Imam Mohammad Ibn Saud Islamic University

College of Computer and Information Sciences

Computer Science Department

CS 361 – Artificial Intelligence
Project: Sudoku puzzle

Semester:	(**1st semester**) 2023-1445
Due Date:	**Saturday** 28/October/2023
Instructor:	**Dr.** Hachemi Bennaceur

Team Member's Name	Students ID's
Abdulrahman Afit Askar Alanazi	441013506
Yazeed Abdullah Bin Shiah	441022937
Salman Wadi Alanazi	441016559
Azam Abdullah Allahim	438012114

Contents:

1.Introduction:

The Sudoku Solver project aims to solve Sudoku puzzles by representing them as Constraint Satisfaction Problems (CSP) and implementing various algorithms to find the solution. The project utilizes the concepts of backtracking, forward checking, and maintaining arc-consistency to efficiently solve Sudoku puzzles.

2.Formulating the Sudoku Puzzle as a Constraint Satisfaction Problem (CSP):

Sudoku is a popular logic-based, number-placement puzzle. In this section, we formulate the Sudoku puzzle as a Constraint Satisfaction Problem (CSP). The aim is to fill a 9x9 grid in such a way that each row, each column, and each of the nine 3x3 grids (also known as boxes) contain all of the numbers from 1 to 9..

- **Variables**: cell_1, cell_2, ..., cell_81: These represent each cell in the 9x9 Sudoku grid.

- **Domains**: Each cell (cell_1, cell_2, ..., cell_81) can take a value from the set {1, 2, 3, 4, 5, 6, 7, 8, 9}.

- **Constraints**:
  1. Row Constraints: Each row must have unique values. This constraint applies to all 9 rows in the grid. For example, in the first row, cell_1 should not be equal to cell_2, cell_3, ..., cell_9.
  2. Column Constraints: Each column must also have unique values. This constraint applies to all 9 columns in the grid. For example, in the first column, cell_1 should not be equal to cell_10, cell_19, ..., cell_81.

  Block Constraints: Each 3x3 block must have unique values. The Sudoku grid is divided into nine 3x3 blocks. This constraint applies to each block. For example, in the first 3x3 block, the cells involved would be cell_1, cell_2, cell_3, cell_10, cell_11, cell_12, cell_19, cell_20, cell_21, and all these should have unique values

- **Implementing the standard backtracking (BT) method**:

The project's goal is to implement the backtracking algorithm, which is a brute-force search strategy that searches the solution space systematically by assigning values to variables and retracing when a constraint violation occurs Figure below show the pseudocode to illustrate how we translated the theoretical aspects into a functional program.

```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (Xi, Xj) <- REMOVE-FIRST(queue)
      if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] - {Xj} do
            add (Xk, Xi) to queue

function REMOVE-INCONSISTENT-VALUES(Xi, Xj) returns true iff we remove a value
   removed <- false
   for each x in DOMAIN[Xi] do
      if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint between Xi and Xj
         then delete x from DOMAIN[Xi]; removed <- true
   return removed
```

Fig. 1. Pseudocode for BT

- **Implementing the Forward Checking (FC) algorithm:** The project's goal is to implement the Forward Checking algorithm, which improves the backtracking process by keeping track of the remaining valid values for unassigned variables. It prunes the search space by removing values that violate restrictions.

```
procedure FC(i)
%Tries to instantiate Vi, then recurses
   for each v_l^i ∈ Di
      si <- v_l^i
      if Domain_l^i = 0 then
         if i = N then
            print s1, ..., sN
         else
            if Check-Forward(i) then
               FC(i+1)
            Restore(i)

function Check-Forward(i)
%Checks si against future variables
   for j = i + 1 to N
      dwo = true
      for each v_m^j ∈ Dj
         if Domain_m^j = 0 then
            if (si, v_m^j) ∈ C{i,j} then
               dwo = false
            else
               Domain_m^j <- i
      if dwo then return(false)
   return(true)

procedure Restore(i)
%Returns Domain to previous state
   for j = i + 1 to N
      for each v_m^j ∈ Dj
         if Domain_m^j = i then
            Domain_m^j <- 0
```

