Report work done by Samuele Vanini s318684

All the work done for/during the course of Computational Intelligence 01URROV @ Politecnico di Torino is described in the following sections. Each section contains a title, content (where the requirements and techniques used are presented), code and the reviews to other colleagues. All the content has been produced by me with all the ispiration or functions cited when used. During the correction of the grammar or to check the tone of the reviews some LLM has been used, but all the content and information reported are completely developed without the use of such tools.

Lab 0

Repo link: https://github.com/SamueleVanini/CI2024_lab0

Content

Submitted following before the deadline. Joke submited:

Bender hangs up a flyer that says: "Bender's computer dating service. Discreet and discrete."

- "Put Your Head on My Shoulders", Futurama S2.E10

Reviews

- Review 1 https://github.com/maxfra01/CI2024_lab0/issues/1
- Review 2 https://github.com/XhoanaShkajoti/CI2024_lab0/issues/2

Lab 1

Repo link: https://github.com/SamueleVanini/CI2024_lab1

Content

For the laboratory I implemented the simulation annealing technique to solve the set-covering problem. I used a multiple mutation as tweek and a custom fitness called range_validity_fitness to incorporate the cost of the solution and a score on the phenotype.

Code

README.md

Usage

To run the laboratory:

- create a virtual environment with `python -m venv <ENV NAME>`
- activate the environment (on linux and mac os use the command `source <ENV NAME>/bin/activate`
- open the notebook `set-cover.ipynb` and run all the cells

Results

Instance	Universe size	Num sets	Density	Cost	Valid Solution
1	100	10	.2	-286	Yes
2	1,000	100	.2	-7091	Yes
3	10,000	1000	.2	-120035	Yes
4	100,000	10,000	.1	-2787985	Yes
5	100,000	10,000	.2	-3768514	Yes
6	100,000	10,000	.3	-4874291	Yes

set-cover.ipynb

```
import time
import numpy as np

from typing import Sequence, Callable
from itertools import accumulate
from matplotlib import pyplot as plt
from icecream import ic
from tqdm import tqdm
UNIVERSE_SIZE = 100_000
NUM_SETS = 10_000
# high density => each set is covering a lot of element
DENSITY = 0.3

rng = np.random.Generator(np.random.PCG64([UNIVERSE_SIZE, NUM_SETS, int(10_000 * DENSITY)]))
# DON'T EDIT THESE LINES!

SETS = np.random.random((NUM_SETS, UNIVERSE_SIZE)) < DENSITY
for s in range(UNIVERSE_SIZE):
    if not np.any(SETS[:, s]):
        SETS[np.random.randint(NUM_SETS), s] = True

# different sets has different costs (we need to minimize the total costs not the number of set
taken)
COSTS = np.pow(SETS.sum(axis=1), 1.1)

def cost(solution):
    """Returns the cost of a solution (to be minimized)"""
    return COSTS[solution].sum()

def valid(solution):
    """Checks wether solution is valid (ie. covers all universe)"""
    phenotype = np.logical_or.reduce(SETS[solution])
    return np.all(phenotype)

def plot_history(history: Sequence):
    plt.figure(figsize=(14, 8))
    plt.plot(
        range(len(history)),
        list(accumulate(history, max)),
        color="red",
    )
    _ = plt.scatter(range(len(history)), history, marker=".")

def single_mutation(solution: np.ndarray) -> np.ndarray:
    """Randomly drop or take one set in the universe
    Args:
        solution (np.ndarray): current solution (which sets are we taking)
    Returns:
        np.ndarray: new solution (starting solution +/- 1 set
    """
    new_sol = solution.copy()
    i = rng.integers(0, NUM_SETS)
    new_sol[i] = not new_sol[i]
    return new_sol

def multiple_mutation(solution: np.ndarray, probability: float = 0.01) -> np.ndarray:
    """Randomly drop or take a number of sets with a certain probability
    Args:
        solution (np.ndarray): current solution
        probability (float, optional): probability of each sets to be taken or droppped. Defaults
to 0.01.
    Returns:
```

```
        np.ndarray: new solution
    """
    new_sol = solution.copy()
    mask = rng.random(NUM_SETS) < probability
    if not np.any(mask):
        mask[rng.integers(NUM_SETS)] = True
    new_sol = np.logical_xor(solution, mask)
    return new_sol

def binary_validity_fitness(solution: np.ndarray):
    # remember that comparison is lexicografic with tupple (i.e. if a tupple is starting with 0
will be less than a valid one even
    # if the cost is less)
    return (valid(solution), -cost(solution))

def range_validity_fitness(solution: np.ndarray):
    phenotype = np.logical_or.reduce(SETS[solution])
    validity_score = UNIVERSE_SIZE - phenotype.sum()
    return (-validity_score, -cost(solution))

def sim_annealing(init_solution: np.ndarray,
                  steps: int,
                  restarts: int,
                  tweak: Callable[[np.ndarray], tuple],
                  fitness: Callable[[np.ndarray], tuple],
                  initial_prob: float,
                  temperature_dec_factor: float) -> tuple[list, np.ndarray]:

    best_solution = init_solution.copy()
    best_fitness = fitness(best_solution)
    current_solution = best_solution
    current_solution_fitness = best_fitness
    history = [ best_fitness[1] ]

    ic(fitness(best_solution))

    for restart in tqdm(range(restarts)):

        prob = initial_prob
        ic(current_solution_fitness)

        for step in range(steps // restarts):

            new_sol = tweak(current_solution, prob)
            f = fitness(new_sol)
            history.append(f[1])

            if f > current_solution_fitness:
                current_solution = new_sol
                current_solution_fitness = f

            prob *= temperature_dec_factor

        if current_solution_fitness > best_fitness:
            best_fitness = current_solution_fitness
            best_solution = current_solution.copy()

        # current_solution = rng.random(NUM_SETS) < 0.1
        # current_solution_fitness = fitness(current_solution)


    return history, best_solution, best_fitness

INITIAL_SOLUTION = np.full((NUM_SETS), False)
INITIAL_SOLUTION[rng.integers(NUM_SETS)] = True
```

```
fitness = range_validity_fitness
tweak = multiple_mutation
his, sol, fit = sim_annealing(INITIAL_SOLUTION, 5000, 5, tweak, fitness, initial_prob=0.01,
temperature_dec_factor=0.9)
ic(fitness(sol))
plot_history(his)
```

Reviews

Reviews done

- Review 1 https://github.com/LucaIanniello/CI2024_lab1/issues/2

  The proposed algorithm (Hill Climbing with multiple mutations) is well implemented, and the addition of the README file is a great touch. I do have a couple of suggestions that might enhance both the structure and performance of the solution:

  1. Reduce Redundancy: the algorithm is duplicated across different test instances, which could make debugging more time-consuming for both you and future readers. Consider wrapping the main logic into one or more functions, which you can then call from different blocks. This will improve maintainability and makes the code easier to modify of extend.

  2. Experiment with Fitness Functions: while your use of Hill Climbing with multiple mutations is solid, you could potentially improve the solution by experimenting with different types of fitness evaluations. For example, instead of simply classifying a solution as valid or invalid, it could be beneficial to assign a range of values to invalid solutions. This would help guide the algorithm more effectively, reducing the possibility of plateaus (mesas).

- Review 2 https://github.com/Roberto34/CI2024_lab1/issues/1

  The proposed algorithms are well implemented and clearly written. Starting from the mandatory set is a clever and effective approach. I have a few suggestions that could further enhance both the quality of the code and the overall repository:

  1. Dependency Management: Consider using a tool like Poetry or a simpler alternative like a requirements.txt file to manage dependencies. This would streamline the setup process, ensuring that the correct versions of libraries are used and making it easier for others to run your code.

  2. Optimize Mandatory Set Check: The check on the mandatory sets doesn't need to occur inside the tweak function, as this variable remains constant during execution. A more efficient approach would be to perform this check once before the algorithm starts.

  3. Clarify Algorithm Usage: It's unclear which of the two implemented algorithms (Random Mutation Hill Climber or RMHC with reset on best fitness plateau) is being used in the test instances.

  4. Balancing Exploration and Exploitation: The idea of increasing exploration through something like the temperature mechanism is a good solution. However, it might be worth considering a way to balance exploration with exploitation when a promising path is found, allowing the algorithm to exploit the current solution.

Reviews received

- Review 1 https://github.com/SamueleVanini/CI2024_lab1/issues/1

  After running and analyzing your code, I would like to say you have done a very good job of implementing the Simulated Annealing approach to solving the set problem. I explored several optimization strategies to try and improve the fitness:

  - Modified mutation functions and probabilities
  - Adjusted simulated annealing parameters
  - Tweaked fitness calculations
  - Applied minor code optimizations Impressively, none of these changes significantly improved performance, solution quality, or runtime. The balance between exploration/exploitation and validity/cost is well-tuned. I just have one suggestion, if you include the plots for all instances it would be great.

