Penerapan Hill Climbing, Genetic Algorithm dan CSP Pada *N-Queen Puzzle*

Muh.Ikhsan[1], Akmal Zuhdy Prasetya[2], Eurico Devon Bura Pakilaran[3], dan Eka Fitri Ramadani[4]
Program Studi Sistem Informasi
Fakultas Matematika dan Ilmu Pengetahuan Alam
Univeristas Hasanuddin, Jl. Tamalanrea Indah Makassar, Indonesia

***Abstract* – N-Queen Puzzle merupakan suatu permasalahan dimana penempatan N buah ratu pada papan catur berukuran $N \times N$. Algoritma Hill Climbing, Genetic Algorithm, dan Constraint Satisfaction Problem dapat digunakan untuk menyelesaikan permasalahan ini. Oleh karena itu pada paper ini akan dibahas perbandingan penerapan algoritma-algoritma tersebut dalam menyelesaikan permasalahan N-Queen Puzzle.**

Kata Kunci* – *N-Queen, Puzzle, Algoritma, Hill Climbing, Genetic Algorithm, Constraint Satisfaction Problem;

I. PENDAHULUAN

N-Queen Puzzle merupakan salah satu bentuk permainan *puzzle* yang pertama kali diperkenalkan pada tahun 1848 oleh seorang pemain catur Max Bezzel. Dari tahun ke tahun, banyak matematikawan termasuk Gauss dan George Cantor telah bekerja keras untuk dapat menyelesaikan masalah N-Queen Puzzle ini. Solusi pertama kali dibentuk oleh Franz Nauck pada tahun 1850. Nauck juga memperluas *puzzle* ke bentuk N-Queen. Pada tahun 1874, S. Gunter menggunakan metode determinan dan J.W.L. Glaisher menyaring pendekatan tersebut.

Cara kerja dari N-Queen Puzzle ini adalah bagaimana kita menempatkan n buah ratu pada papan catur $N \times N$, dimana setiap ratu tersebut tidak dapat saling memakan satu sama lain, serta tidak ada 2 ratu yang terletak dalam satu baris, satu kolom, maupun satu diagonal.

II. LANDASAN TEORI

A. Hill Climbing Algorithm

Hill Climbing adalah teknik pengoptimalan matematis yang termasuk dalam keluarga *local search*. Algoritma ini merupakan algoritma berulang yang dimulai dengan solusi yang tidak konsisten untuk suatu masalah, kemudian mencoba menemukan solusi yang lebih baik dengan membuat perubahan bertahap pada solusi tersebut. Jika perubahan menghasilkan solusi yang lebih baik, perubahan tambahan dilakukan pada solusi baru dan seterusnya hingga tidak ada perbaikan lebih lanjut yang dapat ditemukan.

