AI FINAL PROJECTS DOCUMENTATION

Dr. Arafat Abu Mallouh

CMPT 420 / CMPG 720: Artificial Intelligence

Team Members: Wilber Cortez, Nathan Alex, Hieu Lam

PROJECT III: Solving 8-Queens Problem using Simulated Annealing

I. Introduction

Our project consisted of solving an 8-queens problem using Simulated Annealing. We used different input temperatures to showcase the program's ability to solve the problem. The higher the temperature the better for Simulated Annealing to solve the problem. For the 8-queen, we were able to solve it 100% of the time regardless of temperature.

II. Methods

We investigate solving the problem by implementing:

- Generating a blank chessboard
- Generate the initial state of the Queens.
- Implement attack conditions for the queens as well as checking for the number of queens attacking each other.
- Implement Simulated Annealing to find the solution.

Step 1: Generating a blank chessboard.

Included in the chessboard.py file, we generated a blank matrix to store the location of the queens.

```
def draw_chess_board(self):
    for row in range(8):
        for col in range(8):
```

Step 2: Generating the initial state of the Queens

Used in the queen.py file, the initial_state function was used to inject the location of the queens randomly onto the chessboards.

```
def initial_state(n):
    arr = []
    for i in range(n):
        arr.append(random.randint(0, n))
    return arr
```

Step 3: Implement goal states, attack conditions for the queens as well as checking mechanism for the number of queens attacking each other.

- Attack condition: Queens attacking horizontally and diagonally.

```
def queenAttack(x_1, y_1, x_2, y_2):
    if x_1 == x_2:
        return True
    elif y_1 == y_2:
        return True
    elif abs(x_1 - x_2) == abs(y_1 - y_2):
        return True
    return False
```

- Number of queens attacking one another

```
def numQueensAttack(arr):
    #finds the number of queens that are attacking each other
    h = 0
    for i in range(len(arr)): #uses the first queen in the order to compare with other queens
        qR = arr[i]
        qC = i+1
        for j in range(len(arr)):
            if i == j: continue
            oR, oC = arr[j], j+1
            if queenAttack(qR, qC, oR, oC):
                h = h + 1
    return h
```

Using the first queen to check if it is attacking others. If yes add to counter of the number of queens that first queen is attacking.

- Goal state: no queen is attacking any other queens.

```
def isGoalState(arr):
    if numQueensAttack(arr) == 0:
        return True
    return False
```

Step 4: Implement Simulated Annealing to find the solution.

Simulated-Annealing uses temperature to regulate the probability of bad moves allowed in the algorithm. The higher the initial temperature, the more likely the bad moves are allowed to occur. As the algorithm began executing, if the algorithm were presented with moves that improves the current state, the move is accepted, and the temperature reduces. Else, the algorithm accepts bad moves with probability determined by the temperature. By allowing bad moves to happen, the algorithm avoids getting stuck in local maxima and allows for the possibility of finding better moves.

Note: We used the cooling method called multiplicative cooling. Because of this the iterations of the program are consistently high regardless of the input temperature. This helped us to ensure that the solution can be reached even with low temperatures. However, for testing purposes, a faster cooling method could have been better.

