REPORT CI course - Ali Edrisabadi

All the project and Labs have been done by me based on my understanding.

Lab0 done in first week!

reviews: Issues · AliEdrisabadi/CI2024_lab0

repository: AliEdrisabadi/CI2024_lab0

LAB2

reviews: Issues · AliEdrisabadi/CI2024_lab2

repository: AliEdrisabadi/CI2024_lab2

problem. Given a list of cities and the distances between each pair, find the shortest tour that visits every city exactly once and returns to the start.

This lab implements and compares two approaches:

1. a fast **Greedy** baseline
2. an **Evolutionary Algorithm** that searches over routes

The dataset used in my runs is the **Italian cities** Distances are the great-circle distances in kilometers.

Solution 1 — Greedy

Idea. Start from a chosen city and repeatedly move to the **nearest unvisited** city until all are visited; finally return to the start. This is $O(N^2)$ and very fast, but **not optimal** in general.

My implementation highlights

- Uses the precomputed distance matrix to pick the nearest unvisited neighbor.
- Logs each hop (city → city, distance).
- Returns the closed tour (route + return to origin) and its total distance.

Solution 2 — Evolutionary Algorithm (EA)

Representation. A route is a **permutation** of city indices

Fitness. Total tour distance (shorter is better). I use `1 / distance` for roulette selection.

Operators

- **Selection:** roulette-wheel selection
- **Crossover:** “segment copy + fill” from the other parent
- **Mutation:** swap two random positions
- **Elitism:** *not* explicit in the current loop

Parameters (my runs)

parameter	value
population_size	100
generations	1000
mutation_rate	0.01

Loop

1. Initialize population with random permutations.
2. For each generation:
    - Compute fitness → 1 / route_distance
    - Build a new population by selecting parent and crossing over and mutating.
    - Track and keep the best route found so far

Observations

- **Greedy** is instant and often reasonable, but it can get trapped by locally short hops that make later edges very long.
- **EA** needs more time but can discover **shorter tours**, especially if you:
    - increase generations,
    - add elitism,
    - seed part of the population from a good Greedy tour,
    - add a local optimization step

- The total distance depends strongly on the **start city → greedy** and on the **random seed** (for EA).

LAB3

reviews: Issues · AliEdrisabadi/CI2024_lab3

repository: AliEdrisabadi/CI2024_lab3

Results: Optimal Path Search → Implements the A* algorithm with the Manhattan Distance heuristic for efficient problem-solving.

Visualization → Displays the initial puzzle, step-by-step solution process, and the final solved state

This lab tackles the classic n^2-1 sliding-tile puzzle. The board is an `n × n` grid with tiles `1..(n²−1)` and a blank `0`. A move slides a tile into the blank; the goal is to reach the canonical ordering (blank in the bottom-right) from a scrambled state with as few moves as possible.

For example, for `n = 4` the goal is:

```
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15  0
```

Algorithms Implemented

A* Search (with Manhattan distance)

A* is an informed search that prioritizes nodes by

f(s)=g(s)+h(s)f(s) = g(s) + h(s)

f(s)=g(s)+h(s)

where:

- `g(s)`: cost from start to state `s` (number of moves),
- `h(s)`: heuristic estimate to the goal.

We use **Manhattan distance**: for each tile, the sum of |Δrow| + |Δcol| to its goal position, ignoring the blank.

Why Manhattan?

It is admissible and consistent for sliding tiles, so A* with this heuristic **finds optimal solutions** (fewest moves), provided enough memory.

