OR Spektrum (1998) 20: 229–235

OR Spektrum

A two-machine permutation flow shop scheduling problem with buffers

Czesław Smutnicki

Technical University of Wrocław, Institute of Engineering Cybernetics, Janiszewskiego 11/17, 50-372 Wrocław, Poland
(e-mail: smutnick@ict.pwr.wroc.pl)

Received: 21 July 1997 / Accepted: 6 October 1998

Abstract. Problems with "blocking" (limited intermediate storage space) are used frequently for modelling and scheduling just-in-time and flexible manufacturing systems. In this paper, an approximation algorithm is presented for the problem of finding the minimum makespan in a two-machine permutation flow-shop scheduling problem with the mediating buffer of finite capacity. The algorithm is based on the tabu search approach supported by the reduced neighborhood, search accelerator and technique of back jumps on the search trajectory. Due to some special properties, the proposed algorithm provides makespans very close to optimal in a short time. It has been shown that this algorithm outperforms all known approximation algorithms for the problem stated.

Zusammenfassung. In dieser Arbeit wird eine Heuristik zur Ermittlung der minimalen Gesamtdurchlaufzeit in einem Zwei-Maschinen-Flow-Shop-Problem mit begrenzter Zwischenlagerkapazität vorgestellt. Solche Maschinenbelegungsplanungsprobleme werden häufig zur Modellierung von Just-in-Time und Flexiblen Fertigungssytemen eingesetzt. Der Algorithmus basiert auf dem Tabu-Search-Verfahren mit eingeschränkter Nachbarschaft, beschleunigter Suche und Rücksprungtechnik auf der Suchtrajektorie. Aufgrund einiger spezieller Eigenschaften liefert das vorgestellte Verfahren in kurzer Zeit Lösungsqualitäten nahe dem Optimum.

Key words: Scheduling – Heuristics – Two-machine flow-shop – Buffer – Tabu search

Schlüsselwörter: Maschinenbelegungsplanung – Heuristiken – Zwei-Maschinen-Flow-Shop – Zwischenlager – Tabu-Search

1 Introduction

The flow manufacturing line has been considered as a basic device for modern automated production systems. Its work can be modelled, to our great advantage, by a problem known in the scheduling theory as *flow-shop*. Since this problem belongs to the class of very hard (strongly NP-hard) combinatorial optimisation problems, many different solution algorithms have been proposed and examined. Quite often the classic flow-shop model has been extended by introducing additional constraints derived from industrial needs. The recent survey (Hall and Sriskandarayah, 1996) pinpoints the significance of "no wait" and "blocking" constraints (problems with limited intermediate storage) in modern just-in-time systems, flexible manufacturing systems, etc.

While considerable progress has been made for algorithms for the classic flow-shop problem [see e.g. the paper by Nowicki and Smutnicki (1996a)], the flow-shop problem with limited storage space still remains far from being completely examined. In this paper, we deal with the problem in which a given a priori set of jobs need to be processed on two successive machines with the buffer of limited capacity located between machines. It has been shown that this problem with the buffer of capacity zero is equivalent to the flow-shop with "no wait" constraints. Therefore both "zero-buffer" and "no-wait" two-machine flow-shop problems can be solved in a polynomial time $O(n \log n)$ by the algorithm of Gilmore and Gomory [see the paper by Hall and Sriskandarayah (1996) for brief description of this algorithm]. The case with the unlimited capacity buffer is equivalent to the well-known classic two-machine flow-shop, and thus can be solved by Johnson's algorithm in time $O(n \log n)$. The latter method also provides the lower bound for the problem stated. The problem with the buffer size greater than zero is already NP-hard (Papadimitriu and Kanellakis, 1980).

Several constructive algorithms have been proposed for the two-machine flow-shop problem with buffers (see the survey by Leisten, 1990). Most of these algorithms are based on extensions of algorithms known for the flow-shop problem with zero or infinite capacity buffer. In this paper, we propose an improvement algorithm based on the tabu search technique with the use of some particular properties of the problem. Computational tests up to 200 jobs show that the proposed algorithm provides makespans close to optimal in a short time.

