Pimpri Chinchwad College of Engineering
Computer Laboratory - IV

Assignment No - B1

1 Aim

8-Queens Matrix is Stored using JSON/XML having first Queen placed, use back-tracking to place remaining Queens to generate final 8-queens Matrix using Python. Create a backtracking scenario and use HPC architecture (Preferably BBB) for computation of next placement of a queen.

2 Objective

- To study 8 Queens problem, and implement it using Python

3 Software Requirements

- Linux
- Python

4 Mathematical Model

Input : Size of Board and Initial state of queen.
Output : 8 queen placed in 8*8 matrix in such a way that they do not attack each other.
Formula:

int PlaceQueen(int board[8], int row)
If (Can place queen on ith column)
PlaceQueen(newboard, 0)
Else
PlaceQueen(oldboard,oldplace+1)
End

5 Theory

5.1 8-queen

The 8 queen problem is a case of more general set of problems namely n queen problem. The basic idea: How to place n queen on n by n board, so that they dont attack each other.

As we can expect the complexity of solving the problem increases with n. We will briefly introduce solution by backtracking. First lets explain what is backtracking? The boar should be regarded as a set of constraints and the solution is simply satisfying all constraints.

For example: Q1 attacks some positions, therefore Q2 has to comply with these constraints and take place, not directly attacked by Q1. Placing Q3 is harder, since we have to satisfy constraints of Q1 and Q2.

Going the same way we may reach point, where the constraints make the placement of the next queen impossible. Therefore we need to relax the constraints and find new solution. To do this we are going backwards and finding new admissible solution. To keep everything in order we keep the simple rule: last placed, first displaced.

In other words if we place successfully queen on the ith column but cannot find solution for$(i + 1)th$ queen, then going backwards we will try to find other admissible solution for the ith queen first. This process is called backtrack .

5.2 Beaglebone Black

BeagleBone Black Overview:
The BeagleBone Black is the latest addition to the BeagleBoard.org family and like its predecessors, is designed to address the Open Source Community, early adopters, and anyone interested in a low cost ARM Cortex-A8 based processor. It has been equipped with a minimum set of features to allow the user to experience the power of the processor and is not intended as a

full development platform as many of the features and interfaces supplied by the processor are not accessible from the Beagle Bone Black via on board support of some interfaces. It is not a complete product designed to do any particular function. It is a foundation for experimentation and learning how to program the processor and to access the peripherals by the creation of your own software and hardware. It also offers access to many of the interfaces and allows for the use of add-on boards called capes, to add many different combinations of features. A user may also develop their own board or add their own circuitry.

Board Component Locations:
This section describes the key components on the board. It provides information on their location and function. Familiarize yourself with the various components on the board.

- Connectors, LEDs, and Switches
- DC Power is the main DC input that accepts 5V power.
- Power Button alerts the processor to initiate the power down sequence.
- 10/100 Ethernet is the connection to the LAN.
- Serial Debug is the serial debug port.
- USB Client is a mini USB connection to a PC that can also power the board.
- BOOT switch can be used to force a boot from the SD card.
- There are four blue LEDS that can be used by the user.
- Reset Button allows the user to reset the processor.
- uSD slot is where a uSD card can be installed.
- microHDMI connector is where the display is connected to.
- USB Host can be connected different USB interfaces such asWi-Fi, BT,Keyboard, etc