State & Moves

- State stored as a NumPy `n×n` array; the blank is `0`.
- Legal actions are swaps of the blank with its up/down/left/right neighbor.
- States are serialized

Priority queue

- The *open set* is a min-heap keyed by `f = g + h`.
- Each entry: `(priority, serialized_state, path, g)`

Observations & Notes

- **Optimality:** With Manhattan distance, A* returns an **optimal** (fewest-moves) solution.
- **Scalability:** Runtime and especially **memory** grow fast with board size. 3×3 is trivial; 4×4 may already stress memory/time depending on scramble; 5×5 becomes impractical for pure A*.
- **Randomization:** Using `RANDOMIZE_STEPS` valid moves guarantees the resulting start state is **solvable**.
- **Tuning:**
  - Increase `PUZZLE_DIM` to experiment with larger boards.
  - Adjust `RANDOMIZE_STEPS` to control scramble difficulty.
  - Add a time limit or node cap around the loop to prevent pathological cases.

CI2024 – Symbolic Regression Project (s316628)

Abstract

This project tackles **symbolic regression**: learning closed-form expressions that map inputs xxx to outputs yyy without fixing a parametric model in advance. I implemented a compact tree-based genetic programming (GP) approach and an evaluation pipeline with **protected math (safe ÷,log,,exp,pow\div,\log,\sqrt,\exp,\text{pow}÷,log,,exp,pow)** to keep the search and the plots numerically stable. The final deliverable is a single submission file `s316628.py` exposing eight functions `f1..f8`, plus small scripts to evaluate and visualize results.

Problem statement

Given N pairs (x_i, y_i) with x_i in R^d and y_i in R, learn a human-readable function f: R^d → R such that f(x_i) ≈ y_i, while keeping both prediction error and expression **complexity low**.

Data

Each dataset lives in `data/problem_k.npz` with:

- `x` : shape `(n_vars, n_samples)`
- `y` : shape `(n_samples,)`

We treat the eight problems independently and learn/assess one function per dataset.

Method – one-paragraph overview

Functions are represented as **expression trees** whose internal nodes are primitives (e.g., `+ - * / sin cos exp log abs pow`) and leaves are **terminals** (variables `x[j]` and constants). A GP loop evolves a population of trees with **tournament/rank selection**, **subtree & point mutation**, **subtree crossover**, and **elitism**. Fitness is the **mean squared error (MSE)** over the training samples. To prevent numeric explosions that can derail both training and plotting, all risky operations use **protected variants** (safe division, clipped `exp` / `pow`, etc.) and outputs are post-processed with a `_sanitize` pass. The best expression for each problem is exported to `s316628.py` and evaluated/visualized with `eval_and_plot.py`.

Search & training details

- **Initialization:** random trees up to a max depth.
- **Selection:** tournament or rank selection (both tried).
- **Variation:**
  - *Crossover:* swap random subtrees.
  - *Subtree mutation:* replace a randomly chosen subtree with a fresh random one.
  - *Point mutation:* change an operator or a terminal.
- **Elitism:** top 20% carried over untouched.
- **Objective:** minimize MSE; invalid evaluations get `inf`.
- **Stability first:** protected ops during training **and** in the final exported functions.

Evaluation pipeline

For each problem the script prints MSE and saves two figures:

1. **Predicted vs True** (perfect fit = points on the diagonal).
2. **Residual histogram** (ideal = narrow, zero-centered bell).

   Residuals are clipped to ±1e6 in plots to keep figures readable.

Results (this submission)

```
def f1(x: np.ndarray) → np.ndarray:
    return _sanitize(np.sin(x[0]))


def f2(x: np.ndarray) → np.ndarray:
    return _sanitize((pdiv(((3.3904 * x[0]) * (ppow(9.4913, (6.9304 + x[0])) *
```