Hill Climbing Algorithm Pseudocode

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
     neighbor ← a highest-valued successor of current
     if neighbor.VALUE ≤ current.VALUE then return current.STATE
     current ← neighbor
```

Beberapa hal yang perlu diperhatikan untuk Hill Climbing dalam N-Queen puzzle:

1. Letakkan n queen di papan n × n tanpa dua queen di baris, kolom, atau diagonal yang sama
2. Pindahkan queen untuk mengurangi jumlah konflik
3. Penerus suatu keadaan adalah semua keadaan yang mungkin dihasilkan dengan memindahkan satu queen ke kotak lain dalam kolom yang sama (jadi setiap keadaan memiliki penerus $n \times (n - 1)$)
4. Fungsi biaya heuristik h adalah banyaknya pasang queen yang saling serang, baik secara langsung maupun tidak langsung
5. Minimum global dari fungsi ini adalah nol, yang hanya terjadi pada solusi sempurna

- Disini digunakan **Random Restart** karena keluar dari *shoulder (Merupakan wilayah dataran tinggi yang memiliki tepi menanjak)* dan memiliki peluang tinggi untuk keluar dari optimal local.

B. Genetic Algorithm (GA)

Genetic Algorithm adalah algoritma metaheuristic yang terinspirasi oleh proses seleksi alam yang termasuk dalam kelas *Evolutionary Algorithm* (EA). Algoritma genetic biasanya digunakan untuk menghasilkan solusi berkualitas tinggi untuk masalah pengoptimalan dan pencarian dengan mengandalkan operator yang terinspirasi secara biologis seperti mutase, persilangan, dan seleksi.

Genetic Algorithm Pseudocode

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
     new_population ← empty set
     for i = 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
     population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x); c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Beberapa hal yang perlu diperhatikan untuk Genetic Algorithm dalam N-Queen puzzle:

1. Algoritma genetika (GA) adalah varian dari pencarian berkas stokastik
2. Status penerus dihasilkan dengan menggabungkan dua orang tua, bukan dengan memodifikasi satu status
3. Prosesnya terinspirasi oleh seleksi alam
4. Dimulai dengan k status yang dihasilkan secara acak, disebut populasi. Setiap negara bagian adalah individu
5. Seorang individu biasanya diwakili oleh string 0 dan 1, atau digit, atau himpunan terbatas.Fungsi obyektif disebut fungsi kebugaran: keadaan yang lebih baik memiliki nilai fungsi tness yang tinggi
6. Pasangan individu dipilih secara acak untuk reproduksi dengan beberapa kemungkinan
7. Titik perpotongan dipilih secara acak dalam string
8. Keturunan diciptakan dengan menyilangkan orang tua di titik persilangan
9. Setiap elemen dalam string juga mengalami beberapa mutasi dengan probabilitas kecil
10. Dalam soal 8 ratu, seorang individu dapat diwakili oleh string digit 1 sampai 8, yang mewakili posisi 8 ratu dalam 8 kolom
11. Fungsi fitness yang mungkin adalah jumlah *pasangan non-menyerang* dari *ratu* yang kami tertarik untuk memaksimalkan (yang memiliki nilai maksimum \ (8 \ memilih 2 \) = 28 untuk masalah 8-ratu. Dengan kata lain kita ingin jumlah *menyerang pasangan* dari *ratu* menjadi nol dalam penugasan solusi dari ratu

C. Constraint Satisfaction Problem (CSP)

Constraint Satisfaction Problem adalah suatu pertanyaan matematika yang didefinisikan sebagai sekumpulan objek yang statusnya harus memenuhi sejumlah batasan atau limitasi. CSP merepresentasikan entitas dalam masalah sebagai kumpulan homogen dari batasan hingga variabel, yang diselesaikan dengan metode kepuasan batasan. CSP adalah subjek penelitian intensif baik dalam penelitian kecerdasan buatan maupun operasi, karena keteraturan dalam perumusannya memberikan dasar umm untuk menganalisis dan memecahkan masalah dari banyak famili yang tampaknya tidak terkait. CSP sering kali menujukkan kompleksitas tinggi, yang membutuhkan kombinasi metode heuristik dan pencarian kombinatorial untuk diselesaikan dalam waktu yang wajar.

CSP's Forward CheckingPseudocode

```
Procedure ForwardChecking(i)
 supportCount = conflictCount = 0; ratioArray[]
begin
 for each a ∈ Domain(i) do
   ratio = 0
   for each j such that (i, j) is a constraint do
     for each b ∈ Domain(j) do
       if (a, b) satisfies the constraint (i, j) then
         supportCount=supportCount+1
       else
       delete b from Domain(j)
       conflictCount=conflictCount+1
       endif
     endfor
   if Domain(j) = ∅ then delete a from Domain(i)
   ratio = ratio + conflictCount/supportCount
   endfor
   ratioArray[a] = ratio
 endfor
 return ratioArray
end
```

CSP's Backtracking Search Pseudocode

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
     if value is consistent with assignment then
        add {var = value} to assignment
        inferences ← INFERENCE(csp, var, value)
        if inferences ≠ failure then
           add inferences to assignment
           result ← BACKTRACK(assignment, csp)
           if result ≠ failure then
              return result
     remove {var = value} and inferences from assignment
  return failure
```

Beberapa hal yang perlu diperhatikan untuk CSP-FC dalam N-Queen puzzle:

1. Queen pertama akan ditempatkan di papan kosong
2. Kemudian, ini akan memanggil fungsi modular yang diimplementasikan untuk membuat konsistensi FC ke variabel lain
3. Apabila, Queen tidak bisa ditempatkandi setiap sel di kolom yang tersedia , dan dengan demikian buffering perlu mundur. Jadi, memanggil fungsi yang diimplementasikan bertanggung jawab untuk menghapus batasan yang dibuat oleh Queen yang mengalami konflik dan kemudian menetapkannya kembali ke posisi baru yang valid

