Project Work III. (2024/25 I. semester)
Dynamic Vehicle Routing Problem

December 15, 2024

Dávid Apagyi
apagyidavid@student.elte.hu

Advisor:

Markó Horváth

Introduction

The Dynamic Vehicle Routing Problem (DVRP) involves situations where new delivery requests are constantly being added, requiring vehicle routes to be adjusted in real time. Examples of such scenarios include transporting goods between factories or delivering meals to customers.

In these cases, seeking a solution that is only statically optimal may not always be the best approach, as it may lack the flexibility needed to handle dynamic changes. Instead, a more adaptive schedule is often required, one that can efficiently accommodate new requests while minimizing delays.

During the semester, I focused on a detailed study of a competition problem previously announced by Huawei. By the end of the semester, as part of this work, I implemented a natural algorithm to address the challenges posed by the problem. Although this algorithm can work quite well, it is resource intensive and does not explicitly consider the above aspects.

About the Dynamic Vehicle Routing Problem

The Dynamic Vehicle Routing Problem (DVRP) is a variation of the classical Vehicle Routing Problem (VRP) where changes happen in real-time. In DVRP, new delivery requests or updated traffic conditions can occur while vehicles are already on their routes. The goal is to adjust the routes dynamically to handle these changes efficiently. This problem is critical for industries like logistics and transportation, where real-world challenges make static planning ineffective.

Huawei Competition

Huawei and ICAPS organized an international competition focusing on a DVRP challenge [1]. The advisor of this report, along with several colleagues from SZTAKI, participated in this competition and achieved third place. Throughout this report, we refer to their experiences and work [2].

During the semester, this project focused specifically on solving this competition problem. However, we plan to extend this work to explore other related problems in the future.

Input

We follow the notation system from reference [1]. The input consists of the following:

- A directed graph $G = (F, A)$, where F is the set of factories, and A is the set of arcs connecting the factories. Each arc has a transportation time t_{ij}.
- An order set $O = \{o_i : i = 1, ..., N\}$, where each order $o_i = \left(F_p^i, F_d^i, q^i, t_e^i, t_l^i\right)$ specifies:
  - F_p^i and F_d^i: the pickup and delivery locations.
  - $q^i = \left(q_{\text{strandard}}^i, q_{\text{small}}^i, q_{\text{box}}^i\right)$: the size of the order in pallets and boxes.
  - t_e^i: the creation time of the order.
  - t_l^i: the committed completion time.
- A fleet of vehicles $V = \{v_k : k = 1, ..., K\}$, each with a loading capacity and specific shift times.
- M nodes (factories), where each factory has limited cargo docks and work shifts. Vehicles may need to wait if all docks are busy.

Constraints

The problem must satisfy the following constraints:

1. **Order fulfillment:** All orders must be served.
2. **Completion time:** Orders must be completed before their committed times t_l^i.
3. **Order splitting:** Orders cannot be divided across multiple vehicles unless specified.
4. **Vehicle capacity:** No vehicle can exceed its loading capacity.
5. **Work shifts:** Loading and unloading must occur within shift times. (*We do not take this restriction into account.*)
6. **Dock limitations:** Each factory has limited docks, and vehicles follow a first-come, first-serve rule.

Implicit Constraint Assumptions

There are also some hidden constraints in the problem that are not explicitly mentioned in the problem statement, but are assumed during the validation process:

1. **Order delivery time:** An order is considered delivered when the vehicle arrives at the delivery location, regardless of whether the order has been unloaded or not. This simplifies the validation criteria, but may differ from real-world scenarios where unloading is part of the process.
2. **Handling of oversized orders:** For orders that exceed the maximum capacity of a single vehicle, splitting is allowed. To simplify this process, a greedy approach is assumed: the items in the order are sorted by size in descending order, and items are grouped sequentially until they reach

the maximum capacity of the vehicle. This ensures that oversized orders are distributed across multiple vehicles in a straightforward manner, avoiding complex optimization during splitting.

Objectives

The problem has two main objectives:

1. Minimize the total delay of orders:

$$f_1 = \sum_{i=1}^{N} \max\left(0, a_i^d - t_i^l\right),$$

   where a_i^d is the arrival time of order o_i, t_i^l is the committed completion time, and N is the total number of orders.

2. Minimize the average travel distance of vehicles:

$$f_2 = \frac{1}{K} \sum_{k=1}^{K} \sum_{i=1}^{l_k - 1} d_{n_i^k, n_{i+1}^k},$$

   where n_i^k is the i-th node in the route of vehicle v_k, $d_{n_i^k, n_{i+1}^k}$ is the distance between consecutive nodes, and K is the total number of vehicles.

