Computers and Mathematics with Applications 61 (2011) 630–641

Contents lists available at ScienceDirect

Computers and Mathematics with Applications

journal homepage: www.elsevier.com/locate/camwa

A complete and an incomplete algorithm for automated guided vehicle scheduling in container terminals

Hassan Rashidi [a,*], Edward P.K. Tsang [b,1]

[a] Department of Statistics, Mathematics, and Computer Science, Allameh Tabataba'i University, No. 4, Ahmad Ghasir St., Shahid Beheshti st., Tehran, Iran
[b] School of Computer Science and Electronic System Engineering, University of Essex, Colchester CO4 3SQ, UK

ARTICLE INFO

Article history:
Received 16 June 2009
Received in revised form 6 December 2010
Accepted 6 December 2010

Keywords:
Search methods
Scheduling problems
Network Simplex Algorithm
Optimization
Container terminals

ABSTRACT

In this paper, a scheduling problem for automated guided vehicles in container terminals is defined and formulated as a Minimum Cost Flow model. This problem is then solved by a novel algorithm, *NSA*+, which extended the standard Network Simplex Algorithm (*NSA*). Like *NSA*, *NSA*+ is a complete algorithm, which means that it guarantees optimality of the solution if it finds one within the time available. To complement *NSA*+, an incomplete algorithm Greedy Vehicle Search (*GVS*) is designed and implemented. The *NSA*+ and *GVS* are compared and contrasted to evaluate their relative strength and weakness. With polynomial time complexity, *NSA*+ can be used to solve very large problems, as verified in our experiments. Should the problem be too large for *NSA*+, or the time available for computation is too short (as it would be in dynamic scheduling), *GVS* complements *NSA*+.

0. Introduction

Scheduling problems arise in areas as diverse as production planning, personnel planning, product configuration, and transportation. An overview of the wide range of constraints in scheduling, together with the most powerful propagation algorithms for these constraints are given [1,2]. This paper has been motivated by a need to schedule Automated Guided Vehicles (*AGVs*) in container terminals. The container terminal components that are relevant to our problem include quay cranes (*QC*), container storage areas, rubber tyred gantry crane (*RTGC*) or yard crane, and a road network (see e.g. [3–8]). A transportation requirement in a port is described by a set of jobs, each of which is being characterized by the source location of a container, the destination location and its pick-up or drop-off times on the quay side by the quay crane. Given a number of *AGVs* and their availability, the task is to schedule the *AGVs* to meet the transportation requirements. This transportation problem is formulated as a Minimum Cost Flow (*MCF*) model.

Pricing scheme is certainly an important step in the Network Simplex Algorithm (*NSA*) since the total computational effort to solve a problem heavily depends on its choice. This step does two things. It checks whether the optimality conditions for the non-basic arcs are satisfied, and if not it selects a violated arc to enter the spanning tree structure [9]. The selected arc has a potential to improve the current solution. According to the theory [9] the *NSA* terminates in a finite number of iterations regardless of which profitable candidate is chosen if degeneracy is treated properly. Some well-known schemes in *NSA* are *the steepest edge scheme* (by Goldfarb and Kennedy [10]), the *Mulvey's list* (by Mulvey [11]), the *block pricing scheme* (by Grigoriadis [12]), the *BBG Queue pricing scheme* (by Bradley et al. [13]), the *clustering technique* (by Eppstein [14]), the *multiple pricing schemes* (by Lobel [15]), the *general pricing scheme* (by Maros [16]). Masakazu [17] proposed a new scheme in which the algorithm can start from an arbitrary pair of primal and dual feasible spanning tree. In this paper we present a new pricing scheme, which significantly reduces the CPU-time required to tackle the *MCF* model.

* Corresponding author. Tel.: +98 21 88725400 2.
E-mail addresses: hrashi@gmail.com, hrashi@atu.ac.ir (H. Rashidi), edward@essex.ac.uk (E.P.K. Tsang).
1 Tel.: +44 1206 872774.

0898-1221/$ – see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.camwa.2010.12.009

The structure of this paper is as follows. Section 1 presents the *MCF* model by a new notation. Then the scheduling problem of *AGV* is defined and formulated as a special case of the *MCF* model. We present our problem with the *MCF–AGV* model in Section 2. Section 3 presents two algorithms to tackle the *MCF–AGV* model. Experimental results from applying the two algorithms to tackle the problem are compared in Section 4. Section 5 includes the summary and conclusion.

1. Minimum Cost Flow (MCF) model

In this section, we systematically introduce a formal definition for the *MCF* model:

Definition 1 ([9]). In an informal description of the *MCF* model, let graph $G = (N, A)$ be a directed network defined by a set of nodes, N, together with a set of arcs, A. Each arc $(i, j) \in A$ has an associated cost c_{ij} that denotes the cost per unit flow on that arc. It is assumed that the flow cost varies linearly with the amount of flow. The maximum and minimum amount of flow on each arc $(i, j) \in A$ are limited by M_{ij} and m_{ij} ($m_{ij} \leq M_{ij}$), respectively. A real number b_i is associated with each node, representing its supply/demand. If b_i is greater (less) than zero, node i is a supply (demand) node; and if $b_i = 0$, node i is a transhipment node. The decision variables in the *MCF* model are arc flows, which are represented by f_{ij} for arc $(i, j) \in A$. The standard form of the *MCF* model is as follows:

$$\text{MinCostFlow} = \sum_{(i,j)\in A} c_{ij} f_{ij}$$

$$\text{Subject To} \begin{cases} \sum_{j:(i,j)\in A} f_{ij} - \sum_{j:(j,i)\in A} f_{ji} = b_i, & \text{for all } i \in N \\ m_{ij} \leq f_{ij} \leq M_{ij}, & \text{for all } (i, j) \in A. \end{cases}$$

These constraints state that flows must be feasible and conserve each node. For the feasible flows to exist the *MCF* model must also have $\sum_{i\in N} b_i = 0$, which means that the network is balanced. We now define a special graph for the *MCF* model as follows:

Definition 2. A *MCF* Graph $G_{MCF} = (G, NP, AP)$ consists of a graph G with a couple of properties for the nodes and arcs in G. The NP and AP are the Node's and Arc's Properties, respectively. The node property function $NP : N \to R$ (Real numbers; possibly negative) gives the amount of supply/demand of the nodes. This function for each node is defined as follows:

$$NP(i) = NP_i = b_i \quad \text{where} \begin{cases} b_i > 0 \text{ if node } i \text{ is a supply node} \\ b_i < 0 \text{ if node } i \text{ is a demand node} \\ b_i = 0 \text{ if node } i \text{ is a transshipment node} \end{cases} \text{ so that } \sum_{i\in N} NP(i) = 0.$$

Each arc in A has three properties: a lower bound, an upper bound and a cost. The arc property function AP maps each arc to these properties, $AP : A \to R \times R \times R$ (Real numbers; nonnegative). For each arc $\in A$, we denote the mapping by $AP(i, j)$, or AP_{ij} for short. We denote the lower bound, upper bound and cost by m_{ij}, M_{ij} and c_{ij}. Based on Definitions 1 and 2, we define the standard *MCF* model formally as follows:

Definition 3. A *MCF* model is defined as $MCF = (G_{MCF}, f, D, CS, FC)$ where $G_{MCF} = ((N, A), NP, AP)$ is a graph with nodes and arcs specific to the *MCF* model (Definition 2); f is a finite set of decision variables on A (f stands for flow), $f = \{f_{ij} \mid (i, j) \in A\}$; D = a function which determines a lower and an upper bound for f; $D : f \to R \times R$ (to be pulled out from AP); we shall take $D_{f_{ij}}$ as the lower bound and the upper bound of f_{ij} (D stands for Domain); CS is a finite set of Constraint on NP and f; FC is an objective function for the Flow's Cost on AP and f. The task in a *MCF* model is to assign a value to each f_{ij} that satisfy all constraints in CS with regard to the minimum value of FC. In the standard form of the *MCF* model we have:

(a) For each element in D and f, $D_{f_{ij}} = [m_{ij}, M_{ij}]$, for $\forall (i, j) \in A$;
(b) The CS is $\sum_{j:(i,j)\in A} f_{ij} - \sum_{j:(j,i)\in A} f_{ji} = NP_i$, for $\forall i \in N$;
(c) The FC is $\sum_{j:(j,i)\in A} c_{ij} f_{ij}$.

2. The special case of the *MCF* model for automated guided vehicle scheduling

In this section, a scheduling problem of *AGVs* in the container terminals is introduced. The problem is to deploy several *AGVs* in a port to carry many containers from the quay side to yard side or vice versa. The main reason for choosing this problem is that the efficiency of a container terminal is directly related to the use of the *AGVs* with full efficiency (see e.g. [3–7]). This problem is formulated as a special case of the *MCF* model.

2.1. Assumptions

Assumption 1. The layout of a port container terminal is given [18]. According to a specific layout, the travel time between every combination of Pick-up (P)/Drop-off (D) points is provided.

Assumption 2. The yard is divided into several blocks and *RTGCs* or yard crane resources are always available [4], i.e., the *AGVs* will not suffer delays in the storage yard location or waiting for the yard cranes.

Assumption 3. The quay side consists of several Quay Cranes (*QCs*). For each *QC*, there is a predetermined job sequence, consisting of loading or unloading jobs, or a combination of both. For each loading (unloading) job, there is a predetermined pick-up (drop-off) point in the yard, which is the origin (destination) of the job.

Assumption 4. There are *N* jobs and *M AGVs* in the problem. The source and destination of jobs as well as their appointment time on the quay side are given. Each job has an appointment time on the quay side and the jobs should be served in their appointment time by the *AGVs*.

Assumption 5. The problem is divided into two types, static and dynamic. In the static problem, we assume that the number of vehicles, the number of jobs and the distance between every two points in the container terminal do not change. In the dynamic problem the distance between every two points of the port may be changed. It is due to the fact that the system controller may change the route of *AGVs*, based on congestion in different points of the terminal. Hence it is assumed that there are no Collisions, Livelock and Deadlock [19] problems while the *AGVs* are carrying the containers.

Assumption 6. Our objectives are to minimize (a) the total *AGV* waiting time at the quay side (b) the total *AGV* travelling time in the route to the port (c) the total lateness times to serve the jobs. Cheng et al. minimized the impact of delays and waiting times of the *AGVs* at the quay side [4].

2.2. Formulation of the problem

Here, we present a special case of the *MCF* model for the scheduling problem of *AGVs* in a container terminal. The problem differs primarily in the arrangement of nodes and arcs with their properties. In this special case, the property function of nodes assigns integer values to the nodes. Additionally, the property function of arcs may assign integer values to the lower bound, the upper bound and the cost of each arc. Here, we present the special Graph of G_{MCF} for the *AGV* scheduling ($G_{MCF-AGV}$) and the special case of the *MCF* model for the scheduling problem of *AGVs* (*MCF–AGV*).

Based on Definition 2, we introduce the following definition for the G_{MCF} in a special case:

A *MCF* Graph for *AGV*, $G_{MCF-AGV} = (GS, NPS, APS)$, is a special case of $G_{MCF} = (G, NP, AP)$ (Definition 2). The graph $GS = (NS, AS)$ will be defined in the sub-sections below; the node and arc properties of *GS*, *NPS* and *APS*, are also special cases of *NP* and *AP*, respectively ($NPS : NS \rightarrow N$ and $APS : AS \rightarrow N \times N \times N$; *N* is the set of Natural numbers). We formally describe the components of $G_{MCF-AGV}$ in the following two sub-sections:

2.2.1. Nodes and their properties in the special graph

Let *N* be the number of jobs and *M* be the number of *AGVs* in the problem. The nodes of the *MCF* Graph for the *AGV* scheduling problem are defined as follows:

(a) *Supply nodes*: For each vehicle m, a supply node $AGVN_m$ with one unit supply is considered. Therefore, the set of supply nodes in the graph is $SAGVN = \{AGVN_m \mid m = 1, 2, \ldots, M; NPS(m) = 1\}$.

(b) *Transhipment nodes*: for each job *j*, a couple of nodes, Job-Input and Job-Output, are considered. Hence, the sets of transhipment nodes in the graph are $SJIN \cup SJOUT$ where:

$$SJIN = \{JIN_i \mid i = 1, 2, \ldots, N; NPS(i) = 0\} \quad \text{where } JIN_i \text{ is a node through which an } AGV \text{ enters job } i.$$

$$SJOUT = \{JOUT_i \mid i = 1, 2, \ldots, N; NPS(i) = 0\} \quad \text{where } JOUT_i \text{ is a node from which an } AGV \text{ leaves job } i.$$

(c) *SINK*: It stands for a Sink node or a demand node in the $G_{MCF-AGV}$ with *M* units demand. This node corresponds to the end state of the process, after all container jobs have been served. Hence, for the property of this node, $NPS(SINK) = -M$.

Therefore, there are $M + 2 * N + 1$ nodes in $G_{MCF-AGV} : NS = SAGVN \cup SJIN \cup SJOUT \cup SINK$.

2.2.2. Arcs and their properties in the special graph

Below we describe the four types of arcs that join the nodes in $G_{MCF-AGV}$, together with their properties:

(1) *Intermediate arcs*: These arcs are directed arcs from every Job-Output node *i* to every other Job-Input node *j*.
These arcs with their properties are $ARC_{\text{intermediate}} = \{(i, j) \mid i \in SJOUT, j \in SJIN, j \neq JIN_i, APS(m, j) = [0, 1, C_{ij}]\}$

$$\text{where } C_{ij} = \begin{cases} w_1 \times (t_j - (t_i + DT_{ij})) + w_2 \times DT_{ij} & \text{if } t_j \geq t_i + DT_{ij} \\ P \times (t_i + DT_{ij} - t_j) & \text{otherwise.} \end{cases}$$

In the cost, w_1 and w_2 are the weights of waiting and travelling times of the *AGVs*, respectively; t_j and t_j are the appointment time of jobs *i* and *j*, respectively, on the quay side (to be unloaded or dropped-off); DT_{ij} is travelling time from location of job *i* to location of job *j*; (calculation of the DT_{ij} is illustrated by Fig. 1 in different cases). If an *AGV* can serve job *j* after serving job *i* ($t_j \geq t_i + DT_{ij}$), the waiting and travelling times of the *AGV* are calculated without any lateness time. Otherwise, only the lateness time of serving job *j* with a penalty (*P*) is considered for the cost. This penalty cost causes to serve the job with minimum delay (see Assumption 6).

