# Homework #2: Local search approches & Constraint satisfaction Problem

MESIIN476024 - AI algorithms

**Due: Sunday, December 1, 2024, at 11:59 p.m.**

## Instructions

Read all the instructions below carefully before starting the assignment and making your submission.

- The homework **may be completed individually or in pairs**. Be sure to include the names and lab group of both students in your submission.
- Collaborative discussions are encouraged to facilitate a high-level understanding of the concepts, but sharing exact solutions is strictly prohibited.
- **Notebooks submitted with only code, without explanations for implementation choices, discussions of the problem context, descriptions of the methodology, solution approaches, and analysis of results, will not be evaluated.**
- **This assignment is not intended to assess your Python programming skills. It is designed to evaluate your understanding of the concepts covered in class, your ability to apply these concepts to solve complex problems, and your analytical and critical thinking skills in addressing these challenges.**
- Up to 2 bonus points will be awarded for submissions that demonstrate deep reflection, innovation, and notable commitment in the final deliverable.
- If specific instructions or details are not explicitly provided, you are free to make reasonable assumptions as long as they are justified and consistent with the context.
- **Submission Requirements:**
  - Your submission should be a **zip file** containing a Jupyter notebook (.ipynb) with the Python code for both exercises, along with any additional scripts or files you consider relevant. Name the zip file according to this format: HW2_YourLabGroup_YourName(s) (e.g., HW2_DIA3_PierreFEUILLE).
  - **Due date: December 1, 2024.** Late submissions will incur a penalty of -2 points per day after the due date.
  - Upload your assignments to the designated submission space on DeVinci Learning by the deadline. (Navigate to Homework → Homework#2 → Deposit Homework 2 → DIA_).
- Cite all external sources used. This assignment is your individual responsibility, and plagiarism will not be tolerated. Software will be used to check for similarities in both writing and code. If you have used ChatGPT or any generative AI, please specify on which aspects it was used.

## Topics: Points

- **Exercise 1**: 10 Points
- **Exercise 2**: 10 Points

## Preamble

This assignment consists of two exercises. The first exercise focuses on Local Search techniques, specifically applying Genetic Algorithms (GA) to solve complex optimization problems. The second exercise involves solving the Battleship Puzzle using Constraint Satisfaction Problem (CSP) methods. Both exercises will test your understanding of search algorithms, optimization strategies, and constraint propagation techniques.

# Exercise 1: Solving Optimization Problems with Genetic Algorithms [10 Points]

In this exercise, you will apply a Genetic Algorithm (GA) to solve complex optimization problems. **Select and complete "only one" of the two challenges below**: **Problem A** for optimizing a neural network or **Problem B** for solving the knapsack problem.

## Problem A. Genetic Algorithm for Optimizing a Neural Network

In this exercise, you will implement a Genetic Algorithm to optimize the weights and biases of a neural network without using traditional gradient-based methods. The primary objective is to explore how GA can be applied to evolve the parameters of a simple neural network to minimize the Mean Squared Error (MSE) on a synthetic dataset (see Figure 1).

**Objective:** Use a GA to optimize a neural network's weights and biases to achieve low MSE.

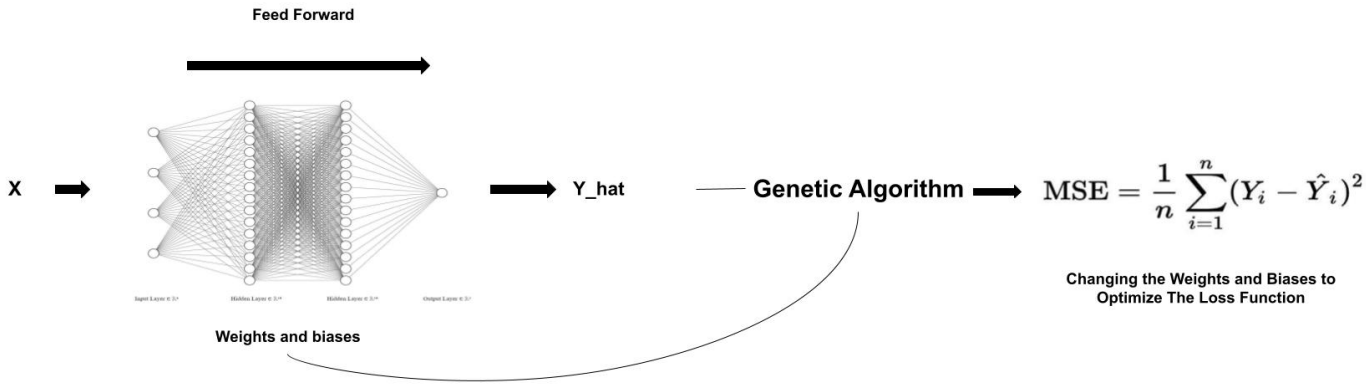


Figure 1: Neural Network Optimization using Genetic Algorithm.