Features of Beaglebone Black

Table 2. BeagleBone Black Features

Feature		
Processor	Sitara AM3359AZCZ100 1GHz, 2000 MIPS	
Graphics Engine	SGX530 3D, 20M Polygons/S	
SDRAM Memory	512MB DDR3L 606MHZ	
Onboard Flash	2GB, 8bit Embedded MMC	
PMIC	TPS65217C PMIC regulator and one additional LDO.	
Debug Support	Optional Onboard 20-pin CTI JTAG, Serial Header	
Power Source	miniUSB USB or DC Jack	5VDC External Via Expansion Header
PCB	3.4" x 2.1"	6 layers
Indicators	1-Power, 2-Ethernet, 4-User Controllable LEDs	
HS USB 2.0 Client Port	Access to USB0, Client mode via miniUSB	
HS USB 2.0 Host Port	Access to USB1, Type A Socket, 500mA LS/FS/HS	
Serial Port	UART0 access via 6 pin 3.3V TTL Header. Header is populated	
Ethernet	10/100, RJ45	
SD/MMC Connector	microSD , 3.3V	
User Input	Reset Button Boot Button Power Button	
Video Out	16b HDMI, 1280x1024 (MAX) 1024x768,1280x720,1440x900 w/EDID Support	
Audio	Via HDMI Interface, Stereo	
Expansion Connectors	Power 5V, 3.3V , VDD_ADC(1.8V) 3.3V I/O on all signals McASP0, SPI1, I2C, GPIO(65), LCD, GPMC, MMC1, MMC2, 7 AIN(1.8V MAX), 4 Timers, 3 Serial Ports, CAN0, EHRPWM(0,2),XDMA Interrupt, Power button, Expansion Board ID (Up to 4 can be stacked)	
Weight	1.4 oz (39.68 grams)	

Key Components

- Sitara AM3359AZCZ100 is the processor for the board.
- Micron 512MB DDR3L is the Dual Data Rate RAM memory.
- TPS65217C PMIC provides the power rails to the various components on the board.
- SMSC Ethernet PHY is the physical interface to the network.
- Micron eMMC is an onboard MMC chip that holds up to 2GB of data.
- HDMI Framer provides control for an HDMI or DVI-D display with an adapter.

Connectivity

- Connect the small connector on the USB cable to the board as shown in Figure 1. The connector is on the bottom side of the board.

- Connect the large connector of the USB cable to your PC or laptop USB port.
- The board will power on and the power LED will be on as shown in Figure 2 below.

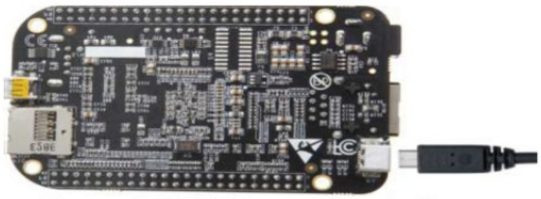


Figure 1. USB Connection to the Board

Figure 2. Board Power LED

- Apply Power: The final step is to plug in the DC power supply to the DC power jack as shown in Figure4 below.
- Booting the Board: As soon as the power is applied to the board, it will start the booting up process. When the board starts to boot the LEDs will come on in sequence as shown in Figure 5 below. It will take a few seconds for the status LEDs to come on, so be patient. The LEDs will be flashing in an erratic manner as it boots the Linux kernel.

Figure 4. External DC Power

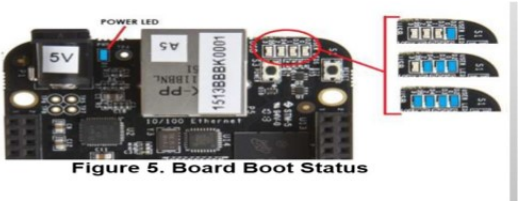


Figure 5. Board Boot Status

5.3 Steps to run program in Beaglebone Black

1. In red hat terminal we need to type following command for accessing the beagle bone terminal. ssh 192.168.7.2
2. Open another redhat terminal for copying the python program from redhat to beagle bone by using command: scp filename.py root@192.168.7.2:
3. Now open beagle bone terminal to check whether program is copied or not by using command ls.
4. To run the program write following command in beagle bone terminal. python filename.py

6 Testing

6.1 Black Box Testing

Black-box testing is a method of software testing that examines the function- ability of an application without peering into its internal structures or workings.
This method of test can be applied to virtually every level of software testing:unit, integration, system and acceptance
Typical Input Data: Position of first queen
Expected Output: Possible positions of remaining 7 queens to satisfy the 8 Queen Problem

