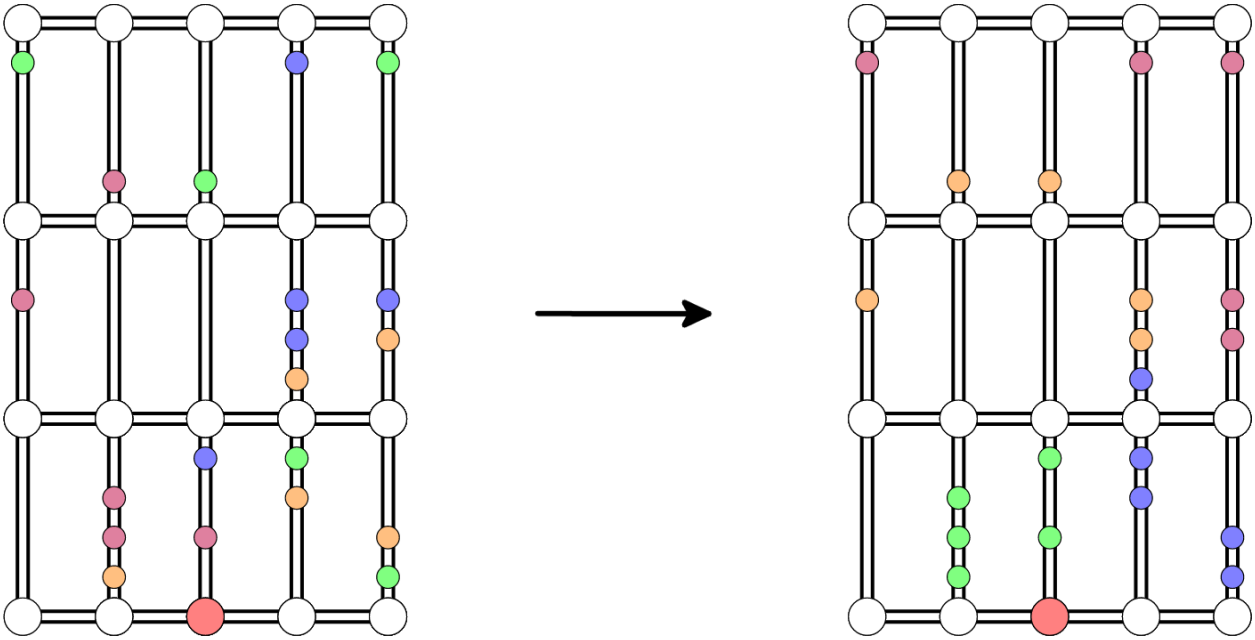
CHALMERS
UNIVERSITY OF TECHNOLOGY



Item batching for picker routing in warehouses

Applying column generation with an exact, novel subproblem algorithm

Master's thesis in Complex Adaptive Systems

Joseph Löfving

DEPARTMENT OF MATHEMATICAL SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

Master's thesis 2023

Item batching for picker routing in warehouses

Applying column generation with an exact, novel subproblem algorithm

Joseph Löfving

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mathematical Sciences
Chalmers University of Technology
Gothenburg, Sweden 2023

Item batching for picker routing in warehouses
Applying column generation with an exact, novel subproblem algorithm
Joseph Löfving

Supervisor: Ann-Brith Strömberg, Department of Mathematical Sciences
Advisor: Johan Härdmark, Ongoing Warehouse AB
Examiner: Ann-Brith Strömberg, Department of Mathematical Sciences

Master's Thesis 2023
Department of Mathematical Sciences
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An example of item batches in a warehouse, without route optimization, and with route optimization.

Typeset in LaTeX
Gothenburg, Sweden 2023

Item batching for picker routing in warehouses
Applying column generation with an exact, novel subproblem algorithm
Joseph Löfving
Department of Mathematical Sciences
Chalmers University of Technology

Abstract

The item batching problem concerns finding an optimal partitioning of items waiting to be picked in a warehouse into batches, such that these batches can be picked with minimal total walking distance. In this project, the item batching problem is solved through column generation, an efficient method for solving linear optimization problems with an enormous number of variables (which, in this case, represent possible batches). In order to achieve optimality through column generation, one needs to be able to solve a problem-specific subproblem to optimality. For the item batching problem, this subproblem is a version of the computationally intensive prize-collecting travelling salesperson problem. To solve this subproblem to optimality, a novel dynamic programming algorithm with a heuristic guide is developed, which solves the subproblem optimally in times ranging from milliseconds to seconds, depending on the details of the problem instance.

Overall, the implemented column generation algorithm finds optimal solutions in a matter of seconds per generated batch for all test cases, but sometimes requires hours to verify that these solutions are optimal. The optimal solutions provide great improvements over current batching strategies, often more than halving the total distance that the warehouse workers need to walk while picking. However, simple, alternative algorithms provide similar improvements in a matter of milliseconds, making the column generation time-consuming, computationally intensive, and unwieldy for very little gain when compared to the simple, alternative algorithms.

The developed subproblem algorithm is easily generalizable, and could be used for more complicated variants of the item batching problem, like the order batching problem, where the simple, alternative algorithms are less likely to perform well.

Keywords: Item batching, order batching, warehouse, column generation, dynamic programming, prize-collecting travelling salesperson, set covering, picker routing problem.

Acknowledgements

I would like to express my gratitude to those who have helped me with this project, both directly and indirectly. In particular, I would like to thank Ongoing Warehouse for the opportunity to write this thesis, and my colleagues at Ongoing, with special thanks to Johan Härdmark and Martin Sigvardsson, for their help with the project.

I would also like to thank my academic supervisor, Ann-Brith Strömberg, for her help with the theory, and for her ideas.

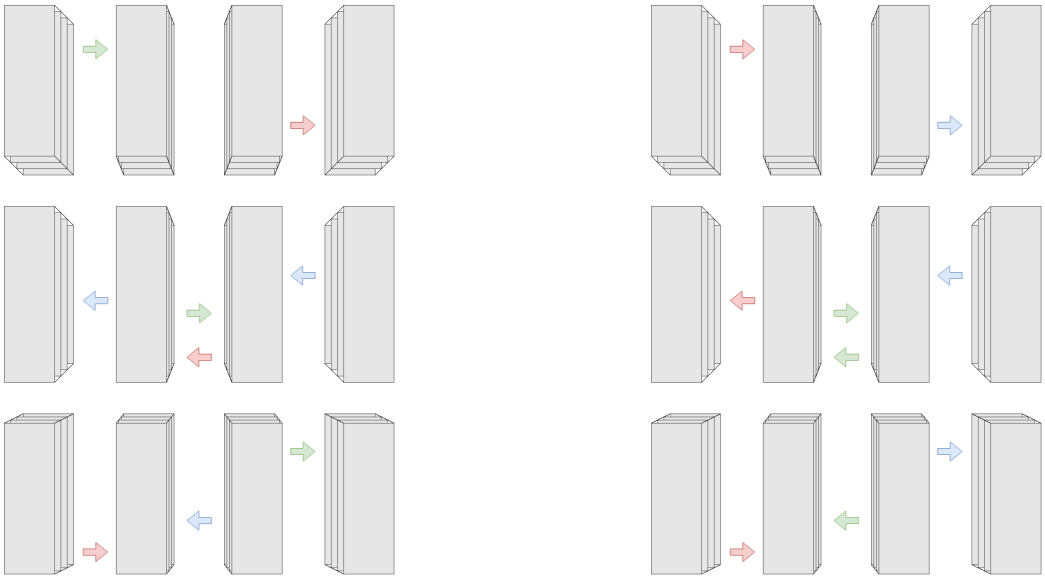
Joseph Löfving, Gothenburg, 2023-10-15

Contents

1 Introduction

In this project, the issue of *item batching* will be investigated. Item batching refers to the problem of optimizing the partitioning of a set of items waiting to be picked in a warehouse into what's known as *batches*, such that the total distance one needs to walk to pick all items in each batch, one batch at a time, is minimized. An example of the effect of optimized item batching is presented in Figure 1.1.

This project is done in collaboration with Ongoing Warehouse, a company that provides warehouse management software.



(a) Unoptimized batches.

(b) Optimized batches.

Figure 1.1: Examples of unoptimized and optimized batching. The images show top-down views of a simple warehouse with nine items spread out over twelve different shelves, with items indicated by arrows. The colour of an arrow indicates which batch it belongs to.

1.1 Overview of the problem

The process of collecting items is one of the most costly operations overall in a warehouse, and has been identified by many as being ripe for cost reductions through productivity improvements [1], with item batching being one promising candidate.

Item batching is, however, not an easy problem, since there are exponentially many ways to partition a set into what we here call batches, and, in order to know how

good a batch is, one needs to find the optimal route for picking the items in said batch, which is a computationally intensive problem in and of itself.

1.1.1 The batching process

In this project, we will investigate the potential for optimized item batching in a simple case with very few constraints; the only constraint imposed on our solutions will be that every item is to be picked by exactly one batch, and that the batches must each contain a specific number of items. This number of items relates to the maximum number of items one can fit into one of the trolleys used while picking, the *trolley capacity*.

Many warehouses use item picking workflows that would require additional constraints, such as requiring certain items to be picked in the same batch (typically, items belonging to the same order are to be picked in the same batch), or having one-way policies in certain shelf aisles, only allowing movement in one direction therein. Such additional constraints are not considered explicitly here, but we will see that the final implementation can be modified slightly to support such constraints.

This problem with its few constraints mirrors the workflow employed in the warehouse of one of Ongoing Warehouse's customers. This warehouse is operated by an online book retailer, and real-world data from that warehouse will be used to evaluate the column generation implementation developed in this project.

The warehouse of study is divided into eight zones, and each warehouse worker works within only one zone. A conveyor belt runs past all zones, leading to either a sorting area—where items that were ordered together with other items are sorted before packaging—or to a packaging area, where items that were ordered by themselves are packaged immediately. Item batches are currently being used within the warehouse, and to accommodate the setup described, they come in two forms: one consisting entirely of items that were ordered by themselves ("lone items"), and one consisting entirely of items ordered together with other items ("non-lone items").

The batching currently used in the warehouse is not optimized for the minimization of walking distance, beyond the fact that each batch only contains items within one warehouse zone; apart from that, no geographical information about the items is used. Within each zone, batches are created based only on a priority value for each item waiting to be picked, which in turn is based primarily on how much time is left until the promised delivery date for said item. The resulting batches often consist of items spread out over the whole zone.

The lone items are picked in batches of 20 items, and the non-lone items are picked in batches of five items. This difference has to do with the fact that non-lone items are picked in crates with compartments, where each item is placed into an assigned compartment in the crate to facilitate sorting, while lone items are picked in crates without compartments, since they do not need to be sorted after the picking.

The picking process starts when a worker picks up an empty crate from the conveyor belt, scans it with a handheld device and gets assigned to a batch. The device displays the shelf number that the worker needs to visit to pick the first item in the batch. The worker goes to the shelf, scans the item, and the device then displays the shelf number for the next item. This process repeats until all items in the batch

have been picked. The full crate is then placed on the conveyor belt and is delivered to the sorting area, or the packaging area. The order in which items are picked in the batch is based on the priority value of the shelf for each item; an item on a shelf with a higher priority value is picked before an item on a shelf with a lower priority value. These shelf priority values are static and predefined.

This kind of picking workflow is what's known as a *pick-and-sort*, since the picking and sorting are handled separately [2]. Such a system allows for more freedom when creating batches compared to more typical picking systems in which each item must be picked in the same batch as all other items in its order, but comes with some drawbacks: for example, a sorting area tends to have a limited number of slots, where each order is assigned a slot while its items are being picked. If too many orders are partially picked, the sorting area may run out of free slots, and the warehouse may need to pause operations to resolve the issue manually.

1.1.2 The optimization problem

The naive way to optimize the batches would be to check every possible partition of items that are waiting to be picked and to choose the partition that results in the shortest calculated total route length. This method, however, quickly becomes intractable; For a problem instance with N items and trolley capacity t, there are $B = \binom{N}{t}$ possible batches, which quickly grows out of hand as N and t increase.

In order to solve this problem more efficiently, one can formulate it as a mixed integer linear optimization problem (MILP) and use column generation to solve the problem efficiently without having to enumerate all possible batches. With column generation, one creates an initial solution (in our case, this could be done by creating batches according to the priority of the considered items, as is currently done in the warehouse of study), and iteratively computes better solutions with more precise lower and upper bounds on the optimal value by solving a subproblem tailored to the problem at hand [3].

1.2 Aim

This project aims to explore the possibility of reducing picking times by generating batches of items with minimal total walking distance using column generation. The intended result is a proof of concept for this method made up of implementations for the two workflows employed by the warehouse of study, with a focus on the development of an algorithm to solve the item batching column generation subproblem to optimality, which is necessary to really achieve the optimal solution to the main problem, among other things.

The implementations will be evaluated by solving item batching problem instances generated from real-world data from the warehouse discussed above. These evaluations will be based on the average total distance required to pick the batches that the implemented algorithms generate, compared to the corresponding value for the batches created by the current implementation in the warehouse in question, and the time required to perform this optimization.

1.3 Contributions

During the course of this project, two unique algorithms for solving the item batching column generation subproblem (column generation subproblems are explained in Section 2.1) were developed. One is what we call the *stair-based subproblem algorithm*, a relatively simple extension of common dynamic programming algorithms for solving the knapsack problem. The stair-based subproblem algorithm quickly generates multiple solutions to the subproblem, but with no guarantees for optimality. It is presented in Section 3.2.

The other subproblem algorithm that was developed is called the *reduced cost-optimal subproblem algorithm*. As the name suggests, this algorithm is guaranteed to generate the solution with the optimal reduced cost to the subproblem. As explained in Section 2.1, having at least one subproblem algorithm with this property is important for solving a problem to optimality when using column generation. The reduced cost-optimal subproblem algorithm is a generalization of a dynamic programming algorithm for solving the picker routing problem—in which one wants to find the optimal route for picking all items in a given batch—developed by Pansart, Cambazard and Catusse [4]. Their algorithm is in turn based on previous work by Cambazard and Catusse [5] and Ratliff and Rosenthal [6]. The reduced cost-optimal subproblem algorithm is presented in Section 3.3.

1.4 Limitations

In this project, the goodness of a batch will be based solely on the calculated route length. In reality, what matters is time, which, for example, is influenced by how easy it is for the workers to follow their picking routes (less intuitive routes may require the worker to stop between every picked item and orient themselves), but such effects are not considered here.

Ideally, one would want to optimize the batching based on all items that are waiting to be picked at the time of the batching, but this would often be very computationally intensive, and will not be done. Instead, the batches will be created from the N items with the highest priority, as described in Subsection 1.1.1, where N is some predefined problem size.

As mentioned in Subsection 1.1.1, one may run into problems with the sorting area "overflowing" if too many orders are partially picked. This problem will not be considered explicitly here; it will be assumed that this can be dealt with by choosing an appropriately small value for N (setting $N = t$, for example, gives the current batching method, which we presume doesn't have these overflow problems), making modifications to how items are prioritized or choosing in which order to pick the generated batches.

1.5 Previous work

During the course of this project, no previous work on the exact problem at hand has been found, but there are other, similar problems that are relevant.

One relevant problem is the *picker routing problem* (PRP), which we already touched on in Section 1.3. In the picker routing problem, one aims to minimize the total

walking distance for a given batch of items to be picked in a warehouse, with known item locations and warehouse layout. The PRP is essentially the classical travelling salesperson problem (TSP), in which one wishes to find the shortest tour in a graph that visits all nodes (which represent items to pick in the PRP), but with the difference that the simple geometry of warehouses makes the PRP easier than more general TSPs. Richard Bellman proposed a dynamic programming algorithm in 1961 that solves the TSP exactly with a time complexity of $\mathcal{O}(N^2 2^N)$, where N is the number of nodes, or items to pick, as above [7].

Pansart, Cambazard and Catusse developed a dynamic programming algorithm for the PRP in 2018 [4], which typically performs better than Bellman's TSP algorithm. Their work expands on previous work for the rectilinear TSP (RTSP) by Cambazard and Catusse in the same year [5], and for the PRP with certain warehouse layouts by Ratliff and Rosenthal in 1983 [6]. This algorithm iteratively creates longer and longer "partial picking tours" in the warehouse, eventually finding or discarding every possible solution, and then selects the best one. The algorithm runs in $\mathcal{O}(hv7^h)$, where h and v both relate to the number of aisles in the warehouse [4], making the algorithm run in $\mathcal{O}(1)$ for a fixed warehouse layout, regardless of the number of items.

The *vehicle routing problem*, which was first introduced under the name of the *truck dispatching problem* by Dantzig and Ramser in 1958 [8], also has similarities with the item batching problem. In the vehicle routing problem, one considers a fleet of vehicles and a set of customers, all expecting a delivery, and the goal is to optimize the partitioning of the customers such that each vehicle serves one such partition, with minimal total distance driven. The item batching problem is essentially a vehicle routing problem, but specially tailored methods for the item batching problem can do better than general vehicle routing problem solutions, once again due to the simple geometry of warehouses. For an overview of the vehicle routing problem, see [9].

The most closely related problem with a significant amount of previous research is the *order batching problem*. The order batching problem is exactly the same as the item batching problem, but with the constraint that all items in an order need to be picked in the same batch, a constraint we have previously mentioned in passing. This problem has likely received more attention due to the fact that most warehouses rely on this kind of batching, and because it is more difficult to solve well heuristically; the item batching problem can in theory be solved well by clustering algorithms or similar, while order batching problems are less amenable to such heuristic methods. For more information on the order batching problem, see [10] and [11].

1.6 Outline

The report starts with Chapter 2, the theory chapter, which describes the basics of the mathematics behind column generation, some graph theory relevant to the problem at hand with a focus on concepts specifically relevant to picking tours in warehouses, and a short explanation of the A* algorithm, which will be useful for solving the column generation subproblem efficiently.

After the theory chapter, Chapter 3, on modelling and implementation, explains the

specifics of how column generation is applied to the problem at hand, and the two subproblem algorithms that are used for the column generation are described.

The report then continues with results in Chapter 4, which shows how the developed column generation implementation compares to the algorithm currently in use in the examined warehouse in terms of average picking tour length and computation time. These results are then evaluated and discussed.

Finally, the report ends with Chapter 5, with conclusions and potential future work.

2 Theory

We begin with a general overview of column generation—the central idea behind our approach to item batching—in Section 2.1. We will look at the mathematical models used for column generation; the branching that is typically necessary when applying column generation to optimize a problem fully (Subsection 2.1.1), with a specific look at the kind of branching rule that we will apply to this specific problem (Subsection 2.1.1.1); and finally, we look at how to acquire upper and lower bounds for the optimal value during the column generation (Subsection 2.1.2), which can help reduce the branching search tree, and can be used to implement early termination criteria based on optimality gaps.

Column generation requires one or more *subproblem algorithms*, the performance of which are critical to the performance of the column generation as a whole. The central contribution of this project—an algorithm that finds an optimal solution to the subproblem—requires some theoretical background, which is presented in Section 2.2, where we look at a particular type of multigraph that represents a warehouse (Subsection 2.2.1); Eulerian cycles, and how they can be used to generate optimal picking tours, which is the goal of the subproblem algorithms (Subsection 2.2.2); how to apply the theory regarding Eulerian cycles to our graph representation in a structured and efficient manner on a subaisle-by-subaisle basis using what we call *transitions* (Subsection 2.2.3); how to view partial solutions to the problem of generating picking tours as *partial picking tours*, and how we can define *equivalent partial picking tours*, which in turn let us significantly reduce the number of possible solutions to explore for our subproblem (Subsection 2.2.4); and a method for obtaining a lower bound on the optimal picking tour length for a given batch (Subsection 2.2.5).

Finally, we conclude with Section 2.3, in which we take a brief look at the A* algorithm, a heuristically guided algorithm for shortest path problems, which we will use for the subproblem algorithm that finds an optimal solution.

2.1 Column generation

The following theory is largely adapted from [3], which the particularly interested reader is encouraged to read for more information about column generation.

Column generation is an approach mainly used for linear programs (LPs) with too many variables to be modelled directly. Consider a mixed-integer linear program

(MILP) of the form

$$z^\star := \min_{\lambda_k} \sum_{k\in\mathcal{K}} c_k\lambda_k\,, \tag{2.1a}$$

$$\text{s.t.} \sum_{k\in\mathcal{K}} a_{ik}\lambda_k \geq b_i\,, \qquad \forall i \in \mathcal{I}, \tag{2.1b}$$

$$\lambda_k \in \{0,1\}\,, \qquad \forall k \in \mathcal{K}\,, \tag{2.1c}$$

where $\mathcal{K}$ denotes the index set of all columns (so-called because they correspond to columns in the matrix representation of the system), where the columns are denoted by the subscript k. $\mathcal{I}$ the set of all constraints. We call this MILP the *master problem*, and we will use this formulation in our implementation, as described in Section 3.1.