III. PENERAPAN ALGORITMA HILL CLIMBING, GENETIC ALGORITHM, DAN CONSTRAINT SATISFACTION PROBLEM DALAM N-QUEEN PUZZLE

HILL CLIMBING

```
  class NQueen:
      def __init__(self, row, column):
          self.row = row
          self.column = column
      def get_row(self):
          return self.row
      def get_column(self):
          return self.column
      def move(self):
          self.row += 1

def is_conflict(self, queen):
  # check rows and columns
    if self.row == queen.get_row() or self.column ==
    queen.get_column():
      return True
  # check diagonals
    elif abs(self.column - queen.get_column()) ==
    abs(self.row - queen.get_row()):
      return True
    return False

class HillClimbingRandomRestart:
  def __init__(self, n):
    self.n = n
    self.status = False
    self.steps_climbed_after_last_restart = 0
    self.steps_climbed = 0
    self.heuristic = 0
    self.random_restarts = 0

  # method to create a new random board
  def generate_board(self):
    start_board = []
    for i in range(self.n):
      start_board.append(NQueen(random.randint(0,
    self.n - 1), i))
    return start_board

  # method to find heuristics of a state
  def find_heuristic(self, state):
    heuristic = 0
    for i in range(len(state)):
      for j in range(i + 1, len(state)):
        if state[i].is_conflict(state[j]):
          heuristic += 1
     return heuristic

# method to get the next board with lower heuristic
  def next_board(self, present_board):
    next_board = []
    tmp_board = []

    present_heuristic =
    self.find_heuristic(present_board)
    best_heuristic = present_heuristic
  temp_h = 0
```

```
    for i in range(self.n):
    # copy present board as best board and temp board
      next_board.append(
      NQueen(present_board[i].get_row(),
      present_board[i].get_column()))
      tmp_board.append(next_board[i])

    # iterate each column
    for i in range(self.n):
      if i > 0:
        tmp_board[i - 1] = NQueen(
        present_board[i - 1].get_row(),
    present_board[i -
    1].get_column()
                  )
      tmp_board[i] = NQueen(0,
    tmp_board[i].get_column())

    # iterate each row
    for j in range(self.n):
      # get the heuristic
      temp_h = self.find_heuristic(tmp_board)
      # check if temp board better than best board
      if temp_h < best_heuristic:
        best_heuristic = temp_h
        # copy the temp board as best board
        for k in range(self.n):
        next_board[k] = NQueen(
        tmp_board[k].get_row(),
        tmp_board[k].get_column())

        # move the queen
        if tmp_board[i].get_row() != self.n - 1:
          tmp_board[i].move()

    # check whether the present bord and the best
    board found have same heuristic
    # then randomly generate new board and assign
it to best board
    if best_heuristic == present_heuristic:
      next_board = self.generate_board()
      self.random_restarts += 1
      self.steps_climbed_after_last_restart = 0
      self.heuristic =
      self.find_heuristic(next_board)
    else:
      self.heuristic = best_heuristic
      self.steps_climbed += 1
      self.steps_climbed_after_last_restart += 1

    return next_board

  # method to print the current state
  def get_state(self, state):
    # creating temporary board from the present
    board
    # temp_board = np.zeros([self.n, self.n],
    dtype=int)
    temp_board = [[] for i in range(self.n)]
    for i in range(self.n):
      for j in range(self.n):
        temp_board[i].append(0)
    # temp_board = [[0, 0, 0, 0],
                    [0, 0, 0, 0],
                    [0, 0, 0, 0],
                    [0, 0, 0, 0]]
    # temp_board = temp_board.tolist()
```

```
    for i in range(self.n):
    # get the positions of queen from the present board
     # and set those positions as 1 in solution board

      temp_board[state[i].get_row()][state[i].get_
      column()] = 1
     return temp_board

  def solve(self):
    if self.n == 2 or self.n == 3:
      print(f"No Solution possible for {self.n}
queens.")
      return

    # initialize present heuristic
    present_heuristic = 0
    # creating the initial board
    present_board = self.generate_board()

    # test if the present board is the solution
    board
    while present_heuristic != 0:
      # get the next board ->
      printState(presentBoard)
      present_board =
      self.next_board(present_board)
      present_heuristic = self.heuristic
    # return state from present board
    self.status = True
    return self.get_state(present_board)

def print_solution_and_status(self):
    print(f"Solving {self.n} queen problem with
    random restart hill climbing")
    # initialize time and memory usage
    start = time.time()
    process = psutil.Process(os.getpid())
    # get the solution
    solution = self.solve()
    # print the solution
    print()
    # print(np.matrix(solution))
    # print(solution)
    for i in range(self.n):
      for j in range(self.n):
        print(solution[i][j], end=" ")
      print()
    # print complexity
    print("\nStatus\t :", "Complete" if
    self.status else "Uncompleted")
    print(f"Memori\t : {process.memory_info().rss
    / 1024 ** 2} MB")
    print(f"Time\t : {time.time() - start}
    seconds")
    print(f"Total number of steps climbed\t :
    {self.steps_climbed}")
    print(f"Number of random restarts\t :
    {self.random_restarts}")
    print(f"Steps climbed after last restart :
    {self.steps_climbed_after_last_restart}")
    # return solution
    return solution

# main
if __name__ == "__main__":
  n_queen_hc = HillClimbingRandomRestart(
  int(input("Masukkan jumlah queen : ")))
  solution =
n_queen_hc.print_solution_and_status()
  # plot(solution)
```