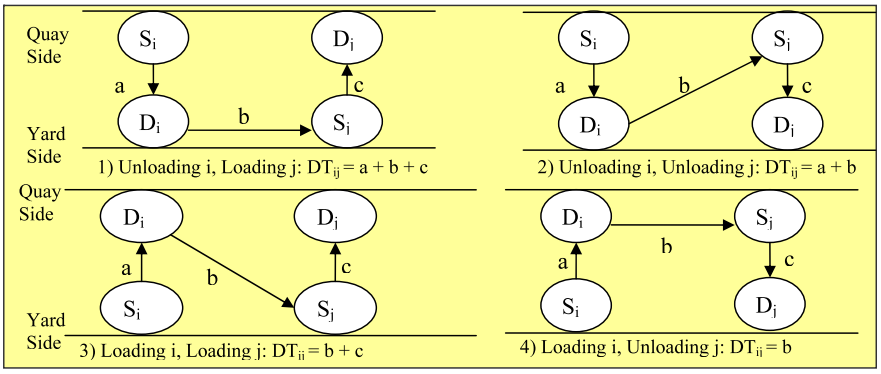

Fig. 1. Travelling time computations between two jobs.

(2) *Inward arcs*: a set of arcs from *SAGVN* to *SJIN*. These arcs along with their properties are:

$$ARC_{\text{inward}} = \{(m, j) \mid m \in SAGVN, j \in SJIN, APS(m, j) = [0, 1, C_{mj}]\}$$

$$\text{where } C_{mj} = \begin{cases} w_1 \times (t_j - (RTA_m + TTA_{mj})) + w_2 \times (RTA_m + TTA_{mj}) & \text{if } (t_j \geq RTA_m + TTA_{mj}) \\ P \times (RTA_m + TTA_{mj} - t_j) & \text{otherwise.} \end{cases}$$

In the cost, w_1 and w_2 are the weights of waiting and travelling times of the *AGVs*, respectively; t_j is the appointment time of job j at the quay side (to be unloaded or dropped-off); RTA_m is the ready time of *AGV* m at the start location, which may be either the quay side or the yard side; TTA_{mj} is the travel time of *AGV* m from the start location to the location of job j on the quay side; (the TTA_{mi} should be calculated in a similar manner as the calculation of DT_{ij}; see Intermediate arcs). If *AGV* m could arrive at the quay side in the appointment time of job j ($t_j \geq RTA_m + TTA_{mj}$), the waiting and travelling times of *AGV* m to serve job j are calculated as the cost. Otherwise, the lateness time to serving job j with a penalty (P) is considered.

(3) *Outward arcs*: These are directed arcs from every Job-Output node i and *AGV* node m to *SINK*. These arcs along with their properties are $ARC_{\text{outward}} = \{(i, j) \mid i \in SAGVN \cup SJOUT, j = SINK; APS(m, j) = [0, 1, 0]\}$. These arcs show that an *AGV* can remain idle after serving any number of jobs or without serving any job. Therefore, a cost of zero is assigned to these arcs.

(4) *Auxiliary arcs*: There is a directed arc from every Job-Input node i to its Job-Output node. These arcs along with their properties are $ARC_{\text{auxiliary}} = \{(i, j) \mid i \in SJIN, j =$ an unique Job-Output node in *SJOUT*, correspond to the Input-Node i; $APS(i, j) = [1, 1, 0]\}$. These arcs guarantee that every Job-Input and Job-Output nodes is visited once only so that each job is served.

There are $M \times N + N \times (N - 1) + M + 2 \times N$ arcs in the graph ($AS = ARC_{\text{inward}} \cup ARC_{\text{intermediate}} \cup ARC_{\text{outward}} \cup ARC_{\text{auxiliary}}$).

2.2.3. The MCF–AGV model for automated guided vehicle scheduling

Now we present our model for the *AGVs* Scheduling with the following definition:

Definition 4. A *MCF–AGV* model is a special case of the *MCF* (see Definition 3) for the scheduling problem of *AGVs* in the container terminals. A *MCF–AGV* model is defined as

$MCF\text{–}AGV = (G_{MCF\text{–}AGV}, f, D, CS, FC)$ where $G_{MCF\text{–}AGV} = (GS, NPS, APS)$ is a graph for the *MCF–AGV* problem; f = a finite set of integer decision variables on AS, $f = \{f_{ij} \mid (i, j) \in AS\}$; D = a function which determines a lower and upper bound for f; $D : f \to N \times N$ (to be pulled out from *APS*). For each element in D, we have:

$$\begin{cases} (1)\ D_{f_{ij}} = [0, 1]; & \text{for } \forall (i, j) \in (ARC_{\text{inward}} \cup ARC_{\text{intermediate}} \cup ARC_{\text{outward}}) \\ (2)\ D_{f_{ij}} = [1, 1]; & \text{for } \forall (i, j) \in ARC_{\text{auxiliary}}. \end{cases}$$

The constraints are given below,

$$CS = \left\{ \begin{array}{l} (1) \displaystyle\sum_{j:(i,j)\in AS} f_{ij} = 1;\ \forall i \in SAGVN;\ \text{Sending one unit flow into the network from each node in } SAGVN \\ (2) \displaystyle\sum_{j:(j,i)\in AS} f_{ji} = M;\ \text{for } i = SINK;\ \text{Receiving } M \text{ units flow (the flows sent from the nodes in } SAGVN \text{ set)} \\ (3) \displaystyle\sum_{j:(i,j)\in AS} f_{ij} - \sum_{j:(j,i)\in AS} f_{ji} = 0;\ \forall i \in SJIN \cup SJOUT;\ \text{Flow balance at every Job-Input and Job-Output node} \end{array} \right\}$$

and $FC = \sum_{(i,j)\in AS} C_{ij} \cdot f_{ij}$.