Fig. 2. Pseudocode for FC

• Implementing the Maintaining Arc-Consistency (MAC) algorithm: The project's goal is to implement the MAC algorithm, which enhances the efficiency of constraint fulfilment by enforcing arc-consistency. It assures that any value in a domain is compatible with the constraints of all other variables.

```
function AC-3( csp) returns the CSP, possibly with reduced domains
   inputs: csp, a binary CSP with variables {X1, X2, ..., Xn}
   local variables: queue, a queue of arcs, initially all the arcs in csp

   while queue is not empty do
      (Xi, Xj) ← REMOVE-FIRST(queue)
      if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
         for each Xk in NEIGHBORS[Xi] - {Xj} do
            add (Xk, Xi) to queue
```

```
function REMOVE-INCONSISTENT-VALUES(Xi, Xj) returns true iff we remove a value
   removed ← false
   for each x in DOMAIN[Xi] do
      if no value y in DOMAIN[Xj] allows (x,y) to satisfy the constraint between Xi and Xj
         then delete x from DOMAIN[Xi];  removed ← true
   return removed
```

Fig. 2. Pseudocode for MAC

- By achieving these aims, the project intends to provide a comprehensive Sudoku solver capable of solving puzzles of varying difficulties. The implemented algorithms and heuristics aim to improve the efficiency and speed of finding the solution while adhering to the Sudoku constraints as we going to test our functions in 20 different Sudoku puzzle.

3.Backtracking (BT) algorithm:

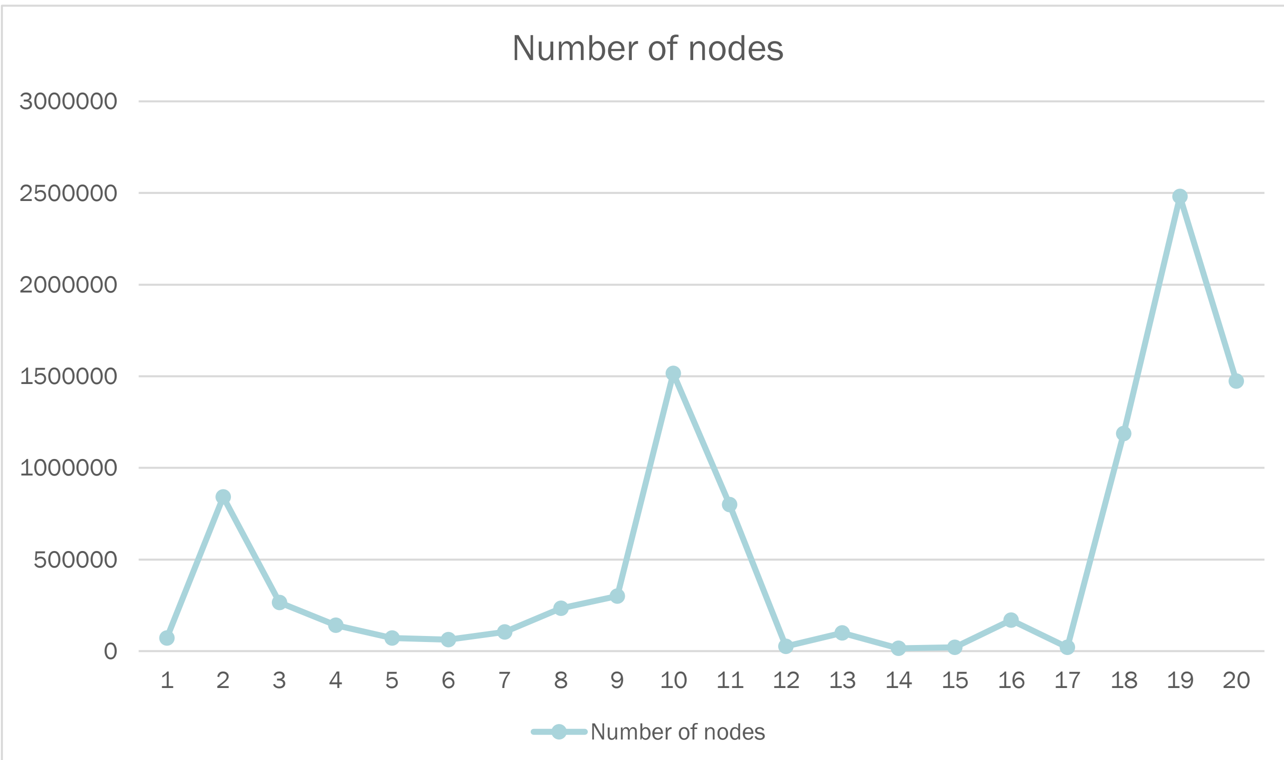
The backtracking algorithm is a versatile approach to that requires considering each potential solution. It is frequently applied to Sudoku-style constraint satisfaction situations. Choosing an empty cell, entering a value, and proceeding to the next empty cell if the value is accurate is how it operates. It goes back to the previous cell and tries a different value if a restriction is broken. This process keeps going until a solution is discovered or every other option is used. Recursion or a stack are used in backtracking, a depth-first search procedure, to monitor states. Further methods such as arc consistency, heuristics, and forward checking can decrease search space and increase performance.