- Review 2 https://github.com/SamueleVanini/CI2024_lab1/issues/2

I think that this algorithm is a well implemented type of simulated annealing. It allows to obtain quite good results. The table that resumes them is very useful, but I did not understand why the cost values are negative, probabily it is a misprint due to the fact that fitness evaluation aims to minimize the cost considering it with the opposite sign. I suppose also that the author has decided not to report the steps number because it is a fixed one. I tried changing the parameters, but I did not find an advantageous combination, so I have nothing to recommend to improve it. The only aspect that, I believe, cans be built in a better way is adapt the total number of steps and restarts to the entity of the problem.

Lab 2

Repo link: https://github.com/SamueleVanini/CI2024_lab2

Content

The laboratory required a solution to the travelling salesman problem. To solve it I test various algorithms and techniques such as:

- Hill Climber with Scramble mutation
- Hill Climber with Inverse mutation
- Hill Climber with simulated annealing and scramble mutation
- Evolutionary Algorithm with stady state

Code

README.md

Usage

To run the laboratory:

- create a virtual environment with `python -m venv <ENV NAME>`
- activate the environment (on linux and mac os use the command `source <ENV NAME>/bin/activate`)
- open the notebook `tsp.ipynb` and run all the cells

Results

All the results proposed are obtained using the `italy.csv` data

Algorithm	Steps	Initial Solution	Cost	Calls to fitness
H.C. + Scramble mut	100_000	greedy	4431	100_000
H.C. + Scramble mut	100_000	sequential	6990	100_000
H.C. + Inverse mut	100_000	greedy	4431	100_000
H.C. + Inverse mut	100_000	sequential	7083	100_000
H.C. + S.A. + Scramble mut	100_000	greedy	4431	100_000
H.C. + S.A. + Scramble mut	100_000	sequential	7979	100_000

Algorithm	Generations	Initial Solution	Cost	Calls to fitness	Parent sel.	Population	Offsprings
EA Stady State	100_000	sequential	5149	2_000_030	uniform	30	20
EA Stady State	100_000	greedy	4431	2_000_030	uniform	30	20

tsp.ipynb

```
import logging
from itertools import combinations
import pandas as pd
import numpy as np
import geopy.distance
```

```
from icecream import ic

logging.basicConfig(level=logging.DEBUG)

from copy import copy
import logging
from typing import Callable
import pandas as pd
import numpy as np
import geopy.distance

from icecream import ic
from itertools import combinations
from functools import partial

from algorithms import BaseSolver, HillClimber, HillClimberSimulatedAnnealing, EASteadyState
from utilities import plot_history, counter, get_console_logger

CITIES = pd.read_csv('cities/italy.csv', header=None, names=['name', 'lat', 'lon'])
DIST_MATRIX = np.zeros((len(CITIES), len(CITIES)))
for c1, c2 in combinations(CITIES.itertuples(), 2):
    DIST_MATRIX[c1.Index, c2.Index] = DIST_MATRIX[c2.Index, c1.Index] = geopy.distance.geodesic(
        (c1.lat, c1.lon), (c2.lat, c2.lon)
    ).km
RNG = np.random.Generator(np.random.PCG64(len(CITIES)))
STEPS = 100_000
logger = get_console_logger(__name__, logging.INFO)
CITIES.head()

def total_tsp_cost(tsp):
    assert tsp[0] == tsp[-1]
    assert set(tsp) == set(range(len(CITIES)))
    tot_cost = 0
    for c1, c2 in zip(tsp, tsp[1:]):
        tot_cost += DIST_MATRIX[c1, c2]
    return tot_cost

def greedy_solution():
    visited = np.full(len(CITIES), False)
    dist = DIST_MATRIX.copy()
    city = 0
    visited[city] = True
    tsp = list()
    tsp.append(int(city))
    while not np.all(visited):
        dist[:, city] = np.inf
        closest = np.argmin(dist[city])
        logger.debug(
            f"step: {CITIES.at[city,'name']} -> {CITIES.at[closest,'name']}
({DIST_MATRIX[city,closest]:.2f}km)"
        )
        visited[closest] = True
        city = closest
        tsp.append(int(city))
    logger.debug(
        f"step: {CITIES.at[tsp[-1],'name']} -> {CITIES.at[tsp[0],'name']}
({DIST_MATRIX[tsp[-1],tsp[0]]:.2f}km)"
    )
    tsp.append(tsp[0])

    logger.info(f"result: Found a path of {len(tsp)-1} steps, total length
{total_tsp_cost(tsp):.2f}km")

    tsp.pop()
    return tsp
```

```python
def scrambe_mut(sol: np.ndarray, rnd: np.random.Generator):
    e1, e2 = rnd.choice(len(sol), size=2, replace=False)
    new_sol = sol.copy()
    new_sol[e2] = sol[e1]
    new_sol[e1] = sol[e2]
    return new_sol


def annealing_scramble_mut(sol: list[int], strenght: float, rnd: np.random.Generator):
    picked = False
    new_sol = copy(sol)
    while picked is False or (rnd.random() < strenght):
        picked = True
        e1, e2 = rnd.choice(len(sol), size=2, replace=False)
        swap = new_sol[e1]
        new_sol[e1] = new_sol[e2]
        new_sol[e2] = swap
    return new_sol


def new_annealing_scramble_mut(sol: list[int], strenght: float, rnd: np.random.Generator):
    new_sol = copy(sol)
    mask = rnd.random(len(sol)) < strenght
    idxs = np.nonzero(mask)
    if len(idxs) < 2:
        e1, e2 = rnd.choice(len(sol), size=2, replace=False)
        swap = new_sol[e1]
        new_sol[e1] = new_sol[e2]
        new_sol[e2] = swap
        return new_sol
    else:
        while len(idxs) > 1:
            e1, e2 = rnd.choice(len(idxs), size=2, replace=False)
            swap = new_sol[idxs[e1]]
            new_sol[idxs[e1]] = new_sol[idxs[e2]]
            new_sol[idxs[e2]] = swap
            idxs.pop(e1)
            idxs.pop(e2)
        return new_sol


def inverse_mut(sol: list[int], rnd: np.random.Generator):
    idx_e1, idx_e2 = rnd.choice(len(sol), size=2, replace=False)
    new_sol = copy(sol)
    if idx_e1 > idx_e2:
        e1 = new_sol.pop(idx_e1)
        new_sol.insert(idx_e2 + 1, e1)
    else:
        e2 = new_sol.pop(idx_e2)
        new_sol.insert(idx_e1 + 1, e2)
    return new_sol

@counter
def fitness(sol: list[int]):
    sol.append(sol[0])
    cost = total_tsp_cost(sol)
    sol.pop()
    return cost

init_sequential_sol = list(range(len(CITIES)))
init_greedy_solution = greedy_solution()

def run_experiment(
    algo: BaseSolver, steps: int, tweak: Callable, fitness: Callable, init_sol: list[int], *args,
```

```
    **kwargs
    ):
        algo_tweak = partial(tweak, rnd=RNG)
        ic(fitness(init_sol))
        fitness.num_calls = 0
        solver = algo(steps, algo_tweak, fitness, init_sol, *args, **kwargs)
        solver.solve()
        ic(solver.solution_fitness)
        ic(fitness.num_calls)
        plot_history(solver.history)

run_experiment(HillClimber, STEPS, scrambe_mut, fitness, init_greedy_solution)
run_experiment(HillClimber, STEPS, scrambe_mut, fitness, init_sequential_sol)
run_experiment(HillClimber, STEPS, inverse_mut, fitness, init_greedy_solution)
run_experiment(HillClimber, STEPS, inverse_mut, fitness, init_sequential_sol)
run_experiment(HillClimberSimulatedAnnealing, STEPS, new_annealing_scramble_mut, fitness,
init_greedy_solution, strenght=0.9)
run_experiment(HillClimberSimulatedAnnealing, STEPS, new_annealing_scramble_mut, fitness,
init_sequential_sol, strenght=0.9)
run_experiment(EAStadyState, STEPS, scrambe_mut, fitness, init_sequential_sol, population=30,
offspring=20, rnd=RNG)
run_experiment(EAStadyState, STEPS, scrambe_mut, fitness, init_greedy_solution, population=30,
offspring=20, rnd=RNG)
```