**Instructions:**

1. **Dataset Creation:** Use `make_blobs` from `scikit-learn` to generate a synthetic dataset with two centers and four features.
2. **Neural Network Design:** Construct a neural network with the following:
   - **Architecture:** Input layer with 4 neurons, Two hidden layers with 16 neurons each, and an Output layer with 1 neuron.
   - **Activation Function:** Use the sigmoid activation function for each layer.
3. **GA Implementation:** Implement a GA to optimize the neural network's weights and biases.
   - Define a population of potential solutions (sets of weights and biases).
   - Evaluate each individual's fitness based on the Mean Squared Error (MSE) between predicted and actual values.
   - Implement selection (*e.g.*, tournament selection), crossover (*e.g.*, two-point crossover), and mutation functions.
4. **GA Parameters:** Configure the GA with the following parameters:
   - Set the population size to 100, the mutation rate to 0.1, and the number of generations to 200.
   - Use elitism to retain the top 10 solutions from each generation.
5. **Optimization Process:** Run the GA and visualize the optimization process:
   - Track and plot the change in MSE over generations.

- Update the neural network with the best set of weights and biases at each generation.
- Display the final MSE on the dataset after the last generation.

6. **Discussion:**

   - Evaluate how the GA's parameters and strategies affect the neural network's performance.
   - Compare GA-based optimization results to traditional methods (like gradient descent) in terms of accuracy and runtime.
   - Discuss the advantages and limitations of using GAs for neural network optimization.

## Problem B. Genetic Algorithm for Solving the Knapsack Problem

The knapsack problem is a well-known combinatorial optimization problem, where the goal is to select a subset of items, each with a specific weight and value, to maximize the total value without exceeding the defined weight capacity of the knapsack.

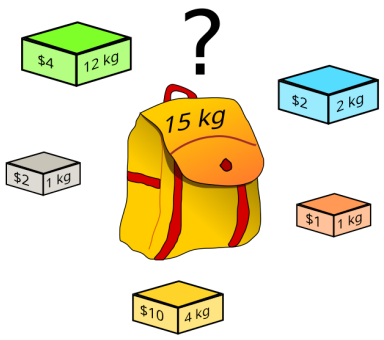


Figure 2: Knapsack Problem Setup

**Objective:** Implement a Genetic Algorithm in Python to solve the Knapsack optimization problem, using diverse GA strategies and comparing them against a greedy or a local search algorithm.

**Problem Description:** A logistics company is preparing a freight container for an overseas shipment. The container has a maximum weight capacity of 10.000 kg. The company has a selection of 500 packages, each with a specific weight and value, representing its importance to the shipment. The challenge is to select packages in a way that maximizes the total value of the shipment without exceeding the container's weight capacity.

**Instructions:**

1. **Data Input:**

   - Load item data (weights and values) from a CSV file (**items.csv**), which contains the weight and value of each package.

2. **Genetic Algorithm Implementation:** Implement a GA with the following components:

   - Define the population, fitness function, selection method, crossover method, mutation, and GA parameters.

3. **Testing and Comparison:**

   - Test at least two different combinations of selection and crossover methods to evaluate their impact on solution quality.
   - Implement a greedy or local search (Simulated Annealing or Hill Climbing) algorithm to use as a baseline for comparison. The greedy algorithm can be one covered in class or one you design.
   - Compare the results of the GA and the greedy approach in terms of solution quality, convergence speed, and runtime.

4. **Output & Discussion:**
    - Present the best solution found (selected items) and its total value.
    - Plot the evolution of fitness over generations.
    - Discuss your implementation choices, comparing the strengths and limitations of different GA strategies relative to the greedy or local search solution.

## Exercise 2: Solving a $6 \times 6$ Battleship Solitaire Puzzle Using Constraint Satisfaction Problem (CSP)

In this exercise, you will write a program to solve Battleship Solitaire Puzzles by formulating it as a Constraint Satisfaction Problem (CSP).

Battleship Solitaire is a single-player variant of the classic Battleship game. Unlike the two-player version, Battleship Solitaire provides hints on the number of ship segments within each row and column. Your goal is to deduce the exact placement and orientation of each ship segment on the grid.

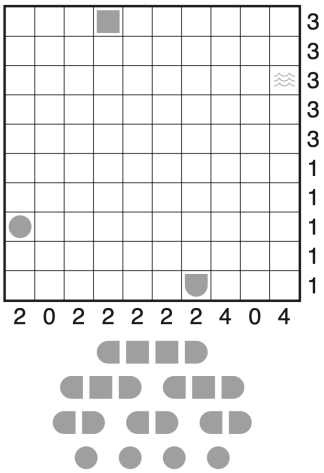


Figure 3: Battleship Puzzle Grid with Ship Shapes and Hints.