Example:
Input:

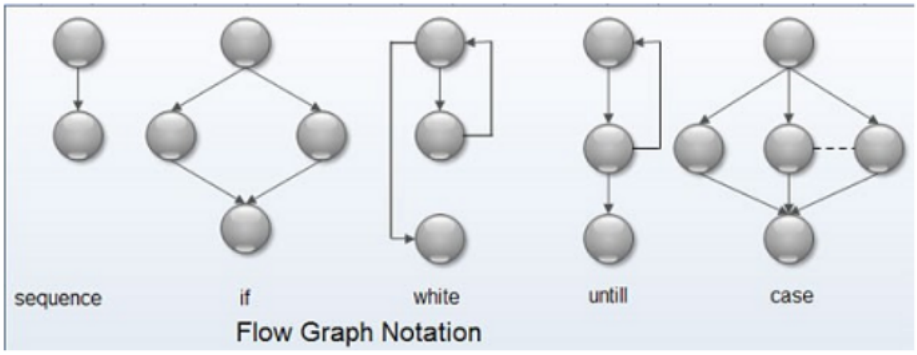
$$X = 4, Y = 4$$

Output:
$[2, 0, 6, 4, 7, 1, 3, 5]$
$[2, 5, 1, 4, 7, 0, 6, 3]$
$[3, 1, 6, 4, 0, 7, 5, 2]$
$[3, 1, 7, 4, 6, 0, 2, 5]$
$[3, 5, 0, 4, 1, 7, 2, 6]$
$[3, 7, 0, 4, 6, 1, 5, 2]$
$[5, 3, 0, 4, 7, 1, 6, 2]$
$[6, 3, 1, 4, 7, 0, 2, 5]$
8

6.2 White Box Testing

White-box testing (also known as clear box testing, glass box testing, transparent box testing, and structural testing) is a method of testing software that tests internal structures or workings of an application, as opposed to its functionality(i.e. black-box testing). While developing test cases for white box testing it is understood that complete testing is impossible. In White Box testing we checkup to which extent the code is being executed, i.e. Covered. There are different kinds of coverage like, statement coverage, path coverage, etc. We will use one of the most popular technique i.e. Statement coverage. Statement coverage is a white box testing technique, which involves the execution of all the statements at least once in the source code. It

is a metric, which is used to calculate and measure the number of statements in the source code which have been executed. For this, we will use Flow Graphs. Flow graphs are, Syntactic abstraction of source code Resembling to classical flow charts Forms the basis for white box test case generation principles.Conventions of flow graph notation,
Sample Input: For integer array int A[] = 1,2,3,4,5,6,7,8,9,10; Function is passed
following arguments: Binary Search(A, 1, 10, 7);
Output obtained: Entered second Entered third Entered first 6
The underlined nodes are the ones being tested. The above output shows that every test region is covered for given input.



Sample Input:
Given coordinates (X,Y) = 4,4
Output obtained:

$ValidCase : [0, 4, 7, 5, 2, 6, 1, 3]$
$ValidCase : [0, 5, 7, 2, 6, 3, 1, 4]$
$ValidCase : [0, 6, 3, 5, 7, 1, 4, 2]$
$ValidCase : [0, 6, 4, 7, 1, 3, 5, 2]$
$ValidCase : [1, 3, 5, 7, 2, 0, 6, 4]$
$ValidCase : [1, 4, 6, 0, 2, 7, 5, 3]$
$ValidCase : [1, 4, 6, 3, 0, 7, 5, 2]$
$ValidCase : [1, 5, 0, 6, 3, 7, 2, 4]$
$ValidCase : [1, 5, 7, 2, 0, 3, 6, 4]$
$ValidCase : [1, 6, 2, 5, 7, 4, 0, 3]$
$ValidCase : [1, 6, 4, 7, 0, 3, 5, 2]$
$ValidCase : [1, 7, 5, 0, 2, 4, 6, 3]$
$SolutionCase : [2, 0, 6, 4, 7, 1, 3, 5]$