algorithms.py

```
from abc import ABC, abstractmethod
from dataclasses import dataclass
import itertools
import numpy as np
from typing import Any, Callable, Collection

from tqdm import tqdm
from icecream import ic

SEED = 42


class BaseSolver(ABC):

    @property
    @abstractmethod
    def history(self) -> list:
        pass

    @property
    @abstractmethod
    def solution(self):
        pass

    @abstractmethod
    def solve():
        pass


class HillClimber(BaseSolver):

    def __init__(self, steps: int, tweak: Callable, fitness: Callable, init_sol) -> None:
        self.steps = steps
        self.tweak = tweak
        self.fitness = fitness
        self._solution = init_sol
        self._history = []
```

```
        self.solution_fitness = self.fitness(self._solution)

    @property
    def history(self) -> list:
        return self._history

    @property
    def solution(self):
        return self._solution

    def solve(self):
        for step in tqdm(range(self.steps)):
            new_sol = self.tweak(self._solution)
            f = self.fitness(new_sol)
            self._history.append(f)

            if f < self.solution_fitness:
                self._solution = new_sol
                self.solution_fitness = f


class HillClimberSimulatedAnnealing(HillClimber):

    BUFFER_SIZE = 5
    ONE_OUT_FIVE = BUFFER_SIZE / 5

    def __init__(
        self,
        steps: int,
        tweak: Callable[[Any, float], Any],
        fitness: Callable[..., Any],
        init_sol,
        strenght: float = 0.5,
    ) -> None:
        super().__init__(steps, tweak, fitness, init_sol)
        self.strenght = strenght
        self.buffer = []

    def solve(self):
        for step in tqdm(range(self.steps)):
            new_sol = self.tweak(self._solution, self.strenght)
            f = self.fitness(new_sol)
            self._history.append(f)

            self.buffer.append(f < self.solution_fitness)
            self.buffer = self.buffer[-self.BUFFER_SIZE :]

            if sum(self.buffer) > self.ONE_OUT_FIVE:
                self.strenght *= 1.3
            elif sum(self.buffer) < self.ONE_OUT_FIVE:
                self.strenght /= 1.0001

            if f < self.solution_fitness:
                self._solution = new_sol
                self.solution_fitness = f


@dataclass
class Individual:
    genome: list[int]
    fitness: float = None


class EASteadyState(BaseSolver):
```

```
    def __init__(
        self,
        generations: int,
        tweak: Callable[[Any, float], Any],
        fitness: Callable[..., Any],
        init_genome=None,
        population: int | list[Individual] = 30,
        offspring: int = 20,
        rnd: np.random.Generator = None,
    ) -> None:
        self.generations = generations
        self.tweak = tweak
        self.fitness = fitness
        self._history = []
        if isinstance(population, int):
            self._pop_size = population
            self._population = [
                Individual(genome, fitness(genome)) for genome in itertools.repeat(init_genome,
population)
            ]
        if isinstance(population, list):
            self._pop_size = len(population)
            self._population = population
        self.n_offspring = offspring
        if rnd is None:
            self.rnd = np.random.default_rng(SEED)
        else:
            self.rnd = rnd

    @property
    def history(self) -> list:
        return self._history

    @property
    def solution(self):
        return self._population[0].genome

    @property
    def solution_fitness(self):
        return self._population[0].fitness

    def solve(self):
        for generation in tqdm(range(self.generations)):
            parrents = self.rnd.choice(self._population, size=self.n_offspring, replace=False)
            offsprings = []
            for parrent in parrents:
                genome = self.tweak(parrent.genome)
                fit = self.fitness(genome)
                self._history.append(fit)
                offsprings.append(Individual(genome, fit))
            self._population.extend(offsprings)
            self._population.sort(key=lambda i: i.fitness)
            self._population = self._population[: self._pop_size]
```

utilities.py

```
from itertools import accumulate
import time
import logging
import icecream as ic

from matplotlib import pyplot as plt
from functools import wraps, partial
```

```
plt.set_loglevel("error")


def execution_time(
    _func=None,
    *,
    logging_level=logging.DEBUG,
):

    if _func is None:
        return partial(execution_time, logging_level=logging_level)

    logname = _func.__module__
    log = logging.getLogger(logname)
    logmsg = _func.__name__

    def decorator(func):
        @wraps(func)
        def wrapper_timer(*args, **kwargs):
            log.log(logging_level, logmsg)
            start_time = time.perf_counter()
            value = func(*args, **kwargs)
            end_time = time.perf_counter()
            run_time = end_time - start_time
            ic(f"Finished {func.__name__}() in {run_time:.4f} secs")
            return value

        return wrapper_timer

    return decorator


def counter(_func):

    @wraps(_func)
    def counter_wrapper(*args, **kwargs):
        counter_wrapper.num_calls += 1
        return _func(*args, **kwargs)

    counter_wrapper.num_calls = 0
    return counter_wrapper


def plot_history(history):
    plt.figure(figsize=(14, 8))
    plt.plot(
        range(len(history)),
        list(accumulate(history, min)),
        color="red",
    )
    plt.scatter(range(len(history)), history, marker=".")


def get_console_logger(name: str, level: int):
    logger = logging.getLogger(name)
    logger.setLevel(level=level)
    console_handler = logging.StreamHandler()
    console_handler.setLevel(level=level)
    formatter = logging.Formatter("{levelname} - {message}", style="{")
    console_handler.setFormatter(formatter)
    logger.addHandler(console_handler)
    return logger
```

Reviews

Reviews done

- Review 1 https://github.com/LorenzoFormentin/CI2024_lab2/issues/1

  The proposed algorithms are well implemented and clearly written. Your initial solution incorporates an excellent amount of genetic material, and the combination of crossover and population management strategies you've applied is very interesting! It's unclear why PMX was removed in the second algorithm—possibly due to the added complexity affecting execution time? Regardless, this approach remains effective overall. Here are a few suggestions that could further enhance the quality of the code and repository:

  1. Dependency Management: To make it easier for others to run your code, consider using a tool like Poetry or a simpler alternative, such as a requirements.txt file, to manage dependencies. This would streamline the setup process and ensure that the correct versions of libraries are used.
  2. Reduce Redundant Sort/Search Operations: Some operations can be optimized, particularly in generational population management. In this snippet, the second line can be removed, as the population is already sorted, so you can simply select the first element of the list. This change becomes more impactful as the population size grows:

```
population = sorted(new_population + population, key=tsp_cost)[:POPULATION_SIZE]
current_best = min(population, key=tsp_cost)
```

  3. Another optimization would be to store the fitness score for the current best individual. Currently, you recalculate the fitness of the current best individual in every generation, and if it's better than the global best, you calculate it again. Storing it in a variable would help here:

```
if tsp_cost(current_best) < best_fitness:
    best_tour = current_best
    best_fitness = tsp_cost(current_best)
```

- Review 2 https://github.com/lorenzo-ll/CI2024_lab2/issues/1

  The proposed algorithm shows some improvements over the greedy solution presented in class. The use of "smart" individuals in the initial population is well thought out and should provide ample genetic diversity, especially given the large population size. Here are a few suggestions that could further enhance the quality of both the code and the repository:

  1. Dependency Management: To make it easier for others to run your code, consider using a tool like Poetry or a simpler alternative, such as a requirements.txt file, for managing dependencies. This would streamline the setup process and ensure the correct versions of libraries are used.
  2. Break Down the Code: Currently, the algorithm is contained in a single large block with several nested functions. Although inner functions are sometimes used to limit scope, here the nested structure reduces readability due to indentation depth and length. Consider breaking down the code into separate functions for improved clarity.
  3. Avoid Duplicate Parent Selection: Sampling two parents with replacement can result in selecting the same individual twice, which creates clones rather than introducing new genetic material. This may be counterproductive, especially given the generational population management approach used. It might be worth adjusting the sampling to avoid this.
  4. Include the Dataset: Since the dataset is relatively small, it could be convenient to include it directly in the repository. This would simplify testing for other users.
  5. Trait Retention in Optimization: Applying pseudo-deterministic optimization on offspring may disrupt the inherited traits from parents, potentially reducing genetic diversity. Consider reviewing this approach, as it could impact the effectiveness of the algorithm over generations.

Reviews received

No reviews were left.

Lab 3

Repo link: https://github.com/SamueleVanini/CI2024_lab3

Content

The laboratory required to find a solution to "The 15 puzzle" problem with some variation on the dimension of the board. To solve it, I implemented a little custom priority queue to be used with the A* algorithm and the manhattan distance as admissible heuristic.

Code

README.md

Usage

To run the laboratory:

- create a virtual environment with `python -m venv <ENV NAME>`
- activate the environment (on linux and mac os use the command `source <ENV NAME>/bin/activate`)
- run the command `python main.py`

Results

All the results proposed are obtained using the `A*` algorithm:

Puzzle size	Distance	Randomized steps	Path Cost	State explored	Execution time
3 x 3	Manhattan	10.000	24	6501	0.1593s
4 x 4	Manhattan	5.000	34	52.247	0.4772s
5 x 5	Manhattan	200	56	8.412.858	137.9832
6 x 6	Manhattan	200	NONE	NONE	NONE
7 x 7	Manhattan	200	NONE	NONE	NONE

External links:

Some ideas/code has been taken from:

- Squillero's code
- Check if puzzle is solvable

main.py

```
from functools import cache
import logging
from typing import Callable, NamedTuple
import numpy as np
import random

from random import choice
from tqdm.auto import tqdm
from itertools import chain
from utilities import execution_time, counter

from data_structure import SvanniPriorityQueue, Item, State
from exp.squillero_code import squillero_initial

PUZZLE_DIM = 6
# RANDOMIZE_STEPS = 100_000
```

```
RANDOMIZE_STEPS = 200
# added only for reproducibility
random.seed(42)


def configure_logging(console_level=logging.INFO, utility_level=logging.INFO):
    logger = logging.getLogger(__name__)
    logger.setLevel(console_level)
    utility_logger = logging.getLogger("utilities")
    utility_logger.setLevel(utility_level)
    console_handler = logging.StreamHandler()
    console_handler.setLevel(console_level)
    formatter = logging.Formatter("%(asctime)s - %(levelname)s - %(name)s - %(message)s",
style="%")
    console_handler.setFormatter(formatter)
    logger.addHandler(console_handler)
    return logger


class Action(NamedTuple):
    pos1: tuple[int]
    pos2: tuple[int]


GOAL = State(np.array(list(range(1, PUZZLE_DIM**2)) + [0]).reshape((PUZZLE_DIM, PUZZLE_DIM)))


def available_actions(state: State) -> list[Action]:
    x, y = [int(_[0]) for _ in np.where(state.value == 0)]
    actions = list()
    if x > 0:
        actions.append(Action((x, y), (x - 1, y)))
    if x < PUZZLE_DIM - 1:
        actions.append(Action((x, y), (x + 1, y)))
    if y > 0:
        actions.append(Action((x, y), (x, y - 1)))
    if y < PUZZLE_DIM - 1:
        actions.append(Action((x, y), (x, y + 1)))
    return actions


@counter
def do_action(state: State, action: Action) -> State:
    new_state = State(state)
    new_state.value[action.pos1], new_state.value[action.pos2] = (
        new_state.value[action.pos2],
        new_state.value[action.pos1],
    )
    new_state.lock_data()
    return new_state


def get_init_state():
    state = State(GOAL)
    for r in tqdm(range(RANDOMIZE_STEPS), desc="Randomizing"):
        state = do_action(state, choice(available_actions(state)))
    init_state = State(state)
    init_state.lock_data()
    return init_state


@execution_time
def a_star(
    state: State, state_cost: dict[int, int], fronteer: SvanniPriorityQueue, distance:
Callable[[State], int]
```

```
) -> State:

    # visited_state must contain only the the cost needed to get to that state
    state_cost[state] = 0

    while not check_goal(state):

        valid_actions = available_actions(state)

        for a in valid_actions:

            new_state = do_action(state, a)

            new_state_cost = state_cost[state] + 1

            if new_state not in fronteer and new_state not in state_cost:
                # create the cost cost (father + 1)
                state_cost[new_state] = new_state_cost
                p = state_cost[new_state] + distance(new_state)
                fronteer.put(Item(p, new_state), block=False)

            elif new_state in fronteer and state_cost[new_state] > new_state_cost:
                # update the fronteer (the state is in the fronteer but we the cost of an old
path, update it to push algo to convergence faster)
                p = state_cost[new_state] + distance(new_state)
                # fronteer.replace(Item(p, new_state), Item(new_state_cost, new_state))

                state_cost[new_state] = new_state_cost

        state = fronteer.get(block=False).data

    return state


@cache
def missing_tiles(state: State) -> int:
    needed_steps = 0
    correct_value = 1
    iter = chain.from_iterable(state.value)
    max_value = PUZZLE_DIM**2
    for value in iter:
        if value and value != (correct_value % max_value):
            needed_steps += 1
        correct_value += 1
    return needed_steps


@cache
def manhattan_distance(state: State) -> int:
    tot_sum = 0
    correct_value = 1
    matrix = state.value
    for i in range(PUZZLE_DIM):
        for j in range(PUZZLE_DIM):
            if matrix[i, j] != correct_value and matrix[i, j]:
                cx = (matrix[i, j] - 1) / PUZZLE_DIM
                cy = (matrix[i, j] - 1) % PUZZLE_DIM
                tot_sum += abs(cx - i) + abs(cy - j)
            correct_value += 1
    return tot_sum


@cache
def check_goal(state: State) -> bool:
    correct_value = 1
```

```
    iter = chain.from_iterable(state.value)
    max_value = PUZZLE_DIM**2
    for value in iter:
        if value != (correct_value % max_value):
            return False
        correct_value += 1
    return True


# It's a bit strange but with the other version the code runs faster, probabily due to the need
of array creation in the middle
# of the operation (malloc is still a heavy operation)

# def check_goal(state: State) -> bool:
#     return np.all(state.current_value == GOAL.current_value)

# def distance(state: State) -> int:
#     return np.sum((state.current_value != GOAL.current_value) & (state.current_value > 0))

N = PUZZLE_DIM


def getInvCount(arr):
    arr1 = []
    for y in arr:
        for x in y:
            arr1.append(x)
    arr = arr1
    inv_count = 0
    for i in range(N * N - 1):
        for j in range(i + 1, N * N):
            # count pairs(arr[i], arr[j]) such that
            # i < j and arr[i] > arr[j]
            if arr[j] and arr[i] and arr[i] > arr[j]:
                inv_count += 1

    return inv_count


# find Position of blank from bottom
def findXPosition(puzzle):
    # start from bottom-right corner of matrix
    for i in range(N - 1, -1, -1):
        for j in range(N - 1, -1, -1):
            if puzzle[i][j] == 0:
                return N - i


# This function returns true if given
# instance of N*N - 1 puzzle is solvable
def isSolvable(puzzle):
    # Count inversions in given puzzle
    invCount = getInvCount(puzzle)

    # If grid is odd, return true if inversion
    # count is even.
    if N & 1:
        return ~(invCount & 1)

    else:  # grid is even
        pos = findXPosition(puzzle)
        if pos & 1:
            return ~(invCount & 1)
        else:
            return invCount & 1
```

```
if __name__ == "__main__":
    logger = configure_logging(logging.INFO, logging.INFO)

    init_state = get_init_state()

    logger.info(init_state.value)
    do_action.num_calls = 0
    state_cost = dict[int, int]()
    fronteer = SvanniPriorityQueue()
    logger.info("Solver starting with %s", manhattan_distance.__qualname__)
    final_state = a_star(init_state, state_cost, fronteer, manhattan_distance)
    logger.info("state=%s", final_state.value)
    logger.info("number of calls=%d", do_action.num_calls)
    logger.info("cost=%d", state_cost[final_state])

    # state_cost.clear()
    # fronteer = SvanniPriorityQueue()

    # logger.info("Solver starting with %s", missing_tiles.__qualname__)
    # do_action.num_calls = 0
    # final_state = a_star(init_state, state_cost, fronteer, missing_tiles)
    # logger.info("state=%s", final_state.value)
    # logger.info("number of calls=%d", do_action.num_calls)
    # logger.info("cost=%d", state_cost[final_state])
```

utilities.py

```
from itertools import accumulate
import time
import logging
import icecream as ic

from matplotlib import pyplot as plt
from functools import wraps, partial

plt.set_loglevel("error")


def execution_time(_func=None, *, logging_level=logging.INFO, handler=logging.StreamHandler()):

    if _func is None:
        return partial(execution_time, logging_level=logging_level)

    # log_module = _func.__module__
    log_module = __name__
    log_func_name = _func.__name__
    log = logging.getLogger(log_module)
    formatter = logging.Formatter("%(asctime)s - %(levelname)s - %(name)s - %(message)s",
style="%")
    handler.setFormatter(formatter)
    log.addHandler(handler)

    @wraps(_func)
    def wrapper_execution_time(*args, **kwargs):
        start_time = time.perf_counter()
        value = _func(*args, **kwargs)
        end_time = time.perf_counter()
        run_time = end_time - start_time
        log.log(logging_level, "Finished %s in %4f secs", log_func_name, run_time)
        return value
```

```
    return wrapper_execution_time


def counter(_func):

    @wraps(_func)
    def counter_wrapper(*args, **kwargs):
        counter_wrapper.num_calls += 1
        return _func(*args, **kwargs)

    counter_wrapper.num_calls = 0
    return counter_wrapper


def plot_history(history):
    plt.figure(figsize=(14, 8))
    plt.plot(
        range(len(history)),
        list(accumulate(history, min)),
        color="red",
    )
    plt.scatter(range(len(history)), history, marker=".")


def get_console_logger(name: str, level: int):
    logger = logging.getLogger(name)
    logger.setLevel(level=level)
    console_handler = logging.StreamHandler()
    console_handler.setLevel(level=level)
    formatter = logging.Formatter("{levelname} - {message}", style="{")
    console_handler.setFormatter(formatter)
    logger.addHandler(console_handler)
    return logger
```

data_structure.py

```
from dataclasses import dataclass, field
from queue import PriorityQueue
from typing import Self

import numpy as np


class State:

    def __init__(self, value: np.ndarray | Self) -> None:
        self.value = value.copy()

    def __eq__(self, other: Self) -> bool:
        return other._hash == self._hash

    def __hash__(self) -> int:
        return self._hash

    def lock_data(self):
        self.value.setflags(write=False)
        self._hash = hash(self.value.tobytes())

    def copy(self):
        return self.value.copy()


@dataclass(order=True)
```

```python
class Item:
    priority: int
    data: State = field(compare=False)

    def __eq__(self, other: Self) -> bool:
        return self.data == other.data


class SvanniPriorityQueue(PriorityQueue):

    def __init__(self, maxsize: int = 0) -> None:
        super().__init__(maxsize)
        self._item_set = set()

    def replace(self, old_item: Item, new_item: Item):
        self.queue.remove(old_item)
        self._item_set.discard(old_item.data)
        self.put(new_item)

    # NOTE: even if the word item is used, we refer to the data contained in the class
Item(priority, data)
    # the name "item" is used to be similar with PriorityQueue interface.
    def __contains__(self, item: State):
        return item in self._item_set

    def _put(self, item: Item):
        assert item.data not in self, f"Trying to add a duplicated item in the priority queue"
        self._item_set.add(item.data)
        super()._put(item)

    def _get(self):
        item = super()._get()
        self._item_set.remove(item.data)
        return item
```

Reviews

Reviews done

- Review 1 https://github.com/MarcoDelCore/CI2024_lab3/issues/2

  The proposed solution for the n-puzzle problem using A* with a combined heuristic is incredibly well done! The integration of Linear Conflict with the traditional Manhattan distance is an excellent choice, as it maintains both admissibility and consistency, enhancing the algorithm's effectiveness. The code is well-structured and consistently written, demonstrating a clear and thoughtful style. Additionally, the README file is comprehensive, including all the relevant information along with a clear explanation of the combined heuristic. Good job!

- Review 2 https://github.com/GiovanniOrani/CI2024_lab3/issues/1

  The proposed algorithm, A* with the Manhattan heuristic, is an excellent choice, as it ensures both admissibility and consistency. Both the code and the README file are well-structured and easy to follow. Here are a few suggestions that could further enhance the quality of the code and the solution:

  1. Stick to NumPy Arrays: Avoid frequently switching or casting between tuples and arrays, as this triggers new memory allocations, which can slow down execution and increase memory usage. Sticking to a consistent data type, such as NumPy arrays, can improve efficiency. Am example would be to wrap the state in a class that implements **eq** and **hash** methods, while also making the array immutable after its creation using a command like ARRAY_NAME.setflags(write=False).

  2. Expand the Heuristic: You could optimize the number of computations by enhancing the heuristic function. For example, consider incorporating the number of tiles that are in their correct row or column but are blocked by others, requiring additional moves to resolve. This idea, as implemented by Marco Del Core, can further improve the algorithm's performance.

  In general, the laboratory is well done, good job.

Reviews received

- Review 1 https://github.com/SamueleVanini/CI2024_lab3/issues/1

  Your code is well structured and commented. I appreciated the different files you created to separate the different parts of the code. Your A* algorithm uses a custom PriorityQueue as frontier, and the Manhattan distance as heuristic. As can be seen from your results, the algorithm works well for puzzles up to 5x5, but it is not able to solve the 6x6 and 7x7 puzzles. This is due to a high number of states that need to be explored, and also on the chosen heuristic.

  To improve your code, you could try to implement a more complex heuristic: for example, consider adding a linear conflict heuristic (Adds penalties for pairs of tiles that are in their goal row or column but reversed). Another possibility is to consider different heuristics, each one with its own weight, and try to find the best combination of weights. The issue with the high number of states explored could be mitigated by implementing some sort of pruning, for example by avoiding symmetrical states. Additionally your custom PriorityQueue is suitable for the problem, but I would suggest to use a built-in data structure, such a binary heap, to improve the performance of the algorithm, reducing the time complexity of the operations.

  Overall, your A* implementation is well done, and with some improvements it could be able to solve larger puzzles.

- Review 2 https://github.com/SamueleVanini/CI2024_lab3/issues/2

  Hi Samuele,

  Your A* implementation for the last algorithm is really well done and shows a solid design with nice use of custom data structures like SvanniPriorityQueue. Using heuristics like manhattan_distance is a good choice, but it gets called many times, which slows things down, especially for bigger puzzles. For example, the 6x6 puzzle struggled a lot, and optimizing the heuristic maybe by caching or precomputing goal position could make it run faster. Also, making sure the queue updates more efficiently when priorities change would help the performance.

  Even with these issues, the algorithm is solid and works well for smaller puzzles. It’s written in a way that makes it easier to improve, and with some tweaks, it could handle larger puzzles much better.

  The randomization to create the starting state works fine, and it’s great that you included the isSolvable check so the algorithm doesn’t waste time on impossible puzzles.

  One suggestion: instead of just showing the final path or result, it would be more clear if the puzzle states were printed step by step after each move. That would make it easier to follow the algorithm and see what it’s doing. But i also appriciate the effort that you put in your readme and that made it easier to understand what i was going to see in the algorithm

  Overall, this is a very good implementation. With a few improvements, it could be a strong solution even for bigger puzzles 😃

- Review 3 https://github.com/SamueleVanini/CI2024_lab3/issues/3

  Your code is well structured and, with the provided comments, it is easy to understand. I still prefer python notebooks, because I think they are more easy to “read”, but that is because I’m more used to them, than pure python files. Anyway the file type was not a problem and your choice is perfectly in line with the laboratory assignments.

  When it comes to the actual implementation, unfortunately I am not able to suggest possible improvements, since my performances were worse than yours (I stopped the code when try to solving the 5x5 puzzle, because it was too slow). Analysing the problem and the performance you reported, it is clear that the main issue is the number of state evaluated before finding the solution, especially when puzzle size grows.

  A possible solution to the problem, that my reviewers have suggested to me, and that showed to be very good in their implementations, is to combine different heuristics, rather than using only the Manhattan Distance. A very popular and simple addition is to give a penalty to states with tiles that are swapped along the correct row or column. I have not tied it yet, but it may allow the algorithm to run faster, evaluating first states that requires less moves to reach the goal configuration.

  Anyway, good job with the laboratory, this was not an easy task, and being able to solve efficiently the lower dimension puzzles is still very good, to me.

Project on Symbolic Regression

Repo link: https://github.com/SamueleVanini/CI2024_project-work/tree/main

Content

The project required the implementation of various techniques to solve symbolic regression problems. Specifically, data points from eight different functions were provided as benchmarks for our solutions. To address this challenge, I developed a small library that leverages tree-based data structures to model mathematical functions. Each tree is a subclass of `list`, where each node is either a `Primitive` or `Terminal` type, generated in a depth-first order.

For population management, I experimented with both generational and steady-state strategies, implemented in the code as `mu_comma_lambda` and `mu_plus_lambda`. The operators tested include:

- Subtree mutation
- Point mutation
- Hoist mutation
- One-point crossover

Every time an offspring is generated, a probability dictates whether crossover is performed or one of the mutation operators, predetermined before the run, is applied instead.

To preserve traits during crossover, I also implemented "Automatically Defined Functions" (ADFs) as proposed by John Koza (1992). For the ADF technique, I developed custom crossover and mutation operators: one variant applies crossover and/or mutation to each defined function within the individual, whereas the other selects a single subtree for applying these operators. Despite these tailored approaches, the ADF method did not perform as well as expected, even after extensive hyperparameter tuning.

Additionally, to control bloat, I introduced a static check on every newly created offspring. If an offspring exceeds a predefined height threshold, one of the parents is selected as the new individual instead of the oversized offspring.

Finally, to guide the algorithm more effectively and promote the formation of a fitness hole, I implemented tournament selection in addiction to uniform selection for choosing parents during crossover or individual to mutate.

Results obtained

Problem	MSE	Generation	Population management	Mu	Lambda	Prob. xover	Prob. mutation	Max height	Min height	Max possible height
1	3.92119e-32	30	mu_comma_lambda	200	600	0.95	0.05	7	2	10
2	2.60167e+15	50	mu_plus_lambda	500	300	0.95	0.05	6	3	10
3	4307.37	200	mu_plus_lambda	500	500	0.8	0.2	9	4	10
4	6.93769	200	mu_plus_lambda	500	500	0.8	0.2	9	4	10
5	5.57281e-16	30	mu_comma_lambda	200	600	0.95	0.05	7	2	10
6	5.2765	30	mu_comma_lambda	200	600	0.95	0.05	7	2	10
7	30119.3	200	mu_plus_lambda	500	500	0.8	0.2	9	4	10
8	1.51013e+08	30	mu_comma_lambda	200	600	0.95	0.05	7	2	10

Code

src.toolbox.gp_algorithms.py

```
import copy
import operator
from random import Random

import numpy as np
from tqdm import tqdm
from src.toolbox.gp_bloat_control import ind_limit
from src.toolbox.gp_objects import (
```

```
    ADFIndividual,
    Compiler,
    Primitive,
    SimpleCompiler,
    Individual,
    PrimitiveSet,
    Terminal,
    Tree,
)
from src.toolbox.gp_expr_op import xover, point_mut, hoist_mut, subtree_mut
from src.gp_random import PyRndGenerator
from src.toolbox.gp_statistics import Statistics


xover = ind_limit(xover, measure=operator.attrgetter("height"), max_limit=10)
point_mut = ind_limit(point_mut, measure=operator.attrgetter("height"), max_limit=10)
# point_mut = ind_limit(hoist_mut, measure=operator.attrgetter("height"), max_limit=10)


def mu_comma_lambda(
    population: list[Individual],
    ngeneration: int,
    mu: int,
    gen_of,
    fitness_func,
    stat: Statistics,
) -> tuple[list[Individual], list[tuple[int, dict[str, float]]], Individual]:
    stats_history = []
    best = population[0]

    for ngen in tqdm(range(1, ngeneration + 1), desc="algo"):

        offsprings = gen_of(population)  # type: ignore

        new_pop = []
        for of in offsprings:
            fit = fitness_func(of)
            if fit != -1:
                of.fitness = fit
                new_pop.append(of)
        new_pop.sort(key=lambda of: of.fitness)

        if len(new_pop) < mu:
            for idx, _ in enumerate(range(mu - len(new_pop))):
                new_pop.append(population[idx])

        population[:] = new_pop[:mu]

        if population[0].fitness < best.fitness:
            best = copy.deepcopy(population[0])

        stats = stat.compute_stats(population)
        stats_history.append((ngen, stats))

    return population, stats_history, best


def mu_plus_lambda(
    population: list[Individual],
    ngeneration: int,
    mu: int,
    gen_off_func,
    fitness_func,
    stat: Statistics,
) -> tuple[list[Individual], list[tuple[int, dict[str, float]]], Individual]:
```

```
    stats_history = []
    best = population[0]

    for ngen in tqdm(range(1, ngeneration + 1), desc="algo"):

        offsprings = gen_off_func(population)  # type: ignore

        for of in offsprings:
            fit = fitness_func(of)
            if fit != -1:
                of.fitness = fit
            else:
                of.fitness = float("inf")

        population.extend(offsprings)
        population.sort(key=lambda ind: ind.fitness)

        population = population[:mu]

        if population[0].fitness < best.fitness:
            best = copy.deepcopy(population[0])
            print(f"\n {best.fitness}")

        stats = stat.compute_stats(population)
        stats_history.append((ngen, stats))

    return population, stats_history, best


def gen_offsprings(
    population: list[Tree], lambd: int, mprob: float, cprob: float, pset: PrimitiveSet,
selection_func
) -> list[Tree]:
    offsprings = []
    noffsprings = 0
    gen: Random = PyRndGenerator().gen
    while noffsprings < lambd:
        op_choice = gen.random()
        if op_choice < cprob:
            ind1, ind2 = selection_func(population, nind=2)
            of1, of2 = xover(ind1, ind2)
            if ind_has_all_var(of1, pset):
                offsprings.append(of1)
                noffsprings += 1
            if ind_has_all_var(of2, pset):
                offsprings.append(of2)
                noffsprings += 1
        else:
            ind1 = selection_func(population, nind=1)[0]
            of = point_mut(ind1, pset)[0]
            if ind_has_all_var(of, pset):
                offsprings.append(of)
                noffsprings += 1
    return offsprings


def tournament_selection(population, tournament_size: int, nind: int):
    inds = []
    gen: Random = PyRndGenerator().gen
    for _ in range(nind):
        selected = gen.choices(population, k=tournament_size)
        inds.append(min(selected, key=lambda ind: ind.fitness))
    return inds
```

```
def uniform_selection(population, nind: int):
    gen: Random = PyRndGenerator().gen
    return gen.choices(population, k=nind)


def ind_has_all_var(ind, pset):
    vars = set(pset.arguments)
    var_found = set()
    for node in ind:
        if isinstance(node, Terminal) and node.is_symbolic:
            var_found.add(node.name)
    return vars == var_found


def custom_adf_gen_offspring(
    population: list[ADFIndividual], mprob: float, cprob: float, lambd: int, pset:
list[PrimitiveSet]
):

    noffsprings = 0
    offsprings = []
    gen: Random = PyRndGenerator().gen

    while noffsprings < lambd:

        if gen.random() < cprob:

            par1, par2 = gen.sample(population, 2)

            of1 = copy.deepcopy(par1)
            of2 = copy.deepcopy(par2)

            prim_idx = gen.randrange(1, len(par1[0]))
            prim_name: str = par1[0][prim_idx].name

            if prim_name.startswith("ADF"):
                adf_idx = int(prim_name.split("_")[1])
                of1[adf_idx] = par2[adf_idx]
                of2[adf_idx] = par1[adf_idx]
                offsprings.append(of1)
                offsprings.append(of2)
                noffsprings += 2

        else:
            ind = gen.choice(population)
            of = copy.deepcopy(ind)
            prim_idx = gen.randrange(0, len(ind[0]))
            prim: Primitive = ind[0][prim_idx]
            prim_name: str = prim.name
            if prim_name.startswith("ADF"):
                adf_idx = int(prim_name.split("_")[1])
                of[adf_idx] = point_mut(ind[adf_idx], pset[adf_idx])[0]
                offsprings.append(of)
            else:
                narity = prim.arity
                if narity != 0:
                    new_prim = gen.choice(pset[0].prim_dict[narity])
                    of[0][prim_idx] = new_prim
                else:
                    new_term = gen.choice(pset[0].terminals)
                    of[0][prim_idx] = new_term

                offsprings.append(of)
```

```
            noffsprings += 1

    return offsprings


def adf_gen_offspring(
    population: list[ADFIndividual], lambd: int, mprob: float, cprob: float, pset:
list[PrimitiveSet]
):
    offsprings = []
    noffsprings = 0
    gen: Random = PyRndGenerator().gen
    while noffsprings < lambd:

        par1, par2 = gen.sample(population, 2)
        of1 = []
        of2 = []
        crossed = False
        for tree1, tree2 in zip(par1, par2):
            if gen.random() < cprob:
                tof1, tof2 = xover(tree1, tree2)
                of1.append(tof1)
                of2.append(tof2)
                crossed = True
        if crossed:
            offsprings.append(ADFIndividual(of1))
            offsprings.append(ADFIndividual(of2))
            noffsprings += 2

        for ind in population:
            mutated = False
            of = []
            for idx, tree in enumerate(ind):
                if gen.random() < cprob:
                    of.append(point_mut(tree, pset[idx])[0])
                    mutated = True
            if mutated:
                offsprings.append(ADFIndividual(of))
                noffsprings += 1
    return offsprings


def mse(ind: Tree, compiler: Compiler, x, y):
    y_pred = []
    func, code = compiler.compile(ind)
    for xvar in x:
        try:
            y_pred.append(func(*xvar))
        except:
            return -1
    return 100 * np.square(y - y_pred).sum() / len(y)
```

src.toolbox.gp_bloat_control.py

```
from functools import wraps
from random import Random
from src.gp_random import PyRndGenerator


def ind_limit(func, measure, max_limit):

    gen: Random = PyRndGenerator().gen
```

```
    @wraps(func)
    def wrapper(*args, **kwargs):
        offsprings = list(func(*args, **kwargs))
        for idx, of in enumerate(offsprings):
            if measure(of) > max_limit:
                parrent_choosen = gen.choice(args)
                offsprings[idx] = parrent_choosen
        return offsprings

    return wrapper
```

src.toolbox.gp_expt_op.py

```
import copy
from random import Random
from typing import Callable, Iterable
from src.gp_random import PyRndGenerator
from src.toolbox.gp_objects import Primitive, PrimitiveSet, Terminal, Tree


###############
# GENERATION #
###############


def gen_half_and_half(pset: PrimitiveSet, min_height: int, max_height: int) -> Iterable[Primitive
| Terminal]:
    gen: Random = PyRndGenerator().gen
    generation_func = gen.choice([gen_full_expr, gen_grow_expr])
    return Tree(generation_func(pset, min_height, max_height))


def gen_full_expr(pset: PrimitiveSet, min_height: int, max_height: int) -> Iterable[Primitive |
Terminal]:
    condition = lambda depth, height: depth == height
    return _gen_expr(pset, min_height, max_height, condition)


def gen_grow_expr(pset: PrimitiveSet, min_height: int, max_height: int) -> Iterable[Primitive |
Terminal]:
    gen = PyRndGenerator().gen
    condition = lambda depth, height: (depth == height) or (depth >= min_height and gen.random()
< pset.terminal_ratio)
    return _gen_expr(pset, min_height, max_height, condition)


def _gen_expr(
    pset: PrimitiveSet, min_height: int, max_height: int, condition: Callable[[int, int], bool]
) -> Iterable[Primitive | Terminal]:
    expr = []
    gen: Random = PyRndGenerator().gen
    height = gen.randint(min_height, max_height)
    stack = [0]
    while len(stack) != 0:
        depth = stack.pop()
        if condition(depth, height):
            term = gen.choice(pset.terminals)
            term.sample_if_needed()
            expr.append(term)
        else:
            prim = gen.choice(pset.primitives)
            expr.append(prim)
            for _ in range(prim.arity):
```

```
                stack.append(depth + 1)
    return expr


#########
# XOVER #
#########


def xover(
    parrent1: Tree, parrent2: Tree, prob_swap_prim: float = 0.5, prob_swap_term: float = 0.3
) -> tuple[Tree, Tree]:

    gen: Random = PyRndGenerator().gen

    if len(parrent1) < 2 or len(parrent2) < 2:
        # No crossover on single node tree
        return parrent1, parrent2

    offspring1 = copy.deepcopy(parrent1)
    offspring2 = copy.deepcopy(parrent2)

    idx1 = None
    idx2 = None
    sample = gen.random()
    if sample < prob_swap_prim:
        idx1 = gen.randrange(0, len(offspring1) - 1)
        while isinstance(offspring1[idx1], Primitive):
            idx1 = gen.randrange(0, len(offspring1) - 1)

        idx2 = gen.randrange(0, len(offspring2) - 1)
        while isinstance(offspring2[idx2], Primitive):
            idx2 = gen.randrange(0, len(offspring2) - 1)

    elif sample < prob_swap_prim + prob_swap_term:
        idx1 = gen.randrange(1, len(offspring1))
        while isinstance(offspring1[idx1], Terminal):
            idx1 = gen.randrange(1, len(offspring1))

        idx2 = gen.randrange(1, len(offspring2))
        while isinstance(offspring2[idx2], Terminal):
            idx2 = gen.randrange(1, len(offspring2))
    else:
        idx1 = gen.randrange(1, len(offspring1))
        idx2 = gen.randrange(1, len(offspring2))

    slice1 = offspring1.getSubTree(idx1)
    slice2 = offspring2.getSubTree(idx2)

    offspring1[slice1], offspring2[slice2] = offspring2[slice2], offspring1[slice1]

    return offspring1, offspring2


##############
# Mutations #
##############


def subtree_mut(
    individual: Tree,
    gen_func: Callable[[PrimitiveSet, int, int], Iterable[Terminal | Primitive]],
    pset: PrimitiveSet,
    min_height: int,
    max_height: int,
```

```
) -> tuple[Tree]:
    gen: Random = PyRndGenerator().gen
    offspring = copy.deepcopy(individual)
    idx = gen.randrange(len(offspring))
    of_slice = offspring.getSubTree(idx)
    offspring[of_slice] = gen_func(pset, min_height, max_height)
    return (offspring,)


def point_mut(individual: Tree, pset: PrimitiveSet) -> tuple[Tree]:
    gen: Random = PyRndGenerator().gen
    offspring = copy.deepcopy(individual)
    idx = gen.randrange(len(individual))
    node: Primitive | Terminal = individual[idx]
    narity = node.arity
    if narity != 0:
        new_prim = gen.choice(pset.prim_dict[narity])
        offspring[idx] = new_prim
    else:
        new_term = gen.choice(pset.terminals)
        offspring[idx] = new_term

    return (offspring,)


def hoist_mut(individual: Tree, pset: PrimitiveSet | None = None) -> tuple[Tree]:
    gen: Random = PyRndGenerator().gen
    idx = gen.randrange(len(individual))
    indslice = individual.getSubTree(idx)
    offspring = Tree(copy.deepcopy(individual[indslice]))
    return (offspring,)
```

src.toolbox.gp_objects.py

```
from abc import ABC
from collections import defaultdict
import numbers
import sys
from typing import Callable, Iterable, Protocol, Self
import numpy as np
from numpy import ufunc


NumberType = float | int | numbers.Real | numbers.Number


class Primitive:

    __slots__ = ("name", "arity", "str_code")

    def __init__(self, name: str, arity: int) -> None:
        self.name = name
        self.arity = arity
        args = ", ".join(map("{{{0}}}".format, range(self.arity)))
        # str_code is formated like func_name({0}, {1}, {2})
        self.str_code = f"{name}({args})"

    def format(self, *args):
        return self.str_code.format(*args)


class PrimitiveSet:
```

```
    def __init__(self, name: str, nvariables: int) -> None:
        self.name = name
        self.nvariables = nvariables
        self.term_count = 0
        self.prims_count = 0
        self.primitives: list[Primitive] = []
        self.prim_dict: dict[int, list[Primitive]] = defaultdict(list)
        self.terminals: list[Terminal] = []
        self.arguments = []
        self.context = {"__builtins__": None}
        for idx in range(nvariables):
            arg_str = "x{idx}".format(idx=idx)
            self.arguments.append(arg_str)
            term = Terminal(arg_str, True)
            self.terminals.append(term)
            self.term_count += 1

    def addPrimitive(self, func: ufunc) -> None:
        name = func.__name__
        arity = func.nin
        prim = Primitive(name, arity)
        self.primitives.append(prim)
        self.prim_dict[arity].append(prim)
        self.prims_count += 1

    def addTerminal(self, value: NumberType | Callable[..., NumberType], name: str | None = None)
-> None:
        term = Terminal(value, False, name)
        self.terminals.append(term)
        self.term_count += 1

    def addADF(self, adfSet: Self):
        prim = Primitive(adfSet.name, adfSet.nvariables)
        self.primitives.append(prim)
        self.prim_dict[adfSet.nvariables].append(prim)
        self.prims_count += 1

    @property
    def terminal_ratio(self) -> float:
        return self.term_count / (self.term_count + self.prims_count)


class Terminal:

    __slots__ = ("_value", "_func", "is_symbolic", "name", "format_class")

    def __init__(
        self, value: str | NumberType | Callable[..., NumberType], is_symbolic: bool, name: str |
None = None
    ) -> None:
        if isinstance(value, Callable):
            self._func = value
            self._value = None
        else:
            self._value = value
            self._func = None
        if name is None:
            self.name = str(value)
        else:
            self.name = name
        self.is_symbolic = is_symbolic
        self.format_class = str if is_symbolic else repr

    @property
    def arity(self) -> int:
```

```
        return 0

    def sample_if_needed(self) -> None:
        if not self.is_symbolic and self._func is not None:
            self._value = self._func()

    def format(self) -> str:
        assert self._value is not None, "ERROR: a terminal value that is not symbolic does not
have a value"
        return self.format_class(self._value)


class Individual(ABC):

    def __init__(self) -> None:
        self._fitness: float = -1

    @property
    def fitness(self) -> float:
        return self._fitness

    @fitness.setter
    def fitness(self, value: float) -> None:
        self._fitness = value


class Tree(list, Individual):

    def __init__(self, content: Iterable[Primitive | Terminal]):
        list.__init__(self, content)
        Individual.__init__(self)

    def __setitem__(self, key, value) -> None:
        if isinstance(key, slice) and isinstance(value, Iterable):
            if key.start >= len(self):
                raise IndexError("Can't set and item in a position %s when tree has size %d" %
(key, len(self)))
            arity_to_satisfy = value[0].arity
            for node in value[1:]:
                arity_to_satisfy += node.arity - 1
            if arity_to_satisfy != 0:
                raise ValueError("Invalid slice assignation, insertion of an incomplete subtree
is not permitted")
        elif value.arity != self[key].arity:
            raise ValueError(
                "The arity of the replacing node (%d) and the current node (%d) does not match"
                % (value.arity, self[key].arity)
            )
        list.__setitem__(self, key, value)

    def __str__(self):
        """Return the expression in a human readable string."""
        string = ""
        stack = []
        for node in self:
            stack.append((node, []))
            while len(stack[-1][1]) == stack[-1][0].arity:
                prim, args = stack.pop()
                string = prim.format(*args)
                if len(stack) == 0:
                    break  # If stack is empty, all nodes should have been seen
                stack[-1][1].append(string)

        return string
```

```
    @property
    def height(self) -> int:
        stack = [0]
        max_height = 0
        for node in self:
            current_depth = stack.pop()
            max_height = max(current_depth, max_height)
            stack.extend([current_depth + 1] * node.arity)
        return max_height

    def getSubTree(self, index: int) -> slice:
        finish = index + 1
        arity_to_satisfy = self[index].arity
        while arity_to_satisfy > 0:
            arity_to_satisfy += self[finish].arity - 1
            finish += 1
        return slice(index, finish)


class ADFIndividual(list, Individual):

    def __init__(self, content: list[Tree]):
        list.__init__(self, content)
        Individual.__init__(self)


class Compiler(Protocol):

    def compile(self, expr: Tree | ADFIndividual): ...


class SimpleCompiler:

    def __init__(self, pset: PrimitiveSet) -> None:
        self.pset = pset

    def compile(self, expr: Tree):
        return self._compile(expr, self.pset)

    @staticmethod
    def _compile(expr: Tree, pset: PrimitiveSet):
        code = str(expr)
        if pset.nvariables > 0:
            args = ",".join(arg for arg in pset.arguments)
            code = "lambda {args}: {code}".format(code=code, args=args)
        try:
            symbol_table = np.__dict__.copy()
            symbol_table["__builtins__"] = None
            if len(pset.context) > 1:
                symbol_table.update(pset.context)
            return eval(code, symbol_table), code
        except MemoryError as me:
            _, _, traceback = sys.exc_info()
            raise MemoryError(
                "Error in tree evaluation :"
                " Python cannot evaluate a tree higher than 90. "
                "To avoid this problem, you should use bloat control on your "
                "operators."
                "Program will now abort."
            ).with_traceback(traceback)


class ADFCompiler:

    def __init__(self, psets: list[PrimitiveSet]) -> None:
```

```
        self.psets = psets

    def compile(self, expr: ADFIndividual):
        adf_context = {}
        func = None
        code = ""
        # for pset, subexpr in reversed(list(zip(self.psets, expr))):
        for pset, subexpr in zip(self.psets[::-1], expr[::-1]):
            pset.context.update(adf_context)
            func, code = SimpleCompiler._compile(subexpr, pset)
            adf_context.update({pset.name: func})
        return func, code
```

src.toolbox.gp_population.py

```
from typing import Any, Callable, Protocol
from src.toolbox.gp_algorithms import ind_has_all_var
from src.toolbox.gp_objects import ADFIndividual, Individual, PrimitiveSet, Tree


class IndividualBuilder(Protocol):

    def gen_ind(self) -> Individual: ...


def get_init_population(builder: IndividualBuilder, fit_func, nind: int = 30) ->
list[Individual]:
    pop = []
    while nind > 0:
        ind = builder.gen_ind()
        fit = fit_func(ind)
        if fit != -1:
            ind.fitness = fit
            pop.append(ind)
            nind -= 1
    pop.sort(key=lambda ind: ind.fitness)
    return pop


class SimpleIndividualBuilder:

    def __init__(self, pset: PrimitiveSet, gen_func, gen_kwargs: dict[str, Any]) -> None:
        self.gen_func = gen_func
        self.pset = pset
        self.gen_kwargs = gen_kwargs

    def gen_ind(self) -> Individual:
        valid = False
        while not valid:
            ind = Tree(self.gen_func(pset=self.pset, **self.gen_kwargs))
            if ind_has_all_var(ind, self.pset):
                valid = True
        return ind


class ADFIndividualBuilder:

    # main must always be first

    def __init__(self, psets: list[PrimitiveSet], gen_funcs: list[Callable], gens_kwargs:
list[dict[str, Any]]) -> None:
        self.gen_funcs = gen_funcs
        self.psets = psets
```

```
        self.gens_kwargs = gens_kwargs

    def gen_ind(self) -> Individual:
        ind = []
        for func, pset, gen_kwargs in zip(self.gen_funcs, self.psets, self.gens_kwargs):
            ind.append(Tree(func(pset=pset, **gen_kwargs)))
        return ADFIndividual(ind)
```

src.toolbox.gp_statistics.py

```
from typing import Iterable

import numpy as np
from src.toolbox.gp_objects import Individual


class Statistics:

    def __init__(self, key_lambd=lambda x: x) -> None:
        self.key = key_lambd

    def compute_stats(self, pop: Iterable[Individual]) -> dict[str, float]:
        results = {}
        values = [self.key(ind) for ind in pop]
        results["avg"] = np.mean(values)
        results["std"] = np.std(values)
        results["min"] = np.min(values)
        results["max"] = np.max(values)
        return results
```

src.config.py

```
SEED = 93
```

src.exp_log.py

```
import copy
import json
from pathlib import Path

from src.toolbox.gp_objects import Individual


class RunLogger:

    def __init__(self, path: Path, file_name: str | None = None) -> None:
        self.base_path = path
        self.current_file_idx = 0
        if file_name is None:
            self.base_file_name = "run"
        else:
            self.base_file_name = file_name
        self.results = {}
        self._idx_cur_prob = None

    def set_problem(self, idx_prob: int) -> None:
        self._idx_cur_prob = idx_prob
```

```
        self.results[f"problem_{idx_prob}"] = {}

    def add_stats(self, stats: list[tuple[int, dict[str, float]]]) -> None:
        self.results[f"problem_{self._idx_cur_prob}"]["trace"] = list()
        for ngen, stat in stats:
            stat["ngen"] = ngen
            self.results[f"problem_{self._idx_cur_prob}"]["trace"].append(stat)

    def add_champion(self, ind: Individual) -> None:
        self.results[f"problem_{self._idx_cur_prob}"]["champ"] = {"function": str(ind),
"fitness": ind.fitness}

    def add_hyp(self, **params) -> None:
        hyp = copy.deepcopy(params)
        hyp["primitives"] = [prim.__name__ for prim in hyp["primitives"]]
        hyp["terminals"] = [term[1] for term in hyp["terminals"]]
        self.results["hyp"] = hyp
        self.results["hyp"]["gen_off_func"] = self.results["hyp"]["gen_off_func"].__name__
        self.results["hyp"]["selection_func"] = self.results["hyp"]["selection_func"].__name__

    def commit(self) -> None:
        full_path = self.base_path / (self.base_file_name + f"_{self.current_file_idx}.json")
        with open(full_path, "w") as f:
            json.dump(self.results, f)
        self.current_file_idx += 1
        self.results = {}
        self._idx_cur_prob = None
```

src.gp_random.py

```
import random
import numpy as np

from typing import Any
from src.config import SEED


class SingletonMeta(type):

    _instances = {}

    def __call__(cls, *args: Any, **kwargs: Any) -> Any:
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]


class NpRndGenerator(metaclass=SingletonMeta):

    def __init__(self) -> None:
        self._gen = np.random.default_rng(SEED)

    @property
    def gen(self) -> np.random.Generator:
        return self._gen


class PyRndGenerator(metaclass=SingletonMeta):

    def __init__(self) -> None:
        self._gen = random.Random(SEED)

    @property
```

```
    def gen(self) -> random.Random:
        return self._gen


if __name__ == "__main__":
    rnd_gen = NpRndGenerator()
    a = rnd_gen.gen.random()
    print(a)
    py_gen = PyRndGenerator().gen
    print(py_gen.choice([1, 2, 3]))
```

External links

As inspiration for the code structure some concept has been taken from:

- https://vision.gel.ulaval.ca/~cgagne/pubs/deap-gecco-2012.pdf
- https://github.com/DEAP/deap