*" statement. The only non feasible path we identified in the *FillAux()* function occurs, once again, when it is required to have a matrix with more than 0 rows and exactly 0 columns, which is obviously impossible to achieve. Lastly, in the *FindMax()* function, the first path we identified as non feasible is again the matrix with more than 0 rows and exactly 0 columns. The last one is when the "*max_of_s*" variable is equal to the first value of the S matrix, since the condition is a "lesser than" it will never enter the *if* statement on the case where the values are equal.

10.3 Data Flow Testing

Moving on to Data Flow testing, we start by enumerating the steps that should be followed when carrying out these tests: (1) draw a data flow graph from the program, (2) select one or more data flow testing criteria, (3) identify paths in the data flow graph satisfying the selection criteria. (4) derive path predicate expressions from the selected paths, (5) solve the path predicate expressions to derive test inputs.

We start by showing the Data Flow Graphs for each individual function in the **Figures 6**, **7** and **8**. These graphs also contain the numbering for each node of the Data Flow so we can easily identify the paths. In **Figure 9** we can see the different tests that were done to each function and the paths through the Data Flow Graphs that were being tested along with some comments about each test.

It is once again important to explain all the paths that we identified as non feasible. In this part of testing we only identified four non feasible paths and all of them occur in the *CreateAuxMatrix()* function. From top to bottom we start with the first one, here it is impossible for C (number of columns) to be 0 when R (number of rows) is not zero. The next unfeasible path occurs because node 8 in the Data Flow Graph of the *CreateAuxMatrix()* function has priority over node 9. The next path we identified as not feasible is due to the fact that node 8 is only accessible if the value of the variable *i* is equal to 0 which only happens once, while variable *j* gets that value multiple times during a loop. Finally, node 9 is only accessible one time per line of the matrix and every time the line changes, the variable *temp* is redifined in node 4.

Just like it happened for the previous tests, the null matrix cannot be the input to the *FindMax()* function because it does not support it.

