Optimization of the Bi-objective Travelling Thief Problem for the GECCO2019 Competition

Esosa Orumwese

13th December 2023

1 Introduction

Real-world optimization problems often consist of several NP-hard combinatorial optimization problems that interact with each other. Such multi-component optimization problems are difficult to solve not only because of the contained hard optimization problems, but in particular, because of the interdependencies between the different c omponents. I nterdependence c omplicates a decision making by forcing each sub-problem to influence t he q uality a nd f easibility o f s olutions o f the other sub-problems. This influence m ight b e e ven s tronger w hen o ne s ub-problem c hanges the data used by another one through a solution construction process. Examples of multi-component problems are vehicle routing problems under loading constraints, maximizing material utilization while respecting a production schedule, and relocation of containers in a port while minimizing idle times of ships.

The goal of this competition was to provide a platform for researchers in computational intelligence working on multi-component optimization problems. The main focus of this competition was on the combination of Travelling Salesman Problem (TSP) and Knapsack Problems (KP). The TSP problem one in which the salesman has n cities and the task is to find t he b est possible route which minimizes the total distance. While for the KP problem, a knapsack has to be filled with items of value b_j and weight w_j without violating the maximum weight constraint Q while maximizing the total profit.

For this Travelling Thief Problem, the two objectives become interwoven because the thief has to go through n cities each with m items and fill up t he knapsack w ithout v iolating t he Q . A nd as a result of the weight of items in his knapsack, his velocity v reduces and so does the time of the tour. So the thief's two objectives now include minimizing the total tour time while maximizing the profit g ained. So t he weight of i tems is b oth considered in t he KP aspect of t he p roblem, as Q can't be violated, as well as in the velocity of the thief as he travels between cities in his tour.

2 Methodology

2.1 Algorithm Survey

In approaching the Traveling Thief Problem (TTP), our team initially contemplated employing a generic genetic algorithm. This method was chosen due to its versatility and efficacy in optimizing complex problems. The idea behind this algorithm was to iteratively improve the solution's fitness by conducting operations within a lengthy k-array that contained both the Traveling Salesperson Problem (TSP) tour and the packing plan. The genetic algorithm would have operated by utilizing evolutionary operations like crossover and mutation, making it a promising choice to explore various solution possibilities and converge towards an optimal or near-optimal solution.

Additionally, we were inclined towards exploring an Ant Colony Optimizer. This choice was motivated by the nature-inspired behavior of ant colonies, which exhibit remarkable efficiency in finding optimal p aths. The idea was to mimic this behavior in solving the TTP by allowing artificial ants to traverse through the solution space, laying pheromone trails, and collectively discovering high-quality solutions. Ant Colony Optimization showed promise in providing robust solutions by leveraging the concept of cooperation among agents and exploiting the information shared among the colony.

Our choice of using a genetic algorithm centered on a k-array setup comprising the TSP tour and packing plan, with an objective function evaluating TSP tour time and KP profit simultaneously while that of the Ant Colony Optimizer employed pheromone matrices to represent TSP edges and KP items, utilizing pheromone trails to guide decision-making. Both methods shared a common objective function assessing TSP distance and KP profit, but the Ant Colony Optimizer leveraged pheromone trails to facilitate exploration.

But for a more informed approach, we decided to look at research papers and finally we choose a non-dominated sorting biased random-key genetic algorithm by Chagas et al [1]. This method showed the most promise for four main reasons.

1. Firstly, it made use of domain knowledge by not initializing the population randomly but instead introducing a small set of solutions (tour plans and packing orders) that are good on the TSP and KP part of the problem separately.

2. Secondly, to avoid the evaluation of invalid solutions, a repair operator was introduced to repair the solutions whose packing orders had broken the maximum weight constraint.

3. Thirdly, as this is a problem which deals with two sets of variable types (permutation for the tour plan, binary decision variable for the packing order), solutions were encoded using the biased-random key principle so that traditional evolutionary genetic operations could easily be performed on solutions.

4. Lastly, using NSGA2 operations, we were able to consider and compare both conflicting objectives non-dominated sorting and crowding distance in the survival selection. Although this couldn't help quantify the quality of solutions, it ensured that final population had a set of non-dominated solutions.

