Université de Genève

Metaheuristics for Optimization
14X013

The Quadratic Assignment Problem

Antoine Sutter
antoine.sutter@etu.unige.ch

October 2024

UNIVERSITÉ
DE GENÈVE
FACULTÉ DES SCIENCES
Département d'informatique

Contents

1. Code explaination 3
2. Results 3
2.1. Improvements 4
3. Questions 5
3.1. In this specific case, what is the neighborhood of an element of the research space? 5
3.2. In terms of n (number of locations/facilities), what is the size of the neighborhood? (i.e., how many neighbors does each permuation have?) 5
4. Main.py 6

1. Code explaination

This report won't cover the code line by line as the code already has comments and is pretty safe explainatory. The code is available in the `main.py` file that should be next to this report. It is also available at the end of this report Section 4.

2. Results

The algorythm has been implemented to accept a variable tenure as well as the option to enable diversification or not. In order to judge this impact of both of these, the algorythm has been ran 10 times for each tenure in $\{1, 0.5n, 0.9n\}$ where n is the width of the square matrix, each time with and without diversification. The algorythm internally iterates 500 times before stopping and returning whatever the best solution is at the time.

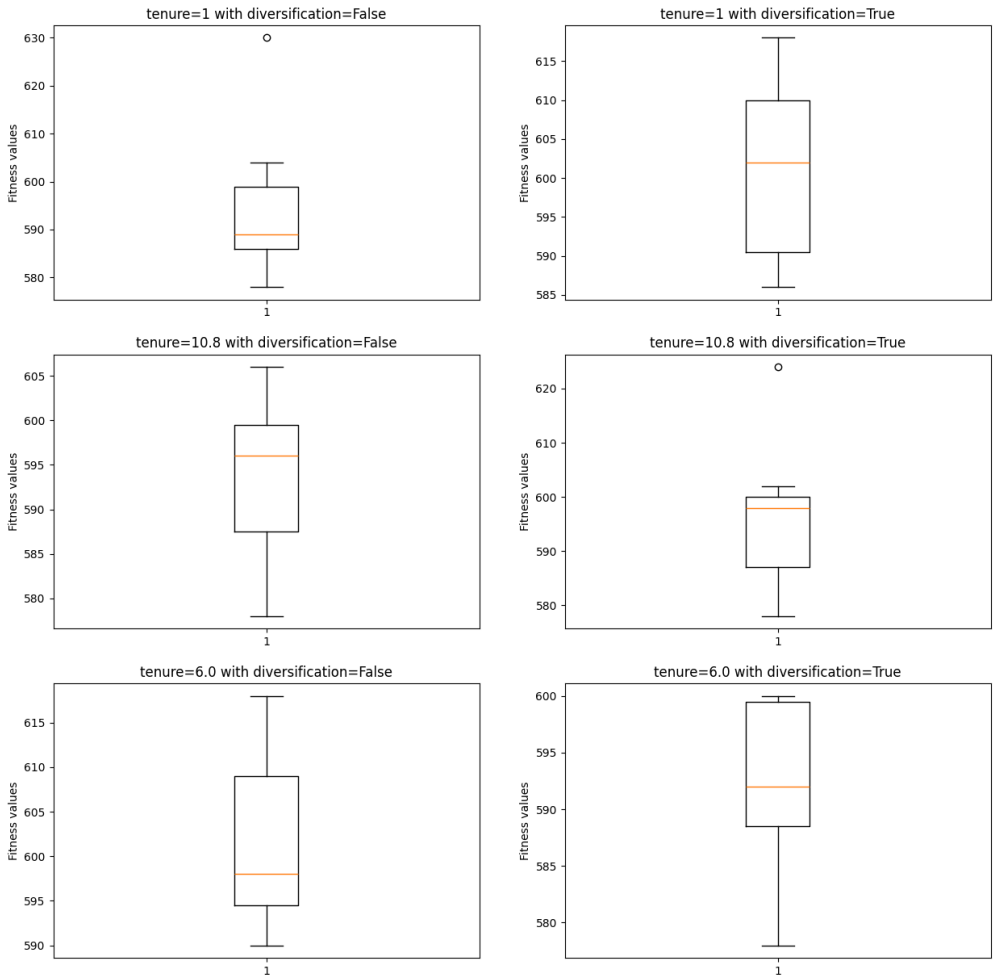

Figure 2: Result per tenure,
with and without diversification

As we can see in Figure 2, when the tenure is low, the results aren't great and diversification doesn't seem to be able to help much. However with increased tenure, the results impove and diversification starts having an impact on the results. However the results vary greatly from the initial element which in this case is a random permuation.

