ARTIFICIAL INTELLIGENCE

PROGRAMMING ASSIGNMENT – 1 REPORT

- By:
  Name: Ashima Garg
  Roll No: PhD19003

AI Programming Assignment-1

Folder Description:

- **Solution 1**
  - main.py : driver functions
  - BFS.py : Code for BFS Algorithm
  - DFS.py : Code for DFS Algorithm
  - DFS.ipynb : Jupyter Python Notebook contains the DFS code tested on google colab.
  - Astar.py : Code for A Star Algorithm
  - AstarNode.py : Node definition for A Star and IDA Star
  - min_heap.py : Contains class for min heap
  - IDA_Star.py : Code for IDA Star Algorithm
  - Utils.py : Utility functions
  - AI_Prog_Assignment_1.ipynb : Python Notebook contains the driver functions of main.py and the final **Analysis Graph.**
- **Solution 2**
  - EightQueens : Code for 8-Queens Problem
  - NQueens : Code for N-queens Problem

Solution 1:

- ### Assumptions

1. To test the code, maximum 3 optimal moves are made in the initial start node to reach the goal state for each value of N with limitations due to CPU and time taken by each algorithm for large values of N.
2. In the Analysis Graph plotted against time and different values of N, DFS is not included with limitations to CPU and time taken is very large. A separate python notebook tested on google colab is included for DFS to calculate the time taken for N = 8 problem.
3. In A star Algorithm, two different heuristic values are calculated and the optimum heuristic value is used out of the two by taking max value of the H1 and H2. Heuristic function 1 calculates “Total Number of misplaced tiles to goal positions in board n” and Heuristic function 2 calculates “sum of distances of misplaced tiles to goal positions in

board n”. Where as in IDA*, the heuristic value is calculated using “Total Number of misplaced tiles to goal positions in board n”.

- **Algorithm**

➢ BFS

```
def bfs(start, goal):
    declare queue
    add start node to queue
    while queue not empty:
            node <- pop(queue.front)
            if node == goal:
                    return “Solved”
            add node to exploredSet
            for each successor of node :
                    if successor is not in exploredSet then:
                            add successor in queue
```

➢ DFS

```
def dfs(start, goal):
    exploredSet = []
    declare stack
    add start node to stack
    while stack not empty:
            node <- pop(stack)
            if node == goal:
                    return “Solved”
    add node to exploredSet
    for each successor of node:
            if successor is not in exploredSet then:
                    add successor in stack
```

➢ A*

```
def calculate heuristic 1(node):
    # return “Total Number of misplaced tiles to goal positions in
    board n”
```

```
def calculate heuristic 2 (node):
    # return "sum of distances of misplaced tiles to goal positions in
    board n"
 def A-Star(start, goal):
    exploredSet = []
    declare min priority queue
    calculate start.f = max(heuristic 1(start), heuristic 2(start))+ g
    add start node to priority_queue
    while priority_queue not empty:
        node <- pop(priority_queue.front)
        if node == goal:
            return "Solved"
    add node to exploredSet
    for each successor of node:
        if successor is not in exploredSet then:
            calculate node.f = max(heuristic 1(successor),
            heuristic 2(successor))+ g
            add successor in priority_queue
```

➢ **IDA***

```
    def calculate heuristic 1(node):
        # return "Total Number of misplaced tiles to goal positions
        in board n"
    def IDA-Star(start, goal):
      for each f in range(2, 24) increment by 2 each iteration if not
    found
            exploredSet = []
        declare min priority queue
        calculate start.f = heuristic 1(start)+ g
        add start node to priority_queue
        while priority_queue not empty:
            node <- pop(priority_queue.front)
            if node == goal:
                return "Solved"
        add node to exploredSet
        for each successor of node:
            if successor is not in exploredSet then:
                calculate node.f = heuristic 1(successor)+ g
                add successor in priority_queue
```

- **Methodology**

- **BFS**

  Declare a function bfs and pass *start* and *goal* node to it. Create an empty queue and exploredSet to track visited nodes using python list. Append the start node to the queue. Check for each front node in the queue if it is equal to the *goal* node, then return and print *"Solved".* Else find all the successors of the current node using *dx* and *dy* direction matrices and enqueue the successors in the queue (exclude the successors which are already included in the queue).

- **DFS**

  Declare a function dfs and pass start and goal node to it. Create an empty stack and explored Set to track visited nodes using python list. Append the start node to the stack. Check for each popped node from the stack if it is equal to the goal node, then return and print *"Solved".* Else find all the successors of the current node using *dx* and *dy* direction matrices and push the successors in the stack (exclude the successors which are already included in the queue).