2.2 Experimentation Process

Given the time constraint that we had as a team, we decided to take a more informed approach by reading through academic papers to find how best the TTP. As the Traveling Thief Problem is benchmark developed by Polyakovskiy et al. [2], we came across a lot of techniques to solve the problem with different suggestions for parameters but for the reasons mentioned above, we chose to still stick with the Non-Dominated Sorting Biased Random-Key Genetic Algorithm (NDSBRKGA). This choice seemed better suited for this task because it allowed to focus on understanding the algorithm design and its implementation.

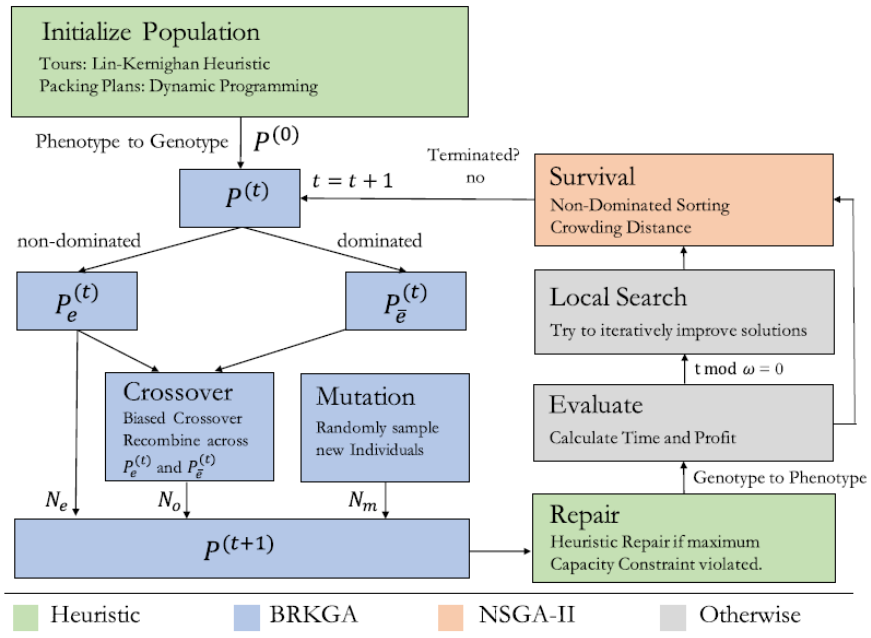

Figure 1: Flowchart of the Non-dominated Sorting Biased Random-Key Genetic Algorithm [1]. Chart by [1]

Working on the algorithm itself, we first decided on how to build the TSP and KP solvers which help in the domain knowledge of the TTP algorithm. While we were able to create KP solver which worked on dynamic programming, creating the TSP solver based on the Lin-Kernighan Heuristic [3] proved challenging so we improved upon our TSP solver created in the last coursework. This did not pose a problem to us because what we need were good tour plans from the TSP solver and not the best because the performance of the algorithm sees no significant change when the best solution is used compared to 'good' solutions [1].

The next research that needed to be done in order to implement the NDSBRKGA was to implement the 2-opt local search heuristic for the TSP aspect of our solutions and the bit-flip random moves for the KP aspect. This was set as an exploitation phase to occur after a specific number of iterations, so that the algorithm focuses more on convergence. As this algorithm is heavily biased on the elites, to promote diversity, a user specified number of randomly generated solutions were added into the population at every iteration. Also our survivor selection method was based on non-dominated sorting and crowding distance which was done via an implementation of NSGA2 operators in our algorithm

Unfortunately, due to the long runtime of the dataset, we could not use the parameters which were suggested in the research paper. As a result we tried running a set of smaller parameters tests to ascertain which would produce the best result but because of the size of the dataset and the specific way in which our implemented functions worked, that still took a significant amount of time and we weren't able to get the right parameters that could have returned the best solutions for the both objectives.

Suggestions that could be made to further speed up the runtime will be to research on ways to efficiently read and store the distance matrix, better ways to iterate through each k-array (which had lengths equal to the number of city), and to also implement some of our functions in a low-level language like C or C++.

