A polynomial algorithm for a simple scheduling problem at cross docking terminals

Ruslan Sadykov

A polynomial algorithm for a simple scheduling problem at cross docking terminals

Ruslan Sadykov*

INRIA Research Report

September 1, 2009

Abstract

At cross docking terminals, products from incoming trucks are sorted according to there destinations and transferred to outgoing trucks using a temporary storage. Such terminals allow companies to reduce storage and transportation costs in supply chain. This paper focuses on the operational activities at cross docking terminals.

We consider the trucks scheduling problem with the objective to minimise the storage usage during the product transfer. We show that a simplification of this NP-hard problem in which the arrival sequences of incoming and outgoing trucks are fixed is polynomially solvable by proposing a dynamic programming algorithm for it.

1 Introduction

Cross docking terminal is a distribution center carrying a considerably reduced amount of stock in contrast to traditional warehouses. Incoming shipments delivered by incoming trucks are unloaded, sorted and loaded onto outgoing trucks waiting at the dock, which forward the shipments to the respective locations within the distribution system. Compared to traditional warehousing, a cost intensive storage and retrieval of goods is eliminated by a synchronization of inbound and outbound flows. An additional advantage of cross docking is efficient usage of truck capacity (i.e. full loads) and the implementation of good scheduling system [1].

In this paper, we consider a simplified cross docking terminal with one receiving, one shipping door and a storage. The incoming trucks fully loaded by products of different types arrive at the receiving door where the products are unloaded. Each truck leaves the platform when it is fully unloaded. Each unloaded product is processed and transferred to the shipping door if the outgoing truck currently staying there demands the products of this type and does

*INRIA Bordeaux — Sud-Ouest, France, e-mail: `Ruslan.Sadykov@inria.fr`

not yet have enough of them. Otherwise, the unloaded product is transferred to the temporary storage. Each outgoing truck can leave the door when it is fully loaded by products moved either directly from incoming trucks or from the storage. The objective is to find the best arrival (or departure) order for incoming and outgoing trucks mixed together to increase the efficiency of the cross docking terminal.

There are several papers in the literature which dealt with this scheduling problem. Yu and Egbelu [4] developed a model for scheduling incoming and outgoing trucks to minimise the makespan (i.e. the maximum completion time) and proposed several heuristic algorithms for it. A simpler but similar problem was considered by Boysen, Fliedner and Scholl [2]. They have proposed some lower bounds and an exact decomposition approach for it. Maknoon, Baptiste and Kone [3] concentrated on the objective function which minimizes the storage cost in a simplified setting in which the sequences of incoming and outgoing trucks are fixed and any outgoing truck demands the products of only one type.

We now define the problem formally. The number of incoming and outgoing trucks are, respectively, n and m. The number of different product types is T. We will denote the i-th incoming truck as $\mathbf{I}_i$ and the o-th outgoing truck as $\mathbf{O}_o$. There are several types of products. Each incoming truck $\mathbf{I}_i$ supplies a_{it} products of type t. Each outgoing truck $\mathbf{O}_o$ demands b_{ot} products of type t. Let also T_o be the set of product types demanded by $\mathbf{O}_o$, i.e. $T_o = \{t : b_{ot} > 0\}$. We suppose that each outgoing truck demands products of at most q different types, i.e $q = \max_{1 \leq o \leq m} \mid T_o \mid$. Note that, for each type t, the total number of supplied products of this type should not be less than the total number of demanded products of this type, otherwise at least one outgoing truck would not be able to depart fully loaded.

A product of type t can be moved directly from an incoming truck at the receiving door to an outgoing truck $\mathbf{O}_o$ at the shipping door if the number of products of type t which are already in $\mathbf{O}_o$ is less than b_{ot}. Otherwise the product can be transferred to the temporary storage at a cost c_t. One product of type t occupies a volume d_t in the storage. The total volume of products in the storage should not exceed its total capacity which is D. The problem consists in finding a policy for unloading and loading products such that the total cost is minimized. For convenience, an equivalent objectif is used which is the maximization of the total cost of the directly transferred products. This problem is NP-hard in the strong sense even for a very restricted case, as shown in Appendix A.

Economically, the storage cost can be also interpreted as the difference between the time needed to transfer a product directly from an incoming to an outgoing truck and via the intermediate storage. So, out objective function is equivalent to the total processing cost. Although this objective is similar to the makespan objective used in [2] and [4], the problems studied there are quite different. The main difference is that we do not allow any concurrent operations. In contrast to this, for example in [4], on can at the same time discharge a product from an incoming truck to the storage and load a product from the storage onto an outgoing truck. As a result, transferring a product via the storage does

not always increase the objective function.