- **A-Star**

  Create a class for *A-star Node* that stores the state information including, f = estimated value, h = heuristic value, g = cost incurred for this state. Create a class for *min priority queue* (based on minimum value of f = g + h) to store the A-Star type nodes to be used in the algorithm. Declare a function A-Star and pass *start* and *goal* node to it. Declare a priority_queue object of the above created priority class and explored Set (python list) to track visited nodes. Check for each popped node from the priority_queue if it is equal to the goal node, then return and print "*Solved".* Else find all the successors of the current node using *dx* and *dy* direction matrices and enqueue the successors in the priority_queue (exclude the successors which are already included in the queue).

- **IDA-Star**

  Create a class for *A-star Node* that stores the state information including, f = estimated value, h = heuristic value, g = cost incurred for this state. Create a class for *min priority queue* (based on minimum value of f = g + h) to store the A-Star type nodes to be used in the algorithm. Declare a function IDA-Star and pass *start* and *goal* node to it. Initialize f_{max} *(f = g+h)* as 2 in the first iteration and apply the A-Star algorithm. Declare a priority_queue object of the above created priority class and explored Set (python list) to track visited nodes. Expand the nodes with *f* values less than f_{max}. Check for each popped node from the priority_queue if it is equal to the goal node, then return and print *"Solved".* Else find all the successors of the current node using *dx* and *dy* direction matrices and enqueue the successors in the priority_queue (exclude the successors which are already included in the queue). Increment f_{max} in steps of two in each iteration.

- **Observations**

  Maximum 3 Optimal Moves are used to test the algorithms.
  For N = 8, following is the start and goal node used to test the function:

1	2	3
5	6	
7	8	4

N = 8, START POSITION

1	2	3
5	8	6
	7	4

N = 8, GOAL POSITION

For N = 16, following is the start and goal node used to test the function:

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

N = 15, START POSITION

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

N = 15, GOAL POSITION

For N = 24, following is the start and goal node used to test the function:

1	2	3	4	5
6	7	8	9	10
11	12	13		15
16	17	18	14	19
21	22	23	24	20

N = 24, START POSITION

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	

N = 24, GOAL POSITION

- **Results**

BFS execution time:

N = 8, time = 0.0020606517791748047 seconds

N = 15, time = 0.0030090808868408203 seconds

N = 24, time = 0.0033240318298339844 seconds

DFS execution time:

N = 8, time = 1.6689300537109375e-06 seconds

A-Star execution time:

N = 8, time = 0.0018434524536132812 seconds

N = 15, time = 0.0025703907012939453 seconds.

N = 24, time = 0.0033228397369384766 seconds

IDA-Star execution time:

N = 8, time = 0.002432107925415039 seconds

N = 15, time = 0.0027375221252441406 seconds

N = 24, time = 0.003545999526977539 seconds

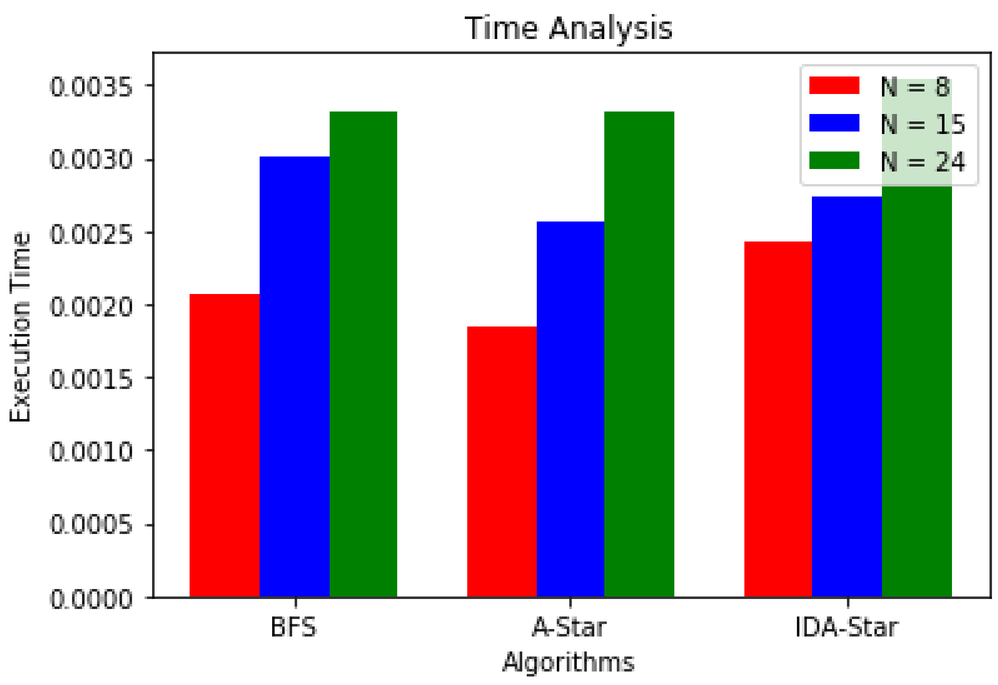

Fig1: Bar graph representing time analysis of Search algorithms for different values of N.