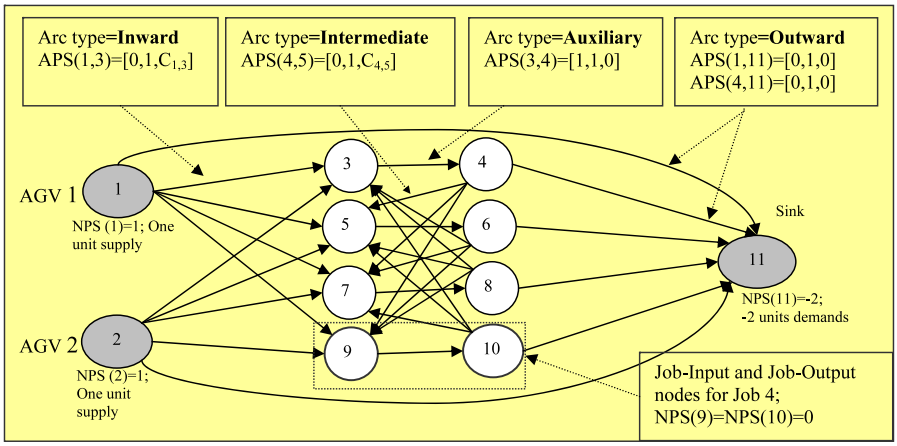

Fig. 2. An example of the *MCF–AGV* model for two *AGVs* and four jobs.

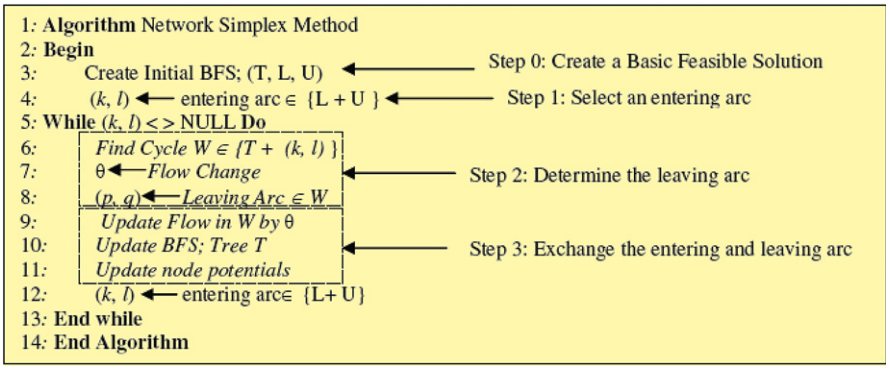

Fig. 3. The Network Simplex Algorithm (*NSA*).

The *MCF–AGV* model can be illustrated by Fig. 2 for two *AGVs* and four jobs. The problem has a huge search space and the solution should provide the optimal paths for each *AGV* from every vehicle node to the sink node. Solving the *MCF–AGV* model generates M paths, each of which commences from a vehicle node and terminates at the sink node. Each path determines a job sequence of every vehicle. Suppose that for some values of arc costs, the paths given by a solution are $1 \to 3 \to 4 \to 9 \to 10 \to 11$ and $2 \to 5 \to 6 \to 7 \to 8 \to 11$. This states that *AGV* 1 is assigned to serve jobs 1 and 4, and *AGV* 2 is assigned to serve jobs 2 and 3, respectively.

3. The algorithms

In this section, a complete algorithm (*NSA*+) and an incomplete search method (*GVS*) to tackle the *MCF–AGV* model are presented. Since *NSA*+ is an extension of the Network Simplex Algorithm (*NSA*), we describe *NSA* first.

3.1. Network Simplex Algorithm (*NSA*)

Every connected network has a spanning tree [9]. The Network Simplex Algorithm maintains a feasible spanning tree at each iteration and successfully goes toward the optimality conditions until it becomes optimal. At each iteration, the arcs in the graph are divided into three sets; the arcs belong to the spanning tree (T); the arcs with flow at their lower bound (L); the arcs with flow at their upper bound (U). A spanning tree structure (T, L, U) is optimal if the reduced cost for every arc $(i, j) \in L$ is greater than zero and at the same time the reduced cost for every arc $(i, j) \in U$ is less than zero [9]. With those conditions, the current solution is optimal. Otherwise, there are arcs in the graph that violate the optimal conditions. An arc is a violated arc if it belongs to $L(U)$ with negative (positive) reduced cost. The algorithm in Fig. 3 specifies the steps of the method [9,20,21].

To create the initial or Basic Feasible Solution (BFS) in Step 0, an artificial node 0 and artificial arcs are appended to the graph. The node '0' will be the root of the spanning tree (T) and the artificial arcs, with sufficiently large costs and capacities, connect the nodes to the root. The set L consists of the main arcs in the graph, and the set U is empty [9]. Selection of a pricing scheme is an important decision in Step 1. During this step the reduced costs of the non-basic arcs are recalculated. If there

```
1: arc  Entering_Arc_Function
2:          If Initialization is needed Then  // the packet is empty
3:             Calculate the number of blocks
4:             Choose the Block-Number  // Randomly or by a Heuristic method
5:          End if
6:          Recalculate the Reduced Costs of the most violated Arcs in the Packet  //  a number of elements at the top of the packet
7:          If the most violated elements satisfy the optimality conditions Then
8:             Remove the elements from the packet
9:          End if
10:         While the Packet has empty place AND there is any violated arc in the graph Do
11:            Calculate the reduced cost of an arc from the block associated with the Block-Number.
12:            Put the arc into the Packet if it violates the optimality condition.
13:            Increase the Block-Number circularly.
14:         End While
15:         If the Packet is Empty Then
16:            Return Null        // The Current Solution is Optimal
17:         End If
18:         Sort the Packet Descending   // Based on the absolute value of the reduced costs by Quick Sort
19:         Return the first element of the Packet
20: End Function
```

Fig. 4. Pseudo-code of selecting an entering arc in the Network Simplex plus Algorithm.

is at least one that violates its optimality condition it is a candidate to enter the basis. In Step 1, appending the entering arc (k, l), an arc with violation, to the spanning tree forms a unique cycle, W, with the arcs of the basis. In order to eliminate this cycle (Step 2), one of its arcs must leave the basis. The cycle is eliminated when we have augmented flow by a sufficient amount to force the flow in one or more arcs of the cycle to their upper or lower bounds. By augmenting the flow in a negative cost augmenting cycle, the objective value of the solution is improved. The first task in determining the leaving arc is the identification of all arcs of the cycle. The flow change is determined by the equation $\theta = \min\{f_{ij} \text{ for all } (i, j) \in W\}$. The leaving arc is selected based on cycle W. The substitution of entering for the leaving arc and the reconstruction of new tree is called a pivot (Step 3). After pivoting to change the basis, the reduced costs for each arc $(i, j) \notin T$ is calculated. If the reduced costs for all $(i, j) \in \{L + U\}$ satisfy the optimality condition then the current basic feasible solution is optimal. Otherwise, an arc (i, j) where there is a violation should be chosen and operations of the algorithm should be repeated.

Different pricing schemes are available for finding out an entering arc for the basic solution. The standard textbook [9] provided a detailed account of the literature on these schemes. The performance of the algorithm is affected by the choice of a proper pricing scheme.

3.2. The Network Simplex plus Algorithm (*NSA+*)

NSA+ is an extension of *NSA*. Compared with the standard version of the blocking scheme presented by Grigoriadis [12] and maintaining the strongly feasible spanning tree [22], *NSA+* has three new features. These features are concerned with the starting point/block for scanning violated arcs, the memory technique and the scanning method. The pricing scheme of *NSA+* is designed based on these features.