If the number of columns in this system is very large, the problem becomes unwieldy and difficult to optimize. What's more, it may be computationally difficult to calculate the cost c_k of a column, its constraint coefficients a_{ik}, or it might be practically impossible to even just enumerate all columns.

If the number of columns isn't very large, but the problem is still intractable (due to having many constraints, say), we can rephrase it using *Dantzig-Wolfe decomposition* (introduced by Dantzig and Wolfe in 1960 [12]) and obtain an alternate formulation in which the number of columns is very large, but other aspects are (hopefully) more manageable (like the number of constraints).

The main idea behind column generation is to treat this problem by simply not considering every possible column. Instead, one starts with only a few columns, and iteratively adds new columns which improve the optimal value of the objective function to the model, until no such improving columns exist. We denote call this small set of columns *restricted column set* $\mathcal{K}' \subset \mathcal{K}$, which initially is just some set of columns for which the problem is feasible. We then define the so-called *restricted master problem* (RMP):

$$z^\star_{\text{RMP}} := \min_{\lambda_k} \sum_{k\in\mathcal{K}'} c_k\lambda_k\,, \tag{2.2a}$$

$$\text{s.t.} \sum_{k\in\mathcal{K}'} a_{ik}\lambda_k \geq b_i\,, \qquad \forall i \in \mathcal{I}, \tag{2.2b}$$

$$\lambda_k \in \{0,1\}\,, \qquad \forall k \in \mathcal{K}'\,. \tag{2.2c}$$

Furthermore, we define the LP relaxation of the RMP, which we abbreviate as RLP:

$$z^\star_{\text{RLP}} := \min_{\lambda_k} \sum_{k\in\mathcal{K}'} c_k\lambda_k\,, \tag{2.3a}$$

$$\text{s.t.} \sum_{k\in\mathcal{K}'} a_{ik}\lambda_k \geq b_i\,, \qquad \forall i \in \mathcal{I}, \tag{2.3b}$$

$$\lambda_k \geq 0\,, \qquad \forall k \in \mathcal{K}'\,. \tag{2.3c}$$

Similarly, we can define the LP relaxation of the master problem, with optimal value $z^\star_{\text{LP}}$ analogously, but with the full column set $\mathcal{K}$ rather than $\mathcal{K}'$.

Now, associate linear programming dual variables with the constraints in (2.3b), with u_i as the optimal value of the variable associated with constraint i. The *reduced cost* of a column k is then given by

$$\bar{c}_k := c_k - \sum_{i \in \mathcal{I}} a_{ik} u_i \,. \tag{2.4}$$

The reduced cost of a variable is, in a sense, the partial derivative of the objective function with respect to the variable; the change in the value of the objective function is proportional to the reduced cost if we increase the value of the variable associated with the column. This has two important consequences in our case.

Firstly, if we can identify a column with a negative reduced cost, we can introduce it into our restricted column set $\mathcal{K}'$ and expect $z^\star_{\text{RLP}}$ to decrease. Secondly, if we can prove that no column with a negative reduced cost exists in $\mathcal{K} \backslash \mathcal{K}'$, we know that no new column can be added to $\mathcal{K}'$ in order to decrease $z^\star_{\text{RLP}}$, meaning we have found a minimum for the LP relaxation of (2.1), and as such, we know that $z^\star_{\text{RLP}} = z^\star_{\text{LP}}$, and we can stop generating new columns. We call a column set $\mathcal{K}'$, for which there are no columns with negative reduced cost, *optimal for the RLP*.

Finding columns with a negative reduced cost is what we refer to as the *subproblem*. The subproblem takes the optimal values of the dual variables as input, and tries to generate one or more columns for which the reduced cost is negative. Exactly how to best solve the subproblem, and what the generated variables represent is highly problem specific; in the item batching problem, each column k represents one possible batch of items, with optimal tour length to pick those items c_k, and $a_{ik} = 1$ if item i is in the batch, and 0 otherwise. As such, solving the subproblem entails simultaneously optimizing the choice of items to include in the batch (to maximize $\sum_{i \in \mathcal{I}} a_{ik} u_i$) and the choice of route to pick those items (to minimize c_k). This type of problem is known as the *prize-collecting travelling salesperson problem*, and was introduced by Balas in 1989 [13].

Proving that no column with negative reduced cost exists is typically most appropriately done by running an algorithm that is guaranteed to find the column with the lowest reduced cost, and finding that the column it generates has a non-negative reduced cost.

To summarize, the basic recipe for column generation is to create an initial feasible set $\mathcal{K}' \subset \mathcal{K}$, solve the RLP and acquire the optimal values of the dual variables, u_i, use u_i to find a column with negative reduced cost, $\bar{c}_k$, include the corresponding column, k, in $\mathcal{K}'$ and to repeat the process until one can prove that no k with a negative reduced cost, $\bar{c}_k$, exists, at which point it is known that the RLP has the same optimal value as the LP, and as such, no more columns are needed.

2.1.1 Branching

Now, of course, we aren't trying to optimize the LP per se; what we are really interested in is optimizing the master problem (2.1), and it is seldom so that a generated column set $\mathcal{K}'$ that is optimal for the RLP is optimal for the RMP. To deal with this issue, one can use a branch-and-bound framework, generating columns until $\mathcal{K}'$ is optimal for the RLP, checking if the optimal solution to the RLP is feasible

for the RMP, and, if not, "branching" the LP into two complementary branches, each with opposing constraints that reduce the domain of the problem and make the optimal RLP solution infeasible [3]. This branch-and-bound framework is known as *branch-and-price*, owing to the fact that we introduce new variables by "pricing" them by their reduced costs. A more thorough look at branch-and-price can be found in [14].

As an example, one typical branching strategy when the optimal RLP solution violates integrality constraints is to branch on a variable λ_k with a non-integral value $\tilde{\lambda}_k$ by creating one branch with the added constraint $\lambda_k \leq \lfloor \tilde{\lambda}_k \rfloor$ and one branch with the added constraint $\lambda_k \geq \lceil \tilde{\lambda}_k \rceil$.

Within each branch, the column generation continues until once again there are no columns with negative reduced cost, at which point another branching occurs if the optimal RLP solution remains infeasible in the RMP. The process terminates once an RMP-feasible solution has been found in one of the branches, and it has been determined that all other branches have a lower bound that is equal to or greater than the value for this solution. How these bounds are calculated is explained in Subsection 2.1.2.

2.1.1.1 Ryan-Foster branching

The typical branching method outlined above, where one chooses a fractional variable to branch on, works quite poorly for set covering problems, like the item batching problem. In such problems, branching on a fractional variable means creating one branch in which the fractional variable is fixed to 1 (forcing the inclusion of the associated batch in the solution), and another where it is fixed to 0 (forbidding the associated batch in the solution). Forcing a variable to 1 and forcing the associated batch changes the problem significantly, but forcing a variable to 0 often has only a miniscule effect, since its blocks just one possible batch out of a vast number of possible batches.

As a result, branching on individual variables like this results in a very unbalanced search tree, and long computation times. To handle this problem, Ryan and Foster developed an alternative branching strategy in 1981, sometimes called *Ryan-Foster branching* [15], which branches on *item combinations*, rather than on individual variables, and produces much more balanced search trees as a result.

The branching strategy is based on the observation that if the optimal solution to a set covering RLP is fractional, there must be some pair of elements i and j such that

$$0 < \sum_{k \in \mathcal{K}'(i,j)} \lambda_k < 1\,, \tag{2.5}$$

where $\mathcal{K}'(i,j)$ is the set of all columns in $\mathcal{K}'$ in which both items i and j are present. In other words, there must be a pair of items for which the current optimal solution to the RLP contains batches that contain both of these items, as well as batches that contain one item but not the other. With this in mind, we can then branch by creating one branch in which

$$\sum_{k \in \mathcal{K}'(i,j)} \lambda_k = 0 \text{ (the } \textit{force-different branch}\text{)}, \tag{2.6}$$

where these two items may only appear in *different* batches, and one in which

$$\sum_{k \in \mathcal{K}'(i,j)} \lambda_k = 1 \text{ (the } \textit{force-same branch}\text{)}, \tag{2.7}$$

where these two items may only appear in the *same* batches.

This can more practically be enforced by forcing all variables whose associated sets contain both i and j to 0 in the force-different-branch, and all variables whose associated sets contain either i or j but not both to 0 in the force-same-branch. Note that sets that contain neither item are allowed in both branches.

2.1.2 Bounds

Obtaining bounds on $z^\star$ is very useful, in particular during the branching, since it allows us to *prune* branches (i.e., discard them without exploring them) if we find that their local lower bound is higher than the global upper bound. At any point during the column generation, we have an upper bound

$$z^\star_{\text{UB}} = z^\star_{\text{RMP}} \geq z^\star \,, \tag{2.8}$$

since the optimal solution to the RMP is also feasible for the unrestricted master problem, and has the same value in both problems.

Furthermore, we can obtain a lower bound on $z^\star$ by first observing that

$$z^\star_{\text{LP}} \leq z^\star \,, \tag{2.9}$$

since every solution to the master problem is also feasible in the LP relaxation of the master problem. If we can determine an upper bound $\kappa \geq \sum_{k \in \mathcal{K}} \lambda_k$, we can in turn find a lower bound for $z^\star_{\text{LP}}$ during our column generation by using

$$z^\star_{\text{LB}} = z^\star_{\text{RLP}} + \kappa \cdot \min_k \bar{c}_k \leq z^\star_{\text{LP}} \leq z^\star \,. \tag{2.10}$$

This lower bound on $z^\star_{\text{LP}}$ is a result of the convexity of the feasible region for the LP.

The value of $z^\star_{\text{RLP}} + \kappa \cdot \min_k \bar{c}_k$ does not necessarily increase monotonically from one iteration to the next, but since it always gives a valid lower bound to $z^\star$, we can use the highest value observed so far as our lower bound. The only restriction to this is that the lower bounds aren't valid globally; a lower value may be achievable in another branch with different constraints. Since branches inherit the constraints of their parent branches, a lower bound is valid for the branch in which it was observed and all its descendant branches, however. The upper bound, on the other hand, is valid globally and can be shared freely among all branches.

With these upper and lower bounds on $z^\star$, we can define the *optimality gap*, the difference between our upper and lower bounds, $z^\star_{\text{UB}} - z^\star_{\text{LB}} \geq 0$. When the optimality gap is zero, we know that $z^\star_{\text{LB}} = z^\star = z^\star_{\text{UB}}$.

In order to conserve computational power, one can set an early termination criterion for the column generation based on the optimality gap; if the gap is within some predefined threshold, the solution is considered good enough and the run terminates.

2.2 Warehouse picking tours

As stated in Section 2.1, one of the most important tasks when performing column generation is to solve the appropriate subproblem—which in the case of item batching is a prize-collecting travelling salesperson problem in a warehouse, or, a prize-collecting picker routing problem (PCPRP)—and being able to solve the subproblem to optimality allows one to verify that the current restricted column set is sufficient to solve the master problem optimally, and to calculate lower bounds for its optimal value throughout the column generation process.

This thesis project has largely been centred around the development of an algorithm that solves this subproblem to optimality. The developed algorithm—which we'll call the *reduced cost-optimal subproblem algorithm*—is an extension of a dynamical programming algorithm for the picker routing problem by Pansart, Cambazard and Catusse [4]. In Subsections 2.2.1–2.2.4, we present some necessary graph theory for our reduced cost-optimal subproblem algorithm. The definitions, lemmas, theorems and proofs in these sections are largely adapted from Pansart, Cambazard and Catusse's paper [4], an earlier paper by Cambazard and Catusse [5], and another paper by Ratliff and Rosenthal [6], on which Cambazard and Catusse based their work. The main difference between the theory as presented here and in the sources is that the theory here has been generalized to work with the PCPRP, and not just the PRP.

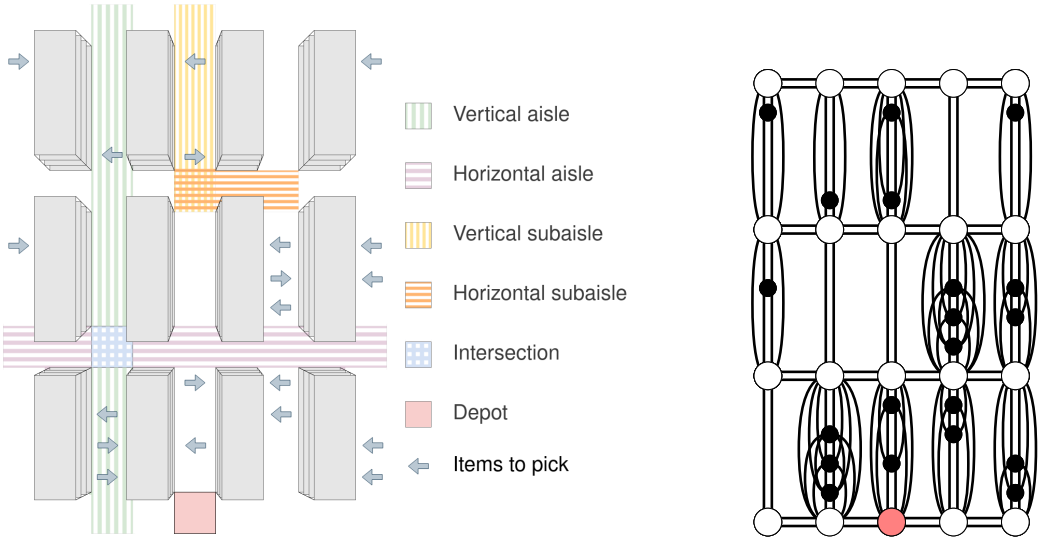
This theory is later applied in Section 3.3, where the reduced cost-optimal subproblem algorithm.

2.2.1 Warehouse multigraph representation

Warehouses tend to have shelves placed in well-aligned columns, with the aisles between them forming a grid. Typically, all shelves run in parallel, but can be broken up by orthogonal through-aisles, allowing warehouse workers to move more quickly from one shelf-aisle to the next. What we refer to here as *aisles* are the full aisles running from one end of the warehouse to another. The locations where two orthogonal aisles intersect are called *intersections*, and the segments of the aisles that run between just two intersections are called *subaisles*. In order to simplify the language somewhat, we will orient the warehouse such that the shelf-aisles run north-to-south (or "vertically") and the through-aisles west-to-east (or "horizontally").

Furthermore, there is typically a location where every picking job originates (a place where the workers pick up the trolleys used while picking, say), and a place where every picking job ends (an area where the collected items are packaged, for example). These locations can be the same place, or different. Here, we will assume that the start and end locations are one and the same, and we call this location the *depot*. We will further assume that the depot is located in the southern end of the warehouse, and that it lies on the intersection closest to the middle of the southernmost horizontal aisle. An example of a warehouse layout with the terms discussed above demonstrated is shown in Figure 2.1a.

For our reduced cost-optimal subproblem algorithm, we represent our warehouse with a multigraph. In our multigraph, nodes represent the shelf locations where there are items to pick (*product nodes*), or intersections (*intersection nodes*). The



(a) (b)

Figure 2.1: (a) An example of a small warehouse with different terms explained, and (b) the corresponding graph representation with white nodes as intersection nodes, black nodes as product nodes, and the depot node marked in red.

edges, meanwhile, represent the possible movements within a subaisle; the nodes in every subaisle (both intersection nodes and all product nodes therein) are doubly fully connected, i.e., for each pair of nodes in the subaisle, there are *two* edges. We call each such pair of edges an *edge pair*. The edges in an edge pair are completely interchangeable. Each edge has an associated length, which is simply the distance between the two points that it connects.

The graph representation of the warehouse in Figure 2.1a is shown in Figure 2.1b.

We will use this multigraph representation to find an optimal solution to the subproblem by finding an optimal subgraph to the multigraph. An example of what that may look like is shown in Figure 2.2.

2.2.2 Eulerian cycles as picking tours

To solve the subproblem to optimality, one needs to find the column—corresponding to a batch with t items—with the lowest reduced cost, where t is the trolley capacity. To phrase this more appropriately for the subproblem itself, we want to find the *picking tour*, T, of cardinality t, for which the reduced cost $\bar{c}(T) := L(T) - \sum_{i \in \mathcal{I}_T} u_i$ is minimal, where $L(T)$ is the distance walked in the picking tour (which corresponds exactly to the objective function coefficient of its associated column, but more on that in Section 3.1), $\mathcal{I}_T$ is the set of the indices of the items that are included in the picking tour T, and u_i is the optimal value of the dual variable associated with item i in the model Eq. (2.3). As mentioned, we will find our optimal picking tour by finding a subgraph to the multigraph described in Subsection 2.2.1. The value of $L(T)$ will be given by the sum of the lengths of the edges in this subgraph.

Now, let's formalize the concept of a picking tour:

Definition 1 (Adapted from [6])**.** A *picking tour* of cardinality t is a tour that

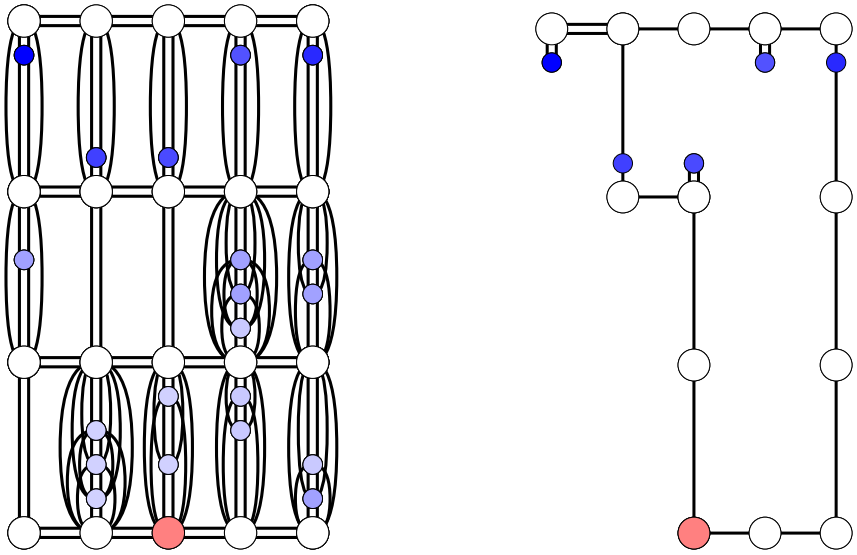


Figure 2.2: An example of a warehouse graph with 20 items, and the subgraph corresponding to the optimal solution to the PCPRP with a trolley capacity of 5. The shade of each product node represents the optimal value of its associated dual variable in model Eq. (2.3), with a darker shade indicating a higher value.

starts and ends at the depot node and picks t items, such that the combination of items picked does not violate any Ryan-Foster branching constraint that is in effect.

A picking tour is *optimal* if it has the lowest reduced cost out of all possible picking tours, and is *batch-optimal* if it is the shortest possible picking tour that picks its batch (i.e., no shorter picking tour which picks the exact same set of items exists).

Remember that the Ryan-Foster branching constraints dictate that two items must or must not be picked together. We need to take these into account when solving the subproblem to optimality to ensure that we can find the optimal column that is valid in the current branch. Note also that an optimal picking tour is also batch-optimal.

To construct this optimal picking tour, we will use *Eulerian cycles*:

Definition 2. An *Eulerian cycle* in a graph G is a tour that traverses every edge in G exactly once.

The reason for using Eulerian cycles is that they provide some convenient tricks that help us construct a picking tour by creating a subgraph to our warehouse multigraph.

Lemma 1. An Eulerian cycle in a connected graph G constitutes a tour that visits every node in G.

Proof of Lemma 1. The Eulerian cycle traverses every edge in G, and traversing an edge means that both endpoints of the edge are visited. Since every node in G constitutes the endpoint of at least one edge (as they are all connected), and all edges are traversed, all nodes are visited by the Eulerian cycle. Eulerian cycles are tours by definition. □

Now, we get into the theory behind constructing optimal picking tours, beginning with a quick lemma, which will let us prove that having two edges for each node pair in every subaisle in our multigraph is sufficient:

Lemma 2 (Adapted from [6])**.** The graph representing a batch-optimal picking tour never contains more than two edges between the same two nodes.

Proof of Lemma 2. Assume that we have a batch-optimal picking tour that contains at least three edges between the same two nodes. Label these nodes A and B, such that the tour moves from A to B through a connecting edge at least twice.

The full picking tour must be as follows: $DA\ AB\ BA\ AB\ BD$, where DA is some path from the depot to A, AB is an edge between A and B, BA is some path from B to A (not necessarily a direct edge between A and B), and BD is some path from B to the depot.

An alternative picking tour, $DA\ \overline{AB}\ BD$, where $\overline{AB}$ is BA backwards, visits the same nodes as the former tour, but is $2|AB|$ shorter. As such, the picking tour isn't batch-optimal, and we have a contradiction. □

Now, we formalize how our Eulerian cycles relate to picking tours:

Lemma 3 (Adapted from [6])**.** Every valid batch-optimal picking tour of cardinality t can be obtained from a connected subgraph to the warehouse multigraph containing the depot and exactly t product nodes, in which there is an Eulerian cycle.