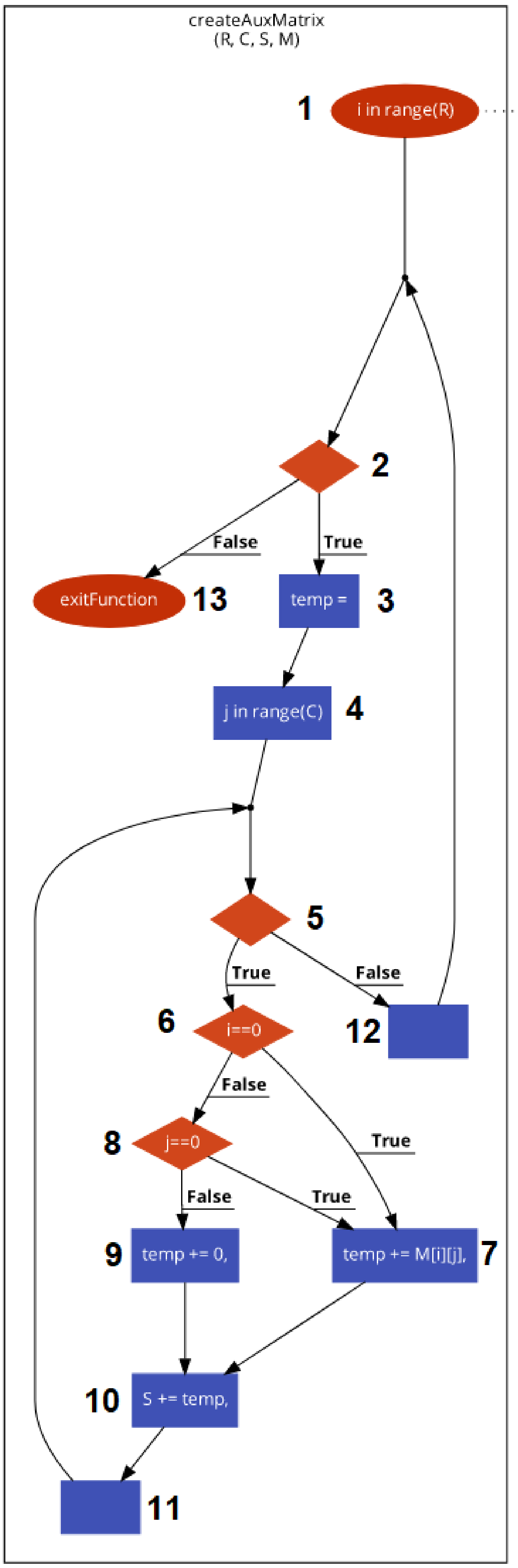


Figure 2: Control Flow Graph - *CreateAuxMatrix()* Function

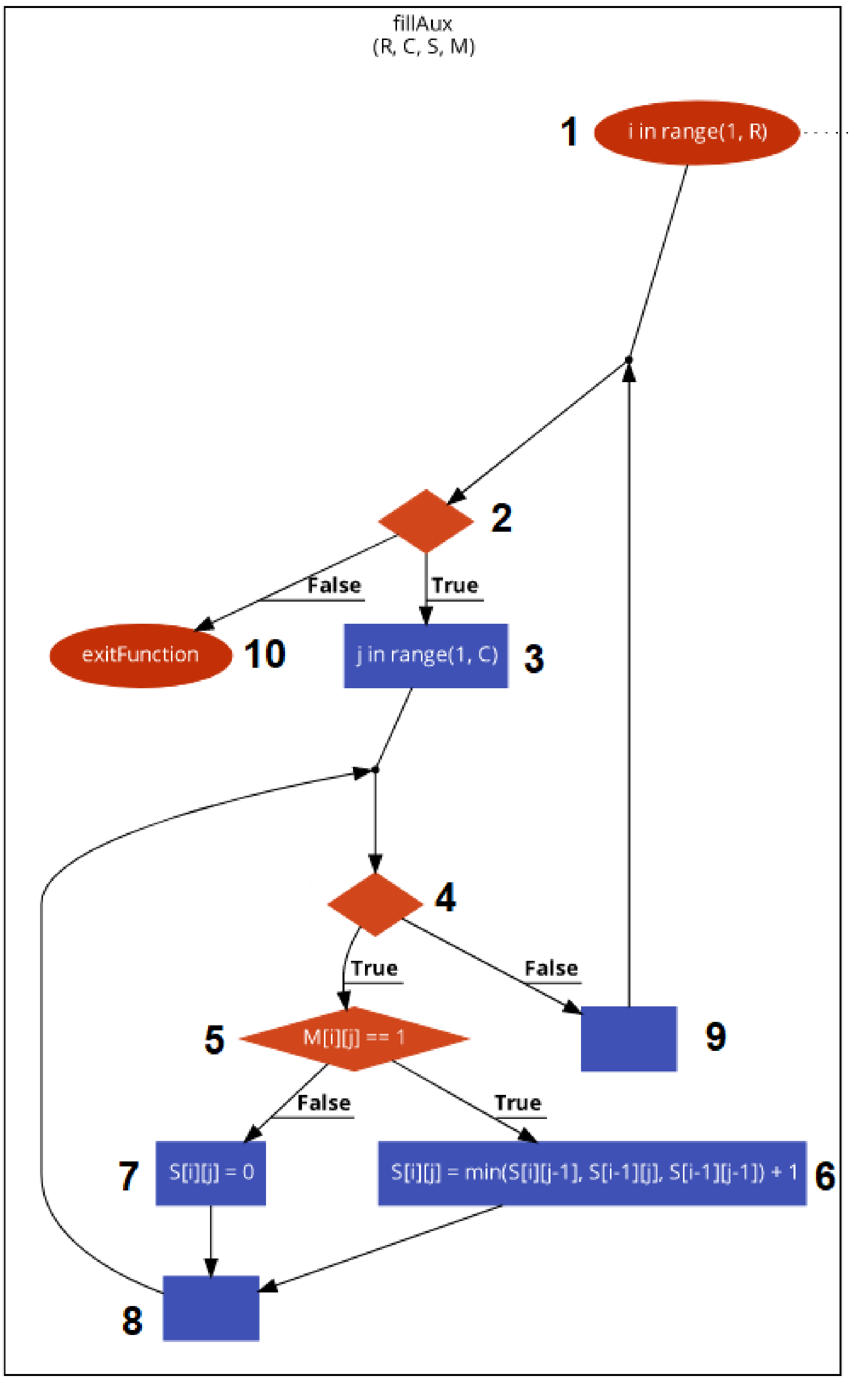