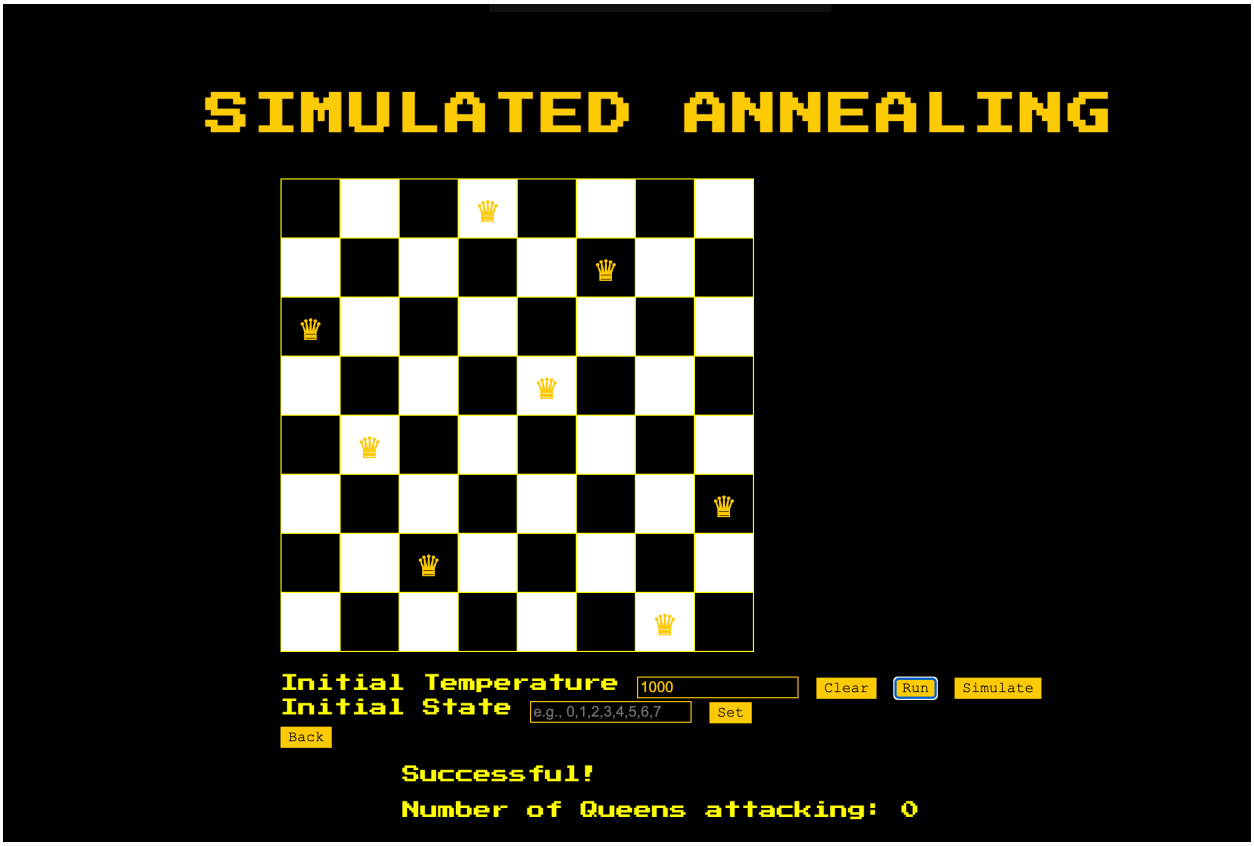
In this part we based our Simulated-Annealing function on this pseudo-code.