Proof of Lemma 3. Assume that we have a valid batch-optimal picking tour of cardinality t with. Create a graph, T, containing only one node that represents the depot. Follow the tour. Whenever you reach an item that you wish to pick or an intersection, add the corresponding node to T, if it hasn't already been added. Every time a node is added or revisited, add an edge between this node and the last node you visited (even if an edge already exists between the two). Every node added to T is in the warehouse multigraph, and so is every edge.

Upon completing the tour, T has been filled out in such a way that the tour can now be represented as an Eulerian cycle; following the tour again, one would traverse every edge in T exactly once, and in the order they were created. The depot and all t product nodes were visited when T was constructed, and so they are all connected. By Lemma 2, we never use more than two edges between the same two nodes, since the picking tour is batch-optimal, and so, there are never more than two edges connecting any two nodes. □

Of course, an optimal picking tour must be a batch-optimal picking tour. Using Lemma 3, we see that the optimal picking tour (which, as a reminder, is the picking tour with the minimal reduced cost) has a corresponding connected subgraph to the warehouse multigraph in which there is an Eulerian cycle that contains the depot and t picked nodes. As such, we can obtain our optimal picking tour by generating this subgraph. To aid us with this, we have Euler's cycle theorem.

Theorem 1 (Euler's cycle theorem)**.** A connected graph with even degree parity at each node has an Eulerian circuit.

This theorem is well-known and won't be proven here. The particularly interested reader can find a proof in [16, Th. 16.2].

2.2.3 Transitions

In Subsection 2.2.2, we saw how we can create an optimal picking tour by finding a certain type of connected subgraph to the warehouse multigraph. Now, deciding exactly which edges to include in our subgraph is a difficult task, due to the combinatorial number of such subgraphs.

To reduce this complexity, we use the theory regarding Eulerian cycles from Subsection 2.2.2 to define *transitions*. A transition is simply a valid choice of which nodes and edges to keep in a given subaisle, or—in other words—a valid choice of subgraph for a single subaisle, such that it can be combined with other transition subgraphs to create a valid picking tour.

When solving the subproblem to optimality, we will construct our subgraphs by choosing which transition to use for each subaisle, creating a picking tour

$$T = \bigcup_{s \in S(G)} \tau_s \,, \tag{2.11}$$

where $S(G)$ is the set of subaisles in our warehouse graph, G, and τ_s is the choice of transition for subaisle s. To simplify the notation, we won't explicitly relate the transitions and subaisles with the subscript s, and we will instead denote the transitions that make up T simply as $\tau \in T$.

The reduced cost of the picking tour T will be given by

$$\bar{c}(T) = \bar{c}\left(\bigcup_{s \in S(G)} \tau_s\right) = \sum_{\tau \in T} \bar{c}(\tau) = \sum_{\tau \in T} \left(L(\tau) - \sum_{i \in \mathcal{I}(\tau)} u_i\right) , \tag{2.12}$$

where $L(\tau)$ is the total length of the edges in transition τ (which corresponds to the total distance walked in τ), $\mathcal{I}(\tau)$ is the set of items with product nodes in τ, and u_i is the optimal value of the dual variable associated with item i.

2.2.3.1 Generating transitions

In our column generation implementation for the item batching problem, we will begin by generating a list of valid transitions for each subaisle, and use this for every subproblem instance throughout the column generation process.

We consider a subaisle s with n_s items, running between intersections I_1 and I_2, where I_1 is the intersection at the southern end of the subaisle if the subaisle is vertical, and the intersection at the western end if it is horizontal. I_2 is the intersection at the northern or eastern end of the subaisle, analogously. We will now look at which transitions we can create for this subaisle. An example subaisle is shown in Figure 2.3.

A valid transition, τ_s, in s needs to meet certain criteria. When creating our optimal picking tour T, we will choose just one transition for subaisle s, and so its product

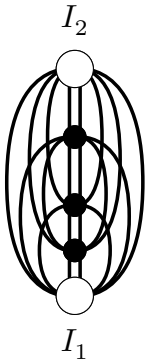


Figure 2.3: An example subaisle with three products to pick. I_1 and I_2 are intersection nodes, and the black nodes are product nodes.

nodes need to meet certain criteria in τ_s, since τ_s is the only transition with any edges in subaisle s in T.

By Theorem 1, we know that all nodes in T need to have even parity and be connected, which in particular means that all product nodes in τ need to have even parity, and cannot be isolated within the subaisle s, such that they cannot be connected to the rest of T. In practice, this means that every product node in τ needs to have even parity and be connected to I_1 or I_2; otherwise they can never be connected to any other node in T.

So, we're trying to find relevant choices for which product nodes, edges, and intersection nodes to keep in subaisle s. The best choice of product nodes will depend on the optimal values of the dual variables, u_i, and the Ryan-Foster branching constraints in a subproblem instance, neither of which are known in advance. As such, we will have to keep all options for exactly which products to pick open, with the one limitation that $|\mathcal{I}_\tau| \leq t$; clearly, we are not allowed to pick more items in one subaisle than we can fit into our picking trolley.

With that, what sets the different transitions for the same choice of items to pick in a subaisle apart from one another at this stage is their length, $L(\tau)$, and their *equivalence classes*.

Definition 3 (Adapted from [6])**.** The *equivalence class* of a transition τ is a 4-tuple, consisting of the following elements:

1. 0 if $I_1 \in \tau$ and I_1 has even parity in τ, 1 if $I_1 \in \tau$ with odd parity, and $-$ if $I \notin \tau$.
2. An analogous entry for I_2.
3. C if τ connects I_1 and I_2, and U (for "unconnected") if it doesn't.
4. $\mathcal{I}(\tau)$, the set of product nodes kept in τ.

The equivalence class of a transition gives us sufficient information about it to know how it affects our ability to form a valid picking tour T. This means that any $\tau \in T$ for a valid picking tour T can be replaced by another transition with the same equivalence class without making T invalid, even when we have branching constraints to take into account.

What this comes down to is that every intersection node $I \in T$ must have even parity and be connected, while these requirements are automatically fulfilled for the product nodes if they are fulfilled for all $I \in T$ if T is made up of valid transitions; like we've said, a valid transition needs to give all its product nodes even parity, and make sure they are each connected to at least one of the intersections in the subaisle. If all intersections in turn are connected, so are all product nodes.

Lemma 4 (Adapted from [6])**.** The equivalence classes possible for a valid transition τ are

- $[\,0,\ 0, \mathrm{C}, \mathcal{I}_\tau]$
- $[\,0,\ 0, \mathrm{U}, \mathcal{I}_\tau]$
- $[\,1,\ 1, \mathrm{C}, \mathcal{I}_\tau]$
- $[\,0, -, \mathrm{U}, \mathcal{I}_\tau]$
- $[-,\ 0, \mathrm{U}, \mathcal{I}_\tau]$
- $[-, -, \mathrm{U}, \ \emptyset]$.

Proof of Lemma 4. Examples of valid transitions with each of the listed equivalence classes are shown in Figure 2.4.

Now, if we fix $\mathcal{I}(\tau)$, there are twelve other equivalence classes we can describe, since there are $3 \cdot 3 \cdot 2 = 18$ ways of choosing our first three entries in the equivalence class.

Eight of these twelve equivalence classes have their first or second element as 1, but not both, which cannot be achieved by a valid transition; τ is a graph, and in all graphs the sum of the degree parity of the nodes is even (since adding an edge always increases the degree by one for two nodes simultaneously). Since one of the intersections has odd parity, but not the other, this must mean that one of the product nodes has odd parity, making the transition invalid.

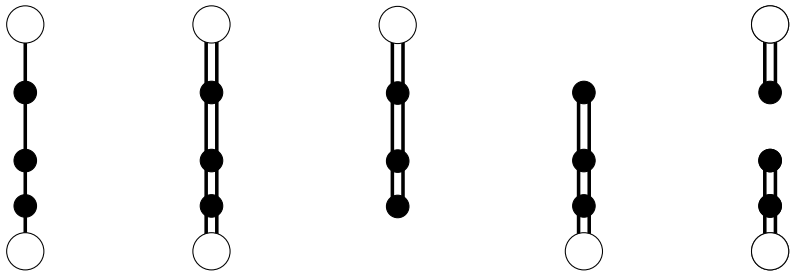
The remaining four describable equivalence classes are

- $[\,1,\ 1, \mathrm{U}, \mathcal{I}_\tau]$
- $[\,0, -, \mathrm{C}, \mathcal{I}_\tau]$
- $[-,\ 0, \mathrm{C}, \mathcal{I}_\tau]$
- $[-, -, \mathrm{C}, \mathcal{I}_\tau]$.

The first one has two unconnected intersections, both with odd parity, meaning we have two components, both with an intersection with odd parity, and—by the same logic as above—at least one product node with odd parity, as such, this equivalence class is invalid.

The final three all require a transition that connects the intersections, but at least one of the intersection isn't even in the transition, which clearly doesn't make sense.

Finally, in the case where $\mathcal{I}(\tau) \neq \emptyset$, $[-, -, \mathrm{U}, \mathcal{I}(\tau)]$ is invalid, since there are product nodes in the transition, but both intersections are missing, meaning the products nodes are isolated from the rest of the graph. □



$[1, 1, C, \mathcal{I}(\tau)]$	$[0, 0, C, \mathcal{I}(\tau)]$	$[-, 0, U, \mathcal{I}(\tau)]$	$[0, -, U, \mathcal{I}(\tau)]$	$[0, 0, U, \mathcal{I}(\tau)]$	$[-, -, U, \emptyset]$
Single transition	Double transition	Northern in-and-out transition	Southern in-and-out transition	Double in-and-out transition	Empty transition

Figure 2.4: Examples of all valid transition equivalence classes, along with the names given to transitions of these classes.

Since all transitions of the same equivalence class are interchangeable in terms of validity, we only need to consider the transition τ for each equivalence class with the shortest length $L(\tau)$; any other choice would lead to a larger value of $\bar{c}(T)$, which we're trying to minimize.

Furthermore, any transition, τ with equivalence class $[0, 0, U, \mathcal{I}(\tau)]$ can be replaced with a transition with equivalence class $[-, 0, U, \mathcal{I}(\tau)]$ or $[0, -, U, \mathcal{I}(\tau)]$ in a picking tour T without making T invalid, and so that equivalence class only needs to be considered if its best transition gives a lower value for $L(\tau)$ than the best transitions for $[-, 0, U, \mathcal{I}(\tau)]$ and $[0, -, U, \mathcal{I}(\tau)]$ do (which is never the case when $|\mathcal{I}(\tau)| \leq 1$).

On top of this, $[-, 0, U, \emptyset]$, $[-, 0, U, \emptyset]$ and $[0, -, U, \emptyset]$ all correspond to adding one or two intersections, and doing nothing else, which just means we get one or two more intersections that we must connect, for no benefit. As such, they are always strictly worse than $[-, -, U, \emptyset]$.

The best transitions for each equivalence class for an example subaisle with three products are shown in Figure 2.5.

2.2.4 Partial picking tours

Now, with the level of abstraction provided by the transitions defined in the previous subsection, our problem has been simplified somewhat, but we still have many possible subgraphs to consider. In this subsection, we will reduce the number of subgraphs to consider further by introducing the concept of *equivalent partial picking tours*, which will allow us to prove that certain partial picking tours are not part of the optimal solution, and as such, any subgraph that includes this partial picking tour will not be optimal, and can be ignored.

We start by defining some helpful notation for our warehouse graphs.

Definition 4. A subgraph $F \subseteq G$, where $G = (V, E)$ is a warehouse graph, is a *warehouse subgraph* if each subaisle either contains all its edge pairs, or none of them, and F is the edge-induced graph obtained from these edge pairs. We

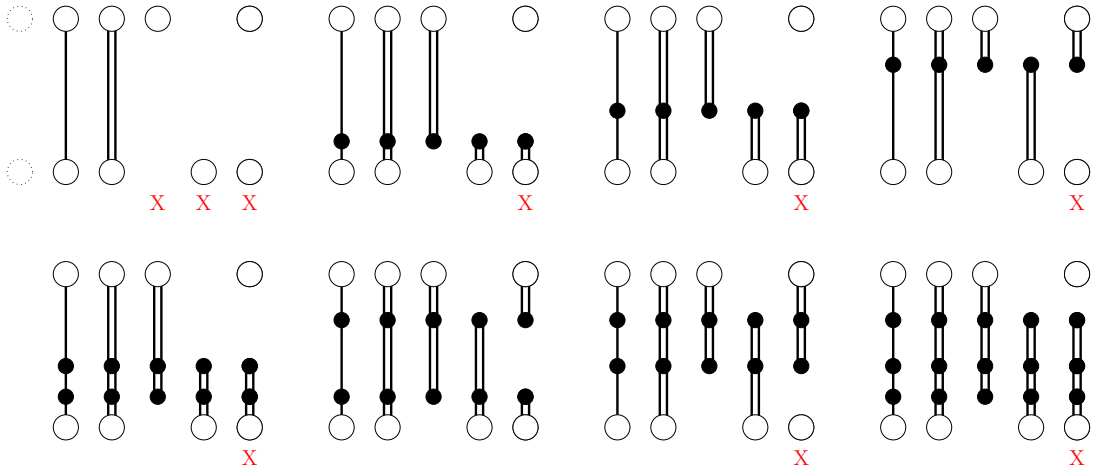


Figure 2.5: The best transition for each valid equivalence class for an example sub-aisle with three products. The dotted intersections represent the empty transition. Each transition marked with a red X is strictly worse than some other transition, and can hence be discarded.

denote this with $F \tilde{\subseteq} G$. The edge-induced graph obtained from the edges $E \backslash F$ is denoted $\bar{F}$.

Definition 5 (Adapted from [6] and [5])**.** For a warehouse subgraph $F \tilde{\subseteq} G$, a subgraph $P \subseteq F$ is an F *partial picking tour subgraph* if there exists a *tour completion subgraph* $C \subseteq \bar{F}$, such that $P \cup C$ is a picking tour in G.

In other words, a subgraph $P \subseteq F$ is an F partial picking tour if we can turn it into a complete picking tour by only adding nodes and edges from $\bar{F}$. An example of a partial picking tour subgraph, a valid tour completion subgraph, and the full picking tour obtained by combining the two are shown in Figure 2.6. Note that the tour completion subgraph C is also a partial picking tour subgraph, with P as its completion.

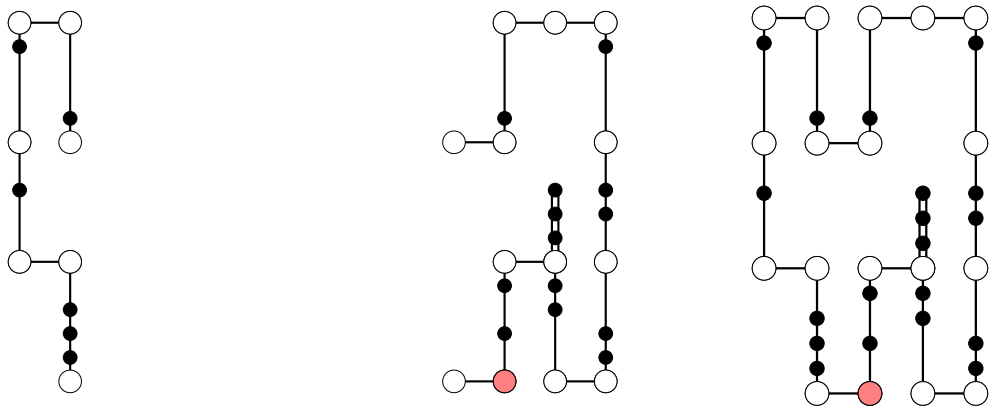
This is useful, since we can split the problem of finding an optimal picking tour into two loosely connected problems: one in which we find an optimal partial picking tour, P, in some F, and an optimal picking tour, C, in $\bar{F}$, such that $P \cup C$ is an optimal picking tour. The only information P and C need about each other is how many items the other one picks (so that they pick the right number in total), how the other one affects the Ryan-Foster branching constraints that are in effect (which we will describe more thoroughly below), and what the other partial picking tour "looks like" at the nodes where they intersect.

We call the set of nodes where F and $\bar{F}$ intersect (and hence, where P and C can intersect) the *border*:

Definition 6 (Adapted from [6] and [5])**.** For a warehouse subgraph $F \tilde{\subseteq} G$ where G is a warehouse graph, the *border node set* (or simply, the *border*) of F is the set of nodes

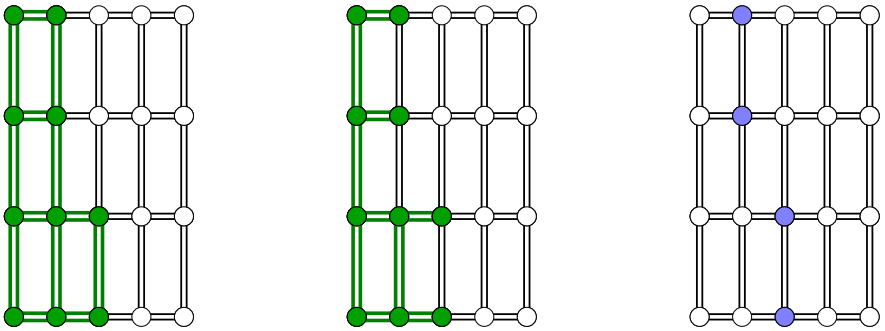
$$B_F := F \cap \bar{F}\,. \tag{2.13}$$

Essentially, the only way to move from F to $\bar{F}$ is by passing through the border



(a) (b) (c)

Figure 2.6: (a) A partial subtour; (b) a tour completion subgraph to the partial subtour in (a); (c) the full picking tour obtained by combining the partial subtour and the tour completion.



(a) (b) (c)

Figure 2.7: Two different subgraphs, marked in green in (a) and (b) with the same border, shown in (c). Note that the subgraphs differ by the fact that the one in (a) includes all subaisles between border nodes, while the one in (b) doesn't.

(hence the name), and so, the border contains some information necessary to create compatible partial picking tours and tour completions.

Note that two different warehouse subgraphs can have the same border. Two subgraphs that have the same border are shown in Figure 2.7.

With the border defined, we define the *border parity vector* and *border component index vector* for a partial picking tour, which are useful for finding valid completions.

Definition 7 (Adapted from [6] and [5])**.** The *border parity vector* of an $F \,\tilde{\subseteq}\, G$ partial picking tour P (with partial picking tours as defined in Definition 5), denoted $B_{\text{par}}(P)$, is a vector with one element for each node in B_F, where each element is 1 if the corresponding node has odd degree in P, and 0 if it has even degree in P, or isn't in P at all.

Definition 8 (Adapted from [6] and [5])**.** The *border component index vector* of an $F \,\tilde{\subseteq}\, G$ partial picking tour P, denoted $B_{\text{comp}}(P)$, is a vector with one element for each node in B_F, where each element is $-$ if the corresponding node isn't in

Figure 2.8: An F partial picking tour, P, with $B_{\text{par}}(P) = [0, 0, 1, 1]$, and $B_{\text{comp}}(P) = [1, -, 2, 2]$, with the first element in a vector corresponding to the bottom node, the second to the second node from the bottom, and so on. The blue nodes and the dotted node make up B_F.

P, or the index of its component if it is in P. The indices are chosen so that 1 is the first to appear in $B_{\text{comp}}(P)$, 2 the second, and so on.

The last sentence in Definition 8 means that $B_{\text{comp}}(P) = [1, -, 2, 1, 3]$ is a valid border component index vector, but $B_{\text{comp}}(P) = [2, -, 1, 2, 3]$ is not, for example.

An example of a partial picking tour is shown in Figure 2.8, with its border parity vector and border component index vector in the caption.

One of the requirements for an $F = (V, E) \subseteq\!\!\!\!\sim\, G$ partial picking tour, P, and a tour completion, C, to form a valid picking tour $P \cup C$, is that $P \cup C$ must be connected, and so $P \cup C$ is valid only if $B_{\text{comp}}(P)$ and $B_{\text{comp}}(C)$ are compatible, meaning that they connect together to form just one component (remember, the border is the only place where they "meet", so these connections cannot be created anywhere else).

Furthermore, for $P \cup C$ to be a valid picking tour, all nodes need to have even parity. Since the nodes in B_F are the only ones in both F and $\bar{F}$, they are the only nodes that *can* have non-zero degree in both P and C. As a consequence, all nodes in $F \backslash B_F$ must have even parity in P, the nodes in $\bar{F} \backslash B_F$ must have the even parity in C, and $B_{\text{par}}(P) = B_{\text{par}}(C)$ must hold.

Beside the border parity vector and border component index vector, every partial picking tour has a constraint status vector:

Definition 9. The *constraint status vector* of an $F \subseteq\!\!\!\!\sim\, G$ partial picking tour, P, denoted $S(P)$, consists of the statuses for each Ryan-Foster branching constraint that is in effect. These statuses come in five variants in total.