The overall objective function is:

$$f = \lambda \cdot f_1 + f_2,$$

where λ is a large positive constant to prioritize minimizing delays. In the validator, it is fixed as $\lambda = \frac{10000}{3600}$.

Simulation framework

The advisor of this project, Markó Horváth, developed a simulation framework in Python using the SimPy library. This framework was released during the semester [3], but we had the opportunity to use it while it was still under development. This framework was used as the basis for implementing the algorithms described in this report.

The framework is designed for general purposes and can handle many related problems beyond the specific focus of this report. A significant amount of time was spent learning how to use this framework effectively, as it plays a central role in testing and evaluating the algorithms.

This framework allows for decision points to occur periodically (every 10 minutes in the given task), during which it provides the current state of the simulation to an external (third-party) algorithm. The provided information includes details about the vehicles and newly received orders. The external algorithm processes this data and returns a decision for the next step.

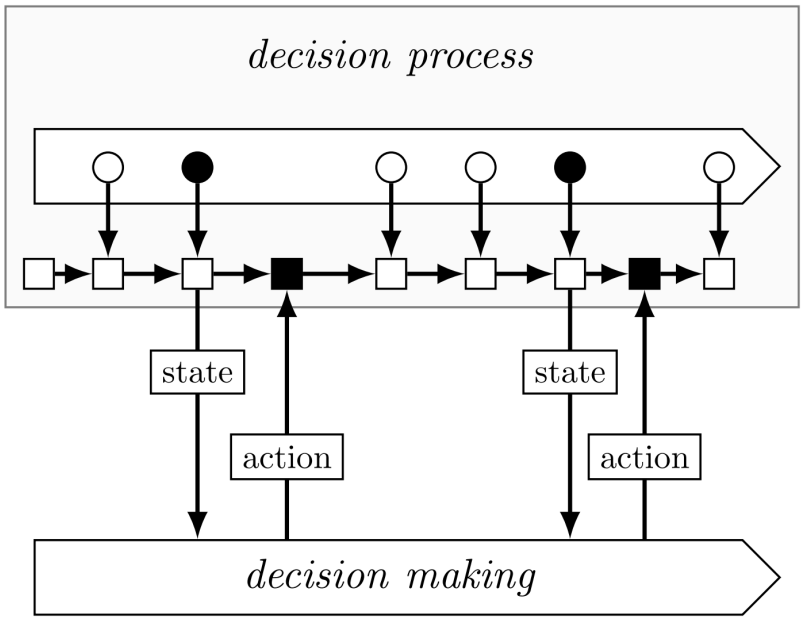

Figure 1: Illustration of the interaction between the simulation framework and the routing algorithm as described in [3].

This structure makes it possible to implement the routing algorithm in any programming language by using JSON files as the communication protocol. For this project, however, we chose to continue using Python for implementing our algorithms.

Algorithms

Naive algorithm used for benchmarking

The competition organizers provided a simulation framework along with a simple example algorithm. This simple algorithm assigns requests to vehicles in a round-robin fashion, cycling through the vehicles while respecting the constraints. However, this approach typically results in inefficient schedules with suboptimal performance.

The details of this algorithm are not discussed further in this report, as it is used only as a baseline for comparison with the algorithms implemented in this project.

Best Insert Method

The best insert method is a heuristic designed to produce better schedules by assigning orders to vehicles. Instead of assigning orders randomly or cyclically, this method evaluates all possible inserts of a new order into the current schedule of each vehicle.

We go through the newly available orders one by one and check where each can be inserted into the existing schedule. For each possible insertion, it runs an embedded simulation to calculate the value of the objective function and selects the insertion that results in the most optimal solution.

This part of the implementation has the potential for parallelization, but it has not been implemented at this stage. While this is more computationally intensive than cyclic assignment, the Best Insert method is more effective at producing schedules that better meet the objectives of the problem.

Key components of the implementation

Simplifying the Decision

The `simplify_decision` and `expand_decision` functions are designed to streamline the decision making process. A vehicle route consists of "visits" that determine which orders will be picked up and delivered at a particular location. We always start with deliveries, followed by pick-ups to ensure that last in first out (LIFO) constraints are met.

This module combines consecutive visits to the same location (with a few simple checks) into a single visit. Without this, vehicles would unnecessarily dock, queue, and then dock again, leading to inefficiencies in the simulation.