Figure 3: Control Flow Graph - *FillAux()* Function

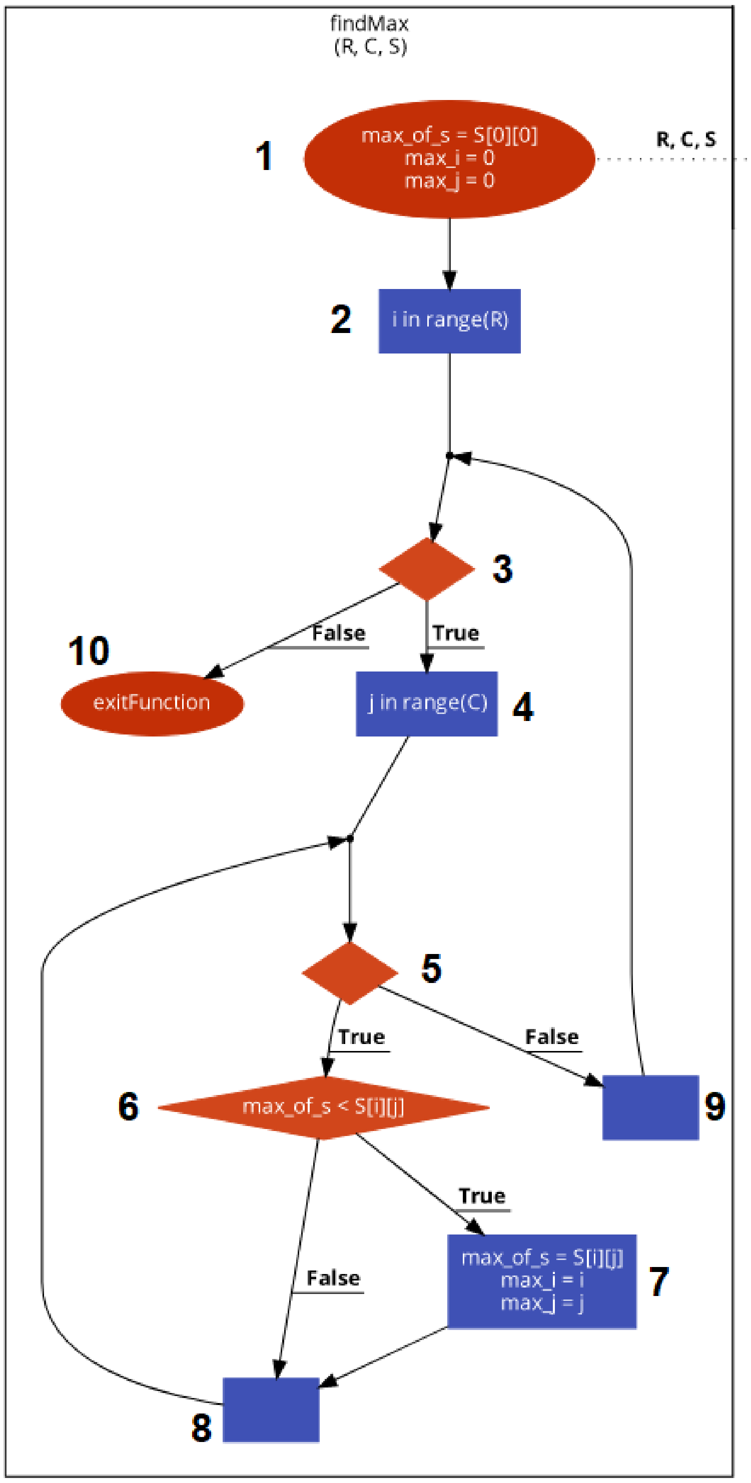
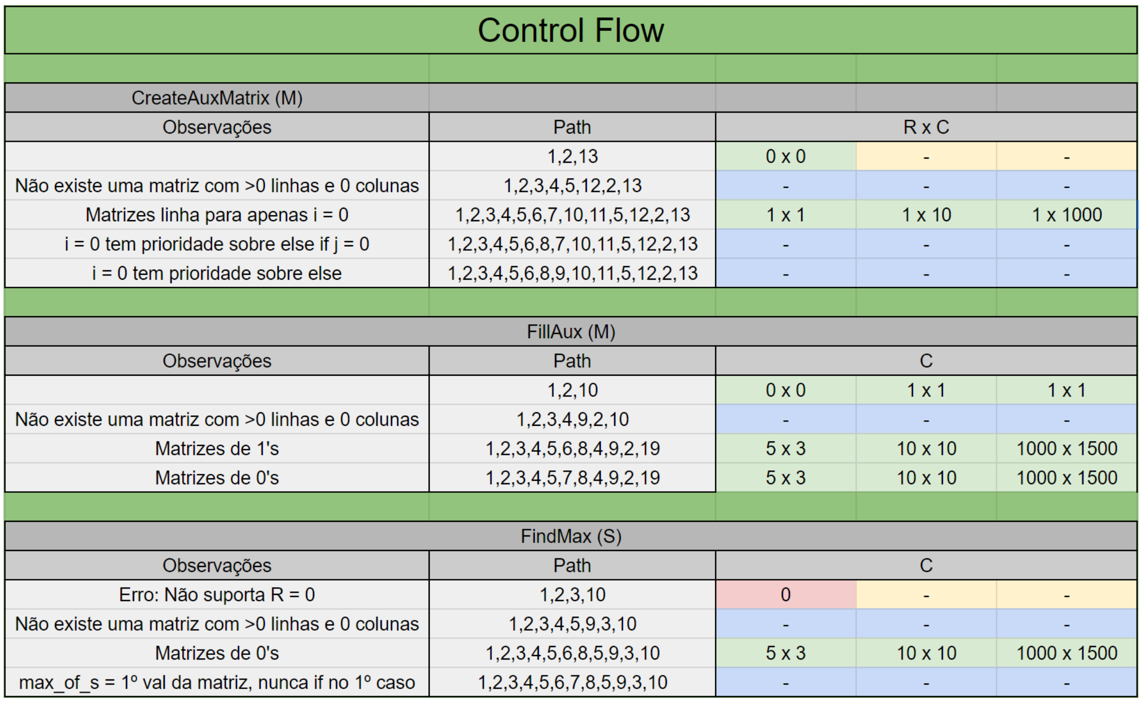