Here is our Python function implementing BT.

```
def solve_backtracking(self):
    if not self.find_empty():
        return True

    row, col = self.find_empty()

    for num in range(1, 10):
        self.nodes_count += 1  # زيادة العداد بعد مراجعة جميع الأرقام
        if self.is_valid(row, col, num):
            self.puzzle[row][col] = num
            if self.solve_backtracking():
                return True
            self.puzzle[row][col] = 0

    return False

def find_empty(self):
    for i in range(9):
        for j in range(9):
            if self.puzzle[i][j] == 0:
                return i, j
    return None

def is_valid(self, row, col, num):
    for i in range(9):
        if self.puzzle[row][i] == num or self.puzzle[i][col] == num:
            self.nodes_count += 1  # زيادة العداد لفحص الصفوف والأعمدة
            return False

    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(3):
        for j in range(3):
            if self.puzzle[start_row + i][start_col + j] == num:
```

3.1-Number of nodes:

BT



Maximum number of node is 2480435

Average number of node is 602926

Minimum number of node is 15255

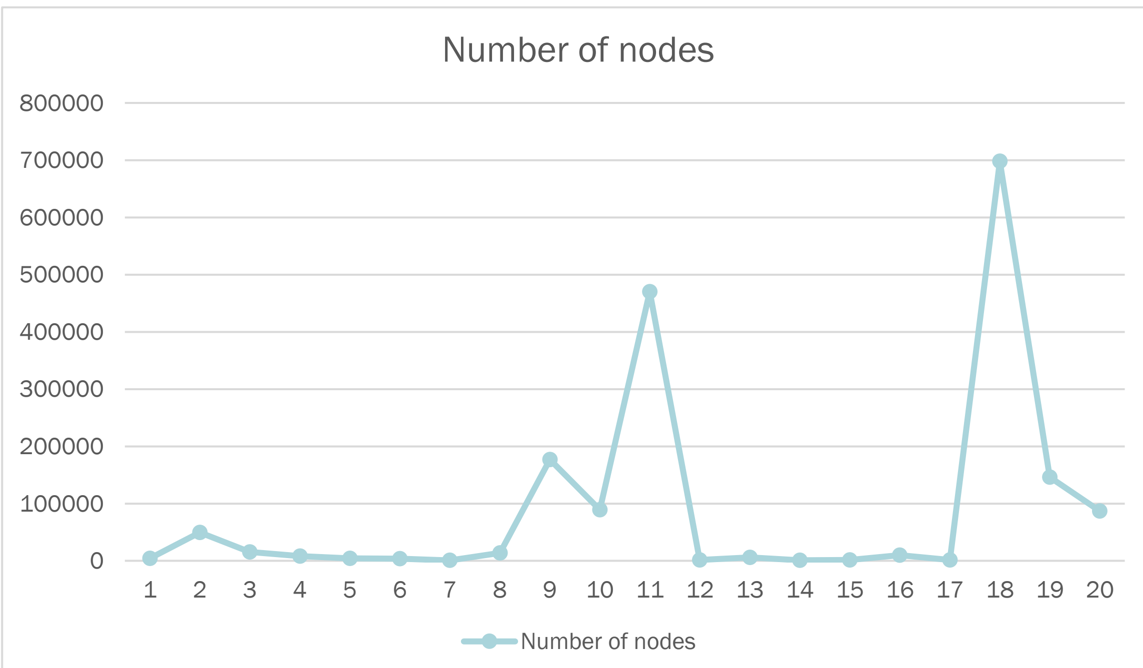
4.Forward Checking (FC) algorithm:

Through search space reduction, the Forward Checking (FC) algorithm improves the efficiency of solving constraint satisfaction problems (CSPs). It improves affected variable domains, assigns values to empty cells in Sudoku or other CSPs, and eliminates assigned values from corresponding cells. A domain reverts if it turns empty. By updating variable domains through distribution, forward checking minimizes branching and detects invalid assignments early on. It may greatly decrease the search space, but it cannot ensure that every CSP can find a solution. Here is Forward Checking function implemented in Python.

```python
def solve_forward_checking(self):
    if not self.is_valid_board():
        return False

    if not self.find_empty():
        return True

    row, col = self.find_empty()
    valid_values = self.get_valid_values(row, col)

    if not valid_values:
        return False

    for num in valid_values:
        self.puzzle[row][col] = num
        self.nodes_count += 1
        if self.solve_forward_checking():
            return True

        self.puzzle[row][col] = 0

    return False

def find_empty(self):
    for i in range(9):
        for j in range(9):
            if self.puzzle[i][j] == 0:
                return i, j
    return None

def get_valid_values(self, row, col):
    valid_values = []
    for num in range(1, 10):
        if self.is_valid(row, col, num):
            valid_values.append(num)
    return valid_values

def solve(self, algorithm='forward_checking'):
    self.nodes_count = 0

    if algorithm == 'forward_checking':
        if self.solve_forward_checking():
            return self.puzzle, self.nodes_count
        else:
            return None, self.nodes_count
```

4.1-Number of nodes:

FC



Maximum number of node is 697927
Average number of node is 55462
Minimum number of node is 653

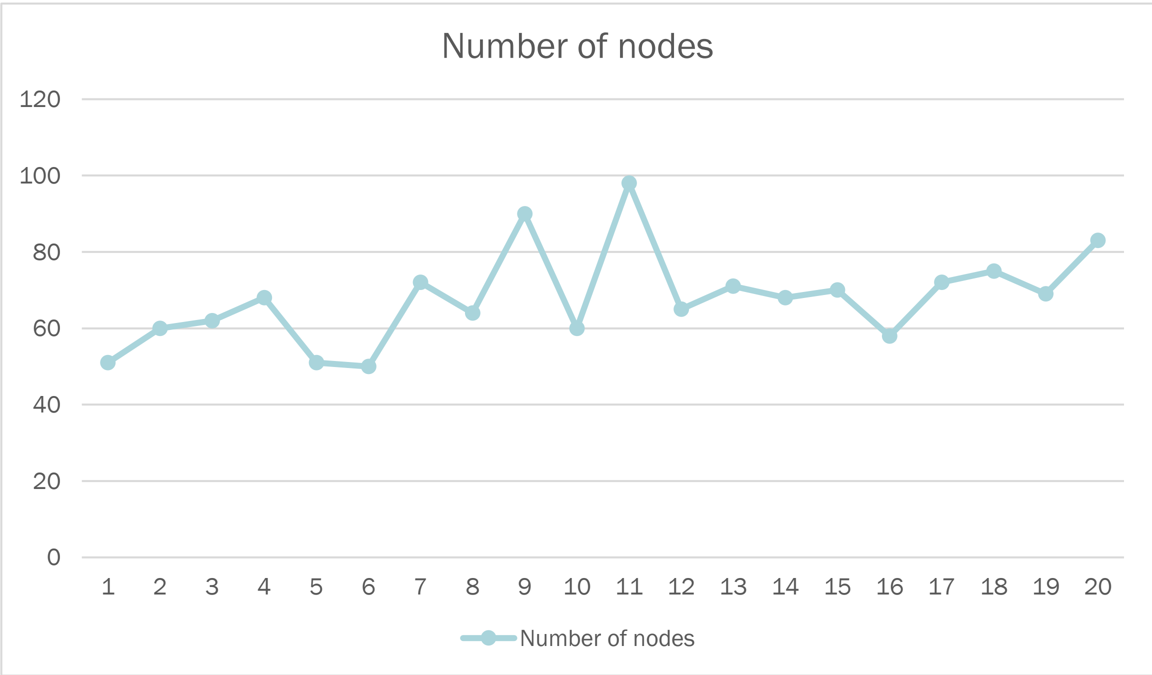
5.Maintaining Arc-Consistency (MAC) algorithm:

A constraint satisfaction problem's (CSP) values will be ensured to be consistent with the constraints applied using the Maintaining Arc-Consistency (MAC) strategy. It uses a queue of arcs to move constraints between variables repeatedly. The algorithm determines whether the domain of one variable contains a consistent value that satisfies the limitation of another variable. Values are eliminated from the domain if no consistent value can be discovered. The MAC technique reduces the search space and outperforms forward checking by taking into account all restrictions for a variable. For larger issues, it may be computationally costly. Below is our Implementation for MAC function in Python.

```
def ac3(self, board):
    queue = []
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                continue
            for x in range(9):
                if x == i:
                    continue
                queue.append((i, j, x, j))
            for y in range(9):
                if y == j:
                    continue
                queue.append((i, j, i, y))
    while queue:
        i, j, x, y = queue.pop(0)
        if self.revise(board, i, j, x, y):
            if not any(cell[0] == x and cell[1] == y for cell in queue):
                queue.append((x, y, x, y))

def _solve_maintaining_arc_consistency(self, board):
    empty_cell = self.find_empty_cell(board)
    if not empty_cell:
        return True
    row, col = empty_cell
    for num in range(1, 10):
        if self.is_valid(board, row, col, num):
            board[row][col] = num
            self.ac3(board)
            if self._solve_maintaining_arc_consistency(board):
                self.branch_counter += 1
                return True
            board[row][col] = 0
    return False

def solve_maintaining_arc_consistency(self):
    board = [row[:] for row in self.puzzle]
    if not self._solve_maintaining_arc_consistency(board):
        return None
    return board
```

5.1-Number of nodes:

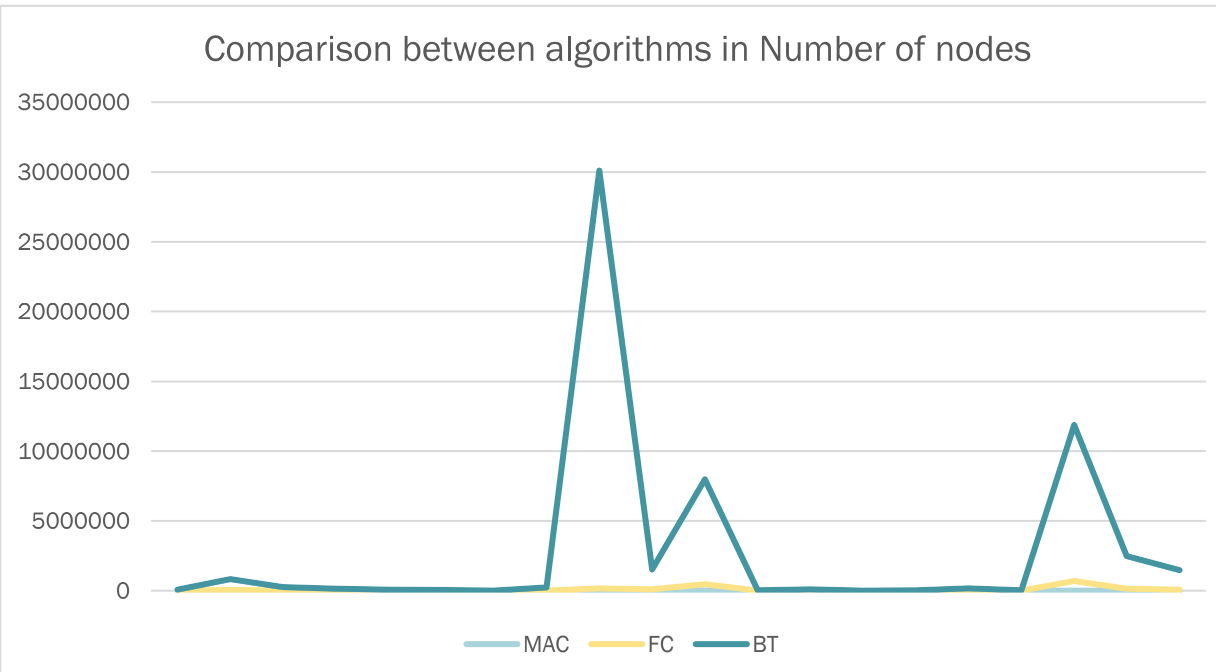
MAC



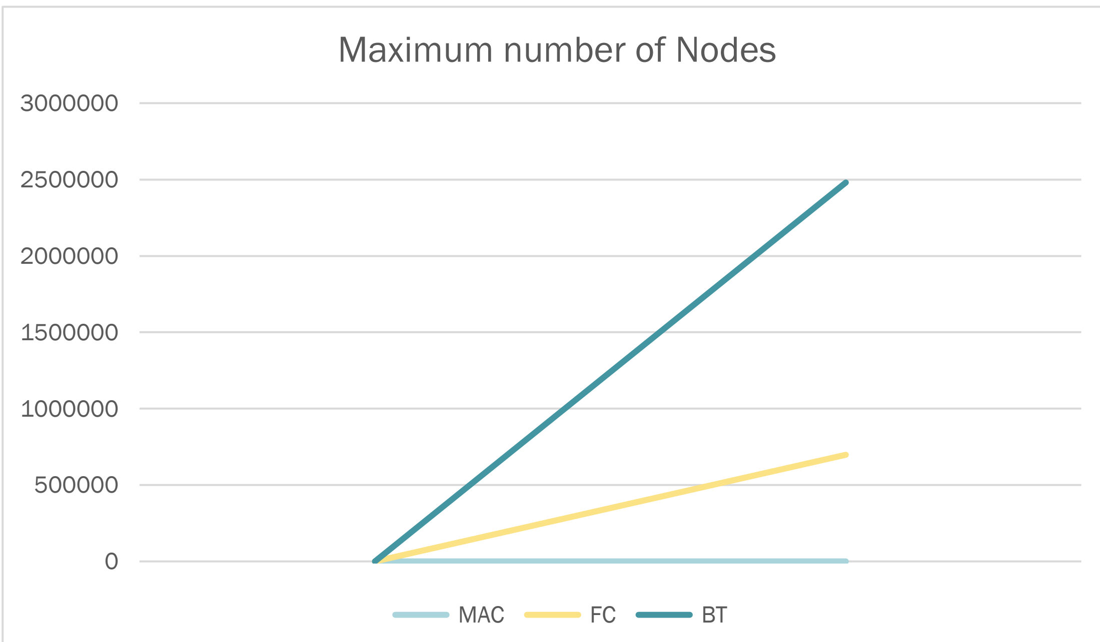
Maximum number of node is 98 Average number of node is 66

Minimum number of node is 50

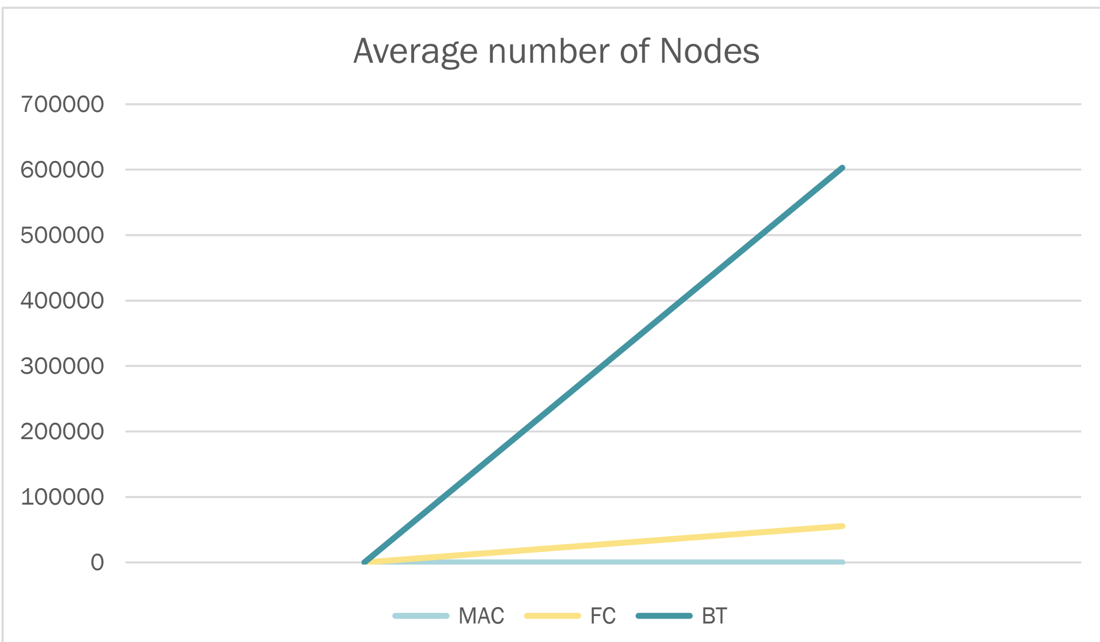
6.Statisatics Comparison :



