U18C0018

Shubham Shekhaliya

AIML

Assignment-4

**Implement N queens problem using below algorithms in prolog.**

**Compare the complexity of both algorithms.**

**1-> Depth First Search**

**DFS.py**

from queue import Queue

N=5

class NQueens:

    def \_\_init\_\_(self, size):

        self.size = size

    def solve\_dfs(self):

        if self.size < 1:

            return []

        solutions = []

        stack = [[]]

        while stack:

            solution = stack.pop()

            if self.conflict(solution):

                continue

            row = len(solution)

            if row == self.size:

                solutions.append(solution)

                continue

            for col in range(self.size):

                queen = (row, col)

                queens = solution.copy()

                queens.append(queen)

                stack.append(queens)

        return solutions

    def conflict(self, queens):

        for i in range(1, len(queens)):

            for j in range(0, i):

                a, b = queens[i]

                c, d = queens[j]

                if a == c or b == d or abs(a - c) == abs(b - d):

                    return True

        return False

    def print(self, queens):

        for i in range(self.size):

            for j in range(self.size):

                p = 'Q' if (i, j) in queens else '-'

                print('%s ' % p, end='')

            print()

def main():

    n\_queens = NQueens(N)

    dfs\_solutions = n\_queens.solve\_dfs()

    for i, solution in enumerate(dfs\_solutions):

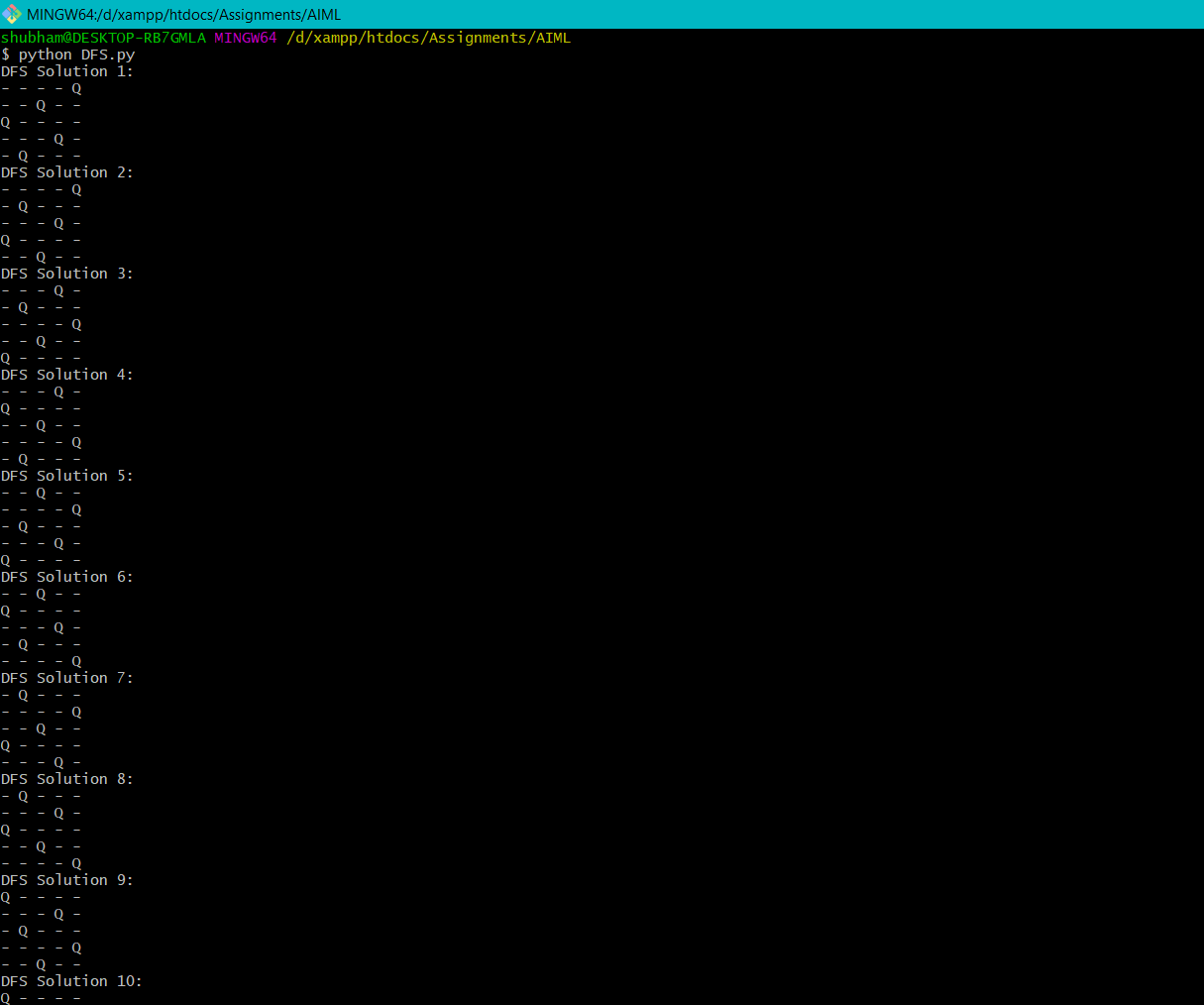
        print('DFS Solution %d:' % (i + 1))

        n\_queens.print(solution)