Figure 4: Control Flow Graph - *FindMax()* Function



Control Flow				
CreateAuxMatrix (M)				
Observações	Path	R x C		
	1,2,13	0 x 0	-	-
Não existe uma matriz com >0 linhas e 0 colunas	1,2,3,4,5,12,2,13	-	-	-
Matrizes linha para apenas i = 0	1,2,3,4,5,6,7,10,11,5,12,2,13	1 x 1	1 x 10	1 x 1000
i = 0 tem prioridade sobre else if j = 0	1,2,3,4,5,6,8,7,10,11,5,12,2,13	-	-	-
i = 0 tem prioridade sobre else	1,2,3,4,5,6,8,9,10,11,5,12,2,13	-	-	-
FillAux (M)				
Observações	Path	C		
	1,2,10	0 x 0	1 x 1	1 x 1
Não existe uma matriz com >0 linhas e 0 colunas	1,2,3,4,9,2,10	-	-	-
Matrizes de 1's	1,2,3,4,5,6,8,4,9,2,19	5 x 3	10 x 10	1000 x 1500
Matrizes de 0's	1,2,3,4,5,7,8,4,9,2,19	5 x 3	10 x 10	1000 x 1500
FindMax (S)				
Observações	Path	C		
Erro: Não suporta R = 0	1,2,3,10	0	-	-
Não existe uma matriz com >0 linhas e 0 colunas	1,2,3,4,5,9,3,10	-	-	-
Matrizes de 0's	1,2,3,4,5,6,8,5,9,3,10	5 x 3	10 x 10	1000 x 1500
max_of_s = 1º val da matriz, nunca if no 1º caso	1,2,3,4,5,6,7,8,5,9,3,10	-	-	-