### Game Rules

The rules for Battleship Solitaire are as follows:

1. The puzzle is set on an $N \times N$ grid with a hidden fleet of ships. The fleet consists of:
    - **Battleships**: 4 cells long ($1 \times 4$)
    - **Carriers**: 3 cells long ($1 \times 3$)
    - **Destroyers**: 2 cells long ($1 \times 2$)
    - **Submarines**: 1 cell long ($1 \times 1$)
2. Each cell in the grid represents either **water** (empty) or **part of a ship**.
3. **Ship shapes:**
    - Battleships, carriers, and destroyers are linear, occupying consecutive cells.
    - Submarines occupy a single cell.
    - Ships can be placed either horizontally or vertically but not diagonally.
4. **Constraints provided:**
    - **Fleet constraints**: Specifies the number of each type of ship in the fleet.
    - **Row and Column constraints**: Numbers at the left or right of each row and at the top or bottom of each column indicate the count of ship segments in that row or column.

- **Hint constraints**: Some cells may have predefined values, specifying either water or a specific ship segment.

5. Ships cannot touch each other, even diagonally. In other words, each ship must be surrounded by at least one cell of water on all sides, including corners.

You can practice Battleship Solitaire puzzles here.

## Task Overview

Your task is to solve a $6 \times 6$ Battleship Solitaire Puzzle using CSP techniques. Specifically, you will:

1. **Program the CSP Solver**: Write a program that encodes the Battleship Solitaire Puzzle as a CSP. Implement a CSP solver using methods such as backtracking search, forward checking, and AC-3 arc consistency.
2. **Optimize with Heuristics**: Enhance the efficiency of your solution by implementing variable and value selection heuristics, such as the Minimum-Remaining-Value, Degree heuristic, and Least-Constraining-Value heuristic.
3. **Evaluate Performance**: Analyze the impact of different heuristics and constraint propagation methods on the performance of your CSP solver.
4. **Optional (Bonus +2 points)**: Implement a minimum-conflicts heuristic to further refine your solution's accuracy.
5. **Reflect on CSP Approach**: Evaluate the effectiveness of the CSP approach for solving Battleship puzzles. Discuss any challenges encountered and how you overcame them.

**Note:** You may implement CSP search strategies like backtracking, forward checking, and AC-3 arc consistency yourself, or you may use constraint programming libraries such as OR-Tools.

## Input Format

Your program will be tested on various Battleship Solitaire puzzles. The input format is as follows:

- The first line describes the row constraints as a sequence of $N = 6$ numbers. These constraints are typically displayed to the left or right of each row in puzzle examples.
- The second line describes the column constraints as a sequence of $N = 6$ numbers. These constraints are usually shown at the top or bottom of each column in puzzle examples.
- The third line specifies the number of each type of ship. The four numbers represent the counts of submarines (3), destroyers (2), cruisers (1), and battleships (0), in that order.
- The remaining lines provide a $6 \times 6$ grid representing the initial puzzle layout. There are eight possible characters for each cell:
  - '0' (zero) represents no hint provided for that square.
  - 'S' represents a submarine.
  - '.' (period) represents water.
  - '<' represents the left end of a horizontal ship.
  - '>' represents the right end of a horizontal ship.
  - '^' represents the top end of a vertical ship.
  - 'v' (lower-case letter 'v') represents the bottom end of a vertical ship.
  - 'M' represents a middle segment of a ship, either horizontal or vertical.

**Example of an Input File:**

**Input file**

```
221212
401311
3210
000000
000000
000000
00M000
000000
000000
```

**Illustration of the Input File**

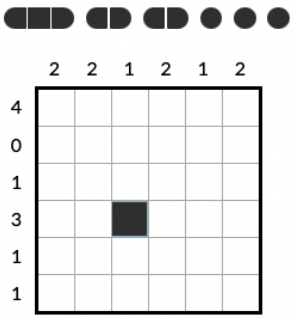


Figure 4: Example $6 \times 6$ Battleship Puzzle with Fleet, Row, and Column Constraints.

## Output Format

The output should be a completed $6 \times 6$ grid representing the solution to the puzzle. Ensure there are no '0' characters remaining in the output. Each cell should display one of the seven possible values, indicating the correct placement of each ship segment.

Below is the correct output for the example given earlier.

**Output file**

```
<>.<>.
......
.....S
.<M>..
.....S
S.....
```

**Illustration of the Output Solution**

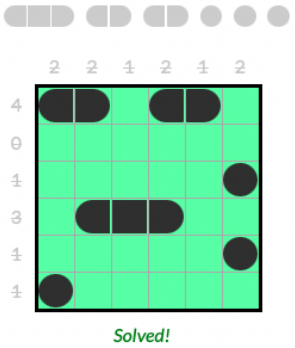


Figure 5: Solution for the $6 \times 6$ Battleship Puzzle shown in Figure 4.

Additional examples of input and output files are available here.

**Note:** You are welcome to propose alternative formats for displaying the output.

## Resources

For additional information on Battleship puzzles, refer to:

- Conceptis Puzzles - Battleship
- Wikipedia - Battleship (puzzle)
- Barbara M. Smith, *Constraint Programming Models for Solitaire Battleships*, 2006, available at bit.ly/cspBs