Optimising the travelling thief problem using an evolutionary algorithm

Introduction

The travelling thief problem (TTP) is a bi-objective optimisation problem combining the travelling salesman problem (TSP) and the knapsack problem (KNP) [1]. The travelling salesman problem is a graphical problem where cities and their connected roads are represented as a graph. The objective is to minimise the amount of time to visit all n cities in the graph. The time is calculated using the known distance between the cities and the traveller's velocity. The traveller's velocity is affected by the number of items they carry. This is how the knapsack problem is integrated, as it is a problem to maximise the profit of the items carried without violating the maximum weight constraints. The traveller can collect items at every city he visits. These items can have different weights. A heavier bag reduces the velocity of the traveller and, as such, increases the time required to finish the tour. The two objectives are to maximise profit and minimise tour length.

Algorithm Survey

To determine the best nature-inspired approach, research was conducted to check previously published work for the travelling thief problem and multi-objective approaches used in other fields.

Previously used nature-inspired approaches to solve the travelling thief problem include using an evolutionary algorithm (EA) [2], ant colony optimisation [3] or simulated annealing [4]. All methods achieve competitive results in comparison to state-of-the-art heuristics [5, 6, 7]. However, the ant colony approach focuses on the tour, reducing the packing into a heuristic such as packiterative [3]. A hill-climber is can then be used on the best solutions in order to improve further. Another possible drawback is that the method is computationally intensive and unsuitable for datasets with larger numbers of cities and items. On the other hand, the evolutionary algorithm maintains the bi-objective nature of the problem using a greedy deterministic algorithm to collect items [2]. This can be incorporated into the Non-dominated Sorting Genetic Algorithm-II (NSGA-II) to optimise solutions. This approach is advantageous over an ant colony approach because evolutionary algorithms can traverse the search space more effectively, as both objectives can be varied. Unlike ant colony optimisation and evolutionary algorithms, simulated annealing attempts to optimise a single solution, modifying it until it does not improve.

Other multi-objective approaches for combinatorial problems include hybrid methods such as co-evolution or multi-objective cellular genetic algorithm (MOCell) [8]. These problems haven't been used in any published research on the travelling thief problem, but they have succeeded in other bi-objective graphical problems, such as bi-objective scheduling problems or military unit path planning problems. This suggests they can be adjusted to solve the travelling thief problem.

We decided to discard approaches without published work associated with the travelling thief problem because creating accurate representation and objective functions for these models would add complexity to the project over established methods. This left us with ant colony optimisation, NSGA-II or simulated annealing. Each method was chosen because it had been implemented in research and had a defined representation. For ant colony optimisation, the problem is represented like the TSP, where the algorithm uses pheromones to optimise a path based on weighting good paths [3]. The KNP is incorporated after a tour is created using a heuristic that considers the items' profit, weight and distance to the final city. The objective function is minimising the tour and then finding the maximum possible profit for that tour, so the priority is given to minimising the tour. The representation for NSGA-II is creating a set of initial candidate solutions using a local heuristic, such as pack optimal for the packing plan, to generate good initial solutions [2]. Each candidate solution is ranked based on the Pareto front it lies in, then the crowding distance. Then, a tournament selection is run twice, and the parents with the best rank and greatest crowding distance are selected. Two children are created using a crossover function on the parents and mutated before being evaluated. This process is repeated in a

generation until there are the same number of children as parents. The entire population ranks, and then crowding distances are compared, and the lowest half is discarded. The process is then repeated a given number of times. The objective is to improve the children in each generation to surpass the previous results. This takes into account both travel time and profit. Finally, simulated annealing is used in an algorithm that iterates between optimising the travel time and the profit [4]. This can be completed using the algorithm CosSolver with 2-OPT and Simulated Annealing (CS2SA). It swaps between the tour and the picking plan being fixed. The tour is improved using a 2-OPT operator. The packing plan is improved using simulated annealing, adapted for the knapsack problem. Its objective is to improve a signal solution iteratively until it has reached a point where it cannot be improved upon anymore. Of these three representations, NSGA-II was chosen because it has more established research around it, it broke down easily into distinct functions, there was a clear way to experiment to improve the model and the group's familiarity with evolutionary algorithms.