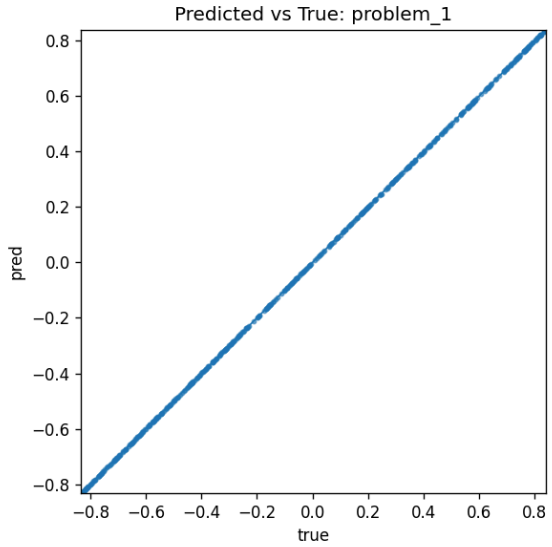
```
pexp(3.3904))), pdiv(plog(np.abs(psqrt(-4.7216))), 9.4913)) + (((x[1] + x[2]) + x[0]) * (ppow(pexp((4.7535 - x[0])), ((x[0] + 9.4913) + (x[1] + x[2]))) * np.abs(((9.4913 + x[0]) + plog(4.7535)))))))


def f3(x: np.ndarray) → np.ndarray:
    return _sanitize((np.abs(-8.0045) * ((((np.sin(x[1]) - (x[1] + x[1])) + np.sin((x[1] * 0.9065))) + np.sin(((x[1] * 0.9065) * 0.9065))) + np.abs((np.tanh((x[2] * x[0])) - x[0])))))


def f4(x: np.ndarray) → np.ndarray:
    return _sanitize(pexp(pdiv(np.cos((np.tanh(pdiv(x[1], 3.2972)) * (np.tanh(x[1]) + x[1]))), np.sin(pexp(np.tanh(psqrt(-6.9863)))))))


def f5(x: np.ndarray) → np.ndarray:
    return _sanitize(ppow(plog(np.tanh(((5.5394 - np.tanh(x[0])) - np.sin(plog(x[0]))))), (np.cos(ppow(psqrt(pdiv(-8.6398, x[0])), psqrt(pdiv(-8.6398, x[0])))) + (np.cos(plog((-3.5072 * -2.8384))) + x[1]))))


def f6(x: np.ndarray) → np.ndarray:
    return _sanitize(((psqrt(pdiv(psqrt(-5.8150), np.sin(np.tanh(3.3138)))) * x[1]) - ((x[0] + np.tanh(np.tanh(pexp(-5.8150)))) * np.tanh(np.abs(np.sin(pdiv(-9.5093, -4.4953)))))))


def f7(x: np.ndarray) → np.ndarray:
    return _sanitize((pexp(((x[1] * x[0]) + psqrt(plog((x[0] - x[1]))))) + (np.abs(plog((x[0] - x[1]))) * ppow(pexp(psqrt((x[0] * x[1])))), psqrt(plog((x[0] - x[1]))))))))


def f8(x: np.ndarray) → np.ndarray:
    return _sanitize(ppow(((np.tanh(np.tanh(np.tanh(0.1487))) * x[5]) + ((np.tanh(np.tanh(-1.5219)) * np.sin(plog(x[4]))) + (np.abs(np.abs(x[5])) * x[5]))),
```

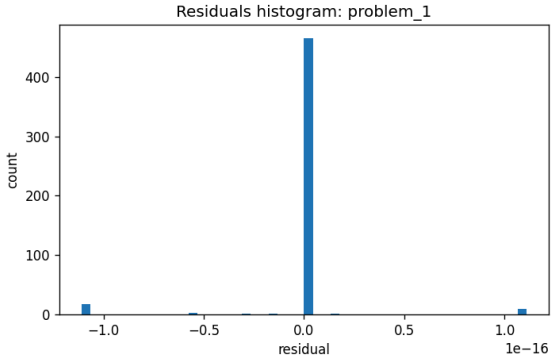
```
np.abs(((psqrt(pdiv(x[4], 1.8753)) * -3.9783) + (np.abs(np.abs(2.5761)) * x[5])))))
```