Genetic Algorithm

```
class GeneticAlgorithm:
  def __init__(self, n=int, max_fitness=float,
  population=list, generation=int):
  self.n = n
  self.max_fitness = max_fitness
  self.population = population
  self.generation = generation
  self.status = False

  # making random chromosomes
  @classmethod
  def random_chromosome(cls, size):
    return [random.randint(1, size) for _ in
    range(size)]

class GeneticAlgorithm:
  def __init__(self, n=int, max_fitness=float,
  population=list, generation=int):
  self.n = n
  self.max_fitness = max_fitness
  self.population = population
  self.generation = generation
  self.status = False

  # making random chromosomes
  @classmethod
  def random_chromosome(cls, size):
    return [random.randint(1, size) for _ in
    range(size)]

  def fitness(self, chromosome):
    horizontal_collisions = (
    sum([chromosome.count(queen) - 1 for queen in
    chromosome]) / 2)
    n = len(chromosome)
    left_diagonal = [0] * 2 * n
    right_diagonal = [0] * 2 * n
    for i in range(n):
      left_diagonal[i + chromosome[i] - 1] += 1
      right_diagonal[len(chromosome) - i +
    chromosome[i] - 2] += 1

    diagonal_collisions = 0
    for i in range(2 * n - 1):
      counter = 0
      if left_diagonal[i] > 1:
        counter += left_diagonal[i] - 1
      if right_diagonal[i] > 1:
        counter += right_diagonal[i] - 1

    diagonal_collisions += counter / (n - abs(i - n
    + 1))

    # 28 - (2 + 3) = 23
    return int(self.max_fitness -
(horizontal_collisions + diagonal_collisions))

def probability(self, chromosome, fitness):
    return fitness(chromosome) / self.max_fitness

  def random_pick(self, population, probabilities):
    population_with_probability = zip(population,
    probabilities)
    total = sum(w for c, w in
    population_with_probability)
    r = random.uniform(0, total)
    upto = 0
```

```
    for c, w in zip(population, probabilities):
      if upto + w >= r:
        return c
      upto += w

    assert False, "Shouldn't get here"

  # doing cross_over between two chromosomes
    def reproduce(self, x, y):
      n = len(x)
      c = random.randint(0, n - 1)
      return x[0:c] + y[c:n]

  # randomly changing the value of a random index of
  a chromosome
  def mutate(self, x):
  n = len(x)
  c = random.randint(0, n - 1)
  m = random.randint(1, n)
  x[c] = m
  return x

  def genetic_queen(self, population, fitness):
    new_population = []
    mutation_probability = 0.03
    probabilities  =  [self.probability(i,  fitness)
    for i in population]

    for i in range(len(population)):
      # best chromosome 1
      x = self.random_pick(population,
    probabilities)
      # best chromosome 2
      y = self.random_pick(population,
    probabilities)
      # creating two new chromosomes from the best 2
    chromosomes
      child = self.reproduce(x, y)

      if random.random() < mutation_probability:
        child = self.mutate(child)
        self.print_chromosome(child)
        new_population.append(child)

      if fitness(child) == self.max_fitness:
        break

    return new_population

  def print_chromosome(self, chromosome):
    print(f"Chromosome = {str(chromosome)},
    fitness = {self.fitness(chromosome)}")

  def solve(self):
  if self.n == 2 or self.n == 3:
    print(f"No  solution  possible  for  {self.n}
    queens.")
    return

    while not self.max_fitness in [
    self.fitness(chromosome) for chromosome in
    self.population]:
        print(f"------------- Generation
        {self.generation} -------------")
    self.population =
        self.genetic_queen(self.population,
        self.fitness)
```

```
    print("")
    print(f"Max. fitness = {max([self.fitness(i) for
    i in self.population])}")
    print("")
    self.generation += 1
    chrom_out = []
    print(f"Solved in generation {self.generation -
    1}")
    for chrom in self.population:
      if self.fitness(chrom) == self.max_fitness:
        print("")
        print("One of the solutions: ")
        chrom_out = chrom
        self.print_chromosome(chrom)

    board = [[] for i in range(n)]
          for i in range(self.n):
            for j in range(self.n):
              board[i].append(0)

        for i in range(self.n):
          board[self.n - chrom_out[i]][i] = 1

        # return board
        self.status = True
        return board

  def print_solution_and_status(self):
    print(f"Solving  {self.n}  queen  problem  with
    genetic algorithm\n")
    # initialize time and memory usage
    start = time.time()
    process = psutil.Process(os.getpid())
    # get the solution
    solution = self.solve()
    # print the solution
    print()
    for i in range(self.n):
      for j in range(self.n):
        print(solution[i][j], end=" ")
      print()
    # print complexity
    print("\nStatus\t :", "Complete" if self.status
    else "Uncompleted")
    print(f"Memori\t : {process.memory_info().rss /
    1024 ** 2} MB")
    print(f"Time\t : {time.time() - start}
    seconds")
    # return solution
        return solution

# main
if __name__ == "__main__":
  n = int(input("Masukkan jumlah queen : "))
  max_fitness = (n * (n - 1)) / 2
  population =
    [GeneticAlgorithm.random_chromosome(n) for _ in
    range(100)]
  generation = 1

  n_queen_ga  =  GeneticAlgorithm(n,  max_fitness,
  population, generation)
  solution = n_queen_ga.print_solution_and_status()

  # plot(solution)
```