For a *force-same constraint* between two items, i and j with product nodes p_i and p_j, an F partial picking tour, P, has one of the following statuses:

- *unreached* if $\{p_i, p_j\} \subset \bar{F}$;
- *cleared* if $\{p_i, p_j\} \subset P$, or $\{p_i, p_j\} \subset F \backslash P$;
- *can't-pick* if $p_i \in F \backslash P$, and $p_j \in \bar{F}$, or if $p_j \in F \backslash P$, and $p_i \in \bar{F}$;
- *must-pick* if $p_i \in P$, and $p_j \in \bar{F}$, or if $p_j \in P$, and $p_i \in \bar{F}$;

Table 2.1: The compatibility of all status combinations. *FS* means that a combination is valid for force-same constraints, *FD* means that it's valid for force-different constraints, and empty, red cells denote combinations that are incompatible or impossible.

	Unreached	Cleared	Can't-pick	Must-pick	Violated
Unreached		FS, FD			
Cleared	FS, FD		FD		
Can't-pick		FD	FS		
Must-pick				FS	
Violated					

- *violated* if $\{p_i, p_j\} \subset F$, and $|\{p_i, p_j\} \cap P| = 1$.

For a *force-different constraint* between items i and j, P, has one of the following statuses:

- *unreached* if $\{p_i, p_j\} \subset \bar{F}$.
- *cleared* if $|\{p_i, p_j\} \cap (F \backslash P)| \geq 1$.
- *can't-pick* if $p_i \in P$, and $p_j \in \bar{F}$, or if $p_j \in P$, and $p_i \in \bar{F}$.
- *violated* if $\{p_i, p_j\} \subset P$.

These constraint statuses need to be considered when trying to find valid picking tour completions, since they tell us how (or if) we need to deal with each constraint in the tour completion; must-pick statuses tell us that valid tour completions must contain certain product nodes (or, from the perspective of picking items in the warehouse, the completion must pick certain items, hence the name), while can't-pick statuses tell us that valid tour completions cannot contain certain product nodes, for example.

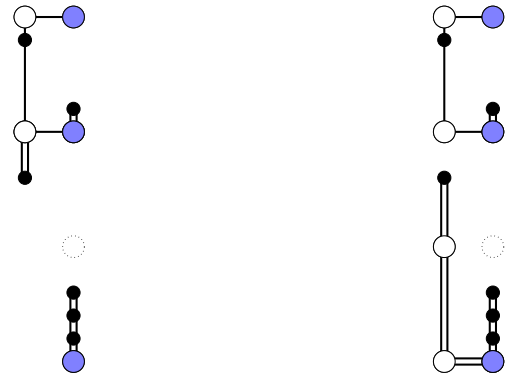
To formalize this concept somewhat, we say that $P \cup C$ is a valid picking tour if $S(P)$ and $S(C)$ are *compatible*, which they are if the combination of the ith element in each vector, $S_i(P)$ and $S_i(C)$, form a compatible combination for every i. All combinations and whether they are compatible are shown in Table 2.1.

We can now define *equivalent partial picking tours*, which are essential for our reduced cost-optimal subproblem algorithm.

Definition 10 (Adapted from [6] and [5])**.** Two $F \tilde{\subseteq} G$ partial picking tours, P_1 and P_2, are *equivalent* if

$$p(P_1) = p(P_2)\,, \tag{2.14a}$$
$$B_{\text{comp}}(P_1) = B_{\text{comp}}(P_2)\,, \tag{2.14b}$$
$$B_{\text{par}}(P_1) = B_{\text{par}}(P_2)\,, \text{ and} \tag{2.14c}$$
$$S(P_1) = S(P_2)\,, \tag{2.14d}$$

where $p(P)$ is the number of product nodes in P. We denote this equivalence by $P_1 \equiv P_2$.



(a) P_1. (b) P_2.

Figure 2.9: Two equivalent partial picking tours P_1 and P_2. $p(P_1) = p(P_2) = 6$, $B_{\text{comp}}(P_1) = B_{\text{comp}}(P_2) = [1, -, 2, 2]$, $B_{\text{par}}(P_1) = B_{\text{par}}(P_2) = [0, 0, 1, 1]$, with the first element in a vector corresponding to the bottom node, the second to the second node from the bottom, and so on. Since P_1 and P_2 include the same product nodes, $S(P_1) = S(P_2)$.

An example of two equivalent partial picking tours is shown in Figure 2.9.

Now, to explain the "equivalent" part of "equivalent partial picking tours":

Lemma 5 (Adapted from [6] and [5])**.** Given a warehouse subgraph $F \subseteq\!\!\!\!\sim\, G = (V, E)$, any tour completion $C \subseteq \bar{F}$ for which $P_1 \cup C$ is a valid picking tour, where P_1 is an F partial picking tour, $P_2 \cup C$ is also a valid picking tour for any $P_2 \equiv P_1$.

Proof of Lemma 5. For $P_2 \cup C$ to be a valid picking tour, we must have that $p(P_2) + p(C) = t$, where t is the total number of items to pick, $S(P_2)$ and $S(C)$ need to be compatible, as do $B_{\text{comp}}(P_2)$ and $B_{\text{comp}}(C)$, and $B_{\text{par}}(P_2) = B_{\text{par}}(C)$.

First off,

$$p(P_2) + p(C) = p(P_1) + p(C) = t\,. \tag{2.15}$$

Furthermore,

$$S(P_1) \text{ compat. with } S(C) \iff S(P_2) \text{ compat. with } S(C)\,, \tag{2.16}$$

since $S(P_1) = S(P_2)$.

Similarly,

$$B_{\text{comp}}(P_1) \text{ compat. with } B_{\text{comp}}(C) \iff B_{\text{comp}}(P_2) \text{ compat. with } B_{\text{comp}}(C)\,. \tag{2.17}$$

Finally,

$$B_{\text{par}}(P_1) = B_{\text{par}}(P_2) = B_{\text{par}}(C)\,, \tag{2.18}$$

and so, all requirements are met. □

Finally, to wrap up the subsection, we prove that we can use the concept of equivalent partial picking tours to ignore certain partial picking tours when solving our problem.

Theorem 2 (Equivalent partial picking tours. Adapted from [6] and [5])**.** A picking tour T with a partial picking tour $P_1 \subseteq T$ for which

$$\exists P_2 \equiv P_1\,,\ \text{such that}\ \ \bar{c}(P_2) < \bar{c}(P_1)\ \text{holds} \tag{2.19}$$

is not optimal.

Proof of Theorem 2. $T = P_1 \cup C$ for some completion C to P_1.
Now,

$$\begin{aligned} \bar{c}(T) &= \bar{c}(P_1 \cup C) \\ &= \bar{c}(P_1) + \bar{c}(C) \\ &> \bar{c}(P_2) + \bar{c}(C) \\ &> \bar{c}(P_2 \cup C)\,. \end{aligned} \tag{2.20}$$

By Lemma 5, $P_2 \cup C$ also constitutes a valid picking tour, and since $\bar{c}(T) > \bar{c}(P_2 \cup C)$, T isn't optimal. □

We will use this frequently in our reduced cost-optimal subproblem algorithm (to be described in Section 3.3); this algorithm works by iteratively extending partial picking tours, and, using Theorem 2, it can safely ignore any partial picking tour P_1 for which $\exists P_2 \equiv P_1\,,\ \bar{c}(P_2) < \bar{c}(P_1)$, along with any extensions of P_1.

2.2.5 A lower bound for picking tour lengths

We have all the theory we need for the reduced cost-optimal subproblem algorithm at this point, but while on the topic of picking tours, we briefly look at a simple way of getting a lower bound on length of a batch-optimal picking tour for a given batch. This will form the basis of our *stair-based subproblem algorithm*, described in Section 3.2, with which we will generate batches that can be picked by simple picking tours of exactly the same length as this lower bound, giving us a fast and easy way of finding the lengths of their corresponding batch-optimal picking tours.

Lemma 6. For an optimal picking tour, T, of cardinality $t \geq 1$, with tour length $L(T)$, we have that

$$L(T) \geq d(D, a) + d(a, b) + d(b, D)\,, \tag{2.21}$$

where D is the depot node, $a, b \in T$ are product nodes, and $d(v, w)$ is the distance of the shortest path from node v to node w.

Proof of Lemma 6. A picking tour T can be described as follows

$$T = (D,\, p_2,\, p_3,\, \ldots,\, p_t,\, p_{t+1},\, D)\,, \tag{2.22}$$

where the sequence denotes the order in which the nodes are visited in T, and all product nodes $p_2, p_3, \ldots, p_{t+1} \in T$. Of course, it follows that

$$L(T) = \sum_{k=1}^{t+1} d(T_k, T_{k+1})\,. \tag{2.23}$$

By the triangle inequality (which trivially holds in this geometry), we have

$$d(q_1, q_3) \leq d(q_1, q_2) + d(q_2, q_3)\,, \tag{2.24}$$

for any nodes q_1, q_2, and q_3, which extends by induction to

$$d(q_1, q_m) \leq \sum_{i=1}^{m-1} d(q_i, q_{i+1}) \tag{2.25}$$

for any sequence of nodes $q_1, q_2, \ldots, q_m$. As such, we have

$$\begin{aligned} L(T) &= \sum_{k=1}^{t+1} d(T_k, T_{k+1}) \\ &= \sum_{k=1}^{i-1} d(T_k, T_{k+1}) + \sum_{k=i}^{j-1} d(T_k, T_{k+1}) + \sum_{k=j}^{t+1} d(T_k, T_{k+1}) \\ &\geq d(T_1, T_i) + d(T_i\,, T_j) + d(T_j, T_{t+2}) \\ &= d(D, p_i) + d(p_i\,, p_j) + d(p_j, D)\,, \end{aligned} \tag{2.26}$$

for any nodes $p_i\,, p_j \in T$, with $i < j$. With $a = p_i$, and $b = p_j$, we get (2.21). The proof may appear to only apply if a is a node earlier in the sequence of nodes in T than b, but since

$$d(D\,, a) + d(a\,, b) + d(b\,, D) = d(D\,, b) + d(b\,, a) + d(a\,, D)\,, \tag{2.27}$$

the proof applies regardless of the order of a and b in T.

For the case $a = b$, the proof is similar, but without the sum from i to $j - 1$ in the second line of (2.26), and without $d(T_i\,, T_j)$ and $d(p_i\,, p_j)$ in the subsequent lines, which works out to (2.21), since $d(a\,, a) = 0$. □

2.3 The A* algorithm

The A* algorithm is an algorithm for finding the shortest path between two points in a graph $G = (V, E)$ with non-negative edge weights. We will use it to speed up the optimal subproblem algorithm, which can be viewed as shortest path problems in an alternative graph in which the nodes are essentially partial picking tour equivalences, as described in Subsection 2.2.4. This approach will be explained in further detail in Subsection 3.3.2.

The A* algorithm is a simple extension of the well-known *Dijkstra's algorithm*. Dijkstra's algorithm finds the shortest path between a start node, v_s, and a terminal node, v_t, in a graph with non-negative edge weights by exploring the graph iteratively. In each iteration, it moves to the node $v \in V \backslash V_\mathrm{visited}$ with minimal $c(p(v))$ in each iteration, where V_visited is the set of all previously visited nodes, $p(v)$ the cheapest path found from v_s to v, and $c(p(v))$ its cost [17].

The A* algorithm aims to improve upon Dijkstra's algorithm by prioritizing which node v to visit not solely based on $c(p(v))$, instead choosing the node $v \in \mathcal{V}$ with

minimal $c(p(v)) + h(v)$, where $h(v)$ is an *admissible* heuristic cost function that estimates the cost of the cheapest path from v to v_t [18]. $h(v)$ being admissible means that it never overestimates the true cost. Note that the A* algorithm may need to revisit previously visited nodes, unlike Dijkstra's algorithm.

The algorithm can be more efficient if the heuristic cost function is also *consistent*, meaning that

$$h(v_1) \geq d(v_1, v_2) + h(v_2)\,,\ \forall (v_1, v_2) \in E\,. \tag{2.28}$$

If the heuristic cost function is consistent, the A* algorithm can be run without ever needing to revisit nodes; it can run as if it were Dijkstra's algorithm, but with edge weights

$$d_{\mathrm{A}*}(v_1, v_2) = d(v_1, v_2) + h(v_1) - h(v_2)\,, \tag{2.29}$$

which doesn't work for non-consistent heuristic functions $h(v)$, since $d_{\mathrm{A}*}(v_1, v_2)$ may be negative for such functions. If the function is consistent, however, (2.28) gives that

$$\begin{aligned} d_{\mathrm{A}*}(v_1, v_2) &= d(v_1, v_2) + h(v_1) - h(v_2) \\ &\geq d(v_1, v_2) + h(v_1) - h(v_1) - d(v_1, v_2) \\ &= 0\,, \end{aligned} \tag{2.30}$$

which, as it happens, holds even if $d(v_1, v_2)$ is negative; as a consequence, the A* algorithm can be used for graphs with negative edge weights (which we will have in our implementation), but only if the heuristic function is consistent.

The A* algorithm with a consistent heuristic cost function is outlined in Algorithm 1.

Algorithm 1 The A* algorithm with a consistent heuristic cost function. *Adapted from [18].*

$v_\mathrm{s} \leftarrow$ The starting node.
$v_\mathrm{t} \leftarrow$ The terminal node.
$c(p_v) \leftarrow$ A function that calculates the cost of a path, p_v.
$h(v) \leftarrow$ A function that calculates a consistent heuristic cost from node v to v_t.
$PQ \leftarrow$ A priority queue consisting of node–path pairs (v, p_v). PQ orders its contents in ascending order by heuristic total cost $c(p_v) + h(v)$.
$A_\mathrm{visited} \leftarrow \emptyset$.
Add $(v_\mathrm{s}, [\,])$ to PQ.
while PQ not empty **do**
    $(v, p_v) \leftarrow$ First tuple in PQ, for which $v \notin A_\mathrm{visited}$.
    $A_\mathrm{visited} \leftarrow A_\mathrm{visited} \cup \{v\}$.
    if $v = v_\mathrm{t}$ **then**
        Return p_v.
    for $\tilde{v}$ in neighbourhood of v **do**
        $p_{\tilde{v}} \leftarrow p_v + [(v, \tilde{v})]$.
        Add $(v, p_{\tilde{v}})$ to PQ.

Using a heuristic cost function in this manner tends to reduce the amount of nodes visited (and as a result, the runtime, hopefully) significantly if the heuristic gives

reasonably accurate estimates, but if the heuristic is too expensive to calculate, this may lead to longer runtimes. As such, one needs to find a heuristic cost function that gives good estimates without requiring heavy computations, and, preferably, the function should be admissible and consistent, or at least admissible.

3 Modelling and implementation

In this chapter we look at the details of the model used to solve the item batching problem, along with its implementation, and the two algorithms used to solve the column generation subproblem.

We start with a look at the details of the column generation master problem and what it represents for the item batching problem in Section 3.1, along with how the column generation is run in practice in Subsection 3.1.1, our choice of the initial restricted column set in Subsection 3.1.2, and a reduction to the domain of the master problem in Subsection 3.1.3.

We continue with a look at the details of a fast subproblem algorithm—which isn't guaranteed to generate the optimal solution to the subproblem—in Section 3.2.

After that, we look at the implementation of our reduced cost-optimal subproblem algorithm—the main focus of this project—in Section 3.3, how we reduce the problem to make it more manageable in Subsection 3.3.1, and how to augment the algorithm by applying the A* algorithm in Subsection 3.3.2.

Finally, we look at some details regarding the Ryan-Foster branching used in our column generation implementation in Section 3.4.

3.1 The master problem

To apply column generation to the batching problem, we of course need to formulate a mixed-integer program. As a reminder, column generation MILPs tend to look something like this:

$$z^\star := \min_{\lambda_k} \quad \sum_{k\in\mathcal{K}} c_k\lambda_k\,, \tag{3.1a}$$

$$\text{s.t.} \quad \sum_{k\in\mathcal{K}} a_{ik}\lambda_k \geq b_i\,, \qquad \forall i \in \mathcal{I}, \tag{3.1b}$$

$$\lambda_k \in \{0,1\}\,, \qquad \forall k \in \mathcal{K}\,, \tag{3.1c}$$

Each column k corresponds to a batch in our case, each batch simply being a set of products, the number of which is limited by the trolley capacity, t. c_k, then, is naturally the "cost" of batch k, i.e., the minimal tour length for a tour that picks all items in the batch associated with column k.

As for the constraints in (3.1b), they are supposed to ensure that each product is picked at least once, and so we have that $a_{ik} = 1$ if the product with index i is in batch k, otherwise $a_{ik} = 0$, and $b_i = 1\,,\ \forall i \in \mathcal{I}$. Furthermore, when solving the

subproblem, we need to ensure that the generated picking tours pick exactly t items. We do this by imposing the constraint $\sum_i a_{ik} = t \ \forall k$ on the subproblem.

The restricted master problem and the LP relaxation of the restricted master problem are defined analogously to (2.2) and (2.3), respectively.

3.1.1 Applying column generation

With our models defined, we need the following things to apply column generation to our problem:

1. An initial batch set, which gives a feasible solution to the LP relaxation
2. A branching scheme
3. A termination criterion
4. Our subproblem algorithms, and rules for when to apply which one.

The initial batch set we use is described in Subsection 3.1.2. The branching scheme we use will be the Ryan-Foster branching, which was outlined in Subsection 2.1.1.1, with some additional notes concerning exactly which one of the potential element combinations to branch on we use in Section 3.4. As for our termination criterion, we will—for the sake of this report—run our column generation until our optimality gap is zero; that is, we run until we have found an optimal solution to the LP relaxation of the master problem and verified that it is indeed optimal.

As for the subproblem algorithms, we will use two: the *stair-based subproblem algorithm* and the *reduced cost-optimal subproblem algorithm*, where the stair-based algorithm is designed to quickly generate multiple columns with negative reduced cost—but doesn't guarantee that any of the columns found are optimal solutions to the subproblem—and the reduced cost-optimal subproblem algorithm is guaranteed to solve the subproblem to optimality, but spends more time doing so, and only generates one column at a time. These algorithms are presented in Section 3.2 and Section 3.3, respectively.

The overarching algorithm used for our column generation implementation is presented in Algorithm 2.

3.1.2 Initial restricted column set

To get the column generation going, we need a set of initial columns for which we can generate a feasible solution to the LP relaxation of the master problem. We could just create batches randomly and use their columns, but if we create a heuristically good initial restricted column set the algorithm starts from a better position and can usually finish more quickly.

One intuitively decent way of creating our initial batch set is to start in the southwest corner of the warehouse, and to follow the westernmost vertical aisle north, giving the first item we encounter number 1, the second number 2 and so on. Once we reach the northern end of the vertical aisle, we continue numbering the items in the next vertical aisle to the east, but here we start from the north and move south through the aisle. Once that aisle is complete, we start from the south in the third vertical aisle from the west going north, and so on until every item has a number.

Algorithm 2 Column generation algorithm

$B_{\text{init}} \leftarrow$ The initial branch of the master problem.
$\mathcal{B} \leftarrow \{B_{\text{init}}\}$.
$\mathcal{K}' \leftarrow$ The initial restricted column set. ▷ Described in Subsection 3.1.2.
$\mathcal{K}'(B) \leftarrow \{k \in \mathcal{K}' : k \text{ feasible in } B\}$.
$\kappa \leftarrow N/t$, with N as the total number of items, and t as the trolley capacity.
$z^\star_{\text{RMP}}(\mathcal{K}) \leftarrow$ The optimal value for the RMP with a column set $\mathcal{K}$.
$z^\star_{\text{RLP}}(\mathcal{K}) \leftarrow$ The optimal value for the RLP with a column set $\mathcal{K}$.
$z^\star_{\text{LB},B_{\text{init}}} \leftarrow 0$.
$z^\star_{\text{UB}} \leftarrow z^\star_{\text{RMP}}(\mathcal{K}')$.

while $\mathcal{B} \neq \emptyset$ **do**
  $B \leftarrow \text{argmin}_{B \in \mathcal{B}} z^\star_{\text{LB}}(B)$.
  repeat
    $\mathcal{K}'_{\text{stair}} \leftarrow$ The columns generated by the stair-based subproblem algorithm with the branching constraints in B. ▷ Described in Section 3.2.
    $\mathcal{K}' \leftarrow \mathcal{K}' \cup \mathcal{K}'_{\text{stair}}$.
    if $\mathcal{K}'_{\text{stair}} = \emptyset$ **then**
      $k \leftarrow$ The column generated by the reduced cost-optimal subproblem algorithm with the branching constraints in B. ▷ Described in Section 3.3.
      if $\bar{c}_k < 0$ **then**
        $\mathcal{K}' \leftarrow \mathcal{K}' \cup \{k\}$.
        $z^\star_{\text{LB},B} \leftarrow \max(z^\star_{\text{LB},B}, z^\star_{\text{RLP}}(B) + \kappa \cdot \bar{c}_k)$.
    $z^\star_{\text{UB}} \leftarrow z^\star_{\text{RMP}}(\mathcal{K}')$
  until $\mathcal{K}'_{\text{stair}} = \emptyset$ and $\bar{c}_k \geq 0$, or $z^\star_{\text{LB},B} > z^\star_{\text{UB}}$.
  if optimal solution to $z^\star_{\text{RLP}}(\mathcal{K}'(B))$ is fractional **then**
    $i, j \leftarrow$ Two items to branch on. ▷ Described in Section 3.4.
    $B_1 \leftarrow B$, but with a force-same constraint between i and j added.
    $B_2 \leftarrow B$, but with a force-different constraint between i and j added.
    $z^\star_{\text{LB},B_1}, z^\star_{\text{LB},B_2} \leftarrow z^\star_{\text{RLP}}(\mathcal{K}'(B))$.
  $\mathcal{B} \leftarrow \mathcal{B} \backslash \{B\}$.
return the optimal solution to $z^\star_{\text{RMP}}(\mathcal{K}')$.