Problem	X shape	y shape	Variables	MSE (eval)
P1	(1, 500)	(500,)	['x0']	**7.12594e-34**
P2	(3, 5000)	(5000,)	['x0', 'x1', 'x2']	**9.56306e+12**
P3	(3, 5000)	(5000,)	['x0', 'x1', 'x2']	**117.037**
P4	(2, 5000)	(5000,)	['x0', 'x1']	**3.71471**
P5	(2, 5000)	(5000,)	['x0', 'x1']	**2.23596e-18**
P6	(2, 5000)	(5000,)	['x0', 'x1']	**2.53209e-06**
P7	(2, 5000)	(5000,)	['x0', 'x1']	**54.3914**
P8	(6, 50000)	(50000,)	['x0', 'x1', 'x2', 'x3', 'x4', 'x5']	**700139**

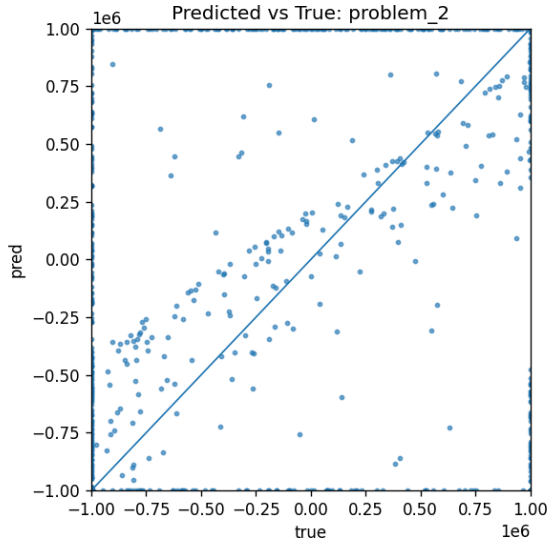
- **Excellent:** P1, P5, P6 (near-zero error).
- **Good:** P4 (captures the nonlinearity but shows mild saturation and bias at high values).
- **Okay:** P3 (nonlinear trend captured).
- **Weak:** P2 and P8

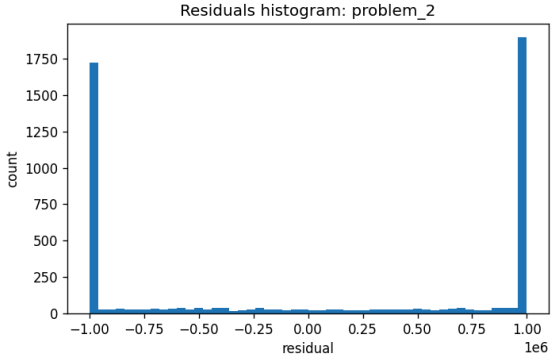
Visual diagnostics



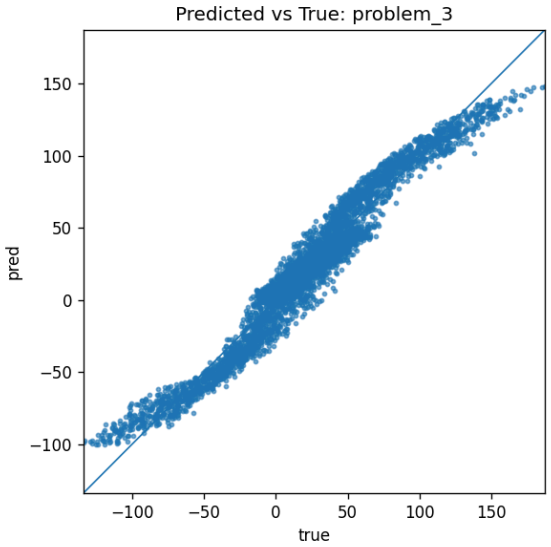


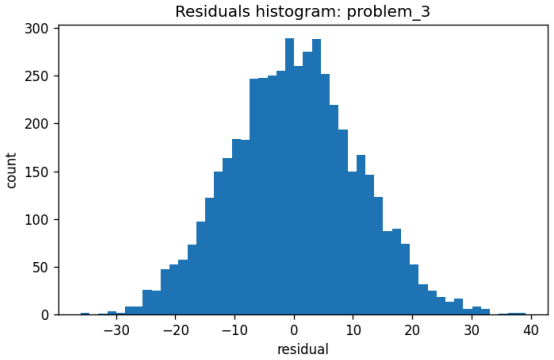
- **Problem 1:** Almost perfect diagonal; residuals look like floating-point noise.