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**Ouptput:-**

****

**2-> Breadth First Search**

**BFS.py**

from queue import Queue

N=5

class NQueens:

    def \_\_init\_\_(self, size):

        self.size = size

    def solve\_bfs(self):

        if self.size < 1:

            return []

        solutions = []

        queue = Queue()

        queue.put([])

        while not queue.empty():

            solution = queue.get()

            if self.conflict(solution):

                continue

            row = len(solution)

            if row == self.size:

                solutions.append(solution)

                continue

            for col in range(self.size):

                queen = (row, col)

                queens = solution.copy()

                queens.append(queen)

                queue.put(queens)

        return solutions

    def conflict(self, queens):

        for i in range(1, len(queens)):

            for j in range(0, i):

                a, b = queens[i]

                c, d = queens[j]

                if a == c or b == d or abs(a - c) == abs(b - d):

                    return True

        return False

    def print(self, queens):

        for i in range(self.size):

            for j in range(self.size):

                p = 'Q' if (i, j) in queens else '-'

                print('%s ' % p, end='')

            print()

def main():

    n\_queens = NQueens(N)

    bfs\_solutions = n\_queens.solve\_bfs()

    for i, solution in enumerate(bfs\_solutions):

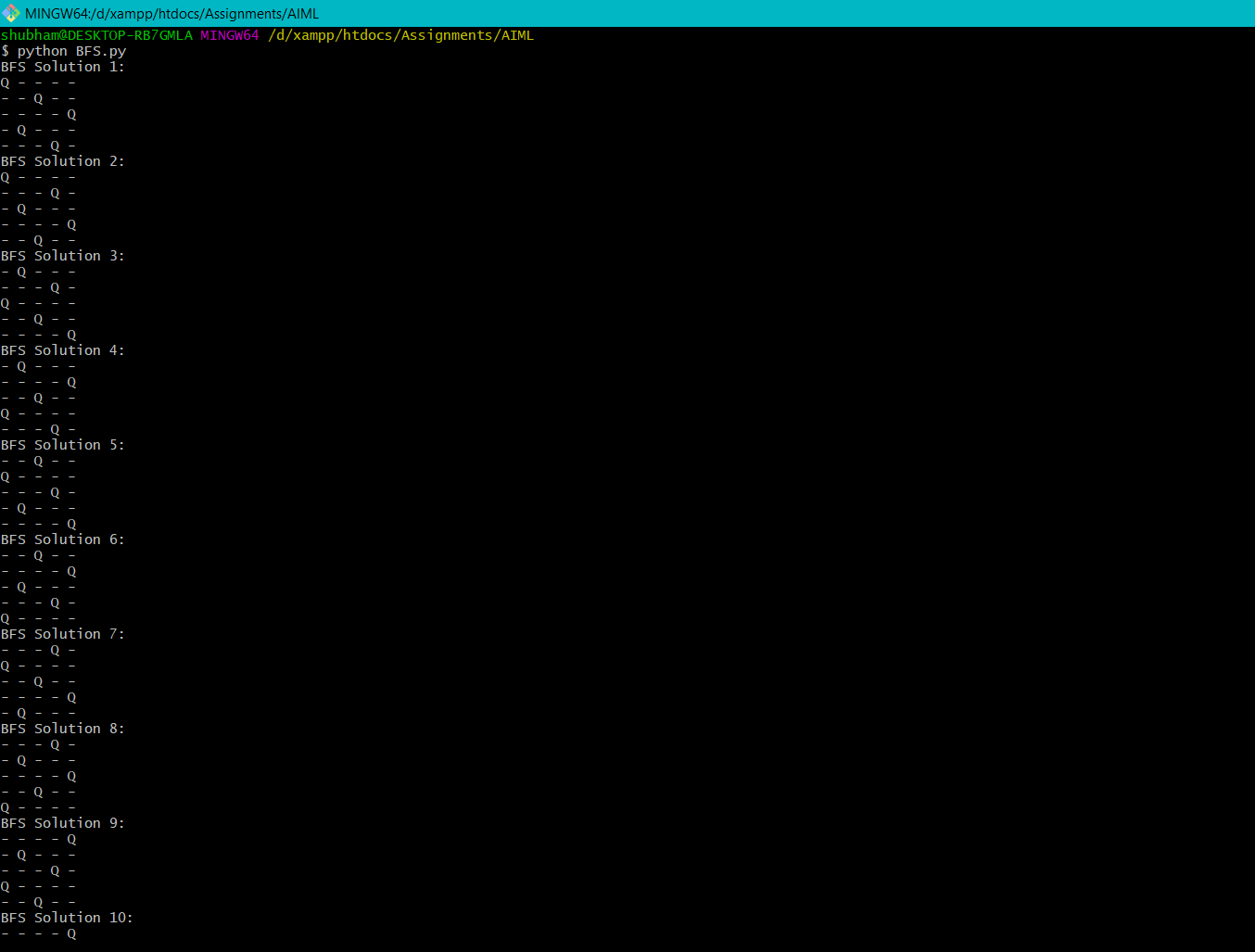
        print('BFS Solution %d:' % (i + 1))

        n\_queens.print(solution)

if \_\_name\_\_ == '\_\_main\_\_':

    main()

**Output:-**

****

**3-> Which algorithm is best suited for implementing N queens’ problem and why ?**

**Comparison Between both the approach**

* The DFS algorithm is quicker than BFS algorithm
* Number of extended node in DFS algorithm is less than BFS
* the required memory for BFS is larger than DFS. Using this method, time and cost of solving n-queens problem is minimized in comparison of old methods.