2.1. Improvements

Some things could be improved in this code. As highlighted in the code's comment, we currently only return a single solution. However, it is possible to find 2 solutions with the same fitness result and we could return a list of solutions instead.

```
        # save it if it's the global best so far
        # we could save a list instead in case we find
        # multiple best solutions
        if neighbor_fitness < best_fitness:
            best_fitness = neighbor_fitness
            best_combination = neighbor
```

We could also think about implementing some caching to avoid duplicated calculations for the code to perform better. This would allows us to increase the amount of internal iterations for better results.

3. Questions

3.1. In this specific case, what is the neighborhood of an element of the research space?

An element is defined a permutation of size n, meaning all unique numbers from $0..n$ in a list. The neighborhood of an element is defined as all the possible unique swaps of 2 values in that list. In this project for example here is how the function that renerates all the swaps is defined:

```python
def get_swaps(n: int) -> list[tuple[int, int]]:
    """
    Generates all the indices to swap two values in a list, without duplicate
    """
    return [
        (i, j)
        for i in range(n)
        for j in range(i + 1, n)
    ]
```

So for $n = 5$, you would get the following output:

```
[
  (0, 1), (0, 2), (0, 3), (0, 4),
  (1, 2), (1, 3), (1, 4),
  (2, 3), (2, 4),
  (3, 4)
]
```

3.2. In terms of n (number of locations/facilities), what is the size of the neighborhood? (i.e., how many neighbors does each permuation have?)

The size of the neighborhood is n chose 2. Given the code written above, it can be defined as