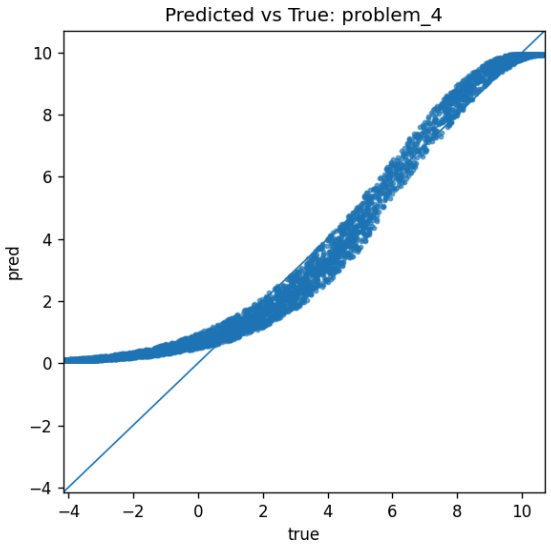


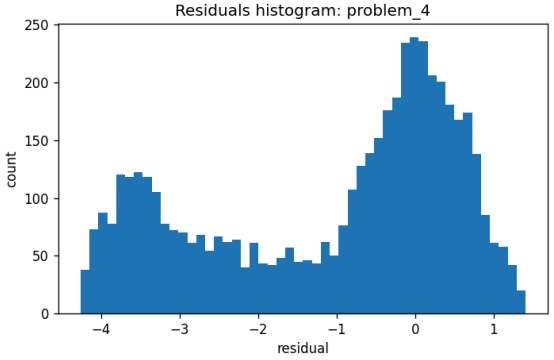
- **Problem 2:** Scatter is wide with many points at plot bounds; residuals show tall spikes at ±1e6 (expected due to clipping).



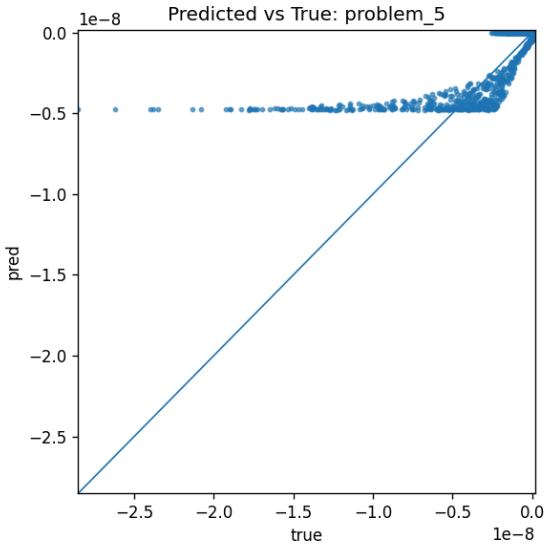


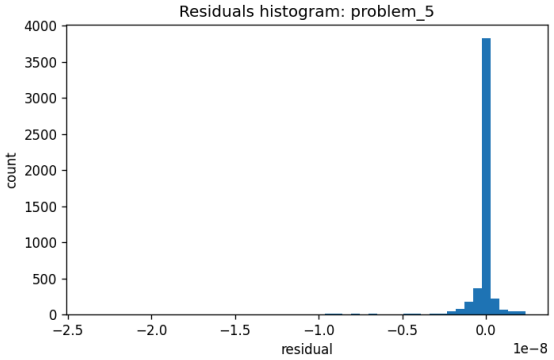
- **Problem 3:** Clear nonlinear structure; moderate bias near the extremes; residuals roughly Gaussian.