Pseudocode Constraint Satisfaction Problem

```
class Unassigned:
  def __init__(self, row, column):
    self.row = row
    self.column = column

  def __eq__(self, other):
  return self.row == other.row and self.column ==
  other.column

  def __hash__(self):
    return hash(self.row) ^ hash(self.column)

class CSPForwardChecking:
  def __init__(self, n):
    self.n = n
    self.status = False

  def get_unassigned_from_constraint(self, board,
  queen):
  result = []
    for row in range(self.n):
      for col in range(queen + 1, self.n):
        if board[row][col] == 0 and
        self.is_correct(board, row, col):
          result.append(Unassigned(row, col))
    return result

  def forward_check(self, board, row, queen):
    act_domain = self.get_rows_proposition(board,
    queen)
    tmp_domain = list(act_domain)
    for proposition_row in act_domain:
      if not self.is_correct(board,
      proposition_row, queen):
        tmp_domain.remove(proposition_row)
    return len(tmp_domain) == 0

  def is_correct(self, board, row, column):
    return (self.is_row_correct(board, row)
      and self.is_column_correct(board, column)
      and self.is_diagonal_correct(board, row,
column))

  def is_row_correct(self, board, row):
    for col in range(self.n):
      if board[row][col] == 1:
        return False
    return True

  def is_column_correct(self, board, column):
    for row in range(self.n):
      if board[row][column] == 1:
        return False
    return True

  def check_upper_diagonal(self, board, row,
column):
    iter_row = row
    iter_col = column
    while iter_col >= 0 and iter_row >= 0:
      if board[iter_row][iter_col] == 1:
        return False
      iter_col -= 1
      iter_row -= 1

    return True
```

```
    def check_lower_diagonal(self, board, row,
column):
      iter_row = row
      iter_col = column

      while iter_col >= 0 and iter_row < self.n:
        if board[iter_row][iter_col] == 1:
          return False

        iter_row += 1
        iter_col -= 1

        return True

    def is_diagonal_correct(self, board, row,
column):
    return self.check_upper_diagonal(board, row,
column) and self.check_lower_diagonal(board, row,
column)

  def get_rows_proposition(self, board, queen):
    rows = []

    for row in range(self.n):
      if self.is_correct(board, row, queen):
        rows.append(row)
    return rows

  def solve(self, board, queen):
    if self.n == queen:
      self.status = True
      return True

    if self.n == 2 or self.n == 3:
      print(f"No Solution possible for {self.n}
queens.")
      return

    rows_proposition =
self.get_rows_proposition(board, queen)

    for row in rows_proposition:
      board[row][queen] = 1
      domain_wipe_out = False

    for variable in
self.get_unassigned_from_constraint(board, queen):
      if self.forward_check(board, variable.row,
variable.column):
        domain_wipe_out = True
          break

    if not domain_wipe_out:
      if self.solve(board, queen + 1):
        return True

    board[row][queen] = 0

  def print_solution_and_status(self, board):
    print(f"Solving {self.n} queen problem with CSP
forward checking")
  # initialize time and memory usage
  start = time.time()
  process = psutil.Process(os.getpid())
  # get the solution
  solution = self.solve(board, 0)
  # print the solution
  # print(board)
```

```
  print()
  for i in range(self.n):
    for j in range(self.n):
      print(board[i][j], end=" ")
    print()
  # print complexity
  print("\nStatus\t :", "Complete" if self.status
else "Uncompleted")
  print(f"Memori\t : {process.memory_info().rss /
1024 ** 2} MB")
  print(f"Time\t : {time.time() - start} seconds")

  # return board
  return board

# main
if __name__ == "__main__":
  n = int(input("Masukkan jumlah queen : "))
  board = [[] for i in range(n)]
  for i in range(n):
    for j in range(n):
      board[i].append(0)

  n_queen_csp_fc = CSPForwardChecking(n)
  solution =
n_queen_csp_fc.print_solution_and_status(board)

  # plot(solution)
```

