Article

# Effective Heuristics for Solving the Multi-Item Uncapacitated Lot-Sizing Problem with Near-Minimal Storage Capacities

Warut Boonphakdee , Duangrat Hirunyasiri * , Peerayuth Charnsethikul

Posted Date: 3 June 2025

doi: 10.20944/preprints202506.0178.v1

Keywords: multi-item; lot size; near-minimal storage capacity; replenishment plan

*Article*

# A Push–Pull Heuristic for Solving the Multi-Item Uncapacitated Lot-Sizing Problem under Near-Minimal Storage Capacities

**Warut Boonphakdee [1], Duangrat Hirunyasiri [2,*] and Peerayuth Charnsethikul [3]**

[1] Department of Industrial Engineering, Faculty of Engineering at Kamphaeng Saen, Kasetsart University, Nakhon Pathom 73140, Thailand

[2] Department of Textile Science, Faculty of Agro-Industry, Kasetsart University, Bangkok 10900, Thailand

[3] Department of Industrial Engineering, Faculty of Engineering at Kamphaeng Saen, Kasetsart University, Nakhon Pathom 73140, Thailand

* Correspondence: duangrat.c@ku.th

**Abstract:** In inventory management, storage capacity constraints complicate multi-item lot-sizing decisions. As the number of items increases, deciding how much of each item to order without exceeding capacity becomes more difficult. Dynamic programming works efficiently for a single item, but when capacity constraints are nearly minimal across multiple items, novel heuristics are required. However, previous heuristics have mainly focused on inventory bound constraints. Therefore, this paper introduces push and pull heuristics to solve the multi-item uncapacitated lot-sizing problem under near-minimal capacities.. First, a dynamic programming based on network flow model was used to generate the initial replenishment plan for the single-item lot-sizing problem. Next, under storage capacity constraints, the push operation moved the selected replenishment quantities from the current period to subsequent periods to meet all demand requirements. Finally, the pull operation shifted the selected replenishment quantities from the current period into earlier periods, ensuring that all demand requirements were satisfied. The results of the random experiment showed that the proposed heuristic generated solutions whose performance compared well with the optimal solution. This heuristic effectively solves all randomly generated instances representing worst-case conditions, ensuring robust operation under near-minimal storage. For large-scale problems under near-minimal storage capacity constraints, the proposed heuristic achieved only small optimality gaps while requiring less running time. However, small- and medium-scale problems can be solved optimally by a Mixed-Integer Programming (MIP) solver with minimal running time.

**Keywords:** multi-item; lot size; near-minimal storage capacity; replenishment plan

## 1. Introduction

In supply chain management, the inventory bound, or limitation storage is an im-portant constraint. Raw materials cannot be stored in huge volumes, although the unit price of a raw material unit is low. If raw materials have many different items and stock-keeping units (SKUs), it results in complex problems. Supply chain management decisions depend on the procurement policy of the organization. It comprises operating on minimum inventory cost while considering whether or not to use an expanded storage. An expanded storage area is a feasible solution, which operates at the lowest inventory cost. However, the investment cost for land lease and acquisition, including the rack and shelving supply, must be considered as well. In the case of not considering an expanded storage area, there are two feasible solutions. Firstly, when inventory is over, it must be kept in another area, such as the production line; notwithstanding, this result is with late disbursement, lost materials, and incorrect counting of the number of remaining materials. Total inventory cost is still low. Secondly, keep the amount of inventory under the storage capacity, but the order frequency must grow evidently. As a result, the storage cost continues to be high. When the number of items or SKUs increase, deciding how much of each item to order without exceeding capacity becomes more difficult. In many process industries such as paper manufacturing, petrochemical manufacturing, refineries, food processing, and pharmaceutical manufacturing, storage capacity has become a limiting factor.

In practice, industries such as trailer assembly processes [11], raw-material perishability in composites [22] must all contend with limited inventory bounds. Tight inventory bounds or near-minimal storage capacities that slightly exceed demand require specialized heuristics. Industries operating under Just-In-Time assembly, such as automotive plants [26], are especially sensitive to these constraints. Therefore, the authors introduce a dynamic programming approach based on network flow to generate an initial replenishment plan for each single item and develop a new heuristic to manage the multi-item lot-sizing problem under near-minimal storage capacities.

Storage capacity has been defined by researchers as comprising two distinct categories, such as: the number of ending inventory only, and the sum of the number for the beginning inventory and replenishment. Firstly, most researchers proposed the storage capacity to be based on the number of ending inventory only. Secondly, other researchers introduced the storage capacity to be the total of the number of the beginning inventory and replenishment. Thus, many researchers have defined inventory as the quantity of goods on hand at the end of a specific period. However, in the real world problem, raw material stores usually receive the replenishment at the beginning of the period and keep the inventory of the previous period together to be not over the storage capacity. Consequently, this paper suggests that the total number of inventory in each period depends on the sum of inventory and replenishment at the beginning of the period. There are two definitions of storage capacity: one based on the number of ending inventory, and the other on the sum of beginning inventory plus replenishment. These different definitions not only affect how inventory is calculated but also directly impact the formulation of inventory planning models. As a result, various algorithms have been developed to solve these models.

Lot sizing problems are typically solved using exact methods such as MIP , dynamic programming and heuristic techniques. The earliest known MIP formulation for the lot-sizing problem in the U. S. petroleum refining industry was introduced by Manne [18] in 1958. In his seminal paper, he presented a mathematical model for the dynamic lot-sizing problem, which develop in production planning and inventory control. Meanwhile,Wagner and Whitin [19] introduced a forward algorithm based on the dynamic programming approach to search for optimal lot size decisions. They established the optimal lot sizes for a single item when demand, inventory holding charges, and setup costs change over time. For the management of procurement of materials with storage capacity, consider both the single item and multi-item.

The dynamic programming approach have been implemented by various researches to solve the single-item dynamic lot size problem. Love [1] introduced the first **dynamic programming formulation for** the Economic Lot-Sizing Problem with Bounded Inventory (ELSB), where inventory levels are constrained by the lower and upper bounds. His model considers both production capacities and storage limitations, which are common in practical applications. It solved in $O(T^3)$ time considering backlogging, time-dependent inventory bounds and piecewise concave production, and storage costs when $T$ is the number of periods in the planning horizon. Toczylowski [21] presented an efficient $O(T^2)$ algorithm for the general single-item dynamic lot-sizing problem with limited inventory levels and nonzero initial and safety stock levels. Loparic et al. [2] derived a dynamic program or the shortest path problem using regeneration intervals to solve a single-item lot-sizing problem with sales constraints and lower bounds on safety stocks. The limitation of this research is that, in practice, safety stock levels often fluctuate. Sedano-Noda et al. [12] introduced an $O(T \log T)$ greedy algorithm to provide optimal policies assuming reorder and linear holding costs without setup costs or backlogging. However, the limitation of this research is that it considers only zero setup costs, which is not practical. Liu and Tu [13] proposed the capacity production-planning (CPP) problem where the production quantity was limited by inventory capacity and stockout. This problem occurs in petrochemical and glass manufacturing, crude oil refining, and food processing. They applied a minimum-cost flow algorithm to construct the network. By applying standard successive-shortest-path methods, they achieved an overall time complexity O($T^3$). Önal et al. [5] modified dynamic programming procedure that restores optimality for the general bounded-inventory lot- sizing problem in $O(T^2)$. However, this study did not include any computational experiments to validate the practical performance of the corrected method. Chu and Chu [14] proposed the dynamic programming approach for the inventory-bounded outsourcing and inventory-bounded outsourcing models. These models execute overall complexity time with $O(T^2 \log T)$ and $O(T^2)$, respectively. The limitation of this approach is impractical for planning horizons longer than a few dozen periods. Hwang and Heuvel [15] presented the $O(T^2)$ algorithm based on dynamic programming and the Monge property for solving a dynamic lot-sizing problem with backlogging and inventory bounds when general production and inventory cost structures are concave. In addition, they introduced the $O(T \log T)$ algorithm using the points-approach and a geometric technique for fixed-charge cost structure as well as the $O(T)$ algorithm using a line-segments approach, including a geometric technique for the fixed-charge cost structure without speculative motives. However,their algorithm does not provide an optimal solution for the ULS-IB problem. Hwang et al. [16] developed the first polynomial-time $O(T^4)$ dynamic-programming algorithm to solve the single-item deterministic Economic Lot-Sizing problem with lost sales and bounded inventory (ELS-LB), under the assumption that each period's inventory capacity is fixed. A drawback of their DP algorithm is that it requires a long running time and a large amount of memory to execute.Boonphakdee and Charnsethikul [23] developed a network- flow based on the dynamic programming approach to solve the single-item uncapacitated lot-sizing problem. In this study, the authors introduce their DP algorithm to generate the initial replenishment plan. Atamtürk and Küçükyavuz [3] proposed a linear programming formulation that achieves tighter relaxations for the single-item lot-sizing problem with inventory bounds and fixed costs.. Gutiérrez et al.[20] extended the classical Wagner–Whitin model by time-varying storage capacities and allowing backlogging. They developed a dynamic programming algorithm with time complexity of $O(T^3)$, where $T$ is the number of periods in the planning horizon Their algorithm applies only when both the production cost and the holding or stockout cost functions are concave. Guan and Liu [4] introduced two stochastic models for the single-item lot-sizing problem under uncertainty including inventory-bound only and the other both inventory-bound and constant order-capacity constraints. They developed dynamic programming algorithms from them with the time complexity $O(T^2)$ and O($T^2 n \mathrm{Log} T$), respectivity, where $T$ is the number of time periods and $n$ is the number of possible order capacities. However, stochastic DP requires complete and precise probability distributions of demand for every period. Chu et al. [6] proposed a single-item dynamic lot-sizing model integrating backlogging, outsourcing, and limited inventory. They developed a dynamic programming algorithm that solves the lot-sizing problem in polynomial time with $O(T^3)$ time complexity, where $T$ is the number of periods in the planning horizon. As a result, their algorithm cannot support concave setup or volume-discount cost structures. Brahimi et al.**[8]** introduce**d** the Two-Level dynamic Lot-Sizing Problem with Bounded Inventory (2LLSP-BI), integrating raw-material procurement and finished-product production planning under finite warehouse capacity constraints**.** They introduced a new Lagrangian-relaxation heuristic which decomposes 2LLSP-BI into N single-item lot-sizing subproblems. Each subproblem is solved by a dynamic programming. The time per Lagrangian iteration is $O(N{\cdot}T^2{+}T{\cdot}I_{max})$, where $I_{max}$ is the number of the capacity bounds. The raw-material inventory has a single static bound, whereas finished-goods storage is unbounded.Finally, Di Summa and Wolsey [24] studied a mixed-integer program that provides a new convex-hull characterization for the single-item discrete lot-sizing problem with a variable upper bound on the initial stock. However, this formulation is in general too large to be practically useful.

In practice, it is hard to handle only a single item in raw-material storage or on the production line. Consequently, managing multiple items can be quite complex. It is difficult to keep each item

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 3 June 2025 doi:10.20944/preprints202506.0178.v1

and to balance holding costs against ordering costs. Heuristic algorithms are commonly used to solve the multi-item dynamic lot-sizing problem. Many researchers have been interested in creating the heuristic algorithm for solving the multi-item uncapacitated lot-sizing problem with inventory bound (MULSP-IB) due to the practical problem in the real world. This problem is like the multi-item capacitated lot-sizing problem (MCLSP), where the items allocate to a machine with a production capacity constraint. The MULSP-IB only has the limitation of on-hand inventory. Dixon and Poh [27] proposed the smoothing approach. They developed the push and pull operations if all weight or volume of inventory is maintained at more than the storage capacity. For the push operation, replenishment in the existing period $t$ is moved to the consecutive period $t$+1. On the other hand, the pull operation postpones the replenishment in a backward direction from the existing period $t$ to a previous period $t$-1 so that all the weight or volume of inventory is maintained at less than the storage capacity. This procedure runs in $O(n \cdot T)$. The authors apply the principle of this smoothing method to implement the push and pull procedures. The push procedure postpones replenishment from the current period to the next period, whereas the pull procedure shifts it back to the previous period. Moreover, the authors compare this heuristic's performance against other heuristics. Park [25] studied the two systems together. He presented the solutions of integrated production and distribution planning and investigated the effectiveness of their integration in a multi-plant, multi-retailer, multi-item, and multi-period logistic environment. Additionally, he introduced the optimization models and a heuristic solution for both integrated and decoupled planning. Akbalik et al. [7] improved the dynamic programming running in $O\ (2^{n}T^{n+1})$ time with a polynomial growth rate if the number of items ($n$) is fixed to solve the MULSP-IB. However, the $T^{n+1}$ factor makes the algorithm impractical for larger multi-item instances. Gutiérrez et al. [17] extended the smoothing technique of Dixon and Poh[27]. explored the multi-item dynamic lot-sizing problem with storage capacities or inventory bounds. The problem has been presented to bound applying the weight of inventory in a previous period plus the weight of replenishment in the current period. Its average solution is over around 5% of the solution computed by CPLEX. The authors also compare its performance against the proposed heuristics. Melo and Ribeiro [9] presented the multi-item uncapacitated lot-sizing problem with shared inventory bounds. They developed two MIP-based heuristics : a rounding scheme for generating the feasible solution and a relax-and-fix heuristic for improving the solution. These MIP-based heuristics yield only near-optimal solutions on average within about 2-4 % of the true optimum. Witt[10] introduced a mathematical model for the Multi-Level Capacitated Lot-Sizing Problem with Inventory Constraints (MLCLSP-IC). His model integrates capacity bounds at each level of a product Bill-of-Materials and explicit work-in-process inventory limits. Although it is multi-level, the approach supports only a single item per level; it cannot accommodate product families that share capacities or complex Bills of Materials with alternative subassemblies. Finally, heuristics and MIP-based heuristics for solving MULSP-IB are further explained in Table 1 as below.

**Table 1.** A systematic overview of heuristics and MIP-based approaches**.**

| References | Model. | Stor. | Algo. | Cap. |
|---|---|---|---|---|
| Dixon and Poh[27] | I&P | BegInv. | DP.&Heu. | Limit.Inv. |
| Park[25] | I&P&R&V | EndInv. | LR.Heu. | Limit.Inv&P&R |
| Akbalik et al. [7] | I&P | EndInv. | DP. | Limit.Inv. |
| Gutiérrez et al. [17] | I&P | BegInv. | DP.&Heu. | Limit.Inv. |
| Melo and Ribeiro [9] | I&P&PT&V | EndInv. | LP.R.Heu.&Heu. | Limit.Inv. |
| Witt[10] | I&P | EndInv. | Heu. | Limit.Inv. |

Abbreviations, Model. Model formulation I&P = inventory and production constraints; I&P&R&V = inventory,production,retailer and vehicle ; &P&PT&V = inventory,production,production time and vehicle constraints constraints; Stor. Storage capacity EndInv.= ending inventory BegInv.= sum of beginning inventory and replenishment Algo. Algorithm DP.= Dynamic programming DP.&Heu.= Dynamic programming and heuristic. LR.Heu.= Lagrangian-relaxation-based heuristic ; LP.Heu.&Heu = Linear programmingrelaxation-based heuristic and heuristic Heu. = Heuristic; Cap. CapacityLimit.Inv = Limited inventory constraint Limit.Inv&P&R= Limited inventory,plant and vehicle.

In Table 1, the researchers considered only limited storage capacity and did not specifically consider on the case of tight capacity constraints. Therefore, the authors are interested in near-minimal storage capacity constraints, which occur in the automotive assembly industry [26].

This study proposes a new heuristic for computing the approximate replenishment plan for multiple items in each period under storage capacity constraints. The novelty of this study considers the proposed procedure, which can be executed effectively under near-minimal or worst-case storage capacity constraint. Storage capacity is defined as the sum of the weight (or volume) of all items carried over from the previous period and the weight (or volume) of all replenishments in the current period. The authors introduce a dynamic programming approach based on network flow [23] to determine the replenishment plan for each item and propose two methods for computing a multi-item replenishment schedule under storage-capacity constraints. The proposed heuristics consist of a push method and a pull method. The push method postpones replenishments forward from period $t$ to period $t + k$ whenever the inventory level in period t exceeds the limited capacity, and repeats this until the inventory in every period does not exceed its capacity. Then the pull method refines the plan by moving replenishments backward from period $t$ to earlier periods

The second section describes the mathematical formulation of MULSP-IB. The proposed heuristic is effective for solving this problem by the modified push and pull operation. It improves comparison with the push operation by Gutiérrez et al. shown in Section 3, and this heuristic is examined by Gutiérrez's example in Section 4. In Section 5, the randomly generated data has been implemented for solving the different problem sizes. The performance of this proposed heuristic is compared with the result of Gutiérrez et al.'s algorithm and the smoothing method. For the robustness condition, the worst-case condition is analyzed, which is the near-minimal storage capacity. In the sensitivity analysis, varying the storage capacities affects the total cost and inventory levels. Finally, conclusions are provided in Section 6.