With this numbering we can create heuristically good batches by letting items 1 through t form a batch, 2 through $t+1$ another one, and so on. We then repeat the whole process, but now start the numbering from the *north*western corner of the warehouse, going south. An example of the two product orderings used is shown in Figure 3.1.

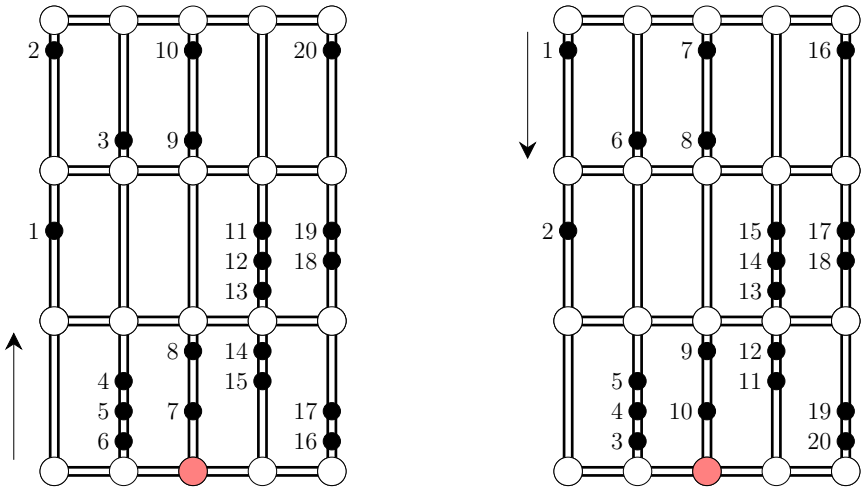


Figure 3.1: An example of the two product orderings used by the algorithm that creates the initial restricted column set. Only a subset of the edges in the warehouse graph have been drawn for legibility.

3.1.3 Domain reduction

Before we get to the subproblem algorithms, we make a useful observation about the composition of an optimal solution to the master problem: it can always be constructed without using any *product skipping* transitions.

By product skipping, we mean picking one item in a subaisle, passing by another item in the same subaisle without picking it, and then picking another item in that subaisle.

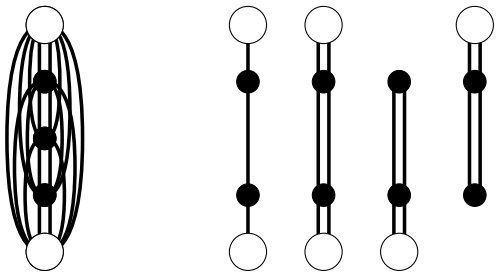
In the language of our transitions, described in Subsection 2.2.3, a transition, τ, in a subaisle, s, uses product skipping if one of its intersections, $I \in \{I_1, I_2\}$, is connected to two products, $p_1, p_2 \in \tau$, for which there is another product, $q \in s \backslash \tau$, such that

$$d(I, p_1) < d(I, q) < d(I, p_2)\,, \tag{3.2}$$

where $d(a, b)$ is the distance between nodes a and b.

A subaisle and its product skipping transitions are shown in Figure 3.2.

The reason for this is quite simple: the batches in our optimal solution to the master problem each have a corresponding optimal picking tour, and if we have one such picking tour, T_1, for which $p_1, p_2 \in T_1$, $q \notin T_1$, we must also have another picking tour, T_2, for which $q \in T_2$ which passes either p_1 or p_2, since q cannot be reached without passing p_1 or p_2. As such, if we were to obtain an optimal solution to the item batching problem like this, we could simply swap q with p_1 or p_2 (whichever product T_2 passes), getting $p_1, q \in T_1$ and $p_2 \in T$ if T_2 passes p_2, for example. This



(a) (b)

Figure 3.2: (a) A subaisle, and (b) its product skipping transitions, all of which skip the middle product.

swap doesn't increase $c_{k_1} = L(T_1)$, nor $c_{k_2} = L(T_2)$, with k_1 and k_2 as the columns with optimal picking tours T_1 and T_2 respectively, since T_1 passed q and T_2 passed p_2 before the swap.

As such, our subproblem algorithms will only consider transitions that don't use product skipping, thus reducing the problem domain somewhat.

3.2 The stair-based subproblem algorithm

The goal of the stair-based subproblem algorithm is to find valid picking tours with lengths exactly equal to the lower bound on picking tour lengths in Lemma 6; such picking tours of course have the optimal tour length for the items they pick, and part of what makes the subproblem difficult to solve is that we need to make sure we attain the optimal tour for any batch we consider.

The basic idea is to, for every intersection $I_{i,j}$ in the warehouse, and every number of items, $k = 0, 1, \ldots, t$, find the *stair path* $P^k_{i,j}$ from the depot, D, to $I_{i,j}$ that picks k items with the lowest reduced cost, $\bar{c}(P^k_{i,j}) = L(P^k_{i,j}) - \sum_{i \in \mathcal{I}(P^k_{i,j})} u_i$, where $L(P)$ is the length of a partial picking tour P, $\mathcal{I}(P)$ contains the indices of the items picked in P, and u_i is the optimal value of the dual variable associated with item i in the restricted master problem.

Definition 11. For an intersection I, an I *stair path*, P, consists of a path between the depot, D, and I, which is minimal in length, and then potentially, a detour into the subaisle north of I (if there is one) before returning to I.

Examples of stair paths are shown in Figure 3.3.

In practice, these paths will consist of a sequence of single transitions, as defined in Subsection 2.2.3, potentially ending with a southern in-and-out transition. For an intersection I that is farther north and west than the depot D, the sequence of single transitions will pass through a sequence of intersections, where each intersection is either north or west of the previous; moving to the east or the south would increase the distance to I, making the path not minimal in length. Analogously, if I is farther north and east than D, the sequence of intersections will be such that every intersection is north or east of the previous, and if I is due north of D, the sequence of intersections will be such that every intersection is north of the previous.

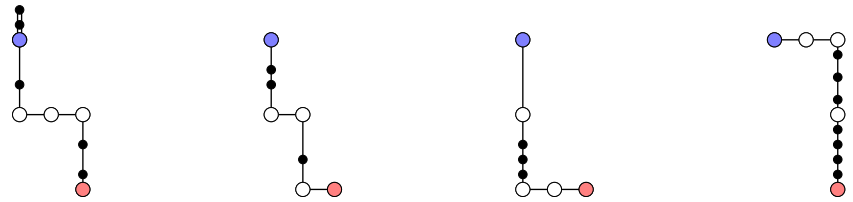


Figure 3.3: Four examples of stair paths between the depot (marked in red), and an intersection (marked in blue) for different numbers of items to pick.

These paths are generated using a typical algorithm for solving the knapsack problem. The interested reader can find more information about the knapsack problem and algorithms for solving it in [19, Ch. 6.4].

After generating these paths, we look for two such paths, P_1 and P_2, that end at intersections I_1 and I_2 respectively, where I_1 and I_2 are in the same horizontal aisle, and for which $|\mathcal{I}(P_1) \cup \mathcal{I}(P_2)| = t$, and $\mathcal{I}(P_1) \cap \mathcal{I}(P_2) = \emptyset$. From these, we create the picking tour $T = P_1 \cup (I_1, I_2) \cup P_2$.

With p_{f}^1 as the product node in P_1 farthest from the depot, and p_{f}^2 as its analogue in P_2, we have

$$\begin{aligned} L(T) &= L(P_1) + d(I_1, I_2) + L(P_2) \\ &= d(D, p_{\mathrm{f}}^1) + d(p_{\mathrm{f}}^1, I_1) + d(I_1, I_2) + d(I_2, p_{\mathrm{f}}^2) + d(p_{\mathrm{f}}^2, D)\,, \end{aligned} \tag{3.3}$$

where $d(a, b)$ is the distance from node a to node b. From Lemma 6, we have the lower bound on picking tour lengths

$$L(T) \geq d(D, p_i) + d(p_i, p_j) + d(p_j, D)\,, \tag{3.4}$$

for any product nodes $p_i, p_j \in T$. The tour length of our stair-based picking tour is exactly equal to this lower bound if

$$d(p_{\mathrm{f}}^1, I_1) + d(I_1, I_2) + d(I_2, p_{\mathrm{f}}^2) = d(p_{\mathrm{f}}^1, p_{\mathrm{f}}^2)\,; \tag{3.5}$$

in other words, the stair-based picking tour has the optimal route if the path taken between p_{f}^1 and p_{f}^2 in T really is the shortest path between the two.

Now, that isn't necessarily the case, but it often is; if p_{f}^1 is north of I_1 and p_{f}^2 is south of I_2 or vice versa, the path taken between p_{f}^1 and p_{f}^2 in T is indeed the shortest possible path. If p_{f}^1 is in the subaisle immediately south of I_1, and p_{f}^2 is in the subaisle immediately south of I_2, meanwhile (or if both are in the subaisles immediately north of the respective intersections), the path taken between them in T is the shortest path possible if

$$d(p_{\mathrm{f}}^1, I_1) + d(I_2, p_{\mathrm{f}}^2) \leq d(p_{\mathrm{f}}^1, \bar{I}_1) + d(\bar{I}_2, p_{\mathrm{f}}^2)\,, \tag{3.6}$$

where $\bar{I}_1$ is the intersection on the opposite side of p_{f}^1 from I_1, and $\bar{I}_2$ is defined analogously for p_{f}^2.

To check that one of these cases holds—and thereby verify that the path combination we are considering has the shortest length possible for a tour that picks the specific items it picks—is trivial. For any path combination T we construct for which one of the cases hold, and which also has a negative reduced cost, we add the corresponding column to our restricted column set.

The algorithm is described in Algorithm 3. The algorithm as outlined here does not consider potential branching constraints that may be in effect. Depending on the implementation in the MILP solver, it may not be problematic per se to generate columns that violate the local branching constraints here, just pointless, and unnecessarily memory intensive. To deal with the branching constraints properly, we don't just store the best path for every intersection and valid product amount, but for every intersection, valid product amount, and *constraint status vector*, as defined in Subsection 2.2.4. The details of this extension are omitted here, but implementing the extension is relatively easy in practice, and was done in the code for this project.

3.3 The reduced cost-optimal subproblem algorithm

To determine whether we have fully optimized the current branch of our search tree, to calculate lower bounds, and to ensure we can achieve optimality, we need an algorithm that solves the subproblem to optimality. For this purpose, we extend the algorithm developed by Pansart, Cambazard and Catusse for the *picker routing problem* [4], which is the problem in which one wants to find the shortest picking tour possible for a given set of items. What follows is essentially a copy of their algorithm, but with some modifications that let us pick a subset of a given cardinality of all available items, while optimizing the reduced cost of the column associated with the resulting picking tour and adhering to all Ryan-Foster branching constraints that are in effect. Some additional changes to the algorithm are made in Subsections 3.3.1 and 3.3.2, which are unique for this project. We call this algorithm "the reduced cost-optimal subproblem algorithm", since it generates the column with the minimal reduced cost for the restricted master problem.

The goal here will be to find a valid picking tour of cardinality t, $T = \bigcup_{\tau \in T} \tau$, where t is the trolley capacity, consisting of transitions τ as described in Subsection 2.2.3, with minimal reduced cost, where the reduced cost of the tour T is

$$\bar{c}(T) = \sum_{\tau \in T} \bar{c}(\tau) = \sum_{\tau \in T} \left(L(\tau) - \sum_{i \in \mathcal{I}(\tau)} u_i \right), \tag{3.7}$$

where $L(\tau)$ is the distance walked in τ, $\mathcal{I}(\tau)$ the set of items picked by τ, and u_i the optimal value for the dual variable associated with item i.

To start, we use the graph representation of the warehouse explained in Subsection 2.2.1 and number every subaisle. We number them such that the farther west a subaisle is, the lower its number, and for subaisles equally far west, we give lower numbers to subaisles farther south. We call these numbers the *layer indices* of the subaisles. An example of a layer index numbering is shown in Figure 3.4.

Algorithm 3 Stair-based subproblem algorithm

$t \leftarrow$ The trolley capacity.
$v, h \leftarrow$ The number of vertical and horizontal aisles in the warehouse, respectively.
$D \leftarrow$ The depot.
$v_D \leftarrow$ The index of the vertical aisle that D lies in.
$I_{i,j} \leftarrow$ The intersection between the ith horizontal aisle and the jth vertical aisle.
$\bar{c}(a) \leftarrow$ The reduced cost of tour, path, or transition a.
$\mathcal{I}(a) \leftarrow$ The set of items picked by tour, path, or transition a.
$L(T) \leftarrow$ Length of tour T.
$d(a, b) \leftarrow$ Distance between nodes a and b.
$I(P) \leftarrow$ Final intersection in path P.
$p_{\mathrm{f}}(P) \leftarrow$ Final product node in path P.

for $i \in [1, 2, \dots, h]$ **do**
  for $j \in [1, 2, \dots, v]$ **do**
    if $i = 1$ **then**
      $P^0_{1,j} \leftarrow (D, I_{1,j})$.
    $s \leftarrow$ The subaisle between $I_{i,j}$ and $I_{i+1,j}$.
    $\mathcal{T}^{\uparrow}_s \leftarrow$ The set of single transitions for s.
    for $k \in [0, 1, \dots, t]$ **do**
      $\mathcal{T}^{\uparrow,k}_s \leftarrow \{\tau \in \mathcal{T}^{\uparrow}_s : |\mathcal{I}(\tau)| = k\}$
      $\tau^k_s \leftarrow \operatorname{argmin}_{\tau \in \mathcal{T}^{\uparrow,k}_s} \bar{c}(\tau)$.
      for $\ell \in [\min(v_D, j), \min(v_D, j) + 1, \dots, \max(v_D, j)]$ **do**
        for $m \in [0, 1, \dots, t - k]$ **do**
          $P_{\mathrm{new}} \leftarrow P^m_{i,\ell} \cup (I_{i,j}, I_{i,\ell}) \cup \tau^k_s$.
          if $P^{m+k}_{i+1,\ell}$ undefined **then**
            $P^{m+k}_{i+1,\ell} \leftarrow P_{\mathrm{new}}$.
          else
            $P_{\mathrm{old}} \leftarrow P^{m+k}_{i+1,\ell}$.
            $P^{m+k}_{i+1,\ell} \leftarrow \operatorname{argmin}_{P \in \{P_{\mathrm{new}}, P_{\mathrm{old}}\}} \bar{c}(P)$.
    $\mathcal{T}^{\mathrm{SIO}}_s \leftarrow$ The set of southern in-and-out transitions for s.
    for $k \in [1, 2, \dots, t]$ **do**
      $\mathcal{T}^{\mathrm{SIO},k}_s \leftarrow \{\tau \in \mathcal{T}^{\mathrm{SIO}}_s : |\mathcal{I}(\tau)| = k\}$.
      $\tau^k_s \leftarrow \operatorname{argmin}_{\tau \in \mathcal{T}^{\mathrm{SIO},k}_s} \bar{c}(\tau)$.
      for $m \in [1, 2, \dots, t - k]$ **do**
        $P_{\mathrm{new}} \leftarrow P^m_{i,\ell} \cup \tau^k_s$.
        $P_{\mathrm{old}} \leftarrow P^{m+k}_{i,\ell}$.
        $P^{m+k}_{i,\ell} \leftarrow \operatorname{argmin}_{P \in \{P_{\mathrm{new}}, P_{\mathrm{old}}\}} \bar{c}(P)$.
$\mathcal{B} \leftarrow \emptyset$.
for $\{P_1, P_2\} \in \{\{P^k_{i,j}, P^{t-k}_{i,\ell}\} : P^k_{i,j}$ and $P^{t-k}_{i,\ell}$ defined, $\mathcal{I}(P^k_{i,j}) \cap \mathcal{I}(P^{t-k}_{i,\ell}) = \emptyset\}$ **do**
  $T \leftarrow P_1 \cup (I(P_1), I(P_2)) \cup P_2$.
  if $L(T) = d(D, p_{\mathrm{f}}(P_1)) + d(p_{\mathrm{f}}(P_1), p_{\mathrm{f}}(P_2)) + d(p_{\mathrm{f}}(P_2), D)$ and $\bar{c}(T) < 0$ **then**
    $\mathcal{B} \leftarrow \mathcal{B} \cup (L(T), \mathcal{I}(T))$.
return $\mathcal{B}$.

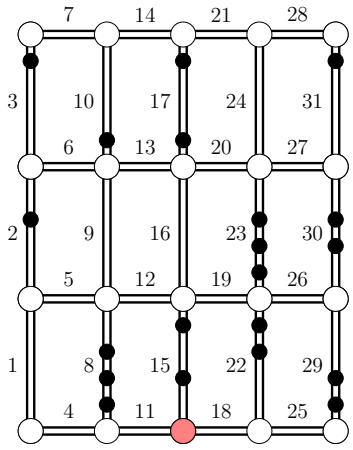


Figure 3.4: An example of a warehouse graph with subaisles numbered by layer index, with only a subset of the edges drawn for legibility.

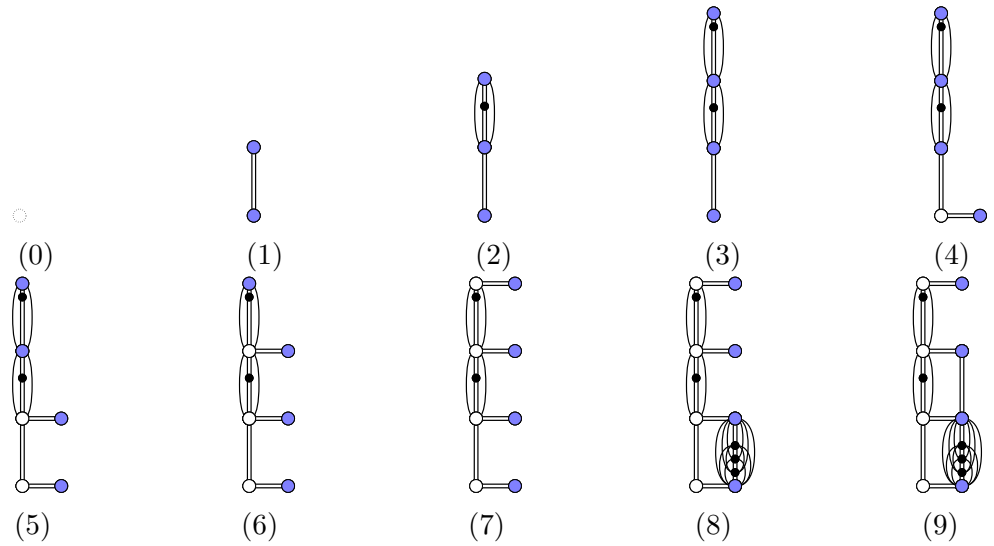


Figure 3.5: The subgraphs associated with the first ten layer indices for the warehouse graph in Figure 3.4 with the corresponding borders marked in blue.

We also associate a subgraph with every layer index; for layer index i, we have the associated subgraph

$$F_i = \bigcup_{j=1}^{i} s_j \,, \tag{3.8}$$

where s_j is the subaisle with layer index j. Finally, every layer index has an associated border, which naturally is the border of F_i for layer i. Examples of this are shown in Figure 3.5.

Using these definitions, we can define states, which will help us to explore partial picking tours in an organized manner.