$$S = \frac{n(n-1)}{2}$$

4. Main.py

```
import argparse
from functools import lru_cache
import matplotlib.pyplot as plt
import numpy as np
import random


def read_input(filename: str) -> tuple[int, np.ndarray, np.ndarray]:
    """
    read .dat file and return n, D, and W

    n: number of facilities
    D: Distance Matrix
    W: Weight Matrix
    """
    n = int(np.loadtxt(filename, skiprows=0, max_rows=1, dtype=int))
    D = np.loadtxt(filename, skiprows=2, max_rows=n, dtype=int)
    W = np.loadtxt(filename, skiprows=3 + n, max_rows=n, dtype=int)

    assert D.shape == (n, n)
    assert W.shape == (n, n)

    return n, D, W


def compute_I(combination: np.ndarray, *, n: int, D: np.ndarray, W: np.ndarray) ->
float:
    """
    compute fitness of a combination
    """
    return np.sum([
        W[i, j] * D[combination[i], combination[j]]
        for i in range(n)
        for j in range(n)
    ])


def compute_delta(combination: np.ndarray, i: int, j: int, *, n: int, D:
np.ndarray, W: np.ndarray) -> float:
    """
    compute delta fitness after swapping facilities in locations i & j (ie. delta
fitness for one neighbor)
    """
    neighbor = swap_two(combination, i, j)
    return compute_I(neighbor, n=n, D=D, W=W) - compute_I(combination, n=n, D=D,
W=W)


@lru_cache()
def get_swaps(n: int) -> list[tuple[int, int]]:
    """
    Generates all the indices to swap two values in a list, without duplicate
    """
    return [
```

```
        (i, j)
        for i in range(n)
        for j in range(i + 1, n)
    ]


def get_non_tabu_swaps(tabu_list: np.ndarray, combination: np.ndarray, *, n: int,
iteration: int) -> list[
    tuple[int, int]]:
    """
    Returns all the swaps that aren't prohibited by the tabu_list
    """
    return [
        swap
        for swap in get_swaps(n)
        if not is_tabu(tabu_list, combination, *swap, iteration=iteration)
    ]


def swap_two(combination: np.ndarray, i: int, j: int) -> np.ndarray:
    """
    returns a new permutation with facilities in i & j swapped
    """
    neighbor = combination.copy()
    neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
    return neighbor


def is_tabu(tabu_list: np.ndarray, combination: np.ndarray, i: int, j: int,
iteration: int):
    """
    Tests if a swap if tabu or not
    """
    neighbor = swap_two(combination, i, j)
    return tabu_list[neighbor[j], i] > iteration and tabu_list[neighbor[i], j] >
iteration


def update_tabu(tabu_list: np.ndarray, combination: np.ndarray, i: int, j: int,
tenure: float):
    """
    For a given (i, j), update the tabu list given the tenure
    """
    neighbor = swap_two(combination, i, j)
    tabu_list[neighbor[j], i] = tabu_list[neighbor[i], j] = tenure


def filter_diversification(div_matrix: np.ndarray, combination: np.ndarray, *, n:
int, iteration: int) -> list[
    tuple[int, int]]:
    """
    Returns a list of all the swaps that haven't been done for n ** 2 iterations
    """
    return [
        (i, j)
```

```
        for i, j in get_swaps(n)
        if iteration - div_matrix[combination[i], combination[j]] > n ** 2
    ]


def update_diversification(div_matrix: np.ndarray, combination: np.ndarray, i:
int, j: int):
    """
    Update how long ago a given swap has been used for the last time
    """
    div_matrix[combination[i], combination[j]] += 1


def solve(*, n: int, D: np.ndarray, W: np.ndarray, tenure: int, iterations: int,
with_diversification: bool) \
        -> tuple[float, np.ndarray]:
    tabu_list = np.zeros((n, n))
    div_matrix = np.zeros((n, n))

    # loop variables
    current_combination = np.random.permutation(n)
    current_fitness = compute_I(current_combination, n=n, D=D, W=W)

    # save the starting position as the best
    best_combination = current_combination
    best_fitness = current_fitness

    for iteration in range(1, iterations + 1):
        # Get the regular non tabu swaps
        swaps = get_non_tabu_swaps(tabu_list, current_combination, n=n,
iteration=iteration)

        if with_diversification:
            # Force the diversitiy swaps over the non tabu ones
            forced = filter_diversification(div_matrix, current_combination, n=n,
iteration=iteration)
            if len(forced) > 0:
                swaps = forced

        if len(swaps) == 0:
            # we have no potential swap for some reason
            # just continue, eventually we will find a swap
            continue

        # compute the deltas for all the swaps
        neighbors_deltas = [
            ((i, j), compute_delta(current_combination, i, j, n=n, D=D, W=W))
            for i, j in swaps
        ]

        # find the best neighbors, might be more than one!
        _, best_delta = min(neighbors_deltas, key=lambda x: x[1])
        best_neighbors = [*filter(lambda x: x[1] == best_delta, neighbors_deltas)]

        # Pick a random neighbor between the list of all best neighbors
```

```
        (i, j), delta = random.choice(best_neighbors)

        # tenure is how long that move will be banned
        update_tabu(tabu_list, current_combination, i, j, iteration + tenure)

        if with_diversification:
            # Also update the diversity matrix
            update_diversification(div_matrix, current_combination, i, j)

        # compute the chosen neighbor
        neighbor = swap_two(current_combination, i, j)
        neighbor_fitness = current_fitness + delta

        # save it if it's the global best so far
        # we could save a list instead in case we find
        # multiple best solutions
        if neighbor_fitness < best_fitness:
            best_fitness = neighbor_fitness
            best_combination = neighbor

        # next loop ready
        current_fitness = neighbor_fitness
        current_combination = neighbor

    # best solution we have I guess
    return best_fitness, best_combination


if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("-f", "--file", choices=["1.dat", "2.dat"],
default="1.dat")
    parser.add_argument("-i", "--iterations", default=500, type=int)
    args, _ = parser.parse_known_args()

    n, D, W = read_input(args.file)

    fig, axs = plt.subplots(3, 2, figsize=(15, 15))
    axs = axs.flatten()

    i = 0
    for tenure in [1, 0.9 * n, 0.5 * n]:
        for with_diversification in [False, True]:
            print(f"tenure={tenure}, diversification={with_diversification}")

            results = [solve(
                n=n,
                D=D,
                W=W,
                iterations=args.iterations,
                tenure=tenure,
                with_diversification=with_diversification
            ) for i in range(10)]

            fitnesses = [fitness for fitness, _ in results]
```

```
            axs[i].boxplot(fitnesses)
            axs[i].set_ylabel('Fitness values')
            axs[i].set_title(f"tenure={tenure} with
diversification={with_diversification}")
            i += 1

    plt.show()
```