Figure 5: Control Flow Tests

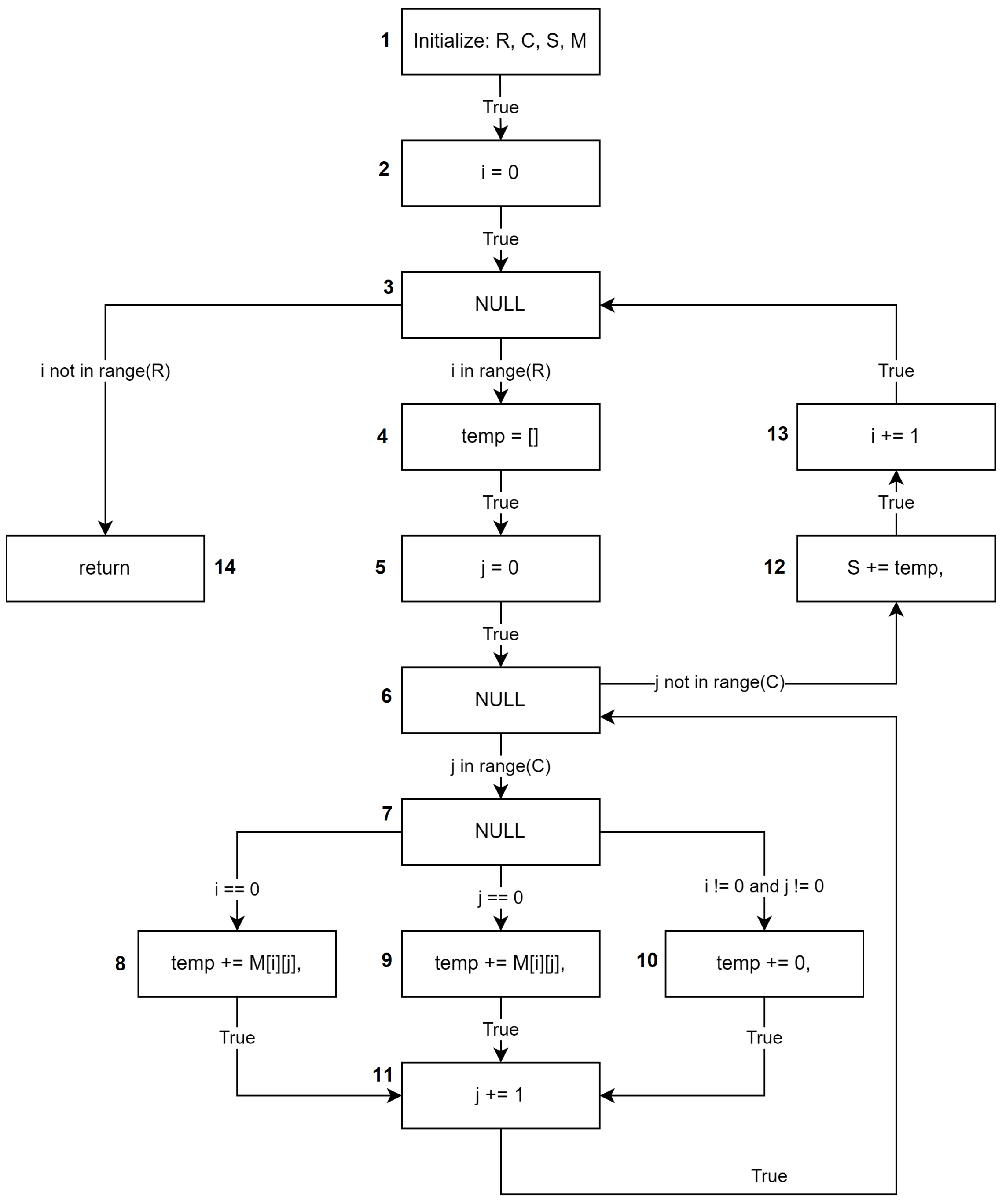


Figure 6: Data Flow Graph - *CreateAuxMatrix()* Function

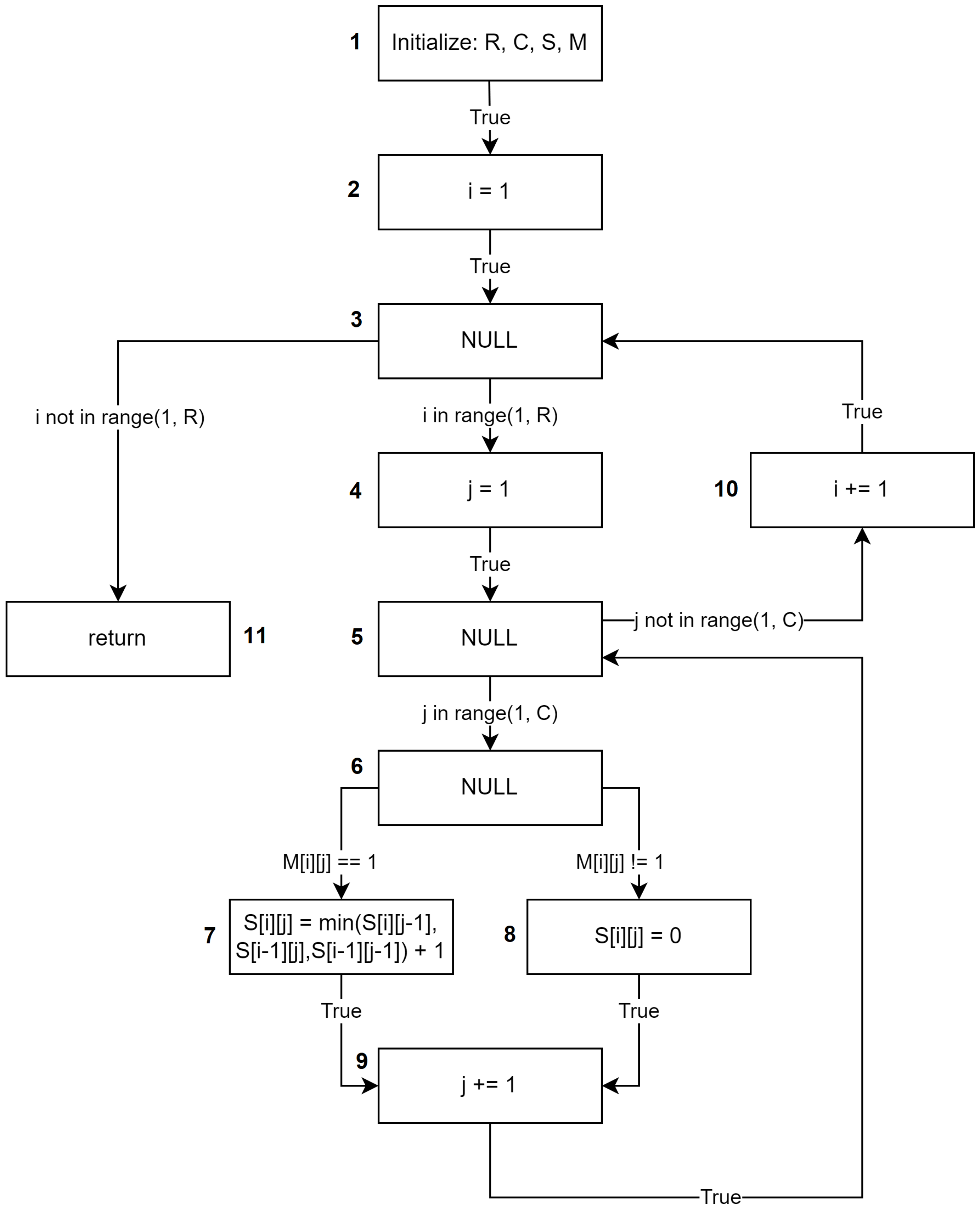


Figure 7: Data Flow Graph - *FillAux()* Function

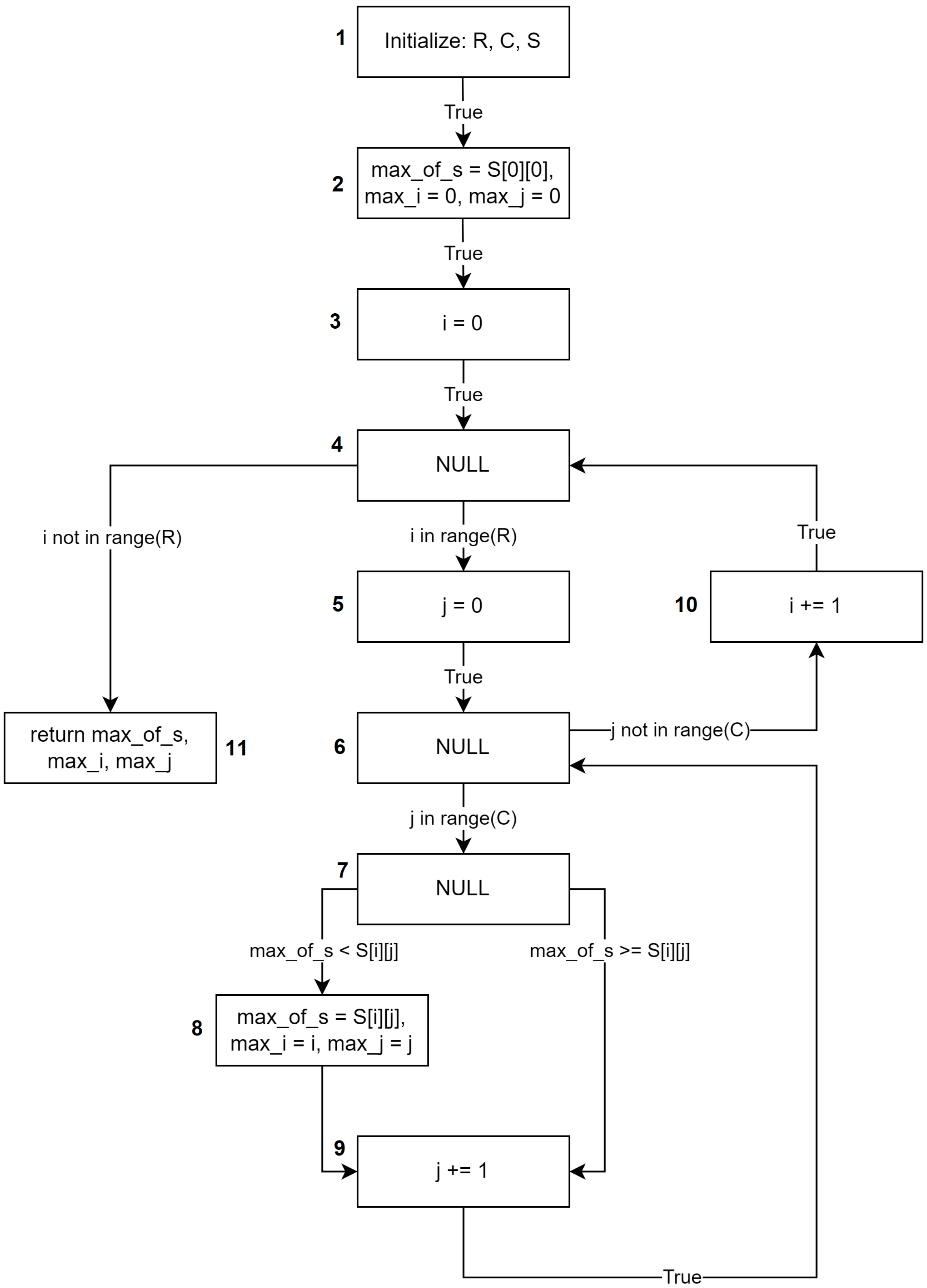
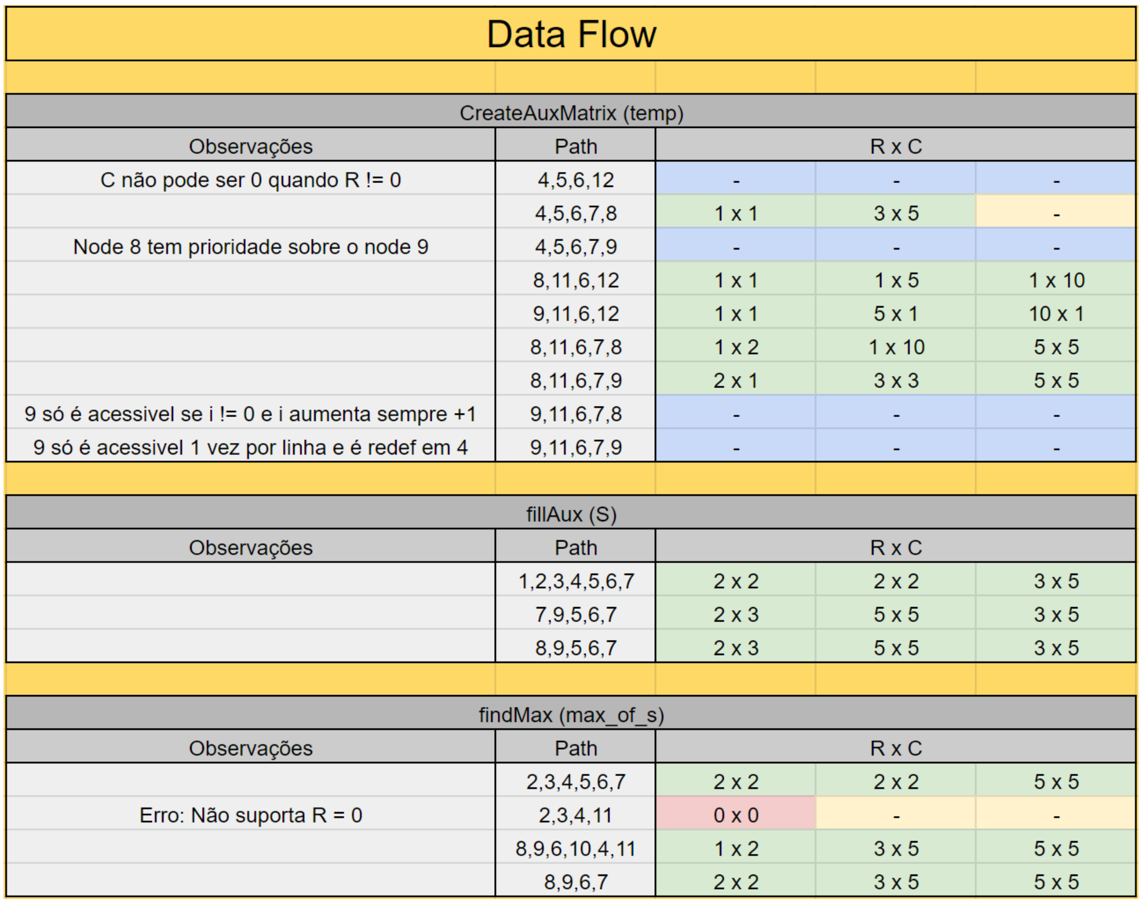


Figure 8: Data Flow Graph - *FindMax()* Function



Data Flow

CreateAuxMatrix (temp)

Observações	Path	R x C		
C não pode ser 0 quando R != 0	4,5,6,12	-	-	-
	4,5,6,7,8	1 x 1	3 x 5	-
Node 8 tem prioridade sobre o node 9	4,5,6,7,9	-	-	-
	8,11,6,12	1 x 1	1 x 5	1 x 10
	9,11,6,12	1 x 1	5 x 1	10 x 1
	8,11,6,7,8	1 x 2	1 x 10	5 x 5
	8,11,6,7,9	2 x 1	3 x 3	5 x 5
9 só é acessivel se i != 0 e i aumenta sempre +1	9,11,6,7,8	-	-	-
9 só é acessivel 1 vez por linha e é redef em 4	9,11,6,7,9	-	-	-

fillAux (S)

Observações	Path	R x C		
	1,2,3,4,5,6,7	2 x 2	2 x 2	3 x 5
	7,9,5,6,7	2 x 3	5 x 5	3 x 5
	8,9,5,6,7	2 x 3	5 x 5	3 x 5

findMax (max_of_s)

Observações	Path	R x C		
	2,3,4,5,6,7	2 x 2	2 x 2	5 x 5
Erro: Não suporta R = 0	2,3,4,11	0 x 0	-	-
	8,9,6,10,4,11	1 x 2	3 x 5	5 x 5
	8,9,6,7	2 x 2	3 x 5	5 x 5

Figure 9: Data Flow Tests