THE HONG KONG
UNIVERSITY OF SCIENCE
AND TECHNOLOGY

COMP3211: Fundamental in AI
Assignment#3

1. **MRV and LCV Implementation in CSPs for Map Coloring (Problem I).**
2. **Simulated Annealing Implementation for 8-queens problem (Problem II).**
3. **Value Iteration Implementation (Problem III)**

21043976 ZHANG, Aoxuan[azhangao@connect.ust.hk]

HongKong, Dec 2023

1. MRV and LCV Implementation in CSPs for Map Coloring (Problem I).

Introduction:
The State Coloring Problem is a classic problem in computer science and graph theory. The goal is to assign colors to states on a map such that no two adjacent states have the same color. In this report, we present the implementation of two heuristics: Minimum Remaining Values (MRV) and Least Constraining Value (LCV) to solve the State Coloring Problem.

MRV Heuristic:
The MRV heuristic is used to select the next state to be evaluated during the search process. The calcMRV() function implements this heuristic. It takes the following parameters:

mc: The Map instance representing the map and its adjacency matrix.
solution: A dictionary mapping states to their assigned colors (-1 if not colored).
domDict: A dictionary mapping states to their valid domain colors.

The function starts by initializing variables to store the index of the state with the minimum remaining values and the minimum count of remaining values found so far. It then loops through all states and checks if a state is unassigned. For each unassigned state, it counts the number of available colors in its domain and compares it with the minimum count. If the count is smaller, the state index and the minimum count are updated. Finally, the function returns the index of the state with the minimum remaining values. If no unassigned state is left, it returns -1.

LCV Heuristic:
The LCV heuristic is used to select the color for a state based on how many times each color appears in the domains of neighboring states. The calcLCV() function implements this heuristic. It takes the following parameters:

mc: The Map instance representing the map and its adjacency matrix.
domDict: A dictionary mapping states to their valid domain colors.
solution: A dictionary mapping states to their assigned colors (-1 if not colored).
states: The index of the states to expand.

The function starts by initializing a dictionary to count the occurrence of each color in the neighboring states. It then loops through all states and checks if a state is a neighbor and unassigned. For each available color in the domain of the neighboring state, it increments its count in the color count dictionary. After counting the occurrences of colors in the neighboring states, the function sorts and returns the colors based on their occurrence count. The color with the least occurrences comes first as it is the least constraining.

Solution:
The implemented heuristics successfully solve the State Coloring Problem. The program generates a solution where each state is assigned a color. My solution is printed as follows:
Solution found:
NSW = B
NT = B
Q = G
SA = R

WA = G
V = G
T = Any color (Since Tasmania is not adjacent to any other state)

Conclusion:
The MRV and LCV heuristics are effective in solving the State Coloring Problem. The MRV heuristic helps in selecting the most promising state to evaluate, while the LCV heuristic assists in selecting the least constraining color for a state based on its neighboring states. By using these heuristics, the program efficiently finds a valid coloring solution for the given map.

2. Simulated Annealing Implementation for 8-queens problem (Problem II).

a. Introduction
The N-Queens problem is a classic puzzle that involves placing N queens on an N×N chessboard in such a way that no two queens threaten each other. In this report, we present an implementation of the Simulated Annealing algorithm to solve the N-Queens problem.

b. Algorithm Overview
The Simulated Annealing algorithm is a probabilistic optimization algorithm that is inspired by the annealing process in metallurgy. It is commonly used to solve combinatorial optimization problems. The algorithm starts with an initial solution and iteratively explores the solution space by making random moves. The algorithm accepts moves that improve the solution or moves that do not improve the solution with a certain probability that decreases over time. This allows the algorithm to escape local optima and converge to a near-optimal solution.

c. Implementation Details
The implementation of the Simulated Annealing algorithm for the N-Queens problem is done in Python. The code consists of two main classes: Board and SimulatedAnnealing. The Board class represents the chessboard and its state, while the SimulatedAnnealing class implements the Simulated Annealing algorithm.

The SimulatedAnnealing class has the following methods:
__init__(): Initializes the SimulatedAnnealing object with the board and parameters.
run(): Runs the Simulated Annealing algorithm to solve the N-Queens problem. It iterates for a fixed number of epochs and updates the temperature according to the decay factor. Inside the loop, it generates a new set of queen positions, calculates the change in cost between the current and new positions, and decides whether to accept the new positions based on the change in cost and the calculated probability.

d. Experimental Setup
To evaluate the effectiveness of the Simulated Annealing algorithm for the 8-Queens problem, we ran the algorithm with different parameter combinations. We used the following parameter settings:
Setting I: T = 4000, e = 10000, d = 0.9
Setting II: T = 2000, e = 10000, d = 0.9
Setting III: T = 1000, e = 10000, d = 0.9

For each parameter setting, we ran the algorithm 10 times to observe the variability in the results.

e. Results

The table below presents the performance of the Simulated Annealing algorithm under different settings:

Settings	Average Time to Convergence	Success Rate
Setting I	7.928ms	20.00%
Setting II	3.952ms	20.00%
Setting III	1.107ms	20.00%

The average time to convergence is the average time taken by the algorithm to find a solution across the 10 runs for each parameter setting. The success rate indicates the proportion of successful runs where the algorithm found a solution without any attacks between queens.