In the rest of the paper we concentrate on the problem in which the sequences of incoming and outgoing trucks are fixed. Formally this means that, for each $i \in \{1, \ldots, n-1\}$, the truck $\mathbf{I}_i$ departs before the arrival of $\mathbf{I}_{i+1}$, and for each $o \in \{1, \ldots, m-1\}$, the truck $\mathbf{O}_o$ departs before the arrival of $\mathbf{O}_{o+1}$.

Note that fixing sequences of incoming and outgoing trucks in problems considered in [2] and [4] make them trivial, and thus polynomially solvable. However, determining the computational complexity of our problem with fixed sequences is not trivial at all. In [3], Maknoon at al. considered this problem with some restrictions including $q = 1$, but left the complexity question open.

In this paper, we answer this question by presenting a polynomial algorithm for the general case of the problem with fixed sequences of trucks.

2 Preliminary observations

In the problem, we need to find an optimal policy which is characterized, among others, by the departure order of trucks. Note that the departure order is fixed within the set of incoming or outgoing trucks but not fixed within the set of all trucks mixed together. A departure order can be represented by a path in the graph depicted in Figure 1. In this graph, each node (i, o) represents the situation in which the trucks $\mathbf{I}_i$ and $\mathbf{O}_o$ are at the doors.

The path shown on this picture corresponds to the policy which is characterized by the following departure order of trucks:

$$(\mathbf{I}_1, \mathbf{I}_2, \mathbf{I}_3, \mathbf{O}_1, \mathbf{O}_2, \mathbf{I}_4, \mathbf{I}_5, \mathbf{O}_3, \mathbf{O}_4, \mathbf{I}_6, \ldots, \mathbf{I}_{n-1}, \mathbf{O}_6, \ldots, \mathbf{O}_{m-1}, \mathbf{I}_n, \mathbf{O}_m).$$

Some paths in the departure order graph are not feasible due to the following storage constraints. First, for each t, the number of products of type t in the storage can never be negative (all demanded products should be first supplied). Second, the overall volume of products in the storage can never exceed its capacity. We say the a node is *infeasible* if it does not belong to any feasible path. Formally, a node (i, o) is infeasible if and only if

$$\exists t : \sum_{k=1}^{i} a_{kt} < \sum_{p=1}^{o-1} b_{pt} \quad \text{or} \quad \sum_{t=1}^{T} d_t \cdot \left(\max \left\{ 0, \sum_{k=1}^{i-1} a_{kt} - \sum_{p=1}^{o} b_{pt} \right\} \right) > D. \quad (1)$$

From (1) it follows that

1. if two nodes (i', o) and (i'', o) such that $i' < i''$ are feasible then all nodes (i, o) such that $i' \leq i \leq i''$ are feasible;

2. if two nodes (i, o') and (i, o'') such that $o' < o''$ are feasible then all nodes (i, o) such that $o' \leq o \leq o''$ are feasible.

A possible situation of infeasible nodes which are in black is given in Figure 1. For a fixed o^*, we denote by $fi(o^*)$ and $li(o^*)$ the smallest and the largest values

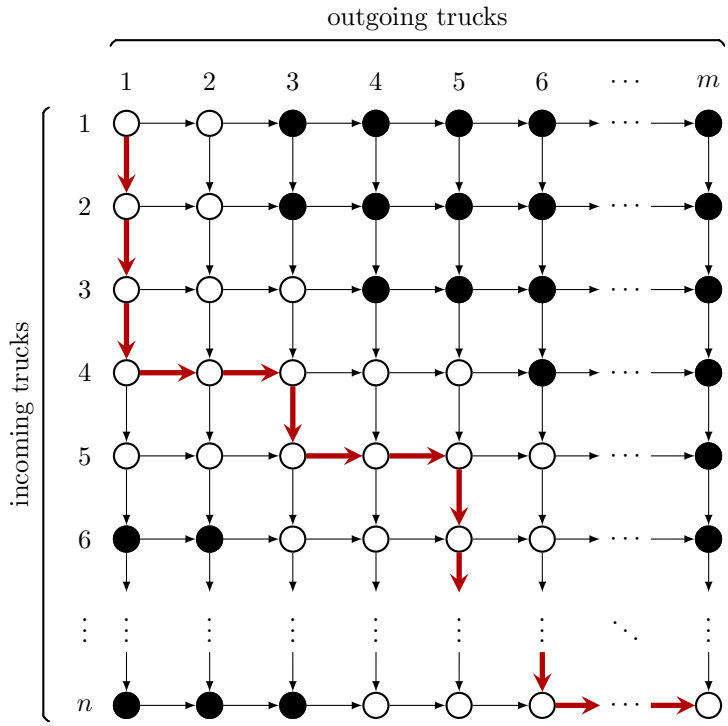

Figure 1: The departure order graph and an example of the path

of i such that node (i, o^*) is feasible. In the same manner, for a fixed i^*, we denote by $fo(i^*)$ and $lo(i^*)$ the smallest and the largest values of o such that node (i^*, o) is feasible.