## 2. Problem Description

### 2.1. Problem Statement

The problem of MULSP-IB can be stated as this: Each demand $d_{it}$ must be partly or entirely replenished at a period $t$ by inventory . In this study, consider that demands and inventory bounds are time-varying, and the total actual weight/volume of inventory is not over storage capacity. The problem is to find the periods and the number of raw materials delivered within these periods. The objective is to construct a replenished plan such that the total cost is minimized.

### 2.2. Problem Assumptions

Assumptions are provided to define certain parameters and decision variables as follows.

Assumption 1. Storage capacity is the upper bound of stock.

Assumption 2. A replenished item is stored first before it is used to satisfy demand. This means that the inventory at the beginning of a period plus the replenish-ment is the actual inventory at the end of the periods.

Assumption 3. The sum of demand is not over storage capacity in all periods, and the demands are satisfied at the end of the period.

Assumption 4. Number of items is independent of the quantity of demand and the number of procurement planning horizon.

Assumption 5. Initial inventory in the first period and ending inventory in the last period are zero.

Assumption 6. Backlogging is not allowed.

Assumption 7. There is no consideration of lead time.

*2.3. Decision Variables and Parameters*

Explanation of decision variables and parameters can be noted in the following.

Indices $i$ the number of items indexed from 1 to $N$ $t$ the number of periods indexed from 1 to $T$
Parameters $d_{it}$ the demand of item $i$ at period $t$
$D_{i,t}$ the accumulative demand of item $i$ from period $t$ to period $T$
$f_{i,t}$ the fixed ordering cost of item $i$ at period $t$
$h_{i,t}$ the holding cost of item $i$ at period $t$
$p_{i,t}$ the cost of procuring raw materials of item $i$ at period $t$
$I_{i,t}$ the inventory level of item $i$ at period $t$
$w_i$ the unit weight of item $i$
$U_t$ the storage capacity at period $t$
Decision variables $x_{i,t}$ the procurement quantity of item $i$ at period $t$
$Y_{i,t}$ if the replenishment of item $i$ at period $t$ occurs, $Y_{it}$ is 1. Otherwise, $Y_{it}$ is 0.

*2.4 Mathematical Model*

his study proposes the model by Gutiérrez et al. [17], which states the MIP formulation as follows:

$$\min \sum_{i=1}^{N}\sum_{t=1}^{T} f_{i,t}Y_{i,t} + p_{i,t}x_{i,t} + h_{i,t}I_{i,t} \tag{1}$$

$$\text{s.t} \quad I_{i,t-1} - I_{i,t} + x_{i,t} = d_{i,t}, \qquad i = 1,\ldots,N, t = 1,\ldots,T \tag{2}$$

$$\sum_{i=1}^{N} w_i(I_{i,t-1} + x_{i,t}) \le U_t, \qquad t = 1,\ldots,T \tag{3}$$

$$x_{i,t} \le Y_{i,t}D_{i,t}, \qquad i = 1,\ldots,N, t = 1,\ldots,T \tag{4}$$

$$I_{i,0} = I_{N,T} = 0, \qquad i = 1,\ldots,N \tag{5}$$

$$x_{i,t}, I_{i,t} \in \square_0 = \square \cup 0, \qquad i = 1,\ldots,N, t = 1,\ldots,T \tag{6}$$

$$Y_{i,t} \in \{0,1\}, \qquad i = 1,\ldots,N, t = 1,\ldots,T \tag{7}$$

The objective function of MULSP-IB minimizes the sum of ordering, purchased and holding costs in constraint (1). Constraint (2) is the balance of the inventory equation. Each purchased unit and inventory unit at the beginning of the period are always kept first, before moving to production/customers following its demand in constraint (3). Constraint (4) link the purchased variables with the binary variables $Y_{it}$ and accumulated demand ( $D_{it} = \sum_{k=t}^{T} d_{ik}$ ) of item $i$ from periods $t$ to $T$ . Constraint (5) is the initial inventory in the first period and the ending inventory in the last period are zeroes. Constraint (6) defines the purchased quantity and inventory, which are not negative. In constraint (7), if replenishment occurs at any period, $Y_{it}$ is 1. Otherwise, $Y_{it}$ is 0.

## 3. The Proposed Heuristic

The push and pull strategies of Dixon and Poh [27] consider that excess storage capacity has occurred. This is reduced by moving a replenishment quantity from the existing period $t$ to $t$+1 when the sum of both inventory and replenishment of all items ($SIR_{allitems}$) for the existing period is over the storage capacity, called the push operation. On the other hand, a replenishment quantity from period $t$ is returned to a previous period when $SIR_{al\ litems}$ is less than the storage capacity, called the pull strategy. The push method focuses on reducing $SIR_{allitems}$ in the existing period until success only. Therefore, each iteration enables the reduction of $SIR_{allitems}$ prominently, while ordering cost will grow as necessary, causing total inventory cost to expand as necessary. Furthermore, Gutiérrez et al.'s he ristic extend the push strategy in so far that a replenishment quantity can be moved from any period *t* to *t*+*k*, $k \in (1,...,T-t)$. For any iteration, Gutiérrez et al.'s heuristic possibly moves a replenishment quantity from a non-existing period to the next period, resulting in$SIR_{allitems}$ in the existing period not also reducing. However, ordering costs still expanding is unnecessary. This study also applies Dixon and Poh's approach to both the push and pull strategies. For the push strategy, consider a replenishment item in the existing period, which has the maximum sum of inventory and the replenishment quantity ($SIR_{max}$). Consider an item of $SIR_{max}$ condition, called $iSIR_{max}$ that has an inventory cost of zero at period *t*+*k*, $k \in (1,...,T-t)$ called on $t_{zero}$. Therefore, a replenishment quantity is equal to its demand in the existing period, then it moves so as to add the original replenished quantity at period $t_{zero}$. After that, the replenishment quantity of the existing item is balanced for all periods. It runs repeatedly until all $SIR_{allitems}$ are less than storage capacities. This procedure is called the push method.

For the pull strategy, find the periods ($t_{min}$ and $t_{max}$) **with** the minimum and maxi mum differences between $SIR_{allitems}$ and their storage capacities ($U$)**, called** $SIR_{allitems}\_U_{min}$ and $SIR_{allitems}\_U_{max}$. If the index of period $t_{min}$ is higher than the index of period $t_{max}$, consider a $SIR_{min}$ item called $iSIR_{min}$ for period $t_{min}$. Return the replenished quantity, which is the demand of $iSIR_{min}$ of period $t_{min}$, to add the original replenished quantity of the same item at period $t_{max}$. Then, calculate all $SIR_{allitems}$ again. If a new $SIR_{allitems}$ for period $t_{max}$ is still greater than storage capacity at that period, find a new item with $SIR_{max}$ called $iSIR_{max}$. Next, determine the period $t_{zero}$ on $iSIR_{max}$ and calculate the round-up of the new $SIR_{allitems}\_U$ for period $t_{max}$ divided by its weight, called $SIR_{allitems}\_U_{roundup}$. Move the replenished quantity, $SIR_{allitems}\_U_{roundup}$ for period $t_{max}$, to period $t_{zero}$. Finally, balance the replenished plan again. This procedure is called the pull method. It can effectively improve the solution generated by the push method. Next, determine the period $t_{zero}$ on $iSIR_{max}$ and calculate the round-up of the new $SIR_{allitems}\_U$ for period $t_{max}$ divided by its weight, called $SIR_{allitems}\_U_{roundup}$. Move the replenished quantity, $SIR_{allitems}\_U_{roundup}$ for period $t_{max}$, to period $t_{zero}$. Finally, balance the replenished plan again. This procedure is called the pull method. It can effectively improve the solution generated by the push method. This study proposes a new procedure for solving MULSP-IB. It has a property suggesting that the sum of both the inventory and the replenished quantity for all items ($SIR_{allitems}$) agree with less storage capacity, satisfying all demands for the existing period or equal storage capacity for that period. This property is presented to clarify the logical flow of arguments as follows:

**Theorem 1.** *If t is a period such that* $x_{i,t} > 0$ *for some items i satisfying* $I_{i,t-1} + x_{i,t} = D_{i,t} - D_{i,t+k}$, $k \in (1,...,T-t)$, *then* $\sum_{i=1}^{N} (I_{i,t-1} + x_{i,t}) < U_t$ .

**Proof of Theorem 1.** For a contradiction method, the assumption is $\sum_{i=1}^{N} (I_{i,t-1} + x_{i,t}) \geq U_t$ .

Given $I_{i,t-1} + x_{i,t} = D_{i,t} - D_{i,k}$. It implies that the total inventory is sufficient to satisfy or exceed the storage limitation. However, some $x_{i,t} > 0$ and $I_{i,t-1} + x_{i,t} = D_{i,t} - D_{i,t+k}$ are used to satisfy the demands such that no additional replenishments can exceed the storage limitation. Therefore, there is no additional inventory available to satisfy or exceed the limitation.

**Theorem 2.** *If t is a period such that* $x_{i,t} > 0$ *for some items i satisfying* $I_{i,t-1} + x_{i,t} \neq D_{i,t} - D_{i,t+k}$, $k \in (1,...,T-t)$, *then* $\sum_{i=1}^{N} (I_{i,t-1} + x_{i,t}) = U_t$ .

**Proof of Theorem 2.** For a contradiction method, let us study a replenishment ( $x_{i,t} > 0$ ) that there is an item *i*, for which the assumption is $\sum_{i=1}^{N} (I_{i,t-1} + x_{i,t}) \neq U_t$ . In the case of $\sum_{i=1}^{N} (I_{i,t-1} + x_{i,t}) < U_t$ , given $I_{i,t-1} + x_{i,t} \neq D_{i,t} - D_{i,t+k}$, the inventory level and extended replenishments ( $I_{i,t-1} + x_{i,t}$ ) are not sufficient to satisfy the sum of the consecutive demand ( $D_{i,t} - D_{i,t+k}$ ). Further, in the case of $\sum_{i=1}^{N}(I_{i,t-1} + x_{i,t}) > U_t$ , there is additional inventory to be over the storage capacity. However, the given condition $I_{i,t-1} + x_{i,t} \neq D_{i,t} - D_{i,k}$ does not satisfy the demand. Therefore, the sum of inventory and replenished quantity cannot exceed the storage capacity. To satisfy demand, the sum of the inventory and replenished quantity of all items in each period must either be less than or equal to the storage capacity explained by Theorems 1 and 2, respectively.

### *3.1. The Push Method*

The objective of this method is to seek the approximate solution of MULSP-IB. The procedure for this method can be explained by the pseudo-algorithms as follows:

```
procedure InitialSolution(N, T, U, d, h, f)
    Input:
        N ← number of items
        T ← number of periods
        U[1..T] ← inventory bounds per period (network-flow based)
        d[1..N][1..T] ← demand of item i in period t
        h[1..N][1..T] ← holding cost of item i in period t
        f[1..N][1..T] ← ordering cost of item i in period t
    Output:
        x[1..N][1..T] ← initial replenishment plan
    for i ← 1 to N do
        x[i][1..T] ← NetworkFlowDP(i, U, d[i], h[i], f[i])
    end for
    return x
end procedure
procedure PushAlgorithm(N, T, U, d, h, f, x)
    Input:
        N ← number of items
        T ← number of periods
        U[1..T] ← storage capacity per period
        d[1..N][1..T] ← demand of item i in period t
        x[1..N][1..T] ← initial replenishment plan
    Output:
        x[1..N][1..T], TotalCost
    for t ← 1 to T do
        // Compute SIR for each item at period t
        for i ← 1 to N do
            if t == 1 then
                inv[i] ← x[i][1] - d[i][1]
            else
                inv[i] ← inv_prev[i] + x[i][t] - d[i][t]
            end if
            tailDemand ← 0
            for k ← t to T do
                tailDemand ← tailDemand + d[i][k]
            end for
            SIR[i] ← inv[i] + tailDemand
            inv_prev[i] ← inv[i]
        end for
        SIRall ← Σ_{i=1..N} SIR[i]
```

```
41:        slack ← SIRall - U[t]
42:        while slack > 0 do
43:            iMax ← argmax_{i=1..N} SIR[i]
44:            Δ ← d[iMax][t]
45:            x[iMax][t] ← x[iMax][t] - Δ
46:            if t < T then
47:                x[iMax][t+1] ← x[iMax][t+1] + Δ
48:            end if
49:            // Recompute inventory and SIR
50:            SIRall ← 0
51:            for i ← 1 to N do
52:                if t == 1 then
53:                    inv[i] ← x[i][1] - d[i][1]
54:                else
55:                    inv[i] ← inv_prev[i] + x[i][t] - d[i][t]
56:                end if
57:                tailDemand ← 0
58:                for k ← t to T do
59:                    tailDemand ← tailDemand + d[i][k]
60:                end for
61:                SIR[i] ← inv[i] + tailDemand
62:                inv_prev[i] ← inv[i]
63:                SIRall ← SIRall + SIR[i]
64:            end for
65:            slack ← SIRall - U[t]
66:        end while
67:    end for
68:    // Calculate total cost
69:    TotalCost ← 0
70:    for i ← 1 to N do
71:        for t ← 1 to T do
72:            if x[i][t] > 0 then
73:                TotalCost ← TotalCost + f[i][t]
74:            end if
75:            TotalCost ← TotalCost + inv_prev[i] * h[i][t]
76:        end for
77:    end for
78:    return x, TotalCost
79: end procedure
```

In line no. 16, pseudo code of dynamic programming based on network flow can be shown in Appendix A.1. The logical sequence including selection rules of the push method can be explained in Figure 1.

