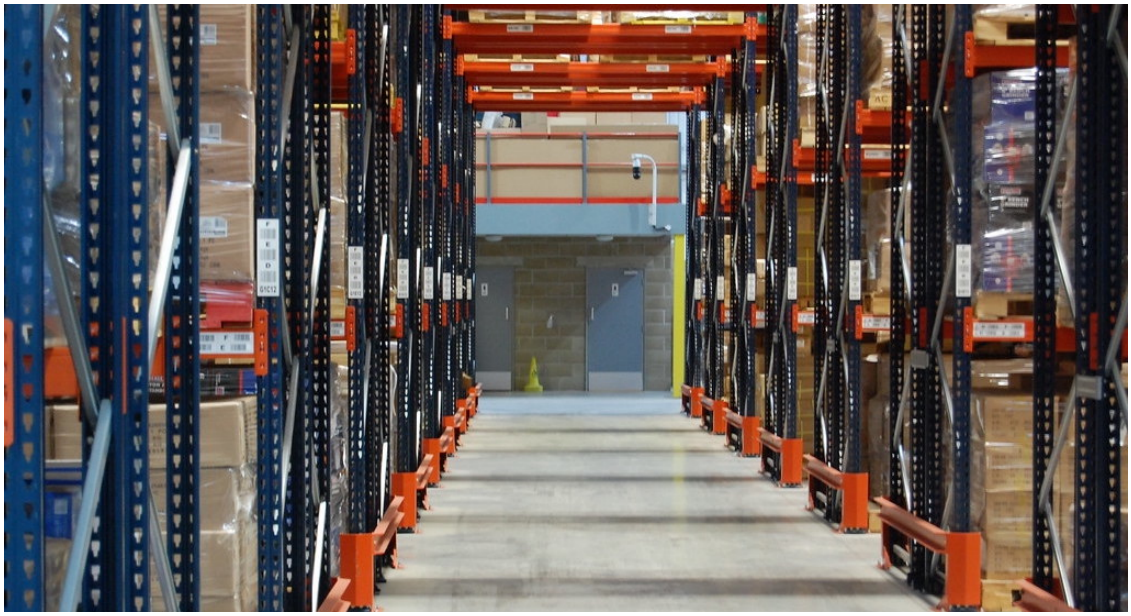

Batch Picking in Warehouse Logistics: Trading Optimality for Feasibility

Master's thesis in Computer science and engineering

LUDWIG HULTQVIST
MATHIAS LAMMERS

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Master's thesis 2022

Batch Picking in Warehouse Logistics: Trading Optimality for Feasibility

LUDWIG HULTQVIST
MATHIAS LAMMERS

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Batch Picking in Warehouse Logistics: Trading Optimality for Feasibility

LUDWIG HULTQVIST
MATHIAS LAMMERS

© LUDWIG HULTQVIST, MATHIAS LAMMERS, 2022.

Supervisor: Jonah Brown-Cohen, Department of Computer Science and Engineering
Advisor: Martin Sigvardsson, Ongoing Warehouse
Examiner: Robin Adams, Department of Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Example of a generic warehouse

Typeset in LaTeX
Gothenburg, Sweden 2022

Batch Picking in Warehouse Logistics: Trading Optimality for Feasibility

LUDWIG HULTQVIST
MATHIAS LAMMERS
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

A significant cost in warehouse logistics is the act of traversing the warehouse to collect orders, where a commonly applied heuristic to reduce this cost is picking several orders at once. Partitioning the full order set into optimal batches is known as the *batch picking problem*, which also involves solving the *travelling salesman problem*. The travelling salesman problem is famously known as $\mathcal{NP}$-hard, making it most likely infeasible to optimally solve larger instances of the batch picking problem in practice. This thesis, in collaboration with *Ongoing Warehouse*, aims to study different algorithms to find close-to-optimal solutions while still being viable in real-world applications.

For the evaluation to be as realistic as possible, the layout of a physical warehouse was used as a model. This, in addition to multiple order sets and the item placements used in the aformentioned warehouse, was supplied by Ongoing Warehouse. Various algorithms for partitioning the order sets and providing necessary traversal paths were implemented in C# and benchmarked with the use of the external library *BenchmarkDotNet*, where the distance of the paths, memory usage and required time were chosen as the evaluation metrics. The benchmarking results indicated that partitioning the order set based on proximity and solving the traversal path with conventional linear programming can be used for reducing the total traversal distance, while still remaining feasible in practice.

Keywords: optimization, algorithms, batch picking, travelling salesman, warehouse, logistics, graphs, benchmarking, complexity

Acknowledgements

We would like to thank Jonah Brown-Cohen and Robin Adams at the Department of Computer Science and Engineering for acting as our supervisor and examiner respectively, providing us with valuable feedback and guidance. We would also like to thank our advisor Martin Sigvardsson and everyone else at Ongoing Warehouse for supporting us while writing this thesis.

Ludwig Hultqvist & Mathias Lammers, Gothenburg, June 2022

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

1

Introduction

Efficient warehouses are undeniably essential to operate in the modern logistics industry. In the last decade, the increasing demand for business-to-consumer shipments has effectively pressured warehouses into establishing new methods for optimizing their operations [1], [2]. A natural improvement is to streamline how warehouse staff manually pick customer orders throughout a physical warehouse, since picking orders can significantly impact operational warehouse costs, as a study in the UK presented in 1988 [3].

A common approach to reduce picking costs is to shorten the walking distances of staff by picking multiple orders at once in batches. Minimizing the picking distance by dividing a set of orders into optimal batches is referred to as the *batch picking problem*. The problem is believed to be computationally $\mathcal{NP}$-hard, and there are subsequently no currently known feasible algorithms for finding optimal solutions [4]. Nevertheless, one may wonder — is there an algorithm for nearly optimizing batch picking that is feasible for logistics applications? To what extent may optimality be traded for feasibility?

The research project is carried out in collaboration with *Ongoing Warehouse*, a large provider of warehouse management software systems for the third-party logistics industry [5]. Modern warehouses utilize management systems for their daily operations, including keeping track of stock and assigning picking tasks to staff. With access to Ongoing Warehouse's systems, optimization algorithms may be evaluated against real-world data. Their insights into the logistics industry are therefore useful for finding feasible, yet close to optimal algorithms.

1.1 Problem Description

Two optimization problems are addressed in the research project – the batch picking problem and the *travelling salesman problem*. In the batch picking problem, a warehouse receives a set of incoming orders, each being several items stored in the warehouse. The orders shall be partitioned into a set of batches that are picked independently. The task is to choose a partition that minimizes the total walking distance for picking all batches. Since batches are picked independently, the task ultimately concerns minimizing the picking distances of single batches or instances of the travelling salesman problem. The two optimization problems are formalized in higher detail in Section 2.2 and Section 2.3.

1.2 Research Questions

There are currently no known algorithms for solving the two optimization problems with better-than-exponential complexity and such algorithms may not exist. On the other hand, there may exist efficient algorithms producing solutions that outperform more naive approaches in existing warehouse applications. The aim of the research project is thus to investigate batch picking and travelling salesman techniques that may be feasible in practice, by showing empirically low complexity while computing close-to-optimal solutions. The aim is further formalized into two research questions:

1. How may the total travelling distance for picking a set of orders in a warehouse be reduced by dividing the set into a set of optimal batches?
2. Is there a feasible algorithm for solving the batch picking optimization problem in practice?

1.3 Delimitations

All algorithms are evaluated against real warehouse data encapsulated into models. To enable a viable project execution, the scope of modelling realistic warehouses is severely restricted. Limited availability of existing warehouse layouts requires warehouse models to be manually crafted. Since this is a time-consuming task which may limit the extent of researching algorithms, only one real-world warehouse is modelled. Algorithm performance may as such be biased towards the selected warehouse, causing results applicable in theory with implausible generalisation prospects to real-world applications.

1.4 Contributions

Implementations of the viable algorithms solving the batch picking and travelling salesman problems are encapsulated into a compact .NET library written in C# using .NET Core. The library consists of a set of projects for computing warehouse models and running optimization algorithms in isolation. Each project also contains the required datasets for various problem instances. The library includes a benchmarking suite for comparing algorithm performance on equal problem instances.

2

Theory

The following sections present how the chosen warehouse is modelled as a complete graph and how the two problems are formalized as optimization problems. Characteristics of all algorithms utilized throughout the project are further described in depth using the problem formalization.

2.1 Warehouse Model

From a bird's-eye view, the layout of the selected warehouse consists of a two-dimensional grid C, as displayed in Figure 2.1. Items stored in the warehouse are defined as a mapping $I \to C$ from a set of identifiers I to warehouse coordinates C. The identifiers are expressed in the format of *A-01-02*, referring to aisle A, column 1, and shelf 2, which are mapped to coordinates using the mappings displayed in Listing A.2 in Appendix A. Further, let $O \subseteq \{O' | O' \subseteq I\}$ denote a set of orders received by the warehouse, each being a subset of item identifiers. An example of a order dataset is displayed in Table A.1. For every coordinate $(x, y) \in C$, let c_{xy} denote a binary variable which is equal to 1 if (x, y) is unoccupied by a shelf. Pickers can only traverse coordinates (x, y) where $c_{xy} = 1$, may only transition vertically, horizontally, and diagonally between adjacent coordinates (x, y) and (x', y') if $c_{xy} = c_{x'y'} = 1$. The cost of transitioning between two coordinates is defined as their euclidean distance, which is either 1 or $\sqrt{2}$ for adjacent coordinates.

The definition of the grid C creates a foundation to model the warehouse as a complete weighted graph $G = (V, E)$. V is a set of nodes computed using a one-to-one mapping $C \to \mathbb{N}$ from the coordinates in C to unique identifiers. The set of weighted edges E further consists of one edge e for every pair of coordinates (x, y) and (x', y'), with a weight w_e of the minimal traversal distance between the two. E may as such be computed in $\mathcal{O}(|V|^3)$ time using Dijkstra's algorithm, starting from every coordinate in C [6]. The complete graph may finally be stored in $\mathcal{O}(|V|^2)$. To further simplify the notation of orders, they are mapped directly from item identifiers to nodes in G and may as such be expressed as subsets of nodes $O' \subseteq V$. The set of orders is then re-defined in G as $O \subseteq \{O' | O' \subseteq V\}$.

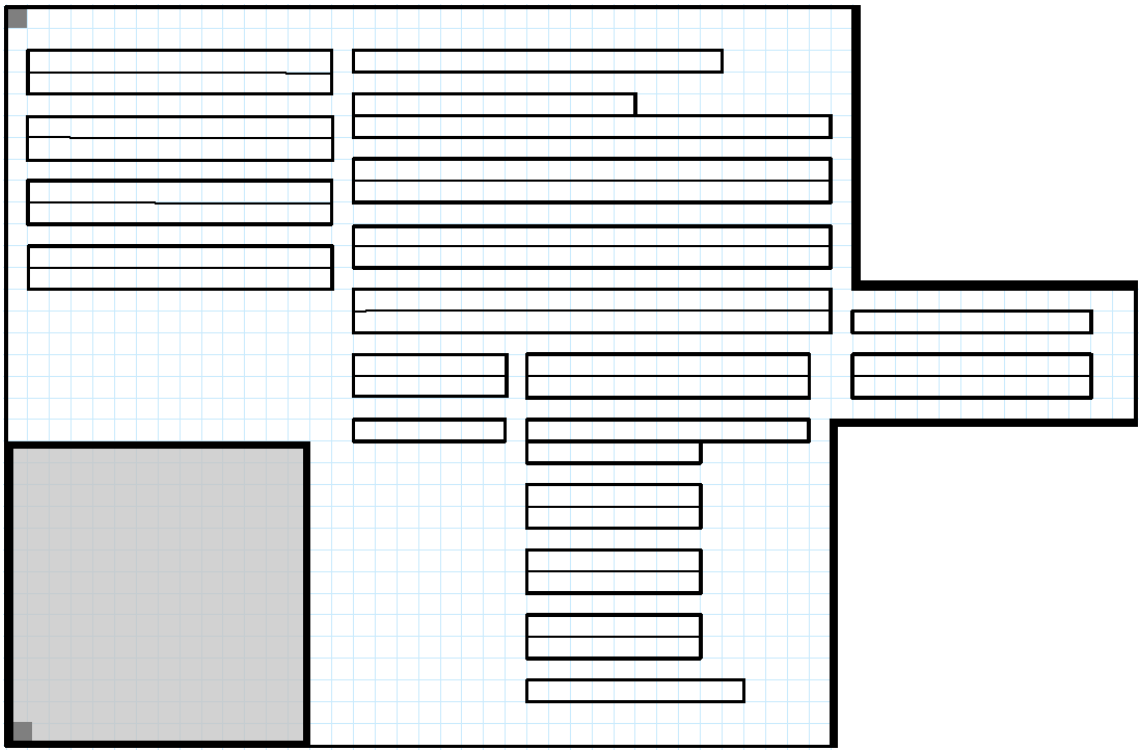

Figure 2.1: Two-dimensional grid of the warehouse layout defined in Listing A.1.

2.2 Travelling Salesman Problem

The travelling salesman problem is a recurring optimization problem first described in the mid-20th century [7]. In the original definition, a salesman is tasked with visiting a set of cities with known travelling distances between each pair of cities. Starting from any city, the travelling salesman shall select a path of minimum distance that visits all cities exactly once and returns to the starting city. To formalize the problem as an optimization problem, it is best defined as a graph problem.

Consider a weighted complete graph $G = (V, E)$. For all pair of nodes $u, v \in V$, let $w_{uv} = w_{vu}$ be the travelling distance between u and v. Further, let $P \subseteq E$ denote any path P visiting all nodes in V exactly once. For all nodes $v \in V$, there must exist exactly one edge $e = (u, v) \in P$ entering v and exactly one edge $e' = (v, u') \in P$ exiting v. P is hence constrained by $|\{(v, u) \in P\}| = |\{(u', v) \in P\}| = 1$. The travelling distance of P is finally defined as $\sum_{(u,v)\in P} w_{uv}$. The travelling salesman problem is hence defined as the optimization problem TSP(G):

$$\begin{aligned} \text{minimize} \quad & \sum_{(u,v)\in P} w_{uv} && P \subseteq E \\ \text{subject to} \quad & |\{(u, v) \in P\}| = 1 && \forall v \in V \\ & |\{(v, u') \in P\}| = 1 && \forall v \in V \end{aligned}$$

While the problem is proven to be computationally $\mathcal{NP}$-hard, there still exist many methods for solving or approximating the problem [8]. A selection of common techniques used in this project is briefly described in the upcoming sections.

2.2.1 Nearest Neighbor

The nearest neighbor algorithm is historically one of the first methods for addressing the travelling salesman problem [9]. It utilizes a simple greedy heuristic to incrementally compute an approximately minimal path. Starting from any node, the path is iteratively extended with the edge from the last visited node to the nearest unvisited node, until all nodes are visited. Since the nearest neighboring node is found in $\mathcal{O}(|V|)$ time, the traversal path is computed in $\mathcal{O}(|V|^2)$. Only $\mathcal{O}(|V|)$ additional space is subsequently required for storing the computed path.

The polynomial complexity of the nearest neighbor algorithm makes it compelling to utilize in practical applications. However, there is no known approximation ratio to optimal solutions. For some instances, it can be proven that the algorithm has a worst-case approximation ratio of $\mathcal{O}(\log|V|)$, while for other instances, the algorithm may compute the worst path [10]–[13].

2.2.2 Held-Karp Algorithm

The Held-Karp algorithm is a well-established dynamic programming algorithm for optimally solving the traveling salesman problem [14], [15]. It utilizes the simple assumption that if a path S between a starting node $s \in V$ and any other node $u \neq s \in V$ is optimal, then any sub-path of S is optimal. The optimal distance of any path traversing all nodes in V may then be computed recursively. Let $s \in V$ be any starting node and consider any subset of remaining nodes $S \subseteq V \setminus \{s\}$. For any node $v \in S$, the shortest distance of any $s - v$ path traversing all nodes in S is defined recursively as:

$$\begin{aligned} &\mathrm{OPT}(S, v) = \min(\mathrm{OPT}(S \setminus \{u\}, u)) + w_{uv} && u \in S \setminus \{v\} \\ &\mathrm{OPT}(\{u\}, u) = w_{s,u} \end{aligned}$$

The optimal traversal distance of V is then defined as $\min(\{\mathrm{OPT}(V \setminus s, u) + w_{us})$ for $y \in V \setminus s\}$. As outlined in Algorithm 2.1, the optimal solution is computed by iteratively filling a table of optimal traversal distances for all subsets of sizes $[1, |V|)$. Using any starting node $s \in V$, the table is initialized with $\mathrm{OPT}(\{v\}, v) = w_{sv}$, for subsets $\{v\}$ of length one for all nodes $v \in V \setminus s$. For all subsets $S \subseteq V \setminus s$, $|S| = j$, of all lengths $j \in [2, |V|)$, the table is subsequently expanded using the objective function. The optimal traversal distance of V is finally returned.

Algorithm 2.1 The Held-Karp algorithm solving the travelling salesman problem.

1: **for** $v \in V \setminus \{s\}$ **do**
2: $\quad \mathrm{OPT}(\{v\}, v) = w_{sv}$
3: **for** $j = 2...n-1$ **do**
4: $\quad$ **for** $S \subseteq V \setminus \{s\}, |S| = j$ **do**
5: $\quad\quad$ **for** $v \in S$ **do**
6: $\quad\quad\quad \mathrm{OPT}(S, v) = \min(\mathrm{OPT}(S \setminus \{v\}, u) + w_{uv}),\ u \in S \setminus \{v\})$
7: **return** $\min(\mathrm{OPT}(V \setminus \{s\}, v) + w_{vs}),\ v \in V \setminus \{s\})$

By reusing already computed optimal sub-paths, the Held-Harp algorithm solves the travelling salesman problem exactly in $\mathcal{O}(|V|^2 \cdot 2^{|V|})$ time, a significant but still infeasible improvement over the $\mathcal{O}(|V|!)$ brute force complexity [15]. However, the improvement requires storing all optimal sub-paths using $\mathcal{O}(|V| \cdot 2^{|V|})$ space, making it impractical for large problem instances.

2.2.3 Christofides Algorithm

Christofides algorithm is a prominent 3/2-approximation algorithm for the travelling salesman problem with the currently lowest known approximation ratio [16], [17]. It computes an approximately optimal path in a few steps. A minimum spanning tree T is first computed from G using algorithms such as Prim's or Kruskal's algorithm [18], [19]. An Eulerian multigraph M is then obtained by adding edges between the nodes with an odd degree in T, such that all nodes have even degrees. By the handshaking lemma, there must be an even number of nodes with odd degrees. An Eulerian circuit C traversing all edges in M is subsequently computed using for instance Fleury's algorithm. The minimum path traversing all nodes is finally obtained from C, by selecting the nodes in the order they were traversed first in C.

The computational complexity of Christofides algorithm is upper bounded by the choice of minimum spanning tree algorithm. It terminates in at most $\mathcal{O}(|V|^3)$ time, requiring $\mathcal{O}(|V|^2)$ space in a complete graph.

2.2.4 Linear Programming

Already in the original definition of the travelling salesman problem, the problem was formulated as an integer linear program [7]. A commonly used formulation today is the Dantzig-Fulkerson-Johnson formulation, defined as follows [20]. For every pair of nodes $u \neq v \in V$, let the binary variable x_{uv} denote if edge (u, v) is selected in the minimal path. The distance of any path is then computed as the sum $\sum_{u \in V} \sum_{v \neq u \in V} w_{uv} x_{uv}$. Since every node u is visited exactly once, there is exactly one selected edge entering and one selected edge exiting u. Hence, $\sum_{u \in V} x_{uv} = \sum_{u \in V} x_{vu} = 1$ for $v \neq u \in V$. To avoid forming sub-paths instead of one coherent path traversing all nodes, a final constraint is also added. The integer linear program is then expressed as:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{u \in V} \sum_{v \neq u \in V} w_{uv} x_{uv} & \\
\text{subject to} \quad & \sum_{u \in V} x_{uv} = 1 & v \neq u \in V \\
& \sum_{u \in V} x_{vu} = 1 & v \neq u \in V \\
& \sum_{u \in Q} \sum_{v \neq u \in Q} x_{uv} \leq |Q| - 1 & \forall Q \subsetneq V, |Q| \geq 2
\end{aligned}
$$

Note that the running times of algorithms solving linear programs are generally unknown. It is however assumed that $\mathcal{O}(|V|^2)$ space is required for storing variables and constraints.

2.2.5 Ant Colony Optimization

Ant colony optimization is a probabilistic technique for optimization problems that are reducible to graph traversal problems, such as the travelling salesman. It draws inspiration from how real-life ants communicate through stigmergy [21]. That is, leaving a trace of pheromones along their path for other ants to follow, where higher pheromone levels indicate a higher probability of following that path. The ant colony optimization technique is then formalized as follows. For every edge $(u, v) \in E$, define a visibility $\eta_{uv} = 1/w_{uv}$, to favor edges with lower distance, and a *pheromone level* τ_{uv}, which is updated iteratively. A colony of k ants then select a path independently by iteratively sampling a next edge (u, v) according to the probability distribution $p((u, v)|S)$, where $S \subseteq V$. Using τ_{uv} and η_{uv}, and constant parameters α and β promoting either exploration or exploitation, $p((u, v)|S)$ is defined as:

$$
p((u,v)|S) = \begin{cases} \frac{\tau_{uv}^{\alpha} \eta_{uv}^{\beta}}{\sum_{v' \notin S} \tau_{uv'}^{\alpha} \eta_{uv'}^{\beta}} & \text{if } v \notin S \\ 0 & \text{otherwise} \end{cases}
$$

As all k ants have finished sampling paths, τ_{uv} is updated is updated for each edge (u, v) using the traversal distance D_k of path S_k and the evaporation rate ρ:

$$
\Delta\tau_{uv}^{[k]} = \begin{cases} \frac{1}{D_k} & \text{if ant } k \text{ traversed edge } e_{uv} \\ 0 & \text{otherwise} \end{cases}
$$

$$
\tau_{uv} \leftarrow (1 - \rho)\tau_{uv} + \sum_{k=1}^{N} \Delta\tau_{uv}^{[k]}
$$

The complete algorithm is summarized in Algorithm 2.2, where the procedure is repeated for N iterations and the shortest sampled path is returned. Assuming that k and N are static, the algorithm has a running time of $\mathcal{O}(|V|)$, but with a large constant number of iterations. The algorithm further requires $\mathcal{O}(|V|)$ space for storing the sampled paths.

Algorithm 2.2 The ant colony optimization algorithm approximating the travelling salesman problem.

initialize τ_{uv} and η_{uv} for all edges $(u, v) \in E$.
initialize distance $D \leftarrow \infty$
for N iterations **do**
    for k ants **do**
        sample S_k from p
        $D \leftarrow \min(D, D_k)$
    $\tau_{uv} \leftarrow (1 - \rho)\tau_{uv} + \sum_{k=1}^{N} \Delta\tau_{uv}^{[k]}$ for all edges $(u, v) \in E$
return D

2.2.6 Simulated Annealing

Simulated annealing is a probabilistic technique inspired by metallurgical practices of rapidly heating metal before slowly cooling it, resulting in more desirable properties [22]. It is used for approximating a global optimum in primarily large discrete search spaces, such as in the travelling salesman problem, outlined as follows. Let T_{min} denote a constant minimal temperature and let T be the current temperature, initialized to a high value. Initialize a solution S randomly, by selecting a random path traversing all nodes. While $T > T_{min}$, S is randomly modified into S' by choosing a random segment of the path that is either reversed or transported to another location in the solution. S' may then be accepted as a new solution according to the Metropolis-Hastings algorithm [23]. That is, if S' has a shorter or equal traversal distance $D_{S'}$ than the traversal distance D_S of S, S' is always accepted as the new solution. However, if $D_{S'} > D_S$, S' is accepted as the new solution with a computed probability. The probability of selecting S' given S is hence defined as:

$$p(S'|S) = \begin{cases} 1 & \text{if } D_{S'} \leq D_S \\ \exp(\frac{D_S - D_{S'}}{T}) & \text{otherwise} \end{cases}$$

The procedure is repeated N times, before decreasing T by a cooling rate of $c < 1$, subsequently decreasing the acceptance probability. This favors exploration while T is high to avoid local optima, while exploitation is increasingly likely as T decreases. The complete algorithm is outlined in Algorithm 2.3.

Algorithm 2.3 The simulated annealing algorithm approximating the travelling salesman problem.

initialize T to a large temperature
initialize S at random
while $T > T_{min}$ **do**
    for N iterations **do**
        sample S' into S by reversing or transporting a random segment
        update S using the Metropolis-Hastings algorithm
    $T \leftarrow T \cdot c$
return D_S

Since T_{min}, c and N are all constant, the algorithm has a running time of $\mathcal{O}(|V|)$ with a large hidden constant factor, requiring $\mathcal{O}(|V|)$ space for storing S and S'.

2.3 Batch Picking Problem

The batch picking problem is a recurring problem in the logistics industry that may be expressed as a graph optimization problem. Consider the complete warehouse graph $G = (V, E)$ receiving a set of orders $O \subseteq \{O' | O' \subseteq V\}$, as described in Section 2.1. O shall be partitioned into a set of batches $B \subseteq \{B' | B' \subseteq O\}$, where every order is contained in a single batch and all batches are disjoint, i.e. $B' \cap B'' = \emptyset, \forall B', B'' \in B$. Batches are further picked using a trolley with n compartments, that may each store a single order in its entirety. The number of orders in each batch may as such not exceed n.

Picking a batch $B' \in B$ requires traversing the nodes of all orders $O' \in B'$, starting and ending at a chosen origin s. By neglecting quantities of nodes occurring more than once in a batch, picking a batch is considered as an instance of the travelling salesman problem described in Section 2.2. For any batch B', let $V_{B'} = \{s\} \cup \bigcup_{O' \in B'} O'$ denote the set of nodes that must be traversed to pick B'. Let $G[V_{B'}]$ be the resulting induced subgraph in G. The picking distance of B' is then equal to $\mathrm{TSP}(G[V_{B'}])$. Since batches are picked independently, the total picking distance of B is defined as the sum of picking all individual batches $B' \in B$. Batch picking may then be formalized as an optimization problem $\mathrm{BPP}(G, O, n, s)$:

$$\begin{aligned} \text{minimize} \sum_{B' \in B} \mathrm{TSP}(G[V_{B'}]) \qquad & B \subseteq \{B' | B' \subseteq O\} \\ \text{subject to } B' \cap B'' = \emptyset \qquad & \forall B' \neq B'' \in B \\ |B'| \leq n \qquad & \forall B' \in B \end{aligned}$$

The techniques used for addressing the batch picking problem are described in the following sections. Since all orders contain at most $|V|$ nodes, $\mathcal{O}(|O| \cdot |V|)$ space is required for storing the sets of orders and the computed batches in all algorithms.

For easier comparison of complexities, only the additionally required space is hence considered. Algorithms are subsequently only required to copy the keys of orders and not the nodes contained in them. Running times and space requirements are further substantially dependent on the techniques used for evaluating the picking distance of single batches containing at most $|V|$ nodes. To simplify the notations, let $\mathcal{T}$ and $\mathcal{S}$ denote the space and time requirements for computing the picking distance of a single batch. The computational complexities of all batch picking techniques are hence expressed as functions of $\mathcal{T}$ and $\mathcal{S}$.

2.3.1 First In, First Out (FIFO)

A simple approach often used in practice is to partition the set of orders according to their creation timestamp, referred to as *first in, first out* or *FIFO* [24]. Notice that the set is partitioned without performing any optimization, which may result in poor picking distances. However, by partitioning orders by FIFO order, warehouses ensure that all orders are guaranteed to be shipped in time, which may be more important in practice.

Since no optimization is performed to partition the set of orders, they are partitioned into $\mathcal{O}(|O|)$ batches in $\mathcal{O}(|O|)$ time without requiring any additional space. However, the picking distance of individual batches are still optimized in $\mathcal{O}(\mathcal{T})$ time and $\mathcal{O}(\mathcal{S})$ space per batch. This results in a time complexity of $\mathcal{O}(|O| \cdot \mathcal{T})$ and a space complexity of $\mathcal{O}(\mathcal{S})$. The low computational costs of the algorithm are naturally practical in real settings, but may likely result in poor optimality. However, the algorithm provides a strong foundation to evaluate how the performance is affected by only optimizing how orders are picked within batches.

2.3.2 Greedy Batch Heuristic

In a prior project at Ongoing Warehouse, Sigvardsson and Persson suggested a greedy approach for finding a single best batch [25]. Starting from an empty batch, the best batch was populated by iteratively adding the order resulting in the minimal picking distance, until the batch was full. The same procedure can also be repeated with the next batch until all orders are partitioned. The intuition behind this approach stems from warehouses in practice, where staff are only concerned with picking one batch at a time from a continuous order set, which is repeatedly updated as the warehouse receives new orders.

Since the picking distances of single batches are evaluated for all orders $\mathcal{O}(|O|)$ times, the algorithm runs in $\mathcal{O}(|O|^2 \cdot \mathcal{T})$ and requires $\mathcal{O}(\mathcal{S})$ space. The procedure may be beneficial for minimizing the total picking distance but can be biased against poor orders. These may potentially be delayed indefinitely, resulting in a long picking distance for later batches.

2.3.3 Greedy Partition Heuristic

The bias against poor orders introduced by the Greedy Batch heuristic may be removed by simply flipping the greedy rule. Instead of iteratively selecting an order

to include in one best batch, each order can select the best batch to be included in. The partition is computed as follows. Initialize a set of batches with a single order each to avoid a bias against empty batches. For all remaining orders, choose the non-full batch resulting in the shortest picking distance. The total picking distance is finally returned.

Although the flipped rule removes the bias against poor orders, it may introduce a new bias against poor batches. Orders may as such not be added to batches with already high picking distances until all other batches are full. Notice that the greedy rule is still applied the same number of times as the greedy batch heuristics without any additional space. The greedy partition hence also requires $\mathcal{O}(|O|^2 \cdot \mathcal{T})$ time and $\mathcal{O}(\mathcal{S})$ space for computing a solution.

2.3.4 Proximity Heuristic

Another issue with the two greedy heuristics is that both evaluate picking distances $\mathcal{O}(|O|^2)$ times. This may however be addressed by partitioning the set of orders using the average proximity between batches instead of computing any exact distances. For any two batches $B', B'' \subseteq O$, let $A_{B'B''}$ denote the average distance from all nodes in B' to all nodes in B'', defined as:

$$A_{B'B''} = \sum_{u \in V_{B'}} \sum_{v \in V_{B''}} \frac{w_{uv}}{|V_{B'}||V_{B''}|}$$

The proximity heuristic is hence outlined as follows. Initialize a set of batches B, each containing a single unique order. The batch $B' \in B$ with the minimum average distance to itself, i.e. $A_{B'B'}$, is then selected. Subsequently select the other batch $B'' \neq B' \in B$ with the minimum average distance $A_{B'B''}$, such that $|B'| + |B''|$ does not exceed the batch size. If no such B'' exists, B' is marked as full, otherwise, it is merged with B''. The procedure is repeated until all batches in B are full. The exact distance for picking B is finally computed once.

The average distance between any pair of batches is computed in $\mathcal{O}(|V|^2)$. Since all $\mathcal{O}(|O|)$ batches are combined with at most all other $\mathcal{O}(|O|)$ batches, the set of orders is partitioned in $\mathcal{O}(|O|^2 \cdot |V|^2)$ time using $\mathcal{O}(|O|)$ additional space. Evaluating the total picking distance finally requires $\mathcal{O}(|O| \cdot \mathcal{T})$ time and $\mathcal{O}(\mathcal{T})$ space, resulting in a complexity of $\mathcal{O}(|O|^2 \cdot |V|^2 + |O| \cdot \mathcal{T})$ time and $\mathcal{O}(|O| + \mathcal{S})$ space.

2.3.5 Simulated Annealing

Since the batch picking problem is an example of a problem with a large discrete search space, the simulated annealing technique introduced is Section 2.2.6 is well-justified to be utilized, with a few modifications. The set of orders is initially padded with empty orders, such that all batches are filled completely. O may then be modelled as a one-dimensional array of orders. The traversal distance is subsequently redefined as the total distance for picking all batches. The algorithm outlined in Algorithm 2.3 may then be applied to approximate the batch picking problem.

O is partitioned into $\mathcal{O}(|O|)$ batches, whose picking distance are evaluated in $\mathcal{O}(\mathcal{T})$ time and $\mathcal{O}(\mathcal{S})$ space each. Per definition of Algorithm 2.3, simulated annealing also requires $\mathcal{O}(|O|)$ additional space. The algorithm hence terminates in $\mathcal{O}(|O| \cdot \mathcal{T})$ time and $\mathcal{O}(|O| + \mathcal{S})$ space with a large constant factor on the time bound.

2.3.6 Genetic Algorithms

Genetic algorithms consist of a set of techniques based on evolution and natural selection [26]. They are often used for approximating problems when finding optimal solutions tend to be computationally difficult. The technique is applied to the batch picking as follows, which is influenced by a previous research project [27]. Let P be a population of n initially random batch partitions, referred to as individuals. For all individuals $i \in P$, let f_i denote its fitness, i.e. the total picking distance. The idea is to continuously improve the population fitness by iteratively selecting a population P' with more fit individuals, using a few steps.

First, the fitness of every individual is evaluated. The current population is then exploited by adding the most elite individual $i \in P$ with the highest f_i to P'. A random individual, referred to as an immigrant, is subsequently added to P', to promote exploration and avoid local optima. The remaining $n - 2$ individuals of the population are iteratively computed as follows. Two individuals $i, j \in P$ are first selected by tournament selection with a preset probability [28]. Crossover is then performed on i and j with a preset probability, by swapping the location of a random set of items from i to j. A mutation is subsequently performed on i and j by randomly swapping an approximate number of items within each individual, before adding them to P'. This is repeated until P' is fully populated. A new population is then iteratively selected for a preset number of iterations until the fittest individual of the final solution is returned. The complete algorithm is outlined in Figure 2.2.

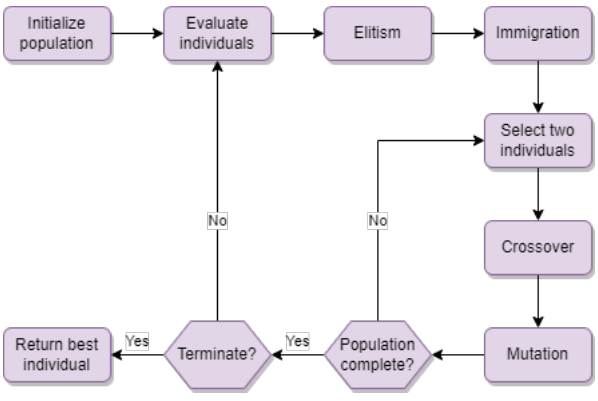

Figure 2.2: The genetic algorithm approximating the batch picking problem.

With a preset number of iterations and population count, the algorithm terminates in a constant number of iterations. In each iteration, the picking distance of a batch partitioning is evaluated in $\mathcal{O}(|O|\mathcal{T})$ time and $\mathcal{O}(|O|\mathcal{T})$. This results in a total running time of $\mathcal{O}(|O| \cdot \mathcal{T})$ with a large constant factor, requiring $\mathcal{O}(|O| + \mathcal{S})$ space.

3
Method

All algorithms are evaluated to assess their optimality and feasibility of being applicable in practice. The following sections present how empirical performance evaluation was performed against the warehouse model defined in Section 2.1.

3.1 Travelling Salesman Algorithms

To measure the performance of all travelling salesman algorithms described in Section 2.2, they are first evaluated in isolation before being applied in combination with batch picking techniques. However, since the travelling salesman problem is a heavily researched topic, the intention is not to evaluate already proven performances in general. Only the optimality and feasibility of applying the techniques to problem instances that may occur in existing warehouses are considered.

The performance evaluation of travelling salesman algorithms is encapsulated into a benchmarking suite. Traversal distances, running times, and memory usage are chosen as performance metrics. In the suite, all algorithms are benchmarked over multiple runs on subgraphs of the warehouse model. Each subgraph respectively consists of 4, 8, 16, 32, and 64 nodes. In each subgraph, the nodes are evenly spaced by identifier index in the graph, to ensure that all algorithms are evaluated against equal subgraphs. The sizes of subgraphs are subsequently chosen by the range of nodes that commonly occurred in batches during early empirical testing of batch picking algorithms. However, a node count of 64 is far greater than empirically occurring sizes and is solely used as an upper bound. The performance benchmarks are summarized in Section 4.2, while the raw benchmarking data is presented in Appendix B.1.

3.2 Batch Picking Algorithms

Since the empirical performance of batch picking algorithms is highly dependent on the chosen travelling salesman method, it is implausible to expect feasible performance if both problems are optimized using techniques. The intention is hence to evaluate if various combinations of techniques have enough optimality while being feasible in practice. That is, finding the extent of how much reductions in picking distances are worth increasing computational costs. A subsequent intention is to also determine to what extent optimally and feasibility may be achieved by optimizing

only one of the two problems.

The performance evaluation of batch picking algorithms is also encapsulated into a benchmarking suite, where traversal distances, running times, and memory usage are the sole performance metrics. All algorithms are benchmarked over multiple runs on the warehouse model in combination with all travelling salesman techniques deemed feasible in batch picking contexts. Each combination is subsequently benchmarked on equal order sets of 60, 107, 223, and 769 orders respectively, the smallest of which is displayed in Appendix A.2. Each set consists of historical order data from the modelled warehouse, with a variety of sizes deemed sufficient for a fair evaluation of the algorithms. A permanent starting coordinate was chosen as the bottom left corner of the layout in Figure 2.1, to model a figurative packing area. The batch size was further set to 16 nodes per batch since this was assumed to be a reasonable size in practice.

A time limit was further set to 15 minutes for all runs before they are terminated. The limit was chosen per informal discussions with Ongoing Warehouse, by the approximate frequency an order set is updated with incoming orders in practice. Any feasible algorithm must hence terminate before the order set is figuratively updated. The benchmarks are summarized in Section 4.2, while the raw data is presented in Appendix B.2.

4

Results

In this chapter, the implementation of the .NET library is described in further detail, followed by the benchmarking performance of all algorithms. The performance metrics are displayed in bar graphs for easy comparison of the performances, with the raw data displayed in tables in Appendix B. All performance was further evaluated on a laptop running a 64-bit build of Windows 10 with 32 GB of memory and an Intel®Core™ i7-8650U processor clocked at 1.90 GHz.

4.1 Implementation

The implementation of the .NET batch picking library is divided into a number of modules displayed in Figure 4.1. Each encapsulates implementations of the algorithms described in section Chapter 2, with datasets required for running various components in isolation. Only the relevant interfaces, abstract classes, and static factory classes are exposed, ensuring minimal modular dependencies. An overview of all modules is briefly described in the following sections.

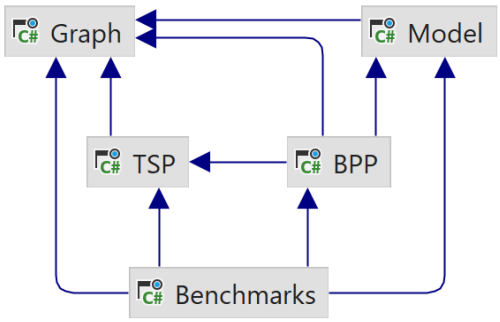

Figure 4.1: Modular dependencies within the batch picking library.

4.1.1 `Graph` Namespace

The core of modelling a warehouse as the complete graphs described in Section 2.1 is composed in the `Graph` namespace. It consists of the `IGraph` interface displayed in Listing 4.1, defining the required graph behavior. The interface dictates that a complete graph exposes a set of unique nodes as integers, all of which are pairwise connected with a single undirected weighted edge exposed by the method `Weight(int,`

`int)`. The `NodeAt((int, int))` method further enables mapping coordinates to nodes, as required for converting shelf mappings to nodes.

Listing 4.1 Interface used for modelling warehouse layouts as complete weighted graphs described in Figure 2.1.

```
interface IGraph
{
    IReadOnlyList<int> Nodes { get; }
    double Weight(int, int);
    int NodeAt((int, int));
}
```

The module contains one implementation of the `IGraph` interface, computing a complete graph of a warehouse layout directly as described in Section 2.1. Note that the implementation has a slight memory overhead for storing nodes, weights, and coordinate mappings. This enable `Weight` and `NodeAt` to be performed in $\mathcal{O}(1)$ time, and `Nodes` to be safely exposed without copying the set of nodes.

4.1.2 `Model` Namespace

The `Model` namespace is essentially wrapping the complete graph representation to compose the full model instance from Section 2.1. Model instances are constituted by the interface `IModel` displayed in Listing 4.2. It dictates models must expose the underlying graph, a set of order identifiers, and a preset batch size. The node representation of items in an order is further exposed using the method `Subset(int)`.

Listing 4.2 Interface used for complete warehouse models described in Figure 2.1.

```
interface IModel
{
    IGraph Graph { get; }
    IReadOnlyList<int> Subsets { get; }
    int Size { get; }
    IEnumerable<int> Subset(int);
}
```

A single implementation of `IModel` is composed in the `Model` namespace. The model instance is created using a pre-computed `IGraph` instance, and a set of orders and shelf mappings defines in files such as in Appendix A.2 and Listing A.2. The order set is modelled as a dictionary mapping identifiers to lists of nodes using the shelf mappings and `IGraph` object. A copy of the order identifiers is further stored in a separate list, enabling them and the nodes of an order to be fetched in $\mathcal{O}(1)$ time. Given as input is also a batch size defining the maximum size of batches in a batch picking instance, and an origin coordinate, which is mapped to a node and added to each order, to model batches as instances of the travelling salesman problem.

4.1.3 TSP Namespace

Implementations of all travelling salesman algorithms described in Section 2.2 are included in the `TSP` namespace as inheritors of the abstract class `TspSolver` displayed in Listing 4.3. It has a single non-abstract method `(double, bool) Solve(IGraph, IReadOnlyList<int>)`, computing the traversal distance of a set of nodes in a provided graph. A boolean is also returned to indicate if computations are terminated due to a timeout. The solution itself is computed using the abstract method `Solve(CancellationToken)`, implemented by all inheritors of `TspSolver` to encapsulate the respective algorithm. To enable externally terminating the method, a cancellation token is provided. Note that the `IGraph`, set of nodes, and computed solution are written to private members, to simplify data access between the wrapping and abstract methods. As the computations terminate, the wrapper method finally verifies that the solution solves the problem in the given instance.

Listing 4.3 Abstract class used for implementing algorithms addressing the travelling salesman problem, as described in Section 2.2 and Section 3.1.

```
abstract class TspSolver
{
    private protected abstract void Solve(CancellationToken);
    (double, bool) Solve(IGraph, IReadOnlyList<int>);
}
```

4.1.4 BPP Namespace

Algorithm implementations of batch picking algorithms described in Section 2.3 are provided in the `BPP` namespace by extending the abstract class `BppSolver` in Listing 4.4. The only non-abstract method `(double, bool) Solve(IModel, TspSolver)` computes the picking distance of an `IModel` instance, with a boolean is indicating if computations are terminated due to timeout. A `TspSolver` is further provided for solving instances of intermediate batches. Solutions are subsequently computed by the method `Solve(CancellationToken)` implemented by inheritors of `BppSolver`, with a cancellation token used for terminating the method. The `IModel` instance, `TspSolver` instance, and solution are all written to private members, for easier data access between the two methods. As computations terminate, the solution is finally verified by the wrapper before the picking distance is returned.

Listing 4.4 Abstract class used for implementing algorithms addressing the batch picking problem, as described in Section 2.3 and Section 3.2.

```
abstract class BppSolver
{
    private protected abstract void Solve(CancellationToken);
    (double, bool) Solve(IModel, TspSolver);
}
```

4.1.5 `Benchmarks` Namespace

The `Benchmarks` namespace provides the implementations of the two benchmarking suites outlined in Chapter 3, implemented using the `BenchmarkDotNet` package [29]. This enables all algorithms to be easily benchmarked in equal settings, with performance metrics automatically computed. A custom column was required to display computed distances in the output since this functionality was not provided by `BenchmarkDotNet`. Distances marked with * indicate that a valid solution was provided, but computations were terminated due to timeout. The units of empirical computation times and memory usage were further post-processed, to enable higher readability. The empirical benchmarks are summarized in Section 4.2 and Section 4.3, with the raw data provided in Appendix B.

4.2 Travelling Salesman Performance

The following graphs and tables present a summary of theoretical and empirical results for the algorithms addressing the travelling salesman problem from Section 2.2. Table 4.1 displays theoretical computational complexities as functions of the graph size $|V|$, as formalized in Section 2.2. Algorithm complexities marked with (*) indicate that they have a large hidden constant factor.

Algorithm	**Time**	**Space**
Nearest Neighbor	$\mathcal{O}(\|V\|^2)$	$\mathcal{O}(\|V\|)$
Held-Karp	$\mathcal{O}(\|V\|^2 \cdot 2^{\|V\|})$	$\mathcal{O}(\|V\| \cdot 2^{\|V\|})$
Christofides	$\mathcal{O}(\|V\|^3)$	$\mathcal{O}(\|V\|^2)$
Linear Programming	-	$\mathcal{O}(\|V\|^2)$
Ant Colony Optimization	$\mathcal{O}(\|V\|)^*$	$\mathcal{O}(\|V\|)$
Simulated Annealing	$\mathcal{O}(\|V\|)^*$	$\mathcal{O}(\|V\|)$

Table 4.1: Computational complexities of travelling salesman algorithms from Section 2.2. Complexities marked with * have a large constant factor.

Performance metrics computed by all algorithms are displayed Figure 4.2. A complete table of all empirical benchmarks is displayed in Table B.1. Since the Held-Karp algorithm ran out of memory and was unable to run until completion for graphs of 32 and 64 nodes, empirical statistics are only displayed for the smaller graphs. Due to significant differences in the empirical running times and memory usage, a logarithmic scale is used to enable higher readability. Termination delays on timeout are also present in some algorithms, which occasionally results in running times extending beyond the timeout boundary.

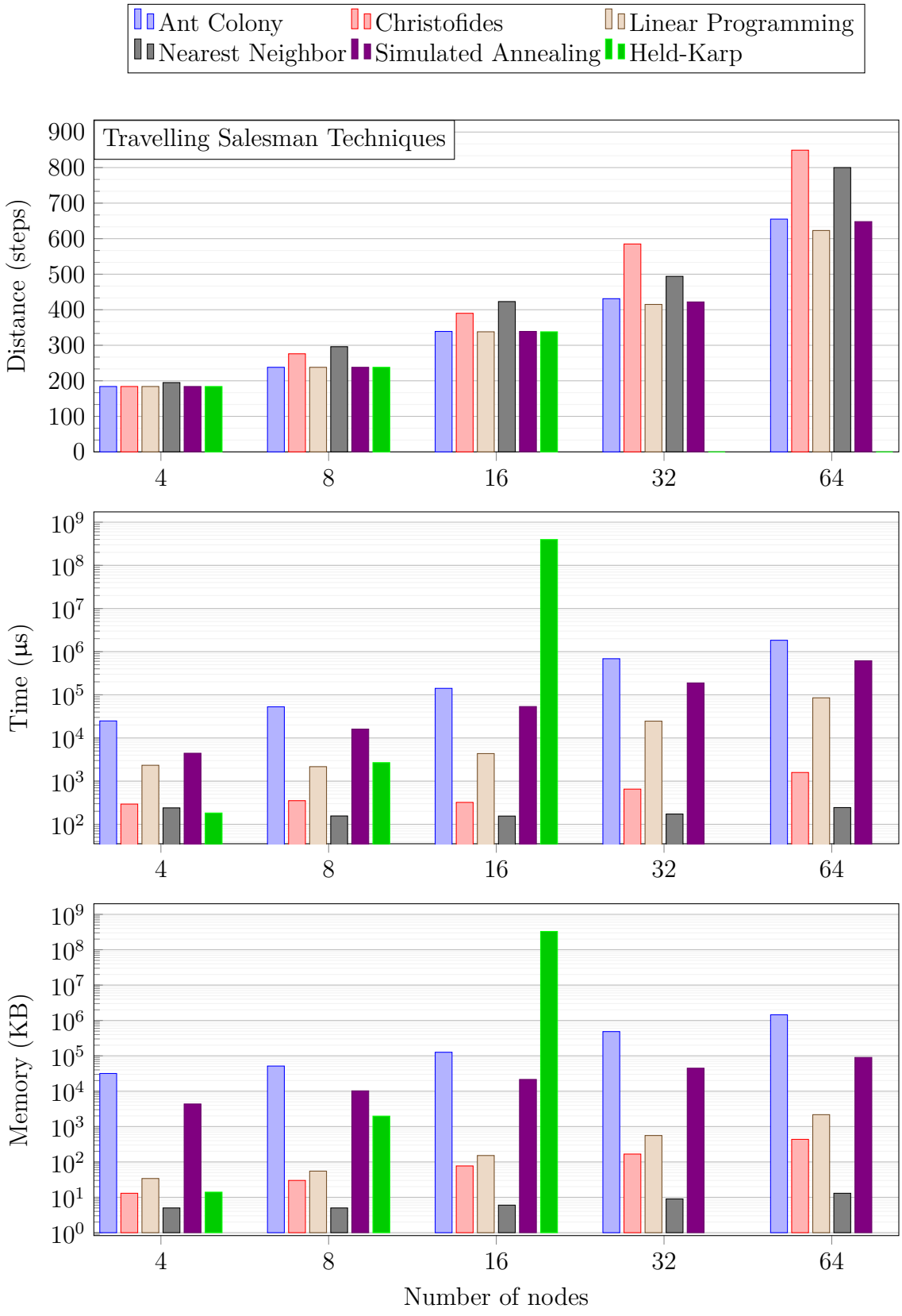

Figure 4.2: Benchmark performance of travelling salesman algorithms.

4.3 Batch Picking Performance

The following section presents a collection of graphs and tables summarizing theoretical and empirical results for the algorithms described in Section 2.3. Table B.2 displays the computational complexities described in Section 2.3 as functions of the order count $|O|$ and the complexity of travelling salesman algorithms. Algorithm complexities marked with $(^*)$ indicate that they have a large hidden constant factor.

Algorithm	**Time**	**Space**
First In, First Out	$\mathcal{O}(\|O\| \cdot \mathcal{T})$	$\mathcal{O}(\mathcal{S})$
Greedy Batching	$\mathcal{O}(\|O\|^2 \cdot \mathcal{T})$	$\mathcal{O}(\mathcal{S})$
Greedy Partition	$\mathcal{O}(\|O\|^2 \cdot \mathcal{T})$	$\mathcal{O}(\mathcal{S})$
Proximity	$\mathcal{O}(\|O\|^2 \cdot \|V\|^2 + \|O\| \cdot \mathcal{T})$	$\mathcal{O}(\|O\| + \mathcal{S})$
Simulated Annealing	$\mathcal{O}(\|O\| \cdot \mathcal{T})^*$	$\mathcal{O}(\|O\| + \mathcal{S})$
Genetic	$\mathcal{O}(\|O\| \cdot \mathcal{T})^*$	$\mathcal{O}(\|O\| + \mathcal{S})$

Table 4.2: Computational complexities batch picking algorithms from Section 2.3. Complexities marked with * have a large constant factor.

The empirical benchmarks of all batch picking algorithms are summarized in the following figures. A complete table of all empirical data is displayed in Table B.2. Each figure displays the performance metrics computed by one algorithm in combination with various travelling salesman algorithms, on order sets of several sizes. Since the performance of the Held-Karp algorithm was infeasible even on small batches, it was excluded from all benchmarks. Distances of zero further indicate that the algorithm was unable to compute a solution before the timeout was reached. Due to significant differences in the empirical running times and memory usage, a logarithmic scale is used to enable higher readability. Termination delays on timeout are also present in some algorithms, which occasionally results in running times extending beyond the timeout boundary.

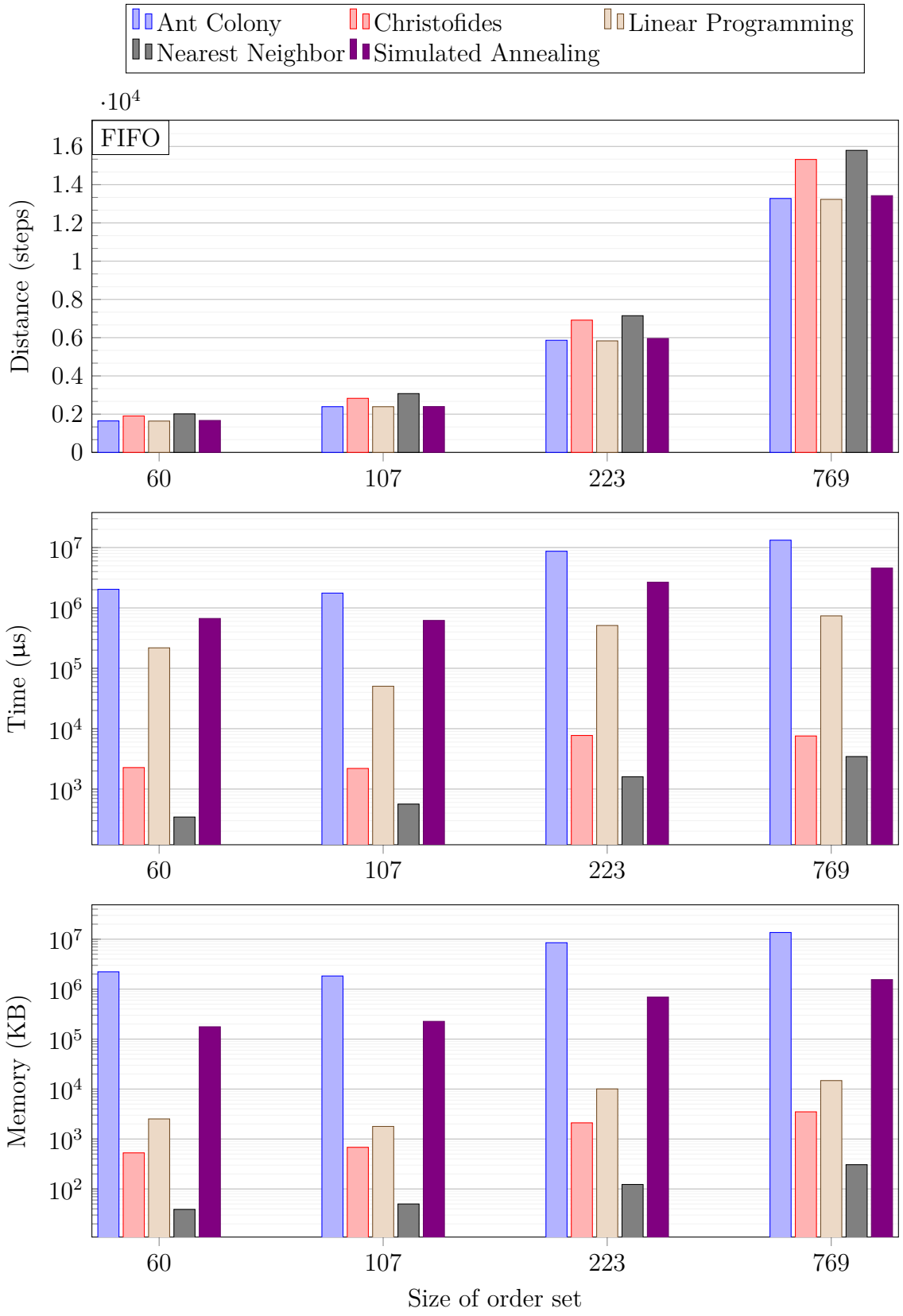

Figure 4.3: Benchmark performance of the FIFO algorithm combined with travelling salesman algorithms.

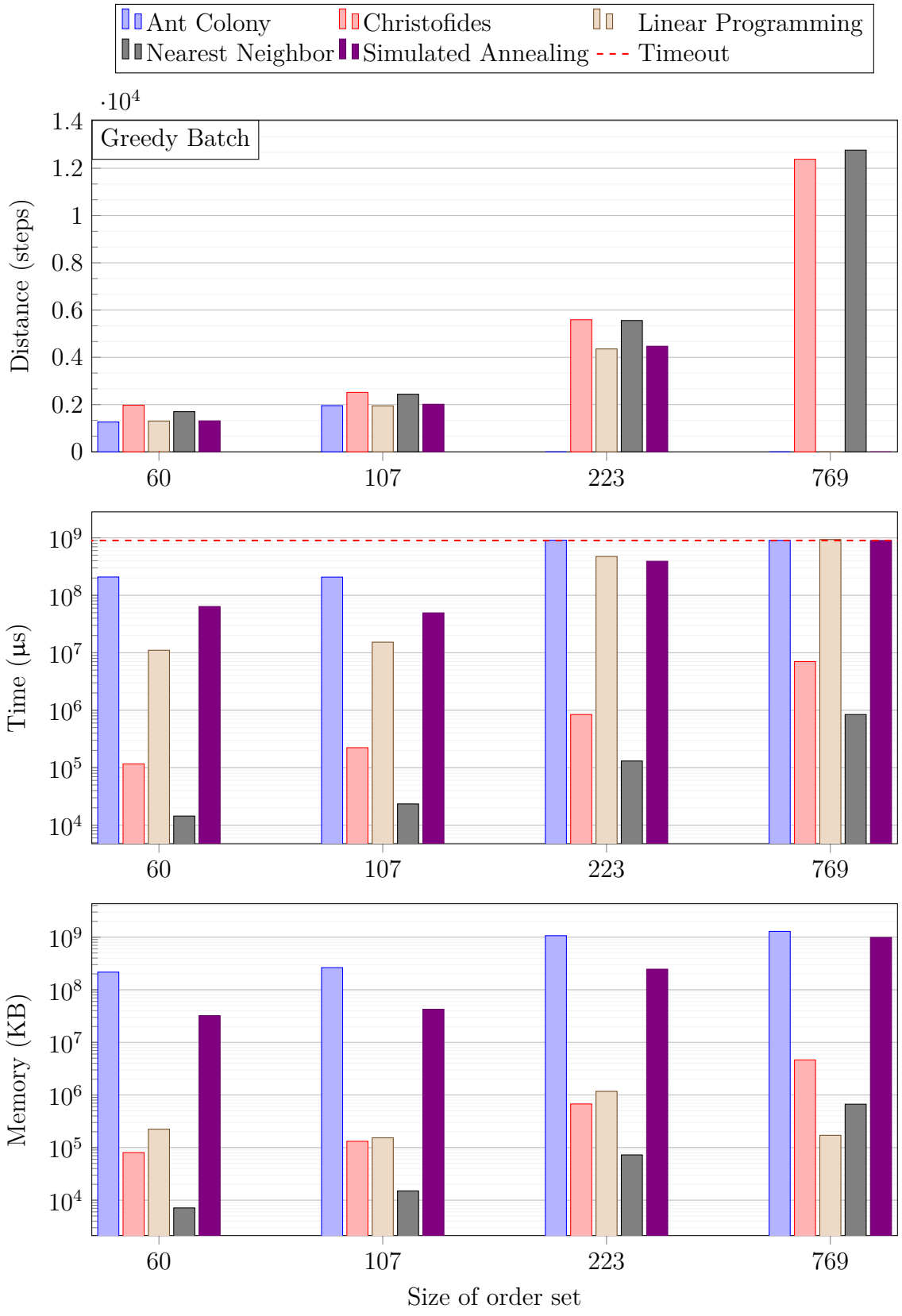

Figure 4.4: Benchmark performance of the greedy batching algorithm combined with travelling salesman algorithms.

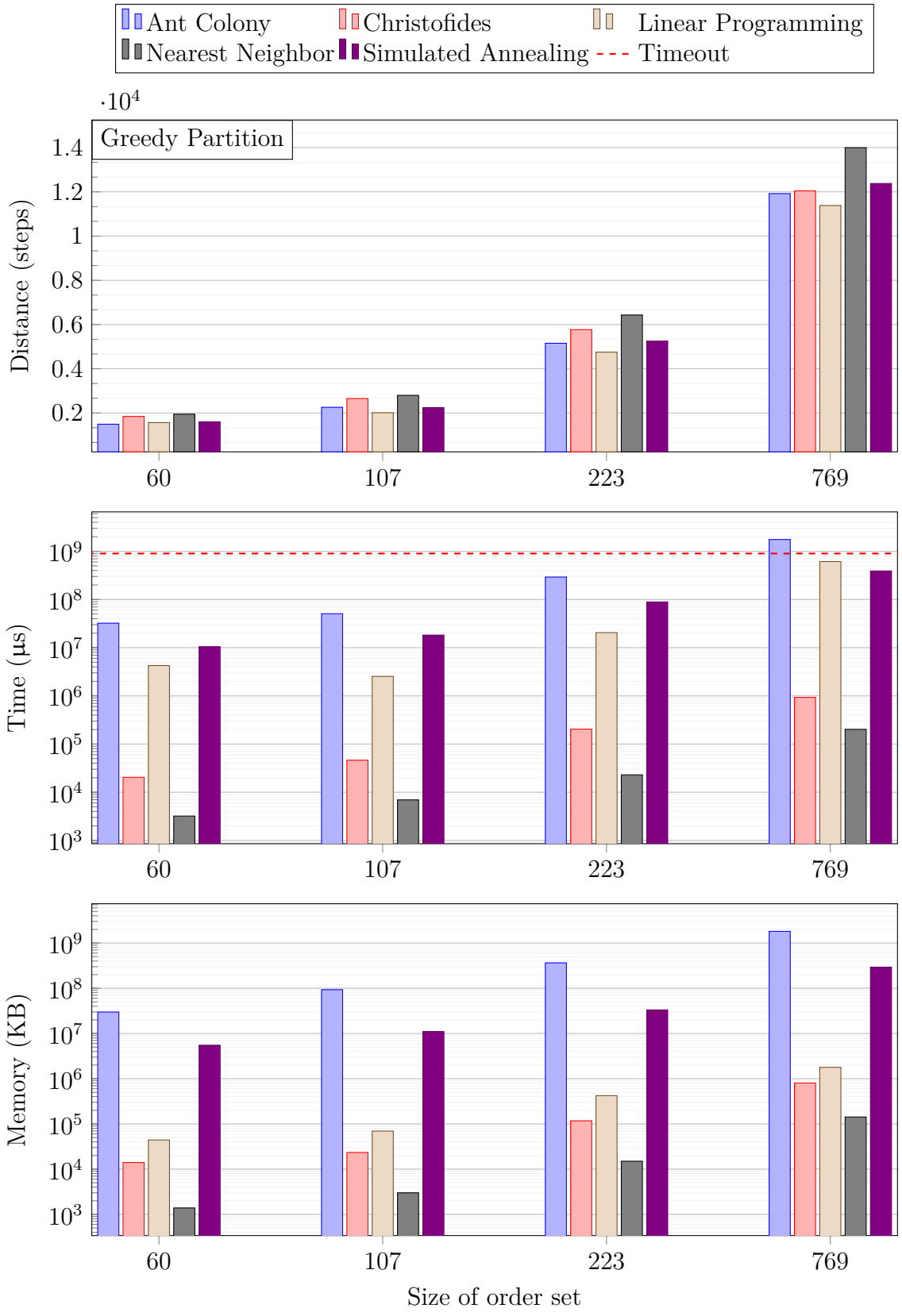

Figure 4.5: Benchmark performance of the greedy partition algorithm combined with travelling salesman algorithms.

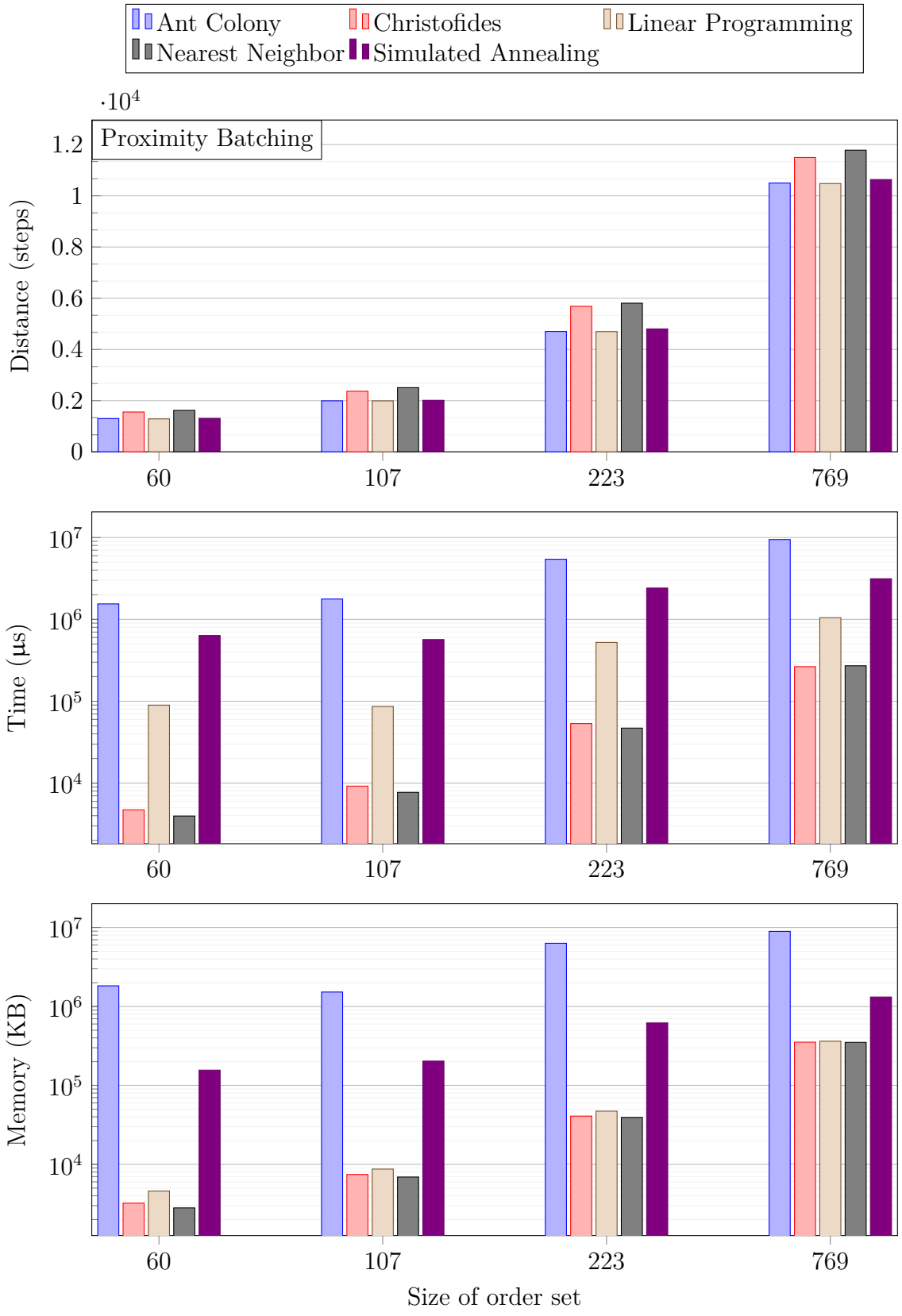

Figure 4.6: Benchmark performance of the proximity algorithm combined with travelling salesman algorithms.

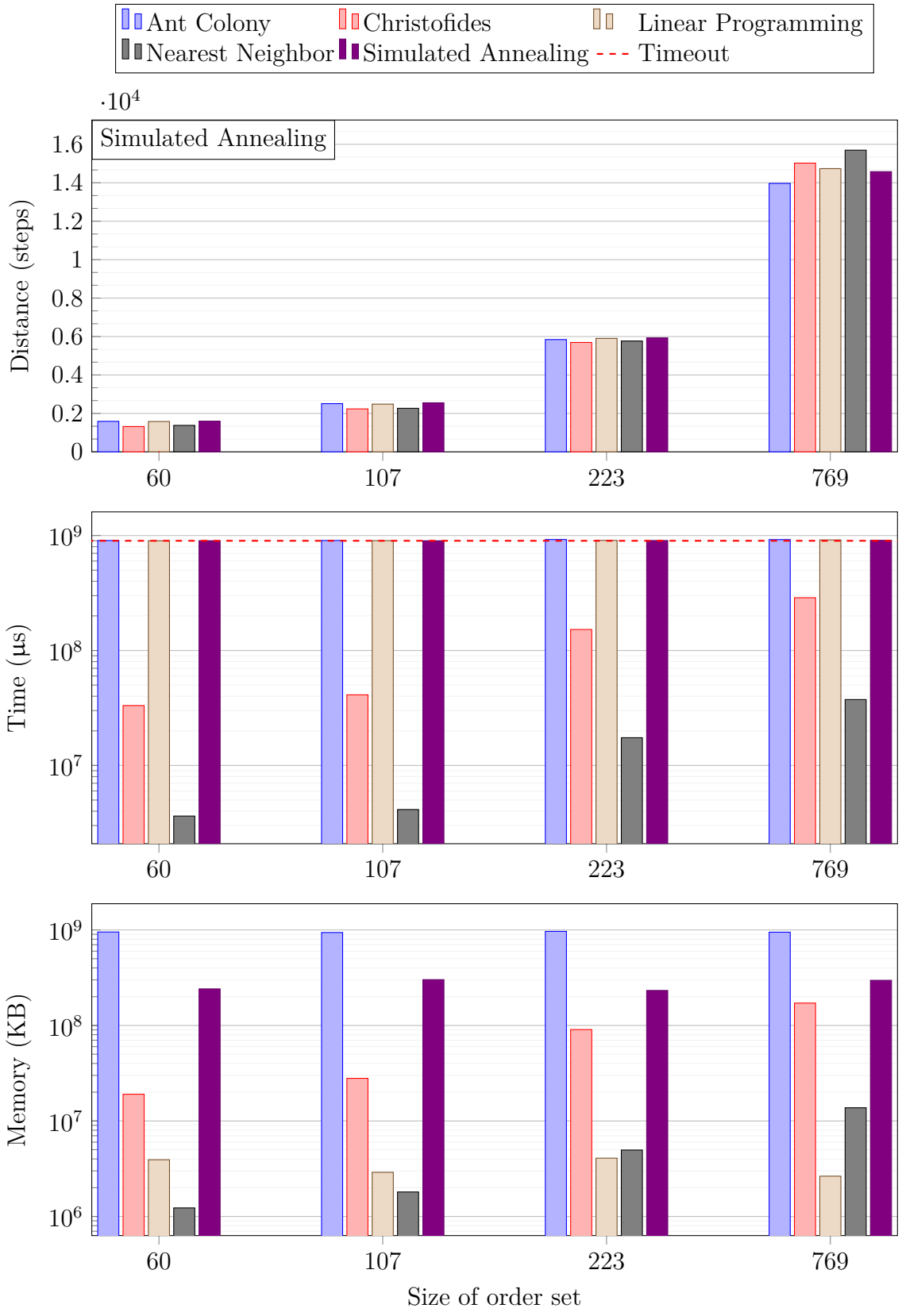

Figure 4.7: Benchmark performance of the simulated annealing algorithm combined with travelling salesman algorithms.

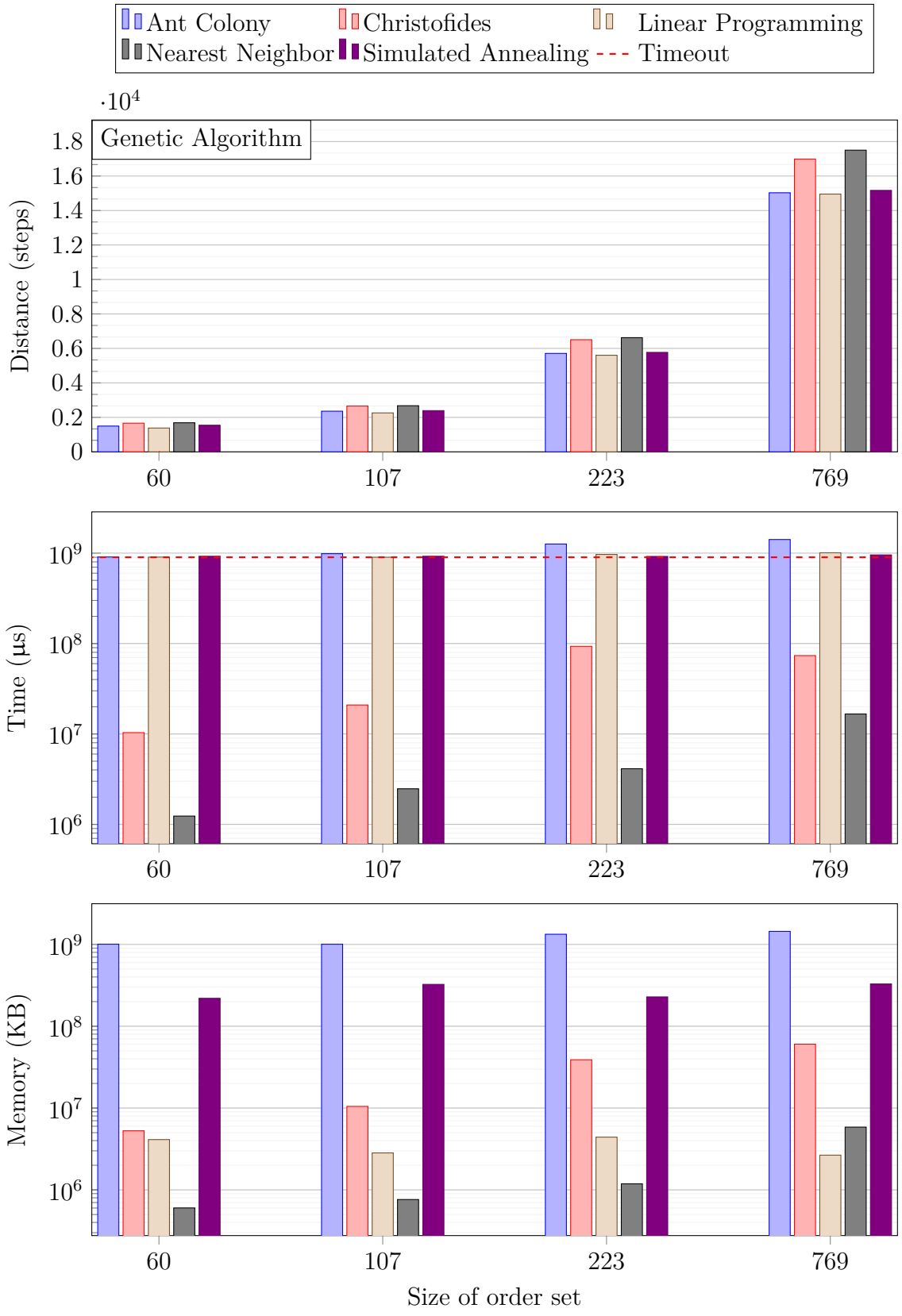

Figure 4.8: Benchmark performance of the genetic algorithm combined with travelling salesman algorithms.

5

Discussion

From Chapter 4, it is evident that certain techniques perform better than others. Although great care has been taken to implement efficient algorithms, the performance is heavily dependent on implementation details and improvements can be made. The results were further produced on an ordinary laptop, which can look differently using high-performance computing clusters, as an example. In the end, however, better hardware should not be a replacement for well-thought-out algorithms, which is why studies on such matters are meaningful.

5.1 Travelling Salesman Evaluation

Figure 4.2 displays how the Held-Karp algorithm finds optimal solutions to problem instances of sizes up to 16 nodes. However, due to its exponential computational costs, huge amounts of time and memory are required already at 16 nodes. As such, the algorithm is unable to a find solution in larger instances, as anticipated, making it infeasible in practice. The linear programming algorithm also finds optimal solutions to smaller instances while providing the shortest distances to the large ones. Since optimal solutions to the larger instances are unknown, the optimality of the linear programming algorithm is indeterminable. Nevertheless, it is plausible that it is close to optimal. It also presents relatively low time and memory requirements with linear growth, making it reasonably feasible for batch picking instances.

The simulated annealing and ant colony techniques present comparable, yet slightly longer distances than the linear program. Although they experience linear computational growth, they require considerably higher time and memory already on smaller instances. They may as such not be feasible when applied repeatedly on batch picking instances. Using a linear program for addressing the batch picking problem over these techniques may hence be justified already.

Christofides algorithms and the nearest neighbour techniques provide the worst distances overall. Their polynomial complexities do however result in considerably cheaper computational costs than other methods. While longer distances may result in poor optimality of batch picking techniques, the low computational costs may permit a larger selection of batch picking techniques.

5.2 Batch Picking Evaluation

Figure 4.3 displays that both distances and computational costs of FIFO experience similar characteristics as travelling salesman techniques do in isolation. This is not surprising, since the techniques are applied repeatedly on equal instances to compute a solution. This subsequently results in the same performance characteristics across the various problem sizes. While the heavier techniques further require higher computational costs, all are far from being infeasible in practice.

The greedy batch heuristics in Figure 4.4 presents overall shorter distances than FIFO across all problem instances. However, all but the polynomial techniques become unable to compute solutions for the larger datasets. Nevertheless, with the polynomial techniques, the heuristic still outperforms the best combination with FIFO as the dataset grows. Figure 4.5 further displays that the greedy partition heuristic computes similar distances with polynomial traversal techniques. The noticeable decrease in computational costs also permits utilizing techniques on the larger datasets, which results in slightly shorter distances. The improvement may however not be worth the perhaps infeasible computational costs.

Figure 4.6 displays how proximity batching results in overall superior distances across all datasets. This is evident for the largest dataset, where its worst performance with the nearest neighbor technique beats all but a few of all other method combinations. The best combination with a linear program is subsequently even better, with only slight increases in computational costs. Similar to FIFO, proximity batching only utilizes traversal techniques for evaluating the final picking distances. As such, their computational performances are nearly identical and far from being infeasible. This further implies that computational performance is highly dependent on the selection of traversal techniques, but computing a good partition of batches is perhaps not.

Figure 4.7 and Figure 4.8 display how simulated annealing and the genetic algorithm both suffer from extremely high computational costs, with picking distances not outperforming even FIFO. The timeout limit is further reached for all but the polynomial techniques across the datasets. While this is permitted, it also results in infeasibly large memory overheads to achieve no picking distance improvements. The high computational times even with polynomial techniques without distance improvements subsequently make the two methods infeasible.

A continuous trend across all batch picking techniques is that combinations with a linear program tend to result in the shortest distances, with feasible increases in computational costs in most cases. Combinations with simulated annealing and ant colony optimization generally require infeasibly high computational costs to achieve no improvements in the distance. The polynomial techniques further tend to achieve noticeably better computational performance, whilst producing slightly longer distances. In the choice of technique combinations, the most important feasibility consideration is hence to balance how lower distances produced by a linear program are worth the computationally higher costs than using polynomial techniques. However, since the superior proximity batching achieves the shortest distances whilst being

computationally feasible with either technique, it is evident that a linear program is the better choice.

5.3 Limitations

Considering all constraints of real-world batch picking, the project scope is naturally limited. How the chosen limitations may have affected the project execution is described in the following sections.

5.3.1 Model Limitations

The several model simplifications may substantially affect the applicability of batch picking techniques in real warehouses. For instance, neglecting sizes, weights, and fragility of orders may result in batches which cannot be picked at once in practice. However, some warehouses are only concerned with handling small items, such as jewelry. While the batch picking techniques may hence not be applicable in all warehouses, they may still be valid in some.

Since warehouses often guarantee that orders are shipped within certain ranges of time, certain orders are of higher picking priority to be delivered on time. However, the timestamps of orders are ignored in the model and orders of high priority may hence be left unpicked indefinitely. As this is not permitted in practice, the applicability of the picking techniques may be questionable. Nevertheless, one may argue they are still applicable on order sets of already prioritized orders.

Modelling warehouses as graphs further requires the assumption that pickers can move freely within the warehouse. Most often than not, several pickers are traversing the aisles simultaneously in reality, which can result in pickers blocking each other. Since this is not considered in the model, the techniques may only be applicable in warehouses with only a single picker or where all pickers can move around without interference.

5.3.2 Benchmarking Limitations

The distance, time and memory usage displayed in Appendix B have all been averaged over three benchmarking runs for all technique combinations. This was deemed sufficient for the deterministic algorithms, as they produced identical distances with approximately equal computational costs in each run. However, the performance of stochastic techniques may likely benefit from additional runs. It was hence considered to expand the number of runs to twelve, another option from `BenchmarkDotNet`, but was deemed out of scope due to limitations of time.

Since all algorithms were further evaluated against only a single warehouse, they may not generalize well. Different layouts, item placement schemes, or item distributions within orders can notably impact technique performance. A significant detail of the batch picking benchmarking suite is subsequently the limited sizes of order sets. Although multiple sets of various sizes were used in the one model, more data may be required to make decisive conclusions on algorithm performance and generalisation.

As displayed in Appendix B.2, certain combinations of techniques, such as the genetic algorithm combined with simulated annealing in Figure 4.8, appear to consume unreasonable amounts of memory for the standard laptop running them. The arguably misleading performance is derived from how `BenchmarkDotNet` monitor memory usage. It measures the total allocated memory over the several runs of a combination, without considering how C# performs garbage collection. Nevertheless, if minimal memory usage is crucial, opting for other techniques is still preferred.

5.4 Ethical Considerations

As with all optimization, the consequences of using it in practice must always be thoroughly analyzed. Since all algorithms produce warehouse locations to visit in specific orders, it is arguably unrealistic that human pickers may directly navigate the locations without difficulties or external interruptions. The presented optimization techniques may hence be better suited for robotic pickers in automated warehouses. If proven more cost-efficient than using human pickers, reduced operational costs may potentially see warehouse staff being replaced.

5.5 Future Work

The project does by no means cover every aspect of the batch picking problem and its role in warehouse logistics. Several extensions to consider are hence briefly discussed below.

5.5.1 Dynamic Model

Since a highly simplified model is utilized throughout the project, a considerable extension is to model the dynamic aspects of warehouse characteristics. One example is to utilize continuous order sets that are frequently updated with incoming orders since warehouses in reality do not stop receiving orders while existing orders are picked. A subsequent extension is to also consider the timestamps of orders since warehouses are generally obligated to guarantee ranges of delivery dates to ensure customer satisfaction. To enable the extensions, all algorithms would require modifications with more extensive solution verification, which may restrict the overall performance.

Another dynamic is to also model the behaviour of pickers since most warehouses are operated by humans. Picking orders may for instance require time and effort beyond just traversal distances, such as correctly navigating the layout, climbing shelves, or stopping to make conversation. Probabilistic traversal costs could hence be introduced to the warehouse model, to simulate various scenarios, or agents traversing the layout can be utilized to simulate how batches are picked.

5.5.2 Solution Visualisation

Since batches and picking paths are currently outputted solely as numbers, verifying the plausibility of solutions is generally counter-intuitive. Visualizing the picking paths of batches and how orders are distributed in warehouses would hence be beneficial to further evaluate the feasibility of using the techniques in practice. By also visualizing how simulated pickers traverse the warehouse, one may verify how intuitive solutions are for human pickers.

5.5.3 Extensive Benchmarking

Since the limited scope of benchmarking against only a single warehouse model may result in techniques not being able to bridge the gap into practice, it is evident that more exhaustive evaluation may be required. An extension would hence be to streamline how warehouse models are created, to enable evaluation against a more vast variety of warehouses. This may however be restricted by the limited availability of warehouse layout data. On the other hand, since batch picking is in essence a special case graph problem, it may be enough to generate problem instances from a variety of existing graphs. This could further enable evaluating how well batch picking algorithms generalize to graphs beyond warehouse layouts.

6

Conclusion

In isolation, the Held-Karp algorithm is optimal in small instances but quickly grows infeasible for usage in batch picking applications. Whilst comparably short paths were produced by several methods, using a conventional linear program was superior. It produced optimal paths on small instances and the best paths on instances with unknown optimality. Unlike other heavier techniques, the computational costs of a linear program were also clearly feasible for most batch picking applications. However, the computational requirements of the polynomial methods are substantially better than all others, with the disadvantage of slightly worse paths. Choosing a technique to work best in practice is hence to balance the optimality of a linear program with the low computational costs of polynomial methods. Proximity batching is overall the superior choice. The shortest distances are produced in combination with a linear program, but the choice of technique matters not in terms of computational costs that are feasible in practice. While other heuristics also produce short distances, their high computational costs limit the selection of feasible combinations. The probabilistic approaches generally perform poor, producing the longest distances with infeasible computational costs for usage in practice. FIFO further produce surprising short distances, since no optimization is performed to partition the batches. However, since proximity batching utilizes equally low computational costs as FIFO, it's the evident choice of technique.

Overall, picking distances of order sets can be substantially reduced by applying a variety of batch picking techniques. Whilst most methods reduce picking distance, only a few are feasible in practice. In the selection of feasible methods, the task at hand is primarily to balance optimality with computational performance. Heavier optimization methods generally gain less in optimality than they lose in computational performance. On the contrary, lighter techniques tend to lose only a little distance, while achieving incredibly cheap computational costs. Heuristics handcrafted from domain knowledge further tend to perform better than generalized algorithms.

Bibliography

[1] N. Boysen, R. de Koster, and F. Weidinger, "Warehousing in the e-commerce era: A survey," *European Journal of Operational Research*, vol. 277, no. 2, pp. 396–411, 2019. DOI: 10.1016/j.ejor.2018.08.023.

[2] P. Yang, Z. Zhao, and H. Guo, "Order batch picking optimization under different storage scenarios for e-commerce warehouses," *Transportation Research Part E: Logistics and Transportation Review*, vol. 136, p. 101 897, 2020, ISSN: 1366-5545. DOI: 10.1016/j.tre.2020.101897.

[3] J. Drury, "Towards more efficient order picking," *IMM monograph*, vol. 1, no. 1, pp. 1–69, 1988.

[4] L. Pansart, N. Catusse, and H. Cambazard, "Exact algorithms for the order picking problem," *Computers & Operations Research*, vol. 100, pp. 117–127, 2018, ISSN: 0305-0548. DOI: 10.1016/j.cor.2018.07.002.

[5] *Ongoing warehouse*, https://ongoingwarehouse.com/, Accessed: 2021-12-02.

[6] E. W. Dijkstra *et al.*, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[7] M. Flood, "The traveling-salesman problem," *Operations Research*, vol. 4, pp. 61–75, 1956. DOI: 10.1287/opre.4.1.61.

[8] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*, Springer, 1972, pp. 85–103.

[9] M. Bellmore and G. L. Nemhauser, "The traveling salesman problem: A survey," *Operations Research*, vol. 16, no. 3, pp. 538–558, 1968.

[10] S. Hougardy and M. Wilde, "On the nearest neighbor rule for the metric traveling salesman problem," *Discrete Applied Mathematics*, vol. 195, 2014. DOI: 10.1016/j.dam.2014.03.012.

[11] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, "Approximate algorithms for the traveling salesperson problem," in *15th Annual Symposium on Switching and Automata Theory (swat 1974)*, 1974, pp. 33–42. DOI: 10.1109/SWAT.1974.4.

[12] G. Gutin, A. Yeo, and A. Zverovich, "Traveling salesman should not be greedy: Domination analysis of greedy-type heuristics for the tsp," *Discrete Applied Mathematics*, vol. 117, no. 1, pp. 81–86, 2002, ISSN: 0166-218X. DOI: 10.1016/S0166-218X(01)00195-0.

[13] G. Gutin and A. Yeo, "The greedy algorithm for the symmetric tsp," *Algorithmic Operations Research*, vol. 2, no. 1, pp. 33–36, 2007.

[14] M. Held and R. M. Karp, "A dynamic programming approach to sequencing problems," *Journal of the Society for Industrial and Applied Mathematics*, vol. 10, no. 1, pp. 196–210, 1962. DOI: 10.1137/0110015.

[15] R. Bellman, "Dynamic programming treatment of the travelling salesman problem," *J. ACM*, vol. 9, no. 1, pp. 61–63, 1962, ISSN: 0004-5411. DOI: 10.1145/321105.321111.

[16] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, Tech. Rep., 1976.

[17] R. van Bevern and V. A. Slugina, "A historical note on the 3/2-approximation algorithm for the metric traveling salesman problem," *Historia Mathematica*, vol. 53, pp. 118–127, 2020, ISSN: 0315-0860. DOI: 10.1016/j.hm.2020.04.003.

[18] R. C. Prim, "Shortest connection networks and some generalizations," *The Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[19] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[20] G. Dantzig, R. Fulkerson, and S. Johnson, "Solution of a large-scale traveling-salesman problem," *Journal of the operations research society of America*, vol. 2, no. 4, pp. 393–410, 1954.

[21] M. Dorigo, M. Birattari, and T. Stutzle, "Ant colony optimization," *IEEE Computational Intelligence Magazine*, vol. 1, no. 4, pp. 28–39, 2006. DOI: 10.1109/MCI.2006.329691.

[22] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[23] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *The journal of chemical physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[24] O. Warehouse, *Using FIFO in practice*, Verbal communication: 2022-02.

[25] M. Sigvardsson and C. Persson. "Walk less, pick more: Choosing optimal batches of orders in a warehouse." (2020), [Online]. Available: https://hdl.handle.net/20.500.12380/301743.

[26] M. Mitchell, *An introduction to genetic algorithms*. MIT press, 1998.

[27] C.-M. Hsu, K.-Y. Chen, and M.-C. Chen, "Batching orders in warehouses by minimizing travel distance with genetic algorithms," *Computers in Industry*, vol. 56, no. 2, pp. 169–178, 2005, ISSN: 0166-3615. DOI: 10.1016/j.compind.2004.06.001.

[28] B. L. Miller, D. E. Goldberg, *et al.*, "Genetic algorithms, tournament selection, and the effects of noise," *Complex systems*, vol. 9, no. 3, pp. 193–212, 1995.

[29] *Benchmarkdotnet*, https://github.com/dotnet/BenchmarkDotNet, Accessed: 2022-02-01.

A

Warehouse Model

A.1 Warehouse Layout

Listing A.1 Two-dimensional grid layout of a warehouse.

```
#############################################################################################
#.........................................................................###################################
#.###########################.#################################.###################################
#.###########################.................................###################################
#.........................###########################.....................#########################
#.###########################.#################################################.#########################
#.###########################.................................................#########################
#.........................###################################################.#########################
#.###########################.#################################################.#########################
#.###########################.................................................#########################
#.........................###################################################.#########################
#.###########################.#################################################.#########################
#.###########################.................................................#########################
#.........................#############################################.....................#
#.........................#############################################.####################.#
#.....................................................................................................#
#.........................###############.########################....#####################.#
#.........................###############.########################....#####################.#
#.....................................................................................................#
#.........................###############.########################.#########################
#.....................................................#################.....#########################
#.....................................................#################.#####################################
#.....................................................###############.#####################################
#.....................................................###############.#####################################
#.....................................................###############.#####################################
#.....................................................#################.#####################################
#.....................................................###############.#####################################
#.....................................................###############.#####################################
#.....................................................#################.#####################################
#.....................................................###############.#####################################
#.....................................................###############.#####################################
#.....................................................#################.#####################################
#.....................................................###################.####################################
#.....................................................#################.#####################################
#############################################################################################
```

A.2 Example of order dataset

Table A.1: Order dataset consisting of 60 orders.

Order ID	Item ID	Order ID	Item ID	Order ID	Item ID
159413	I-38-02	185777	G-02-03	185858	E-08-04
181958	K-21-02	185777	PB-11-02	185858	E-10-03
181958	J-05-03	185777	K-18-06	185858	E-09-02
181958	K-21-05	185777	K-18-06	185858	E-07-03
181958	J-17-04	185777	X	185858	E-15-02
181958	K-22-01	185815	G-23-04	185858	E-09-04
181958	E-13-05	185816	I-10-01	185858	E-10-03
181958	E-13-06	185816	PB-02-01	185858	I-01-02
181958	PG-04-01	185816	X	185858	E-10-01
181958	G-16-05	185816	X	185858	A-05-03
181958	PH-10-01	185818	H-20-03	185858	E-07-03
181958	H-14-05	185818	PB-06-01	185858	E-08-03
181958	G-15-02	185818	PC-13-01	185858	E-08-04
181958	H-22-01	185818	B-11-03	185858	E-08-05
182749	K-18-06	185818	PB-36-02	185860	TF-14
182749	J-18-05	185818	H-07-06	185862	TC-02
182749	K-18-06	185818	X	185864	F-13-03
182749	F-30-02	185818	PG-06-01	185864	D-32-03
182749	K-18-06	185818	E-13-04	185864	F-10-05
182749	PG-04-01	185818	PJ-04-01	185864	F-05-05
182749	G-15-02	185818	X	185868	G-02-01
182749	H-14-05	185820	G-38-04	185868	E-26-04
182749	H-22-01	185820	E-42-05	185868	K-22-04
182749	G-16-05	185820	G-04-02	185868	TE-04
182749	E-13-06	185820	G-02-03	185868	E-36-02
182749	E-13-05	185822	TF-05	185868	F-23-01
182749	PH-10-01	185823	G-13-05	185869	H-04-05
182903	J-18-06	185824	PF-04-02	185872	F-25-02
182903	G-14-02	185824	PK-05-01	185872	F-25-02
182903	PB-11-01	185824	A-24-06	185872	F-25-02
182903	PB-06-01	185824	F-36-03	185873	F-06-06
182903	G-04-02	185824	I-04-03	185875	F-18-03
182903	G-07-05	185824	G-03-03	185875	F-17-04
182903	G-02-04	185824	E-13-06	185875	I-04-03
182903	TC-19	185824	F-19-06	185878	G-43-03
182903	TE-04	185824	G-04-05	185878	B-18-06
182903	PH-10-01	185825	TH-01	185879	G-20-06
183396	G-08-03	185842	F-23-01	185879	A-26-05
183396	H-06-02	185842	E-36-02	185883	E-04-02
183396	PA-31-02	185842	E-43-04	185883	X -
183396	H-06-02	185842	E-30-05	185884	B-19-03
183396	G-02-05	185843	TH-07	185884	X -

Continued on next page

Table A.1 continued from previous page

Order ID	Item ID	Order ID	Item ID	Order ID	Item ID
183396	G-15-06	185843	D-35-01	185885	PC-08-01
183396	G-08-04	185848	B-11-03	185886	PI-04-02
183396	H-06-01	185850	E-34-05	185887	PA-33-01
183396	H-16-02	185851	PA-33-01	185887	L-11-06
183396	G-02-04	185851	PI-04-01	185888	G-22-03
183396	G-05-02	185851	PI-04-01	185888	PE-18-02
183396	G-14-05	185851	L-11-06	185888	F-18-03
183396	TC-09	185852	X	185890	G-13-02
184608	X	185853	H-26-01	185890	G-02-01
184608	X	185854	PI-11-02	185891	X -
184616	G-32-06	185855	PJ-18-02	185891	X -
184616	X	185855	X	185891	G-20-03
184901	H-35-01	185856	K-02-04	185891	PJ-04-01
184901	X	185857	TE-05	185891	PG-06-01
185268	X	185857	X	185891	X -
185308	X	185858	E-10-06	185893	PB-17-01
185345	X	185858	E-08-04	185894	PB-22-02
185354	TF-14	185858	B-05-04	185895	X
185354	X	185858	E-08-01	185895	TE-05-
185363	X	185858	C-19-02	185896	G-08-02
185495	X	185858	D-04-04	185896	G-08-04
185559	X	185858	C-19-04	185896	G-08-04
185589	X	185858	E-09-01		
185641	X	185858	E-10-04		

A.3 Coordinate Mappings

Listing A.2 Coordinate mappings from shelf identifiers to two-dimensional coordinates.


```
{
    "A":  [{"Row": 3 , "Column": 31, "Start": 1 , "End": 33}],
    "B":  [{"Row": 3 , "Column": 31, "Start": 1 , "End": 24}],
    "C":  [{"Row": 6 , "Column": 31, "Start": 43, "End": 1 }],
    "D":  [{"Row": 6 , "Column": 31, "Start": 43, "End": 1 }],
    "E":  [{"Row": 9 , "Column": 31, "Start": 1 , "End": 43}],
    "F":  [{"Row": 9 , "Column": 31, "Start": 1 , "End": 43}],
    "G":  [{"Row": 12, "Column": 31, "Start": 43, "End": 1 }],
    "H":  [{"Row": 12, "Column": 31, "Start": 43, "End": 1 }],
    "I":  [{"Row": 15, "Column": 31, "Start": 1 , "End": 43}],
    "J":  [{"Row": 15, "Column": 74, "Start": 1 , "End": 22}],
    "K":  [{"Row": 15, "Column": 74, "Start": 1 , "End": 22}],
    "L":  [{"Row": 18, "Column": 74, "Start": 22, "End": 1 }],
    "PA": [{"Row": 15, "Column": 31, "Start": 1 , "End": 24},
           {"Row": 15, "Column": 46, "Start": 28, "End": 42}],
    "PB": [{"Row": 18, "Column": 31, "Start": 42, "End": 28},
           {"Row": 18, "Column": 46, "Start": 24, "End": 1 }],
    "PC": [{"Row": 18, "Column": 31, "Start": 42, "End": 28},
           {"Row": 18, "Column": 46, "Start": 24, "End": 1 }],
    "PD": [{"Row": 21, "Column": 46, "Start": 4 , "End": 18}],
    "PE": [{"Row": 21, "Column": 46, "Start": 4 , "End": 18}],
    "PF": [{"Row": 24, "Column": 46, "Start": 18, "End": 4 }],
    "PG": [{"Row": 24, "Column": 46, "Start": 18, "End": 4 }],
    "PH": [{"Row": 27, "Column": 46, "Start": 4 , "End": 18}],
    "PI": [{"Row": 27, "Column": 46, "Start": 4 , "End": 18}],
    "PJ": [{"Row": 30, "Column": 46, "Start": 18, "End": 4 }],
    "PK": [{"Row": 30, "Column": 46, "Start": 18, "End": 1 }],
    "TA": [{"Row": 1 , "Column": 2 , "Start": 1 , "End": 27}],
    "TB": [{"Row": 4 , "Column": 2 , "Start": 1 , "End": 27}],
    "TC": [{"Row": 4 , "Column": 2 , "Start": 1 , "End": 27}],
    "TD": [{"Row": 7 , "Column": 2 , "Start": 27, "End": 1 }],
    "TE": [{"Row": 7 , "Column": 2 , "Start": 27, "End": 1 }],
    "TF": [{"Row": 10, "Column": 2 , "Start": 1 , "End": 27}],
    "TG": [{"Row": 10, "Column": 2 , "Start": 1 , "End": 27}],
    "TH": [{"Row": 13, "Column": 2 , "Start": 1 , "End": 27}],
    "X":  [{"Row": 26, "Column": 30, "Start": 1 , "End": 1 }]
}
```

B

Benchmark tables

B.1 Travelling Salesman Performance

Table B.1: Performance benchmarks of travelling salesman algorithms.

Nodes	TSP Algorithm	Distance	Time	Memory
4	Ant Colony	184	24.8 ms	31.6 MB
4	Christofides	184	294 us	13 KB
4	Held-Karp	184	183 us	14 KB
4	Linear Programming	184	2.3 ms	34 KB
4	Nearest Neighbor	195	240 us	5 KB
4	Simulated Annealing	184	4.4 ms	4.3 MB
8	Ant Colony	238	52.8 ms	51.4 MB
8	Christofides	276	353 us	30 KB
8	Held-Karp	238	2.7 ms	1.9 MB
8	Linear Programming	238	2.2 ms	55 KB
8	Nearest Neighbor	296	156 us	5 KB
8	Simulated Annealing	238	16.1 ms	10.2 MB
16	Ant Colony	339	141.2 ms	126.2 MB
16	Christofides	390	322 us	77 KB
16	Held-Karp	338	6:36 m	329.5 GB
16	Linear Programming	338	4.4 ms	152 KB
16	Nearest Neighbor	423	155 us	6 KB
16	Simulated Annealing	339	53.6 ms	21.5 MB
32	Ant Colony	431	687.1 ms	484.7 MB
32	Christofides	585	653 us	167 KB
32	Held-Karp	-	-	-
32	Linear Programming	415	24.6 ms	558 KB
32	Nearest Neighbor	494	173 us	9 KB
32	Simulated Annealing	422	188.8 ms	44.6 MB
64	Ant Colony	655	1.8 s	1.4 GB
64	Christofides	849	1.6 ms	435 KB
64	Held-Karp	-	-	-
64	Linear Programming	623	85.2 ms	2.1 MB
64	Nearest Neighbor	800	244 us	13 KB
64	Simulated Annealing	648	615.9 ms	90.1 MB

B.2 Batch Picking Performance

Table B.2: Performance benchmarks of batch picking algorithms.

Orders	BPP Algorithm	TSP Algorithm	Distance	Time	Memory
60	FIFO	Ant Colony	1646	2.0 s	2.2 GB
60	FIFO	Christofides	1906	2.3 ms	528 KB
60	FIFO	Linear Programming	1639	218.2 ms	2.5 MB
60	FIFO	Nearest Neighbor	2013	344 us	39 KB
60	FIFO	Simulated Annealing	1671	668.5 ms	176.7 MB
60	Genetic Algorithm	Ant Colony	1495*	15:6 m	1.0 TB
60	Genetic Algorithm	Christofides	1662	10.3 s	5.2 GB
60	Genetic Algorithm	Linear Programming	1382*	15:4 m	4.1 GB
60	Genetic Algorithm	Nearest Neighbor	1687	1.2 s	601.5 MB
60	Genetic Algorithm	Simulated Annealing	1545*	15:23 m	219.4 GB
60	Greedy Batching	Ant Colony	1262	3:28 m	216.7 GB
60	Greedy Batching	Christofides	1974	116.4 ms	80.2 MB
60	Greedy Batching	Linear Programming	1302	11.1 s	223.8 MB
60	Greedy Batching	Nearest Neighbor	1701	14.4 ms	7.1 MB
60	Greedy Batching	Simulated Annealing	1312	1:3 m	32.2 GB
60	Greedy Partition	Ant Colony	1493	32.2 s	29.7 GB
60	Greedy Partition	Christofides	1844	20.4 ms	14.0 MB
60	Greedy Partition	Linear Programming	1564	4.3 s	44.0 MB
60	Greedy Partition	Nearest Neighbor	1946	3.2 ms	1.3 MB
60	Greedy Partition	Simulated Annealing	1601	10.5 s	5.4 GB
60	Proximity	Ant Colony	1300	1.5 s	1.8 GB
60	Proximity	Christofides	1558	4.7 ms	3.2 MB
60	Proximity	Linear Programming	1287	89.7 ms	4.5 MB
60	Proximity	Nearest Neighbor	1620	3.10 ms	2.8 MB
60	Proximity	Simulated Annealing	1310	635.7 ms	156.1 MB
60	Simulated Annealing	Ant Colony	1587*	15:4 m	951.7 GB
60	Simulated Annealing	Christofides	1315	33.2 s	19.0 GB
60	Simulated Annealing	Linear Programming	1579*	15:0 m	3.9 GB
60	Simulated Annealing	Nearest Neighbor	1377	3.6 s	1.2 GB
60	Simulated Annealing	Simulated Annealing	1597*	15:1 m	241.4 GB
107	FIFO	Ant Colony	2391	1.8 s	1.8 GB
107	FIFO	Christofides	2827	2.2 ms	682 KB
107	FIFO	Linear Programming	2390	50.5 ms	1.7 MB
107	FIFO	Nearest Neighbor	3075	564 us	50 KB
107	FIFO	Simulated Annealing	2396	620.4 ms	226.6 MB
107	Genetic Algorithm	Ant Colony	2358*	16:25 m	1.0 TB
107	Genetic Algorithm	Christofides	2656	20.9 s	10.4 GB
107	Genetic Algorithm	Linear Programming	2259*	15:1 m	2.8 GB
107	Genetic Algorithm	Nearest Neighbor	2676	2.5 s	762.1 MB

Continued on next page

Table B.2 continued from previous page

Orders	BPP Algorithm	TSP Algorithm	Distance	Time	Memory
107	Genetic Algorithm	Simulated Annealing	2392*	15:23 m	325.0 GB
107	Greedy Batching	Ant Colony	1954	3:27 m	263.6 GB
107	Greedy Batching	Christofides	2516	223.0 ms	131.8 MB
107	Greedy Batching	Linear Programming	1944	15.3 s	153.5 MB
107	Greedy Batching	Nearest Neighbor	2438	23.4 ms	15.0 MB
107	Greedy Batching	Simulated Annealing	2015	49.5 s	42.6 GB
107	Greedy Partition	Ant Colony	2257	50.6 s	93.0 GB
107	Greedy Partition	Christofides	2654	46.3 ms	23.3 MB
107	Greedy Partition	Linear Programming	2012	2.5 s	69.4 MB
107	Greedy Partition	Nearest Neighbor	2798	6.9 ms	2.9 MB
107	Greedy Partition	Simulated Annealing	2244	18.3 s	11.0 GB
107	Proximity	Ant Colony	1994	1.8 s	1.5 GB
107	Proximity	Christofides	2367	9.2 ms	7.4 MB
107	Proximity	Linear Programming	1992	86.3 ms	8.7 MB
107	Proximity	Nearest Neighbor	2508	7.7 ms	6.9 MB
107	Proximity	Simulated Annealing	2015	567.2 ms	203.4 MB
107	Simulated Annealing	Ant Colony	2509*	15:5 m	940.1 GB
107	Simulated Annealing	Christofides	2235	41.2 s	27.9 GB
107	Simulated Annealing	Linear Programming	2482*	15:2 m	2.8 GB
107	Simulated Annealing	Nearest Neighbor	2266	4.1 s	1.8 GB
107	Simulated Annealing	Simulated Annealing	2552*	15:1 m	301.8 GB
223	FIFO	Ant Colony	5862	8.7 s	8.4 GB
223	FIFO	Christofides	6917	7.7 ms	2.1 MB
223	FIFO	Linear Programming	5831	511.8 ms	10.0 MB
223	FIFO	Nearest Neighbor	7148	1.6 ms	123 KB
223	FIFO	Simulated Annealing	5957	2.7 s	693.8 MB
223	Genetic Algorithm	Ant Colony	5709*	21:1 m	1.3 TB
223	Genetic Algorithm	Christofides	6505	1:33 m	38.9 GB
223	Genetic Algorithm	Linear Programming	5604*	16:6 m	4.4 GB
223	Genetic Algorithm	Nearest Neighbor	6623	4.1 s	1.1 GB
223	Genetic Algorithm	Simulated Annealing	5770*	15:20 m	228.4 GB
223	Greedy Batching	Ant Colony	-	15:12 m	1.0 TB
223	Greedy Batching	Christofides	5592	842.10 ms	674.8 MB
223	Greedy Batching	Linear Programming	4356	7:53 m	1.1 GB
223	Greedy Batching	Nearest Neighbor	5559	130.8 ms	72.6 MB
223	Greedy Batching	Simulated Annealing	4470	6:31 m	246.3 GB
223	Greedy Partition	Ant Colony	5154	4:52 m	364.9 GB
223	Greedy Partition	Christofides	5770	203.8 ms	117.3 MB
223	Greedy Partition	Linear Programming	4750	20.5 s	420.7 MB
223	Greedy Partition	Nearest Neighbor	6431	22.10 ms	14.9 MB
223	Greedy Partition	Simulated Annealing	5251	1:28 m	33.2 GB

Continued on next page

Table B.2 continued from previous page

Orders	BPP Algorithm	TSP Algorithm	Distance	Time	Memory
223	Proximity	Ant Colony	4702	5.4 s	6.3 GB
223	Proximity	Christofides	5684	53.4 ms	40.8 MB
223	Proximity	Linear Programming	4697	523.3 ms	47.2 MB
223	Proximity	Nearest Neighbor	5806	47.0 ms	39.3 MB
223	Proximity	Simulated Annealing	4803	2.4 s	619.0 MB
223	Simulated Annealing	Ant Colony	5842*	15:23 m	965.8 GB
223	Simulated Annealing	Christofides	5691	2:32 m	90.5 GB
223	Simulated Annealing	Linear Programming	59024*	15:7 m	4.0 GB
223	Simulated Annealing	Nearest Neighbor	5767	17.4 s	4.9 GB
223	Simulated Annealing	Simulated Annealing	5931*	15:4 m	232.4 GB
769	FIFO	Ant Colony	13278	13.2 s	13.6 GB
769	FIFO	Christofides	15320	7.6 ms	3.5 MB
769	FIFO	Linear Programming	13234	739.0 ms	14.7 MB
769	FIFO	Nearest Neighbor	15795	3.5 ms	306 KB
769	FIFO	Simulated Annealing	13425	4.6 s	1.5 GB
769	Genetic Algorithm	Ant Colony	15029*	23:34 m	1.4 TB
769	Genetic Algorithm	Christofides	16987	1:13 m	60.3 GB
769	Genetic Algorithm	Linear Programming	14954*	16:49 m	2.6 GB
769	Genetic Algorithm	Nearest Neighbor	17500	16.6 s	5.8 GB
769	Genetic Algorithm	Simulated Annealing	15167*	15:55 m	328.6 GB
769	Greedy Batching	Ant Colony	-	15:5 m	1.2 TB
769	Greedy Batching	Christofides	12382	7.0 s	4.6 GB
769	Greedy Batching	Linear Programming	-	15:35 m	171.1 MB
769	Greedy Batching	Nearest Neighbor	12766	841.6 ms	667.7 MB
769	Greedy Batching	Simulated Annealing	-	15:0 m	990.5 GB
769	Greedy Partition	Ant Colony	11921*	29:15 m	1.8 TB
769	Greedy Partition	Christofides	12045	930.6 ms	802.2 MB
769	Greedy Partition	Linear Programming	11383*	10:11 m	1.7 GB
769	Greedy Partition	Nearest Neighbor	13998	202.1 ms	142.3 MB
769	Greedy Partition	Simulated Annealing	12379	6:30 m	293.1 GB
769	Proximity	Ant Colony	10498	9.5 s	8.9 GB
769	Proximity	Christofides	11494	264.6 ms	353.2 MB
769	Proximity	Linear Programming	10478	1.0 s	363.2 MB
769	Proximity	Nearest Neighbor	11781	271.4 ms	351.0 MB
769	Proximity	Simulated Annealing	10633	3.1 s	1.3 GB
769	Simulated Annealing	Ant Colony	13967*	15:22 m	947.8 GB
769	Simulated Annealing	Christofides	15022	4:47 m	171.4 GB
769	Simulated Annealing	Linear Programming	14735*	15:15 m	2.6 GB
769	Simulated Annealing	Nearest Neighbor	15695	37.5 s	13.7 GB
769	Simulated Annealing	Simulated Annealing	14579*	15:8 m	296.8 GB