2.3 Teamwork Process

As a team of six, only four of us participated throughout the timeline of this project. Even with that constraint on manpower, we still tried to make the best use of our resources. Given the algorithm design that we got from our research paper, we formulated an algorithm design, as is detailed in our group report, which revolved around the flowchart in 1. Each of the processes were broken down into functions and each team member was given a set of functions to research for algorithms detailed in research papers, and implement. Then we had meetings in the Babbage computer lab to stitch up our code together.

Since our TSP and KP solvers could be run outside of the main algorithm, we ran it before meeting so as to get results. And when we meet together and formed our whole algorithm, we ran only 6 out of the 9 datasets on separate desktops in Babbage computer lab, Innovation Center. The other datasets couldn't be worked on because the TSP solver couldn't produce results after 1 day due to its large amount of cities. Given more time, further research would have been conducted to find the most efficient way of dealing with such a large dataset given the computational resources that we had.

The most challenging aspects that we experienced working as a team was communication but this only occurred at the start of the project and it was because we were an international team. After a few days of communicating, we got used to each other. Another aspect that proved challenging to us was working together remotely. We noticed that we ran into a lot of issues when we weren't working physically. Apart from the negatives, one major bonus of working in the team was that we all brought in different ideas to each challenge and that help ease and speed up the workflow.

2.4 My Contribution to the Work Flow

I was in-charge of assigning tasks and also guiding the implementation and merging of each function in the algorithm such that it was inline with what the author intended. To better guide my teammates, I had to ensure that I properly read and understood our reference research paper so that I could better pick out the flaws in each function that was created by each team member.

Seeing as offer the Multi-Objective Optimization and Decision Making module, I used my knowledge to help build the NSGA2 operators which were used in the survival section of our algorithm and I implemented the local search function which performed 2-opt local search and

random bitflip on the TSP and KP component of each selected solutions. Code snippets can be seen in Appendix A.

The first set of codes that I implement where made to handle the NSGA2 aspects which were needed in the algorithm. Selection of the best solutions needed to be made and given that there were two conflicting objectives, selection criteria was made on non-dominated sorting and crowding distances. So my function took in the parent and offspring population and released an elites list and a non-elites list. The second function which I implement was the local search which performed 2-opt local search on the TSP component of each solution. For a specific number of iterations, it swaps two edges and tried to get solutions that dominated the initial solution. It also performed random bit-flip on the KP component of the solution

3 Conclusion

In summary, our approach proved beneficial because instead of solving the problem through some unguided means, we made use of scholarly material to help guide us towards the answer. As discussed in the group report, although our implementation of the algorithm was right, the runtime required for the large dataset was much and as a result we could not use the parameters that were suggested in the study. And because we could run multiple tests to ascertain the best parameters for the project given the smaller epochs that we wanted to work with, we ended up having an average result. Given more time and better computational resources, our implementation could have come out successful as we would have been able to easily make suggestions if our results didn't initially meet our expectations. As a team, for the four of us who participate, our teamwork was excellent as everyone spent late night hours trying to deliver their aspect of the work. Other approaches I would have tested was to research more on how ACO could be implemented to solve this TTP problem. As discussed earlier, ACO seems as a viable option but the implementation to this interwoven bi-objective problem would prove tricky and would require more research to be done.

4 Appendix A: Code Snippets

```python
def non_dominated_sorting(pop_df):
    """
    Perform Non-dominated Sorting on a Population where pop refers to the population dataframe
    Each row represents a solution and the columns are TSP, KP, cost, rank and crowding distance
    # check out column for the rank of each solution??
    """
    pop_size = len(pop_df) # gets the number of solution

    # Initialize Domination Stats
    domination_set = [[] for _ in range(pop_size)] # identifies the solutions that each solution dominates. Just stores and empty list for now
    dominated_count = [0 for _ in range(pop_size)] # how many solutions dominate each solution

    # Initialize Pareto Fronts <non-dominated fronts>
    F = [[]]

    # Find the first Pareto Front
    for i in range(pop_size):
        for j in range(i+1, pop_size):
            # Check if i dominates j
            if dominates(pop_df.loc[i], pop_df.loc[j]):
                #if dominates(pop[i], pop[j]):
                domination_set[i].append(j) # if i dom j, then add j to its domination list
                dominated_count[j] += 1 # add 1 to the number of solutions that dominate j

            # Check if j dominates i
            elif dominates(pop_df.loc[j], pop_df.loc[i]):
                #elif dominates(pop[j], pop[i]):
                domination_set[j].append(i)
                dominated_count[i] += 1

        # If i is not dominated at all
        if dominated_count[i] == 0:
            pop_df.loc[i,'rank'] = 0
            #pop[i]['rank'] = 0
            F[0].append(i) # add to Pareto optimal front
```