DFS Backtracking in Python

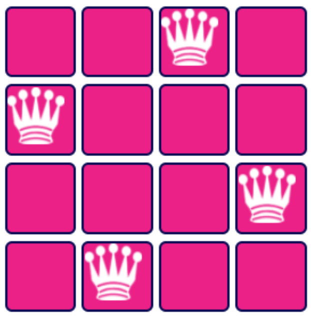
```
class BacktrackingDFS:
  def __init__(self, n):
    self.n = n
    self.status = False
    # is it possible to place a queen into (y,x)?
    def possible(self, board, y, x):
      # check for queens on row y
      for i in range(self.n):
        # if exist return false
          if board[y][i] == 1:
            return False

      # check for queens on column x
      for i in range(self.n):
        # if exists return false
        if board[i][x] == 1:
          return False

      # loop through all rows
      for i in range(self.n):
        # and columns
        for j in range(self.n):
          # if there is a queen
          if board[i][j] == 1:
            # and if there is another on a diagonal
            if abs(i - y) == abs(j - x):
              # return false
              return False

      # if every check clears, we can return true
      return True

  def solve(self, board):
    if self.n == 2 or self.n == 3:
      print(f"No Solution possible for {self.n}
queens.")
      return

    # for every row
    for y in range(self.n):
      # for every column
      for x in range(self.n):
        # we can place if there is no queen in
given position
        if board[y][x] == 0:
          # if empty, check if we can place a queen
          if self.possible(board, y, x):
            # if we can, then place it
            board[y][x] = 1
            # pass board for recursive solution
            self.solve(board)
            # if we end up here, means we searched
through all children branches
            # if there are 8 queens
            if sum(sum(a) for a in board) ==
self.n:
              # we are successful so return
              self.status = True
              return board

              # remove the previous placed queen
              board[y][x] = 0

    # means we searched the space, we can return
our result
    return board

  def print_solution_and_status(self, board):
    print(f"Solving {self.n} queen problem with
backtracking")
    # initialize time and memory usage
    start = time.time()
    process = psutil.Process(os.getpid())
    # get the solution
    solution = self.solve(board.copy())
    # print the solution
    print()
    for i in range(self.n):
      for j in range(self.n):
        print(solution[i][j], end=" ")
      print()
    # print complexity
    print("\nStatus\t :", "Complete" if self.status
else "Uncompleted")
    print(f"Memori\t : {process.memory_info().rss /
1024 ** 2} MB")
    print(f"Time\t : {time.time() - start}
seconds")

    # return solution
    return solution

# main
if __name__ == "__main__":
  n = int(input("Masukkan jumlah queen : "))
  board = [[] for i in range(n)]
  for i in range(n):
    for j in range(n):
      board[i].append(0)

  n_queen_backtracking = BacktrackingDFS(n)
  solution =
n_queen_backtracking.print_solution_and_status(boar
d)
  # plot(solution)
```

