# Set Cover problem

See: https://en.wikipedia.org/wiki/Set_cover_problem

Bunch of different strategies implemented to solve set cover problem. The final one is not reported here, look at it in set-cover.ipynb

```
from itertools import accumulate
import numpy as np
from tqdm.auto import tqdm
import matplotlib.pyplot as plt

from icecream import ic
```

## Reproducible Initialization

If you want to get reproducible results, use `rng` (and restart the kernel); for non-reproducible ones, use `np.random`.

```
UNIVERSE_SIZE = 1000
NUM_SETS = 100
DENSITY = 0.2

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
COSTS = np.pow(SETS.sum(axis=1), 1.1)
```

## Instances generation

### Data

```
universe_sizes = [100, 1000, 10_000, 100_000, 100_000, 100_000]
num_sets_sizes = [10, 100, 1000, 10_000, 10_000, 10_000]
densities = [.2, .2, .2, .1, .2, .3]

INIT_SOL_TH = 1
```

### Generator function

```
def generate_data(universe_size, num_sets, density):
    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
    COSTS = np.pow(SETS.sum(axis=1), 1.1)

    return SETS, COSTS
```

## Helper Functions

```
def valid(sets, solution):
    """Checks wether solution is valid (ie. covers all universe)"""
    phenotype = np.logical_or.reduce(sets[solution])     # at least each element
covered by a set
    return np.all(phenotype)                             # all elements are
covered

def coverage(sets, solution):
    """Returns the number of covered elements in the universe"""
    phenotype = np.logical_or.reduce(sets[solution])    # at least each element
covered by a set
    return np.sum(phenotype)                             # number of covered
elements

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

def fitness(sets: np.ndarray, costs: np.ndarray, solution: np.ndarray):
    """Returns the fitness of the given solution"""
    return (coverage(sets, solution), -cost(costs, solution))
```

```
def single_mutation(solution: np.ndarray):
    pos = rng.integers(0, solution.shape[0])
    solution[pos] = not solution[pos]
    return solution
```

```
def multiple_mutation(solution: np.ndarray):
    mask = rng.random(solution.shape[0]) < 0.99
    new_solution = np.logical_xor(mask, solution)
    return new_solution

def multiple_mutation_strength(solution: np.ndarray, strength: float = 0.3) ->
np.ndarray:
    mask = rng.random(solution.shape[0]) < strength
    if not np.any(mask):
        mask[np.random.randint(solution.shape[0])] = True

    new_sol = np.logical_xor(solution, mask)

    return new_sol
```

RM hill climbing with single mutation

```
def solve_single_mutation_HC(sets, costs, num_sets, num_steps=10_000,
th=INIT_SOL_TH):

    history = []
    solution = rng.random(num_sets) < INIT_SOL_TH
    sol_fitness = fitness(sets, costs, solution)

    print(f"Initial fitness: {sol_fitness}")

    history.append(float(sol_fitness[1]))
    for _ in tqdm(range(num_steps)):
        current = single_mutation(solution.copy())
        curr_fitness = fitness(sets, costs, current)

        #print(curr_fitness, sol_fitness)

        history.append(float(curr_fitness[1]))
        if curr_fitness > sol_fitness:
            solution = current
            sol_fitness = curr_fitness

    print(f"Final fitness: {sol_fitness}")
    print(f"Last update at iteration {history.index(float(sol_fitness[1]))}")

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

    return sol_fitness
```

## RM hill climbing with multiple mutation

```
def solve_multiple_mutation_HC(sets, costs, num_sets, num_steps=10_000,
th=INIT_SOL_TH):
    history = []
    solution = rng.random(num_sets) < th
    sol_fitness = fitness(sets, costs, solution)

    print(f"Initial fitness: {sol_fitness}")

    history.append(sol_fitness[1])
    for _ in tqdm(range(num_steps)):
        current = single_mutation(solution.copy())
        curr_fitness = fitness(sets, costs, current)

        #print(curr_fitness, sol_fitness)

        history.append(curr_fitness[1])
        if curr_fitness > sol_fitness:
            solution = current
            sol_fitness = curr_fitness

    print(f"Final fitness: {sol_fitness}")
    print(f"Last update at iteration {history.index(float(sol_fitness[1]))}")

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

    return sol_fitness
```