There is a function for the pricing scheme to find out an entering arc. The pseudo-code for this function is illustrated in Fig. 4. The arcs in the graph of the *MCF* model are divided into several blocks with the same size and each block is identified by a specific number, known as Block-Number. For each problem, the number of blocks is calculated by dividing the number of arcs in the graph by the block's size.

At first iteration, when the initialization is needed and the packet is empty, the number of blocks is calculated and the first one to be scanned for the optimality condition is chosen (see lines 2–5). The function selects the first block randomly or by a heuristic method (based on location of the biggest cost, for example). Note that at first iteration the lines 6–9 do not perform anything because the packet is empty (these will be activated from the second iteration; when there is at least one violated arc in the packet). Scanning of the arcs for violation among different blocks is chosen circularly. At each scan one violating arc (at most) from each block is put into the packet as long as it has empty place and there is any violated arc (see lines 10–14). The capacity of the packet is more than the block's size and the most violating arcs are kept at the top of the packet. At the end of the function, if the packet is empty, the current solution is optimal (see lines 15–17). Otherwise the packet will be sorted in descending order, based on the absolute value of the reduced costs, and the most violated arc will be chosen as the entering arc (see lines 18–19).

The memory technique will be activated from the second iteration. It uses a few elements at the top of the packet of the previous iteration. The size of this memory may be a percentage of the block's size. The reduced costs of the most violated arcs of the previous iteration are recalculated (see line 6). If they violate the optimality conditions again, they are kept in the packet. Otherwise they must be removed from the packet, which can be replaced by new violating arcs (see lines 7–9). The remaining part of the function acts as before.

As we mentioned, there are two options to choose the first block to be scanned; randomly and heuristically. Hence, *NSA+* has two extensions: (a) *NSA* + *R*: The entering arc function chooses the first block by random selection; (b) *NSA* + *H*: The entering arc function chooses the first block by a heuristic method (based on the location of the largest cost in the graph).

The main parameters involved in the function are as follows:

n: the number of arcs in the graph model.
B: the size of each block in the blocking scheme.
K: the size of packet, i.e. the number of violated arcs in the packet. (In *NSA+*, $K > B$.)
P: the size of the memory, i.e. the number of violated arcs at the top of the packet to be kept for the next iteration. (In *NSA+*, $P = K - B$.)

At each iteration, the elements in the packet are either from the previous iteration or new violated arcs. Clearly, the size of the memory must not be too big since it prevents the function from collecting new violated arcs. To get a better performance in solving large scale problems, we set $K = 225,\ B = 200$ and $P = 25$ by trial and error. The computational evidence shows [23] that both the extensions of *NSA+* are significantly more efficient than *NSA* in CPU-time required to tackle the problems. Moreover, *NSA* + *H* does better than *NSA* + *R*.

Without the *heuristic approach and memory technique*, described above, it is instructive to see how some known pricing schemes can be obtained by appropriate setting of the parameters and variation of the scanning method for violation of the optimality conditions. In these special cases the largest violated arc in the packet is selected as the entering arc, without any effort to sort it.

1. The *first improving candidate scheme* [20] can be obtained if we set $K = 1,\ P = 0$ and the scanning method scans the arcs sequentially without considering the blocks and their sizes ($B = n$).
2. The *Dantzig pricing* [24] or *full pricing* is achieved by the following setting of the parameters: $B = K = n,\ P = 0$. Clearly, here we do not expect that the number of improving arcs will be equal to n, but by this setting we can force the algorithm to scan all the non-basic arcs at each iteration.
3. The *blocking scheme* [12] is obtained by setting $B > 1,\ K \leq B,\ P = 0$ and pricing one block fully per each iteration, taking one block after the other. A block size between 1% and 8.5% of the size of the arcs in the graph has been recommended by Grigoriadis, for large MCF models.
4. If we have $B > 1,\ K \leq B,\ P = 0$, and the scanning method scans the blocks at constant intervals, called the skip factor, throughout the entire arc set then we get the *arc sample scheme* [20].
5. If we have $K > 1,\ P = 0$ and the scanning method scans the arcs sequentially without considering the blocks ($B = n$), then we get the *Mulvey strategy/scheme* [11].
6. If we permit that the scanning method collects more than one violated arc from a specified number of blocks of the graph, then we get the *General Pricing scheme* [16]. That pricing scheme is controlled by three parameters: (a) the number of blocks in the graph; (b) the number of blocks to be scanned for violation of the optimality conditions; and (c) the number of violated arcs to be collected from the blocks.

More details on comparison between these pricing schemes were performed by Kelly and Neill [20]. They implemented several pricing schemes and ran their software for different classes of Minimum Cost Flow problems. In their results, the block pricing scheme provided a better performance compared with others. Ahuja et al. [25] studied the minimum cost simple equal flow problem and presented a parametric simplex method on the problem. Andrew [26] studied practical implementation of minimum cost flow algorithms and presented an efficient implementation that worked very well over a wide range of problems.

3.3. Greedy Vehicle Search method

NSA+ is a complete algorithm. Although it is efficient, it can only work on problems with certain limits in size. To complement *NSA+*, a Greedy Vehicle Search (*GVS*) is designed and implemented. *GVS* will be useful for problems whose sizes go beyond the limit of *NSA+* or when the time available for computation is too short (as it would be in dynamic scheduling). This simple search method behaves as a *Taxi Service System*. For any unassigned job and the list of idles *AGVs*, a job is assigned to a vehicle with minimum cost, including waiting and travelling times of the vehicles as well as lateness of the jobs. The pseudo-code of *GVS* is demonstrated in Fig. 5.

4. Experimental results

In this section, the experimental results from the implementation and running the algorithms to tackle the static and dynamic problems are presented. In the literature, there are several algorithms to tackle the *MCF* model [9]. The reason for choosing *NSA* is that it is the fastest algorithm. As we mentioned, the performance of the algorithm is affected by choosing a proper pricing scheme. Among the schemes in the literature, the blocking scheme was the fastest [20]. We implemented the blocking scheme with a block size of 5% of the size of the arcs of the *MCF* model. Then we compared the implementation of the blocking scheme in *NSA* and the new scheme in *NSA+* for the static automated vehicle scheduling problem [23]. The results show that *NSA+* is 25% faster and significantly more efficient than *NSA*.

```
If there is any remaining job and there is any idle vehicle
   Calculate the objective function (v, j): the travelling times and waiting times of the AGVs as well as the lateness
         time of jobs.
   For v=1, 2, ...,#Idle vehicles; for j=1, 2,.., #Remaining jobs
Else
   Stop
End If
Again: Select a vehicle m and a job j with the minimum cost.
         Assign the vehicle m to the job j.
         Remove the vehicle m from the list of idle vehicles and the job j from the remaining job.
         If there is any idle vehicle and any left jobs
            Go to Again
         End If
Stop
```

Fig. 5. Pseudo-code of simple heuristic search.

Table 1
Value of parameters in the simulation.