Experimental Process

We conducted a series of experiments on the smallest of the nine datasets to find the optimum parameters. In each experiment, all other variables were fixed, whilst one was changed. This allowed us to see the effect of that variable in isolation and, based on the results, pick the correct parameter. First we tested different population sizes of 20,40,60,80 and 100. These results can be seen in Figure 1. The better result for each set of experiments is the curve closest to the graph's bottom left corner. For the population size, 40 produced the best Pareto front, with 100 a close second. Next, we varied Tour Crossover types and found that OX1_single_point performed the best by a considerable margin. We then varied the packing plan initialisation methods, and pack_optimal performed the best. Finally, we varied tour and packing plan mutations. Single swap performed better for the tour mutation and 25% bit flipping for the packing plan. This led us to the set of parameters: population size of 40, packing plan initialisation using packoptimal, tour crossover of OX1_single_point, tour mutation of single swap, and packing plan mutation of 25% bit flip.

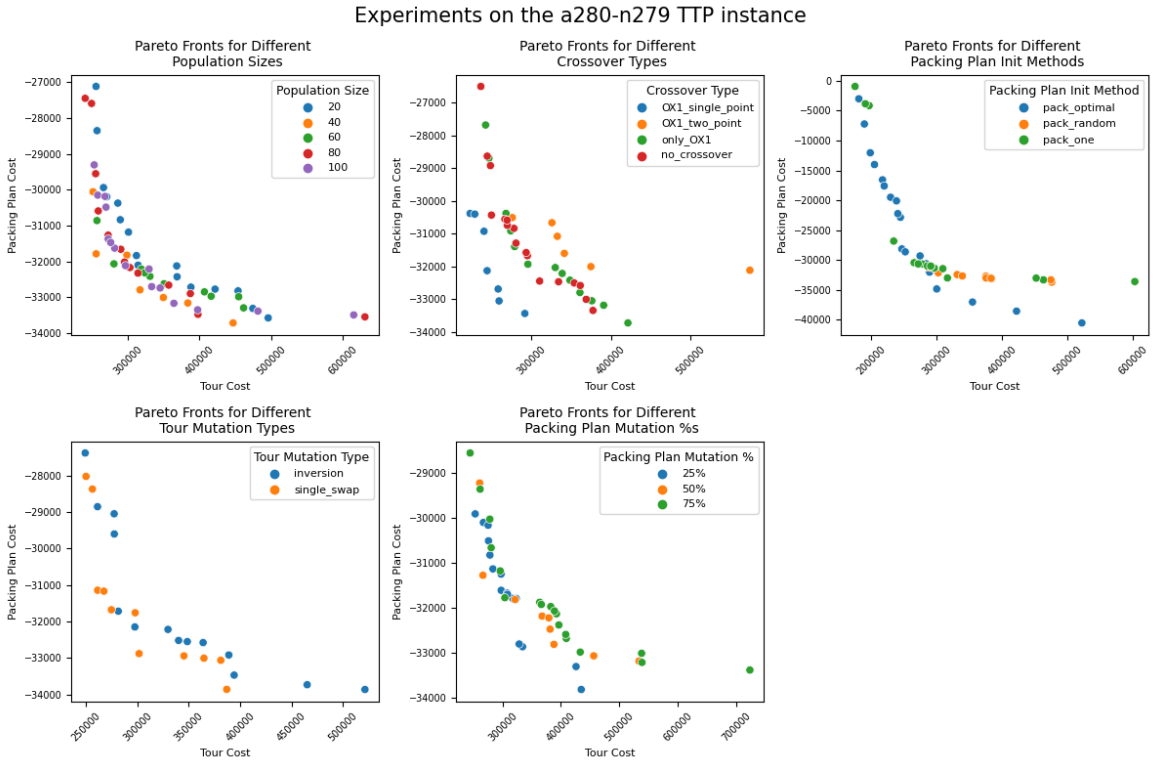

Figure 1: Pareto front of each of the different experiments with each of the same type of parameter plotted on the same graph for easy comparison.

The termination criteria for the smaller instances was completing 100 generations. However, this took too long for the larger datasets, so the termination criteria was reduced to 20 generations. I did attempt to improve the time complexity of the code after the first run of one of the larger datasets but could not make significant improvements in the timeframe to reduce overall runtime because it would

have required a significant redesign of several key functions.