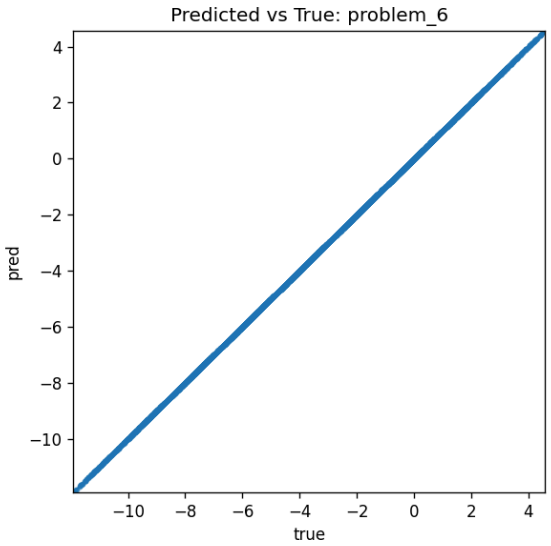


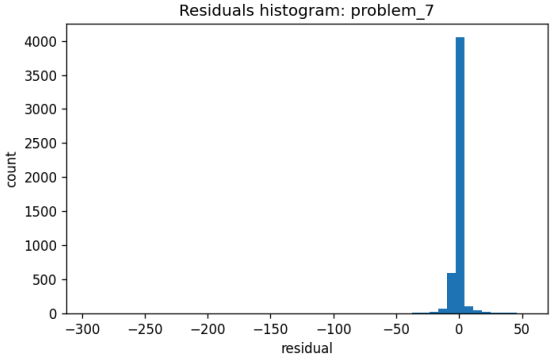
- **Problem 4:** S-shaped fit; slightly underestimates mid-range; asymmetric residuals with a left tail.



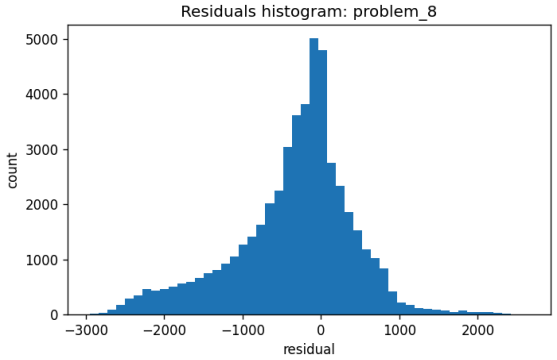


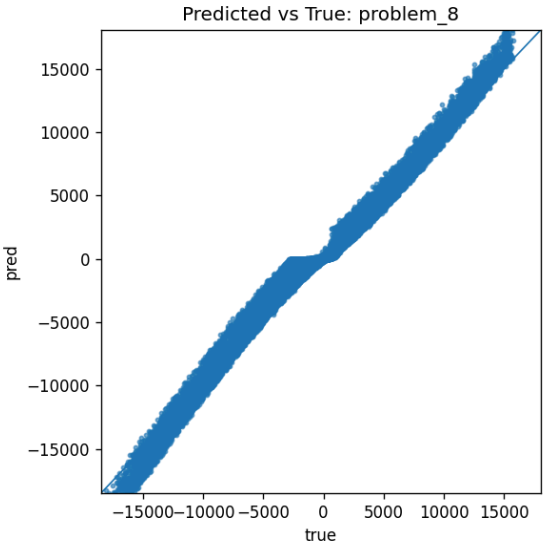
- **Problem 5 & 6:** Tight diagonals, very narrow residuals—high fidelity.





- **Problem 7:** Noticeable spread around the diagonal; broader residuals indicate both random error and some bias.





- **Problem 8:** Large variance and outliers; residuals dominated by a handful of hard points

Final notes :

- **Determinism:** Setting a fixed seed (when training) stabilizes runs, but GP still has inherent randomness.
- **Compactness vs. accuracy:** Depth limits and point mutation help avoid bloat; if accuracy stalls, relaxing depth and expanding the constant pool can help—at the cost of readability.