Description of the parameters	Values
Number of vehicles in the port	50
Number of quay cranes	7
Number of blocks in the yard (storage area inside the port)	32
Time window of the cranes	120 s
Travelling time between any two points in the port (see Assumption 1)	Random between 1 and 100 s
Weight of waiting times for the *AGVs* in the cost of the objective function	1
Weight of travelling times for the *AGVs* in the cost of the objective function	5
P as the penalty in the costs of the objective function	10,000

4.1. Static problems

To test the model and make a comparison between *NSA+* and *GVS*, a hypothetical port was designed. The parameters in Table 1 were used to define the port, the objective function, the number of vehicles and generate the jobs. We implemented our software (*DSAGV*) in Borland C++. Then, *DSAGV* has been run to solve several random problems. The sources and destinations of jobs were chosen randomly.

The first aspect of the comparison is *CPU-time* to solve the problems by the two algorithms, which has been drawn in Figs. 6 and 7, according to the number of jobs. Also the power estimation for those two curves has been shown in the figures. All experiments were run on a Pentium-4 2.4 GHz PC with 1 GMB RAM. As can be seen in the figure, *NSA+* can find the global optimal solution for 3000 jobs and 10 millions arcs in the *MCF–AGV* model within 2 min. *GVS* is fast and could find a local optimum for 8000 jobs within 35 s. Another aspect of the comparison is the solution quality. We ran the software for the same static problems. The value of the objective function by *GVS* and *NSA* is shown in Fig. 8. As is expected, the quality of the solutions by *NSA+* is substantially significant compared with *GVS*.

Given N jobs and *MAGVs* in the problem ($N \gg M$), the complexities of the two algorithms are calculated as follows:

- *NSA+*: Assume the maximum flow, MF, in each of the m arcs, at maximum cost, C, for the *MCF* model. So there is an upper bound on the value of the objective function. This upper bound is given by $m \times C \times MF$. There are two different types of pivots in the algorithm, non-degenerate and degenerate pivots. The former is bounded by $m \times C$ because the number of non-degenerate pivots in the algorithm is bounded by $m \times C \times MF$ ($MF = 1$ in the *MCF–AGV* model). The number of degenerate pivots is determined by the sum of nodes potential and maintaining the strongly feasible spanning tree. Given n as the number of nodes in the graph model, the sum of nodes potential is bounded by $n^2 \times C$. It is decreased at each iteration when the spanning tree is strongly feasible [27]. A series of degenerate pivots may occur between each pair of non-degenerate pivots, and thus a bound on the total number of iterations is $m \times n^2 \times C^2$. Finding the entering arc is the $O(m)$ operation and sorting the packet is the $O(K \times \mathrm{Log}K)$ operation (K is size of the packet, $K = 225$). Finding the cycle, amount of flow change, leaving arc and updating the tree are $O(n)$ operations. Hence the complexity of each pivot is $O((m+n)K \times \mathrm{Log}K)$. Based on the complexity of the number of iterations and the complexity of each pivot, the total complexity of this algorithm is determined by the following equation:

$$O((m+n)mn^2C^2K\mathrm{Log}K).$$

  Since $m = O(N^2)$; $n = O(N)$, the total complexity of *NSA+* to tackle the *MCF–AGV* model is $O(N^6)$.
- *GVS*: For static problems, we assume that every job has to be served by the vehicles. The algorithm operates as follows: In the first run, it finds out one job with minimum cost (among N jobs) for a vehicle (among M *AGVs*). In the second run,

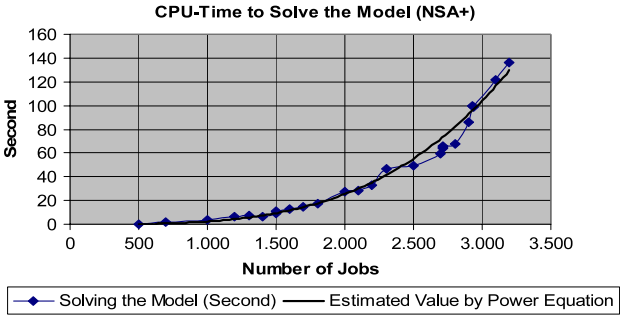

Fig. 6. CPU-time to solve the static problem by *NSA+*, based on the number of jobs.

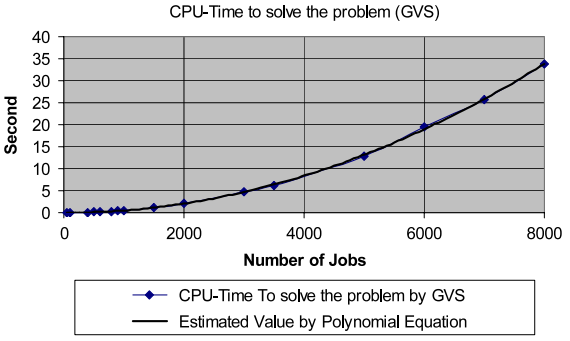

Fig. 7. CPU-time to solve the static problem by *GVS*, based on the number of jobs.

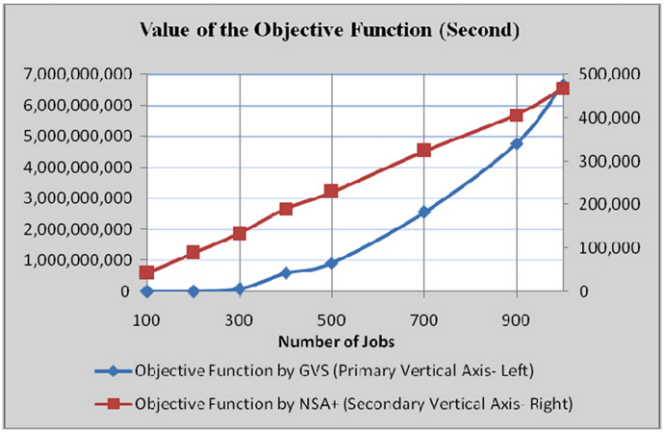

Fig. 8. The value of objective function in static problem by *GVS* and *NSA+*, based on the number of jobs.

another job among $N-1$ jobs is assigned to a vehicle, which could be the selected vehicle in the first run or others. This process is continued until there are no remaining jobs. Hence, the number of runs of *GVS* is calculated by the following equation:

$$M \times N + M \times (N-1) + M \times (N-2) + \cdots + M \times 1 = \frac{M \times N \times (N+1)}{2}.$$

Therefore, the complexity of *GVS* is $O(M \times N^2)$. It is significantly less than the complexity of *NSA+*.

For the dynamic problems, we assume that only one job has to be assigned to each vehicle. When a job arrives or an *AGV* served the current job, the algorithm is then run and finds out one job (among the remaining jobs) for a vehicle (among the idle *AGVs*) with minimum cost. The complexity of the algorithm for the dynamic problems is hence $O(N \times M)$.