Teamwork Process

The team did not make full use of the resources available. This is because some team members could not contribute as much as others due to other commitments. Additionally, some members did not have sufficient prior knowledge of programming or algorithms and teaching them these skills was beyond the project's scope, given the timeframe. This led to an imbalance in contributions within the team, with Jack and I taking on the majority of the coding for this project. Another key issue was our choice to program in Python. The code written was not optimised for efficiency, so it took over 50 hours to complete 20 generations on the largest dataset. A better approach would have been implementing multi-threading and generating the children concurrently instead of sequentially. Writing this program in C or Java would have been more effective as they are much faster for larger problems. However, this was impossible because only two group members had previously coded in an object-oriented language.

One of the most difficult challenges of the project was assigning tasks that matched each group member's capabilities and still split the work evenly. The project easily broke down into functions that could be assigned to be worked on in parallel. This would have been an effective and fast approach; however, due to the aforementioned lack of coding abilities, it meant the speed at which each person could produce a working function varied greatly. This meant those more familiar with coding wrote a lot more, resulting in the less confident members not understanding how to integrate other functions into their code or how it operated despite demonstrations. This had a knock-on effect as they could not produce experiments due to their lack of understanding, so experiments had to be written for four of the team members.

Despite the challenges, there were enjoyable parts of the teamwork process. I enjoyed the ability to lead meetings so that the project stayed on track. We had weekly meetings that gave every member accountability and support where necessary. I was also given the opportunity to teach my team members new skills through demonstrations, such as how to commit to a GitHub repository, access elements of a data structure, and cite academic papers correctly. I wanted to create a positive environment and regularly asked if anyone was struggling with anything or needed help. This meant we could help each other when problems arose so we could be much more productive.

Contributions

Each team member attended most, if not all, of the meetings and contributed to the project. Table 1 shows the lines of code committed by every member to the code base.

Team Member	Ursula	Jack	Nafees	Kaiyuan	Swarwoop	Kanchan
Lines of code	22,092	37,560	1,390	1,295	333	305

Table 1: Line of code committed to GitHub repository for each team member, including data generation.

Overall, Jack was an exemplary group member, without which this project would not be possible. His contributions and competency allowed the project to proceed in a timely manner. He wrote 14 of the 22 functions required to execute the evolutionary algorithm. He wrote and ran all the experiments for the smallest dataset and created the final results code Nafees, Kaiyuan, Swarwoop, and Kanchan ran. Were we to do the project again, the only thing he would need to do differently would be to focus on the time complexity of the functions he is writing.

Nafees wrote one of the EA functions and the first section of the group report. He ran the final results algorithm created by Jack for a dataset. He communicated well but did not have much experience writing technical reports, so his section had to be proofread, and several rounds of revisions were made.

Kaiyuan began the project by implementing a method to extract the data into data structures. However, the code was inefficient and could not be used to initialise the larger datasets. He implemented one of the functions used in the EA and ran the final results algorithm created by Jack for a dataset. He effectively tested his code but struggled to articulate what he had done.

Swarwoop produced one of the functions for the EA and ran the final results algorithm created by Jack for a dataset. His code was well-tested and performed as expected. Due to other time commitments, he could not contribute to the project as much as other members but ensured all the work he produced was of good quality.

Kanchan was the weakest coder of the group, stating at the start she had never programmed in python before. However, with help from other members, she produced a function for the EA based on the code she had written for the individual project. She ran the final results algorithm created by Jack for a dataset and volunteered to write the results section of the group report; unfortunately, this had to be rewritten due to her lack of understanding. She knew when to ask for help and took on constructive feedback well.

Personal Contributions

I contributed to the team by leading and organising every meeting. I made sure to confirm everyone's availability, and always have another meeting set up every week to keep the project on track and confirm that everyone had a task and knew what they were doing. After the first meeting, I set up a GitHub repository and group latex document for the team to work with. I supported other members, completing one-on-one meetings to teach them how to commit code to GitHub and demonstrating other skills, such as referencing, in group meetings. I checked in on everyone midweek to see if they had any issues that another member or I could help resolve. After every meeting, I wrote up minutes and made them available to all the team members, along with a list of tasks to complete for the next week. I checked the code or written work of anyone who requested it, providing constructive feedback and praise to help them improve.