IV. Pengujian Algoritma

Seteleh membuat program *Hill Climbing Random Restart*, *Genetic Algorithm*, dan *CSP Forward Checking* untuk kasus N-Queen Problem maka selanjutnya akan dilakukan benchmark terhadap hasil yang diperoleh tiap algoritma.

Hasil Benchmark

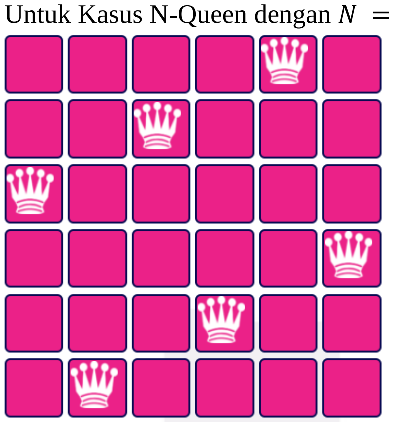
- Untuk Kasus N-Queen dengan $N = 4$



Hasil Benchmark:

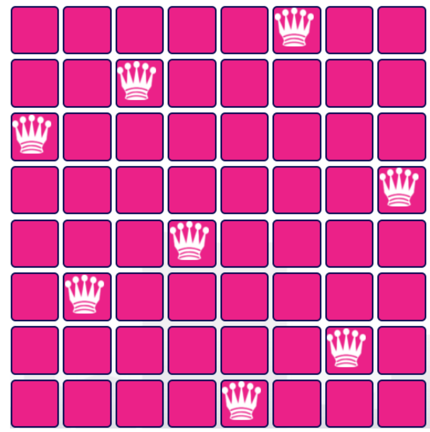
Algoritma	Waktu (s)	Memori (MB)	Complete
Hill Climbing	0,00299	13,4492	TRUE
Genetic Algorithm	0,12500	13,4844	TRUE
CSP-Forward Checking	0,00608	13,4609	TRUE

- Untuk Kasus N-Queen dengan $N = 6$



Algoritma	Waktu (s)	Memori (MB)	Complete
Hill Climbing	0,01095	13,4414	TRUE
Genetic Algorithm	2,37712	13,4414	TRUE
CSP-Forward Checking	0,02032	13,4453	TRUE

- Untuk Kasus N-Queen dengan $N = 8$



Algoritma	Waktu (s)	Memori (MB)	Complete
Hill Climbing	0,01926	13,4492	TRUE
Genetic Algorithm	2,02425	13,4570	TRUE
CSP-Forward Checking	0,09644	13,4688	TRUE

V. Kesimpulan

Meskipun telah ada algoritma seperti backtracking dalam menyelesaikan N-Queen tetapi disini kita gunakan pendekatan AI. Dalam menyelesaikan N-Queen Problem menunjukkan bahwa algoritma Hill Climbing memberikan solusi yang efisien dengan waktu yang paling singkat dibandingkan *genetic algorithm* dan *CSP*, meskipun tidak selalu menjamin solusi yang benar secara global.

Referensi

[1] Russel, Stuart J. Peter Norvig (Ed). 2009. *Artificial Intelligence: A Modern Approach* Third Edition. Prentice Hall.

[2] https://www.slideshare.net/eganghukz/makalah-nqueen-problem

[3] https://en.wikipedia.org/wiki/Hill_climbing

[4] https://en.wikipedia.org/wiki/Genetic_algorithm

[5] https://en.wikipedia.org/wiki/Constraint_satisfaction_problem