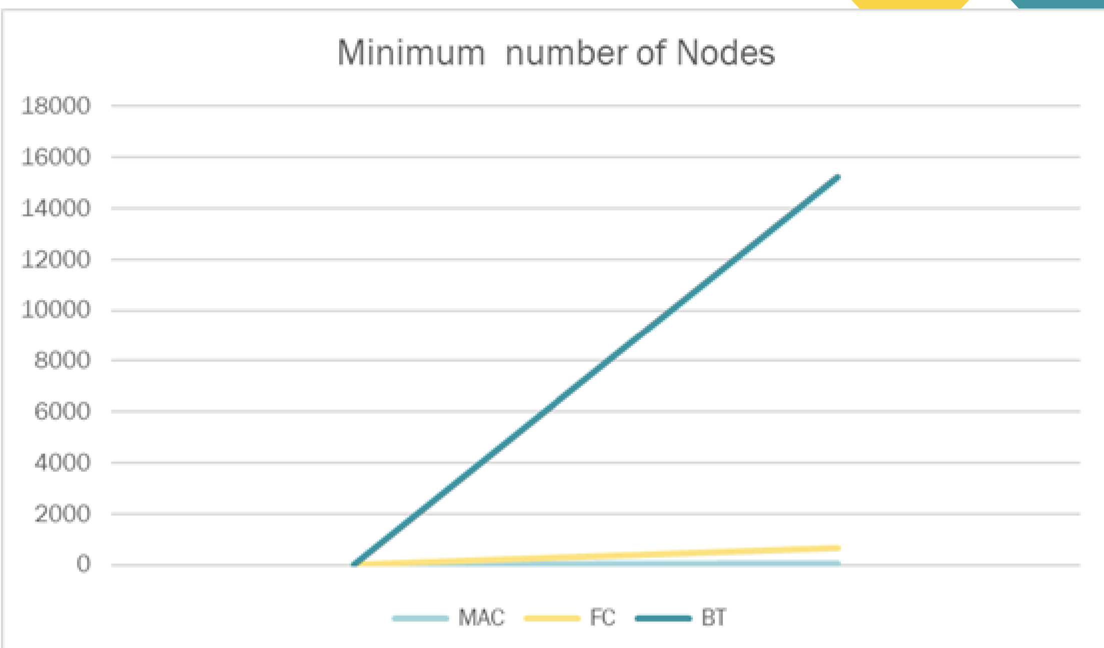
6.1-Maximum number of Nodes:



6.2- Average number of Nodes:



6.3- Minimum number of Nodes:



7. Conclusion:

Backtracking (BT), Forward checking (FC), and Maintaining arc-consistency (MAC) are three different algorithmic approaches used to solve constraint satisfaction problems (CSPs) like Sudoku puzzles. Each of these methods has its strengths and limitations. Backtracking is a general-purpose algorithm that exhaustively explores the solution space by systematically trying alternative values for each variable. While it is guaranteed to find a solution if one exists, it can be computationally expensive for complex problems, as indicated by its maximum number of nodes examined, which was 2,480,435.

The average and minimum numbers of nodes examined were 602,926 and 15,255, respectively. Forward checking is an extension of backtracking that maintains and updates the domains of unassigned variables during the search process. This approach decreases the branching factor and prunes the search space by identifying and eliminating incorrect assignments early on. It has a maximum number of nodes examined of 697,927, with an average of 55,462 nodes and a minimum of 653 nodes. While it can significantly compress the search space, it does not guarantee a solution for all CSP.

Maintaining arc-consistency (MAC) is a more advanced algorithm that ensures all values in variable domains are consistent with the constraints imposed by other variables. It iteratively propagates knowledge about constraints, reducing the search space and helping to identify conflicts early. MAC's maximum number of nodes examined is significantly lower at 98, with an average of 66 nodes and a minimum of 50 nodes. However, it can be computationally costly for large CSP. In summary, the choice of algorithm depends on the specific problem and its characteristics. Backtracking is a reliable method but may be slow for complex problems. Forward checking provides faster answers in many cases, but it is not a guaranteed solution. Maintaining arc-consistency is the most efficient in terms of the number of nodes examined but comes with its own computational challenges. Therefore, the selection of the appropriate algorithm should consider the problem's complexity and requirements.