- **Inference**

➢ As the value of N increases or the size of the problem, i.e. from N = 8 to N = 24, time taken by each algorithm increases.

➢ Time taken by A-Star is the least for each value of N. Time taken by DFS is exponentially large as compared to other algorithms.

➢ Execution time of IDA* is more than A* in this case because of the fact it starts with f_{max} value in the initial iteration and if it is not able to expand till the goal node, it starts all over again from beginning with increased f_{max} .

Solution 2:

- **Assumptions**
  1. Complete State formulation as described by *Stuart Russell and Peter Norvig* is used to initialize the initial population according to which N queens are already placed on the board in different columns. So, no vertical attacks in the initial state itself. This helps in convergence.
  2. Maximum no of iterations of the algorithm for 8-Queens is set to 10000 and 1,00,000 for N > 8 after which even if the final goal state is not found the algorithm stops.

- **Algorithm**
  - **N-Queen/8-Queens**

```
Total Nonattacking = N * (N -1)/2
Population_Size = 10
Class Chromosome:
        pos = []    #list of positions of queens to be placed on board
        board = []  #Actual board configuration with 'Q' representing
        Queens and '.' For empty positions.
        fitness = 0  #fitness value of each chromosome

def calculate_fitness():
    attacks = 0
    attacks += calculate_horizontal_attacks()
    attacks += calculate_vertical_attacks()
    attacks += calculate_diagnol_attacks()
    return (Total Nonattacking – attacks)

def reproduce_mutation(parent):
   num = randomly_sample(6)
   Swap 3 times at these num positions in parent array
  return  child

def reproduce_crossover():
    m = N /2
    child.pos = parent1[ :m] + parent2[ m:]
    #IMP STEP: Helped in convergence
     if child == parent1 or child == parent2:
          select 2 numbers randomly and swap them in pos array of
          chromosome.

    child.board = generate_board()

    child.fitness = calculate_fitness(child)

    return child


def geneticAlgo():
    create_initial_population()
    found = False
    iteration = 1
    while not found:
          population.sort(x.fitness, reverse = True)  #decreasing order of
fitness value.
          If population[0].fitness == Total Nonattacking:
              found = True
             print iteration no, best chromosome, and fitness value
             return
          keep 30% of best chromosomes from previous population
          generate 70% new population using this 30%
                  parent1, parent2 <- select 2 best parents from 5% best
                  chromosomes
                  if iteration == 50:
                          child = reproduce_mutate(parent2)
                  else:
                          child = reproduce_crossover(parent1, parent2)
                  add child to new generation
```

- **Methodology**

  Initialize value of *Total_NonAttacking_Pairs* = N*(N-1)/2 which is equal to total non-attacking pairs in the final or goal position. Initialize *Population_Size* = 10.
  Create a class of chromosome to store pos[] - list of positions of queens to be placed in each column of board, board[] – N*N matrix contains

Queens represented by ‘Q’ and empty positions as ‘.’, fitness value – corresponding to each chromosome.
Declare a geneticAlgo() function.
Create_Initial_Population() of size *Population_Size.* Initial population is created such that no queens attack atleast in the same column, i.e. all the N-Queens are placed in different columns as per the complete state formulation as described by *Stuart Russell and Peter Norvig.* Calculate fitness value of each chromosome in the initial population.
Sort the population in the reverse order of fitness value of chromosomes. If population[0].fitness = *Total_NonAttacking_Pairs* then print Iteration No., Best Chromosome, Fitness of Best Chromosome.
Keep 30% of best population and generate 70% new population using parent crossover as described in the above function of *reproduce_crossover()* and *reproduce_mutate()* in every 50th iteration.

- **Observations**

  Code is tested for values of N = 4, N = 8, N = 16.
- **Results**

  For N = 4, the algorithm converges in 1-10 iterations.
  For N = 8, the algorithm converges in 400-450 iterations.
  For N = 16, the algorithm converges after 30000 iterations.
- **Inference**
  - As the value of N increases, time required for convergence increases exponentially.

References:

1) [Book] Artificial Intelligence-Stuart Russell & PeterNorvig
2) https://python-graph-gallery.com/11-grouped-barplot/