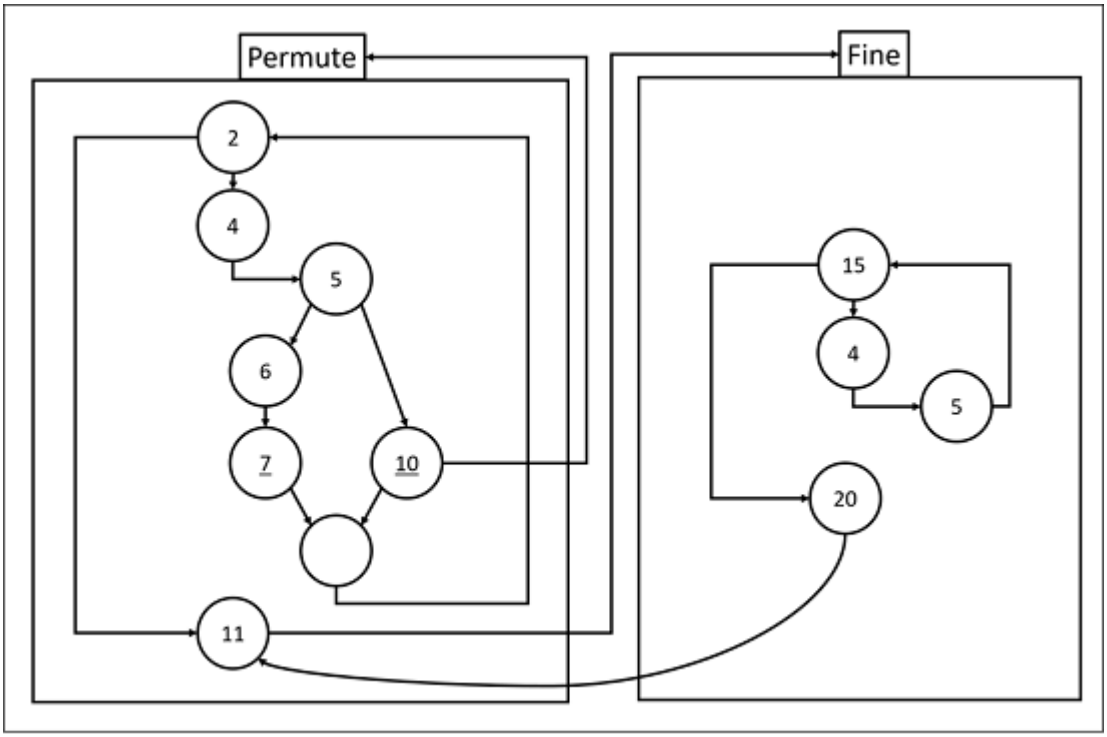
```
def Permute(queens, row):
    for i in range(8):
        queens[row] = i
        if Fine(queens, row):
            if row == 7:
                if queens[x-1] == y:
                    print(queens)
                    globals()["solutions"] = globals()["solutions"] + 1
            else:
                Permute(queens, row+1)

def Fine(queens, row):
    c = 0
    derga = True
    for i in range(row):
        c, cur, oth = c+1, queens[row], queens[row-i-1]
        if (cur == oth) or (cur-c == oth) or (cur+c == oth):
            derga = False
            break
    return(derga)