Route Generation

This component is responsible for generating potential insertion points for the orders. These points are used in the subsequent evaluation phase discussed in the next section.

The task is to add a new interval to a laminar interval system, taking into account the LIFO constraints. Two pointers are maintained to track where we want to insert the pickup and delivery visits, and these pointers are moved as far as possible. In addition to respecting the LIFO constraints, the capacity limits are also taken into account during this step.

Embedded decision evaluation

In addition to the available simulation framework, an evaluator was needed to evaluate the decision candidates based on the cost functions defined for the task. Essentially, we simulate the future operation to evaluate these decisions. This is done by continuously tracking when each vehicle's state will next change, and we focus on the vehicle with the earliest state change.

This component consists of two main phases: In the first phase, we initialize the next state changes for the vehicles based on their current states. In the second phase, we run the simulation.

Results

We tested the above methods on some smaller input instances.

ID	Number of orders	Naive algorithm	Best insert algorithm
#17	300	228.9	72.4
#18	300	238.0	74.4
#19	300	246.3	89.1
#22	300	237.6	86.2
#25	500	414.7	127.3
#26	500	431.7	128.5
#27	500	389.3	113.5
#28	500	345.2	99.9

Table 1: Comparison of the results between the naive algorithm and the best insert algorithm on smaller instances shows a clear difference in performance. It is important to note that the metrics used represent the cost of the routing algorithm, meaning that lower values indicate better performance.

Conclusions

As expected, the simple algorithm performed significantly better than the naive approach. On the instances tested, **it consistently found better solutions, with an average cost reduction of 68.79%.** However, a deeper analysis of the simulation results shows that on these smaller instances, the vehicles rarely, if ever, had to wait. In contrast, based on the experience of this report's advisor, wait times become critical for larger instances. Their third-place approach in the competition focused on optimizing these wait times.

For instances that require waiting, further analysis and simulations are needed to explore potential improvements.

Future work

One possible direction for improvement is to implement local search methods. This approach has already been tested, at least for instances without waiting times, as part of an individual project (see the work of I. Hatala [4]). We expect that at the beginning of scheduling, when only a few orders have been received, a very well-optimized schedule can make a big difference. In discussions with Markó Horváth, based on their experience in the competition, they found that after a few iterations, local search did not provide significant improvements to their objective function.

Another possible idea, similar the approach of M. Horváth et al. [2], is to modify the objective function to produce better schedules. This could involve adding additional terms with carefully chosen weights. These adjustments could help regularize the solution, making it more flexible and better prepared for future changes.

Another interesting direction is to define policies. By this we mean simple rules that make intuitive sense. For example, to reduce waiting times, we could limit the number of vehicles assigned to a

single location and dynamically adjust that limit based on past requests. This kind of approach could also open up many possibilities for experimentation.

Beyond this individual project, I will continue to work on these ideas as part of my master's thesis.

Bibliography

[1] J. Hao *et al.*, "Introduction to The Dynamic Pickup and Delivery Problem Benchmark - ICAPS 2021 Competition," *CoRR*, 2022, [Online]. Available: https://arxiv.org/abs/2202.01256

[2] M. Horváth, T. Kis, and P. Györgyi, "A cost function approximation method for dynamic vehicle routing with docking and LIFO constraints." [Online]. Available: https://arxiv.org/abs/2405.01915

[3] M. Horváth and T. Tamási, "A general modeling and simulation framework for dynamic vehicle routing." [Online]. Available: https://arxiv.org/abs/2411.12406

[4] I. Hatala, "Dinamikus jármű útvonaltervezés [Dynamic Vehicle Routing]." [Online]. Available: https://math-projects.elte.hu/media/works/343/report/beszamolo.pdf

[5] N. Soeffker, M. W. Ulmer, and D. C. Mattfeld, "Stochastic dynamic vehicle routing in the light of prescriptive analytics: A review," *European Journal of Operational Research*, vol. 298, no. 3, pp. 801–820, 2022, doi: https://doi.org/10.1016/j.ejor.2021.07.014.

[6] M. W. Ulmer, J. C. Goodson, D. C. Mattfeld, and B. W. Thomas, "On modeling stochastic dynamic vehicle routing problems," *EURO Journal on Transportation and Logistics*, vol. 9, no. 2, p. 100008, 2020, doi: https://doi.org/10.1016/j.ejtl.2020.100008.

[7] X. Li *et al.*, "Learning to Optimize Industry-Scale Dynamic Pickup and Delivery Problems," *CoRR*, 2021, [Online]. Available: https://arxiv.org/abs/2105.12899