## Simulated annealing

It seems to perform worst than a RMHC: too much going around and not exploit neighboring solutions.

```
def solve_simulated_annealing_HC(sets, costs, num_sets, num_steps=10_000,
th=INIT_SOL_TH):

    def complete(covered):
        return covered == sets.shape[1]

    history = []
    solution = rng.random(num_sets) < th
    sol_fitness = fitness(sets, costs, solution)
    final_sol_fitness = sol_fitness
```

```
    print(f"Initial fitness: {sol_fitness}")

    history.append(sol_fitness[1])
    for i in tqdm(range(num_steps)):
        current = multiple_mutation(solution.copy())      # using single mutation
to avoid too much exploration
        curr_fitness = fitness(sets, costs, current)

        # Exploring when high coverage, exploiting otherwise
        # Min temperature set to 1 to avoid numerical issues in scalar power
        temperature = max(1, 10 * (sol_fitness[0] / sets.shape[1]) + 0.01)

        history.append(curr_fitness[1])

        logp = (curr_fitness[1] - sol_fitness[1]) / temperature + 1e-6

        if curr_fitness < sol_fitness and np.log(rng.random() + 1e-6) < logp or
curr_fitness > sol_fitness:

            if curr_fitness > final_sol_fitness and complete(curr_fitness[0]):
                final_sol_fitness = curr_fitness

            sol_fitness = curr_fitness
            solution = current

    print(f"Final fitness: {final_sol_fitness}")
    print(f"Last update at iteration {history.index(final_sol_fitness[1])}")

    plt.figure(figsize=(14, 8))
    plt.plot(
        range(len(history)),
        list(accumulate(history, max)),
        color="red",
    )
    plt.scatter(range(len(history)), history, marker=".", color="blue")

    return final_sol_fitness
```

## Simulated annealing with linear self-adaption

Simulated annealing approach but with linear self-adaption. The parameter *strength*, that acts as *temperature*, is increased (or decreased) by a 20% factor, depending on the success of at least one trial out of last five ones.

```
def solve_linear_SAHC(sets, costs, num_sets, num_steps=10_000, buf_size=5):
    history = []
    buffer = []
    solution = np.full(num_sets, True)
    sol_fitness = fitness(sets, costs, solution)

    ic(sol_fitness)
    history.append(float(sol_fitness[1]))
```

```
    strength = 0.5

    for steps in tqdm(range(num_steps)):

        new_sol = multiple_mutation_strength(solution, strength)
        new_sol_fitness = fitness(sets, costs, new_sol)

        history.append(float(new_sol_fitness[1]))

        buffer.append(new_sol_fitness > sol_fitness)
        buffer = buffer[-buf_size: ]

        if sum(buffer) > 1:
            strength *= 1.2

        elif sum(buffer) == 0:
            strength /= 1.2

        if new_sol_fitness > sol_fitness:
            solution = new_sol
            sol_fitness = fitness(sets, costs, solution)

    ic(sol_fitness)
    ic(history.index(sol_fitness[1]))

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

    return sol_fitness
```

## General solver

Script to solve task with multiple strategies and perform comparisons

```
class Strategies:
    SINGLE_MUTATION_HC = "Single mutation hill climber"
    MULTIPLE_MUTATION_HC = "Multiple mutation hill climber"
    SIMULATED_ANNEALING_EXP = "Simulated annealing hill climber - Exponential
adaption"
    SIMULATED_ANNEALING_LINEAR = "Simulated annealing hill climber - Linear
adaption"

    def to_list():
        return [
            Strategies.SINGLE_MUTATION_HC,
```

```
            Strategies.MULTIPLE_MUTATION_HC,
            Strategies.SIMULATED_ANNEALING_EXP,
            Strategies.SIMULATED_ANNEALING_LINEAR
        ]

def solve(sets: np.ndarray, costs: np.ndarray, strategy: str):
    n = sets.shape[0]
    u = sets.shape[1]
    steps = int(min(10_000, max(n*u // 50, 100)))
    th_start = 0.95 if n < 1000 else INIT_SOL_TH
    match strategy:
        case Strategies.SINGLE_MUTATION_HC:
            return solve_single_mutation_HC(sets, costs, n, num_steps=steps)
        case Strategies.MULTIPLE_MUTATION_HC:
            return solve_multiple_mutation_HC(sets, costs, n, num_steps=steps)
        case Strategies.SIMULATED_ANNEALING_EXP:
            return solve_simulated_annealing_HC(sets, costs, n, num_steps=10_000)
        case Strategies.SIMULATED_ANNEALING_LINEAR:
            return solve_linear_SAHC(sets, costs, n)
```

```
for (i, (universe_size, num_sets, density)) in list(enumerate(zip(universe_sizes,
num_sets_sizes, densities)))[:3]:

    print(f"Generating instance {i+1}")

    SETS, COSTS = generate_data(universe_size, num_sets, density)

    print(f"Solving instance {i+1}")

    fitnesses = {}
    for strategy_name in Strategies.to_list():
        fitnesses[strategy_name] = solve(SETS, COSTS, strategy_name)
        plt.show()

    for (strategy, fitness_val) in fitnesses.items():
        print(f"{strategy}: {fitness_val}")
```