```

Function

$ValidCase : [2, 0, 6, 4, 7, 1, 3, 5]$
$ValidCase : [2, 4, 1, 7, 0, 6, 3, 5]$
$ValidCase : [2, 4, 1, 7, 5, 3, 6, 0]$
$ValidCase : [2, 4, 6, 0, 3, 1, 7, 5]$
$ValidCase : [2, 4, 7, 3, 0, 6, 1, 5]$
$SolutionCase : [2, 5, 1, 4, 7, 0, 6, 3]$
$ValidCase : [2, 5, 1, 4, 7, 0, 6, 3]$
$ValidCase : [2, 5, 1, 6, 0, 3, 7, 4]$
$ValidCase : [2, 5, 1, 6, 4, 0, 7, 3]$
$ValidCase : [2, 5, 3, 0, 7, 4, 6, 1]$
$ValidCase : [2, 5, 3, 1, 7, 4, 6, 0]$
$ValidCase : [2, 5, 7, 0, 3, 6, 4, 1]$
$ValidCase : [2, 5, 7, 0, 4, 6, 1, 3]$
$ValidCase : [2, 5, 7, 1, 3, 0, 6, 4]$
$ValidCase : [2, 6, 1, 7, 4, 0, 3, 5]$
$ValidCase : [2, 6, 1, 7, 5, 3, 0, 4]$
$ValidCase : [2, 7, 3, 6, 0, 5, 1, 4]$
$ValidCase : [3, 0, 4, 7, 1, 6, 2, 5]$
$ValidCase : [3, 0, 4, 7, 5, 2, 6, 1]$
$ValidCase : [3, 1, 4, 7, 5, 0, 2, 6]$



Flow Graph Notation

$ValidCase$: $[3, 1, 6, 2, 5, 7, 0, 4]$
$ValidCase$: $[3, 1, 6, 2, 5, 7, 4, 0]$
$SolutionCase$: $[3, 1, 6, 4, 0, 7, 5, 2]$
$ValidCase$: $[3, 1, 6, 4, 0, 7, 5, 2]$
$SolutionCase$: $[3, 1, 7, 4, 6, 0, 2, 5]$
$ValidCase$: $[3, 1, 7, 4, 6, 0, 2, 5]$
$ValidCase$: $[3, 1, 7, 5, 0, 2, 4, 6]$
$SolutionCase$: $[3, 5, 0, 4, 1, 7, 2, 6]$
$ValidCase$: $[3, 5, 0, 4, 1, 7, 2, 6]$
$ValidCase$: $[3, 5, 7, 1, 6, 0, 2, 4]$
$ValidCase$: $[3, 5, 7, 2, 0, 6, 4, 1]$
$ValidCase$: $[3, 6, 0, 7, 4, 1, 5, 2]$
$ValidCase$: $[3, 6, 2, 7, 1, 4, 0, 5]$
$ValidCase$: $[3, 6, 4, 1, 5, 0, 2, 7]$
$ValidCase$: $[3, 6, 4, 2, 0, 5, 7, 1]$
$ValidCase$: $[3, 7, 0, 2, 5, 1, 6, 4]$
$SolutionCase$: $[3, 7, 0, 4, 6, 1, 5, 2]$
$ValidCase$: $[3, 7, 0, 4, 6, 1, 5, 2]$
$ValidCase$: $[3, 7, 4, 2, 0, 6, 1, 5]$