Figure 2: Non-dominated sorting function

```python
def calc_crowding_distance(pop_df, F):
    """Calculate the crowding distance for a given population with the Fronts"""

    # Number of Pareto fronts (ranks)
    pareto_count = len(F)

    # Number of Objective Functions
    n_obj = len(pop_df.loc[0, 'cost'])

    # Iterate over Pareto fronts
    for k in range(pareto_count):
        # get all the costs for the elements in the Pareto front. Results in a matrix of size pareto_count x n_obj
        costs = np.array([pop_df.loc[i,'cost'] for i in F[k]])
        n = len(F[k]) # number of members in k-th front
        d = np.zeros((n, n_obj)) # creates a zero matrix of same shape as costs to store crowding distance

        # Iterate over objectives, column by column
        for j in range(n_obj):
            idx = np.argsort(costs[:, j]) # for the jth obj, get the idx which sorts it. You're basically looking for the boundary solutions
            # assign boundary solutions a distance = infinity
            d[idx[0], j] = np.inf  # first
            d[idx[-1], j] = np.inf # last

            # iterate through the remaining values excluding the bounday solutions
            for i in range(1, n-1):
                # either calculates Euclidean distance or Manhattan to gauge distance
                d[idx[i], j] = costs[idx[i+1], j] - costs[idx[i-1], j]
                d[idx[i], j] /= costs[idx[-1], j] - costs[idx[0], j]

        # Calculate Crowding Distance
        for i in range(n):
            pop_df.loc[F[k][i], 'crowding_distance'] = sum(d[i, :])
            #pop_df[F[k][i]]['crowding_distance'] = sum(d[i, :]) # assigns crowding distance based on elements in each front

    return pop_df
```

Figure 3: Crowding distance function

```python
# Local Search - Exploitation Phase
def local_Search(P_e, W, df, Q, distances, n_of_Ne=0.1):
    '''
    Performs the local search aspect of the algorithm on a subset of the elite population.
      Parameters:
        P_e: current elite population
        W: Weight of items in each city
        df: unknown?? Need to be confirmed from Xiao
        Q: Total weight constraint for the knapsack
        n_of_Ne: How many of the elite population do we want. Default=0.1
      Returns:
        P_e with added modified elites

    '''
    # get subset of elites that we would exploit to make better
    n_Pe_hat = round(n_of_Ne*len(P_e))
    Pe_hat = np.random.choice(P_e,n_Pe_hat)

    for elite in Pe_hat:
        # perform 2-opt local search on the TSP aspect
        elite = two_opt(elite, df, distances, Q)

        # perform bitflip on the KP aspect
        elite, P_e = bitflip_exploit(elite, W, Q, P_e, df, distances)

    return P_e #only because it contains all the exploited elites

def bitflip_exploit(soln, W, Q, P_e, df, distances):
    '''
    Performs bit-flip given a solution for about min(100,m) moves
    where m is the number of items.
    Solution format: soln = {'TSP':[], 'KP':[], 'cost': [dist,prof]}
    W is the weight and I'm using it's the same length as sol['KP']
```

Figure 4: Local Search function

References

[1] J. B. Chagas, J. Blank, M. Wagner, M. J. Souza, and K. Deb, "A non-dominated sorting based customized random-key genetic algorithm for the bi-objective traveling thief problem," *Journal of Heuristics*, vol. 27, no. 3, pp. 267–301, 2021.

[2] S. Polyakovskiy, M. R. Bonyadi, M. Wagner, Z. Michalewicz, and F. Neumann, "A comprehensive benchmark set and heuristics for the traveling thief problem," in *Proceedings of the 2014 annual conference on genetic and evolutionary computation*, 2014, pp. 477–484.

[3] S. Lin and B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," *Operations research*, vol. 21, no. 2, pp. 498–516, 1973.