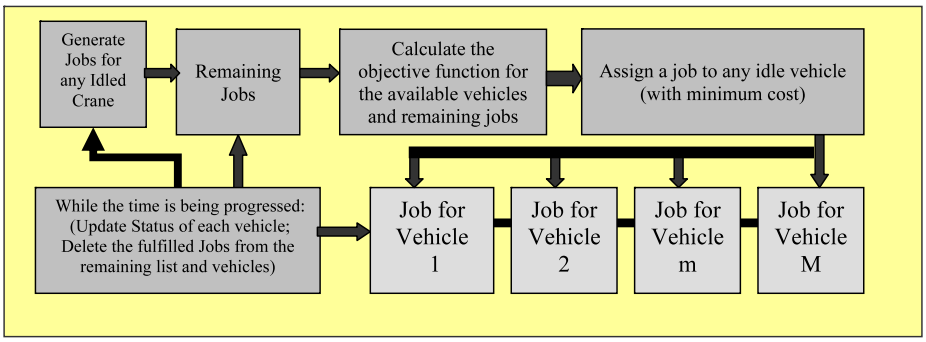

Fig. 9. Block diagram Greedy Vehicle Search in dynamic aspect.

4.2. Dynamic problems

A major part of the literature treats the vehicle scheduling problems in the container terminal as static problems, where all the jobs and travelling time are known beforehand. In reality, the problem is dynamic. New jobs arrive from time to time. Delays or breakdown of vehicles change the travelling time and availability of vehicles.

The main architecture of *GVS* is shown in Fig. 9. A similar architecture and operations for *NSA*+ are considered in our software. We shall use Fig. 9 to describe the architecture and operations briefly. At the start of process, the Job Generator generates a few jobs for the cranes. These jobs will be appended to the remaining jobs, which is empty at the beginning. The remaining jobs are used by the *GVS* and the output of this method is an individual job for every vehicle. When the time is running, the travelled and waited times of every vehicle should be updated. At the same time, if the vehicle picks up the job from the quay side, the assigned job for the vehicle will be deleted and removed from the list of remaining jobs. If the job should be delivered to the crane on the quay side, it could not be removed until meeting time between the crane and the vehicle (the appointment place is on the quay side, not the yard side). The Job Generator has to generate a few new jobs, when it finds out any idle crane.

To evaluate the relative strength and weakness of *GVS* and *NSA*+ in the dynamic scheduling problem, we used randomly generated problems. Distance between every two points in the port as well as the source and destination of jobs were chosen randomly. We did a simulation for six hours subject to generating 5 jobs for any idle crane. Other parameters for this simulation were the same as Table 1. Figs. 10–12 demonstrate some outputs of the software during the six hours. As we can see from Fig. 10, waiting times of the vehicles in *GVS* is significantly greater than waiting times of the vehicle in *NSA*+, although travelling times of the vehicles for both algorithms are almost the same during the six hours. The main reason for this result is that, being a complete algorithm, *NSA*+ finds the optimal solution for the MCF model whereas GVS is an incomplete search method. The number of jobs carried out by the end of six-hour (21,600 s) for both algorithms is approximately the same (see Fig. 11). Generally, due to the tight schedules of the quay cranes, it is undesirable for containers to be served early or too late for the appointment. Therefore, the average lateness from the appointment times is another indicator for evaluating the algorithms. Given the number of served jobs, N, the time at which the job i is served, ACT_i, and the appointment time of job i, APT_i, the schedule's average lateness is calculated by the following equation:

$$\text{Average Lateness} = \frac{\sum_{i=1}^{N} (ACT_i - APT_i)^+}{N}.$$

Fig. 12 presents the Average Lateness indicator for both *NSA*+ and *GVS* during the six-hour simulation. The figure shows that both algorithms performed well, but *GVS* is superior to *NSA*+ in terms of the Average Lateness. As a greedy algorithm, *GVS* achieved this by sacrificing the waiting and travelling times of the *AGVs*. Consequently, the objective value of *NSA*+ is significantly better than that of *GVS*, as seen clearly in Fig. 8. This is due to the fact that GVS finds a local optimum for each problem whereas *NSA*+ finds the global optimum solution.

4.3. A discussion over GVS and meta-heuristic

A discussion can arise in using *GVS* compared with some well-known meta-heuristics (stochastic search methods) such as Genetic Algorithms, Tabu Search, Simulated Annealing method and others when the problem is too big or when the time available to tackle the problem is too short. In the literature, we reviewed these solution methods [23], including general considerations and major specific considerations in them. In this section, we have a short discussion on the matter.

According to the literature, *GVS* could be considered as a heuristic. Voß (2000) [28] defines heuristic as follows: "A heuristic is a technique (consisting of a rule or a set of rules) which seeks (and hopefully finds) good solutions at a reasonable

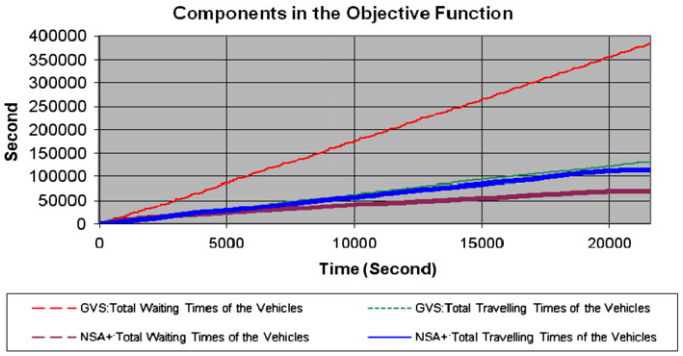

Fig. 10. A comparison of travelling and waiting times in dynamic problems.

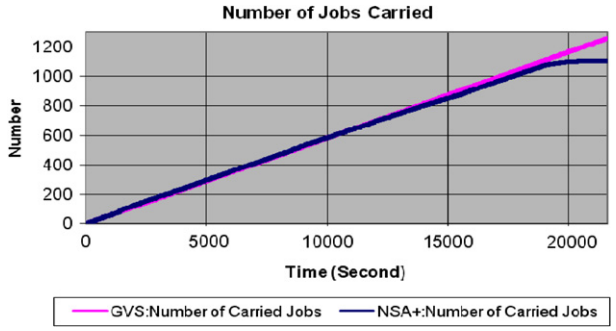

Fig. 11. The number of jobs carried by the algorithms.

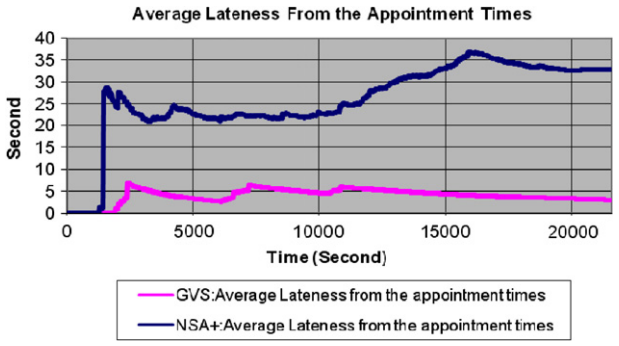

Fig. 12. Average lateness from the appointment time.

computational cost. A heuristic is approximate in the sense that it provides (hopefully) a good solution for relatively little effort, but it does not guarantee optimality". We based *GVS* on two simple rules, the idle vehicles and jobs remained. Moreover, *GVS* does not get too much *CPU-time* to tackle the problem and for that reason it finds out a local optimum solution.