$ValidCase : [4, 0, 3, 5, 7, 1, 6, 2]$
$ValidCase : [4, 0, 7, 3, 1, 6, 2, 5]$
$ValidCase : [4, 0, 7, 5, 2, 6, 1, 3]$
$ValidCase : [4, 1, 3, 5, 7, 2, 0, 6]$
$ValidCase : [4, 1, 3, 6, 2, 7, 5, 0]$
$ValidCase : [4, 1, 5, 0, 6, 3, 7, 2]$
$ValidCase : [4, 1, 7, 0, 3, 6, 2, 5]$
$ValidCase : [4, 2, 0, 5, 7, 1, 3, 6]$
$ValidCase : [4, 2, 0, 6, 1, 7, 5, 3]$
$ValidCase : [4, 2, 7, 3, 6, 0, 5, 1]$
$ValidCase : [4, 6, 0, 2, 7, 5, 3, 1]$
$ValidCase : [4, 6, 0, 3, 1, 7, 5, 2]$
$ValidCase : [4, 6, 1, 3, 7, 0, 2, 5]$
$ValidCase : [4, 6, 1, 5, 2, 0, 3, 7]$
$ValidCase : [4, 6, 1, 5, 2, 0, 7, 3]$
$ValidCase : [4, 6, 3, 0, 2, 7, 5, 1]$
$ValidCase : [4, 7, 3, 0, 2, 5, 1, 6]$
$ValidCase : [4, 7, 3, 0, 6, 1, 5, 2]$
$ValidCase : [5, 0, 4, 1, 7, 2, 6, 3]$
$ValidCase : [5, 1, 6, 0, 2, 4, 7, 3]$
$ValidCase : [5, 1, 6, 0, 3, 7, 4, 2]$
$ValidCase : [5, 2, 0, 6, 4, 7, 1, 3]$
$ValidCase : [5, 2, 0, 7, 3, 1, 6, 4]$
$ValidCase : [5, 2, 0, 7, 4, 1, 3, 6]$
$ValidCase : [5, 2, 4, 6, 0, 3, 1, 7]$
$ValidCase : [5, 2, 4, 7, 0, 3, 1, 6]$
$ValidCase : [5, 2, 6, 1, 3, 7, 0, 4]$
$ValidCase : [5, 2, 6, 1, 7, 4, 0, 3]$
$ValidCase : [5, 2, 6, 3, 0, 7, 1, 4]$
$SolutionCase : [5, 3, 0, 4, 7, 1, 6, 2]$
$ValidCase : [5, 3, 0, 4, 7, 1, 6, 2]$
$ValidCase : [5, 3, 1, 7, 4, 6, 0, 2]$
$ValidCase : [5, 3, 6, 0, 2, 4, 1, 7]$
$ValidCase : [5, 3, 6, 0, 7, 1, 4, 2]$
$ValidCase : [5, 7, 1, 3, 0, 6, 4, 2]$
$ValidCase : [6, 0, 2, 7, 5, 3, 1, 4]$
$ValidCase : [6, 1, 3, 0, 7, 4, 2, 5]$
$ValidCase : [6, 1, 5, 2, 0, 3, 7, 4]$
$ValidCase : [6, 2, 0, 5, 7, 4, 1, 3]$
$ValidCase : [6, 2, 7, 1, 4, 0, 5, 3]$
$SolutionCase : [6, 3, 1, 4, 7, 0, 2, 5]$

$ValidCase$: $[6, 3, 1, 4, 7, 0, 2, 5]$
$ValidCase$: $[6, 3, 1, 7, 5, 0, 2, 4]$
$ValidCase$: $[6, 4, 2, 0, 5, 7, 1, 3]$
$ValidCase$: $[7, 1, 3, 0, 6, 4, 2, 5]$
$ValidCase$: $[7, 1, 4, 2, 0, 6, 3, 5]$
$ValidCase$: $[7, 2, 0, 5, 1, 4, 6, 3]$
$ValidCase$: $[7, 3, 0, 2, 5, 1, 6, 4]$
8

6.3 Positive Negative Testing

Positive Testing :
If proper position is entered then all solutions are found.
Negative Testing :
if uknown position is entered the program must fail to produce outputs

6.4 Advanced Testing Technique

Google Testing Framework can be used in future for testing the code

7 Algorithm

#move right(column increment)
for col in range(1, size + 1):
#skip the row in which queen was placed initially
if row == x and col == y:
placequeen(row + 1)
if not danger(row, col):
board.append((row, col))
#move down(row increment)
placequeen(row + 1)
board.remove((row,col))

8 Conclusion

We have studied 8 - Queens problem and have successfully implemented it in python.