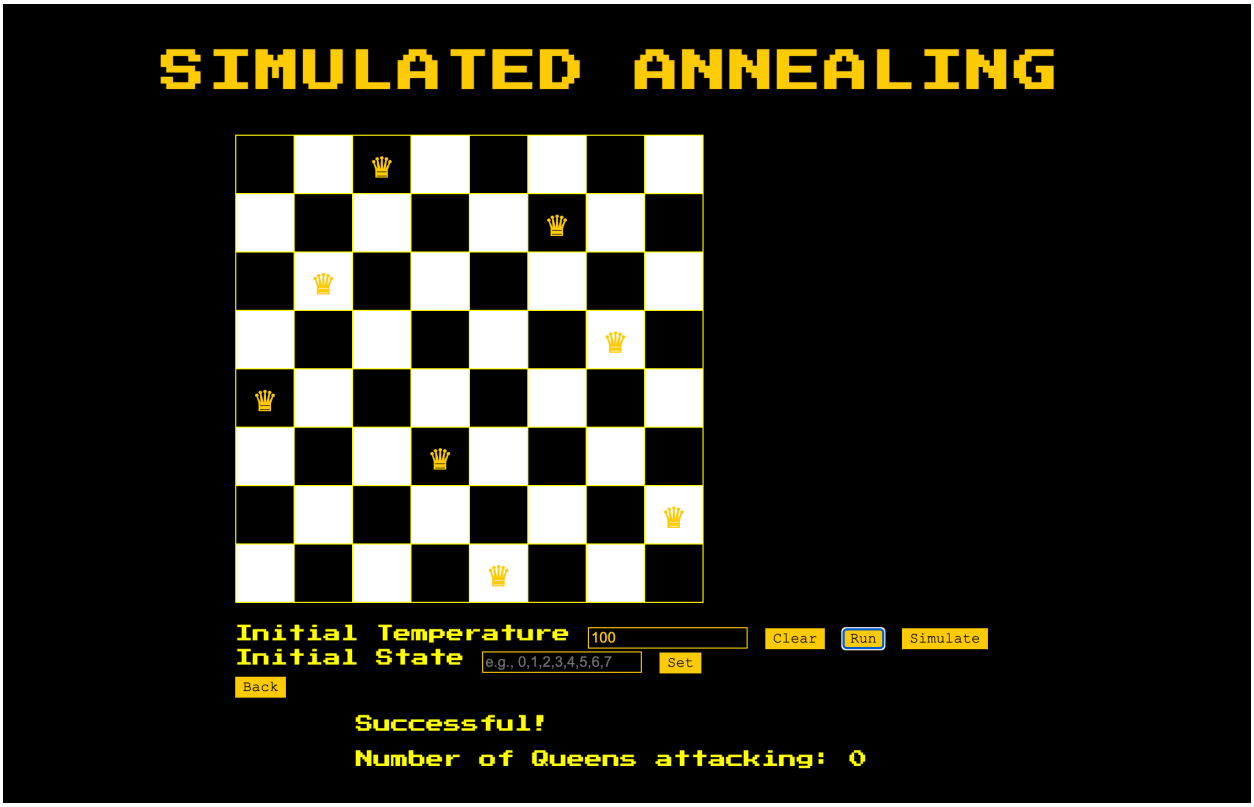
```
function Simulated-Annealing(problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     T, a "temperature" controlling prob. of downward steps

    current ← Make-Node(Initial-State[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        next ← a randomly selected successor of current
```

III. Results and Discussion

TEST RUN(S):





Spreadsheets of our previous test runs

Project	8 Queen			
	Temperature	Iterations	Average Time Taken(secs)	Sucess Rate
	100	20	0.03099999999	100%
	200	20	0.06200000001	100%
	300	20	0.156	100%
	400	20	0.11	100%
	500	20	0.156	100%
	2500	20	0.07800000001	100%
	5000	20	0.04699999999	100%
	10000	20	0.297	100%
	1000000	20	0.219	100%
	2000000	20	0.203	100%

Within a short time frame, the program were able to yield correct answer for temperature levels of both high and low. The two test runs have confirmed it.

IV. Conclusion

The program was implemented successfully. We were able to create a complete solution for the 8-queen puzzle using simulated annealing.

PS: When using the app via the .exe, when you want to use a different algorithm please reload the app and select the desired one.

V. Technology

- Virtual Studio Code + Python (Source code)
- Eel Python Library (GUIs)

PROJECT VI: Solving N-Queens Problem using backtracking

I. Introduction

Our project consisted of solving an N-queens problem using backtracking. The user inputs the size of the chessboard. The program then creates a blank chessboard with randomized initial position of the queens. The Backtracking algorithm was then implemented to solve the N-queen puzzle.

II. Methods

We investigate solving the problem by implementing:

- Generating a blank chessboard
- Generate the initial state of the queens.
- Implement rules and conditions for the queens.
- Implement constraints to improve Backtracking efficiency.
- Implement Backtracking algorithm to find the solution.

Step 1: Generating a blank chessboard.

Included in the chessboard.py file, we generated a blank matrix to store the location of the queens.

```
def draw_chess_board(self):
    for row in range(8):
        for col in range(8):
```

Step 2: Generating the initial state of the Queens

Used in the queen.py file, the initial_state function was used to inject the location of the queens randomly onto the chessboards.

```
def initial_state(n):
    arr = []
    for i in range(n):
        arr.append(random.randint(0, n))
    return arr
```

Step 3: Implement goal states, attack conditions for the queens as well as checking mechanism for the number of queens attacking each other.

- Attack condition: Queens attacking horizontally and diagonally.

```
def queenAttack(x_1, y_1, x_2, y_2):
    if x_1 == x_2:
        return True
    elif y_1 == y_2:
        return True
    elif abs(x_1 - x_2) == abs(y_1 - y_2):
        return True
    return False
```

- Number of queens attacking one another

```
def numQueensAttack(arr):
    #finds the number of queens that are attacking each other
    h = 0
    for i in range(len(arr)): #uses the first queen in the order to compare with other queens
        qR = arr[i]
        qC = i+1
        for j in range(len(arr)):
            if i == j: continue
            oR, oC = arr[j], j+1
            if queenAttack(qR, qC, oR, oC):
                h = h + 1
    return h
```

Using the first queen to check if it is attacking others. If yes add to counter of the number of queens that first queen is attacking

- Goal state: no queen is attacking any other queens.

```
def is_Complete(assignment):
    #check if every variable in assignment has a value
    assigned_columns = 0
    for i in assignment:
        if i != None:
            assigned_columns += 1
    if assigned_columns != len(assignment):
        return False

    if numQueensAttack(assignment) == 0:
        return True
    return False
```

Step 4: Implement constraints to improve Backtracking efficiency.

Constraints are methods that we implement to avoid the algorithm from investigating an inevitable failure path. Thus, improve efficiency.

1. Get legal placements

For us to implement constraints, a profile for legal moves must be created for every column on the board. At the start, every location is valid. As queens' location being injected into the board, the number of legal placements will reduce. Thus, lessening the amount states that the algorithm must evaluate.

```
def get_legal_placements(assignment, row, column, vars):
    n = len(assignment)
    assignment_copy = assignment.copy()
    assignment_copy[column] = row
    count = 0
    for c in vars:
        if c == column: continue #if the column being accessed is the one currently bein
        for i in range(n):
            assignment_copy[c] = i #assign a queen to the remaining columns
            if numQueensAttack(assignment_copy) == 0: count += 1 # if the assigned queen
            assignment_copy[c] = None

    return count
```

2. Most constrained variable

Defined in our program as [most_constrained_value], we used most constrained variable to help the algorithm to choose a location that has the most connection with other state space and use that for the placement of the queen. This is useless at the start, when the board is empty, because every placement will have an equal effect on its neighboring

state. This is useful later as more queens being place on the board, the constraint will allow the algorithm to choose the location that is most unaffected by the placement of the previous queens.

```
def most_contrained_value(assignment, vars, n):
    values = {}
    for i in vars:
        values.update({i: get_legal_moves(assignment, i, n)})

    max = -1
    max_index = -1
    for key, value in values.items():
        if value >= max:
            max = value
            max_index = key

    return max_index
```

3. Least constrained variable

Defined in our program as [least_constrained_values], we then have least constrained variable to help the algorithm to choose the location that would leave us with the most legal moves. This will help the algorithm to pick a queen placement least damaging to the placement option of future queens.

```
def least_constrained_values(a, column, vars):
    assignment = a.copy()
    domain = [] #list to hold the LCV queen positions

    n = len(a)

    for row in range(n):
        assignment[column] = row
        #here we will check if the above queen position is violating any contraints (attacking any queen)
        if numQueensAttack(assignment) > 0:
            assignment[column] = None
            continue
        legal_placements = get_legal_placements(assignment, row, column, vars)
        domain.append(Position(row, column, legal_placements))

    sorted_domain = sorted(domain, key=lambda placement: placement.value, reverse=True)


    return sorted_domain
```

Step 5: Implement Backtracking to find the solution.

Backtracking search algorithm is a depth-first search algorithm that investigates the states following an initial choice. The algorithm will then search until it reaches the end of those states then determine if the end state was the solution. If not, then it will recursively backtrack and try a different option until it finds the correct path to the solution or until it ran through all the possible state and determined that there is no solution to the problem.

In this part we based our Backtracking function on this pseudo-code.