Definition 12. The *state*, σ, of an F_i partial picking tour, P, is defined by its border parity vector, $B_{\text{par}}(P)$; border component index vector, $B_{\text{comp}}(P)$; constraint status vector, $S(P)$; set of items picked, $\mathcal{I}(P)$, and its layer index,

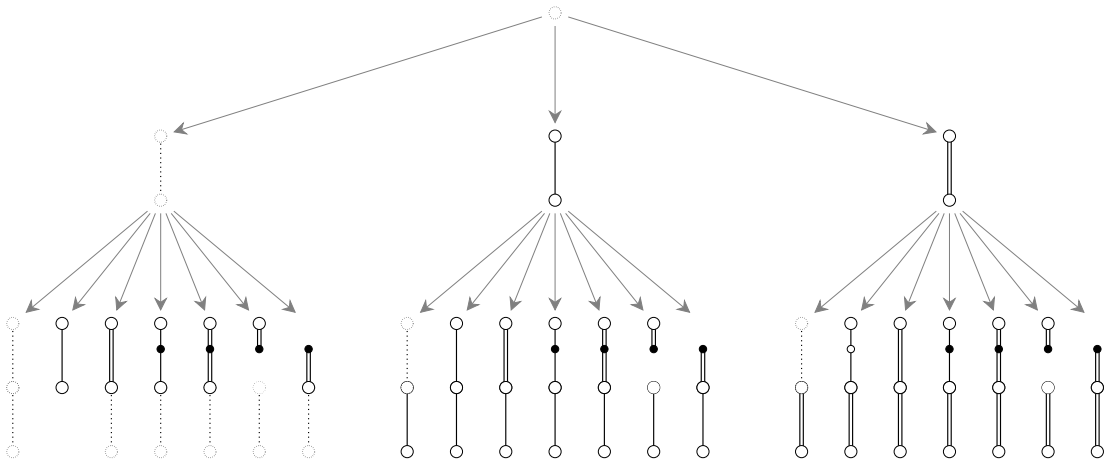


Figure 3.6: All partial picking tours for layers 0, 1, and 2 for the warehouse graph in Figure 3.4, with empty transitions drawn as dotted single transitions.

$\ell(P) = i$.

A state, σ, is a *complete state*, and corresponds to a valid picking tour, if the following criteria are met:

- $B_{\text{par}}(\sigma) = \vec{0}$
- $B_{\text{comp}}(\sigma)$ contains only $-$ or 1
- $S(\sigma)$ contains only *cleared* statuses
- $|\mathcal{I}(\sigma)| = t$
- $\ell(\sigma) = \ell_{\max}$, where $\ell_{\max}$ is the highest layer index for the warehouse graph,

where $B_{\text{par}}(\sigma) = B_{\text{par}}(P)$, $B_{\text{comp}}(\sigma) = B_{\text{comp}}(P)$, etc.

Now, we can explore the set of all relevant partial picking tours in an orderly fashion, layer by layer; for layer 1, we create all F_1 partial picking tours, i.e., every partial picking tour that consists of just one of the transitions for subaisle 0. For layer 2, we create every possible F_2 partial picking tour by simply combining every partial picking tour from layer 1 with the transitions available for subaisle 1—provided that the combination doesn't violate any of the rules for a valid partial picking tour—and so on.

If at any point we generate a partial picking tour for which we have already generated another partial picking tour with the same state, we know from Theorem 2 that we can discard the more expensive one; since they have the same layer index, i, they are also picking tours for the same subgraph, F_i, and since they also have the same equivalence class, they are equivalent. If two partial picking tours with the same state have the same cost, we choose one to discard arbitrarily.

Finally, when we have gone through all layers, we simply retrieve the picking tour in the final layer with the lowest reduced cost. The partial tours generated in layers 0, 1, and 2 are shown in Figure 3.6.

This algorithm is very slow, of course; there is an immense number of valid possible states, and we need to explore all of them. To make this at all useable in practice,

we need to make some changes, which are outlined in the following sections.

3.3.1 Reducing the problem

In this subsection, we look at a couple of ways to simplify the reduced cost-optimal subproblem algorithm.

3.3.1.1 Making branching constraints redundant

Sometimes, we have a branching constraint that applies to two items that are in the same subaisle. In that case, we can make sure we never violate it by simply removing every transition that would violate it (i.e., any transition that picks both of these items in a force-different constraint, or picks one item but not the other in a force-same constraint). In practice, whenever we branch, we copy the graph—which has all valid transitions stored—check if both items in the new constraint are in the same subaisle, and, if so, remove the violating constraints. If they aren't in the same subaisle, we store the constraint in the graph (along with any previously introduced branching constraints) and include their statuses in our state definition.

3.3.1.2 Excluding double vertical transitions

Somewhat counterintuitively, it turns out we can often discard all double transitions in vertical subaisles. This is the case if our problem is set in a warehouse with a *grid layout*, which is typically the case.

Definition 13. Take a warehouse with a set of intersections and subaisles. Create two infinite lines through each intersection, one running vertically and one running horizontally. If there is a bijection between the intersections of these infinite lines and the intersections in the warehouse, and a bijection between the segments of these lines that run between the intersections and the subaisles in the warehouse, the warehouse has a *grid layout*.

Less formally, a warehouse has a grid layout if its intersections and subaisles form a grid with 90° angles without any gaps. All warehouses we've looked at so far have had a grid layout.

The reason we care about whether the warehouse has a grid layout or not is that in warehouses with a grid layout (apart from trivial cases with just one aisle), every horizontal subaisle has at least one parallel subaisle that starts equally far west and ends equally far east, and every vertical subaisle has at least one vertical parallel subaisle analogously. Two such parallel subaisles have the same length, and so we can sometimes move transitions between them without changing their lengths, thereby making changes to our picking tour without increasing its reduced cost.

Theorem 3. For a warehouse graph, G, with a grid layout, vertical double transitions are necessary to construct the optimal picking tour only if all product nodes in the optimal picking tour lie in the same vertical aisle as the depot.

Proof of Theorem 3. Assume that we have an optimal picking tour, $T \subseteq G$, with at least one product node $p_i \in T$ that is not in the same vertical aisle as the depot, and that there is at least one vertical double transition in T.

This vertical double transition lies in a sequence of vertical double transitions

$(\tau_1, \tau_2, \dots, \tau_n)$ with $1 \leq n \leq h-1$ with h as the number of horizontal aisles in G, and where

$$I_{\mathrm{N}}(\tau_i) = I_{\mathrm{S}}(\tau_{i+1}),\ \forall i \in \{1, 2, \dots, n-1\}\,, \tag{3.9}$$

with $I_{\mathrm{S}}(\tau)$ as the southern intersection in τ, and $I_{\mathrm{N}}(\tau)$ as the northern intersection, and no $I_{\mathrm{N}}(\tau) \in \{\tau_1, \tau_2, \dots, n-1\}$ is also part of a horizontal single or double transition in T (i.e., no intersection in this sequence apart from the ones at the ends may be in any other transition).

Now, $I_{\mathrm{N}}(\tau_n) \in \tau_{\mathrm{N}}$ for some $\tau_{\mathrm{N}} \neq \tau_n$, $\tau_{\mathrm{N}} \in T$; if τ_n were the only thing connecting to this intersection, T could be improved by replacing τ_n with the southern in-and-out transition that picks the same items, making T suboptimal, giving a contradiction.

τ_{N} must either be a southern in-and-out transition, or a horizontal single or double transition; we could have another vertical double transition, but since τ_{N} isn't in our sequence of vertical double transitions, that would mean that $I_{\mathrm{N}}(\tau_n)$ is in a horizontal single or double transition. τ_{N} could also be a vertical single transition, but that would imply that $I_{\mathrm{N}}(\tau_n)$ is in a horizontal single transition as well, since $I_{\mathrm{N}}(\tau_n)$ would have odd parity otherwise.

We have a similar situation for $I_{\mathrm{S}}(\tau_1)$, but there may not actually be a $\tau_{\mathrm{S}} \neq \tau_1$, $\tau_{\mathrm{S}} \in T$ for which $I_{\mathrm{S}}(\tau_1) \in \tau_{\mathrm{S}}$; if $I_{\mathrm{S}}(\tau_1)$ is the depot, D, replacing τ_1 with a northern in-and-out may mean $D \notin T$, making T invalid.

All in all, this leaves us with the following possibilities:

1. $I_{\mathrm{S}}(\tau_1) = D$
2. $\exists \tau_{\mathrm{S}}$ that is a horizontal single or double transition for which $I_{\mathrm{S}}(\tau_1) \in \tau_{\mathrm{S}}$
3. $\exists \tau_{\mathrm{N}}$ that is a horizontal single or double transition for which $I_{\mathrm{N}}(\tau_n) \in \tau_{\mathrm{N}}$
4. $\exists \tau_{\mathrm{N}}$ that is a southern in-and-out for which $I_{\mathrm{N}}(\tau_n) \in \tau_{\mathrm{N}}$
5. $\exists \tau_{\mathrm{S}}$ that is a northern in-and-out for which $I_{\mathrm{S}}(\tau_1) \in \tau_{\mathrm{S}}$.

If neither alternative 2 nor 3 is true, that must mean that

$$T = \tilde{T} := \bigcup_{i=1}^{n} \tau_i \cup \tau_{\mathrm{S}} \cup \tau_{\mathrm{N}}\,, \tag{3.10}$$

since none of the intersections in $\tilde{T}$ are part of any other transitions, leaving it isolated, and there cannot be two unconnected components in T. However, $\tilde{T}$, is located entirely within one vertical aisle, but there are product nodes, p_i, in different vertical aisles from D, meaning that either $D \notin \tilde{T}$, or $\exists p_i \in T,\ p_i \notin \tilde{T}$, meaning $T \neq \tilde{T}$, giving a contradiction.

That must mean that we have a horizontal single or double transition $\tau_{\mathrm{h}} \in T$ for which $I_{\mathrm{S}}(\tau_1) \in \tau_{\mathrm{h}}$ or $I_{\mathrm{N}}(\tau_n) \in \tau_{\mathrm{h}}$. Assume that it's the case that $I_{\mathrm{N}}(\tau_n) \in \tau_{\mathrm{h}}$ (the proof is symmetrical for $I_{\mathrm{S}}(\tau_1) \in \tau_{\mathrm{h}}$).

If τ_{h} is a single transition, remove it from T, and replace it with another single transition, τ_{h}^1, in the subaisle one step south, so that $I_{\mathrm{S}}(\tau_n) \in \tau_{\mathrm{h}}^1$. If it is a double transition, remove it from T, and replace it with a single transition in the same place, τ_{h}^2, and create τ_{h}^1 one step south as in the other case. Then, replace

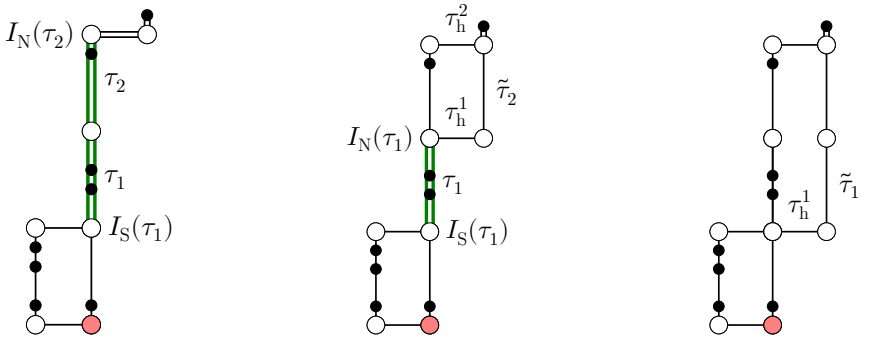


Figure 3.7: From left to right, an optimal picking tour with double vertical transitions being modified in steps to avoid the double vertical transitions, marked in green.

τ_n with a single transition in the same subaisle that picks the same products, and create another vertical single transition, $\tilde{\tau}_n$, one subaisle to the west or east, such that $I_S(\tilde{\tau}_n) \in \tau_h^1$, and $I_N(\tilde{\tau}_n) \in \tau_h^2$, if τ_h^2 has been created.

The only intersections whose parity or connectedness has been affected by this change are $I_S(p_n), I_N(p_n), I_S(\tilde{p}_n)$, and $I_N(\tilde{p}_n)$, but these are all still valid; our changes increased or decreased the degree in each one of these nodes by zero or two, and all four nodes are connected, meaning that this change did not make T invalid.

This process has not increased $\bar{c}(T)$, the reduced cost for T, but has reduced the sequence of vertical double transitions. Repeating this for all sequences of vertical double transitions, we eventually end up with no vertical double transitions, without having made T non-optimal.

We could potentially run into a problem wherein one of the vertical transitions created, $\tilde{\tau}_i$ is in a subaisle with a pre-existing non-empty transition, $\hat{\tau}_i \in T$. If $\hat{\tau}_i$ is a vertical single transition, we remove it and make $\tilde{\tau}_i$ a double transition that picks $\mathcal{I}(\hat{\tau}_i)$. If $\hat{\tau}_i$ is any other kind of transition, we could make $\tilde{\tau}_i$ a single transition that picks the same items, but that would reduce $\bar{c}(T)$, contradicting the assumption that T is optimal.

Of course, replacing $\hat{\tau}_i$ with a double transition, $\tilde{\tau}_i$, has a flaw; doing so creates a new double transition that we have to remove, and we may end up recreating τ_i when we try to get rid of $\tilde{\tau}_i$, creating an infinite loop when trying to iteratively eliminate these vertical double transitions. These loops can be avoided, but the proof for this special case is omitted here. □

An example of this process is shown in Figure 3.7.

Now, this reduction does create a problem: we no longer solve the subproblem to optimality if the column with minimal reduced cost only contains items in the same vertical aisle as the depot. This isn't a problem in practice, however, since such columns can be generated by the stair-based subproblem algorithm (see Section 3.2),

and it is, in fact, guaranteed to generate the optimal such batch. Since we always run the stair-based subproblem algorithm before the reduced cost-optimal subproblem algorithm in every iteration—only running the reduced cost-optimal subproblem algorithm if the stair-based algorithm fails to generate any columns with negative reduced cost—we know that there are no columns only containing items in the same vertical aisle as the depot with negative reduced cost possible when the reduced cost-optimal subproblem algorithm runs.

3.3.1.3 Excluding dually suboptimal transitions

One of the most important modifications we make is to "ignore dually suboptimal transitions". To do this, we create an alternative definition for a transition's equivalence class, as previously defined in Subsection 2.2.3.1: the *known-constraint equivalence class* for a transition, τ, is the same as its equivalence class, except instead of the set of product nodes in τ, $\mathcal{I}(\tau)$, we use $|\mathcal{I}(\tau)|$ and $\mathcal{I}_{\mathrm{C}} \cap \mathcal{I}(\tau)$, where $\mathcal{I}_{\mathrm{C}}$ is the set of items which are constrained by the branching constraints that are in effect. In other words, we no longer differentiate transitions based on exactly which items they pick, but only the total number and which constrained items they pick.

Then, we simply discard every transition, τ_1, for which there is another transition, τ_2 of the same known-constraint equivalence class in the same subaisle with $\bar{c}(\tau_2) < \bar{c}(\tau_1)$; any picking tour T for which $\tau_1 \in T$ can be improved by replacing τ_1 with τ_2; they have the same effect in terms of parity and connectedness for the intersections in their subaisle, they pick the same number of items, and they interact with the branching constraints in exactly the same way, since they pick the same set of constrained items. As such, τ_1 will never be of interest, and can be ignored.

If $\bar{c}(\tau_1) = \bar{c}(\tau_2)$, only one of τ_1 and τ_2 needs to be kept, and we can choose which one to keep arbitrarily.

This reduction usually simplifies the problem quite dramatically; originally, we had $\mathcal{O}(5 \cdot 2^n)$ transitions for a subaisle with n product node, since we had 2^n ways of choosing which items to pick, and at most five relevant transitions for every such item combination. We then reduced this to $\mathcal{O}(4 \cdot 2^n)$ in Subsection 3.3.1.2 by getting rid of the vertical double transitions, and now we have reduced it to just $\mathcal{O}(4n)$.

3.3.2 Applying the A* algorithm

One way of thinking about the reduced cost-optimal subproblem algorithm is that it converts the problem from a prize-collecting travelling salesperson problem to a shortest path problem; in a sense, we are trying to find the shortest path through a graph of states (similar to what is shown in Figure 3.6, but extended to cover all layers and in terms of states, not partial picking tours) from the initial state to any complete state, where the path length is given by the sum of the reduced costs of the transitions used. If we think about the problem in those terms, it is quite clear that our approach is inefficient; we are essentially solving a shortest path problem through a breadth-first search (BFS), which is typically a bad approach.

To improve upon the algorithm, we replace this BFS approach with the A* algorithm as described in Algorithm 1, which is typically much more efficient. Recalling the theory regarding the A* algorithm presented in Section 2.3, we know that we need a function that gives a heuristic value for the remaining cost to a terminal node

(in this case, a complete state). Furthermore, for the A* algorithm to guarantee optimality, our heuristic function needs to be consistent, since we can have negative edge weights (as some transitions will have negative reduced costs).

Now, to get from a state σ to a complete state, we want to construct a tour completion, C gives a complete state $\sigma \cup C$. There is some notational abuse here; we're talking about creating a tour completion to a state, but tour completions are made for partial picking tours. What we mean is that we want to create a tour completion for any partial picking tour with state σ; remember, all equivalent picking tours are compatible with the exact same set of tour completions, hence the conflation.

The heuristic function is supposed to emulate the process of finding a tour completion. To do this, we first relax our problem; we combine $B_{\text{par}}(\sigma)$ and $B_{\text{comp}}(\sigma)$ into a *heuristic connection requirement*, K, that contains some information necessary (but not all) to find a completion C that gives valid $B_{\text{par}}(\sigma \cup C)$ and $B_{\text{comp}}(\sigma \cup C)$.

The idea now is to find a tour completion C, but using K instead of $B_{\text{par}}(\sigma)$ and $B_{\text{comp}}(\sigma)$, which is much easier. We do this with a dynamic programming algorithm that recursively explores the set of possible tour completions in this relaxed problem.

Before we do that, however, we need to define K. With h nodes in our border node set, K is a vector containing $h-1$ elements, each corresponding to a "row" of vertical subaisles in the warehouse. More specifically, these entries correspond to the amount of single vertical transitions we need to use for that specific row to connect all border nodes and achieve even parity for all border nodes.

We start by identifying any border nodes with odd parity, and pair them up two-and-two from the south going north, saying that we will want to give these nodes even parity by connecting each such pair with *one* single transition for each row between them, which gives those rows a connection requirement of 1.

We then assume these connections will be made, and consider the border component index vector as it would look with those connections having been made. If this component vector contains $B^{\text{u}}_{\text{comp}}$ unique component indices, we need to make $B^{\text{u}}_{\text{comp}} - 1$ connections between components to end up with just one component, but exactly how those connections are best made is unclear at this stage. As such, we generate every possible way of making these connections without any overlap (essentially finding every relevant set of $B^{\text{u}}_{\text{comp}} - 1$ pairs of border nodes such that connecting these pairs would leave us with just one component, without creating redundant connections), and for each one, we create a connection requirement vector in which we set the connection requirement to 2 for every row between the nodes in the pairs that we need to connect, since we need to use two single transitions for these rows to connect the unconnected components without giving some nodes odd parity.

Here is an example of what this looks like. In each segment, the first vector is $B_{\text{par}}(\sigma)$, the second is $B_{\text{comp}}(\sigma)$ (but with the modifications described above after the first step), and the third is K:

$$\begin{bmatrix}1\\0\\1\\0\end{bmatrix}, \begin{bmatrix}2\\3\\2\\1\end{bmatrix}, \begin{bmatrix}0\\0\\0\end{bmatrix} \rightarrow \begin{bmatrix}1\\0\\1\\0\end{bmatrix}, \begin{bmatrix}2\\2\\2\\1\end{bmatrix}, \begin{bmatrix}1\\1\\0\end{bmatrix} \rightarrow \begin{bmatrix}1\\0\\1\\0\end{bmatrix}, \begin{bmatrix}2\\2\\2\\1\end{bmatrix}, \begin{bmatrix}1\\1\\2\end{bmatrix}. \tag{3.11}$$

And, as another example:

$$\begin{bmatrix}0\\0\\0\\1\\1\\0\end{bmatrix}, \begin{bmatrix}2\\3\\2\\1\\1\\1\end{bmatrix}, \begin{bmatrix}0\\0\\0\\0\\0\end{bmatrix} \rightarrow \begin{bmatrix}0\\0\\0\\1\\1\\0\end{bmatrix}, \begin{bmatrix}2\\3\\2\\1\\1\\1\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\\0\end{bmatrix} \rightarrow \begin{bmatrix}0\\0\\0\\1\\1\\0\end{bmatrix}, \begin{bmatrix}2\\3\\2\\1\\1\\1\end{bmatrix}, \begin{bmatrix}2\\0\\2\\1\\0\end{bmatrix} \text{ or } \begin{bmatrix}0\\2\\2\\1\\0\end{bmatrix}, \tag{3.12}$$

where we get two options for K in the end, since there are two ways to connect components 2 and 3.

An algorithm for generating these connection requirements is shown in Algorithm 4.

We also define a function $K(\tau)$, which gives the effect on K that a transition, τ, has; $K(\tau)$ is a zero vector of the same dimensions as K, with the element in the row corresponding to the subaisle τ is in as 1 if τ is a single transition.

Now, we further relax the problem by reducing the set of transitions for each vertical subaisle. For every vertical subaisle, we keep just the transition with the lowest reduced cost for each *transition category*, where the transition category for a transition is determined by three properties:

1. how many items it picks
2. whether it is a single transition
3. which items subject to any branching constraints it picks.