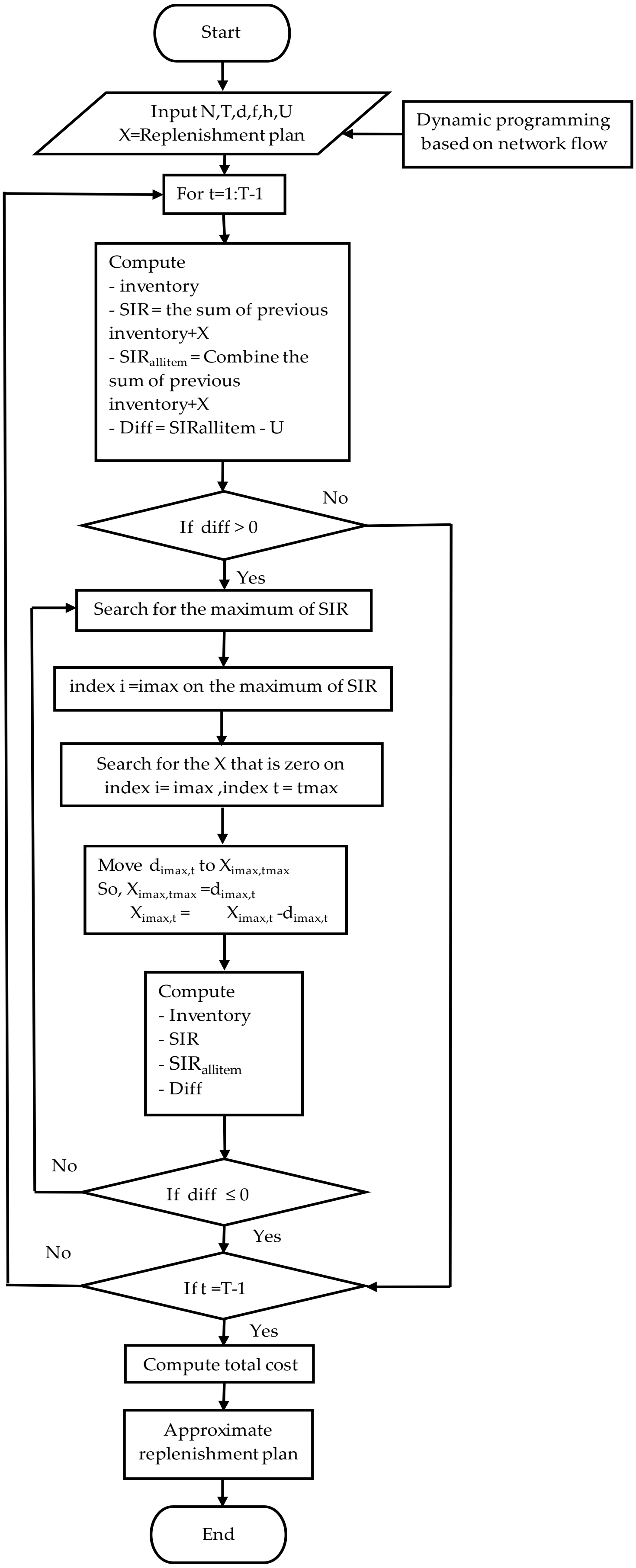
Start
Input N,T,d,f,h,U
X=Replenishment plan
Dynamic programming based on network flow
For t=1:T-1
Compute
- inventory
- SIR = the sum of previous inventory+X
- $SIR_{allitem}$ = Combine the sum of previous inventory+X
- Diff = SIRallitem - U
If diff > 0
No
Yes
Search for the maximum of SIR
index i =imax on the maximum of SIR
Search for the X that is zero on index i= imax ,index t = tmax
Move $d_{imax,t}$ to $X_{imax,tmax}$
So, $X_{imax,tmax}$ =$d_{imax,t}$
$X_{imax,t}$ = $X_{imax,t}$ -$d_{imax,t}$
Compute
- Inventory
- SIR
- $SIR_{allitem}$
- Diff
No
If diff ≤ 0
Yes
No
If t =T-1
Yes
Compute total cost
Approximate replenishment plan
End

**Figure 1.** Flow chart of the push method.

In Fig. 1, the algorithm stops computing if the difference (diff) between the sum of previous inventory plus replenishment quantity (SIR$_{\text{alitem}}$) and the storage capacity (U) for each period is less than or equal to zero. This selection rule determines whether to perform the next step of the algorithm. From the pseudo code, dynamic programming based on network flow algorithm has time complexity $O(NTD^2_{\max})$ that depends on $D_{\max}$, the largest demand value across all items *i* and all time periods *t*. Therefore, large demand values significantly influence the running time when solving large-scale problems. The push algorithm has time complexity $O(N \cdot T \cdot W)$ where W is number of iterations of the while loop per period. If the total inventory exceeds the storage capacity by a large amount, many iterations (W) will be needed to reduce the overcapacity ;otherwise, iteration stops. Therefore, in large-scale problems, inventory overcapacity is likely, causing the proposed algorithm to have a high running time.

The replenishment plan generated by the push method can be improved to reduce the total cost using the pull method, which will be discussed in the next section.

### 3.2. The Pull Method

The objective of this method is to improve the replenishment plan, which is computed by the push method. Some replenishments may be returned from the existing period to the previous period so that the sum of inventory and replenished quantity of all items for the previous period is added to equal its storage capacity. The procedure for the pull method shows the pseudocode of algorithms as follows

```
PROCEDURE Pull method
INPUT:
    X [i, t]← initial replenished quantity for item i in period t
    computed by the push method
    demand[i, t]← demand of item i in period t
    SIR[i, t] ← the sum of previous inventory and replenished
    quantity of item i in period t
    U← storage capacity for period t
    w[i, t] ←weight (or size) of item i in period t
    T←total number of periods
OUTPUT:
    replenishment plan[i, t] ←adjusted replenished quantities
    inventoryCost ←updated total inventory cost
// Compute initial aggregate SIR per period
FOR t ← 1 TO T DO
    SIRallitems[t] ← $\Sigma_i$ SIR[i, t]
    SIRallitems_SC[t] ← $\Sigma_i$ (SIR[i, t] · w[i, t])
END FOR
// Find periods with min/max aggregate SC usage
Tmin ← arg $\min_t$ SIRallitems_SC[t]
Tmax ← arg $\max_t$ SIRallitems_SC[t]
// Loop until the lightest-loaded period index is not after the heaviest
IF Tmax< Tmin DO
    // 1) Move the smallest-rate demand from Tmin to Tmax
    i_min ← arg $\min_i$ SIR[i, Tmin]
    qty ← demand[i_min, Tmin]
    X[i_min, Tmax] ← X[i_min, Tmin] + qty
    X[i_min, Tmin] ← X[i_min, Tmin] - qty
    // 2) Rebalance and update all metrics
```

```
        CALL UpdateMetrics()
        // 3) If Tmax still exceeds capacity, boost its replenishment
        IF SIRallitems_SC[Tmax] > U THEN
            i_max ← $\arg\max_i$ SIR[i, Tmax]
            adjustment ← [SIRallitems_SC[Tmax] / w[i_max, Tmax]]
            X[i_max, Tmax] ← X[i_max, Tmax] + adjustment
            CALL UpdateMetrics()
        END IF
        // 4) Recompute Tmin and Tmax for next iteration
        Tmin ← $\arg\min_t$ SIRallitems_U[t]
        Tmax ← $\arg\max_t$ SIRallitems_U[t]
    END IF
    RETURN (X, inventoryCost)
END PROCEDURE
// Subroutine to recalculate inventory levels, SIR, aggregate metrics, and cost
PROCEDURE UpdateMetrics
    FOR t ← 1 TO T DO
        FOR each item i DO
            // Recompute SIR[i, t] based on new replenishment qty and demand
            SIR[i, t] ← ComputeSIR(X[i, t], demand[i, t])
        END FOR
        SIRallitems[t] ← $\Sigma_i$ SIR[i, t]
        SIRallitems_U[t] ← $\Sigma_i$ (SIR[i, t] · w[i, t])
    END FOR
    inventoryCost ← ComputeTotalCost(X, demand, holdingCosts, orderingCosts)
END PROCEDURE
```

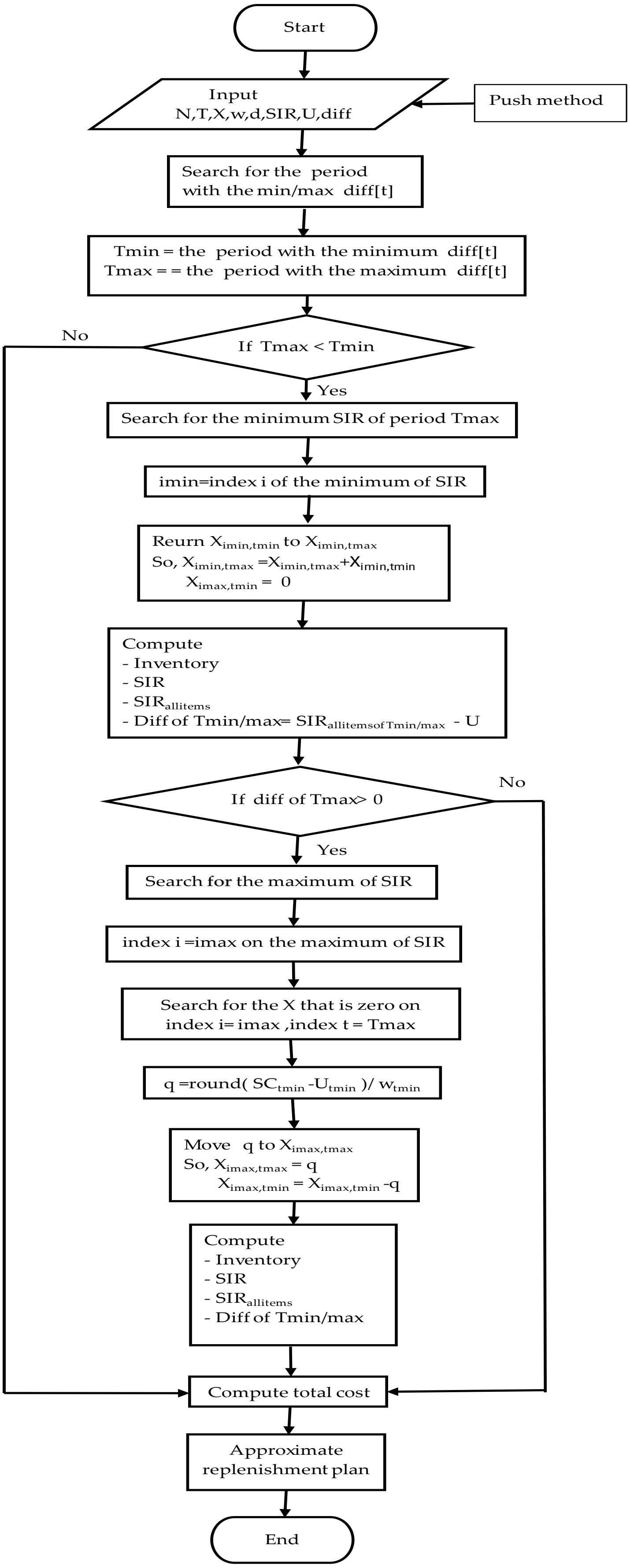
Start
Input
N,T,X,w,d,SIR,U,diff
Push method
Search for the period with the min/max diff[t]
Tmin = the period with the minimum diff[t]
Tmax = = the period with the maximum diff[t]
No
If Tmax < Tmin
Yes
Search for the minimum SIR of period Tmax
imin=index i of the minimum of SIR
Reurn $X_{imin,tmin}$ to $X_{imin,tmax}$
So, $X_{imin,tmax}$ =$X_{imin,tmax}$+$X_{imin,tmin}$
$X_{imax,tmin}$ = 0
Compute
- Inventory
- SIR
- $SIR_{allitems}$
- Diff of Tmin/max= $SIR_{allitemsofTmin/max}$ - U
No
If diff of Tmax> 0
Yes
Search for the maximum of SIR
index i =imax on the maximum of SIR
Search for the X that is zero on index i= imax ,index t = Tmax
q =round( $SC_{tmin}$ -$U_{tmin}$ )/ $w_{tmin}$
Move q to $X_{imax,tmax}$
So, $X_{imax,tmax}$ = q
$X_{imax,tmin}$ = $X_{imax,tmin}$ -q
Compute
- Inventory
- SIR
- $SIR_{allitems}$
- Diff of Tmin/max
Compute total cost
Approximate replenishment plan
End

**Figure 2.** Flow chart of the pull method.

In Fig. 2, the algorithm proceeds to compute an improved solution if the index of the period with the minimum *diff* (*T*min) is greater than the index of the period with the maximum *diff* (*T*max). This selection rule determines whether to perform the next step of the algorithm. The push algorithm has time complexity $O(N^2T^2)$ when $L \le N \cdot T$ , where $L$ is the number of loop. The total running time depends on how many iterations **L** the algorithm performs to balance the load between periods. If **L** grows large (close to N·T in the worst case), running time can grow quadratically. To explain the heuristic algorithm, the authors use a numerical example to demonstrate it in the next section.

## 4. A Numerical Example

This study presents the simple example by Gutiérrez et al. [17] to explain the procedure for both proposed methods. Data from this example is shown in Table 2.

**Table 2.** A simple example by Gutiérrez et al. [17].

| **Periods *t*** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|
| $U_t$ | 756 | 673 | 633 | 758 | 608 |
| Item 1,$w_1$=1 | | | | | |
| $d_{1,t}$ | 115 | 114 | 96 | 106 | 136 |
| $D_{1,t}$ | 567 | 452 | 338 | 242 | 136 |
| $f_{1,t}$ | 595 | 100 | 969 | 240 | 945 |
| $p_{1,t}$ | 4 | 7 | 9 | 10 | 4 |
| $h_{1,t}$ | 1 | 1 | 1 | 1 | 1 |
| Item 2,$w_1$=4 | | | | | |
| $d_{2,,t}$ | 87 | 52 | 111 | 142 | 118 |
| $D_{2,t}$ | 510 | 423 | 371 | 260 | 118 |
| $f_{2,t}$ | 255 | 696 | 125 | 637 | 249 |
| $p_{2,t}$ | 3 | 3 | 0 | 8 | 4 |
| $h_{2,t}$ | 1 | 1 | 1 | 1 | 1 |

### 4.1. The Initial Solution

The optimal replenished plan for each item, which is independent, can be solved by the network flow based on a dynamic programming approach [22]. Thus, the optimal replenished plan is the initial solution for this example, as shown in Table 3.

**Table 3.** Initial solution for each item.

| **Periods *t* Item** | | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|---|
| Replenished plan | | 567 | 0 | 0 | 0 | 0 |
| | 1 | | | | | |
| | | 139 | 0 | 371 | 0 | 0 |
| | 2 | | | | | |
| Inventory | | 452 | 338 | 242 | 136 | 0 |
| | 1 | | | | | |
| | | 52 | 0 | 260 | 118 | 0 |
| | 2 | | | | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| Inventory cost | 1 | 4x567+1x452+595 =3,315 | 338 | 242 | 136 | 0 |
| | 2 | 3x139+1x52+255 =724 | 0 | 385 | 118 | 0 |
| Sum of inventory and replenishment (*SIR*) | 1 | 567x1=567 | 452 | 338 | 242 | 136 |
| | 2 | 139x4=556 | 208 | 1,484 | 1,040 | 472 |
| $SIR_{allitems}$ | | 1,123 | 660 | 1,822 | 1,282 | 608 |
| *SC* | | 756 | 673 | 633 | 758 | 608 |
| $SIR_{allitems}\_U$ | | +367 | -13 | 1,189 | 524 | 0 |

### *4.2. Solution of the Push Method*

From Table 3, $SIR_{allitems}$ values of periods 1, 3, and 4 are positive and their $SIR_{allitems}$ values are certainly excess. Next, the replenished quantity for period one is moved as follows.

Period 1

Iteration 1

1. Select item 1, which has the maximum *SIR* ($SIR_{max}$) of 567, and find period $t_{zero}$ = 5 on item 1, which has an inventory cost of zero.
2. Insert the replenished quantity, which equals the demand of item 1 for period $t_{zero}$ =
3. 136 units, on a replenished plan on item 1 for period $t_{zero}$. For balance demand, de
4. crease the replenishment of item 1 for period 1 to 567-136 =431 units.
5. Balance a replenished plan and update inventory, *SIR*,$SIR_{allitems}$, $SIR_{allitems}\_U$, and total inventory cost (see Table 4).
6. $SIR_{allitems}\_U$ for period 1 is reduced to 987-756 = +231. Afterward, go to steps 1-4.

Iteration 2

1. Select item 2, which has $SIR_{max}$ of 556, and find period $t_{zero}$ = 2 on item 2, which has an inventory cost of zero.
2. Insert the replenished quantity, which equals the demand of item 2 for period $t_{zero}$ = 52 units, on a replenished plan on item 2 for period $t_{zero}$. For balance demand,
3. decrease the replenishment of item 2 for period 1 to 139-52 =87 units.
4. Balance a replenished plan and update inventory, *SIR*, $SIR_{allitems}$, $SIR_{allitems}\_U$, and
5. total inventory cost (see Table 5).
6. $SIR_{allitems}\_U$ for period 2 is still reduced to 779-756 = +23. Then, go to steps 1- 4.

Iteration 3

1. Select item 1, which has $SIR_{max}$ of 431, and find period $t_{zero}$ = 4 on item 1, which has an inventory cost of zero.

2. Insert the replenished quantity, which equals the demand of item 2 for period $t_{zero}$= 106 units, on a replenished plan on item 1 for period $t_{zero}$. For balance demand, decrease the replenishment of item 1 for period 1 to 431-106 = 325 units.
3. Balance a replenished plan and update inventory, *SIR*,$SIR_{allitems}$, $SIR_{allitems}$_*U*, and total
4. inventory cost (see Table 6).
5. $SIR_{allitems}$_*U* for period 1 is reduced to 673-756 = -83. Stop the iteration and select period 3, which has the $SIR_{allitems}$_*U* value of +947. Proceed to steps 1-4 for period 3.

Period 3

Iteration 1

1. Select item 2, which has $SIR_{max}$ of 1,484, and find period $t_{zero}$ = 5 on item 2, which has
2. an inventory cost of zero.
3. Insert the replenished quantity, which equals the demand of item 2 for period $t_{zero}$=118 units, on a replenished plan on item 2 at period $t_{zero}$. For balance demand, decrease the replenishment of item 2 for period 3 to 371-118 = 253 units.
4. Balance a replenished plan and update inventory, *SIR*,$SIR_{allitems}$, $SIR_{allitems}$_*U*,
5. and total inventory cost (see Table 7).
6. $SIR_{allitems}$_*U* for period 3 is still reduced to 1,108-633 = +475. Afterward, go to steps 1- 4.