Roll No.	Name of Student	Date of Performance	Date of Submission
302	Abhinav Bakshi	20/1/16	3/2/16

9 Plagarism Report

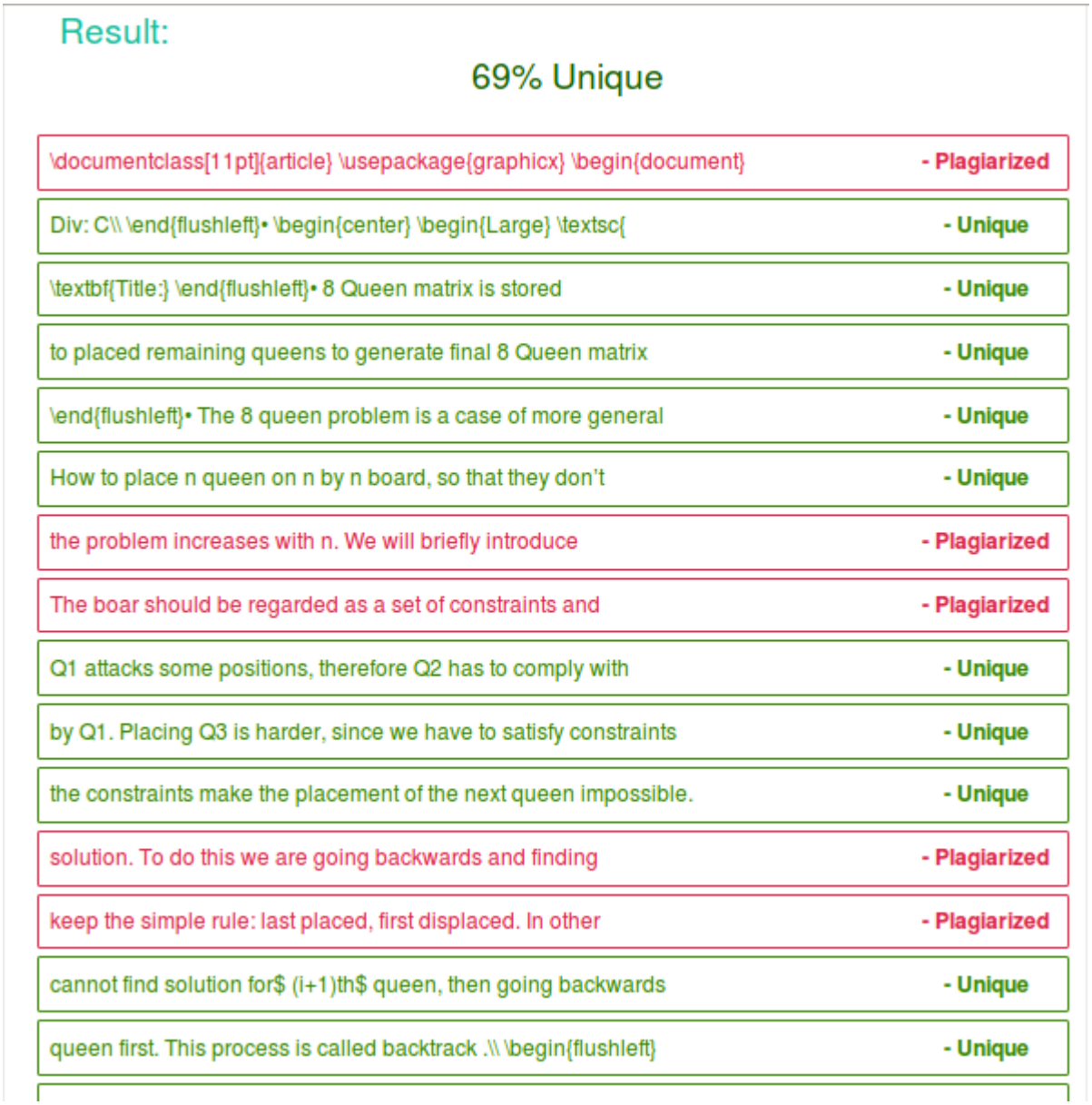


Figure 1: Plagarism Checker: www.smallseotools.com/plagarism-checker