This is analogous to what we did in Subsection 3.3.1.3, but applied to this new, relaxed problem.

Of course the first entry in the category is relevant when constructing our heuristic tour completion out of transitions, since we want to pick a specific number of items in total. The second entry is relevant, because it is the vertical transitions that change the connection requirement vector, and we're trying make $K = \vec{0}$ since this is our relaxed analogue to achieving valid $B_{\text{par}}(\sigma)$ and $B_{\text{comp}}(\sigma)$. The third entry is relevant since the branching constraints can only be violated by picking or not picking items subject those constraints.

The list of valid transitions we create is denoted $\mathcal{T}_\ell$ for the subaisle with layer index ℓ.

Now, we define the heuristic cost function for a state with layer index ℓ, remaining amount of trolley slots to fill r, connection requirements K and branching constraint vector S as

$$h\left(\ell, r, K, S\right) = \begin{cases} h_{\text{v}}(\ell, r, K, S) & \text{if } s_\ell \text{ is vertical} \\ h_{\text{h}}(\ell, r, K, S) & \text{if } s_\ell \text{ is horizontal,} \end{cases} \tag{3.13}$$

where s_ℓ is the subaisle with layer index ℓ. The first of these functions is

$$h_{\text{v}}(\ell, r, K, S) = \min_{\tau \in \mathcal{T}_\ell} \left[\bar{c}(\tau) + h\left(\ell + 1, r - p(\tau), \left|K - K(\tau)\right|, S + S(\tau)\right)\right], \tag{3.14}$$

where $p(\tau)$ is the number of items picked by τ, and $S + S(\tau)$ gives the constraint status vector after including τ in a partial picking tour with constraint vector S.

Algorithm 4 Algorithm for generating heuristic connection requirements

$h \leftarrow$ The number of horizontal aisles in the warehouse.
$K \leftarrow \vec{0}_{h-1}$.
$B_{\text{par}} \leftarrow$ The parities of the border nodes.
$B_{\text{comp}} \leftarrow$ The component indices of the border nodes.
$i \leftarrow 1$.
while $i \leq h-1$ **do**
  if $B^i_{\text{comp}} = 1$ **then** ▷ B^i_{comp} is the ith element in B_{comp}.
    $j \leftarrow$ Lowest $j > i$ for which $B^j_{\text{comp}} = 1$.
    $\tilde{B}_{\text{comp}} \leftarrow \emptyset$. ▷ $\tilde{B}_{\text{comp}}$ stores the indices of components between nodes i and j.
    for $m \in [i, i+1, \dots, j]$ **do**
      if $B^m_{\text{comp}} \neq -$ **then**
        $\tilde{B}_{\text{comp}} \leftarrow \tilde{B}_{\text{comp}} \cup \{B^m_{\text{comp}}\}$.
    for $m \in [1, 2, \dots, h]$ **do**
      if $B^m_{\text{comp}} \in \tilde{B}_{\text{comp}}$ **then**
        $B^m_{\text{comp}} \leftarrow \min \tilde{B}_{\text{comp}}$.
    $i \leftarrow j + 1$.
  else
    $i \leftarrow i + 1$.
$\hat{B}_{\text{comp}} \leftarrow \emptyset$. ▷ $\hat{B}_{\text{comp}}$ stores indices of pairs of nodes to potentially connect.
for $i \in [1, 2, \dots, h]$ **do**
  $j \leftarrow$ Lowest $j > i$ for which $B^j_{\text{comp}} \neq -$.
  if $B^i_{\text{comp}} \neq B^j_{\text{comp}}$ **then**
    $\hat{B}_{\text{comp}} \leftarrow \hat{B}_{\text{comp}} \cup \{(i, j)\}$.
$B^{\text{u}}_{\text{comp}} \leftarrow$ Amount of unique indices in B_{comp}.
$\mathcal{S} \leftarrow \{S \in \binom{\hat{B}_{\text{comp}}}{B^{\text{u}}_{\text{comp}}-1} : \text{Connecting every pair in } S \text{ leaves only one component in } B_{\text{comp}}\}$.
$\mathcal{K} \leftarrow \emptyset$.
for $S \in \mathcal{S}$ **do**
  $\Gamma \leftarrow K$
  for $(i, j) \in S$ **do**
    for $m \in [i, i+1, \dots, j-1]$ **do**
      $\Gamma^m \leftarrow 2$.
  $\mathcal{K} \leftarrow \mathcal{K} \cup \{\Gamma\}$.
return $\mathcal{K}$.

The second function is

$$h_{\mathrm{h}}(\ell, r, K, S) = \left|K_{i(s_\ell)} - K_{i(s_\ell)-1}\right| L(s_\ell) + h\left(\ell + 1, r, K, S\right) , \tag{3.15}$$

where $i(s_\ell)$ is the index of the element in K corresponding to the row s_ℓ is in, K_j is the jth element in K, with $K_0 = K_h = 0$, and $L(s_\ell)$ is the length of s_ℓ.

Eq. (3.15) has its form mainly because the connection requirements are very vague in terms of what horizontal connections are needed; $K = [2, 2]$, for example, can be the result of a state σ with $B_{\mathrm{par}}(\sigma) = \vec{0}$, and $B_{\mathrm{comp}}(\sigma) = [1, -, 2]$ or $B_{\mathrm{comp}}(\sigma) = [1, 2, 3]$. The first case would require two horizontal double transitions as to not isolate any component, while the second case would require three horizontal double transitions. Since we need the heuristic to be admissible, we must assume the situation that permits the cheapest connections. As such, we assume that any contiguous sequence of twos is caused by just two unconnected components (with nothing of relevance in between), and as such requires two double transitions. We similarly assume that a contiguous sequence of ones requires just two single transitions.

To ensure that these requirements are handled consistently, for any contiguous non-zero sequence in K, we position the two required single or double transitions at the ends of the sequence. If a sequence of ones and a sequence of twos are adjacent, we position a single transition in their "seam", seeing as placing a double transition there would violate the parity requirement for the corresponding border node. This choice of whether to use an empty, single, or double transition is handled by the expression $\left|K_{i(s_\ell)} - K_{i(s_\ell)-1}\right|$ in Eq. (3.15), since this evaluates to 1 whenever a 1 is adjacent to a 0 or a 2 in K, and 2 whenever a 2 is adjacent to a 0 in K (once again, using $K_0 = K_h = 0$).

Furthermore, we have

$$h\left(\ell_{\max}, r, K, S\right) = \begin{cases} 0 & \text{if } r = 0,\ K = \vec{0} \text{ and all statuses in } \vec{B} \text{ are } \textit{cleared}, \\ \infty & \text{otherwise.} \end{cases} \tag{3.16}$$

The heuristic in Eq. (3.13) is consistent, meaning that we can apply the A* algorithm with this heuristic function, despite having edges with negative edge weights. We won't prove this fully, but essentially, it is consistent since the graph onto which it is applied (the shortest path state graph mentioned in this subsection) is acyclic, and the heuristic cost function recursively underestimates the cost for each subgraph that it operates on.

The heuristic function as described so far is terribly slow; every time we call it, it recursively explores a massive number of heuristic states, but luckily, we can circumvent this problem easily by using *memoization*; whenever we call the function, we store the arguments we called it with and the value it returned in a cache. If we later call the function again with the same arguments, we can look up the value it returned last time and simply return it again, without making any further recursive calls. With this modification, the heuristic function tends to require a lot of recursive calls the first time it is called, but after that, the number of calls stays very low.

Of course, storing these values means using up more memory, but most memoization libraries allow for cache limits, letting the programmer tell it to only keep a certain number of arguments and return values in memory, and how to free memory once this cap is reached up, where a common approach is to discard the least recently used cache entry to free up more memory.

3.4 Ryan-Foster branching candidate selection

When applying Ryan-Foster branching, we will first identify pairs of items for which we can create branching constraints, as described in Subsection 2.1.1.1. We will refer to such item pairs as *branching candidates*.

The choice of which branching candidate to branch on turns out to be quite relevant; choosing a candidate for which both items lie in the same subaisle, for example, is very convenient, since the problem reduction in Subsection 3.3.1.1 handles such constraints once, making them completely irrelevant going forward.

Furthermore, to ensure the optimality of the reduced cost-optimal subproblem algorithm, we need to double the number of transitions kept when excluding dually suboptimal transitions for a subaisle—as described in Subsection 3.3.1.3—for each constraint with just one item in that subaisle, since we need to retain the best transition for every known-constraint equivalence class in each subaisle, and making a new item in the subaisle constrained doubles the number of possible known-constraint equivalence classes. As such, it is a good idea to try to keep the number of constraints with just one item in the subaisle as low as possible for every subaisle.

On top of this, whenever we add a new branching constraint, we potentially create new implicit constraints. For example, if we have a force-same constraint between items 1 and 2 and create another force-same constraint between items 2 and 3, we get an *implicit* force-same constraint between items 1 and 3. Anecdotal evidence indicates that having more implicit constraints means that the reduced cost-optimal subproblem algorithm becomes faster, and that the column generation converges more quickly, and so maximizing the number of these implicit constraints can also be of use.

Finally, more anecdotal evidence indicates that branching candidates for which the items i, j result in a sum $\sum_{k \in \mathcal{K}'(i,j)} \lambda_k$ that is close to 0.5 also yield faster convergence, likely since a sum close to 0.5 means the two items appear about as often in the same batch as they do separately, and so the force-same and the force-different branch have similar impacts on the optimal value of the RLP.

All in all, these ideas are used to create a candidate priority scheme based on the following rules, where each rule takes precedence over every subsequent rule:

1. Candidates for which both items lie in the same subaisle are preferred over candidates for which that is not the case.
2. Candidates for which the items share their subaisle with fewer items in other, pre-existing branching constraints are preferred.
3. Candidates whose branching constraints would create more implicit constraints are preferred

4. Candidates with items i, j for which $\sum_{k \in \mathcal{K}'(i,j)} \lambda_k$ is closer to 0.5 are preferred.

This scheme hasn't been evaluated fully, and likely has room for improvement. This is discussed further in Subsection 5.2.5.

4 Tests, results, and discussion

In this chapter we present the results, and discuss them. The results are based on a comparison between batches and picking routes generated by the column generation algorithm, and real-world data. The real-world data was gathered during the month of March 2023 at a warehouse operated by one of Ongoing Warehouse's customers (briefly described in Subsection 1.1.1), and includes information about the warehouse layout, the locations of all items picked during the time period, and how these items were batched.

In Section 4.1 we discuss some modifications that are made to the real-world data to make the comparison more useful. We briefly present the hardware that was used for the tests, along with the software used for the implementation in Section 4.2, followed by some further descriptions of the tests run for evaluation in Section 4.3. Finally, we will look at the results, and discuss them in Section 4.4.

4.1 Treating the data

As mentioned in Subsection 1.1.1, the warehouse of study is divided into smaller zones where workers pick items from only one zone at a time. For our comparisons, we will use data from just one of these zones.

Looking at the data, it is clear that many of the batches generated in reality were non-full; that is, many batches for the five item workflow contained four or fewer items, and many batches for the 20 item workflow contained 19 items or fewer. The exact reasons for this are unknown, but could relate to urgency for certain orders, a lack of items to pick in the zone, or other logistical reasons. These reasons would be difficult to take into consideration when making a comparison—particularly since the exact reasons are unknown—but ignoring them, creating batches through column generation and comparing to the batches that were picked in reality would make for a rather poor comparison, since the column generation would outperform the real batches by default simply by ensuring that all batches are full.

To make the comparison more fair, we redefine the real batches; from the data we can tell at which time each item was assigned to its batch, which corresponds to the priority it had at the time of batching. Since this priority is the sole basis for batching, we simply take all the items in the data, sort them by the time when they were added to their batches, and go through the sorted data, creating a batch from the first five non-lone items, another from the next five, and so on, and the same for the lone items, in batches of twenty items each. As such, we ensure that the comparison will be made with full batches, while still emulating the batching algorithm used in reality. From here on, when we talk about the "current batches",

we are referring to these redefined batches.

Furthermore, it should be noted that each of the warehouse zones in the warehouse of study has only two horizontal aisles. Both the subproblem algorithms used in this project have been developed to allow for an arbitrary number of horizontal aisles, and their runtimes are strongly related to the amount of subaisles in the problem. As such, we create two "alternative warehouse zones" which have three and four horizontal aisles, respectively, and populate them with the items from the real warehouse zone we're focusing on, but randomly move each item up zero, one, or two subaisles (but retain its relative location inside its subaisle) by sampling numbers from a uniform distribution. This gives us semi-realistic data to run our tests on, but any results should be taken with a grain of salt if the model is considered for real-world implementation; realistically, items are more likely to be located in the subaisles farther south, because it is naturally preferred to store items close to the depot if one has the option of doing so (i.e., if the warehouse zone is not fully stocked), as it reduces the distance one needs to walk, both to store it there, and to retrieve it later.

4.2 Hardware and software details

The column generation was implemented using the MILP solver SCIP [20], more specifically through its Python interface, PySCIPOpt [21]. The remaining code, such the as subproblem algorithms, tests, and data manipulation, were written in Python.

All tests were run on a desktop computer with an *AMD Ryzen 5 3600X 4.4 GHz* processor, and *Corsair 2x8 GB DDR4 3200 MHz CL16 Vengeance LPX* memory. The heuristic cache used for the memoization in the reduced cost-optimal subproblem algorithm heuristic was limited to 2^{16} entries.

4.3 Test details

The model was tested with all 18 combinations of the following parameters:

- the trolley capacity, t, with $t \in \{5, 20\}$;
- the number of batches created in each run of the column generation, b, with $b \in \{2, 10, 20\}$ for $t = 5$, and $b \in \{2, 4, 6\}$ for $t = 20$;
- The number of horizontal aisles, h, with $h \in \{2, 3, 4\}$, where the tests with $h \in \{3, 4\}$ used the alternative warehouse layouts described in Section 4.1.

We'll call a combination of these parameters a *test case*.

For each test case, b current batches that were created in sequence were selected randomly, using a uniform distribution. The total route length for all b batches, denoted z_{current}, was calculated using the order in which their items were supposed to be picked in reality, which is based on the shelf priority mentioned in Subsection 1.1.1.

These routes aren't optimized to have minimal total distance, unlike the routes used in the column generation, which means that the solutions from the column generation are very likely to noticeably outperform the current batches, simply because

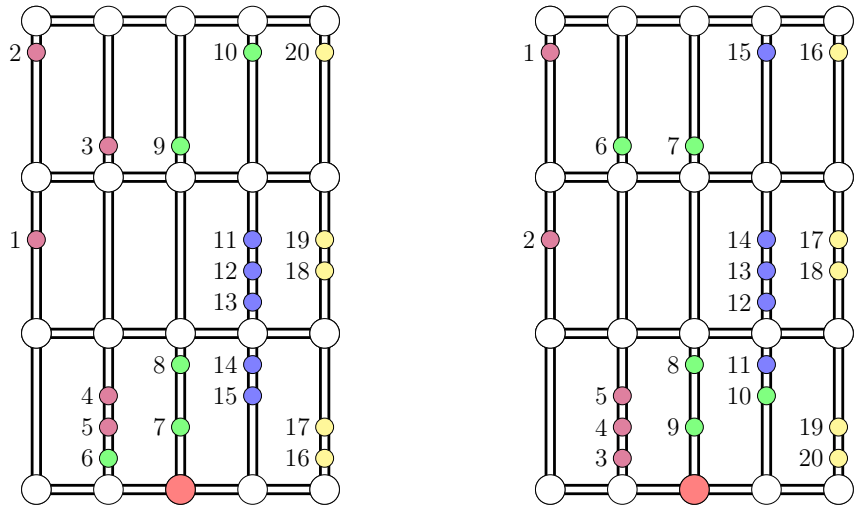


Figure 4.1: An example of the two item batching solutions generated by the simple batching algorithm, for a case with $t = 5$, $b = 4$ and $h = 4$. The product node colours represent the batches.

it actually optimizes the routes it creates. To get a more fair comparison, an alternative total route length, z_{PRP}, was calculated based on the optimal route for each of the b selected batches, using the PRP algorithm by Pansart, Cambazard and Catusse in [4].

The items in the b batches were then subjected to item batching using the column generation, creating b new batches that were optimal according to our item batching MILP, (3.1). The total route length for these batches is denoted by $z^\star$. The column generation was run until the optimality gap had been closed, or for a maximum of three hours for each problem instance. Every test case was run for five different problem instances with its parameters, and all values were averaged over these five runs.

Finally, the batching was also run for an alternative, simple algorithm which hasn't been discussed previously. When running the column generation, the optimal values for the initial restricted column sets turned out to be very close to the optimal values for the unrestricted column sets. To determine whether these results could be recreated without any linear programming at all, a simple algorithm was developed that mimicked the batches in the initial restricted column set, as described in Subsection 3.1.2, by using the two item orderings used by the algorithm for creating the initial restricted column set. For each of these orderings, the initial restricted column set algorithm created batches from items 1 through t, 2 through $t + 1$ and so on. For our simple batching algorithm, we instead create batches from items 1 through t, $t + 1$ through $2t$, and so on. We then evaluate the total route lengths for these two batch sets separately, and pick the best value. This total route length is denoted z_{simple}. Examples of batches generated by the simple batching algorithm are shown in Figure 4.1.

In our tests, we will compare z_{PRP}, z_{simple}, and $z^\star$ with z_{current} to see how much shorter our total route lengths become. Furthermore, we will look at the time it

takes for the column generation implementation to achieve $z^\star$, denoted t_{opt}, and the time it takes to close the optimality gap, denoted t_{ver}, for "verification time"; only when the optimality gap is closed have we verified that our best known solution is truly optimal. The time necessary to find the solutions corresponding to z_{PRP}, z_{simple}, and z_{current} are negligible and won't be considered.

The time measurements, t_{opt} and t_{ver}, will be given in seconds per batch, to allow for easier comparison between test cases with different values for b.

4.4 Results and discussion

The results of our tests as described in Section 4.3 are shown in Figures 4.2 and 4.3.

Overall, we see a noticeable improvement when applying the PRP route optimization to the current batches, reducing the total route length by some 15–30% for all test cases with $t = 5$, and by some 50–65% for the test cases with $t = 20$. This difference for different values of t is to be expected, since route optimization effects are less noticeable for smaller batches.

The column generation achieves further improvements; we typically see $z^\star \approx z_{\text{PRP}}/2$ for all but the test case with the fewest items, $t \cdot b = 5 \cdot 2 = 10$. Furthermore, the optimal value is achieved in a matter of seconds per batch for even the largest test cases, with the highest t_{opt} observed being 7.4 seconds per batch, for $t = 20$, $b = 6$ and $h = 2$. Several test cases actually obtain the optimal value with just the initial restricted column set, reporting $t_{\text{opt}} = 0$, with such occurrences being more common for the test cases with fewer items.

The time to close the optimality gap, t_{ver}, is, however, quite long for some test cases, particularly the ones with more items or more horizontal aisles. In a total of six instances, the optimality gap wasn't closed within the full three hours each instance was allotted, which likely would be much longer than one would run the column generation for in a real-world application. In that sense, the column generation implementation leaves a lot to be desired for these more difficult cases, but for practical use it doesn't matter much that the verification time is long; reasonably, one could terminate the column generation when the optimality gap is sufficiently small, when the upper bound hasn't improved for a certain number of iterations, or when a certain time limit has been reached, and still get a good solution without verifying that the solution is truly optimal.

Regardless, the column generation frankly seems unnecessary for the item batching problem; in all test cases, the simple batching algorithm generated batches with almost optimal solutions, giving practically the same improvement when compared to the current batches as the batches from the column generation implementation, differing only by a few percentage points in improvement. As such, for most real-world applications, the simple batching algorithm would be a better choice, since it requires much less computational power and time, is much easier to implement and maintain, and more generally applicable—for example, the column generation implementation assumes that the warehouse has a grid layout, as defined in Subsection 3.3.1.2, which is the case for many warehouses, but not all—while still providing almost optimal solutions.

Furthermore, the simple batching algorithm runs in $\mathcal{O}(N \log(N))$ time, where N is the total number of items to batch (essentially, all it needs to do is sort all N items), and as such it scales very well, unlike the column generation implementation. This doesn't mean that the simple batching algorithm is ever better than the column generation implementation, however; the batches the simple batching algorithm uses are available in the initial restricted column set, and so the initial optimal solution in the column generation is always at least as good as the simple batching algorithm solution.

All in all, the column generation implementation is likely only potentially of use in situations where the quality of the batches is crucial, and getting those additional percentage points of improvement compared to the simple batching algorithm is particularly meaningful, but in most typical use cases, the simple batching algorithm is likely sufficient.