Iteration 2

1. Select item 2, which has $SIR_{max}$ of 348, and find period $t_{zero}$ = 4 on item 2,
2. which has an inventory cost of zero.
3. Insert the replenishment, which equals the demand of item 2 at period $t_{zero}$ =
4. 142 units on a replenished plan on item 2 at period $t_{zero}$. For balance demand, decrease the replenishment of item 2 at period 3 to 253-142 = 111 units.
5. Balance a replenished plan and update inventory, *SIR*,$SIR_{allitems}$, $SIR_{allitems}$_*U*,
6. and total inventory cost (see Table 8).
7. $SIR_{allitems}$_*U* for period 3 is reduced to 540-633= = -93. Stop the loop at
8. period 3. Then, find the next $SIR_{allitems}$_*U* to be positive. However, all $SIR_{allitems}$_*U* values are negative and zero (-83, -255, -93, -84, 0). Thereafter, stop all iterations of the push method.

**Table 4.** Execution flow of iteration 1 for period 1 using the push method.

| **Periods *t*** | **Item** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|---|
| Replenished plan | 1 | 567-136=431 | 0 | 0 | 0 | 136 |
| | 2 | 139 | 0 | 371 | 0 | 0 |
| Ending inventory | 1 | 316 | 202 | 106 | 0 | 0 |
| | 2 | 52 | 0 | 260 | 118 | 0 |
| Inventory cost | 1 | 4x431+1x316+595 =2,635 | 202 | 106 | 0 | 1,489 |
| | 2 | 724 | 0 | 385 | 118 | 0 |
| Sum of inventory and replenishment (*SIR*) | 1 | 1x431=431 | 316 | 202 | 106 | 136 |
| | 2 | 556 | 208 | 1,484 | 1,040 | 472 |
| $SIR_{allitems}$ | | 987 | 524 | 1,686 | 1,146 | 608 |
| *SC* | | 756 | 673 | 633 | 758 | 608 |
| $SIR_{allitems}$_*U* | | +231 | -149 | +1,053 | +388 | 0 |

**Table 5.** Execution flow of iteration 2 for period 1 using the push method.

| **Periods *t*** | **Item** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|---|
| Replenished plan | 1 | 431 | 0 | 0 | 0 | 136 |
| | 2 | 139-52 =87 | 52 | 371 | 0 | 0 |
| Ending inventory | 1 | 316 | 202 | 106 | 0 | 0 |
| | 2 | 0 | 0 | 260 | 118 | 0 |
| Inventory cost | 1 | 2,635 | 202 | 106 | 0 | 1,489 |
| | 2 | 3x87+255 =516 | 3x52+696 =852 | 3x371+1x260+125 =385 | 118 | 0 |
| Sum of inventory and replenishment (*SIR*) | 1 | 431 | 316 | 202 | 106 | 136 |
| | 2 | 4x87=348 | 4x87=208 | 1,484 | 1,040 | 472 |
| $SIR_{allitems}$ | | 779 | 524 | 1,686 | 1,146 | 608 |
| *SC* | | 756 | 673 | 633 | 758 | 608 |
| $SIR_{allitems}$_*U* | | +23 | -149 | +1,053 | +388 | 0 |

**Table 6.** Execution flow of iteration 3 for period 1 using the push method.

| **Periods *t*** | **Item** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|---|
| Replenished plan | 1 | 431-106=325 | 0 | 0 | 106 | 136 |
| | 2 | 87 | 52 | 371 | 0 | 0 |
| Ending inventory | 1 | 210 | 96 | 0 | 0 | 0 |
| | 2 | 0 | 0 | 260 | 118 | 0 |
| Inventory cost | 1 | 4x325+1x210+595 =2,105 | 96 | 0 | 1,300 | 1,489 |
| | 2 | 516 | 852 | 385 | 118 | 0 |
| Sum of inventory and replenishment (*SIR*) | 1 | 325 | 210 | 96 | 106 | 136 |
| | 2 | 348 | 208 | 1,484 | 1,040 | 472 |
| $SIR_{allitems}$ | | 673 | 418 | 1,580 | 1,146 | 608 |
| *SC* | | 756 | 673 | 633 | 758 | 608 |
| $SIR_{allitems}$_*U* | | -83 | - 255 | +947 | +388 | 0 |

**Table 7.** Execution flow of iteration 1 for period 3 using the push method.

| **Periods *t*** | **Item** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|---|
| Replenished plan | 1 | 325 | 0 | 0 | 106 | 136 |
| | 2 | 87 | 52 | 371-118=253 | 0 | 118 |
| Ending inventory | 1 | 210 | 96 | 0 | 1,300 | 1,489 |
| | 2 | 0 | 0 | 142 | 0 | 0 |
| Inventory cost | 1 | 2,105 | 96 | 0 | 1,300 | 1,489 |
| | 2 | 516 | 852 | 1x142+125=267 | 0 | 721 |
| Sum of inventory and replenishment (*SIR*) | 1 | 325 | 210 | 96 | 106 | 136 |
| | 2 | 348 | 208 | 4x253=1,012 | 568 | 472 |
| $SIR_{allitems}$ | | 673 | 418 | 1,108 | 674 | 608 |
| *SC* | | 756 | 673 | 633 | 758 | 608 |

| | | | | | |
|---|---|---|---|---|---|
| $SIR_{allitems}\_U$ | -83 | -255 | +475 | -84 | 0 |

**Table 8.** Execution flow of iteration 2 for period 3 using the push method**.**

| **Periods *t* Item** | | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|---|
| Replenished plan | 1 | 325 | 0 | 0 | 106 | 136 |
| | 2 | 87 | 52 | 253-142=111 | 142 | 118 |
| Ending inventory | 1 | 210 | 96 | 0 | 0 | 0 |
| | 2 | 0 | 0 | 0 | 0 | 0 |
| Inventory cost | 1 | 2,105 | 96 | 0 | 1,300 | 1,489 |
| | 2 | 516 | 852 | 125 | 8x142+637 = 1,773 | 721 |
| Sum of inventory and replenishment (*SIR*) | 1 | 325 | 210 | 96 | 106 | 136 |
| | 2 | 348 | 208 | 4x111=444 | 568 | 472 |
| $SIR_{allitems}$ | | 673 | 418 | 540 | 674 | 608 |
| *SC* | | 756 | 673 | 633 | 758 | 608 |
| $SIR_{allitems}\_U$ | | -83 | -255 | -93 | -84 | 0 |

From Table 8, the push method can generate a total inventory cost equal to 8,977. However, the GAMS/CPLEX solver can execute this problem with an optimal solution of 8,521. The gap between these solutions is $\left[\frac{8,977-8,521}{8,521}\right]$x100% = 5.35%. This gap is still high. Therefore, this study proposes the pull method to improve the solution.

## *4.3. Solution of the Pull Method*

From Table 8, all $SIR_{allitems}\_U$ are negative or zero. Therefore, the pull method can generate an improved replenished plan. The procedure for this method can beexplained as follows.

1. Search the $SIR_{allitems}\_U_{min}$ and $SIR_{allitems}\_SC_{max}$ to be -255 and -83 for periods 2 and 1 from Table 8. So,the index of both periods is $t_{min}$=2 and $t_{max}$=1. So, $t_{min}$ is more than $t_{max}$.
2. Find the $SIR_{min}$ for period $t_{min}$ to be 208 on item 2 from Table 8. Return the replenishment of item 2 at period 2 to add the original replenished quantity on item 2 for period $t_{max}$=1. Thus, the new amount replenished quantity of item 2 for period 1 is 87+52 = 139 units. For balance demand, the replenished quantity of item 2 for period $t_{min}$ is reduced to zero (see Table 9).
3. Recalculate *SIR*,$SIR_{allitems}$, $SIR_{allitems}\_U$, and total inventory cost (see Table 9).
4. $SIR_{allitems}\_U$ of item 2 for period 1 (= +125) is still a positive number. Thus, search the item with $SIR_{max}$ (= 325) excluding item 2 at period 1, to be 1.
5. For reducing $SIR_{allitems}\_U$ to zero at period 1, the $SIR_{allitems}\_U$ is divided by the weight of item 1 for period 1 (+125 / 1=125) on item 1 at period $t_{min}$ =2 to be 0+125 = 125 units (see Table 9). For balance demand, reduce the replenishment of item 1 in the previous period (*t*=1) to be = 325-125 = 200 units.
6. Recalculate *SIR*,$SIR_{allitems}$, $SIR_{allitems}\_U$, and total inventory cost (see Table 10).
7. $SIR_{allitems}\_U$ of period 1 is zero and $SIR_{allitems}\_U$ of period 2 ($t_{min}$) is still -255 (see Table 10), which is the same as Table 9.
8. Find the $SIR_{allitems}\_U_{min}$ and $SIR_{allitems}\_U_{max}$ to be -255 and -84 in periods 2 and 3 from Table 10. The index of both periods is $t_{min}$=2 and $t_{max}$=3. So, $t_{min}$ is less than $t_{max}$. Then,

stop the iteration.

**Table 9.** Execution flow in steps 1 to 4 using the pull method.

| **Periods *t*** | **Item** | **1** | **2** | **3** | **4** | **5** |
|---|---|---|---|---|---|---|
| Replenished plan | 1 | 325 | 0 | 0 | 106 | 136 |
| | 2 | 87+52<br>=139 | 52-<br>52<br>=0 | 111 | 142 | 118 |
| Ending inventory | 1 | 210 | 96 | 0 | 0 | 0 |
| | 2 | 52 | 0 | 0 | 0 | 0 |
| Inventory cost | 1 | 2,105 | 96 | 0 | 1,300 | 1,489 |
| | 2 | 3x139+1x52+255<br>= 724 | 0 | 125 | 1,773 | 721 |
| Sum of inventory and replenishment (*SIR*) | 1 | 1x325<br>=325 | 210 | 96 | 106 | 136 |
| | 2 | 4x139<br>=556 | 208 | 444 | 568 | 472 |
| $SIR_{allitems}$ | | 881 | 418 | 540 | 674 | 608 |
| *SC* | | 756 | 673 | 633 | 758 | 608 |
| $SIR_{allitems}$_*U* | | +125 | -<br>255 | -93 | -84 | 0 |

**Table 10.** Execution flow in steps 1 to 4 using the pull method.

| **Periods *t*** | Item | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| Replenished plan | 1 | 325-125<br>=200 | 0+125=125 | 0 | 106 | 136 |
| | 2 | 139 | 0 | 111 | 142 | 118 |
| Ending inventory | 1 | 85 | 96 | 0 | 0 | 0 |
| | 2 | 52 | 0 | 0 | 0 | 0 |
| Inventory cost | 1 | 4x200+1x85+595<br>=1,480 | 1,071 | 0 | 1,300 | 1,489 |
| | 2 | 724 | 0 | 125 | 1,773 | 721 |
| Sum of inventory and replenishment (*SIR*) | 1 | 1x200<br>=200 | 210 | 96 | 106 | 136 |
| | 2 | 556 | 208 | 444 | 568 | 472 |
| $SIR_{allitems}$ | | 756 | 418 | 540 | 674 | 608 |
| *SC* | | 756 | 673 | 633 | 758 | 608 |
| $SIR_{allitems}$_*U* | | 0 | -255 | -93 | -84 | 0 |

Therefore, the inventory cost is effectively improved to 8,683. The GAMS/CPLEX solver can calculate the optimal solution of 8,521 units. Both the proposed heuristic and Gutiérrez et al. [17] can also calculate the approximate solution the same as 8,683 and the smoothing method of Nixon and Poh [27] can run about 9,494 (see Table11). Its replenished plan is shown in Table 11.

**Table 11.** Replenished plan solved by Gutiérrez et al. [17] and smoothing method [27].

| **Gutiérrez et al. [17]** | | | | | | Inventory cost |
|---|---|---|---|---|---|---|
| **Period** | **1** | **2** | **3** | **4** | **5** | |
| Item 1 | 200 | 125 | 0 | 106 | 136 | 5,340 |
| 2 | 139 | 0 | 111 | 142 | 118 | 3,343 |
| Total inventory cost | | | | | | 8,683 |
| Nixon and Poh [27] | | | | | | Inventory cost |
| **Period** | **1** | **2** | **3** | **4** | **5** | |
| Item 1 | 115 | 210 | 0 | 106 | 136 | 5,510 |
| 2 | 87 | 52 | 111 | 142 | 118 | 3,987 |
| Total inventory cost | | | | | | 9,497 |

From Tables 10 and 11, the gaps between the proposed heuristic, Gutiérrez et al. [17], the smoothing method [27], and GAMS/CPLEX are 1.9 %, 1.9 %, and 11.45 %, respectively. Both the proposed heuristic and Gutiérrez et al. [17] execute approximately five replenishment orders, whereas the smoothing method executes about six. Consequently, the smoothing method incurs a higher total inventory cost than the other approaches due to the increased ordering cost. For further testing, Minner[18] recommended generating test instances as follows: products varied between 3 and 10 and periods varied between 4 and 18, demands are drawn from a uniform (or normal) distribution over a specified range (e.g.\ U[0,100], setup costs are drawn similarly (e.g.\ U[50,150], unit production costs are drawn from U[1,10], holding costs are held constant $h$ =1, weights are drawn from U[1,N], and warehouse capacity bounds are taken as $A=\sum_{n=1}^{N} w_i d_{i,t}$, $U_t = \left[ A + B\cdot(\sum_{n=1}^{N} w_i D_{i,t+1}) \right]$ a fixed fraction B =10%. In the format for the storage capacity, parameter A is the sum of the demand onall items at period t with the lower bound (B=1%). Authors generate the example data following Minner [18] procedure as shown in Table 12.

**Table 12.** A simple example by formatted data of Minner [18].

| **Periods *t*** | **1** | **2** | **3** | **4** | **5** | **6** |
|---|---|---|---|---|---|---|
| $U_t$ | 1161 | 529 | 768 | 973 | 721 | 806 |
| Item 1,$w_1$=2 | | | | | | |
| $d_{1,t}$ | 44 | 47 | 64 | 67 | 67 | 9 |
| $f_1$ | | | 96 | | | |
| $p_1$ | | | 6 | | | |
| $h_1$ | | | 1 | | | |
| Item 2,$w_2$=5 | | | | | | |
| $d_{2,,t}$ | 83 | 21 | 36 | 87 | 70 | 88 |
| $f_2$ | | | 74 | | | |
| $p_2$ | | | 10 | | | |

| | | | | | | |
|---|---|---|---|---|---|---|
| $h_2$ | | | 1 | | | |
| Item 3,$w_1$=4 | | | | | | |
| $d_{,3,t}$ | 88 | 12 | 58 | 65 | 39 | 87 |
| $f_3$ | | | 67 | | | |
| $p_3$ | | | 9 | | | |
| $h_3$ | | | 1 | | | |

A instance of Minner [18] was computed by heuristics and the resulting solutions are presented in Table 13.

**Table 13.** Total cost and gap solution obtained by the proposed heuristic, Gutiérrez et al. [17] and smoothing [27].

| Heuristics/MIP solver | GAMS/ CPLEX | Push and Pull | Gutiérrez et al. [17] | Smoothing [27] |
|---|---|---|---|---|
| Total cost | 9,928 | 9,928 | 10,054 | 10,030 |
| % Gap solution | - | 0 | 1.27 | 1.02 |
| No. of additional orders | - | 3 | 4 | 4 |

In Table 13, the push-and-pull heuristic achieves an optimality gap of approximately 0 %, outperforming Gutiérrez et al. [17], which has a gap of 1.27 %, and the smoothing method [27], with a gap of 1.02 %. The proposed heuristic places about three replenishment orders, whereas both Gutiérrez et al. [17] and the smoothing method place around four. Conse quently, the push-and-pull heuristic's performance is further validated in Section 5 on a set of randomly generated problem instances.

## 5. Computational Result

For confidence in using heuristics, this study compares the solutions for the proposed heuristic, algorithm by Gutiérrez et al., and GAMS/CPLEX solver. The set of randomly generated problems is identical to the cost framework of Minner [18]. Each problem runs on formatting parameters, as shown in Table 14.

**Table 14.** Formatting parameters.

| **Number of periods, *T*** | **6** | **12** | **24** |
|---|---|---|---|
| Number of items, *N* | 10,20,40,60,80,…,160 | 10,20,30,…,80 | 10,20,…,160 |
| Number of instances | 10 | 10 | 5 |
| Weight distribution, $w_i$ | Uniform, $w_i \sim [1,10]$ | | |
| Demand distribution, $d_{i,t}$ | Uniform, $d_{i,t} \sim [30,150]$ | | |
| Ordering cost distribution, $f_{i,t}$ | Uniform, $f_{i,t} \sim [100,150]$ | | |

| | |
|---|---|
| Inventory bounds, $U_t$ | $A = \sum_{n=1}^{N} w_i d_{i,t}$, $U_t = \left[ A + B \cdot (\sum_{n=1}^{N} w_i D_{i,t+1}) \right]$,<br>B = {1%,5%,10%, 20%} |
| Cost of procuring raw materials, $p_{i,t}$ = zero and holding cost, $h_{i,t}$ = 1 | |

The parameter B is the additional capacity generated from the accumulative demand of period *t*+1 with the upper bound. If the upper bound is high, such as B = 20%, the problem can be solved more easily. Otherwise, it is more difficult to address.

This study implements MATLAB 2024 A software to solve the network flow algorithm based on dynamic programming for the initial solution, the proposed algorithm [28], and Gutiérrez et al.’s algorithm [29].The solution for the MIP model is generated by GAMS 46.3.0 licensed for continuous and discrete problems. An HP Pavilion X360 Notebook running Windows 10 with an Intel Core i7 64-bit processor at 1.99 GHz and 24 GB of RAM was used to execute both the heuristics and MIP formulation. MATLAB software uses general-purpose programming that is more flexible and allows users to apply specified code. For generating the optimization solution, the GAMS/CPLEX solver is concentrated on optimization, which is less flexible but powerful for LP, MIP, and NLP problems [31].

The solution for the GAMS/CPLEX solver compares all the results of the experiment. It can be explained with the solution gap equation below.

$$\text{Solution gap (\%)} = \left[ \frac{\text{Solution of heuristic-Solution of GAMS/CPLEX}}{\text{Solution of GAMS/CPLEX}} \right] \text{x } 100$$

*5.1. Experiment Results*

This study divides the category for the random example into three sub-categories: A small-scale problem based on the number of periods *N* =6, a medium-scale problem based on the number of periods *N*=12, and a large-scale problem based on the number of periods *N*=24. This experiment shows their solution gaps and computation times varying the parameter B in Tables 15-18, as follows

**Table 15.** Computation times and solution gaps with near-minimal storage capacity using parameter B = 1%.

| | | | | **Min. gap (%)** | | **Max. gap (%)** | | **Avg. gap (%)** | |
|---|---|---|---|---|---|---|---|---|---|
| **NxT** | **Avg. Push & pull heuristic time (s.)** | **Avg. Gutiérrez’s heuristic time (s.)** | **Avg. GAMS/CPLEX time (s.)** | **Push & pull heuristic** | **Gutiérrez ’s heuristic** | **Pull & push heuristic** | **Gutiérrez ’s heuristic** | **Pull & push heuristic** | **Gutiérrez ’s heuristic** |
| 10x6 | 6.26 | 6.24 | 0.49 | 0.00 | 0.21 | 2.77 | 3.78 | 0.57 | 1.89 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 20x6 | 6 | 11.45 | 11.16 | 0.88 | 0.38 | 1.03 | 5.29 | 8.30 | 1.04 |
| 40x6 | 21.90 | 21.24 | 2.56 | 0.28 | 2.87 | 0.88 | 5.02 | 0.72 | 3.48 |
| 60x6 | 32.48 | 30.99 | 5.31 | 0.44 | 2.24 | 1.05 | 4.53 | 0.82 | 3.57 |
| 80x6 | 44.25 | 41.81 | 7.27 | 0.67 | 2.65 | 1.13 | 4.47 | 0.91 | 3.44 |
| 100x6 | 56.24 | 51.91 | 11.92 | 0.47 | 3.14 | 1.17 | 4.99 | 0.92 | 3.82 |
| 120x6 | 69.42 | 60.82 | 12.28 | 0.55 | 2.90 | 1.15 | 4.60 | 0.89 | 3.64 |
| 140x6 | 84.99 | 71.33 | 15.84 | 0.52 | 3.03 | 1.76 | 5.05 | 0.94 | 4.07 |
| 160x6 | 100.72 | 81.98 | 57.13 | 0.04 | 2.98 | 8.51 | 11.90 | 2.08 | 5.09 |
| | 36.73 | 34.53 | 6.26 | | | | | 1.16 | 3.99 |
| 10x12 | 61.85 | 64.78 | 0.85 | 0.22 | 2.54 | 1.32 | 7.27 | 0.79 | 5.36 |
| 20x12 | 131.74 | 140.63 | 3.67 | 0.73 | 2.44 | 1.68 | 7.31 | 1.23 | 4.53 |
| 30x12 | 204.64 | 216.34 | 8.57 | 1.24 | 4.21 | 1.93 | 7.17 | 1.51 | 5.66 |
| 40x12 | 260.38 | 273.55 | 39.66 | 1.08 | 4.14 | 1.89 | 7.85 | 1.42 | 5.54 |
| 50x12 | 328.35 | 321.77 | 74.41 | 1.15 | 4.89 | 1.77 | 6.64 | 1.39 | 5.71 |
| 60x12 | 409.28 | 458.70 | 102.65 | 1.18 | 4.98 | 1.90 | 6.16 | 1.39 | 5.65 |
| 70x12 | 495.31 | 679.99 | 380.50 | 0.23 | 4.97 | 1.46 | 6.77 | 1.24 | 5.61 |
| 80x12 | 520.24 | 537.58 | 2388.1 | 1.21 | 4.22 | 1.48 | 6.56 | 1.30 | 5.67 |
| | 271.5 | 370.23 | 333.20 | | | | | 1.33 | 5.57 |
| 10x24 | 3,037.37 | 3,429.14 | 2.31 | 0.95 | 6.04$^{**}$ | 1.58 | 7.73$^{**}$ | 1.27 | 6.89$^{**}$ |
| 20x24 | 6,507.96 | 5,765.73 | 59.68 | 1.58 | 7.16$^{*}$ | 2.20 | 7.16$^{*}$ | 1.83 | 7.16$^{*}$ |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 30x24 | 7,303.46 | 8,694.35 | 3,549.2 | 1.11 | 7.42* | 2.21 | 7.42 | 1.84 | 7.42* |
| 40x24 | 12,903.55 | 13,484.40 | 57,788.2 | 0.91 | 7.49* | 2.07 | 7.49* | 1.54 | 7.49* |
| | 5,994.42 | 5,357.88 | 12,280 | | | | | 1.66 | 7.30 |

*,** One and two instances 10x6-160x6 small-scale problem, 10x12-80x12 medium-scale problem and 10x24-40x24 large-scale problem.

**Table 16.** Computation times and solution gaps when parameter B = 5%.

| NxT | Avg. Push & pull heuristic time (s.) | Avg. Gutiérrez's heuristic time (s.) | Avg. GAMS /CPLEX time (s.) | Min. gap (%) | | Max. gap (%) | | Avg. gap (%) | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | Push & pull heuristic | Gutiérrez's heuristic | Pull & push heuristic | Gutiérrez's heuristic | Pull & push heuristic | Gutiérrez's heuristic |
| 10x6 | 5.65 | 5.67 | 0.56 | 0.00 | 0.77 | 3.33 | 6.54 | 1.39 | 3.96 |
| 20x6 | 9.77 | 10.36 | 1.04 | 0.89 | 1.66 | 2.35 | 9.30 | 1.59 | 5.33 |
| 40x6 | 21.41 | 23.14 | 5.91 | 1.23 | 3.27 | 2.23 | 8.20 | 1.72 | 6.46 |
| 60x6 | 29 | 31.99 | 10.34 | 1.00 | 5.13 | 2.32 | 9.23 | 1.73 | 7.31 |
| 80x6 | 35.13 | 36.28 | 27.42 | 1.00 | 3.98 | 2.59 | 8.53 | 1.76 | 6.88 |
| 100x6 | 44.65 | 46.03 | 33.74 | 0.92 | 5.84 | 1.80 | 8.29 | 1.60 | 6.83 |
| 120x6 | 53.20 | 54.55 | 29.79 | 1.48 | 5.53 | 2.11 | 7.68 | 1.58 | 6.71 |
| 140x6 | 63.23 | 64.96 | 57.33 | 0.92 | 4.89 | 1.85 | 6.95 | 1.53 | 6.23 |
| 160x6 | 72.58 | 76.85 | 33.79 | 0.98 | 5.48 | 2.88 | 8.21 | 1.70 | 6.29 |
| | 29.11 | 30.33 | 18.43 | | | | | 1.64 | 6.56 |
| 10x12 | 71.52 | 62.86 | 0.89 | 0.62 | 0.96 | 2.49 | 12.53 | 1.47 | 7.01 |
| 20x12 | 142.7 | 132.49 | 2.33 | 0.73 | 2.18 | 1.62 | 7.89 | 1.21 | 4.69 |
| 30x12 | 189.3 | 213.94 | 3.90 | 0.65 | 1.50 | 1.35 | 7.76 | 1.02 | 4.91 |
| 40x12 | 254.2 | 256.25 | 7.00 | 0.63 | 2.90 | 1.23 | 6.24 | 0.93 | 4.38 |
| 50x12 | 319.9 | 339.75 | 14.92 | 0.64 | 0.13 | 1.01 | 5.38 | 0.88 | 3.96 |
| 60x12 | 417.6 | 422.97 | 18.73 | 0.66 | 2.91 | 1.27 | 5.62 | 0.93 | 4.30 |
| 70x12 | 498.1 | 526.02 | 36.40 | 0.69 | 2.99 | 1.18 | 5.36 | 0.87 | 3.94 |
| 80x12 | 523.7 | 519.15 | 24.03 | 0.67 | 2.11 | 1.15 | 6.65 | 0.82 | 4.01 |
| | 272.22 | 368.08 | 12.03 | | | | | 0.92 | 4.43 |
| 10x24 | 1,398.7 | 1,578.05 | 0.45 | 0.14 | 1.68 | 1.68 | 5.01 | 0.94 | 3.19 |
| 20x24 | 6,592.9 | 7,140.05 | 6.22 | 0.31 | 1.62 | 0.79 | 3.08 | 0.59 | 2.48 |
| 30x24 | 9,608.8 | 9,364.59 | 47.16 | 0.24 | 1.06 | 0.80 | 2.46 | 0.58 | 1.95 |
| 40x24 | 12,824.7 | 13,504.5 | 61.94 | 0.22 | 1.91 | 0.58 | 3.07 | 0.40 | 2.60 |
| | 8,117.58 | 8,465.86 | 5.98 | | | | | 0.55 | 2.41 |

10x6-160x6 small-scale problem, 10x12-80x12 medium-scale problem and 10x24-40x24 large-scale problem .

**Table 17.** Computation times and solution gaps when parameter B = 10%.

| NxT | Avg. Push & pull heuristic time (s.) | Avg. Gutiérrez’s heuristic time (s.) | Avg. GAMS/CPLEX time (s.) | Min. gap (%) | | Max. gap (%) | | Avg. gap (%) | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | Push & pull heuristic | Gutiérrez ’s heuristic | Pull & push heuristic | Gutiérrez ’s heuristic | Pull & push heuristic | Gutiérrez ’s heuristic |
| 10x6 | 5.83 | 5.64 | 0.31 | 0.00 | 0.02 | 1.71 | 12.28 | 0.84 | 4.93 |
| 20x6 | 10.10 | 10.19 | 0.42 | 0.44 | 0.73 | 1.62 | 13.44 | 1.11 | 5.00 |
| 40x6 | 20.40 | 22.22 | 2.35 | 0.51 | 2.29 | 2.16 | 6.55 | 1.12 | 4.66 |
| 60x6 | 29.51 | 28.04 | 2.04 | 0.54 | 2.12 | 1.88 | 7.13 | 1.10 | 3.97 |
| 80x6 | 35.23 | 36.34 | 4.82 | 0.50 | 2.55 | 1.55 | 5.87 | 1.01 | 4.30 |
| 100x6 | 44.68 | 45.50 | 7.19 | 0.48 | 2.00 | 1.28 | 6.22 | 0.87 | 4.06 |
| 120x6 | 54.61 | 54.74 | 11.24 | 0.63 | 2.19 | 4.05 | 5.38 | 1.12 | 4.06 |
| 140x6 | 62.91 | 63.74 | 7.35 | 0.58 | 2.65 | 1.06 | 5.10 | 0.80 | 3.40 |
| 160x6 | 73.68 | 73.68 | 7.30 | 0.56 | 2.45 | 2.04 | 5.25 | 0.96 | 3.63 |
| | 29.25 | 29.6 | 3.97 | | | | | 0.93 | 3.92 |
| 10x12 | 66.41 | 62.41 | 0.45 | 0.04 | 0.31 | 1.74 | 4.26 | 0.74 | 1.74 |
| 20x12 | 141.27 | 132.11 | 0.69 | 0.25 | 0.25 | 0.86 | 4.26 | 0.57 | 1.58 |
| 30x12 | 187.59 | 210.50 | 1.25 | 0.39 | 1.17 | 1.11 | 3.27 | 0.59 | 1.86 |
| 40x12 | 264.08 | 256.14 | 2.23 | 0.36 | 0.45 | 0.74 | 2.32 | 0.48 | 1.42 |
| 50x12 | 325.22 | 317.78 | 2.91 | 0.26 | 1.02 | 0.64 | 2.92 | 0.46 | 1.67 |
| 60x12 | 414.79 | 408.41 | 3.13 | 0.28 | 0.20 | 0.63 | 3.22 | 0.47 | 2.01 |
| 70x12 | 496.28 | 495.38 | 5.75 | 0.31 | 1.43 | 0.77 | 2.73 | 0.48 | 1.98 |
| 80x12 | 517.69 | 524.16 | 6.43 | 0.32 | 1.41 | 0.61 | 2.33 | 0.45 | 1.85 |
| | 272.6 | 358.76 | 2.58 | | | | | 0.49 | 1.81 |
| 10x24 | 3,177.71 | 3,508.17 | 0.53 | 1.60 | 1.08 | 2.52 | 2.65 | 0.56 | 1.74 |
| 20x24 | 6,924.87 | 7,841.10 | 1.72 | 0.98 | 0.88 | 1.90 | 1.74 | 0.45 | 1.15 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 30x24 | 9,188.50 | 9,681.13 | 2.38 | 0.14 | 0.28 | 0.45 | 1.82 | 0.31 | 1.10 |
| 40x24 | 13,527.06 | 13,764.70 | 3.32 | 0.16 | 1.03 | 0.65 | 1.78 | 0.34 | 1.24 |
| | 8,278.51 | 8,773.25 | 2.14 | | | | | 0.38 | 1.23 |

10x6-160x6 small-sclae problem, 10x12-80x12 medium-scale problem and 10x24-40x24 large-scale problem.

**Table 18.** Computation times and solution gaps when parameter B = 20 %.

| NxT | Avg. Push & pull heuristic time (s.) | Avg. Gutiérrez’s heuristic time (s.) | Avg. GAMS/CPLEX time (s.) | Min. gap (%) | | Max. gap (%) | | Avg. gap (%) | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | Push & pull heuristic | Gutiérrez’s heuristic | Pull & push heuristic | Gutiérrez’s heuristic | Pull & push heuristic | Gutiérrez’s heuristic |
| 10x6 | 5.85 | 5.71 | 0.29 | 0.00 | 0.65 | 2.50 | 4.83 | 0.83 | 2.48 |
| 20x6 | 10.02 | 10.25 | 0.34 | 0.09 | 0.14 | 1.51 | 2.18 | 0.68 | 1.18 |
| 40x6 | 22.22 | 22.94 | 1.74 | 0.26 | 1.07 | 1.29 | 2.40 | 0.57 | 1.76 |
| 60x6 | 29.54 | 29.66 | 0.87 | 0.20 | 1.21 | 1.14 | 2.34 | 0.60 | 1.75 |
| 80x6 | 35.24 | 35.71 | 0.90 | 0.44 | 1.18 | 1.03 | 2.17 | 0.62 | 1.71 |
| 100x6 | 45.18 | 45.34 | 2.02 | 0.31 | 1.14 | 0.84 | 2.07 | 0.61 | 1.71 |
| 120x6 | 53.90 | 55.09 | 1.87 | 0.36 | 1.53 | 0.89 | 1.53 | 0.65 | 1.72 |
| 140x6 | 62.99 | 64.38 | 1.66 | 0.47 | 1.67 | 0.92 | 2.75 | 0.67 | 1.97 |
| 160x6 | 73.13 | 74.91 | 1.81 | 0.35 | 1.53 | 1.66 | 2.82 | 0.76 | 1.87 |
| | 37.56 | 38.22 | 1.28 | | | | | 0.66 | 1.79 |
| 10x12 | 36.14 | 32.01 | 0.28 | 0.00 | 0.00 | 1.07 | 2.34 | 0.39 | 0.65 |
| 20x12 | 71.55 | 62.07 | 0.45 | 0.00 | 0.00 | 0.88 | 1.65 | 0.34 | 0.91 |
| 30x12 | 185.60 | 205.13 | 0.52 | 0.12 | 0.25 | 0.76 | 1.42 | 0.37 | 0.87 |
| 40x12 | 264.97 | 258.89 | 0.70 | 0.17 | 0.60 | 0.49 | 1.24 | 0.36 | 0.92 |
| 50x12 | 330.85 | 322.59 | 1.20 | 0.19 | 0.61 | 0.53 | 2.38 | 0.35 | 1.03 |
| 60x12 | 491.05 | 501.27 | 3.97 | 0.20 | 0.61 | 0.47 | 1.30 | 0.33 | 0.94 |
| 70x12 | 491.05 | 501.27 | 3.97 | 0.20 | 0.61 | 0.47 | 1.30 | 0.33 | 0.94 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 80x12 | 515.24 | 541.88 | 2.10 | 0.25 | 0.59 | 0.44 | 1.16 | 0.34 | 0.84 |
| | 264.63 | 358.40 | 1.61 | | | | | 0.36 | 0.87 |
| 10x24 | 3,018.94 | 3,196.81 | 0.63 | 0.82 | 0.08 | 1.64 | 1.07 | 0.34 | 0.60 |
| 20x24 | 7,084.67 | 6,422.94 | 1.14 | 0.17 | 0.16 | 2.41 | 0.73 | 0.28 | 0.47 |
| 30x24 | 9,889.41 | 9,612.80 | 1.40 | 0.15 | 0.38 | 0.61 | 0.75 | 0.31 | 0.57 |
| 40x24 | 13,706.49 | 12,863.77 | 1.78 | 0.13 | 0.41 | 0.31 | 0.73 | 0.20 | 0.56 |
| | 8,506.37 | 8,101.12 | 2.12 | | | | | 0.27 | 0.55 |

10x6-160x6 small-scale problem, 10x12-80x12 medium-scale problem and 10x24-40x24 large-scale problem.

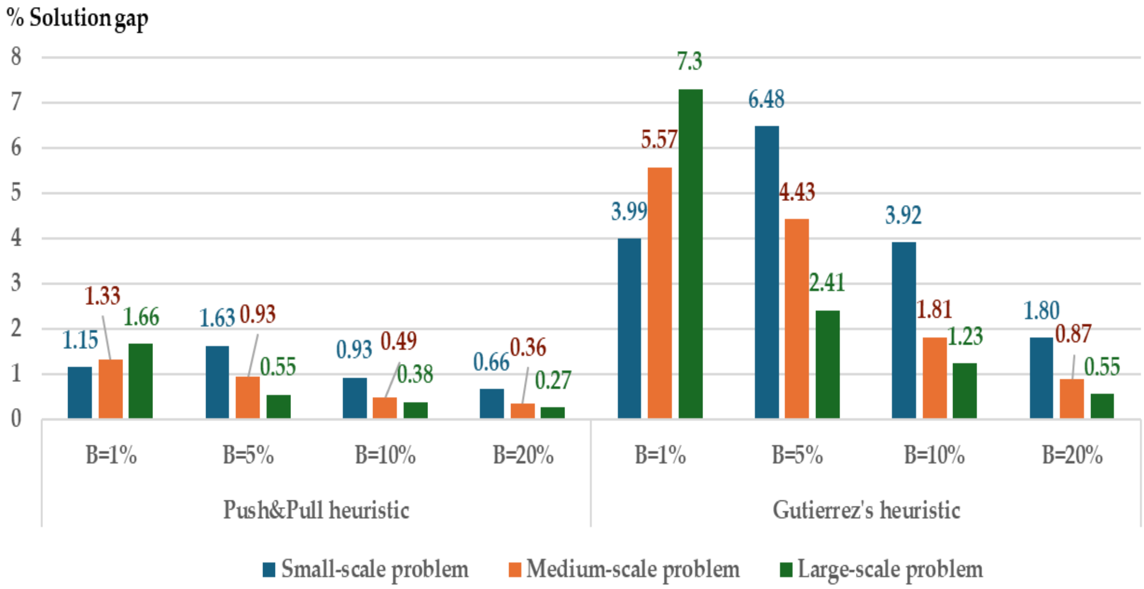

**Figure 3.** Average solution gap of both heuristics under storage capacities when parameter B=1,5,10 and 20%.

### *5.1. Solution Gap*

In Figure 3, the average solution gaps for the push and pull algorithm on large-scale problem are 0.55%,0.38%, and 0.20% for parameter B = 5%, 10% and 20%, respectively. Gutiérrez et al.'s heuristic generates solution gaps of about 2.41%, 1.23% and 0.55 %. When comparing the solution gaps, the proposed heuristic performs better than Gutiérrez et al.'s heuristic. The performance of the solution gap depends on the value of parameter B. If parameter B increases, the solution gap is lower . The proposed algorithm can determine the number of replenished quantities to move relaxed while satisfying all demands with a high parameter B. Gutiérrez et al.'s heuristic sometimes moves the replenished quantity from any period to period *t*+*k*, when $k \in \{1, 2, ..., T-t\}$ .Ordering cost must be paid more frequently when inserting the replenished quantity for period *t*+*k* in more time, causing total inventory cost to grow. Unfortunately, the amount of $SIR_{allitem}\_U$ for period *t* does not also decline. While the push and pull heuristic moves each replenished quantity from period *t* to a consecutive period only with zero inventory cost, it can certainly reduce the replenished quantity for period *t* so that the amount of $SIR_{allitems}$ _*U* reduces.

### 5.2. Worst Cases Analysis

For the robust condition of the push & pull heuristic, this study introduces the near-minimal storage capacity. The performance of a heuristic depends on the value of the storage capacity. Suppose each storage capacity is likely near the sum of demand for each period, called near-minimal storage capacity. Moving the partial or whole replenished quantities to the next period is difficult.

In Table 15, it is difficult for Gutiérrez et al.'s heuristic to execute any random instances with near-minimal storage capacities. It can calculate only one and two from five and ten instances, such as 20x24, 30x24, and 40x24 problems (*N*x*T*). Other solutions cannot satisfy the demand. Meanwhile, the push and pull heuristic can calculate all random instances with these storage capacities. Its solution gap performs well on the small-and medium-scale problem, at about 1.15% and 1.33%, the same as the other instances with high storage capacities (parameter B =5-20%).

At the same time, Gutiérrez et al.'s heuristic gap solution is about 3.99% and 5.57%. Therefore, the push and pull heuristic enables computing the replenished plan significantly better with near-minimal storage capacities.

For large-scale problems (*T*=24), MATLAB cannot run on the extension of the number of periods due to being out-of-memory. When increasing the number of periods, its memory usage exceeds 76 GB. The limitation of the system memory space (RAM and swap file) used by MATLAB for this computer is about 76 GB.

To strengthen empirical benchmarking and provide better justification, compare the proposed heuristics with additional baseline methods, including the smoothing heuristic [27] , implemented using the state-of-the-art code [31]. The authors compare their solution gap performance, which is shown in Table 19.

**Table 19.** Gap performance of the proposed, Gutiérrez et al., and smoothing heuristics under storage capacities with parameter B = 1%, 5%, 10%, and 20%.

| Parameter B | The push and pull heuristic | | | |
|---|---|---|---|---|
| | **Problem size** | **10X6 small-scale problem** | **10X12 medium-scale problem** | **10X24 large-scale problem** |
| 1% | Avg. | 0.57 | 0.79 | **1.27** |
| | Max. | 0.73 | 1.32 | 1.58 |
| 5% | Avg. | 1.39 | 1.47 | 0.94 |
| | Max. | 2.62 | 2.49 | 1.27 |
| 10% | Avg. | 0.85 | 0.74 | 0.56 |
| | Max. | 1.81 | 1.74 | 1.2 |
| 20% | Avg. | 1.84 | 0.34 | 0.34 |
| | Max. | 2.5 | 0.88 | 1.64 |
| | Gutiérrez et al.'s heuristic | | | |
| 1% | Avg. | 1.89 | 5.36 | **6.89** |
| | Max. | 3.51 | 7.27 | 7.73 |
| 5% | Avg. | 3.96 | 7 | 0.94 |
| | Max. | 6.54 | 11.84 | 1.04 |

| | | | | |
|---|---|---|---|---|
| 10% | Avg. | 4.92 | 1.74 | 1.74 |
| | Max. | 12.28 | 4.26 | 2.65 |
| 20% | Avg. | 2.48 | 0.65 | 0.6 |
| | Max. | 4.83 | 2.34 | 1.07 |
| | Smoothing heuristic [27] | | | |
| 1% | Avg. | 1.89 | 4.24 | **5.97** |
| | Max. | 3.4 | 6.24 | 6.79 |
| 5% | Avg. | 3.96 | 4.68 | 2.3 |
| | Max. | 6.54 | 7.53 | 2.46 |
| 10% | Avg. | 4.13 | 2.04 | 2.8 |
| | Max. | 9.83 | 3.51 | 8.46 |
| 20% | Avg. | 1.7 | 0.73 | 0.77 |
| | Max. | 3.35 | 1.53 | 1.35 |

In Table 19, the proposed heuristic shows good average gap performance on large-scale problems, such as the 10x24 case, with gaps of about 1.27% and 0.34% under storage capacities B = 1% and 20%, respectively. In comparison, Gutiérrez et al.'s heuristic has gaps of approximately 6.89% and 1.07%, while the smoothing heuristic has gaps of about 5.97% and 1.35%, respectively. As a result, the gap performances of Gutiérrez et al.'s and the smoothing heuristics differ by only a small amount. Therefore, the proposed heuristic is able to compute an approximate replenishment plan that is better than the previous heuristics.

### *5.3. Computation Time*

This study implements the codes based on the state-of-the-art methods for both heuristics. Both the push & pull heuristic and the heuristic by Gutiérrez et al. have the same time complexity, denoted as $O(W \cdot N \cdot T^2)$. Therefore, the running times of the two heuristics are nearly the same.The time complexity for the network flow based on dynamic programming algorithm [22] has $O(NTD_{\max}^2)$ for generating the initial solution. The computation time of both heuristics combines the running time for the network flow based on dynamic programming for the initial solution with the running time of each heuristic. The worst-case complexity of the GAMS/CPLEX solver has $O(2^{NxT})$, which is an exponential growth rate. However, this solver enhances performance with the branch-cut and benders decomposition algorithm for reducing the running time efficiency in large-scale problems [28] when compared with MATLAB software. For computation time, this study focuses on small-to-medium and large-scale problems with near-minimal storage capacity. Therefore, the data in Figure 4 include both small-to-medium and large-scale problems (see Table 15) with storage capacities parameter B=1-20%. Both the heuristics and MIP solver generate solutions under storage capacities. The computation time for these conditions is shown in Figures 4-7, as follows:

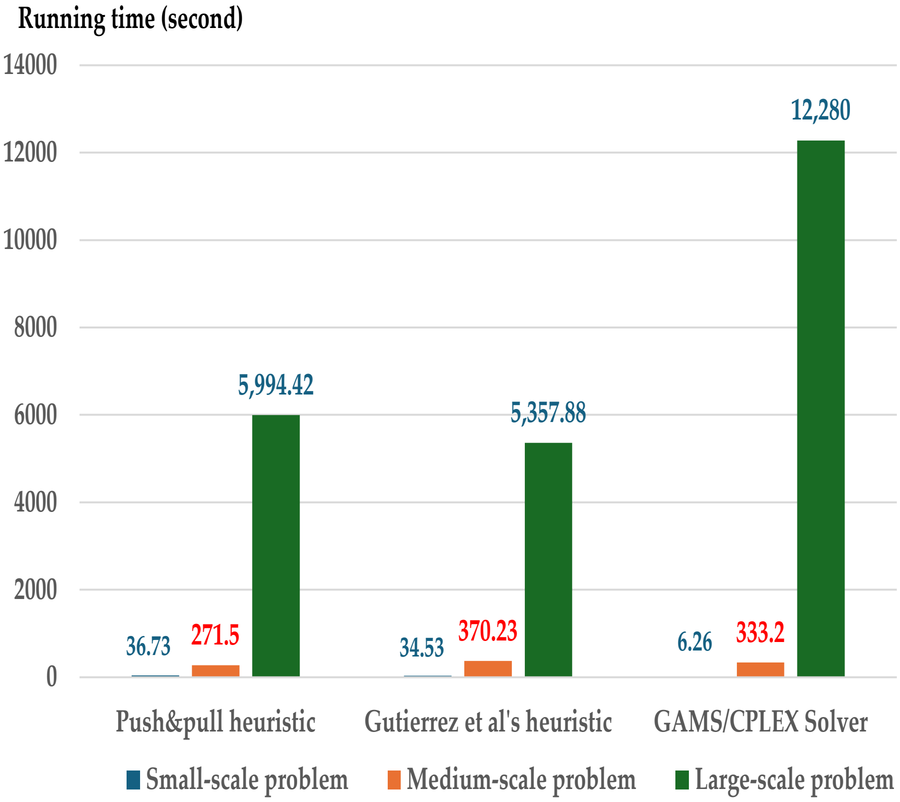

**Figure 4.** Average computation time with near-minimal storage capacity (B=1%) using a) push and pull heuristic b) Gutiérrez et al.'s heuristic, , and c) GAMS/CPLEX solver.

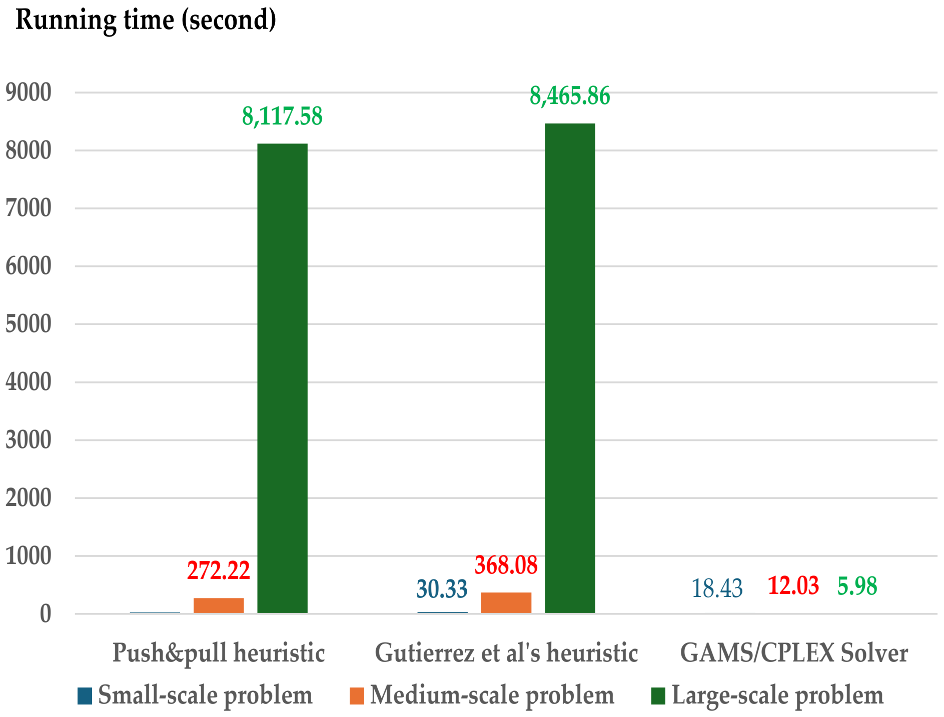

**Figure 5.** Average computation time with near-minimal storage capacity (B=5%) using a) push and pull heuristic b) Gutiérrez et al.'s heuristic, , and c) GAMS/CPLEX solver.

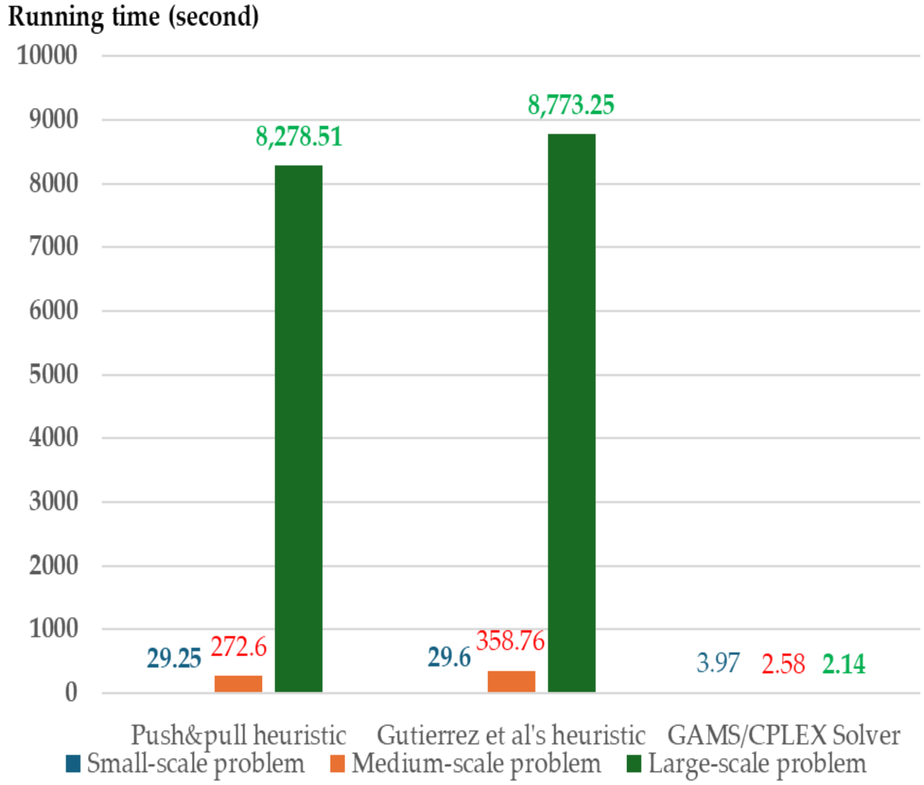

**Figure 6.** Average computation time with near-minimal storage capacity (B=10%) using a) push and pull heuristic b) Gutiérrez et al.'s heuristic, , and c) GAMS/CPLEX solver.

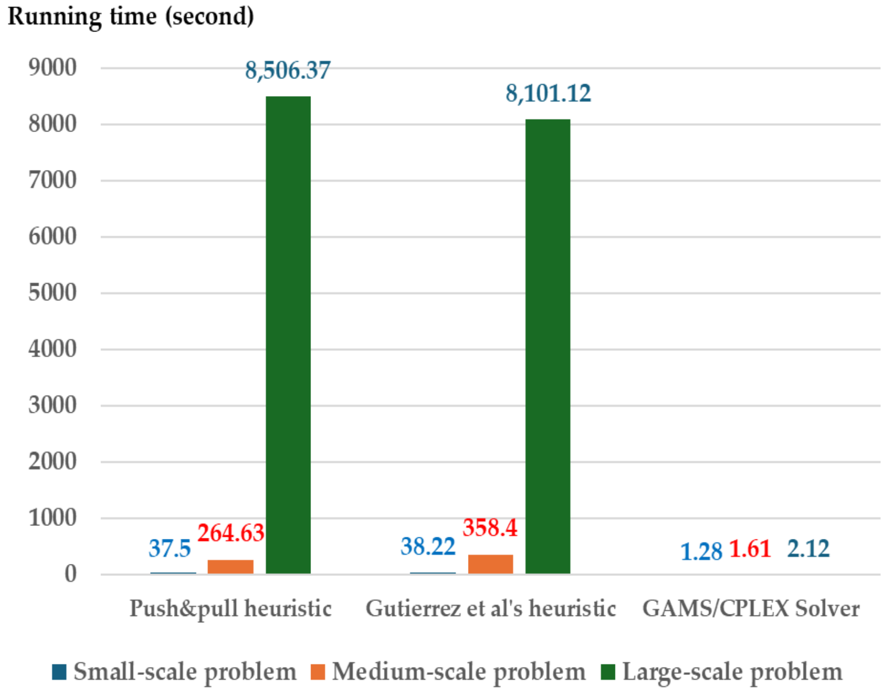

**Figure 7.** Average computation time with near-minimal storage capacity (B=20%) using a) push and pull heuristic b) Gutiérrez et al.'s heuristic, and c) GAMS/CPLEX solver.

doi:10.20944/preprints202506.0178.v1

In Figure 4, the computation time generated by the GAMS/CPLEX solver increases exponentially for the large-scale problem under near-minimal storage capacities (B = 1%). In general, the CPLEX solver must execute effectively with a branch-and-cut algorithm and special heuristic. However, the running time for solving a large-scale problem includes poor performance with near-minimal storage capacities. The MIP solver must determine lot size with high running time to generate an optimal solution. This is a limitation of the MIP solver. In contrast, the computation time for both heuristics, which generate approximate solutions for large-scale problems, performs well compared to the MIP solver. Nevertheless, the computation time of the MIP solver on small- to medium-scale and large-scale problems performs well compared to both heuristics when the storage capacities have higher B values (see Figures 5–7). The running times for both heuristics are nearly the same due to their similar time complexity.

Therefore, considering the storage capacity constraints, both heuristics perform well under near-minimal capacity for large-scale problems, whereas the MIP solver computes efficiently with shorter running times for small- and medium-scale problems. For high storage capacity constraints (B = 5–20%), the MIP solver performs well with shorter running times across all problem scales.

### 5.4. Sensitivity Analysis

Authors present a sensitivity analysis on the varied parameter B to show its impact on the cost performance of the proposed heuristics, order frequency, and inventory levels. The results of this analysis are shown in Figures 8 to 10 below.

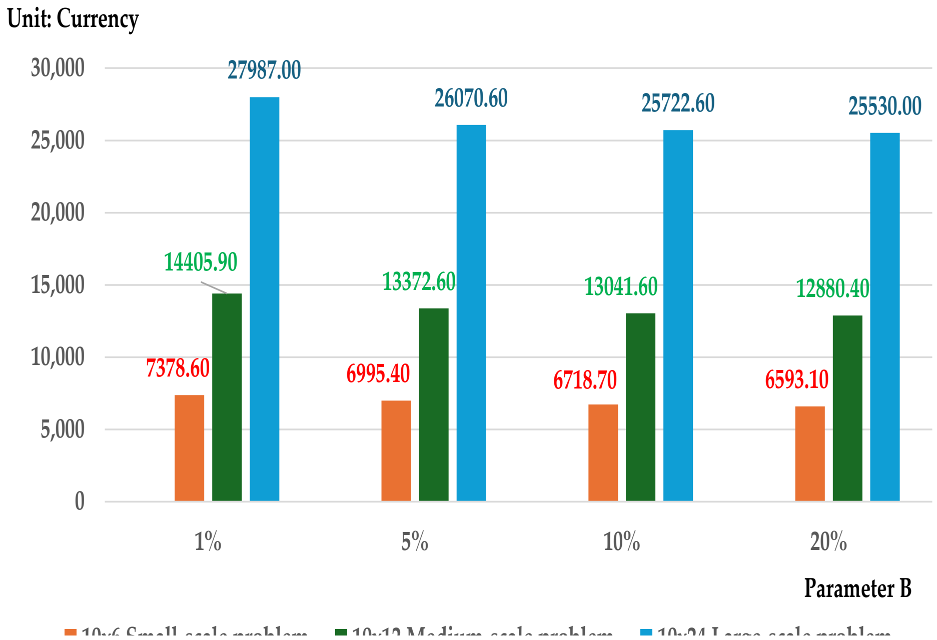

**Figure 8.** Total cost vs. storage capacity parameter B (1%–20%) for different problem Scales.

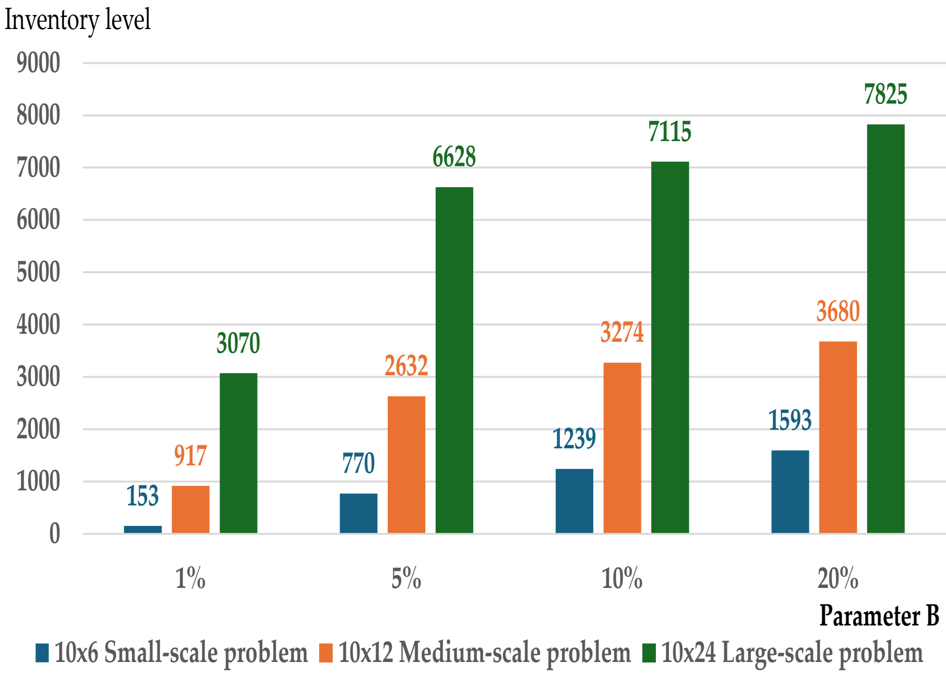

**Figure 9.** Inventory level vs. storage capacity parameter B (1%–20%) for different problem scales

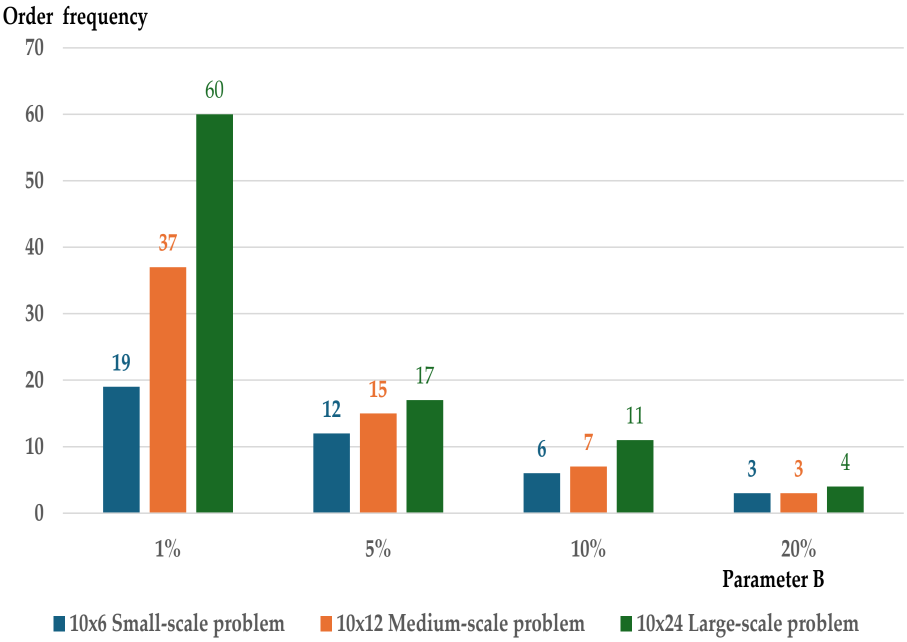

**Figure 10.** Order frequency vs. storage capacity parameter B (1%–20%) for different problem scales.

In Figure 8, the total cost for each problem size is high when the storage capacity is near minimal, and it decreases as the storage capacity increases. Under the near-min imal storage capacity constraint, the inventory level for each problem size is low due to the limited storage space (see Figure 9). This results in more frequent orders with smaller replenishment quantities to meet all demand. The higher order frequency causes an increase in the total cost (see Figure 10).

Under high storage capacity constraints, the total cost for each problem size is low, but the inventory level is higher due to the increased storage space. The order frequency is also lower in order to reduce the total cost.

In summary, tight storage space leads to higher total costs due to increased or der frequency. On the other hand, larger storage capacity results in lower total costs but requires higher investment to expand the storage space.

### *5.5. Statistical Validation of Heuristic Stability and Reliability*

For validation of heuristic stability and reliability, the authors evaluate the total cost and running time of the proposed heuristic using statistical parameters such as average, standard deviation, minimum and maximum values, and confidence intervals, as shown in Table 20.

**Table 20.** Statistics parameters of total cost computed by the push and pull methods under near-minimal storage capacities (B=1%).

| **Problem** | **Average (currency)** | **Standard deviation** | **Lower confidence interval**[(a)] | **Upper confidence interval**[(a)] | Min. total cost | Max. total cost |
|---|---|---|---|---|---|---|
| 10x6 | 7378.6 | 158.1 | 6904.4 | 7852.7 | 7114 | 7640 |
| 20x6 | 14570 | 261.2 | 13786.2 | 15353.79 | 14200 | 15081 |
| 40x6 | 29012.2 | 340.2 | 27991.6 | 30032.8 | 28409 | 29498 |
| 60x6 | 43445 | 400.0 | 42244.9 | 44645.1 | 42930 | 43941 |
| 80x6 | 58025.2 | 544.1 | 56393.1 | 59657.3 | 57059 | 58883 |
| 100x6 | 72482.8 | 565.9 | 70784.9 | 74180.6 | 71498 | 73354 |
| 120x6 | 86833 | 607.5 | 85010.5 | 88655.5 | 85551 | 87392 |
| 140x6 | 101220.8 | 645.9 | 99282.9 | 103158.7 | 99805 | 101875 |
| 160x6 | 115562.2 | 710.6 | 113430.5 | 117693.9 | 114175 | 116367 |
| 10x12 | 14405.9 | 184.0 | 13853.8 | 14957.9 | 14160 | 14742 |
| 20x12 | 28599.9 | 232.3 | 27902.9 | 29296.9 | 28134 | 28871 |
| 30x12 | 42776.4 | 329.6 | 41787.7 | 43765.1 | 42106 | 43239 |
| 40x12 | 56862.5 | 365.5 | 55765.9 | 57959.1 | 56166 | 57551 |
| 50x12 | 70941.3 | 335.1 | 69936.1 | 71946.5 | 70393 | 71384 |
| 60x12 | 85055.2 | 584.1 | 83302.9 | 86807.5 | 83928 | 85927 |
| 70x12 | 99057.1 | 546.9 | 97416.1 | 100698.1 | 97851 | 99605 |
| 80x12 | 113184.8 | 514.6 | 111641.1 | 114728.5 | 112018 | 114040 |
| 10x24 | 27987 | 335.1 | 26981.6 | 28992.4 | 27722 | 28410 |
| 20x24 | 55547.6 | 460.6 | 54165.7 | 56929.5 | 54820 | 56051 |
| 30x24 | 82909 | 476.8 | 81478.7 | 84339.3 | 82590 | 83730 |
| 40x24 | 110034.8 | 762.3 | 107748.0 | 112321.6 | 108971 | 111105 |

[(a)] Lower and upper confidence interval $=\overline{x}\pm 3s$ with 99.7% confidence interval, s = standard deviation.

**Table 21.** Statistics parameters of running time computed by the push and pull methods under near-minimal storage capacities (B=1%).

| **Problem** | **Average (second)** | **Standard deviation** | **Lower confidence interval**[(a)] | **Upper confidence interval**[(a)] | Min. total cost | Max. total cost |
|---|---|---|---|---|---|---|
| 10x6 | 6.25 | 0.54 | 4.63 | 7.88 | 5.79 | 7.59 |
| 20x6 | 11.45 | 0.31 | 10.53 | 12.38 | 11.04 | 12.02 |
| 40x6 | 21.90 | 0.37 | 20.78 | 23.04 | 21.37 | 22.54 |
| 60x6 | 32.48 | 0.46 | 31.09 | 33.87 | 31.75 | 33.17 |
| 80x6 | 44.25 | 0.93 | 41.45 | 47.05 | 43.04 | 45.86 |
| 100x6 | 56.24 | 0.94 | 53.43 | 59.05 | 55.02 | 57.71 |
| 120x6 | 69.42 | 1.26 | 65.65 | 73.18 | 67.90 | 71.21 |
| 140x6 | 84.98 | 2.49 | 77.52 | 92.46 | 82.41 | 91.25 |
| 160x6 | 100.72 | 1.46 | 96.35 | 105.09 | 98.57 | 103.74 |
| 10x12 | 61.85 | 5.27 | 46.03 | 77.68 | 55.48 | 71.94 |
| 20x12 | 131.74 | 4.84 | 117.23 | 146.26 | 126.34 | 141.33 |
| 30x12 | 204.63 | 16.03 | 156.53 | 252.74 | 182.15 | 236.27 |
| 40x12 | 260.38 | 23.67 | 189.36 | 331.41 | 229.37 | 311.10 |
| 50x12 | 328.35 | 34.155 | 225.89 | 430.82 | 279.38 | 402.51 |
| 60x12 | 409.28 | 40.99 | 286.31 | 532.24 | 351.43 | 492.47 |
| 70x12 | 495.31 | 40.97 | 372.41 | 618.22 | 443.17 | 585.65 |
| 80x12 | 520.24 | 26.18 | 441.69 | 598.79 | 478.54 | 566.50 |
| 10x24 | 3257.11 | 608.49 | 1431.63 | 5082.59 | 2647.19 | 4108.03 |
| 20x24 | 6507.96 | 340.99 | 5485 | 7530.92 | 6159.20 | 6964.99 |
| 30x24 | 9103.45 | 416.01 | 7855.43 | 10351.48 | 8471.77 | 9610.50 |
| 40x24 | 14628.15 | 1370.03 | 10518.05 | 18738.25 | 11694.79 | 14786.62 |

[(a)]Lower and upper confidence interval $= \overline{x} \pm 3s$ with 99.7% confidence interval , s = standard deviation.

Tables 20 and 21, all total cost and running time values fall within the lower and upper bounds of the 99.7% confidence intervals around the average values. It indicates that the heuristic's performance is reliable, and low variability is a direct measure of its stability.

## 6. Conclusions

This study proposes a novel push-pull heuristic for solving the multi-item uncapacitated lot-sizing problem under near-minimal storage capacities**.** When capacity constraints are nearly minimal across multiple items, novel heuristics are required. Pior heuristics did not directly consider the tight storage capacity constraints. The just-in-time operation in the assembly automobile industry is difficult to share the storage capacity on the multi-item parts. The proposed heuristic can be applied to manage tight storage capacity while keeping multiple items. To compute the initial replenishment plan, authors implement a dynamic programming based on network flow to generate a single-item lot size plan for all periods under unlimited storage capacity of each period. The push procedure identifies iteratively the maximal sum of beginning inventory plus the replenishment quantity which moves to the next period without violating the near -minimal storage capacity. Each iteration will increase inventory cost with the ordering cost .The push and pull procedure requires fewer iterations during computation. In comparison, Gutiérrez et al.'s heuristic selects successive periods, resulting in more iterations and increased ordering costs to meet the near-minimal capacity constraints.The result of computation shows that the proposed heuristic performs well on the gap solution under near-minimal storage capacities. The running time of the proposed heuristic performs well on large-scale problem, whereas GAMS/CPLEX solver run with minimal run time on small-and medium-scale problem. However, in the sensitivity analysis, a near-minimal storage capacity constraint results in high inventory costs due to the increased frequency of orders.

Future research could expand this proposed heuristic for applying this proposed heuristic then evaluating in a assembly automobile plant to improve the practical applicability and credibility. Another extension, it could run with the stochastic demand or lead time constraints to make it applicable across a wider range of academic settings.

**Author Contributions:** Conceptualization, W.B. and D.H. and P.C.; methodology, W.B. and D.H.; software, W.B.; validation, W.B. and D.H.; formal analysis, W.B. and D.H.; investigation, W.B. and D.H.; resources, D.H.; data curation, W.B. and D.H.; writing—original draft preparation, W.B. and D.H. ; writing—review and editing, W.B.,D.H. and P.C.; visualization, W.B. and D.H.; supervision, P.C.; projectadministration, W.B. and D.H.; funding acquisition, W.B. and D.H. All authors have readand agreed to the published version of the manuscript

**Funding:** This study received no external funding.

**Data Availability Statement:** The data presented in this study are available and can request to the corresponding author.

**Acknowledgments:** The authors would like to express our sincere appreciations to all constructive comments and recommendations from reviewers leading to theimproved final version of this manuscript. This study was supported by the Faculty of Engineering at Kamphaeng Saen, Kasetsart University, Thailand.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## Appendix A

### *Appendix A.1*

Pseudo code of dynamic programming based on network flow

```
PROCEDURE MultiItemLotSizing
// d[N][T] ← demand matrix (rows: items, cols: periods)
// order[N][T] ← fixed ordering cost matrix
// weight[N]← per-item weight (for capacity constraint)
// pc[N][T]← per-unit production cost matrix
// h← unit holding cost
// Output:
// sol[N][T]← lot-sizes for each item and each period
// invencost[N] ← total inventory cost per item
CONST M ← 10^10// "infinite" penalty
CONST extra ← 1// cost offset for indexing

FOR g ← 1 TO N DO// for each item
    // 2. Compute sumdemand and cumdemand
    FOR k ← 1 TO T DO
        sumdemand[g][k] ← Σ_{i=1..T} d[g][i]
```

```
END FOR
cumdemand[g][1] ← sumdemand[g][1]
FOR k ← 2 TO T DO
    cumdemand[g][k] ← cumdemand[g][k−1] + sumdemand[g][k]
END FOR

// 3. Build DP network of size S = T + Σ_k sumdemand[g][k] + 1
S ← T + Σ_{k=1..T} sumdemand[g][k] + 1
ALLOCATE dcost[1..S][1..S]

// 4. Fill dcost for "period 0" (building initial inventory)
FOR inv ← 0 TO sumdemand[g][1] DO
    lot ← inv − 0 + 0
    IF lot > 0 THEN
        y ← 1
    ELSE
        y ← 0
    END IF
    dcost[1][inv+1] ← inv*h + lot*pc[g][1] + y*order[g][1] + extra
END FOR

// 5. Fill dcost for periods 1..T−1
node_i ← 1
FOR p ← 1 TO T−1 DO
    prev_node_i ← node_i
    FOR inv_prev ← 0 TO sumdemand[g][p] DO
        node_i ← prev_node_i + inv_prev
        FOR inv_curr ← 0 TO sumdemand[g][p+1] DO
            lot ← inv_curr − inv_prev + d[g][p]
            IF lot > 0 THEN
                y ← 1
            ELSE
                y ← 0
            END IF
            // compute holding penalty
            IF lot == 0 AND inv_prev − inv_curr == d[g][p] THEN
                hold ← inv_curr * h
            ELSE IF lot > 0 AND inv_prev − inv_curr + lot == d[g][p] THEN
                hold ← inv_curr * h
            ELSE
                hold ← M
            END IF
            // assemble cost
            IF hold == M THEN
                cost ← M
            ELSE
                cost ← hold + lot*pc[g][p] + y*order[g][p]
            END IF
            dcost[node_i][node_i + sumdemand[g][p+1] + 1] ← cost + extra
        END FOR
    END FOR
    node_i ← node_i + sumdemand[g][p+1] + 1
END FOR

// 6. Fill dcost for final period T
FOR inv_prev ← 0 TO sumdemand[g][T] DO
    node_i ← node_i + inv_prev
    lot ← 0 − inv_prev + d[g][T] // end inventory is forced to 0
    IF lot > 0 THEN
        y ← 1
    ELSE
        y ← 0
    END IF
    IF lot == 0 AND inv_prev − 0 == d[g][T] THEN
        hold ← 0 * h
    ELSE IF lot > 0 AND inv_prev − 0 + lot == d[g][T] THEN
        hold ← 0 * h
    ELSE
        hold ← M
    END IF
    IF hold == M THEN
        cost ← M
    ELSE
        cost ← hold + lot*pc[g][T] + y*order[g][T]
    END IF
    dcost[node_i][S] ← cost + extra
END FOR

// 7. Solve DP by backward recursion
ALLOCATE fn[1..S+1] ← 0
ALLOCATE fnmat[1..S][1..S]
// 7.1 Initialize last column
FOR i ← 1 TO S DO
    fnmat[i][S] ← dcost[i][S]
    END FOR
    fn[S] ← MIN_{i=1..S} fnmat[S][i]
    // 7.2 Recurrence
    FOR i ← S−1 DOWNTO 1 DO
        FOR j ← i TO S DO
            IF dcost[i][j] > 0 THEN
                fnmat[i][j] ← dcost[i][j] + fn[j+1]
            END IF
        END FOR
        fn[i] ← MIN_{j=i..S} fnmat[i][j]
    END FOR

    // 8. Trace optimal path
    INITIALIZE optimalsol[0..T+1][1..5] ← 0
    current_node ← 1
    FOR period ← 0 TO T DO
        // find next node j where fn[current_node] == fnmat[current_node][j]
        SELECT smallest j ≥ current_node such that fn[current_node] == fnmat[current_node][j]
        lot ← corresponding lot-size on arc (current_node→j)
        inv ← previous_inv − demand + lot
        optimalsol[period+1] ← (prev_inv, lot, demand, inv, arc_cost − extra)
        current_node ← j + 1
    END FOR

    // 9. Record item-level solution
    FOR p ← 1 TO T DO
        sol[g][p] ← max(0, optimalsol[p+1].lot)
    END FOR
    invencost[g] ← fn[1] − extra
END FOR

END PROCEDURE
```

## References

1. Love; S.F. Bounded Production and Inventory Models with Piecewise Concave Costs. Manag. Sci. **1973**, *20*, 313-318.
2. Loparic, M.; Pochet,Y.; Wolsey,L.A. The Uncapacited Lot-Sizing Problem with Sales and Safety Stocks. Math. Program. **2001**, *89*, 487-504.
3. Atamtürk, A.; and Küçükyavuz, A. Lot Sizing with Inventory Bounds and Fixed Costs: Polyhedral Study and Computation. Oper. Research. **2005**, *53*, 711-730.
4. Guan, Y.; Liu, T. Stochastic Lot-Sizing Problem with Inventory-Bounds an Constant Order-Capacities. Eur. J. Oper. Res. **2010**, *207*, 1398-1409.

5. Önal,M.; Heuvel, W. V.D.; Liu,T. A Note on the Economic Lot Sizing Problem with Inventory Bounds. Eur. J. Oper. Res. **2012**, *223*, 290-294.
6. Chu, C.; Chu, F.; Zhong, F.; Yang, S. A Polynomial Algorithm for a Lot-Sizing Problem with Backlogging, Outsourcing, and Limited Inventory, Comput. Ind. Eng. **2013**, *64*, 200-210.
7. Akbalik, A.; Penz, B.; Rapine, C. Multi-Item Uncapacitated Lot Sizing Problem with Inventory Bounds. Optim. Lett. **2015**, *9*, 143–154.
8. Brahimi1, N.; Absi, N.; Dauzère-Pérès, S.; and Kedad-Sidhoum, S. Models and Lagrangian Heuristics for a Two-Level Lot-Sizing Problem with Bounded Inventory. OR Spectrum. **2015**, *37*, 983-1006.
9. Melo, R.A.; Ribeiro, C.C. Formulations and Heuristics for the Multi-Item Uncapacitated Lot-Sizing Problem with Inventory Bounds. Int. J. Prod. Res. **2015**, *55*, 576-592.
10. Witt, A. A Heuristic for the Multi-Level Capacitated Lot Sizing Problem with Inventory Constraints. Int. J. Manag. Sci. Eng. Manag. **2019**, *14*, 249-252.
11. Mohammadi, A,; Shegarian, E. A Mixed Integer Linear Programming Model for the Multi-Item Uncapacitated Lot-Sizing Problem: a case study in the trailer manufacturing industry. Int. J. Multivar. Data Anal. **2017**,1, 173-199.
12. Sedeno-Noda, A.; Gutierrez, J.; Abdul-Julbar, B.; Sicilia, J. An *O* (*T* log *T*) Algorithm for the Dynamic Lot Size Problem with Limited Storage and Linear Costs. Comput. Optim. Appl. **2004**, *28*, 311-323.
13. Liu, X.; Tu, Y. Production Planning with Limited Inventory Capacity and Allow Stockout. Int. J. Prod. Econ. **2008**, *111*, 180-191.
14. Chu, F.; Chu, C. Single-Item Dynamic Lot-Sizing Models with Bounded Inventory and Outsourcing", IEEE Trans. Syst. Man. Hum. **2008**, *38*, 70-77.
15. Hwang, H.-C.; Heuvel, W.V.D. Improved Algorithms for a Lot-Sizing Problem with Inventory Bounds and Backlogging. Nav. Res. Logist. **2012**, *59*, 244-253.
16. Hwang, H.-C.; Heuvel, W. V. D.; Wagelmans, A. P. M. The Economic Lot- Sizing Problem with Lost Sales and Bounded Inventory. IIE TRANS. **2013**, *45*, 912-924.
17. Gutiérrez, J.; Colebrook, M.; Abdul-Jalbar, B.; Sicilia, J. Effective Replenishment Policies for the Multi-Item Dynamic Lot- Sizing Problem with Storage Capacities. Comput. Oper. Res. 2013, 40, 2844-2851.
18. Minner, S. A Comparison of Simple Heuristics for Multi-Product Dynamic Demand Lot-Sizing with Limited Warehouse Capacity. Int. J. Prod. Econ. 2009, 118, 305–310.
19. Wagner, H.M.; Whitin, T.M. Dynamic Version of the Economic Lot Size Model. Manag. Sci. **1958**, *5*, 89-96.
20. Gutiérrez, J.; Sedeño-Noda, A.; Colebrook, M.; Sicilia, J. A Polynomial Algorithm for the Production/Ordering Planning Problem with Limited Storage. Comput. Oper. Res., **2007**, 34, 934–937.
21. Toczylowski, E. An $O(T^2)$ Algorithm for the Lot-Sizing Problem with Limited Inventory Levels. In Proceedings of International Conference on Emerging Technologies and Factory Automation (ETFA) , Paris, France, 10-13 October 1995, pp.78-Available online: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=496709&tag=1 (accessed on 9
22. September 2024)
23. Ojeda,A. Multi-level production planning with raw-material perishability and inventory bounds. Ph.D. (Industrial Engineering) of Concordia University , Montreal, Canada , September, 2019

Boonphakdee, W.; Charnsethikul, P. Column Generation Approach for Solving Uncapacitated Dynamic Lot-Sizing Problems with Time-Varying Cost.Int. J. Math. Oper. Res. **2022**, *23*, 55-75.

24. Di Summa, M.; Wolsey, L.A. Lot-Sizing with Stock Upper Bounds and Fixed Charges. SIAM J. Discret. Math.**2010**, 24, 853- 875.

25. Park, Y.B. An Integrated Approach for Production and Distribution Planning in Supply Chain Management. Int. J. Prod. Res. **2005**, 43, 1205-1224.
26. Emde, S. Sequencing Assembly Lines to Facilitate Synchronized Just-In-Time Part Supply. **J. Sched.2019**, 22, 607–621.

27.Dixon, P.S.; Poh, C.L. Heuristic Procedures for Multi-Item Inventory Planning with Limited Storage. IIE TRANS. **1990**, *22*,112-123.

27. 28.MATLAB code for Heuristic for the multi-item lot-sizing with storage capacities available online : https://www.math works.com/matlabcentral/fileexchange/179289-heuristic-for-the-multi-item-lot-sizinng-with-storage-cap, MATLAB Central File Exchange. Retrieved January 18, 2025.
28. MATLAB code for Heurictic of Gutierrez et al. 2013 algorithm Available online: https://www.mathworks.com/matlabcen tral/fileexchange/179294-heurictic-of-gutierrez-et-al-2013-algorithm, MATLAB Central File Exchange. RetrievedJanuary 18, 2025.
29. MATLAB code for Smoothing Heuristic Multi-item Lot size with Storage cap. Available online: https://www.mathworks.com/matlabcentral/fileexchange/181148-smoothing-heuristic-multi-item-lot-size-with-storage-cap, MATLAB CentralFile Exchange. Retrieved May 15, 2025.
30. CPLEX solver Available online: https://documentation.aimms.com/platform/solvers/cplex.html (accessed on 23 Dec 23, 2024)

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.