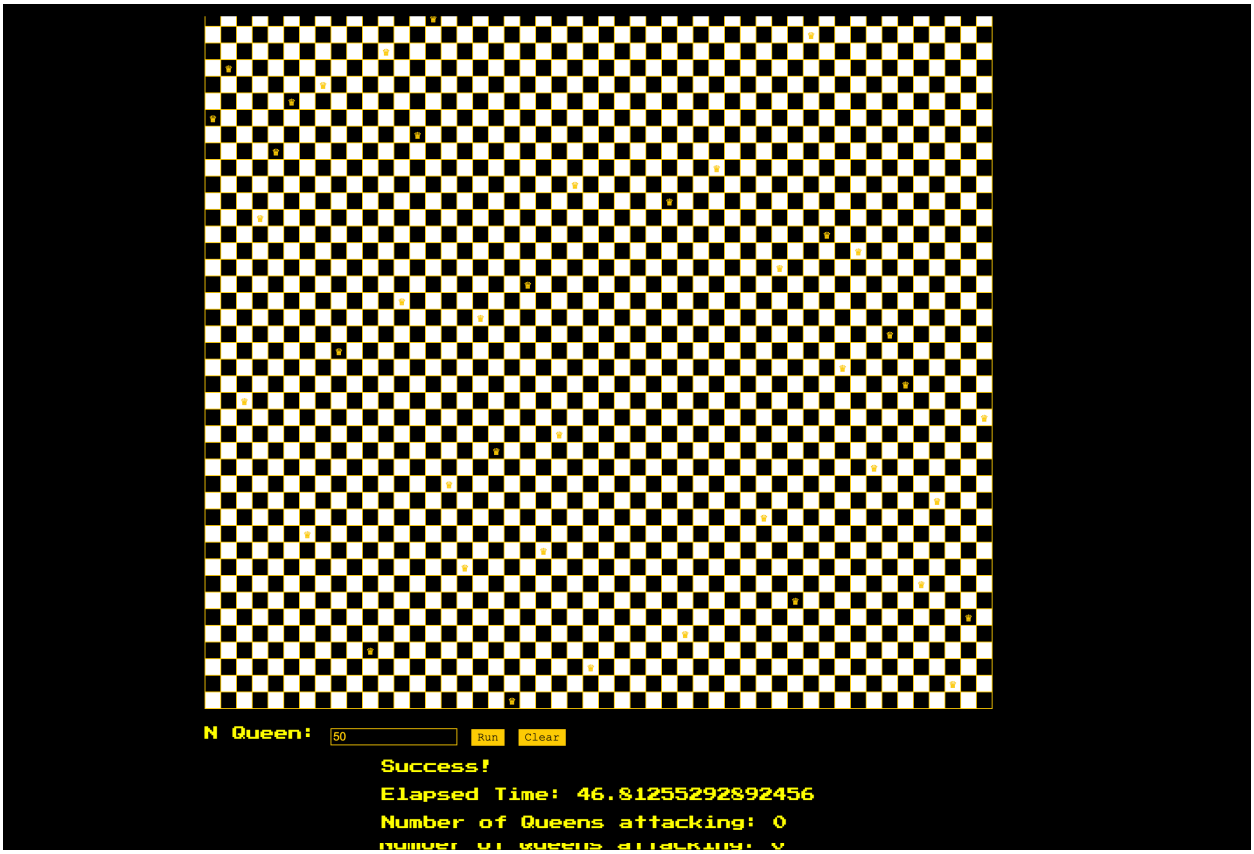
```
function BACKTRACKING-SEARCH( csp) returns a solution, or failure
    return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING( assignment,csp) returns a solution, or
failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE( Variables[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to Constraints[csp] then
            add { var = value } to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failue then return result
            remove { var = value } from assignment
    return failure
```

III. Results and Discussion

TEST RUN(S):

Spreadsheets of our previous test runs



Project	n Queen	Backtracing Algorithim		
	N	Iterations	Average Time Taken(secs)	Sucess Rate
	8	20	0.125	100%
	10	20	0.07800000001	100%
	15	20	0.109	100%
	30	20	53.203	100%
	50	20	Takes long	100%
	100	20	Takes long	100%

The documented results showed that the program was successful 100% of the time. With longer process times needed for the project to finish depending on the board size.

IV. Conclusion

The program was implemented successfully. We were able to create a complete solution for the N-queen puzzle using backtracking algorithm.

V. Technology

- Virtual Studio Code + Python (Source code)
- Eel Python Library (GUIs)