We had the problem with memory to put the *MCF–AGV* model into. One of the reasons to use *GVS* is that it has no memory technique. In the literature, we studied that "the meta-heuristics manipulate a complete (or incomplete) single solution or a collection of solutions at each iteration" [28]. In order to do that, they require memory, whereas the memory usage of *GVS* is teeny, compared with the meta-heuristics.

Our work shows that *GVS* solves a huge problem in a short time [23]. The dynamic problem of 50 *AGVs* could be solved in a second. Additionally, *GVS* is effective in the average lateness to serve the jobs. The weakness of the meta-heuristics is that effectiveness could be sensitive to choice of parameter values and operators [29]. Basically, finding out a set of suitable parameters for the meta-heuristics and their training to tackle the problem will be beyond of the scope of this paper.

5. Concluding summary

In this paper, the automated guided vehicle scheduling problem in container terminals was formulated as a special case of the minimum cost flow problem. Then, two novel algorithms, namely *NSA+* and *GVS*, for tackling the problem were presented. *NSA+* is a complete algorithm and the state-of-the-art algorithm to tackle the *MCF* model. Our experimental results suggested that it could find the global optimal solution for 3000 jobs and 10 millions arcs in the graph model within 2 min by running on a 2.4 GHz Pentium PC. *GVS* is an incomplete algorithm. It is useful when the problem is too big for *NSA+* to solve, or when time available for computation is too short, as could be the case in dynamic scheduling. The two algorithms were compared on both static and dynamic problems. Our experimental results suggested that *NSA+* is efficient and effective in minimizing both the waiting and travelling times of the vehicles, whereas *GVS* is more effective in minimizing the average lateness. The two algorithms complement each other and can be used according to the situation and the user's needs. Based on our experiment, *NSA+* and *GVS* together are practical algorithms for automatic vehicle scheduling.

References

[1] P. Baptiste, C. Le Pape, W. Nuijten, Constraint-Based Scheduling, Applying Constraint Programming to Scheduling Problems, in: Kluwer's International Series, 2001.
[2] P. Baptiste, C. Le Pape, W. Nuijten, Incorporating efficient operations research algorithms in constraint-based scheduling, in: First International Workshop on Artificial Intelligence and Operations Research, Timberline Lodge, Oregon, 1995.
[3] J. Böse, T. Reiners, D. Steenken, S. Voß, Vehicle dispatching at seaport container terminals using evolutionary algorithms, in: Proceedings of the 33rd Annual Hawaii International Conference on System Sciences, IEEE, 2000, pp. 1–10.
[4] Y. Cheng, H. Sen, K. Natarajan, C. Teo, K. Tan, Dispatching automated guided vehicles in a container terminal, Technical Report, National University of Singapore, 2003.
[5] Y. Huang, W.J. Hsu, Two equivalent integer programming models for dispatching vehicles at a container terminal, CAIS, Technical Report, School of Computer Engineering, Nanyang Technological University, Singapore 639798, 2002.
[6] J.M. Patrick, R. Dekker, Operations research supports container handling, Technical Report EI 2001-22, Erasmus University of Rotterdam, Econometric Institute, 2003.
[7] H. Sen, Dynamic AGV-container job deployment, Technical Report, HPCES Programme, Singapore-MIT Alliance, 2001.
[8] K.G. Murty, L. Jiyin, W. Yat-Wah, C. Zhang, Maria C.L. Tsang, Richard J. Linn, DSS (decision support system) for operations in a container terminal, Decision Support System 39 (2002) 309–332.
[9] R.K. Ahuja, T.L. Magnanti, J.B. Orlin, Network Flows: Theory, Algorithms and Applications, Prentice Hall, 1993.
[10] A.V. Goldberg, R. Kennedy, An efficient cost scaling algorithm for the assignment problem, Technical Report, Stanford University, 1993.
[11] J.M. Mulvey, Pivot strategies for primal simplex network codes, Journal of the Association for Computing Machinery 25 (1978) 266–270.
[12] M.D. Grigoriadis, An efficient implementation of the network simplex method, Mathematical Programming Study 26 (1983) 83–111.
[13] G. Bradley, G. Brown, G. Graves, Design and implementation of large scale primal transhipment algorithms, Management Science 24 (1977) 1–38.
[14] D. Eppstein, Clustering for faster network simplex pivots, in: Proc. 5th ACM–SIAM Symposium, Discrete Algorithms, 1999, pp. 160–166.
[15] A. Löbel, A network simplex implementation, Technical Report, Konrad-Zuse-Zentrum für Informationstechnik Berlin, ZIB, 2000.
[16] I. Maros, A general pricing scheme for the simplex method, Technical Report, Department of Computing, Imperial College, London, 2003.
[17] M. Masakazu, On network simplex method using primal–dual symmetric pivoting rule, Journal of Operations Research of Japan 43 (1999) 149–161.
[18] J.W. Bae, K.H. Kim, A pooled dispatching strategy for automated guided vehicles in port container terminals, International Journal of Management Science 6 (2) (2000) 47–60.
[19] L. Qiu, W.-J. Hsu, S.-Y. Huang, H. Wang, Scheduling and routing algorithms for AGVs: a survey, International Journal of Production Research 40 (3) (2002) 745–760. Taylor & Francis Ltd.
[20] D.J. Kelly, G.M. ONeill, The minimum cost flow problem and the network simplex solution method, Master Degree Dissertation, University College, Dublin, 1993.
[21] R. Helgason, J. Kennington, Primal simplex algorithms for minimum cost network flows, in: Handbook on Operations Research and Management Science, vol. 7, North-Holland, Amsterdam, 1995, pp. 85–133.
[22] W.H. Cunningham, Theoretical properties of the network simplex method, Mathematics of Operations Research 4 (2) (1979) 196–208.
[23] H. Rashidi, Dynamic scheduling of automated guided vehicles in container terminals, Ph.D. Thesis, Department of Computer Science, University of Essex, 2006.
[24] G.B. Dantzig, Linear Programming and Extensions, Princeton University Press, Princeton, New Jersey, 1963.
[25] R.K. Ahuja, J.B. Orlin, M.S. Giovanni, P. Zuddas, Algorithms for the simple equal flow problem, Management Science 45 (10) (1999) 1440–1455.
[26] V.G. Andrew, An efficient implementation of a scaling minimum-cost flow algorithm, Journal of Algorithms 22 (1997) 1–29.
[27] R.K. Ahuja, J.B. Orlin, P. Sharma, P.T. Sokkalingam, A network simplex algorithm with $O(N)$ consecutive degenerate pivots, Operations Research 30 (3) (2002) 141–148.
[28] S. Voß, Meta-heuristic: the state of the art, in: Local Search for Planning and Scheduling: ECAI 2000 Workshop, Berlin, Germany, 2000.
[29] E.P.K. Tsang, Scheduling techniques—a comparative study, British Telecom Technology Journal 13 (1) (1995) 16–28. Martlesham Heath, Ipswich, UK.