The next fact is very important and serves as the base for the algorithm.

Observation 1 *There exists an optimal policy in which, each time trucks $\mathbf{I}_k$ and $\mathbf{O}_j$ are at the doors, for each t, $\mathbf{I}_k$ transfers directly to $\mathbf{O}_j$ as many products of type t as possible, i.e. the minimum between the number of products of type t still available in $\mathbf{I}_k$ and the number of products of type t which are still demanded by $\mathbf{O}_j$.*

It is easy to see that this observation is correct. Suppose, in an optimal policy, $\mathbf{I}_k$ does not transfer to $\mathbf{O}_j$ z products it can and "saves" them for consequent outgoing truck(s). Then $\mathbf{O}_j$ is obliged to take this m products from the storage. In the modified policy, these m products are transferred directly from $\mathbf{I}_k$ to $\mathbf{O}_j$, and products, transferred directly from $\mathbf{I}_k$ to consequent outgoing truck(s) in the original policy, are taken from the storage. The cost of the modified policy which comply with the observation do not increase.

We will call a policy which complies with Observation 1 *direct first*. Each path in the departure order graph corresponds to exactly one direct first policy.

3 The algorithm

We now present a dynamic programming algorithm for the problem which finds an optimal direct first policy.

Each state in our dynamic programming algorithm does not correspond to a node in the departure order graph but to an edge, which a path can follow just after "turning". To clarify the presentation, we first present in Figure 2 the underlying directed graph of the dynamic programming algorithm. In this graph, each node corresponds to a set of states of the algorithm, each edge defines all possible moves between a state in the origin of this edge and a state in the destination of this edge. The paths shown in Figures 1 and 2 correspond to the same policy. Again, the black nodes in the underlying directed graph are infeasible nodes.

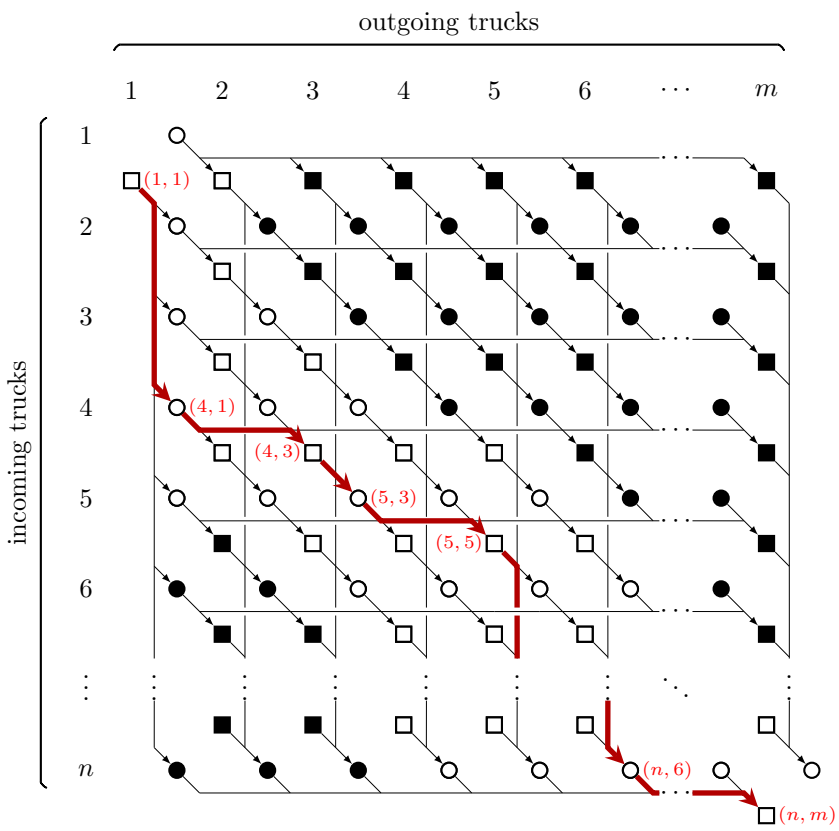

Figure 2: The underlying directed graph for the dynamic programming algorithm

Each "square node" (i, o) in the graph in Figure 2 represents the situation in which truck $\mathbf{I}_i$ has departed, truck $\mathbf{I}_{i+1}$ is going to arrive, truck $\mathbf{O}_o$ is at the

shipping door, and $\mathbf{O}_o$ had arrived after the departure of the truck $\mathbf{I}_{i-1}$. This means that $\mathbf{O}_o$ could not get any products directly from $\mathbf{I}_{i-1}$.

In our dynamic programming algorithm, for each "square node" (i, o), we define the states $\mathbf{S}^{out}(i, o, \{f_t\}_{t \in T_o})$, where f_t is the number of products of type t transferred directly by truck $\mathbf{I}_i$ to $\mathbf{O}_o$.

Each "circle node" (i, o) represents the situation in which truck $\mathbf{O}_o$ has departed, truck $\mathbf{O}_{o+1}$ is going to arrive, truck $\mathbf{I}_i$ is at the shipping door, and $\mathbf{I}_i$ had arrived after the departure of the truck $\mathbf{O}_{o-1}$. This means that $\mathbf{I}_i$ could not ship any products directly to $\mathbf{O}_{o-1}$.

In our dynamic programming algorithm, for each "circle node", we define the states $\mathbf{S}^{inc}(i, o, \{f_t\}_{t \in T_o})$, where f_t is the number of products of type t transferred directly by truck $\mathbf{I}_i$ to $\mathbf{O}_o$.

Now, every policy corresponds to exactly one sequence of states or a states path. Therefore, in the following, we will use the same notation for a states path and the corresponding policy. For each state $\mathbf{S}^{out}(i, o, \{f_t\}_{t \in T_o})$ and $\mathbf{S}^{inc}(i, o, \{f_t\}_{t \in T_o})$, we keep and update the objective function value of the best path to this state. Let $V^{out}(i, o, \{f_t\}_{t \in T_o})$ and $V^{inc}(i, o, \{f_t\}_{t \in T_o})$ be these values. To solve the problem, we need to find the best path terminating at a state $\mathbf{S}^{out}(n, m, \{f_t\}_{t \in T_o})$ or $\mathbf{S}^{out}(n, m, \{f_t\}_{t \in T_o})$.

To simplify the presentation, when there is no ambiguity, we will use the shortened notations $\mathbf{S}^{inc}(i, o, f)$, $\mathbf{S}^{out}(i, o, f)$, $V^{inc}(i, o, f)$, and $V^{out}(i, o, f)$, where f is a vector.

Note that, in the algorithm, we handle only the states belonging to at least one path which corresponds to a direct first policy. The total number of such states can be different for different instances of the same size.

Suppose we are in a state $\mathbf{S}^{inc}(i, o, f)$. From this state it is only possible to move to a state $\mathbf{S}^{out}(i, o', f')$, where $\max\{o+1, fo(i+1)\} \leq o' \leq lo(i)$ and $fo(n+1) = m$. When we make such a move, truck $\mathbf{I}_i$ transfers directly to every truck $\mathbf{O}_j$, $o < j \leq o'$, as much products as possible.

The formal procedure for making moves from a state $\mathbf{S}^{inc}(i, o, f)$ is presented in Algorithm 1. The complexity of this procedure is $O\big(m(q+\rho)\big)$, where ρ is the number of operations needed to check if a state has been already encountered earlier in the algorithm. We will compute ρ later in the paper.

Suppose now we are in a state $\mathbf{S}^{out}(i, o, f)$. From this state it is only possible to move to a state $\mathbf{S}^{inc}(i', o, f')$, where $\max\{i+1, fi(o+1)\} \leq i' \leq li(o)$, and $fi(m+1) = n$. When we make such a move, truck $\mathbf{O}_o$ receive directly from every truck $\mathbf{I}_k$, $i < k \leq i'$, as much products as possible.

The formal procedure for making moves from a state $\mathbf{S}^{inc}(i, o, f)$ is presented in Algorithm 2. The complexity of this procedure is $O\big(n(q+\rho)\big)$.

In the full algorithm, presented as Algorithm 3, we look through all the states and make all possible moves from them as described above. To obtain an optimal policy, it suffices to store, for each state, along the value V, the path which gives it. At the end of the algorithm, the best path stored for one of the states $\mathbf{S}^{out}(n, m, f)$ and $\mathbf{S}^{inc}(n, m, f)$ corresponds to an optimal policy.

We will now determine the complexity of the algorithm. The critical point here is estimating the total number of created states in the algorithm.

```
for t = 1 to T do  r[t] ← a_{it};
for t ∈ T_o do  r[t] ← a_{it} − f_t;
v ← V^{inc}(i, o, f);
for j ← o + 1 to lo(i) do
    for t ∈ T_j do
        dt[t] ← min{r[t], b_{jt}};
        v ← v + c_t · dt[t];
    if state S^{out}(i, j, {dt[t]}_{t∈T_j}) does not exist then
        create it: V^{out}(i, j, {dt[t]}_{t∈T_j}) ← −∞;
    if j ≥ fo(i + 1) and v > V^{out}(i, j, {dt[t]}_{t∈T_j}) then
        V^{out}(i, j, {dt[t]}_{t∈T_j}) ← v;
    for t ∈ T_j do
        r[t] ← r[t] − dt[t];
        dt[t] ← 0;
```

Algorithm 1: Algorithm to make moves from a state $\mathbf{S}^{inc}(i, o, f)$.

```
for t ∈ T_o do  r[t] ← f_t;
v ← V^{out}(i, o, f);
for k ← i + 1 to li(o) do
    for t ∈ T_o do
        dt[t] ← min{b_{ot} − r[t], a_{kt}};
        v ← v + c_t · dt[t];
    if state S^{inc}(k, o, {dt[t]}_{t∈T_o}) does not exist then
        create it: V^{inc}(k, o, {dt[t]}_{t∈T_o}) ← −∞;
    if k ≥ fi(o + 1) and v > V^{inc}(k, o, {dt[t]}_{t∈T_o}) then
        V^{inc}(k, o, {dt[t]}_{t∈T_o}) ← v;
    for t ∈ T_o do r[t] ← r[t] + dt[t];
```

Algorithm 2: Algorithm to make moves from a state $\mathbf{S}^{out}(i, o, f)$.

```
for t ∈ T_1 do dt[t] ← min{a_1t, b_1t};
v ← Σ_{t∈T_1} c_t · dt[t];
V^inc(1, 1, {a_1t − dt[t]}_{t∈T_1}) ← v;
run Algorithm 1 for the state S^inc(1, 1, {dt[t]}_{t∈T_1});
V^out(1, 1, {dt[t]}_{t∈T_1}) ← v;
run Algorithm 2 for the state S^out(1, 1, {dt[t]}_{t∈T_1});
for i ← 1 to n do
    if i > 1 then
        for o ← fo(i) to min{lo(i − 1), lo(i) − 1} do
            run Algorithm 1 for all created states S^inc(i, o, f);
    if i < n then
        for o ← max{fo(i) + 1, fo(i + 1)} to lo(i) do
            run Algorithm 2 for all created states S^inc(i, o, f);
return max {V^out(n, m, f), V^inc(n, m, f)}
```

Algorithm 3: The full algorithm

To do it, we will need the following lemma. But first we define some additional notations. Let $\mathcal{P}^{out}(i,o,f)$ and $\mathcal{P}^{inc}(i,o,f)$ be the sets of paths which teminate at states $\mathbf{S}^{out}(i,o,f)$ and $\mathbf{S}^{inc}(i,o,f)$. Let also $\mathcal{P}^{out}(i,o,f,o',f')$ be the set of paths which terminate at state $\mathbf{S}^{out}(i,o,f)$ and contains state $\mathbf{S}^{inc}(i,o',f')$ such that $o' < o$. Similarly, let $\mathcal{P}^{inc}(i,o,f,i',f')$ be the set of paths with terminate at state $\mathbf{S}^{inc}(i,o,f)$ and contains state $\mathbf{S}^{out}(i',o,f')$ such that $i' < i$. Finally, for a path P, we will denote as P_- the sub-path in which the terminating state of P is excluded.

Lemma 1

1. *For any two paths $P' \in \mathcal{P}^{out}(i,o,f')$ and $P'' \in \mathcal{P}^{out}(i,o,f'')$, $f' \neq f''$, which correspond to direct first policies, if, for some type $t' \in T_o$, $f'_{t'} < f''_{t'}$, then $f'_t \leq f''_t$ for all types $t \in T_o$.*

2. *Analogously, for any two paths $P' \in \mathcal{P}^{inc}(i,o,f')$ and $P'' \in \mathcal{P}^{inc}(i,o,f'')$, $f' \neq f''$, which correspond to direct first policies, if, for some type $t' \in T_o$, $f'_{t'} < f''_{t'}$, then $f'_t \leq f''_t$ for all types $t \in T_o$.*

Proof. We will prove this lemma by induction.

Suppose that the claim 2 is true for $i \leq i^*$ and $o < o^*$. We will prove the claim 1 for $i = i^*$ and $o = o^*$. Without loss of generality, let $P' \in \mathcal{P}^{out}(i^*, o^*, f', o', \bar{f}')$ and $P'' \in \mathcal{P}^{out}(i^*, o^*, f'', o'', \bar{f}'')$, $f' \neq f''$. As $f'_{t'} < f''_{t'}$, truck $\mathbf{O}_{o^*}$ receives less products of type t' from $\mathbf{I}_{i^*}$ in policy P' than in policy P''. Therefore, $\mathbf{I}_{i^*}$ transfers to trucks $\mathbf{O}_o$, $o' \leq o < o^*$ more products of type t' in policy P' than to all trucks $\mathbf{O}_o$, $o'' \leq o < o^*$, in policy P''. As P' and P'' are direct first policies, there are two possible cases:

- $o' = o''$ and $\bar{f}'_{t'} > \bar{f}''_{t'}$. Then, as $P'_- \in \mathcal{P}^{inc}(i^*, o', \bar{f}')$ and $P''_- \in \mathcal{P}^{inc}(i^*, o', \bar{f}'')$, by claim 2, $\bar{f}'_t \geq \bar{f}''_t$ for all types $t \in T_{o'}$.
- $o' < o''$.

In both cases, as P' and P'' are direct first policies, for every t, truck $\mathbf{I}_{i^*}$ does not transfer less products of type t to all trucks $\mathbf{O}_o$, $o' \leq o < o^*$ in policy P' than to all trucks $\mathbf{O}_o$, $o'' \leq o < o^*$, in policy P'', and $f'_t \leq f''_t$ for all types $t \in T_{o^*}$.

Analogously, if we suppose that claim 1 is true for $i < i^*$ and $o \leq o^*$, we can prove the claim 2 for $i = i^*$ and $o = o^*$.

As the base of the reduction for the claim 1, we can take the case $i = 2$. In this case, we can only have $o' < o''$, as, for any fixed o^*, there is only one path which terminates at a state $\mathbf{S}^{inc}(2, o^*, f)$.

Analogously, as the base of the reduction of the claim 2, we can take the case $o = 2$. □

Proposition 1 *The total number of created states in Algorithm 3 is $O(qnm^2)$.*

Proof. In the following, when a path contains a state $\mathbf{S}^{out}(i, o, f)$ or a state $\mathbf{S}^{inc}(i, o, f)$, we will say that, for each type t,

- if $t \in T_o$, the path contains triple $(i, o, f)^{out}_t$ or $(i, o, f)^{inc}_t$;
- if $t \notin T_o$, the path contains triple $(i, o, 0)^{out}_t$ or $(i, o, 0)^{inc}_t$.

Let now fix type t^*. A path can contain $O(nm)$ triples $(i, o, 0)^{out}_{t^*}$, $(i, o, b_{ot^*})^{out}_{t^*}$, $(i, o, a_{it^*})^{inc}_{t^*}$, and $(i, o, 0)^{inc}_{t^*}$. We will call such triples t^*-*canonical*.

Consider a path P' which contains triple $(i', o', f')^{out}_{t^*}$ and corresponds to a direct first policy. In this policy, truck $\mathbf{O}_{o'}$ receives directly from every truck $\mathbf{I}_i$, $i' < i \leq i''$, $\nu^{out}_{i'io't^*}(f')$ products of type t^*, where

$$\nu^{out}_{i'io't^*}(f') = \max\left\{0, \min\left\{a_{it^*}, b_{o't^*} - f' - \sum_{k=i'+1}^{i-1} a_{kt^*}\right\}\right\}.$$

Therefore, path P contains a triple $(i'', o', \nu^{out}_{i'io't^*}(f'))^{inc}$, where $i'' > i'$. An important observation is that there is at most one value i'' for which such triple is not t^*-canonical. We can say that any triple $(i', o', f')^{out}_{t^*}$ "generates" at most one triple $(i'', o', f'')^{inc}_{t^*}$ such that $i'' > i'$ which is not t^*-canonical.

Consider now a path P' which contains triple $(i', o', f')^{inc}_{t^*}$ and corresponds to a direct first policy. In this policy, truck $\mathbf{I}_{i'}$ transfers directly to every truck $\mathbf{O}_o$, $o' < o \leq o''$, $\nu^{inc}_{i'o'ot^*}(f')$ products of type t^*, where

$$\nu^{inc}_{i'o'ot^*}(f') = \max\left\{0, \min\left\{b_{ot^*}, a_{i't^*} - f' - \sum_{j=o'+1}^{o-1} b_{jt^*}\right\}\right\}.$$

Therefore, path P contains a triple $(i', o'', \nu^{inc}_{i'o'ot^*}(f))^{out}_{t^*}$, where $o'' > o'$. Again, an important observation is that there is at most one value o'' for which such

triple is not t^*-canonical. We can say that any triple $(i', o', f')^{inc}_{t^*}$ "generates" at most one triple $(i', o'', f')^{out}_{t^*}$ such that $o'' > o'$ which is not t^*-canonical.

We now fix a value o^*. From the above analysis, it follows that every t^*-canonical triple $(i, o, f)^{inc}_{t^*}$ or $(i, o, f)^{out}_{t^*}$ such that $o \leq o^*$ "generates" at most one non-t^*-canonical triple $(i', o^*, f')^{inc}_{t^*}$ or $(i', o^*, f')^{out}_{t^*}$ such that $i' > i$. Therefore, the number of different triples $(i, o^*, f)^{inc}_{t^*}$ or $(i, o^*, f)^{out}_{t^*}$ is $O(nm)$.

A straightforward analysis now indicates that the total number of states in the algorithm is $O((nm^2)^q)$. However, it is possible to get rid of the exponent q by the following reasoning.

We fix values i^* and o^*, and define a complete lexicographic order for states $\mathbf{S}^{out}(i^*, o^*, f)$. A state $\mathbf{S}^{out}(i^*, o^*, f')$ is lexicographically smaller than a state $\mathbf{S}^{out}(i^*, o^*, f'')$, $f'' \neq f'$, if and only if $f'_t \leq f''_t$ for all $t \in T_o$. It is always possible to compare in this way two such states, as we cannot have $f'_{t'} < f''_{t'}$ and $f'_{t''} > f''_{t''}$ for any two types $t', t'' \in T_{o^*}$ by Lemma 1. Now, the total number of states $\mathbf{S}^{out}(i^*, o^*, f)$ is equal to the sum, for every type $t \in T_{o^*}$, of different triples $(i^*, o^*, f)^{out}_t$, as, in order to pass from a state $\mathbf{S}^{out}(i^*, o^*, f)$ to the lexicographically next state, at least one of the values f_t, $t \in T_{o^*}$ should be increased. Analogously, the total number of states $\mathbf{S}^{inc}(i^*, o^*, f)$ is equal to the sum, for every type $t \in T_{o^*}$, of different triples $(i^*, o^*, f)^{inc}_t$.

Consequently, for every fixed value o^*, the total number of states $\mathbf{S}^{out}(i, o^*, f)$ and $\mathbf{S}^{inc}(i, o^*, f)$ is equal to $O(qnm)$. From this, we conclude that the total number of states is $O(qnm^2)$. □

We now return to the question of estimation ρ, which is the number of operations needed to check whether a state $\mathbf{S}^{out}(i, o, f)$ or $\mathbf{S}^{inc}(i, o, f)$ has been already created. From the proof of Proposition 1, remember that, given fixed values i^* and o^*, the number of different states $\mathbf{S}^{out}(i^*, o^*, f)$ is not more than $O(qnm)$. As these states can be lexicographically ordered, the storage and the search of these states can be done using a binary tree. Therefore, to check whether a state $\mathbf{S}^{out}(i, o, f)$ or, analogously, a state $\mathbf{S}^{inc}(i, o, f)$, has been already created and retrieve the value $V^{out}(i, o, f)$ or $V^{inc}(i, o, f)$ we need $O\big(\log(qnm)\big)$ operations. This concludes the estimation of the complexity of the algorithm.

Proposition 2 *The complexity of the dynamic programming algorithm is* $O\big(qnm^2(n+m)(q + \log m + \log n)\big)$. □

4 Conclusion

In this paper we presented a polynomial dynamic programming algorithm for a scheduling problem with fixed sequences of trucks to minimise the storage cost in cross docking terminal. This algorithm allows to determine the computational complexity of the problem for the first time.

Although polynomial, the complexity of the algorithm is high. However, the theoretical estimation of the number of created states can be much higher that

what we have in reality. Therefore, we hope that the algorithm may be used in practice especially when coupled with some bounding procedure and when q is small. As an indirect evidence of this, we can mention good experimental results reported in [3] for an algorithm proposed for the case $q = 1$ of our problem. That algorithm resembles our dynamic programming one, but its correctness was not shown formally.

One interesting direction for a future research is to try to find a linear programming formulation for the problem, since it was shown to be polynomially solvable. Such a formulation would help a lot in developing methods for solving more practical generalisation of the problem in which the truck sequences are not fixed.

References

[1] U. M. Apte and S. Viswanathan. Effective cross dockng for improving distribution efficiencies. *International Journal of Logistics Research and Applications*, 3:291–302, 2000.

[2] N. Boysen, M. Fliender, and A. Scholl. Scheduling inbound and outbound trucks at cross docking terminals. *OR Spectrum*, page to appear, 2008.

[3] M.Y. Maknoon, P. Baptiste, and O. Kone. Optimal loading and unloading policy in cross docking platform. In *Proceedings of 13th IFAC Symposium on Information Control Problems in Manufacturing*, pages 1263–1268, Moscow, Russia, June 2009.

[4] W. Yu and P. J. Egbelu. Scheduling of inbound and outbound tracks in cross docking systems with temporary storage. *European Journal of Operations Research*, 184:377–396, 2008.

Appendix A NP-hardness proof

Here we consider the general problem in which the sequences of trucks are not fixed. We show the this problem is NP-hard in the strong sense even for the case in which each incoming truck supplies products of at most two types, each outgoing truck demands products of at most one type, all the storage costs and product volumes are unitary, and the storage capacity is unlimited. We will perform a reduction from the 3-partition problem.

Remember that, in the 3-partition problem, we are given an integer B and a set of $3n$ integers $r_1, r_2, \ldots, r_{3n}$ such that $\sum_{i=1}^{3n} r_i = Bn$ and $B/4 < r_i < B/2$ for each i. We need to decide whether there exists a partition of the set of indexes $\{1, 2, \ldots, 3n\}$ into n sets $\{A_1, A_2, \ldots, A_n\}$ such that $\sum_{i \in A_j} r_i = B$, $\forall j = 1, \ldots, n$. Note that, if such a partition exists, each subset A_j contains exactly 3 indexes.

Given an instances of the 3-partition problem, we now define the corresponding instance of our cross docking problem. There are $3n$ incoming, $4n$ outgoing

trucks ($3n$ of the first type, n of the second type) and two types of products. The supplies and demands are the following:

$$\begin{array}{ll} a_{i1} = 1, & i = 1, \ldots, 3n, \\ a_{i2} = 2n + r_i, & i = 1, \ldots, 3n, \\ b_{i1} = 1, & i = 1, \ldots, 3n, \\ b_{i2} = 0, & i = 1, \ldots, 3n, \\ b_{i1} = 0, & i = 3n+1, \ldots, 4n, \\ b_{i2} = 6n + B, & i = 3n+1, \ldots, 4n. \end{array}$$

We claim that there exists a 3-partition if and only if at most n products are transferred via the storage.

Suppose that there exists a 3-partition $\{A_1, A_2, \ldots, A_n\}$, where $A_j = \{i_{j1}, i_{j2}, i_{j3}\}$. Then, the trucks are sequenced in n groups. Group j, $1 \leq j \leq n$, includes set A_j of incoming trucks and outgoing trucks $\mathbf{O}_{2j-1}$, $\mathbf{O}_{2j}$ and $\mathbf{O}_{3n+j}$. The departure order of group j is the following

$$\mathbf{O}_{2j-1}, \mathbf{I}_{i_{j1}}, \mathbf{I}_{i_{j2}}, \mathbf{O}_{3n+j}, \mathbf{I}_{i_{j3}}, \mathbf{O}_{2j}.$$

As $r_{i_{j1}} + r_{i_{j2}} + r_{i_{j3}} = B$, the incoming trucks transfer $6n + B$ products of type 2 directly to truck $\mathbf{O}_{3n+j}$, $\mathbf{I}_{i_{j1}}$ transfers 1 product of type 1 directly to $\mathbf{O}_{2j-1}$, and $\mathbf{I}_{i_{j3}}$ transfers 1 product of type 1 directly to $\mathbf{O}_{2j}$. Only 1 product of type 1 is transferred to the storage from $\mathbf{I}_{i_{j2}}$. These transfers are depicted in Figure 3. As there are n groups, n products in total are transferred to the storage and then put to the outgoing trucks $\mathbf{O}_{2n+1}, \ldots, \mathbf{O}_{3n}$, which are sequenced at the end.

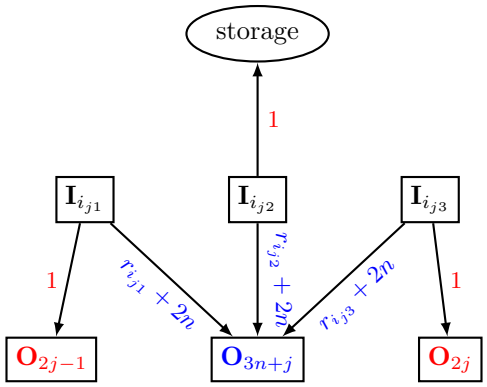

Figure 3: Product transfers within a group j of trucks

Suppose now there is a sequence of trucks such that at most n products are transferred through the storage. Then, every outgoing truck $\mathbf{O}_i$, $i = 3n+1, \ldots, 4n$ must receive products directly from at least 3 incoming trucks. Otherwise it would receive from the storage at least $2n$ products of type 2. Therefore, at least 1 product of type 1 goes to the storage while each such truck

$\mathbf{O}_i$ is supplied, and the total number of products of type 1 transferred through the storage is at least n, meaning that all products of type 2 should be transferred directly. Then, each incoming truck can transfer directly products of type 2 to exactly one outgoing truck. Otherwise, between two outgoing trucks of type 2, only one outgoing truck can receive a product of type 1 directly, and the total number of products of type 1 transferred through the storage would exceed n. We conclude that there should exist a partition of incoming trucks into triples $\{A_1, A_2, \ldots, A_n\}$ such that $\sum_{i \in A_j} r_i = B$. Otherwise there would exist a triple A_j such that $r_{i_{j1}} + r_{i_{j2}} + r_{i_{j3}} < B$, and the outgoing truck which is supplied by the incoming trucks in A_j would need to take at least one product of type 2 from the storage.