In addition to that, I wrote the code to determine the ranks for the candidate solutions using the non-dominated sort proposed by A Seshadri [9] and the associated crowding distance. I found this challenging but enjoyable and spent some time decomposing the pseudocode to understand and implement an effective function. After that, I began to test both functions, first on small ten cities and ten items datasets, and then when it worked as expected, I tested the functions on data from the used datasets. I verified the results mathematically and using MatPlotLib figures which showed the fronts in different colours. This meant that when I had to explain my functions to other team members, I could show the graphs and quick examples to illustrate my point.

After that, I completed helper functions such as functions to get the Pareto front of a solution set and functions that would graph the results. I picked up any of the functions that weren't complete and completed them, and I debugged the algorithm as a whole, looking to improve efficiency and check all of the functions integrated as intended.

I completed the final algorithm runs for the three largest datasets: pla33810_n33809, pla33810_n169045 and pla33810_n338090. Due to their size, each dataset took 20 to 50 hours to complete 20 generations. For the smallest of the three, I used my laptop and ran it overnight. However, this was not the best use of resources, so I used the high-performance computers on campus for the other two datasets, allowing the program to finish faster. Whilst doing this, I went through the finished code and attempted again to improve efficiency, making some small changes such as initialising static variables outside loops which would make small improvements to the time complexity.

I wrote the 'Breaking the Problem Down' section of the group report. Collating information from GitHub and meeting minutes to concisely explain the division of the algorithm into parts and who completed them. I also produced the evolutionary algorithm figure for the 'Details of the Developed Algorithm' section to make the algorithm implemented clearer. Finally, I proofread and edited the whole document, fixing any grammatical errors and suggesting improvements or updates to other members.

Conclusion

Overall, the project was successful, and the evolutionary algorithms improved the initially generated solutions for all nine datasets. Through experimentation on the smallest dataset, we can conclude empirically on the best parameters for the evolutionary algorithm. These parameters were then used to generate solutions for the other eight datasets. The program's greatest weakness was the run time for large datasets. Good communication was demonstrated throughout the project by effectively utilising weekly meetings and Microsoft Teams. The main challenge of the project was assigning tasks to every member that fit their skill set and time requirements. This led to Jack and I taking on much more of the coding than the other members. In the future I would definitely consider programming the children generation to be multi-threaded and picking a language with a faster computational speed to avoid the algorithm taking over 50 hours to run on the largest instance.

References

[1] M. R. Bonyadi, Z. Michalewicz, and L. Barone, "The travelling thief problem: The first step in the transition from theoretical problems to realistic problems," in *2013 IEEE Congress on Evolutionary Computation*, pp. 1037–1044, 2013.

[2] J. Blank, K. Deb, and S. Mostaghim, "Solving the bi-objective traveling thief problem with multi-objective evolutionary algorithms," in *Evolutionary Multi-Criterion Optimization: 9th International Conference, EMO 2017, Münster, Germany, March 19-22, 2017, Proceedings 9*, pp. 46–60, Springer, 2017.

[3] M. Wagner, "Stealing items more efficiently with ants: a swarm intelligence approach to the travelling thief problem," in *Swarm Intelligence: 10th International Conference, ANTS 2016, Brussels, Belgium, September 7-9, 2016, Proceedings 10*, pp. 273–281, Springer, 2016.

[4] M. El Yafrani and B. Ahiod, "Population-based vs. single-solution heuristics for the travelling thief problem," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, (New York, NY, USA), p. 317–324, Association for Computing Machinery, 2016.

[5] W. Zouari, I. Alaya, and M. Tagina, "A new hybrid ant colony algorithms for the traveling thief problem," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 95–96, 2019.

[6] C. Wachter, *Solving the travelling thief problem with an evolutionary algorithm.* PhD thesis, Wien, 2015.

[7] H. Ali, M. Z. Rafique, M. S. Sarfraz, M. S. A. Malik, M. A. Alqahtani, and J. S. Alqurni, "A novel approach for solving travelling thief problem using enhanced simulated annealing," *PeerJ Computer Science*, vol. 7, p. e377, 2021.

[8] J. R. Figueira and E.-G. Talbi, "Emergent nature inspired algorithms for multi-objective optimization," *Computers & Operations Research*, vol. 40, no. 6, pp. 1521–1523, 2013.

[9] A. Seshadri, "A fast elitist multiobjective genetic algorithm: Nsga-ii," *MATLAB Central*, vol. 182, pp. 182–197, 2006.