Settings	Average Time to Convergence	Success Rate
Setting I	14.824ms	13.00%
Setting II	22.910ms	7.00%
Setting III	5.300ms	16.00%

n=100, 100 runs

Settings	Average Time Running Time	Success Rate
Setting I	9.141ms	13.90%
Setting II	11.010ms	12.60%
Setting III	9.426ms	11.50%

n=1000, 1000 runs

f. Conclusion

In this report, we presented an implementation of the Simulated Annealing algorithm to solve the N-Queens problem. The algorithm provides a probabilistic approach to finding a solution and is effective in escaping local optima. The code can be used to solve N-Queens problems of different sizes by adjusting the parameters. Further improvements can be made to optimize the algorithm or extend it to solve other combinatorial optimization problems.

3. Value Iteration Implementation (Problem III)

a. Introduction:

This report presents the results of applying the Value Iteration algorithm to the Wumpus World problem. The objective of the problem is to control an agent as it navigates through a grid-like environment in search of gold while avoiding deadly pits and the Wumpus creature. The Wumpus World is represented as a 4x4 grid, where each cell represents a state. The agent starts at grid coordinate (0,0) and has the following objectives:

Finding the gold, which provides a significant positive reward (+10).

Avoiding pits and the Wumpus, which are associated with negative penalties (-5 for each pit and -10 for the Wumpus).

Minimizing the incurred movement penalty (-0.4 for each non-goal cell).

The algorithm used to solve the problem is Value Iteration, which is an iterative algorithm for finding the optimal values and policies for each state in an MDP (Markov Decision Process). The algorithm consists of iteratively updating the utility values of the states until convergence is reached.

b. Parameter Settings

The program was executed with the following parameter settings:

Setting I: gamma = 0.3, eta = 0.1, max_iter = 10000

Setting II: gamma = 0.6, eta = 0.1, max_iter = 10000

Setting III: gamma = 0.9, eta = 0.1, max_iter = 10000

c. Results

The program outputs the utilities and policies for each state in the Wumpus World for each parameter setting. The utilities represent the expected cumulative discounted reward for each state, considering future states. The policies indicate the best action to take in each state, given the utilities.

The results for each parameter setting are as follows:

1) Setting I: $\gamma = 0.3$, $\eta = 0.1$, $e = 10000$

   Utilities and Policy for the Given Wumpus World:
   -0.41 → | 0.14 → | 2.29 → | 10.94 ← |
   -0.52 → | -0.38 → | 0.20 ↑ | 2.29 ↑ |
   -0.54 ↓ | -0.66 ↑ | -4.99 ↑ | -0.01 ↑ |
   -0.43 ← | -10.34 ↑ | -5.41 → | -0.41 → |

2) Setting II: $\gamma = 0.6$, $\eta = 0.1$, $e = 10000$

   Utilities and Policy for the Given Wumpus World:
   1.26 → | 3.37 → | 7.31 → | 14.80 ← |
   0.34 → | 1.49 → | 3.60 ↑ | 7.31 ↑ |
   -0.30 ↑ | 0.08 ↑ | -3.13 ↑ | 3.07 ↑ |
   -0.45 ← | -10.35 ↑ | -5.17 → | 0.76 ↑ |

3) Setting III: $\gamma = 0.9$, $\eta = 0.1$, $e = 10000$

Utilities and Policy for the Given Wumpus World:
27.79 → | 32.73 → | 38.47 → | 45.14 ← |
24.50 → | 28.57 → | 33.25 ↑ | 38.47 ↑ |
21.24 ↑ | 24.15 ↑ | 23.93 ↑ | 32.27 ↑ |
17.31 ↑ | 10.48 ↑ | 17.97 → | 26.78 ↑ |

d. Conclusion

The Value Iteration algorithm was successfully applied to the Wumpus World problem with different parameter settings. The results show the utilities and policies for each state in the Wumpus World, considering the objectives of finding the gold, avoiding pits and the Wumpus, and minimizing movement penalties. The parameter settings influenced the values and policies obtained, demonstrating the impact of discount factor and convergence threshold on the solution.