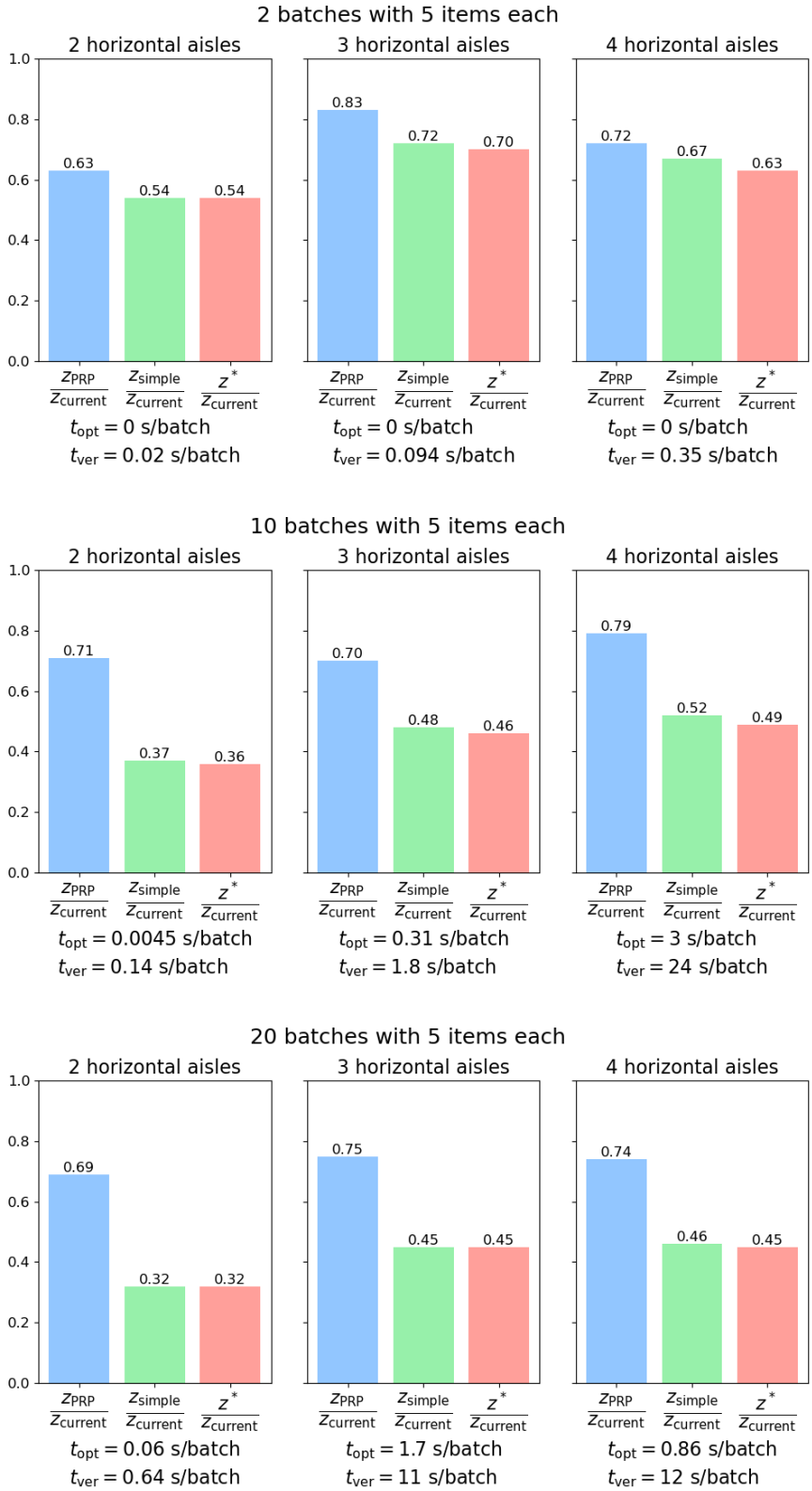


Figure 4.2: Results from the test cases with $t = 5$, $b \in \{2, 10, 20\}$ and $h \in \{2, 3, 4\}$. The results for each test case were averaged over five runs. z_{PRP}, z_{simple}, $z^\star$, z_{current}, t_{opt}, and t_{ver} are defined as in Section 4.3. $t_{\text{opt}} = 0$ means that the optimal solution was obtained from the initial restricted column set.

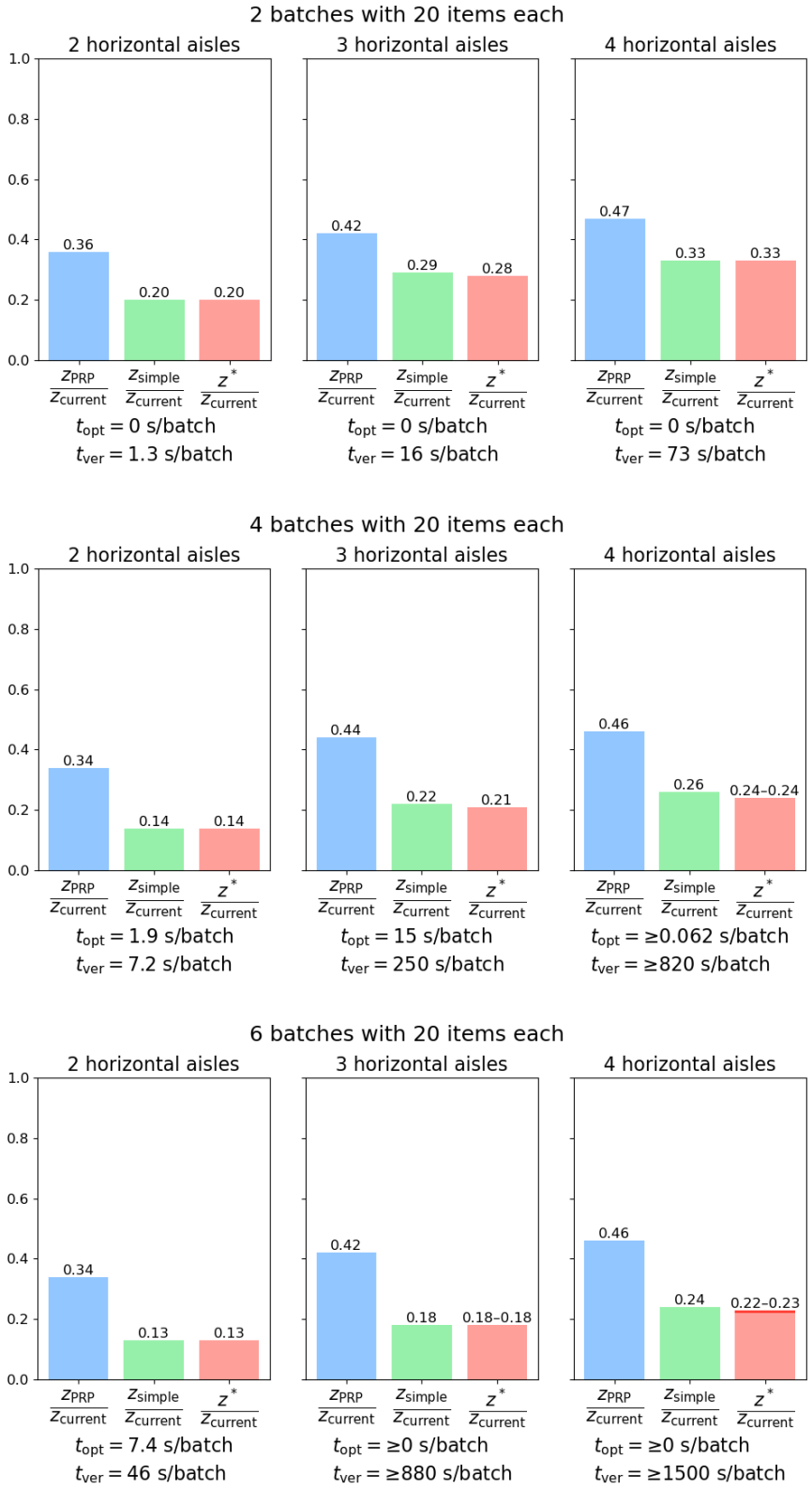


Figure 4.3: Results from the test cases with $t = 5$, $b \in \{2, 4, 6\}$ and $h \in \{2, 3, 4\}$. The results for each test case were averaged over five runs. z_{PRP}, z_{simple}, $z^\star$, z_{current}, t_{opt}, and t_{ver} are defined as in Section 4.3. $t_{\text{opt}} = 0$ means that the optimal solution was obtained from the initial restricted column set. Some runs did not close the optimality gap within the allotted time limit of three hours, and so the true values of $z^\star$, t_{opt}, and t_{ver} aren't known exactly. For test cases with such runs, these values are presented based on their known bounds.

5 Conclusions and future work

5.1 Conclusions

All in all, it is clear from the results that column generation does offer great potential speed-ups when applied to the item batching problem, but that similar results can be achieved much more easily through heuristic methods, like the simple batching algorithm described in Section 4.3. As such, column generation appears to be more difficult to implement than it's worth for anything but the most extreme cases, where optimality is of utmost importance.

5.2 Future work

While implementing column generation-based item batching is likely more trouble than it's worth in most cases, the work presented in this report has a lot of room for interesting future work. In particular, the reduced cost-optimal subproblem algorithm is especially interesting; there are many ways in which it could be extended and improved, none of which have been researched, since it was developed specifically for this project. With some extension, the reduced cost-optimal subproblem algorithm could potentially be used to successfully apply column generation to more constrained versions of the item batching problem for which heuristic methods—like the simple algorithm discussed in Section 4.3—don't work as well.

Examples of item batching problems for which extensions of the reduced cost-optimal subproblem algorithm may be valuable include problems with weight and volume constraints on the batches, as opposed to the cardinality constraints used here (Subsection 5.2.1); item batching in warehouses without a grid layout (which the algorithm currently relies on) (Subsection 5.2.2); problems with path restrictions, for example limiting certain subaisles to be one-way only (Subsection 5.2.3); and, most interestingly, the order batching case, in which each item needs to be in the same batch as all other items in its order (Subsection 5.2.4).

There also appears to be room for improvement in the branching candidate selection, described in Section 3.4. These potential improvements are discussed in Subsection 5.2.5.

5.2.1 Weight constraints

This project has focused on a warehouse used by an online book retailer. Since they only keep books in their warehouses, and books tend to be reasonably similar in weight, limiting batches by number of items picked is quite reasonable. In other warehouses where the items vary more significantly in weight, it may be more useful to limit the batches by weight instead.

Such constraints could be implemented in the reduced cost-optimal subproblem algorithm by removing the number of items picked in the partial picking tour equivalence definition, and discarding any partial picking tour, P_1, where there is another partial picking tour, $P_2 \equiv P_1$, with $\bar{c}(P_2) < \bar{c}(P_1)$ *and* $W(P_2) < W(P_1)$, where $W(P)$ is the total weight of the items picked by P.

Furthermore, any picking tour, T that picks items with $W(T) > 0$ would be valid if all current requirements are met, apart from the cardinality requirement.

Unfortunately, the heuristic function for the reduced cost-optimal subproblem algorithm would likely need to be adapted significantly for this to work, but the basic idea behind the algorithm is still sound.

5.2.2 General warehouse layouts

The reduced cost-optimal subproblem algorithm in its current form only works for warehouses with grid layouts. This was mentioned explicitly in Subsection 3.3.1.2, but in truth, that assumption is also necessary for the heuristic in its current form. Unfortunately, not all warehouses have grid layouts in reality; sometimes there are through-aisles that don't span the length of the warehouse and sometimes some facet of the building interferes with the grid structure (say, having a support column where an aisle would run or having a non-rectangular building). One could implement circumvent the problem by modelling the warehouse as a warehouse with grid layout, ignoring through-aisles that don't span the length of the warehouse, adding non-existent subaisles to fill out missing parts of the grid, et cetera. This would allow the algorithm as a whole to run, but the results may be bad in practice. To better handle such non-gridlike warehouses, the vertical double transition reduction would likely have to be scrapped, and the heuristic function would need to be modified.

5.2.3 Path restrictions

In some warehouses, the trolleys used are wide in relation to the aisles, since wider trolleys mean more products can be moved at once, and narrower subaisles means more shelves can be fit into the floor plan of the warehouse. If the trolleys are too wide in relation to the aisles, however, it may be impossible for two trolleys to pass one another. To avoid collisions, some warehouses enforce one-way policies in some or all of their aisles, so that trolleys don't need to pass each other in the opposite direction.

Such restrictions cannot be enforced in the subproblem algorithm as it is implemented currently, but it would not be difficult to adapt the algorithm to allow them. The most basic adaptation for a warehouse where all aisles have a one-way policy would be to reduce the set of available transitions for any state by removing all in-and-out transitions (they have the worker turning around in the middle of the subaisle, and so clearly violated the one-way policy), considering all edges to be directed and replacing the parities in the state definitions with the difference between the in-degree and the out-degree (i.e., how many more ingoing edges than outgoing does a node have), and requiring this difference to be zero in a full picking tour, rather than simply having an even parity.

Beyond that, it would be useful to modify the heuristic to make it more accurate,

given these new restrictions, but it wouldn't be absolutely necessary to do so; the heuristic, as it is formulated right now, heuristically solves a relaxed version of this one-way policy problem (since it just ignores it), and as such still constitutes a consistent heuristic.

5.2.4 Order restrictions

The model and subproblem algorithms developed in this project were designed without explicit consideration for which order an item belongs to; items belonging to the same order can be split into different batches without problem. In most warehouses, this is not the case; all items in an order should be part of the same batch. The main reason why is that this allows the workers to sort the items into their respective orders (for example by having one slot in the trolley for each order they are picking as part of their picking tour) *while picking*.

While order constraints weren't an intended feature of the subproblem algorithms, they do actually support them; an order constraint can be modelled as a set of force-same constraints. For example, if items 1, 2, and 3 belong to the same order, the order constraint could be modelled as a force-same constraint between 1 and 2, and one between 2 and 3. In other words, the column generation implementation developed during the course of this project can be applied to order batching with minimal modifications. Investigating its performance for that case would be interesting. Furthermore, there may very well be additional adjustments that can be made to the subproblem algorithms and/or the model as a whole to better suit the order batching problem.

Applying the column generation implementation to the order batching problem would be particularly interesting because that problem is much more difficult to solve well heuristically; in this project, we saw that column generation can be applied to the item batching problem with great improvements over non-geographical batching, but that similar improvements can be achieved very easily with heuristic batches. The heuristic batching used in this project would likely not work well at all for the order batching problem, and intuitively, it seems that there is no heuristic batching that works nearly as well in general without significant computational costs.

5.2.5 Improved branching candidate selection

The branching candidate selection algorithm, as described in Section 3.4, has not been evaluated fully, and is likely quite suboptimal. The algorithm was primarily designed to make the constraints as easy as possible to handle for the reduced cost-optimal subproblem algorithm, with reasonable success, but possibly at the cost of making the column generation converge more slowly.

To improve the runtime of the column generation as a whole, it seems likely that candidates with items i, j for which $\sum_{k \in \mathcal{K}'(i,j)} \lambda_k$ is close to 0.5 should be given higher priority, as the effect this has on the convergence likely outweighs the benefits for the reduced cost-optimal subproblem algorithm the other prioritization rules give. This change could either be implemented by giving this rule a higher priority, or by prioritizing candidates by some weighted sum of all rules, so that this rule has an

effect beyond just being a tiebreaker.

Furthermore, the rule that prioritizes candidates that have their items in subaisles with as few previous constrained items as possible likely has an adverse effect; after many branchings, the constrained items tend to be spread out across basically all subaisles (which of course is driven by this rule), which increases the likelihood that a constraint contains items in two subaisles which are very far apart in terms of layer index. This is quite bad, since a constraint only affects the algorithm for layers between the layer indices of the constraint's constrained items; for any layer before that, all states will have an *unreached* status for that constraint, and for every layer after, every valid state will have a *cleared* status for that constraint. Only for layers between will the states be differentiated based on this constraint status. As such, by minimizing the distance (in terms of layers) between the subaisles of the constrained items in every constraint, one would minimize how much one needs to take it into account, but the current candidate selection algorithm essentially does the opposite.

As such, the rule in question should probably either be removed, adapted to take this "constraint layer distance" into account, or replaced with a new rule that only takes the constraint layer distance into account.

Bibliography

1. Gu, J., Goetschalckx, M. & McGinnis, L. F. Research on Warehouse Operation: A Comprehensive Review. *European Journal of Operational Research* **177(1),** 1–21 (16th Feb. 2007).
2. Marchet, G., Melacini, M. & Perotti, S. Investigating Order Picking System Adoption: A Case-Study-Based Approach. *International Journal of Logistics* **18** (2nd Jan. 2015).
3. Desrosiers, J. & Lübbecke, M. *A Primer in Column Generation. Ch 1* in Desaulniers, G., Desrosiers, J. & Solomon, M. M. *Column Generation* 1–32 (2005).
4. Pansart, L., Catusse, N. & Cambazard, H. Exact Algorithms for the Order Picking Problem. *Computers & Operations Research* **100,** 117–127 (1st Dec. 2018).
5. Cambazard, H. & Catusse, N. Fixed-Parameter Algorithms for Rectilinear Steiner Tree and Rectilinear Traveling Salesman Problem in the Plane. *European Journal of Operational Research* **270(2),** 419–429 (16th Oct. 2018).
6. Ratliff, H. & Rosenthal, A. Order-Picking in a Rectangular Warehouse: A Solvable Case of the Traveling Salesman Problem. *Operations Research* **31,** 507–521 (1st June 1983).
7. Bellman, R. Dynamic Programming Treatment of the Travelling Salesman Problem. *Journal of the ACM* **9(1),** 61–63 (Jan. 1962).
8. Dantzig, G. B. & Ramser, J. H. The Truck Dispatching Problem. *Management Science* **6(1),** 80–91. JSTOR: 2627477 (1959).
9. Cordeau, J.-F., Laporte, G., Savelsbergh, M. W. P. & Vigo, D. *Chapter 6 Vehicle Routing* in *Handbooks in Operations Research and Management Science* (eds Barnhart, C. & Laporte, G.) 367–428 (Elsevier, 1st Jan. 2007).
10. Briant, O., Cambazard, H., Cattaruzza, D., Catusse, N., Ladier, A.-L. & Ogier, M. An Efficient and General Approach for the Joint Order Batching and Picker Routing Problem. *European Journal of Operational Research* **285(2),** 497–512 (1st Sept. 2020).
11. Gademann, N. & van de Velde, S. Order Batching to Minimize Total Travel Time in a Parallel-Aisle Warehouse. *IIE Transactions* **37(1),** 63–75 (1st Jan. 2005).
12. Dantzig, G. B. & Wolfe, P. Decomposition Principle for Linear Programs. *Operations Research* **8(1),** 101–111. JSTOR: 167547 (1960).
13. Balas, E. The Prize Collecting Traveling Salesman Problem. *Networks* **19(6),** 621–636 (1989).

14. Barnhart, C., Johnson, E. L., Nemhauser, G. L., Savelsbergh, M. W. P. & Vance, P. H. Branch-and-Price: Column Generation for Solving Huge Integer Programs. *Operations Research* **46(3),** 316–329. JSTOR: 222825 (1998).
15. Ryan, D. & Foster, B. An Integer Programming Approach to Scheduling. *Computer Scheduling of Public Transport* **1,** 269 (1st Jan. 1981).
16. Lewis, H. & Zax, R. *Essential Discrete Mathematics for Computer Science* (Princeton University Press, 19th Mar. 2019).
17. Dijkstra, E. W. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik* **1,** 269–271 (1st Dec. 1959).
18. Hart, P., Nilsson, N. & Raphael, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* **4,** 100 (1st July 1968).
19. Kleinberg, J. & Tardos, E. *Algorithm Design* 1st edition (Pearson, Boston, 16th Mar. 2005).
20. Bestuzheva, K., Besançon, M., Chen, W.-K., Chmiela, A., Donkiewicz, T., van Doornmalen, J., Eifler, L., Gaul, O., Gamrath, G., Gleixner, A., Gottwald, L., Graczyk, C., Halbig, K., Hoen, A., Hojny, C., van der Hulst, R., Koch, T., Lübbecke, M., Maher, S. J., Matter, F., Mühmer, E., Müller, B., Pfetsch, M. E., Rehfeldt, D., Schlein, S., Schlösser, F., Serrano, F., Shinano, Y., Sofranac, B., Turner, M., Vigerske, S., Wegscheider, F., Wellner, P., Weninger, D. & Witzig, J. *The SCIP Optimization Suite 8.0* ZIB-Report 21-41 (Zuse Institute Berlin, Dec. 2021).
21. Maher, S., Miltenberger, M., Pedroso, J. P., Rehfeldt, D., Schwarz, R. & Serrano, F. *PySCIPOpt: Mathematical Programming in Python with the SCIP Optimization Suite* in *Mathematical Software – ICMS 2016* (eds Greuel, G.-M., Koch, T., Paule, P. & Sommese, A.) (Springer International Publishing, 2016), 301–307.

DEPARTMENT OF MATHEMATICAL SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY