MILP FORMULATIONS FOR THE ORDER BATCHING PROBLEM IN LOW-LEVEL PICKER-TO-PART WAREHOUSE SYSTEMS

(SİPARİŞ GRUPLAMA PROBLEMİ İÇİN KARMA TAM SAYILI DOĞRUSAL PROGRAMLAMA GÖSTERİMLERİ)

by

Merve ÇAĞIRICI, B.S.

Thesis

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Date of Submission : Apr 8, 2014

Date of Defense Examination : Apr 11, 2014

Supervisor : Assoc. Prof. Dr. Temel ÖNCAN

Committee Members : Assoc. Prof. Dr. M. Ebru Angün

Asst. Prof. İbrahim Muter (BAU)

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor Assoc. Prof. Temel Öncan for his exceptional support, insight and encouragement during the process of preparing this thesis.

I would like to express my thanks to committee members, Assoc. Prof. Dr. M. Ebru Angün, and Asst. Prof. İbrahim Muter, for their comments and words of encouragements.

I would like to thank to my family and my friends for their encouragement, love and support.

Merve ÇAĞIRICI
Apr, 2014

Table of Contents

List of Symbols

ACO	: Ant Colony Algorithm
BB	: Branch and Bound
BP	: Branch and Price
CW	: Clark and Wright
CW(ii)	: Clark and Wright (ii)
FCFC	: First Come First Served
GA	: Genetic Algorithm
ILS	: Iterated Local Search
KDTP	: Karma Tam Sayılı Doğrusal Programlama
MILP	: Mixed Integer Linear Programming
MIP	: Mixed Integer Programming
OBP	: Order Batching Problem
PLMNE	: Programmation Linéaire Mixte en Nombres Entiers
PTC	: Problème de Traitement des Commandes
RBAS	: Ranked Based Ant System
SA	: Simulating Annealing
SGP	: Sipariş Gruplama Problemi
SPP	: Set Partitioning Problem
TS	: Tabu Search
VNS	: Variable Neighborhood Search

VRP : Vehicle Routing Problem

List of Figures

List of Tables

Abstract

Warehouse systems have many functions including receiving, storage, order picking and shipping. Order picking is the operation of retrieving products from their storage locations in order to satisfy customer requests. A picker performs retrieval operations according to a given pick list from the storage location to the input/output point. Order picking operations consist of order batching and order picker's routing. Order batching is the grouping of customer orders. Order picker's routing operation consists of determining the sequence of the order picking.

In this study, we consider the Order Batching Problem (OBP) which is shown to be NP-hard. Given both a list of customer orders and order picker's routing policy, the OBP deals with constructing batches of customer orders such that the total travel length of the pickers is minimized. Furthermore, the travel time has a substantial role in customer satisfaction since the shorter the travel time is; the sooner the requested items are ready for shipping.

To the best of our knowledge, there are no MILP formulations suggested for the OBPs with traversal and return routing policies in the literature. In the traversal routing policy, which is also known as the S-shape algorithm, the picker starts from the I/O point, visits every aisle where an item is required to be picked up and returns the I/O point. The picker enters an aisle from one end and leaves from the opposite end. In case the number of aisles that must be visited is odd then the picker enters and returns in the right-most aisle when it retrieves the item in that aisle. Note that, only in that case the picker does not necessarily traverses along the right-most aisle completely. In the return routing policy, a picker starts from the I/O point and proceeds along the front aisle. The picker enters each aisle where an item has to be picked up and travels along

this aisle as far as the deepest location where he must pick up an item, then returns along the aisle and leaves the aisle from the same end.

Basically, we introduce MILP formulations for the OBP and we also perform computational study to better expose the strength of the proposed MILP formulations. For the purpose, we compare the performance of the MILP formulations with the savings algorithm which is known to be one of the best performing construction heuristics for the OBP.

We have produced our test instances to carry out computational experiments. In the test instances that we have randomly generated, the number of orders is selected between 10 and 100 with an increment of 10. For each number of orders we have generated 10 instances which totally make 100 OBP test problems. The number of items for each order is randomly chosen between 2 and 10. The items are randomly assigned to locations. According to the capacity of the order picker we have three classes of randomly generated test instances. These are the Class I, Class II and the Class III instances, which assume picker capacities $Q = 24$, $Q = 36$, and $Q = 48$, respectively.

The computational results show the usefulness of the MILP and savings heuristic for the OBP. According to our computational experiments, comparing both methods, savings heuristic yields significantly better results in reasonable CPU times. Thus, considering the trade-off between accuracy and efficiency, savings heuristic gives a reasonable solution approach in terms of CPU times. Savings heuristics clearly outperform MILP approach.

From the experimental results, we observe that the proposed formulations yield quite good upper bounds and hence they can be used for benchmarking purposes. These MILP formulations can also be used as benchmarks for other studies which propose heuristic and meta-heuristics for the OBP. Moreover, there is also room to devise better MILP formulations for the OBP.

As a further research avenue, MILP formulations for the OBP with largest gap and composite policies can be developed. Besides, developing a branch and bound algorithm exploiting the structure of the problem would be an interesting work for further developments on this problem. Branch and cut algorithms can be also developed by suggesting valid inequalities based on the proposed MILP formulations.

Résumé

Les systèmes de gestion des stocks ont plusieurs fonctions incluant la réception, le stockage, la préparation des commandes et l'expédition. La préparation des commandes est une opération de récupération des produits de leurs emplacements de stockage afin de satisfaire les demandes des clients. Un préparateur de commandes effectue les opérations de prélèvements conformément à une liste de sélection donnée du lieu de stockage au point d'entrée/sortie (E/S). L'opération de préparation des commandes consiste à prélever et rassembler les articles de la commande avant son expédition. Le traitement par lots des commandes est le regroupement des commandes des clients. L'opération de routage des commandes consiste à déterminer le séquençage de la préparation de ces commandes.

Dans cette étude nous considérons le Problème de Traitement des Commandes (PTC) qui est NP_dur. Disposant à la fois d'une liste des commandes des clients et d'une politique d'acheminement, PTC traite de la construction de lots de commandes des clients de telle sorte à ce que la longueur totale du voyage du préparateur de commandes est minimisée. De plus, le temps de voyage a un rôle important dans la satisfaction de la clientèle puisque plus le temps de voyage est court, plus tôt les articles demandés sont prêts à être expédié.

A notre connaissance, il n'y a pas de formulations de Programmation Linéaire Mixte en Nombres Entiers (PLMNE) proposées pour les PTC avec la traversée et les politiques de retour de routage dans la littérature. Dans la politique de routage de traversée, qui est également connu comme l'algorithme S-forme, le préparateur de commandes commence à partir du point d'E/S, des visites chaque couloir où un élément doit être ramassé et retourne au point d'E/S. Le préparateur de commandes entre dans un couloir d'une extrémité et sort de l'autre l'extrémité. Dans le cas où le nombre de couloirs qui

doivent être visités est impair alors le préparateur de commandes entre et retourne dans la plus droite couloir quand il récupère l'élément dans cette allée. Notez que, dans ce cas seulement le préparateur de commandes ne traverse pas nécessairement le long de la plus droite couloir complètement. Dans la politique de retour de routage, un préparateur de commandes commence au point d'E/S et produit le long du couloir avant. Le préparateur de commandes entre chaque couloir où un article doit être ramassé et se déplace le long cette couloir aussi loin que l'endroit le plus profond où il doit ramasser un objet, puis retourne le long du couloir et quitte cet couloir de la même extrémité.

Fondamentalement, nous introduisons les formulations PLMNE pour PTC et nous effectuons également une étude de calcul afin de mieux exposer la force des formulations PLMNE proposées. Pour cela, nous comparons les performances des formulations PLMNE avec l'algorithme de gains qui est connu pour être l'un des heuristiques de construction les plus performants pour PTC.

Nous avons produit les problèmes de teste pour réaliser des expériences de calcul. Les problèmes de teste ont été générés de manière aléatoire, le nombre de commandes est choisi entre 10 et 100 avec un incrément de 10. Pour chaque nombre de commandes nous avons généré 10 problèmes qui forment au total 100 problèmes de test. Le nombre d'articles pour chaque commande est choisi aléatoirement entre 2 et 10. Les éléments sont aléatoirement répartis dans les emplacements. Selon la capacité du préparateur de commande, nous avons trois catégories de cas de test générés aléatoirement. Il s'agit des instances de la classe I, de la classe II et la classe III, qui admettent, respectivement, des capacités de prélèvement de Q = 24, Q = 36 et Q = 48.

Les résultats des calculs montrent l'utilité des formulations de PLMNE pour PTC. Selon nos expériences de calcul, en comparant les deux méthodes, les rendements de l'algorithme de gains donnent de meilleurs résultats du point de vue des temps CPU. Ainsi, en considérant le compromis entre la précision et l'efficacité, l'algorithme de gains offre une solution raisonnable dans une période de temps admissible. Les algorithmes de gains sont donc nettement supérieurs à l'approche PLMNE.

A partir des résultats expérimentaux, nous observons que les formulations proposées offrent des bornes supérieures acceptables et par conséquent, elles peuvent être utilisées

à des fins de benchmarking. Ces formulations de PLMNE peuvent également être utilisées comme points de référence pour d'autres études qui proposent les heuristiques et les méta-heuristiques pour PTC. En outre, il est également possible de mettre au point de meilleures formulations de PLMNE pour PTC.

Comme autre piste de recherche, les formulations de PLMNE pour PTC peuvent être développées en considérant d'outres stratégies de routage. En outre, l'élaboration d'une algorithme de Branch and Bound qui exploite la structure du problème serait un travail intéressant. En plus, les méthodes de Branch and Cut peuvent également être aussi développés en suggérant des inégalités valides sur la base des formulations de PLMNE proposées.

Özet

Günümüz depo sistemleri, ürünlerin depolara alınması, saklanması, siparişlerin toplanması ve gönderilmesi gibi birçok operasyonel işi gerekli kılar. Sipariş toplama; sipariş listesinin tamamlanması amacıyla ürünlerin hücrelerinden toplanma işlemidir. Toplayıcı kişi oluşan sipariş listesine göre ürünleri hücrelerden çıkış kapısına getirir. Sipariş gruplama ise depoya gelen siparişleri gruplayarak bir arada toplama işlemidir.

Bu çalışmada parça toplayıcıların depodan siparişleri çektiği ortamda, Sipariş Gruplama Problemi (SGP) ele alınmıştır. Günümüz depo sistemlerinde karşılaşılan bu problem NP-zor olarak bilinmektedir. SGP, birbirlerine benzer siparişleri gruplayarak, belirli rotalama stratejileri altında parça toplayıcıların kat ettiği toplam mesafeyi en küçüklemeyi amaçlamaktadır. Doğal olarak, kat edilen mesafenin azalması, siparişin teslim sürecini hızlandırdığı için müşteri memnuniyetinin artmasını sağlar.

Bu çalışmada toplayıcı olarak işçilerin çalıştığı ve toplayıcıların parçaları almaya gittikleri SGP için Karma Tam Sayılı Doğrusal Programlama (KDTP) gösterimleri geliştirilmiştir. S rotası olarak da bilinmekte olan geçişli stratejide parça toplayıcı siparişin olduğu her koridora girer ve ters taraftan koridoru terk ederek başlangıç noktasına döner. Ziyaret edilecek koridor sayısının tek olması durumunda, parça toplayıcı son (en sağdaki) koridorda en uzak yerde bulunan parçayı alarak başlangıç noktasına geri döner. Ziyaret edilecek koridor sayısının tek olduğu durumlarda, parça toplayıcı son koridoru tamamen geçmek zorunda değildir. Dönüşlü stratejide, parça toplayıcı ön koridorda hareket ederek sipariş olan koridora girer ve en uzaktaki parçayı alarak koridora girdiği taraftan koridoru terk eder.

Bu çalışmada, SGP için KTDP gösterimleri geliştirdik ve başarılarını sergilemek için bilgisayısal deneyler yaptık. Bu amaçla, SGP için geliştirilen KTDP gösterimlerini çözüm kurucu sezgisel algoritmalarından biri olan kazanç algoritması ile karşılaştırdık.

Bilgisayısal çalışmalarımızı gerçekleştirmek için test örnekleri ürettik. Bu örnekler rasgele üretilmiştir. Bu örneklerin sipariş sayıları 10 ile 100 arasında 10'ar artarak değişmektedir. Her sipariş numarası için 10 örnek ürettik. Toplamda SGP için 100 test problem oluşturduk. Her sipariş için parça sayısı 2 ile 10 arasında rasgele seçildi. Sipariş toplayıcının kapasitesine göre 3 farklı problem sınıfı elde ettik. Birinci sınıf problemler 24 parça kapasiteli, ikinci sınıf problemler 36 parça kapasiteli ve üçüncü sınıf problemler 48 parça kapasiteli olarak üretildi.

Bilgisayısal çalışmalar KTDP ve kazanç sezgiselinin SGP için olumlu sonuçlar verdiğini gösteriyor. Uyguladığımız sayısal çalışmaların sonuçlarına göre, her iki metodu karşılaştırdığımızda kazanç sezgiselinin daha kabul edilebilir bir sürede sonuç verdiğini söyleyebiliriz. Etkinlik ve çözüm niteliği unsurlarını düşündüğümüzde kazanç sezgiselinin oldukça başarılı olduğunu söyleyebiliriz. Ancak, KTDP gösterimleri küçük boyutlu problemler için en iyi çözümü bulmaktadır. Bu nedenle, KTDP gösterimleri, sezgisel ve meta sezgisel çalışmalar için karşılaştırmalı değerlendirme amacıyla kullanılabilir.

Gelecek araştırma konusu SGP'ine özel, dal sınır ve dal kesme yöntemleri geliştirilebilir.

1. INTRODUCTION

Warehouse systems have several functions including receiving, storage, order picking and shipping. Among these functions, order picking is known to be the most labor intensive and costly function (Drury, 1988). Storage is the place where the items are stocked and retrieved from. Order picking is the process of retrieving products from their storage locations in order to satisfy customer requests. Order picking costs are estimated to be as much as of 65 % of total warehouse operating expenses (Drury, 1988, Coyle et al., 1996, Tompkins et al., 2003).

A picker performs retrieval operations according to a given pick list from the storage location to the input/output point. Order picking operations consist of order batching and order picker's routing operations. Order batching is the grouping of customer orders. Order picker's routing consists of the sequence of the order picking.

In this study, we consider the Order Batching Problem (OBP) which is shown to be NP-hard by Gademann and van de Velde (2005). Given both a list of customer orders and order picking routing policy, the OBP deals with constructing batches of customer orders such that the total travel length of the pickers is minimized.

Broadly speaking, order-picking systems can be grouped in two categories according to the material handling equipment used: picker-to-parts systems and parts-to-picker systems. In picker-to-parts systems, order pickers travel along the warehouse and retrieve the items requested. On the other hand, in parts-to-picker systems the requested items are handled and transported by automatic storage and retrieval systems (AS/RSs) to order pickers (Wäscher, 2004, De Koster et al., 2007). Particularly, there exist two types of picker-to-parts systems: low-level and high-level picking systems. In low-level

picking systems, the picker travels along the aisles in order to pick the requested items from the storage bins or racks. In high-level systems, the pickers drive a truck or a crane to reach the pick locations.

To be specific, we focus on the OBP in a low level picker to part warehouse. For the sake of clarity, given customer orders and order picking routing policy our problem is to find groups of customer orders such that the total travel length of all pickers is minimum. The aisles are numbered in increasing order from the left hand side to the right hand side. The warehouse is considered rectangular and the input/output point is assumed to be situated in the leftmost position in front of the first aisle of the front aisle. Inter-aisle distance of the parallel aisles is fixed and symbolized with w. The distance between the front aisle and the back aisle is denoted by L. Starting from the entrance of the aisle a picker is responsible for the retrieval of all items in the batch.

We particularly adress low-level picker-to-parts picking systems employing human pickers. De Koster et al. (2007) have claimed that 80 % of all order-picking systems in Western Europe are of this type. Moreover, the research of European Logistics Association indicates the significance of warehousing, which is 25% of total logistic cost (European Logistics Association and A. T. Kearney, 2004). In addition, order picking is 50% of the total warehousing operation costs (Frazelle, 2002).

In order picking systems, the service level basically consists of order delivery time, order integrity and accuracy (De Koster et al., 2007). Order delivery time is closely related with the travel time of the picker. As pointed out by Tompkins et al. (2003) almost half of the order picker time is wasted while travelling. Despite several other activities other than travelling requires a considerable amount of the picker's time (Hall, 1993, Petersen, 1997, Roodbergen and De Koster, 2001), the time devoted to the travel activity is seen as the most time consuming activity (De Koster et al., 2007). Furthermore, the travel time has a substantial role in customer satisfaction since the shorter the travel time is; the sooner the requested items are ready for shipping. Hence, among several objective functions that can be taken into consideration such as the minimization of order throughput, maximization of item accessibility, maximization of

labor use; minimization of pickers' total travel distance is the most widely considered one (De Koster et al., 2007).

To the best of our knowledge, there are no MILP formulations suggested for the OBPs with traversal and return routing policies in the literature. This is the basic motivation of this study. First of all, we introduce two MILP formulations for the OBP and we also perform a computational study to better expose the strength of the proposed MILP formulations. The MILP formulations suggested in this study have been discussed in detail by Çağırıcı and Öncan (2013). To this end, we compare the performance of the MILP formulations with the savings algorithm which is known to be one of the best performing construction heuristics for the OBP (De Koster et al. 1999). The rest of this work is organized as follows. Section 2 presents a literature survey on the OBP. Then, in Section 3 devises two MILP formulations for the OBP with each of them addressing a different routing strategy (e.g. traversal and return routing strategies). This is followed by the computational results in Section 4. Finally, concluding remarks are given in Section 5.

2. LITERATURE SURVEY

In the literature several order picking routing policies have been introduced. These are traversal (Goetschalckx and Ratliff, 1998), return, midpoint, largest gap (Hall, 1993), composite and optimal (Ratliff and Rosenthal, 1983) routing policies. Generally speaking, Petersen (1997) has asserted that the routing policies range from simple to more complex in that order. Namely, traversal, return and midpoint strategies are simpler than the largest gap, composite and optimal routing policies. According to the experiments by Petersen (1997), the optimal routing strategy is the winner at the expense of its disadvantages such as discernible pattern and the routes with backtracks. However, the author states also that the heuristic routing policies (traversal, return, midpoint, largest gap and composite routing policies) are easy to use and they are more apt to construct similar routes. Besides, Petersen (1997) states that composite and largest gap policies are the second best choices after the optimal routing strategy. Note that, complex routing policies may yield congestion problems when several pickers share long, narrow and two way aisles. Furthermore simple routing policies may arise to be useful especially for complex order picking systems with many pickers.

In their early study, De Koster et al. (1999) have reported a comparative computational study of several OBP heuristics. Among them the authors have highlighted that the seed algorithm and a variant of the Clarke and Wright (1964) heuristic arise to be the most promising according to both traversal and largest gap policies. Later on, Hwang and Kim (2005) have also considered several OBP heuristics to perform an in-depth computational analysis. Furthermore, the authors have designed an efficient OBP heuristic based on cluster analysis. They have tested their algorithm on randomly generated 300 instances. They have observed that when the number order size exceeds

20 their heuristic is the winner. Further, they have noted that, for small size instances the seed algorithm yields an outstanding performance.

Now we will introduce a presentation of several routing policies considered for the OBP. Next we will briefly outline the heuristic procedure for the OBP and then, we will summarize meta-heuristic procedures devised for the OBP.

2.1. Routing Policies for the OBP

Recall that, in this study we concentrate only on the OBP considering traversal and return routing policies. First of all, we introduce the definition of these routing policies. In the traversal routing policy, which is also known as the S-shape algorithm, the picker starts from the I/O point, visits every aisle where an item is required to be picked up and returns the I/O point. The picker enters an aisle from one end and leaves from the opposite end. In case the number of aisles that must be visited is odd then the picker enters and returns in the right-most aisle when it retrieves the item in that aisle. Note that, only in that case the picker does not necessarily traverses along the right-most aisle completely. In the return routing policy, a picker starts from the I/O point and proceeds along the front aisle. The picker enters each aisle where an item has to be picked up and travels along this aisle as far as the deepest location where he must pick up an item, then returns along the aisle and leaves the aisle from the same end. The midpoint policy divides the warehouse in two sections by drawing horizontal line in the middle of aisles. A picker which leaves input/output point first crosses the first aisle entirely then, in the next aisle the picker travels towards to the midpoint and returns back whenever he picks up an item. In the largest gap policy, the picker enters each aisle which contains items from the front and back sides such that the maximum distance between two neighbor items is not crossed. In this policy, the picker travels along the leftmost aisle and the rightmost aisle which he must pick up an item. The composite policy is a combination of traversal and return routing policies. The picker may travel along the aisle completely or the picker can leave the aisle from same end. This policy is considered by means of a dynamic programming approach.

For the sake of clearness, we present with Figure 2.1. an illustration of the warehouse layout that we focus on here. In Figure 2.1., we consider three orders, i.e. order 1, order 2 and order 3 which include 4, 3 and 5 items, respectively. Note that, the locations of these items are indicated with order numbers. The shape of the warehouse is assumed to be rectangular with parallel storage. The warehouse totally incorporates 10 parallel aisles. The input output (I/O) point located in the left-most corner of the front aisle. The picking area has the capacity to store 200 items. Each order must be assigned into a batch. Each order consists of at least one item. The locations of items are known a priori. The total number of items which belong to the orders assigned to a batch should not exceed the capacity of the picker responsible of that batch. The quantity to be picked up of each item is assumed to be one unit. For the OBP test problems, we assume that the horizontal distance within stocking aisles is negligible and the picker does not need additional time for entering and leaving the aisles. In Figure 2.2., Figure 2.3., Figure 2.4. and Figure 2.5. we present the routes of the picker serving all of three orders considering traversal, return, midpoint and largest gap policies, respectively.

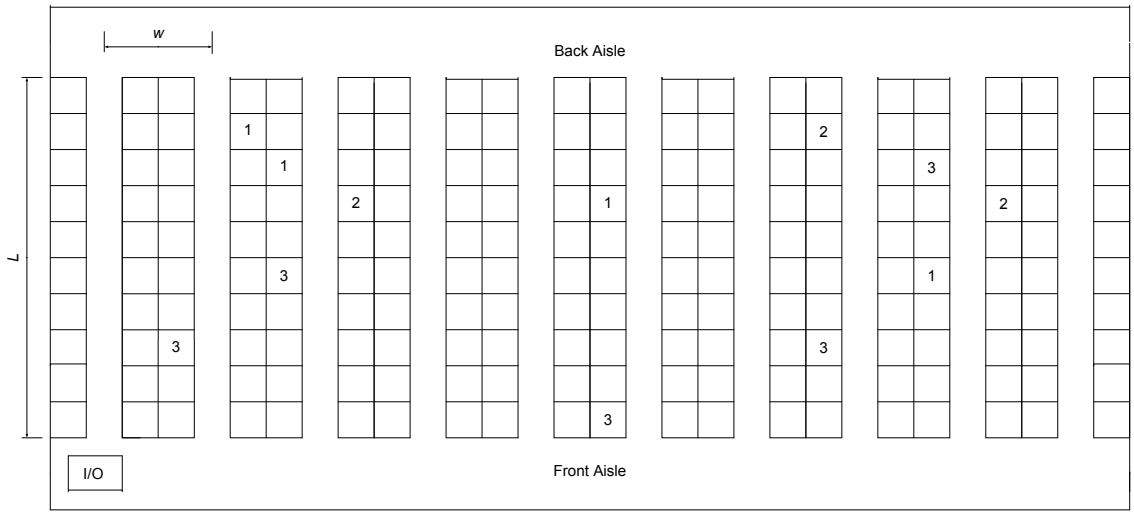

Figure 2.1. Layout of a rectangular warehouse

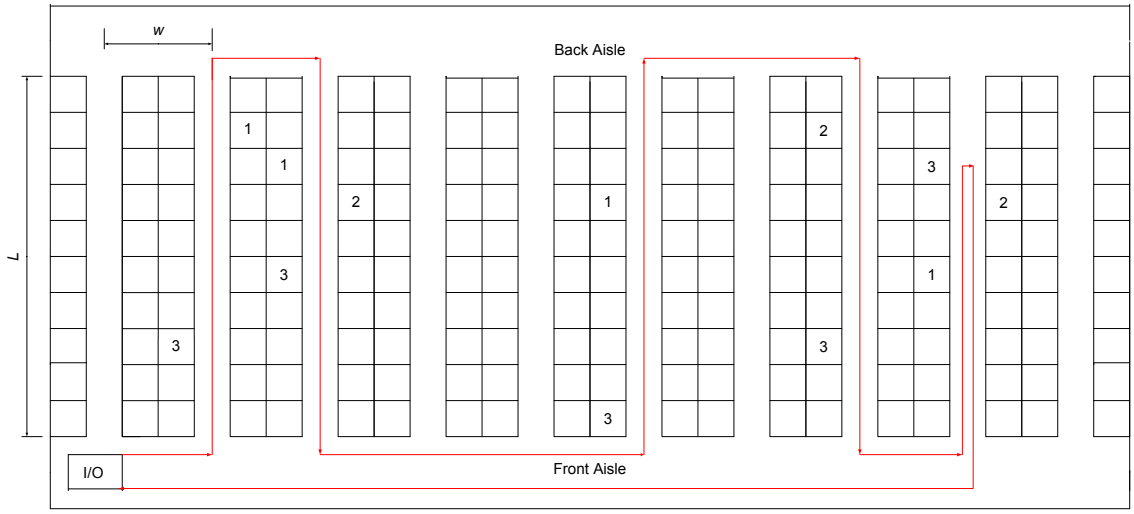

Figure 2.2. Traversal Routing Policy

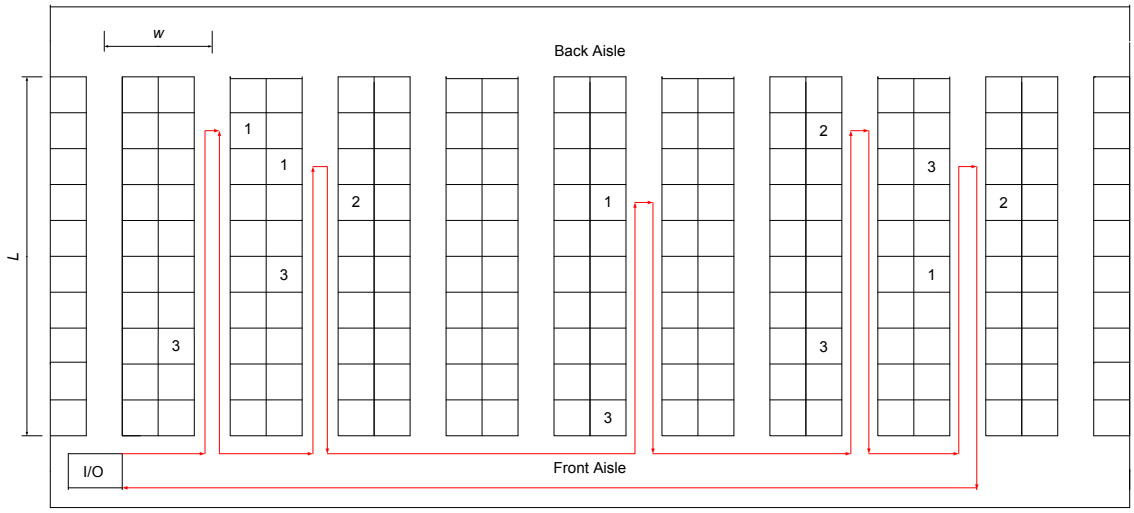

Figure 2.3. Return Routing Policy

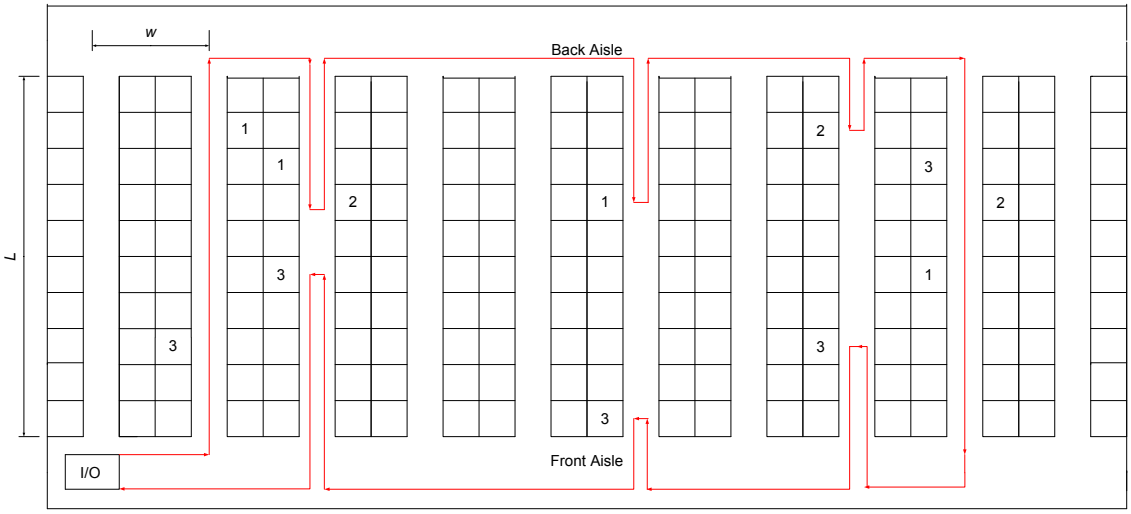

Figure 2.4. Midpoint Routing Policy

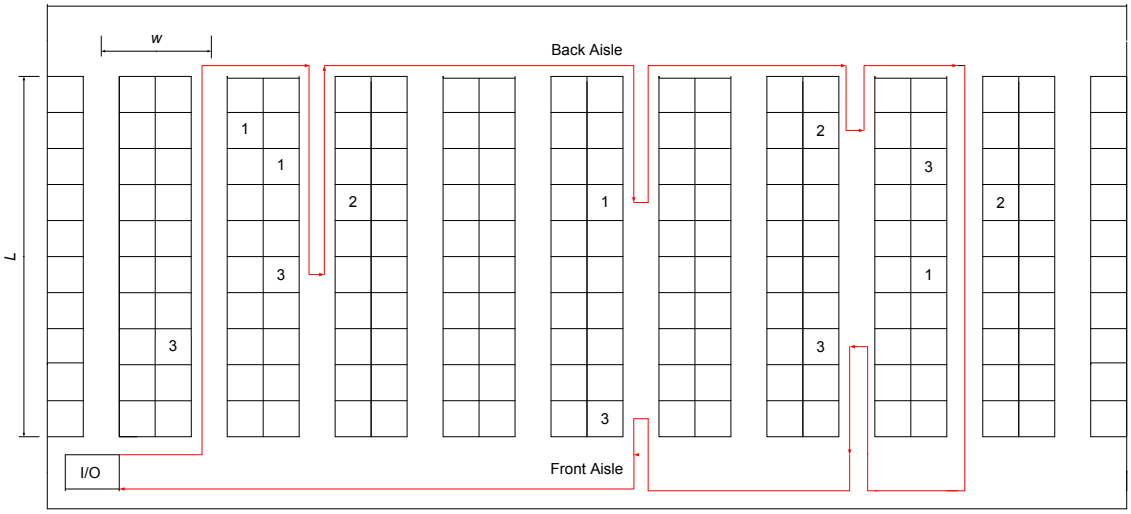

Figure 2.5. Largest Gap Routing Policy

2.2. Heuristic Solution Procedures for the OBP

To the best of our knowledge, there are only very few studies addressing the exact solution of the OBP and no MILP formulation of the OBP has ever been proposed. Gademann et al. (2001) have designed a Branch and Bound (BB) algorithm for the OBP with the objective of minimizing the maximum travel time of the pickers. The OBP has been formulated as a Set Partitioning Problem (SPP) by Gademann and van de Velde (2005) where the authors have devised a Branch and Price (BP) algorithm and have reported the optimum solution of problems with up to 32 customer orders. For a revised and simplified version of OBP considering the traversal routing policy, Bozer and Kile (2008) have proposed a Mixed Integer Programming (MIP) formulation however they could solve small size instances (up to 25 customers) to optimality. The revised version of the OBP addressed by Bozer and Kile (2008) is quite different than the original OBP considering the traversal routing policy. The authors have addressed only the traversal routing policy when the number of traversals is even. Their formulation does not compromise the case when the number of traversals is odd and the picker returns back in the last aisle whenever he retrieves the last requested item. Recently, Henn and Wäscher, (2012) have claimed that after generating all possible feasible batches they could solve OBP instances with up to 40 customer orders by solving the SPP formulation by Gademann and van de Velde (2005). Note that, this approach may become quite costly since the generation of all possible feasible batches is a painstaking task.

Several heuristic algorithms have also been developed for the OBP. Among them we can mention the first fit envelope based batching heuristic by Ruben and Jacobs (1999), the priority rule based algorithms (Gibson and Sharp, 1992), the seed algorithms (Elsayed, 1981, Ho et al., 2008, Ho and Tseng, 2006) and the savings algorithms (Clarke and Wright, 1964). Hwang and Kim (2005) proposed an order batching algorithm based on cluster analysis. Data mining approaches have been developed by Chen and Wu (2005a) and Chen et al. (2005b). de Koster et al. (1999) have computationally tested several construction heuristic procedures and the best

performing heuristic is the seed algorithm and the Clarke and Wright savings algorithm(CW).

In their comparative analysis work, de Koster et al. (1999) indicate that seed algorithms include two different steps. At first, one order which is not assigned to a batch is selected as a seed (or initial) order. Subsequently, customer orders which are not assigned to a batch are added to the seed order without violating capacity limitation. There are many seed selection rules such as a random customer order (Gibson and Sharp, 1992) or the customer order which has the farthest item or the customer order which take longest time for picking between the others etc. Also there are different rules for addition rules such as the minimization of the sum item distances based on the seed order (Gibson and Sharp, 1992) or minimization of additional aisles which have to be entered (Rosenwein, 1996), etc…

Savings algorithm is inspired from the Clarke and Wright (1964) algorithm which is originally developed for the vehicle routing problem (VRP). This algorithm compares total travel length which the picker collects the orders *i* and *j* separately with the total travel length such that the picker collects the orders in the same route. Thus, $S_{ij} = t_i + t_j - t_{ij}$. S_{ij} indicates the difference between two situations. Here $t_i(t_j)$ stands for the required travel distance to pick up items in order $i(j)$ and t_{ij} stands for the distance required to collect items in both orders i and j. Briefly, savings algorithm consists of the following steps.

Step 1. *(Initialization)* Each order is assigned to a batch

Step 2. Compute saving costs S_{ij} of combining batch *i* and batch *j*.

$S_{ij} = t_i + t_j - t_{ij}$ where t_{ij} is the cost of serving batch i and batch j with a single picker.

$t_i(t_j)$ is the cost of serving batch $i(j)$ using a single picker.

Step 3. Considering the capacity restriction merge two batches with the highest S_{ij} is selected.

Step 4. Update S_{ij} values

Step 5. Check for the stopping rule. We perform CW algorithm until no further merge operation is possible due to either the capacity constraints or because we cannot find any batch pair with positive saving value S_{ij}.

Step 6. Go to Step 3.

2.3. Meta-Heuristic Algorithms for the OBP

As meta-heuristic algorithms designed for the OBP, we can mention the Genetic Algorithms (GAs) proposed by Hsu et al. (2005), Tabu Search (TS) algorithm by Henn and Wäscher (2012) and the Variable Neighborhood Search (VNS) algorithm by Albareda-Sambola et al. (2009). Tsai et al. (2008) have simultaneously addressed the OBP and the routing problem considering both travel distance and order due time. The authors have proposed a GA for this combined problem.

The local search heuristics basically consider neighbor solutions and try to find a new solution with a better objective function value. To search neighbor solution, local search heuristics apply some simple operators. In many commentarial optimization problems MOVE and SWAP operators are widely used as straightforward neighborhood search schemes. MOVE operator selects an item from its location and inserts it into another location. SWAP operator exchanges the location of two different items. The local search heuristics perform neighborhood search operations until no further improvement in the objective function value is possible. Then the local search heuristic outputs the best solution found during the neighborhood search phase. For the sake of clarity we give the steps of the local search heuristics.

Step 1.(*Initialization*) Generate initial solution S and compute the objective function value

Step 2. Obtain a neighbor S' of the solution S and compute objective function value.

Step 3. If the neighbor solution have a smaller objective value than the current objective function, then S' replaces S as the incumbent solution.

Step 4. Check for stopping criteria. Go to *Step* 2.

One inconvenience of the local search heuristic is that the solution output may have not a desired accuracy. However local search heuristics are widely used by many researchers because of their ease of implementation and short computational time requirement. For the order batching problem Gademann and van de Velde (2005) proposed the first local search heuristic. The initial solution is obtained by applying FCFS method as a neighborhood search approach they have adopted SWAP operations. In their local search approach, whenever a local minimum is obtained, this solution is modified by changing locations of three customer orders from their assigned batches randomly, and this step is so-called the perturbation phase. Later on, Henn et al.(2009) have also devised local search algorithm which consist of two phases: perturbation and local search phases. Different than Gademann and van de Velde (2005) the authors have employed SWAP and MOVE operators in the local search phase. In the perturbation phase two randomly selected items have been exchanged without harming the feasibility of the solution.

In addition to, several heuristic approaches, various meta-heuristic algorithms have also been proposed. Two recent ones are Iterated Local Search (ILS) and the Rank-Based Ant System (RBAS) (Henn et al., 2009). The ILS includes two phases: a local search phase and a perturbation phase. In the first phase, a feasible solution is improved considering the objective function addressed (e.g. maximization of picker usage and/or minimization of total route length travelled by the pickers). The ILS tries to obtain a local optimum with an improved objective function value. In this phase, SWAP and/or SHIFT operators are used for that purpose. SWAP operation tries to exchange the assignment of two orders which belong to two different batches. On the other hand SHIFT is used to move an order from its batch into another batch. These two operators are performed consecutively until no further improvement has been observed in the objective function. In the second phase, namely perturbation phase, two batches are randomly chosen. Then, the items in these batches, which are also randomly fixed, are exchanged without harming the feasibility of the solution.

For the OBP, initial solution can be obtained by FCFS. After assigning the customer orders into the batches, local search operators, such as the SHIFT and/or SHIFT, have been applied with the hope to improve the objective function value.

Ant Colony Optimization Algorithms (ACOAs), as well as the RBAS, are inspired from a natural system, namely an ant colony. Generally speaking, the ACOAs minimize the length of route of an ant colony. Particularly, the RBAS is based on the savings algorithm which is quite often used for these types of problems. At the start of algorithm, each order constitutes a single batch. Then batches are constructed without violating capacity limitation and considering the saving value s_{kl} and a pheromone intensity t_{kl}. The batches are constructed taking into consideration both saving values and intensities. Then whenever a feasible solution is constructed a local search procedure is applied in order to improve the solution quality. In the RBAS each feasible solution corresponds to an ant and several ants have been generated during the run of the algorithm. Similar to the natural selection process a percentage of ants are removed from the system. Furthermore, each solution value of the RBAS corresponds to a pheromone. Those pheromones can be evaporated during the search process, as well. The solutions obtained during the algorithm are ordered according to their pheromones. A fraction of them will be rewarded by increasing their pheromone intensity. Then the solutions with high pheromone intensities are eligible for the subsequent iteration of the algorithm. For the details of the algorithm we refer to the study by Henn et al. (2009)

The meta-heuristics can obtain up to 20 % improvement in total travel distance in comparison with the FCFS solution. Also CW (ii) + Local Search can obtain more than 17 % improvement approximately compared to FCFS solution. Henn et al. (2009) indicate that meta-heuristics such as RBAS and ILS gives better solutions than CW (ii) and Local Search in acceptable CPU time.

Genetic Algorithms (GAs) have been devised from the biological evolution processes. Solutions in GAs are represented by chromosomes and each chromosome is constructed by a sequence of genes. A GAs tries to find the best sequence of genes by moving from one solution (chromosome) to another solution. This process is performed by

exchanging the genes of a chromosome. For that purpose various operators such as reproduction, crossover and mutation are applied during the run of the GA. The reproduction operator generates a new poll of chromosomes from a previous solution set. In other words reproduction operator selects the best chromosome to the next generation. The crossover operator randomly selects two different genes from two different chromosomes and exchanges their locations. Mutation operator modifies the gene sequences by reallocating them within chromosome. Each solution is represented with an encoding scheme which serves to translate a solution into a string of genes from a chromosome.

During the run of the GA a suitable encoding of the solution is crucial for the performance of the algorithm. Namely, a good representation helps to clearly define crossover, reproduction and mutation operators. Initial population also affects the performance of the genetic algorithm. In the literature the initial population is generally constructed by a set of randomly generated chromosomes. For all we know, two GA have been proposed for the OBP. For the details we refer to the studies by Hsu et al. (2005) and Öncan (2013).

Tabu search (TS) is a local search based algorithm which has been suggested by Glover (1986). Tabu search keeps a tabu list to in order to prevent cycling during the local search phase. The tabu list is used to memorize moves which have been applied in previous iterations.

The algorithm starts from an initial solution and each iteration moves from the current solution to the best one in a subset of its neighborhood. These moves are performed even if they cause worse solutions. In order to avoid cycling solutions with some attributes are declared tabu or forbidden for a fixed number of iterations, namely tabu tenure. The tabu search algorithm stops when a priori defined rule satisfied. The tabu search algorithm keeps track of short term and long term memories during its run. Henn and Wäscher (2012) obtained initial solution by applying FCFS and C&W(ii) for the order batching problem. Neighbor solutions can be reached with only SWAP moves, SHIFT moves and SWAP or SHIFT moves.

Simulating Annealing (SA) has first been proposed inspired from an analogy between the annealing of solids and the problem of solving the optimization problem. The SA algorithm avoids to get stuck into local optimum by employing random selection and acceptance strategy. The random acceptance strategy allows worse solutions with certain probability which is controlled by a temperature parameter. Furthermore the temperature parameter is updated according to a cooling schedule. Recently, an SA algorithm for OBP has been proposed by Matusiak et al. (2013).

The variable neighborhood search (VNS) heuristic is first designed by Mladenovic (1995) and Mladenovic and Hansen (1997) for solving optimization problems inspired by the idea of systematically modifying the neighborhood setting to escape from local optimal. A recent VNS application for the OBP has been suggested by Albareda-Sambola et al. (2009).

3. MILP FORMULATIONS FOR THE ORDER BATCHING PROBLEM

In this section, we propose two MILP formulations for the OBP considering traversal and return routing strategies. The proposed MILP formulations are inspired from the grouping models used in Islam and Sarker (2000) and Hwang and Kim (2005).

In these models, we use the following notation. Given a set of orders, $i = 1, \dots, n$ and aisles $k = 1, \dots, K$; let L denotes the vertical length of the aisle, w stands for the width between aisles, Q indicates the capacity of picker and m_i be the number of items in order i. We assume homogeneous capacity for all pickers. Let x_{ij} equal to 1 if and only if order i is assigned to batch j. Moreover, note that $x_{jj} = 1$ holds if and only if batch j is represented by order j.

Table 3.1. Notation for the Basic Formulation

$i = 1, \dots, n$	The set of orders
$k = 1, \dots, K$	The set of aisles
L	The vertical length of aisle
w	The width between aisles
Q	The capacity of picker
m_i	The number of items in order i
x_{ij}	1 if order i is assigned to batch j, and 0 otherwise
x_{jj}	1 if order j represents batch j, and 0 otherwise

Now we present the constraint set developed by Hwang and Kim (2005) to analyze several similarity measures for the batching of customer orders. The authors have inspired from an early study by Islam and Sarker (2000) who have devised Binary Integer Programming formulation for the machine-cell or part-families grouping problem. The basic formulation is as follows:

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \text{for } i = 1, \dots, n, \tag{1}$$

$$x_{ij} \leq x_{jj} \qquad \text{for } i, j = 1, \dots, n, \tag{2}$$

$$\sum_{i=1}^{n} m_i x_{ij} \leq Q \qquad \text{for } j = 1, \dots, n. \tag{3}$$

Here, constraints (1) assert that each order should be assigned to exactly one batch. Constraints (2) guarantee that order i is assigned to batch j then batch j is represented by order j. Constraints (3) enforce that the items of all orders assigned to batch j satisfy the capacity of the picker.

3.1. Traversal Policy

Now we give the MILP formulation designed for the OBP considering the traversal policy. For that purpose we define the following parameters and decision variables. Parameter d_{ik} stands for the vertical distance, that a picker serving order i should travel, through the aisle k starting from the front aisle. Note that d_{ik} is set to zero when there is no item to be picked up in aisle k by the picker serving order i. Binary variable y_{jk} equals to 1 if and only if the picker serving batch j traverses aisle k. Binary variable c_j

equals to 1 if and only if the number of aisles that must be visited by the picker serving batch j is odd.

Binary variable p_{jk} equals to 1 if and only if aisle k is the rightmost aisle that is visited by a picker serving batch j. Integer variable v_j denotes the half of the number of aisles visited by the picker serving batch j if the number of aisles that must be visited by picker serving batch j is odd else the half of one plus number of the aisles visited by the picker serving batch j. Continuous variable h_j stands for the one way horizontal travel distance from the I/O point by the picker serving batch j. Continuous variable u_{jk} indicates the vertical one-way distance travelled in the rightmost aisle k, by the picker serving batch j and visits totally an odd number of aisles.

Table 3.2. Notation used for the OBP with Traversal Policy

d_{ik}	The vertical distance, that a picker serving order i should travel
y_{jk}	1 if and only if the picker serving batch j traverses aisle k, and 0 otherwise
c_j	1 if and only if the number of aisles that must be visited by the picker serving batch j is odd, and 0 otherwise
p_{jk}	1 if and only if aisle k is the rightmost aisle that is visited by a picker serving batch j, and 0 otherwise
v_j	The half of the number of aisles visited by the picker serving batch j if the number of aisles that must be visited by picker serving batch j is even else the half of one plus number of the aisles visited by the picker serving batch j
h_j	The one way horizontal travel distance from the I/O point by the picker serving batch j
u_{jk}	The vertical one-way distance travelled in the rightmost aisle k, by the picker serving batch j and visits totally an odd number of aisles.

Traversal: $Min\ z = 2\sum_{j=1}^{n}\sum_{k=1}^{K} u_{jk} + 2\sum_{j=1}^{n} h_j + 2L\sum_{j=1}^{n}(v_j - c_j)$ (4)

subject to

(1) – (3) (5)

$\sum_{i=1}^{n} d_{ik}x_{ij} \le Lny_{jk}$ for $j = 1, \dots, n;\ k = 1, \dots, K,$ (6)

$y_{jk} \le \sum_{i=1}^{n} d_{ik}x_{ij}$ for $j = 1, \dots, n;\ k = 1, \dots, K,$ *(7)*

$(k-1)wy_{jk} \le h_j$ for $j = 1, \dots, n;\ k = 2, \dots, K,$ (8)

$\sum_{k=1}^{K} y_{jk} + c_j = 2v_j$ for $j = 1, \dots, n,$ (9)

$d_{ik}x_{ij} \le u_{jk} + Ln\left(1 - p_{jk}\right) + Ln(1 - c_j)$ for $i, j = 1, \dots, n;\ k = 2, \dots, K,$ (10)

$y_{jk} - \sum_{l=k+1}^{K} y_{jl} \le p_{jk} \le y_{jk}$ for $j = 1, \dots, n;\ k = 1, \dots, K,$ (11)

v_j integer for $j = 1, \dots, n,$ (12)

$x_{ij} \in \{0,1\}$ for $i, j = 1, \dots, n,$ *(13)*

$p_{jk}, y_{jk} \in \{0,1\}$ for $j = 1, \dots, n;\ k = 1, \dots, K,$ (14)

$c_j \in \{0,1\}$ for $j = 1, \dots, n,$ (15)

$h_j \geq 0$ for $j = 1, \dots, n,$ (16)

$u_{jk} \geq 0$ for $j = 1, \dots, n;\ k = 1, \dots, K$ (17)

The objective function minimizes the total distance travelled by all pickers. Constraints (6) state that when there exists at least one item which belongs to order i and which is located in aisle k, then the picker serving the batch j which includes order i, must enter into the aisle k. Constraints (7) guarantee that when none of the orders belonging to batch j visits aisle k then y_{jk} is set to zero.

Constraints (8) calculate the maximum horizontal distance from the I/O point travelled towards the right of the front aisle by the picker serving batch j. Constraints (9) ensure that when the number of aisles traversed by a picker serving batch j is odd then c_j is set

to 1. Recall that in case that number of aisles to be crossed is odd then the picker performs a round trip along the rightmost aisle.

Constraints (10) compute the one-way vertical distance travelled in the rightmost aisle by the picker serving batch j when the number of aisles visited by that picker is odd. Constraints (11) state that p_{jk} should be equal to 1 when aisle k visited by the picker serving batch j is the rightmost aisle visited by that picker. Finally, constraints (12) - (17) are for the domain definition of the decision variables.

3.2 Return Policy

In addition to the parameters and variables defined above, the MILP formulation for the OBP considering return policy employs the following additional decision variable. Continuous decision variable s_{jk} indicates one way maximum vertical distance travelled in aisle k from the front aisle to the location of an item which belongs to an order assigned to batch j.

Table 3.3. Notation used for the OBP with Return Policy

s_{jk}	one way maximum vertical distance travelled in aisle k from the front aisle to the location of an item which belongs to an order assigned to batch j

$$\text{Min } z=\sum_{j=1}^{n} 2h_j + \sum_{j=1}^{n}\sum_{k=1}^{K} 2s_{jk} \quad (18)$$

subject to

(1) - (3), (6) - (8) (19)

$$d_{ik}x_{ij} \leq s_{jk} \qquad \text{for } i,j = 1,\dots,n;\ k = 1,\dots K, \qquad (20)$$

$$x_{ij}\,, y_{ij}\ \in \{0,1\} \qquad \text{for } i,j = 1,\dots,n, \qquad (21)$$

$$s_{jk} \geq 0 \qquad \text{for } j = 1,\dots,n;\ k = 1,\dots,K, \qquad (22)$$

$$h_j \geq 0 \qquad \text{for } j = 1,\dots,n. \qquad (23)$$

The objective function (18) calculates the total distance travelled by all pickers. Constraints (20) compute the maximum vertical distance travelled in each aisle k by a picker serving batch j. Constraints (21) – (22) give the domain definitions.

4. COMPUTATIONAL EXPERIMENTS

In this section we present the details of our computational experiments. The algorithms are coded in C++ and tested on a Dell Server PE2900 with two 3.16 GHz Quad Core Processors and 32 GB RAM with Microsoft Windows Server 2003 operating system. MILP problems are solved by CPLEX 11.0 solver with default options.

In the literature, there is no standard test library for the OBP. Hence we have produced our test instances to carry out computational experiments. In the test instances that we have randomly produced, the number of orders is selected between 10 and 100 with an increment of 10. For each number of orders we have generated 10 instances which totally make 100 OBP test problems. The number of items for each order is randomly chosen between 2 and 10. The items are randomly assigned to locations. According to the capacity of the order picker we have three classes of randomly generated test instances. These are Class I, Class II and Class III test instances, for picker capacities $Q = 24$, $Q = 36$, and $Q = 48$, respectively.

In Table 1, Table 2 and Table 3, we report the computational results obtained with OBP considering the traversal routing policy and picker capacities $Q = 24$, $Q = 36$ and $Q = 48$, respectively. Then in Table 4 and Table 5, we report the computational results obtained with OBP considering the return routing policy and picker capacities $Q = 24$, $Q = 36$ and $Q = 48$, respectively.

The first columns in all tables denote the instance names and sizes. The last row of all tables include the overall column averages. The number of orders n is followed by the capacity of the picker Q. For example the row 20_36 stands for the computational experiments obtained with 10 OBP test instances with 20 orders and picker capacity $Q = 36$. The next two columns include the experimental results obtained with the MILP model and the last two columns report the results obtained with the savings

heuristic. We have chosen the savings heuristic as the benchmarking algorithm because of its promising performance as pointed out by de Koster et al. (1999).

The CPU times reported are in seconds and UB stands for the upper bound value. We assess the performance of the proposed MILP models in terms of solution accuracy within a CPU time limit on randomly generated test problems. For OBP test instances with number of orders from 10 to 50 (60 to 100) we have imposed a CPU time limit to 1800 secs (10800 secs.). Therefore, the UB values reported are obtained with the feasible solutions output by the CPLEX. Note that, when CPLEX returns a solution value in less than these computational time limits, then the reported upper bound is the optimal solution value. The average percent improvements obtained with the MILP formulation for the OBP considering traversal policy over the savings algorithm are 0,03 %, 2,80 % and 5,99 % for picker capacity $Q = 24$, $Q = 36$ and $Q = 48$, respectively. These values are -2,67 %, 1,38 % and 2,28 % for the OBP considering return policy.

The formulation used to calculate the average percent improvements is

$$100 \times \frac{Z_{UB}^{S} - Z_{UB}^{MILP}}{Z_{UB}^{S}} \qquad (24)$$

where Z_{UB}^{S} and Z_{UB}^{MILP} are respectively the upper bounds obtained with the savings algorithm and CPLEX MILP solver. As can be observed, the performance of MILP formulations improves for larger values of Q. Furthermore, considering the overall average percent improvements the MILP formulation for the OBP considering traversal and return policies, which are 2,94 % and 0,33 % respectively. Moreover, when we consider the CPU times required by the MILP formulations and the savings algorithm; the winner is the saving algorithm. However, we believe the attempts to obtain exact solution of the OBP are worthwhile.

Table 4.1. Computational results with the MILP formulation and saving algorithm for the OBP considering traversal routing policy with picker capacity *Q = 24*.

TRAVERSAL	MILP		SAVINGS HEURISTIC	
	UB	CPU	UB	CPU
10_24	367,0	5	391,4	-
20_24	651,7	1.800	691,8	-
30_24	938,3	1.800	968,4	-
40_24	1.277,2	1.800	1.265,8	0,1
50_24	1.562,4	10.800	1.528,8	0,2
60_24	1.867,8	10.800	1.814,1	0,7
70_24	2.200,3	10.800	2.134,1	1,4
80_24	2.419,4	10.800	2.395,0	2,9
90_24	2.726,8	10.800	2.668,6	4,6
100_24	3.027,3	10.800	2.954,5	8,4
Average	1.703,8	6.120	1.681,3	1,9

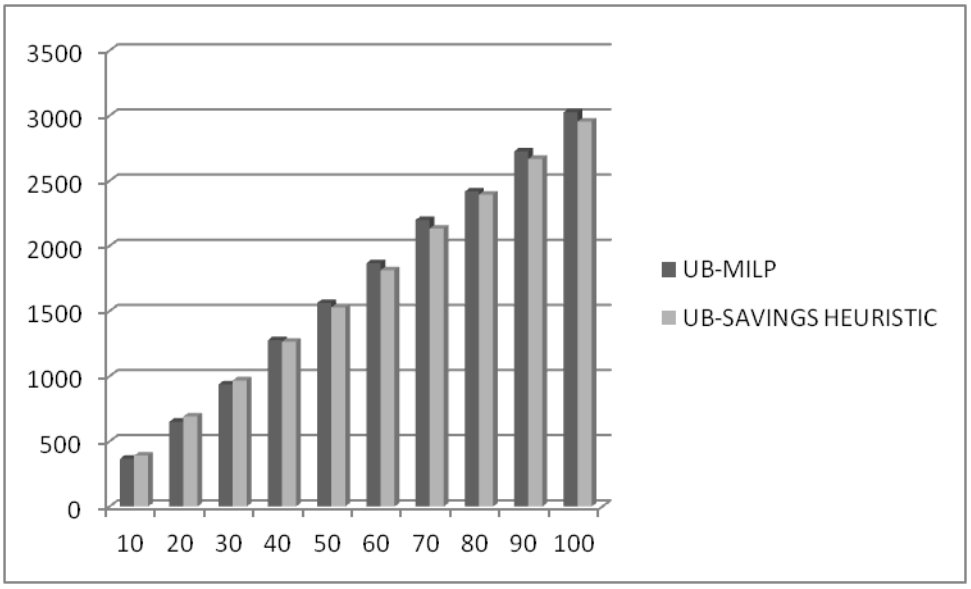

Figure 4.1. Histogram of the UB values are obtained with Q=24 and traversal policy

Table 4.2. Computational results with the MILP formulation and saving algorithm for the OBP considering traversal routing policy with picker capacity *Q = 36*.

TRAVERSAL	MILP		SAVINGS HEURISTIC	
	UB	CPU	UB	CPU
10_36	272,2	1.800	291,6	-
20_36	473,4	1.800	511,4	-
30_36	645,0	1.800	687,4	-
40_36	880,7	1.800	922,0	0,1
50_36	1.052,6	10.800	1.107,3	0,2
60_36	1.295,1	10.800	1.307,0	0,6
70_36	1.510,2	10.800	1.534,4	1,2
80_36	1.685,8	10.800	1.691,0	2,2
90_36	1.920,8	10.800	1.913,4	4,0
100_36	2.193,5	10.800	2.107,7	6,4
Average	1.192,9	7.200	1.207,3	1,5

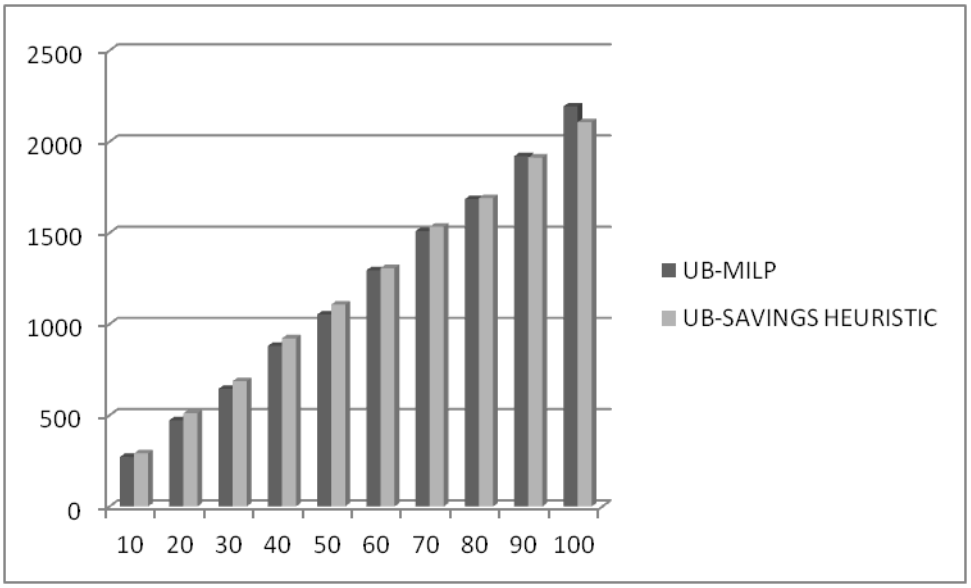

Figure 4.2. Histogram of the UB values are obtained with Q=36 and traversal policy

Table 4.3. Computational results with the MILP formulation and saving algorithm for the OBP considering traversal routing policy with picker capacity *Q = 48*.

TRAVERSAL	MILP		SAVINGS HEURISTIC	
	UB	CPU	UB	CPU
10_48	237,7	1.342	249,7	-
20_48	383,3	1.800	408,2	-
30_48	516,3	1.800	576,6	-
40_48	681,0	1.800	748,3	0,1
50_48	820,3	1.800	889,0	0,2
60_48	970,6	10.800	1.046,2	0,6
70_48	1.150,6	10.800	1.206,5	1,1
80_48	1.293,9	10.800	1.342,7	2,2
90_48	1.455,6	10.800	1.532,2	3,9
100_48	1.642,2	10.800	1.664,2	6,5
Average	915,2	6.254	966,4	1,5

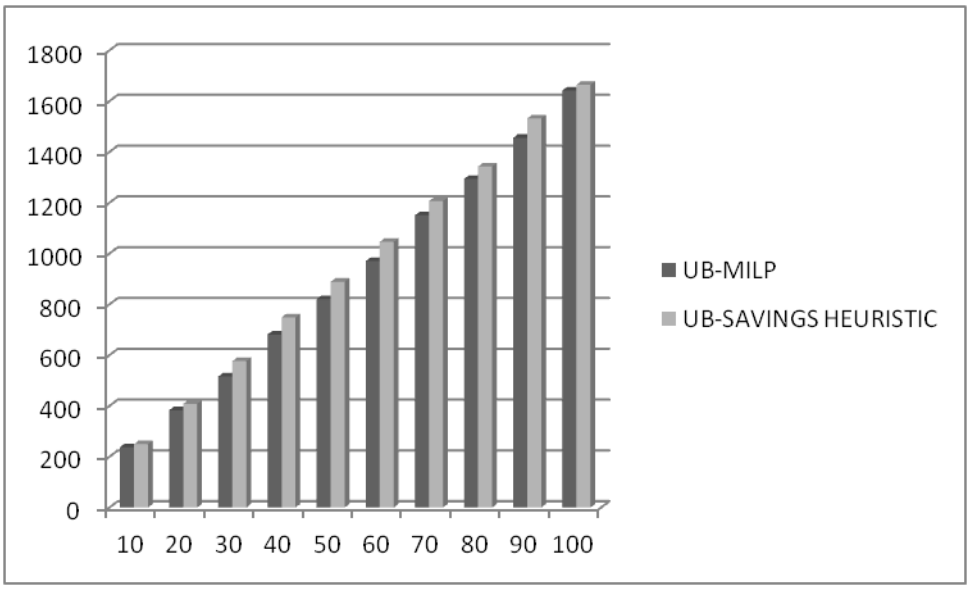

Figure 4.3. Histogram of the UB values are obtained with Q=48 and traversal policy

Table 4.4. Computational results with the MILP formulation and saving algorithm for the OBP considering return routing policy with picker capacity *Q = 24*.

	MILP		SAVINGS HEURISTIC	
	UB	**CPU**	**UB**	**CPU**
10_24	458,5	6	471,7	-
20_24	815,8	1.800	855,1	-
30_24	1.179,7	1.800	1.216,6	-
40_24	1.655,2	1.800	1.569,2	0,1
50_24	2.061,4	1.800	1.913,2	0,3
60_24	2.451,2	10.800	2.272,2	0,7
70_24	2.796,5	10.800	2.664,2	1,5
80_24	3.124,5	10.800	2.974,0	2,7
90_24	3.254,8	10.800	3.125,4	3,3
100_24	3.458,1	10.800	3.395,7	3,9
Average	2.125,6	6.121	2.045,7	1,3

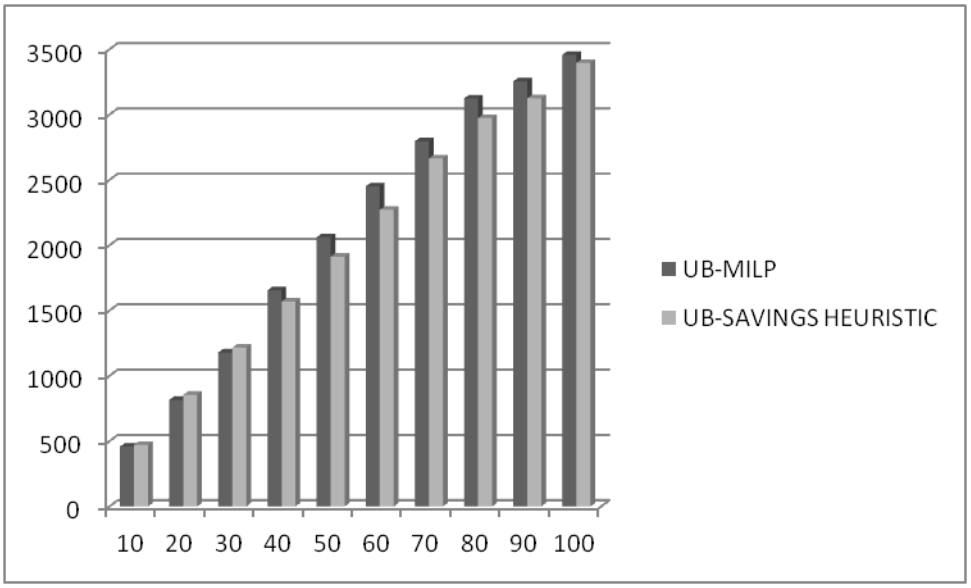

Figure 4.4. Histogram of the UB values are obtained with Q=24 and return policy

Table 4.5. Computational results with the MILP formulation and saving algorithm for the OBP considering return routing policy with picker capacity *Q = 36*.

RETURN	MILP		SAVINGS HEURISTIC	
	UB	CPU	UB	CPU
10_36	358,4	1.800	369,3	-
20_36	628,7	1.800	672,6	-
30_36	890,4	1.800	931,5	-
40_36	1.227,1	1.800	1.229,2	0,1
50_36	1.545,0	1.800	1.485,8	0,2
60_36	1.837,4	10.800	1.761,7	0,5
70_36	2.228,8	10.800	2.043,6	1,1
80_36	2.517,7	10.800	2.289,3	2,1
90_36	2.892,1	10.800	2.592,5	3,7
100_36	3.257,8	10.800	2.840,9	6,1
Average	1.738,3	6.300	1.621,6	1,4

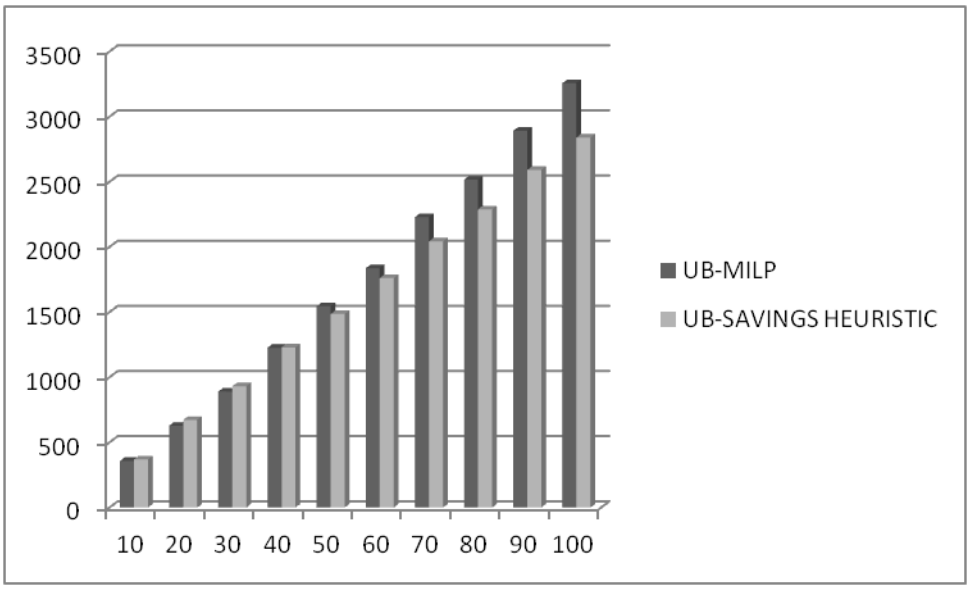

Figure 4.5. Histogram of the UB values are obtained with Q=36 and return policy

Table 4.6. Computational results with the MILP formulation and saving algorithm for the OBP considering return routing policy with picker capacity *Q = 48*.

RETURN	MILP		SAVINGS HEURISTIC	
	UB	CPU	UB	CPU
10_48	318,9	843	334,2	-
20_48	519,4	1.800	568,1	-
30_48	735,8	1.800	784,5	-
40_48	1.013,2	1.800	1.022,1	0,1
50_48	1.242,0	1.800	1.228,6	0,2
60_48	1.470,3	10.800	1.450,2	0,5
70_48	1.759,5	10.800	1.713,3	1,1
80_48	2.003,2	10.800	1.883,0	2,1
90_48	2.344,6	10.800	2.147,9	3,7
100_48	2.637,4	10.800	2.343,6	6,2
Average	1.404,4	6.204	1.347,6	1,4

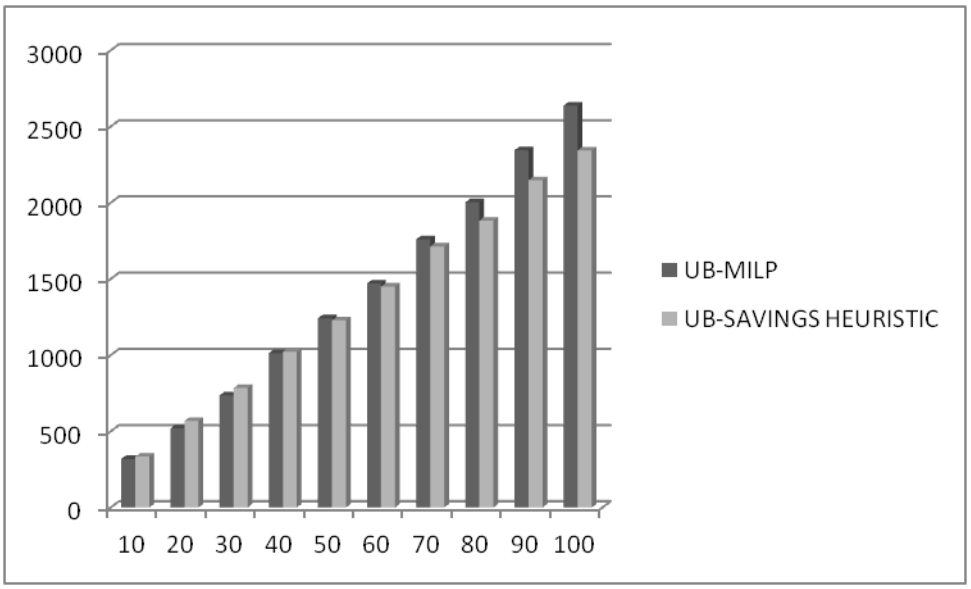

Figure 4.6. Histogram of the UB values are obtained with Q=48 and return policy

5. CONCLUSION

We have investigated the Order Batching Problem (OBP) which is known to be NP-hard. To the best of our knowledge, in the literature there are no Mixed Integer Linear Programming (MILP) formulations devised for the OBP considering traversal and return routing policies. The heuristic routing policies are easy to use and they are more applicable to construct similar routes. The ones proposed with this study are the very first attempts to suggest MILP formulations for the OBP. In this work, the proposed MILP formulations have been tested on randomly generated instances and they have been compared with the savings algorithm which is known to be one of the most promising construction heuristics for the OBP.

The computational results indicate that MILP and savings heuristic are useful for the OBP. According to our computational experiments, comparing both methods, savings heuristic yields significantly better results in reasonable CPU times. Thus, considering the trade-off between accuracy and efficiency, savings heuristic gives a reasonable solution approach in point of CPU times. Savings heuristics clearly outperform MILP approach.

From the experimental results, we observe that the proposed formulations yield quite good upper bounds and hence they can be used for benchmarking purposes. These MILP formulations can also be used as benchmarks for other studies which propose heuristic and meta-heuristics for the OBP. Moreover, there is also room to devise better MILP formulations for the OBP.

As a further research avenue, MILP formulations for the OBP with largest gap and composite policies can be developed. Besides, developing a Branch and Bound

algorithm exploiting the structure of the problem would be an interesting future study. Branch and cut and branch and cut and price algorithms can be also developed by suggesting valid inequalities based on the proposed MILP formulations.

REFERENCES

Albareda-Sambola, M., Alonso-Ayuso, M., Molina, E., and Simon de Blas, C. (2009). Variable neighborhood search for order batching in a warehouse. Asia-Pacic Journal of Operational Research, 26 (5), 655–683.

Bozer, Y.A., and Kile, J.W. (2008). Order batching in walk-and-pick order picking systems. International Journal of Production Research, 46 (7), 1887–1909.

Chen, M.-C., and Wu, H.-P. (2005a). An association-based clustering approach to order batching considering customer demand patterns. Omega - The International Journal of Management Science, 33 (4), 333–343.

Chen, M.-C., Huang, C.-L., Chen, K.-Y., and Wu, H.-P. (2005b). Aggregation of orders in distribution centers using data mining. Expert Systems with Applications, 28 (3), 453–460.

Clarke, G., and Wright, J.W. (1964). Scheduling of vehicles from a central depot to a number of delivery points. Operations Research, 12 (4), 568–581.

Coyle, J. J., Bardi, E. J. and Langley, C. J. (1996). The Management of Business Logistics. St Paul: West Publishing.

Çağırıcı, M. and Öncan, T. (2013). MILP Formulations for the Order Batching Problem in Low-Level Picker-to-Part Warehouse Systems. Proceeding of the 2013 IFAC Conference on Manufacturing, Modelling, Management and Control, Saint Petersburg State University and Saint Petersburg National Research University of Information Technologies, Mechanics and Optics, Saint Petersburg, Russia, June 19-21, 2013.

De Koster, R., Van Der Poort, E., and Wolters, M. (1999). Efficient orderbatching methods in warehouses. International Journal of Production Research, 37 (7), 1479–1504.

De Koster, R., Le-Duc, T., and Roodbergen, K.J. (2007). Design and control of warehouse order picking: A literature review. European Journal of Operational Research, 182, 481–501.

Drury, J. (1988). Towards more efficient order picking. IMM monograph no. 1, The Institute of Materials Managements: Cranfield, UK.

Elsayed, E.A. (1981). Algorithms for optimal material handling in automatic warehousing systems. International Journal of Production Research, 19 (5), 525– 535.

European Logistics Association and Kearney, A.T. (2004). Excellence in Logistics 2004, European Logistics Association, Brussels.

Frazelle, E. (2002). World-Class Warehousing and Material Handling. New York: McGraw-Hill.

Gademann, N., Van den Berg, J., and Van der Hoff , H. (2001). An order batching algorithm for wave picking in a parallel-aisle warehouse. IIE Transactions, 33(5), 385–398.

Gademann, N., and Van de Velde, S. (2005). Order batching to minimize total travel time in a parallel-aisle warehouse. IIE Transactions, 37(1), 63–75.

Glover, F., 1986. Future paths for integer programming and links to artificial intelligence Computer&Operation Research 13, 533-549.

Gibson, D.R., and Sharp, G.P.(1992). Order Batching Procedures. European Journal of Operational Research, 58, 57–67.

Goetschalckx, M., and Ratliff, H.D. (1998) Order picking in an aisle. IIE Transactions, 20(1), 53–62.

Hall, R.W. (1993). Distance approximations for routing manual pickers in a warehouse. IIE Transactions, 24 (4), 76–87.

Henn, S., Koch, S., Karl, D., Strauss, C., and Wäscher, G. (2009). Meta-heuristics for the order batching problem in manual order picking systems. Working Paper 20/2009, Faculty of Economics and Management, Otto-von-Guericke-University Magdeburg.

Henn, S., and Wäscher, G. (2012). Tabu search heuristics for the order batching problem in manual order picking systems. European Journal of Operational Research, 222, 484–494.

Ho, Y.-C., and Tseng, Y.-Y. (2006). A study on orderbatching methods of order-picking in a distribution centre with two cross-aisles. International Journal of Production Research, 44(17), 3391–3417.

Ho, Y.-C., Su, T.-S., and Shi, Z.-B. (2008). Order-bathcing methods for an order-picking warehouse with two cross aisles. Computers and Industrial Engineering, 55(2), 321–347.

Hsu, C.-M., Chen, K.-Y., and Chen, M.-C. (2005). Batching orders in warehouses by minimizing travel distance with genetic algorithms. Computers in Industry, 56(2), 169–178.

Hwang, H., and Kim, D.G. (2005). Order-batching heuristics based on cluster analysis in low-level picker-topart warehousing system. International Journal of Production Research, 43 (17), 3657–3670.

Islam, K.M.S., and Sarker, B.R. (2000). A similarity coefficient measure and machine-parts grouping in cellular manufacturing systems. International Journal of Production Research, 38 (3), 699–720.

Matusiak, M., de Koster, R., Kroon, L.,Saarinen, J. (2013). A fast simulated annealing method for batching precedence-constrained customer orders in a warehouse. European Journal of Operational Research.

Mladenovic, N.,(1995), A variable neighborhood algorithm- a new meta-heuristic for combinatorial optimization. In Abstract of papers presented at Optimization Days, page 112, Montreal

Mladenovic, N. and Hansen, P. (1997). Variable neighborhood search. Computers Operations Research, 24:1097-1100

Öncan, T. (2013). A genetic algorithm for the order batching problem in low-level picker-to-part warehouse systems. International MultiConference of Engineers and Computer Scientist 2013. Vol 1, IMECS 2013, March 13-15, 2013, Hong Kong.

Petersen, C.G. (1997). An evaluation of order picking rout ing policies. International Journal of Operations and Production Management, 17 (11), 1098–1111.

Ratliff, H.D., and Rosenthal, A.S. (1983). Orderpicking in a rectangular warehouse: a solvable case of the traveling salesman problem. Operations Research, 31, 507–521.

Roodbergen, K.J., and De Koster, R.. (2001). Routing methods for warehouses with multiple cross aisles. International Journal of Production Research, 39(9), 1865–1883.

Rosenwein, M.B. (1996). A comparison of heuristics for the problem of batching orders for warehouse selection. International Journal of Production Research, 34, 657-664.

Ruben, R.A., and Jacobs, F.R. (1999). Batch construction and storage assignment. Management Science, 45(4), 575–596.

Tompkins, J.A., White, J.A., Bozer, Y.A., and Tanchoco, J.M.A. (2003). Facilities Planning. John Wiley & Sons, New Jersey, 3rd edition.

Tsai, C.-Y., Liou, J.J.M., and Huang, T.-M. (2008). Using a multiple-GA method to solve the batch picking problem: considering travel distance and order due time. International Journal of Production Research, 46(22), 6533–6555.

Wäscher, G. (2004). Order picking: a survey of planning problems ans methods. In: H. Dyckoff, R. Lackes, J. Reeves (eds.) Supply Chain Management and Reverse Logistics, pages 323–347, Springer, Berlin.

APPENDIX

APPENDIX A : C ++ Codes of Traversal Strategy

```
#include <ilcplex/ilocplex.h>
ILOSTLBEGIN
#include <cmath>
#include <fstream>
#include <string>
#include <time.h>

using namespace std;

double cpu_time() {
   clock_t t;  static clock_t last = (clock_t)-1;
   //  TEST_MESSAGE("clock()");
   t = clock();
   if (last == (clock_t)-1) last = t;
     return (double)(t-last)/CLOCKS_PER_SEC;
}

ILOSTLBEGIN
#define EPSCUT 1E-2
#define BIGVAL 10E+12
#define EPSVAL 1E-4
#define myrandom01() (rand() / ((double)RAND_MAX + 1.0))

typedef IloArray<IloBoolVarArray> BoolVarMatrix;
```

```
typedef IloArray<IloArray<IloBoolVarArray> > BoolVarMatrix2;
typedef IloArray<IloNumArray> NumMatrix;
typedef IloArray<IloArray<IloNumArray> > NumMatrix2;
typedef IloArray<IloNumVarArray> NumVarMatrix;
typedef IloArray<IloArray<IloNumVarArray> > NumVarMatrix2;
typedef IloArray<IloIntVarArray> IntVarMatrix;
typedef IloArray<BoolVarMatrix>  BoolVar3DimArray;
typedef IloArray<IloBoolArray> BoolMatrix;

extern IloInt n;
extern IloInt pi;
extern IloNumVarArray H;
extern BoolVarMatrix Y;
extern IloIntVarArray V;
extern IloIntVarArray CV;
extern BoolVarMatrix X;
extern NumVarMatrix BA;
extern BoolVarMatrix POS;
extern IloBoolVarArray CT;
extern NumMatrix distances;

ILOSTLBEGIN

NumMatrix distances;
IloBoolVarArray Xarray;
IloNumVarArray H;
BoolVarMatrix Y;
IloIntVarArray V;
IloIntVarArray CV;
BoolVarMatrix X;
BoolVarMatrix POS;
IloBoolVarArray CT;
```

```
NumVarMatrix BA;
IloInt n;
IloInt pi;

int main(int argc, char **argv) {

  IloEnv env;
double bas,son;

double sag;

  try {
     IloModel model(env);
   ifstream inFile;
                inFile.open(argv[1]);
        if (!inFile ) {
     cout << "Unable to open file";
           exit(1); // terminate with error
  }

        ofstream outFile;
        outFile.open(argv[2],ios::app );
        if (!outFile ) {
     cout << "Unable to open file";
           exit(1); // terminate with error
  }
        outFile<<argv[1]<<"\t";

        char name[128];
        IloInt Aisles=10;
        IloInt Capacity=24;
        IloNum Width=2.4;
```

```
        IloInt i,j,k,s,l,ll,kk;
        IloInt orders;
        IloInt items;
inFile >> orders;

        NumMatrix QQ_X=NumMatrix(env, orders+1);
        NumMatrix QQ_Y=NumMatrix(env, orders+1);
        for (i = 0; i <= orders; i++){
           QQ_X[i] = IloNumArray(env, 2*orders+1);
           QQ_Y[i] = IloNumArray(env, 2*orders+1);
        }
        for (i = 0; i <= orders; i++){
                for (j = 0; j <= 2*orders; j++){
                        QQ_X[i][j]=0;
                        QQ_Y[i][j]=0;
                }
        }

 IloNumArray SAYAC=IloNumArray(env, orders+1);
 for(j=0;j<=orders;j++){       SAYAC[j]=0;}
for (i = 1; i <= orders; i++){
        inFile >> items;
SAYAC[i]=items;
        for (j = 1; j <= items; j++){
                inFile >>ll;
                inFile >>kk;
                QQ_X[i][j]=ll;
                QQ_Y[i][j]=kk;
        }
}
```

```
inFile.close();
        NumMatrix Q_ALT=NumMatrix(env, orders+1);
        for (i = 0; i <= orders; i++){
           Q_ALT[i] = IloNumArray(env, 11);
        }

                for (i = 0; i <= orders; i++){
                for (j = 0; j <= 10; j++){
                        Q_ALT[i][j]=0;
                }
        }
for(i=1;i<=orders;i++){
        for (j = 1; j <= SAYAC[i]; j++){
                        ll=QQ_X[i][j];
                        kk=QQ_Y[i][j];
                        if(kk>Q_ALT[i][ll])  Q_ALT[i][ll]=kk;

        }
}

NumMatrix AA=NumMatrix(env, orders+1);

for (i = 0; i <= orders; i++){
           AA[i] = IloNumArray(env, 11);
        }
for (i = 0; i <= orders; i++){
        for (j = 0; j <= 10; j++){
                        AA[i][j]=0;
                }
        }
```

```
for(i=1;i<=orders;i++){
        for (j = 1; j <= SAYAC[i]; j++){
        ll=QQ_X[i][j];
AA[i][ll]=1;
        }
}
cout<<"X \n";
X = BoolVarMatrix(env, orders+1);
for (i = 1; i <=orders; i++) {
            X[i] = IloBoolVarArray(env, orders+1);
            for (j = 1; j <= orders; j++) {
                        sprintf_s(name, "X[%ld][%ld]", i, j);
                        X[i][j] = IloBoolVar(env,0,1, name);
                }
 }
   CT = IloBoolVarArray(env, orders+1);

   for (j = 1; j <= orders; j++) {
                sprintf_s(name, "CT[%ld]", j);
                CT[j] = IloBoolVar(env,0,1, name);
   }
  Y = BoolVarMatrix(env, orders+1);

 for (i = 1; i <=orders; i++) {
   Y[i] = IloBoolVarArray(env, 11);

        for (j = 1; j <= 10; j++) {
                sprintf_s(name, "Y[%ld][%ld]", i, j);
                Y[i][j] = IloBoolVar(env,0,1, name);
   }
 }
```

```
H= IloNumVarArray(env, orders+1);

for (j = 1; j <=orders; j++) {
                sprintf_s(name, "H[%ld]", j);
                H[j] = IloNumVar(env,0,24, name);
 }

V= IloIntVarArray(env, orders+1);

for (j = 1; j <=orders; j++) {
                sprintf_s(name, "V[%ld]", j);
                V[j] = IloIntVar(env, 0, 10, name);
}

CV= IloIntVarArray(env, orders+1);

for (j = 1; j <=orders; j++) {
                sprintf_s(name, "CV[%ld]", j);
                CV[j] = IloIntVar(env, 0, 10, name);
}

POS = BoolVarMatrix(env, orders+1);

for (i = 1; i <=orders; i++) {
          POS[i] = IloBoolVarArray(env, 11);
           for (j = 1; j <= 10; j++) {
                        sprintf_s(name, "POS[%ld][%ld]", i, j);
                        POS[i][j] = IloBoolVar(env,0,1, name);
                }
}
```

```
BA = NumVarMatrix(env, orders+1);
 for (i = 1; i <=orders; i++) {
 BA[i] = IloNumVarArray(env, 11);
  for (j = 1; j <= 10; j++) {
                sprintf_s(name, "BA[%ld][%ld]", i, j);
                BA[i][j] = IloNumVar(env,0,10, name);
   }
  }

        for(i=1; i<=orders; i++){
                IloExpr sum1(env);
                for(j=1; j<=orders; j++){
                        sum1+=IloExpr(X[i][j]);
                }
                model.add(sum1 == 1);
                sum1.end();
        }

        for(i=1; i<=orders; i++){
                for(j=1; j<=orders; j++){
                        if(i!=j){
                                IloExpr sum1(env);
                                sum1+=IloExpr(X[i][j]);
                                sum1-=IloExpr(X[j][j]);

                                model.add(sum1 <= 0);
                                sum1.end();
                        }
                }
        }

for(j=1; j<=orders; j++){
```

```
        IloExpr sum1(env);
        for(i=1; i<=orders; i++){
        sum1+=IloExpr(SAYAC[i]*X[i][j]);
        }
        model.add(sum1 <= Capacity);
        sum1.end();
}

for(j=1; j<=orders; j++){
        for(k=1; k<=10; k++){
                IloExpr sum1(env);
                ll=0;
                for(i=1; i<=orders; i++){
                        sum1+=IloExpr(AA[i][k]*X[i][j]);
                        if (AA[i][k])ll++;
                }
                if(ll>0){
                sum1-=IloExpr(orders*Y[j][k]);
                model.add(sum1 <= 0);
                }
                sum1.end();
        }
}

cout<<"c1 \n";

for(j=1; j<=orders; j++){
        for(k=2; k<=10; k++){
                IloExpr sum1(env);

                sum1+=IloExpr((k-1)*Width*Y[j][k]);
                sum1-=IloExpr(H[j]);
```

```
                model.add(sum1 <= 0);
                sum1.end();
        }
}

for(j=1; j<=orders; j++){
        IloExpr sum1(env);
        for(k=1; k<=10; k++){
                sum1-=IloExpr(Y[j][k]);
        }
        sum1+=IloExpr(2*V[j]);
        sum1-=IloExpr(CT[j]);
        model.add(sum1 == 0);
        sum1.end();
}


for(j=1; j<=orders; j++){
        IloExpr sum1(env);
        sum1+=IloExpr(V[j]);
        sum1-=IloExpr(CT[j]);
        sum1-=IloExpr(CV[j]);
        model.add(sum1 <= 0);
        sum1.end();
}

for(j=1; j<=orders; j++){
        for(k=1; k<=10; k++){
                IloExpr sum1(env);
                for(l=k+1; l<=10; l++){       sum1-=IloExpr(Y[j][l]);       }
                sum1+=IloExpr(Y[j][k]);
```

```
                sum1-=IloExpr((POS[j][k]));
                model.add(sum1 <= 0);
                sum1.end();
        }
}

cout<<"c22 \n";

for(j=1; j<=orders; j++){
        for(k=1; k<=10; k++){
                IloExpr sum1(env);
                //for(l=k+1; l<=10; l++){       sum1-=IloExpr(1-Y[j][l]);     }
                //sum1-=IloExpr(10*orders*(Y[j][k]));
                sum1-=IloExpr((Y[j][k]));

                sum1+=IloExpr(POS[j][k]);
                model.add(sum1 <= 0);
                sum1.end();
        }
}

cout<<"c3 \n";

for(j=1; j<=orders; j++){
        for(k=1; k<=10; k++){
                IloExpr sum1(env);
                for(i=1; i<=orders; i++){sum1-=IloExpr(Q_ALT[i][k]*X[i][j]);   }
                sum1+=IloExpr(Y[j][k]);
                model.add(sum1 <= 0);
                sum1.end();
        }
}
```

```
for(i=1; i<=orders; i++){
        for(j=1; j<=orders;j++){
                for(k=1; k<=10; k++){
                        if(Q_ALT[i][k]){
                                IloExpr sum1(env);
                                sum1+=IloExpr(Q_ALT[i][k]*X[i][j]);
                                sum1-=IloExpr(10*orders*(1-POS[j][k]));

                                sum1-=IloExpr(10*orders*(1-CT[j]));

                                sum1-=IloExpr(BA[j][k]);
                                model.add(sum1 <= 0);
                                sum1.end();
                        }
                }
        }
}

IloExpr objective(env);
for(i=1; i<=orders; i++){
                objective+=IloExpr(2*10*(CV[i]));
                objective+=IloExpr(2*H[i]);
                }

for(j=1; j<=orders; j++){
        for(k=1; k<=10; k++){
//objective+=IloExpr(POS[j][k]);
objective+=IloExpr(2*BA[j][k]);
        }

}
```

```
model.add(IloMinimize(env, objective));
objective.end();
IloCplex cplex(model);
bas=cpu_time() ;
cplex.setParam(IloCplex::TiLim, 10800);

if ( !cplex.solve() ) {
        env.error() << "Failed to optimize LP" << endl;
                    throw(-1);
    }

son=cpu_time() ;
env.out() << "Solution status = " << cplex.getStatus() << endl;
outFile << "Solution status = " << cplex.getStatus() << " ";
env.out() << "Solution value  = " << cplex.getObjValue() << endl;
outFile << "Solution value  = " << cplex.getObjValue() << " ";
outFile.close();

 env.end();
  }
  catch (IloException& e) {
    cerr << "ERROR: " << e.getMessage() << endl;
  }
  catch (...) {
    cerr << "Error" << endl;
  }



  env.end();
  return 0; }
```

APPENDIX B : C ++ Codes of Return Strategy

```
#include <ilcplex/ilocplex.h>
ILOSTLBEGIN
#include <cmath>
#include <fstream>
#include <string>
#include <time.h>

using namespace std;


double cpu_time() {
    clock_t t;  static clock_t last = (clock_t)-1;
    t = clock();
    if (last == (clock_t)-1) last = t;
      return (double)(t-last)/CLOCKS_PER_SEC;
}


ILOSTLBEGIN
#define EPSCUT 1E-2
#define BIGVAL 10E+12
#define EPSVAL 1E-4
#define myrandom01() (rand() / ((double)RAND_MAX + 1.0))



typedef IloArray<IloBoolVarArray> BoolVarMatrix; //cordeau
typedef IloArray<IloArray<IloBoolVarArray> > BoolVarMatrix2; //cordeau
typedef IloArray<IloNumArray> NumMatrix;  //cordeau
typedef IloArray<IloArray<IloNumArray> > NumMatrix2; //cordeau
typedef IloArray<IloNumVarArray> NumVarMatrix;
typedef IloArray<IloArray<IloNumVarArray> > NumVarMatrix2;
typedef IloArray<IloIntVarArray> IntVarMatrix;
typedef IloArray<BoolVarMatrix>  BoolVar3DimArray;
typedef IloArray<IloBoolArray> BoolMatrix;

extern IloInt n;
```

```
extern IloInt pi;
extern IloNumVarArray H;
extern BoolVarMatrix Y;
extern NumVarMatrix B;
extern BoolVarMatrix X;
extern NumMatrix distances;

ILOSTLBEGIN
NumMatrix distances;
IloBoolVarArray Xarray;
IloNumVarArray H;
BoolVarMatrix Y;
NumVarMatrix B;
BoolVarMatrix X;
IloInt n;
IloInt pi;

int main(int argc, char **argv) {

IloEnv env;
double bas,son;

double sag;
 try {

    IloModel model(env);
     ifstream inFile;
            inFile.open(argv[1]);
      if (!inFile ) {
         cout << "Unable to open file";
           exit(1); // terminate with error
    }


ofstream outFile;
outFile.open(argv[2],ios::app );
if (!outFile ) {
         cout << "Unable to open file";
           exit(1);
```

```
    }


outFile<<argv[1]<<"\t";
char name[128];
IloInt Aisles=10;
IloInt Capacity=24;
IloNum Width=2.4;
IloInt i,j,k,s,ll,kk;
IloInt orders;
IloInt items;

inFile >> orders;

NumMatrix QQ_X=NumMatrix(env, orders+1);
NumMatrix QQ_Y=NumMatrix(env, orders+1);
for (i = 0; i <= orders; i++){
       QQ_X[i] = IloNumArray(env, 2*orders+1);
       QQ_Y[i] = IloNumArray(env, 2*orders+1);
      }
for (i = 0; i <= orders; i++){
      for (j = 0; j <= 2*orders; j++){
                  QQ_X[i][j]=0;
                  QQ_Y[i][j]=0;
            }
      }

IloNumArray SAYAC=IloNumArray(env, orders+1);

for(j=0;j<=orders;j++){ SAYAC[j]=0;}

for (i = 1; i <= orders; i++){
      inFile >> items;
SAYAC[i]=items;
      for (j = 1; j <= items; j++){
            inFile >>ll;
            inFile >>kk;
            QQ_X[i][j]=ll;
            QQ_Y[i][j]=kk;
```

```
      }
}

inFile.close();

NumMatrix AA=NumMatrix(env, orders+1);
for (i = 0; i <= orders; i++){
          AA[i] = IloNumArray(env, 11);
      }

for (i = 0; i <= orders; i++){
      for (j = 0; j <= 10; j++){
            AA[i][j]=0;
            }
      }

for(i=1;i<=orders;i++){
      for (j = 1; j <= SAYAC[i]; j++){
            ll=QQ_X[i][j];
            kk=QQ_Y[i][j];

            if(kk>AA[i][ll])AA[i][ll]=kk;
      }
}

X = BoolVarMatrix(env, orders+1);
  for (i = 1; i <=orders; i++) {
          X[i] = IloBoolVarArray(env, orders+1);
          for (j = 1; j <= orders; j++) {
                  sprintf_s(name, "X[%ld][%ld]", i, j);
                  X[i][j] = IloBoolVar(env,0,1, name);
            }
  }



  Y = BoolVarMatrix(env, orders+1);
  for (i = 1; i <=orders; i++) {
    Y[i] = IloBoolVarArray(env, 11);
```

```
    for (j = 1; j <= 10; j++) {
            sprintf_s(name, "Y[%ld][%ld]", i, j);
            Y[i][j] = IloBoolVar(env,0,1, name);
    }
  }

   H= IloNumVarArray(env, orders+1);
   for (j = 1; j <=orders; j++) {
            sprintf_s(name, "H[%ld]", j);
            H[j] = IloNumVar(env,0,BIGVAL, name);
    }

   B= NumVarMatrix(env, orders+1);
   for (i = 1; i <=orders; i++) {
         B[i] = IloNumVarArray(env, 11);
               for (j = 1; j <= 10; j++) {
                       sprintf_s(name, "B[%ld][%ld]", i, j);
                       B[i][j] = IloNumVar(env, 0, 10, name);
                 }
   }

for(i=1; i<=orders; i++){
      IloExpr sum1(env);
      for(j=1; j<=orders; j++){
                  sum1+=IloExpr(X[i][j]);
            }
      model.add(sum1 == 1);
      sum1.end();
      }

for(i=1; i<=orders; i++){
      for(j=1; j<=orders; j++){
            if(i!=j){
                  IloExpr sum1(env);
                  sum1+=IloExpr(X[i][j]);
                  sum1-=IloExpr(X[j][j]);

                  model.add(sum1 <= 0);
                  sum1.end();
```

```
                }
            }
        }

for(j=1; j<=orders; j++){
        IloExpr sum1(env);
        for(i=1; i<=orders; i++){
        sum1+=IloExpr(SAYAC[i]*X[i][j]);
        }
        model.add(sum1 <= Capacity);
        sum1.end();
}

for(j=1; j<=orders; j++){
        for(k=1; k<=10; k++){
            IloExpr sum1(env);
            ll=0;
            for(i=1; i<=orders; i++){
                sum1+=IloExpr(AA[i][k]*X[i][j]);
                if (AA[i][k])ll++;
            }
            if(ll>0){
            sum1-=IloExpr(10*orders*Y[j][k]);
            model.add(sum1 <= 0);
            }
            sum1.end();
        }
}

for(j=1; j<=orders; j++){
        for(k=2; k<=10; k++){
            IloExpr sum1(env);

            sum1+=IloExpr((k-1)*Width*Y[j][k]);
            sum1-=IloExpr(H[j]);
            model.add(sum1 <= 0);
            sum1.end();
        }
}
```

```
for(i=1; i<=orders; i++){
      for(j=1; j<=orders; j++){
            for(k=1; k<=10; k++){
                  if(AA[i][k]>0){
                        IloExpr sum1(env);
                        sum1+=IloExpr(AA[i][k]*X[i][j]);
                        sum1-=IloExpr(B[j][k]);
                        model.add(sum1 <= 0);
                        sum1.end();
                  }
            }
      }
}
IloExpr objective(env);
for(i=1; i<=orders; i++){
            objective+=IloExpr(2*H[i]);
      }

for(i=1; i<=orders; i++){
      for(k=1; k<=10; k++){
      objective+=IloExpr(2*B[i][k]);
            }
      }

model.add(IloMinimize(env, objective));
objective.end();
IloCplex cplex(model);
bas=cpu_time() ;
cplex.exportModel("mod.lp");

cplex.setParam(IloCplex::TiLim, 1800);
if ( !cplex.solve() ) {
         env.error() << "Failed to optimize LP" << endl;
       throw(-1);
      }
son=cpu_time() ;
        cout<<"  CPU:    "<<son-bas<<endl;
        outFile<<"  CPU:    "<<son-bas<<"\t";
```

```
        env.out() << "Solution status = " << cplex.getStatus() <<
endl;
      outFile << "Solution status = " << cplex.getStatus() << "\t";

        env.out() << "Solution value  = " << cplex.getObjValue() <<
endl;
        outFile << "Solution value  = \t " << cplex.getObjValue() <<
endl;

for (i = 1; i <= orders; i++){
      for (j = 1; j <= orders; j++){
                  if (   cplex.getValue(X[i][j]) > EPSVAL)
                  {
                  cout << "X[" << i << "][" << j << "] = "
<<cplex.getValue(X[i][j]) << endl;
                        }
            }}

outFile.close();


env.end();
   }
   catch (IloException& e) {
      cerr << "ERROR: " << e.getMessage() << endl;
   }
   catch (...) {
      cerr << "Error" << endl;
   }

   env.end();
   return 0;
}
```

BIOGRAPHICAL SKETCH

Merve ÇAĞIRICI was born December 7, 1987 in İzmir. She has studied at Aydın Science High School where she graduated in 2006. She started her undergraduate studies at Industrial Engineering Department of Galatasaray University in 2006.

She has published the paper titled "MILP Formulations for the Order Batching Problem in Low-Level Picker-to-Part Warehouse Systems" in the proceeding of the 2013 IFAC Conference on Manufacturing, Modelling, Management, and Control, Saint Petersburg State University and Saint Petersburg National Research University of Information Technologies, Mechanics and Optics, Saint Petersburg, Russia, June 19-21, 2013.

She has been working as a sales system developer analyst at ETİ since July 2012.