Sabancı University

Faculty of Engineering and Natural Sciences
CS204 Advanced Programming
Spring 2021

Homework 4 – Pattern Search via Stacks

Due: 7/4/2021, Wednesday, 21:00

This is semi-optional homework. It will be collected and graded. However, if this homework reduces your overall course grade or if you do not submit, it will not be considered in overall grade calculation. In such a case, the weight of this homework will be distributed to other homework assignments.

PLEASE NOTE:

Your program should be a robust one such that you have to consider all relevant programmer mistakes and extreme cases; you are expected to take actions accordingly!

You can NOT collaborate with your friends and discuss solutions. You have to write down the code on your own. Plagiarism will not be tolerated!

Introduction

In this homework, you are asked to implement a pattern searching program which makes use of *stacks*. The pattern to be searched will be a string of bits i.e. a string of 0's and 1's of arbitrary length. Your program will search the string within a matrix of bits. This matrix will be input from a file. The search will be snaky such that consecutive bits of the search string may not lie on a line; but may follow a tortuous path. The details of the methods, algorithms and the data structure to be used will be given in the subsequent sections of this homework specification.

Input and output

Firstly, the program asks the number of rows and columns of the matrix and reads them from keyboard. Then, program asks and reads the name of the file which contains the matrix with the given number of rows and columns. After reading the matrix from the file, the program will ask for a string that will be searched. After the search, the results will be displayed and the program will ask again for a new string. It will repeat this process until **CTRL-Z** is entered.

You can assume that the row and column values entered at the beginning are positive integers; no need to make an input check for those. You can also assume that the file has data for a correct rectangular matrix in terms of row and column counts entered by the user. You do not need to make an input check for the file content, but you should check whether the file exists or not (continue reading new file name until successfully opened). Please use C++ convention for the indexing of the matrix cells (i.e. indices start from 0 and first index is for row, second one is for column). A sample input matrix is given below.

```
011111
110001
010011
100001
011010
```

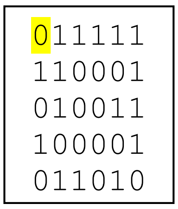
Fig 1. Sample input file

The Search Rules

The search for a given bit string must always start at (0, 0) cell (upper-left). The flow of search must be toward right (east) or down (south) or mix of these (other directions cannot be used). That means if ith bit of the search string is at (r, c) cell, then you have to consider only (r+1,c) or (r, c+1) cells for (i+1)th bit, not the other neighbors. This way, we can define a set of consecutive cells to represent the search string bits as a path. The purpose of the program is to find such a specific path. There might be more than one possible path; in such a case, finding one of them is sufficient; you do not need to find all. More details on how you can perform the search will be revealed in the upcoming sections of the document.

Now, let us give same examples to clarify the search mechanism and rules. Assume the input file is given in Figure 1, which contains a 5 x 6 matrix. In the examples below, the solutions are marked with highlighted cells in the matrices.

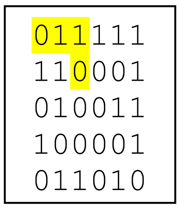
<u>Example 1:</u> If the string '0' is searched, it will be found it at the starting position.



```
011111
110001
010011
100001
011010
```

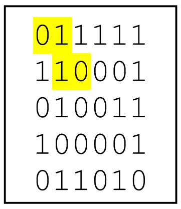
Fig 2. Result of Example 1

<u>Example 2:</u> If the string '0110' is searched, three different occurrences can be found. Displaying only one of the solutions is enough for the homework, so any of the below is acceptable.



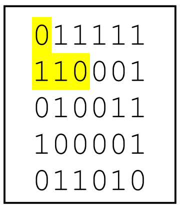
```
011111
110001
010011
100001
011010
```

Fig 3. One of the solutions of Example 2



```
011111
110001
010011
100001
011010
```

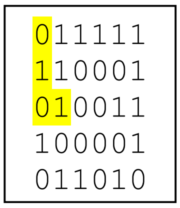
Fig 4. Another solution of Example 2



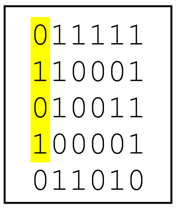
```
011111
110001
010011
100001
011010
```

Fig 5. Another solution of Example 2

Example 3: If the string ‘0101’ is searched, two different solutions can be found as shown below. Displaying only one of the solutions is enough.



```
011111
110001
010011
100001
011010
```

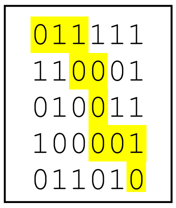


```
011111
110001
010011
100001
011010
```

Fig 6. One of the solutions of Example 3

Fig 7. Another solution of Example 3

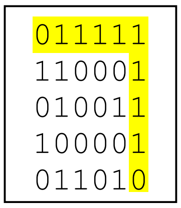
Example 4: If the string ‘0110000010’ is searched, again multiple solutions exist. One of the paths that traces the search string is below.



```
011111
110001
010011
100001
011010
```

Fig 8. One of the solutions of Example 4

Example 5: In case the string ‘0111111110’ is searched, the following path will be the only solution.



```
011111
110001
010011
100001
011010
```

Fig 9. The only solution of Example 5

Examples of not finding: If ‘0100’ is to be searched, there will be no solutions to display. Similarly if the string ‘1100’ is searched, again no solution can be found. In fact, no string starting with a ‘1’ can be found in this example, because all searches start at position (0,0) and at (0, 0), there is a ‘0’.

Data structures to be used

Each element of the matrix is a character. In addition to the actual element, you need to store a Boolean flag along with it (this will be needed in the search algorithm which, will be explained in the next section). You can encapsulate these two primitive containers (`char` and `bool`) under a *`struct`*, let us call **`cellStruct`**. In your program, you will create a **dynamic `cellStruct` matrix**, without using `vector` class, to store the elements of the matrix.

Another data structure that you will use is a *stack*. As will be discussed in the next section, in order to keep track of the visited cells while searching for the string, you must use a stack data structure. The data that you will keep in this stack are the row and column indices of some of the cells of the matrix. We are expecting you to implement a dynamic stack class for this application and use it in the main program. The dynamic stack class that you will implement must contain only *push*, *pop*, and *isEmpty* member functions other than the constructor and destructor. In other words, you are not allowed to add extra public member functions that violate the spirit behind the stack (i.e. member functions in order to display the content, check the top element without popping, insert in the middle, etc. are **NOT** allowed). Normally, your algorithm would only need constructor, destructor, push, pop and isEmpty functions of the stack.

If you explicitly or implicitly call copy constructor, you have to implement copy constructor for the stack class as well. Otherwise, you do not have to. Similarly, if you use assignment operator, implement it; otherwise, no need.

You are not allowed to use any other container in this homework.

Algorithmic details and hints

Since it may not be so clear to you how to search for a specific pattern which can exist in an snaky path (different than the classical crossword layout), we give some algorithmic hints in this section. The main idea of the algorithm that you will use is to keep track of all visited cells (by pushing them onto the stack) and backtrack to a previously visited cell (by popping from the stack) when you are stuck. This way, when you notice that you cannot move on from an element, you can return to a previously found path and try to continue from that point. [illegible]

We have previously mentioned that every element of the matrix is struct that contains the binary value (0 or 1) as a character and Boolean flag. Initially this flag is FALSE meaning that the cell can potentially be on the path of solution. During your search, if you encounter that you cannot move forward from a particular cell, you have to mark its flag by changing its value to TRUE. In other words, if the Boolean flag of a cell is TRUE, it indicates that your program visited that element but could not include it into the solution path. Therefore, you have to mark (by making the flag TRUE) any matrix elements that would get you stuck. Moreover, you have to check the flag before visiting an element, since you do not want to visit an element which will prevent you from moving forward.

The algorithm that you can implement for searching a string in the matrix is as follows (of course, you can follow another algorithm, but you have to use stack for backtracking).

1. ~~Firstly, you have to focus at the starting cell (0, 0). If the starting element does not allow a path for the searched string (i.e. if the matrix value at (0, 0) is not the 0th element of the search string), you set the flag of (0, 0) to TRUE before checking the rest (i.e. before getting into the loop to check the rest).~~
2. Then you start to search the rest within a loop. This loop should stop when you find the desired path or the flag of (0, 0) is TRUE.
    a. In each step of your search (in the loop), you have to check on a specific element of the matrix (initially it is (0, 0), but at each iteration it may change as explained below). You have to check if the flag of this element is TRUE or FALSE. If it is FALSE and if the matrix value at this element is the same as the corresponding bit in the search string, then it means this position is a candidate path element. In such a case:
        i. First ~~you have to push that position on your stack~~ (in order to remember that position later).
        ii. You have to check if the entire string has been found.

        iii. ~~Else if there still are some bits to search for, you will move toward either right or down to be checked in the next iteration. Of course, this move will be done if the flag of the neighbors is not TRUE~~ (i.e. if not blocked).
        iv. Else if both directions are blocked~~, you should change the flag of the current element to TRUE,~~ since you cannot move forward from it as well and it blocks you. That means, the current cell cannot be on the path and you have ~~backtrack to~~ the previous matched bit. In such a case, you backtrack by popping from the stack.
    b. Else if the flag of the current matrix element is TRUE or if the value is not the same as the corresponding bit of the search string, then this position blocks your search. In such a case:
        i. first you have to set the flag of this position to TRUE mentioning it is blocked.
        ii. You have to backtrack to previous match by popping the last visited cell from the stack.
3. At the end of the loop, either you found path for the search string or it has not been found.
    a. If found, the resulting path must be displayed by printing out the row and column indices of the elements in the order that they appear on the path starting from (0, 0). Hint: the elements of the path are already in the stack after the loop, but in reverse order. You will need to reverse the stack content via another stack (pop from one and push to other) and display the content of the reversed stack by popping from it.
    b. If not found, you have to display an appropriate message. Please see the sample runs for message format. Understanding that a path is not found is simply by checking the flag of (0,0) element of the matrix. If this flag is TRUE, then a path is not found. This flag may become TRUE either before the loop starts or within the loop by backtracking.

This algorithm is only for one search string; as mentioned previously, you have to be able to search several strings one after another until CTRL-Z is entered.

There also exist recursive solutions to the pattern matching problem. However, we **strongly recommend** you **not** to use recursion for this problem. If you use recursion, for some big (actually not so big) matrices, these recursive calls cause to use up all available memory and your program crashes due to stack overflow. If you write your code recursively and if your recursive program does not work with our grading cases, we do **not** take any responsibility.

Some Important Rules

In order to get a full credit, your programs must be efficient and well presented, presence of any redundant computation or bad indentation, or missing, irrelevant comments are going to decrease your grades. You also have to use understandable identifier names, informative introduction and prompts. Modularity is also important; you have to use functions wherever needed and appropriate. Since using classes is mandated in this homework, a proper object-oriented design and implementation will also be considered in grading.

Since you will use dynamic memory allocation in this homework, it is very crucial to properly manage the allocated area and return the deleted parts to the heap whenever appropriate. Inefficient use of memory may reduce your grade.

When we grade your homework we pay attention to these issues. Moreover, in order to observe the real performance of your codes, we may run your programs in *Release* mode and **we may test your programs with very large test cases**. Of course, your program should work in *Debug* mode as well.

Please do not use any non-ASCII characters (Turkish or other) in your code.

You are allowed to use sample codes shared with the class by the instructor and TA, but you are not allowed to use any codes that you might find somewhere else (any online, machine or human source). However, you cannot start with an existing .cpp or .h file directly and update it; you have start with an empty file. Only the necessary parts of the shared code files can be used and these parts must be clearly marked in your homework by putting comments like the following. Even if you take a piece of code and update it slightly, you have to put a similar marking (by adding "`and updated`" to the comments below.

```
/* Begin: code taken from ptrfunc.cpp */

…

/* End: code taken from ptrfunc.cpp */
```

Sample runs

Some sample runs are given below, but these are not comprehensive, therefore you have to consider **all possible cases** to get full mark. Please do not make any assumptions on the names of files.

Sample Run 1:

File: matrix1.txt (shown here for your convenience; you do not need to display the matrix)

```
00110100
01110100
10010100
10110000
01110000
01011101
```

```
Please enter the number of rows: 6
Please enter the number of columns: 8
Please enter the name of the input file that contains the matrix: mat.txt
File cannot be opened.
Please enter the name of the input file again: matrix.txt
File cannot be opened.
Please enter the name of the input file again: matrix1.txt

Please enter a string of bits to search (CTRL+Z to quit): 0
The bit string 0 is found following these cells:
(0,0)
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): 1
The bit string 1 could not be found.
```

```
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): 00
The bit string 00 is found following these cells:
(0,0) (0,1)
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): 0010
The bit string 0010 is found following these cells:
(0,0) (0,1) (1,1) (2,1)
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): 001110
The bit string 001110 is found following these cells:
(0,0) (0,1) (0,2) (0,3) (1,3) (1,4)
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): 01
The bit string 01 could not be found.
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): 0011010000001
The bit string 0011010000001 is found following these cells:
(0,0)  (0,1)  (0,2)  (0,3)  (0,4)  (0,5)  (0,6)  (0,7)  (1,7)  (2,7)  (3,7)  (4,7)
(5,7)
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): 0011001011101
The bit string 0011001011101 is found following these cells:
(0,0)  (1,0)  (2,0)  (3,0)  (4,0)  (5,0)  (5,1)  (5,2)  (5,3)  (5,4)  (5,5)  (5,6)
(5,7)
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): 001001
The bit string 001001 is found following these cells:
(0,0) (0,1) (1,1) (2,1) (2,2) (2,3)
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): ^Z
Program ended successfully!

Press any key to close this window . . .
```

Sample run 2:
File: matrix2.txt (shown here for your convenience; you do not need to display the matrix)

```
011
111
111
```

```
Please enter the number of rows: 3
Please enter the number of columns: 3
Please  enter  the  name  of  the  input  file  that  contains  the  matrix:
matrix2.txt

Please enter a string of bits to search (CTRL+Z to quit): 100
The bit string 100 could not be found.
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): 010
The bit string 010 could not be found.
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): 000
The bit string 000 could not be found.
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): 011
The bit string 011 is found following these cells:
```

```
(0,0) (0,1) (0,2)
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): 01
The bit string 01 is found following these cells:
(0,0) (0,1)
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): 00010110001
The bit string 00010110001 could not be found.
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): 0
The bit string 0 is found following these cells:
(0,0)
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): 2x3
The bit string 2x3 could not be found.
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): random
The bit string random could not be found.
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): ^Z
Program ended successfully!

Press any key to close this window . . .
```

Sample run 3:
File: matrix3.txt (shown here for your convenience; you do not need to display the matrix)

```
101101000101
011101001010
000101000101
101100000101
011100000110
010111010110
010111010101
101010101110
000111111010
```

```
Please enter the number of rows: 9
Please enter the number of columns: 12
Please enter the name of the input file that contains the matrix:
matrix3.txt

Please enter a string of bits to search (CTRL+Z to quit): 1001
The bit string 1001 is found following these cells:
(0,0) (1,0) (2,0) (3,0)
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): 1001000101011
The bit string 1001000101011 is found following these cells:
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (6,1) (6,2) (6,3) (7,3) (7,4)
(8,4)
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit):
10110100010101100100
The bit string 10110100010101100100 is found following these cells:
```

```
(0,0) (0,1) (0,2) (0,3) (0,4) (0,5) (0,6) (0,7) (0,8) (0,9) (0,10) (0,11)
(1,11) (2,11) (3,11) (4,11) (5,11) (6,11) (7,11) (8,11)
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit):
10110000000000011010
The bit string 10110000000000011010 is found following these cells:
(0,0) (0,1) (0,2) (0,3) (0,4) (1,4) (2,4) (3,4) (3,5) (3,6) (3,7) (3,8)
(4,8) (5,8) (6,8) (6,9) (7,9) (8,9) (8,10) (8,11)
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit):
101100000000000110101
The bit string 101100000000000110101 could not be found.
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): ^Z
Program ended successfully!

Press any key to close this window . . .
```

Sample run 4:
File: matrix4.txt (see the homework package for the file)

```
Please enter the number of rows: 50
Please enter the number of columns: 50
Please enter the name of the input file that contains the matrix:
matrix4.txt

Please enter a string of bits to search (CTRL+Z to quit):
11111111111111111111111111111111111111111111111110
The bit string 11111111111111111111111111111111111111111111111110 is found
following these cells:
(0,0) (0,1) (0,2) (0,3) (0,4) (0,5) (0,6) (0,7) (0,8) (0,9) (0,10) (0,11)
(0,12) (0,13) (0,14) (0,15) (0,16) (0,17) (0,18) (0,19) (0,20) (0,21)
(0,22) (0,23) (0,24) (0,25) (0,26) (0,27) (0,28) (0,29) (0,30) (0,31)
(0,32) (0,33) (0,34) (0,35) (0,36) (0,37) (0,38) (0,39) (0,40) (0,41)
(0,42) (0,43) (0,44) (0,45) (0,46) (0,47) (0,48) (0,49)
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): ^Z
Program ended successfully!

Press any key to close this window . . .
```

Sample run 5:
File: matrix5.txt (see the homework package for the file)

```
Please enter the number of rows: 50
Please enter the number of columns: 50
Please enter the name of the input file that contains the matrix:
matrix5.txt

Please enter a string of bits to search (CTRL+Z to quit):
11111111111111111111111111111111111111111111111110
The bit string 11111111111111111111111111111111111111111111111110 is found
following these cells:
(0,0) (1,0) (2,0) (3,0) (4,0) (5,0) (6,0) (7,0) (8,0) (9,0) (10,0) (11,0)
(12,0) (13,0) (14,0) (15,0) (16,0) (17,0) (18,0) (19,0) (20,0) (21,0)
(22,0) (23,0) (24,0) (25,0) (26,0) (27,0) (28,0) (29,0) (30,0) (31,0)
(32,0) (33,0) (34,0) (35,0) (36,0) (37,0) (38,0) (39,0) (40,0) (41,0)
(42,0) (43,0) (44,0) (45,0) (46,0) (47,0) (48,0) (49,0)
---------------------------------------------------------
Please enter a string of bits to search (CTRL+Z to quit): ^Z
```

```
Program ended successfully!

Press any key to close this window . . .
```

What and where to submit (PLEASE READ, IMPORTANT)

You should prepare (or at least test) your program using MS Visual Studio 2012 C++. We will use the standard C++ compiler and libraries of the abovementioned platform while testing your homework. It'd be a good idea to write your name and last name in the program (as a comment line of course).

Submissions guidelines are below. Some parts of the grading process might be automatic. Students are expected to strictly follow these guidelines in order to have a smooth grading process. If you do not follow these guidelines, depending on the severity of the problem created during the grading process, 5 or more penalty points are to be deducted from the grade.

Name your cpp file that contains your main program using the following convention:

"SUCourseUserName_YourLastname_YourName_HWnumber.cpp"

Your SUCourse user name is your SUNet user name which is used for checking sabanciuniv e-mails. Do NOT use any spaces, non-ASCII and Turkish characters in the file name. For example, if your SUCourse user name is cago, name is Çağlayan, and last name is Özbugsızkodyazaroğlu, then the file name must be:

Cago_Ozbugsizkodyazaroglu_Caglayan_hw3.cpp

In some homework assignments, you may need to have more than one .cpp or .h files to submit. In this case, use the same filename format but add informative phrases after the hw number (e.g. Cago_Ozbugsizkodyazaroglu_Caglayan_hw3_myclass.cpp or Cago_Ozbugsizkodyazaroglu_Caglayan_hw3_myclass.h). However, do not add any other character or phrase to the file names. Sometimes, you may want to use some user defined libraries (such as strutils of Tapestry); in such cases, you have to provide the necessary .cpp and .h files of them as well. If you use standard C++ libraries, you do not need to provide extra files for them.

These source files are the ones that you are going to submit as your homework. However, even if you have a single file to submit, you have to compress it using ZIP format. To do so, first create a folder that follows the abovementioned naming convention ("SUCourseUserName_YourLastname_YourName_HWnumber"). Then, copy your source file(s) there. And finally compress this folder using WINZIP or WINRAR programs (or another mechanism). Please use "zip" compression. "rar" or another compression mechanism is NOT allowed. Our homework processing system works only with zip files. Therefore, make sure that the resulting compressed file has a zip extension. Check that your compressed file opens up correctly and it contains all of the files that belong to the latest version of your homework.

You will receive zero if your compressed zip file does not expand or it does not contain the correct files. The naming convention of the zip file is the same. The name of the zip file should be as follows:

SUCourseUserName_YourLastname_YourName_HWnumber.zip

For example, zubzipler_Zipleroglu_Zubeyir_hw4.zip is a valid name, but

Hw4_hoz_HasanOz.zip, HasanOzHoz.zip

are **NOT** valid names.

Submit via SUCourse ONLY! You will receive no credits if you submit by other means (e-mail, paper, etc.).

Successful submission is one of the requirements of the homework. If, for some reason, you cannot successfully submit your homework and we cannot grade it, your grade will be 0.

Good Luck!

Albert Levi, Vedat Peran