2 The problem

There is a set of jobs $J = \{1, 2, \ldots, n\}$ which has to be processed in the system consisting of two machines and a buffer. Each job has the same route: first, it is processed on machine 1 in time a_j, next it goes to the buffer, and at the end it is processed on machine 2 in time b_j. Several additional constraints exist: (a) each machine can execute at most one job at a time, (b) each job can be processed on, at most, one machine at a time, (c) processing of a job on a machine cannot be interrupted, (d) both machines process jobs in the same order, (e) the buffer has FIFO service rule and capacity $z \geq 0$, i.e. can hold at most z jobs at a time [1], (f) a job completed on machine 1 can be sent to the buffer if it is not completely filled (at most $z-1$ jobs are stored at this moment of time), otherwise this job must remain on this machine until the free place appears in the buffer [2]. A feasible schedule (A, B), $A = (A_1, \ldots, A_n)$, $B = (B_1, \ldots, B_n)$ is defined by the completion times A_j, B_j of the job j on machines 1 and 2, respectively, such that the above constraints are satisfied. The problem is to find a feasible schedule that minimises the makespan $\max_{j \in J} B_j$.

Now we provide a formal mathematical model of the problem. For this purpose, we introduce the notion of *job processing order*. The job processing order can be represented by a permutation $\pi = (\pi(1), \ldots, \pi(n))$ on the set J; $\pi(j)$ denotes the element of J which is in position j in π. Let Π denote the set of all permutations on a set J. For a given $\pi \in \Pi$, a feasible schedule (A, B) respecting the job order π, which minimise $B_{\pi(n)}$ can be found using the following recursive formulae

$$A_{\pi(j)} = \max(A_{\pi(j-1)},\ B_{\pi(j-z-2)}) + a_{\pi(j)} \qquad (1)$$

$$B_{\pi(j)} = \max(B_{\pi(j-1)},\ A_{\pi(j)}) + b_{\pi(j)} \qquad (2)$$

calculated for $j = 1, 2, \ldots, n$, where $\pi(j) = 0$ if $j \leq 0$, $A_0 = 0 = B_0$. Now we can rephrase our problem as that of finding the job processing order $\pi \in \Pi$ which minimises the makespan $C_{\max}(\pi) = B_{\pi(n)}$, where $B_{\pi(n)}$ follows from (1)–(2). Let C_j denote the time moment when machine 1 becomes idle after the completion of job j. It is easy to verify that

$$C_{\pi(j)} = \max(A_{\pi(j)},\ B_{\pi(j-z)} - b_{\pi(j-z)}),\ j \in J. \qquad (3)$$

A feasible schedule, job starting and completion times as well as times of releasing the machine for an instance and the job processing order $\pi = (1, 2, \ldots, n)$ is shown in Fig. 1.

In the analysis, it is convenient to refer to an auxiliary graph model associated with a fixed job processing order. For the processing order π, $\pi \in \Pi$, we create the graph $G(\pi) = (K \cup L, V \cup H \cup S)$ (see Fig. 2) with the set of nodes $K \cup L$, where $K = \{1, \ldots, n\}$, $L = \{n+1, \ldots, 2n\}$, and sets of "vertical" arcs $V = \bigcup_{j=1}^{n-1}\{(j, n+j)\}$, "horizontal"

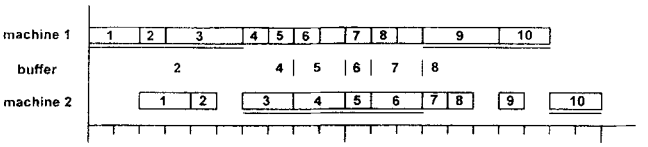

Fig. 1. Gantt Chart of the two-machine problem with a buffer of unit capacity

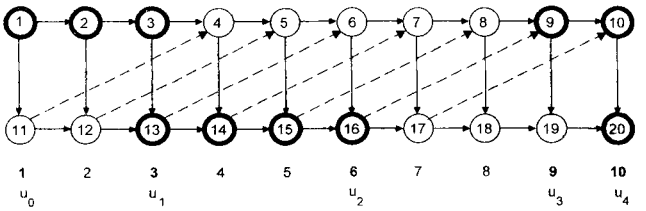

Fig. 2. The graph $G(\pi)$ for $n = 10$, $\pi = (1, 2, \ldots, 10)$ and $z = 1$; nodes on the ciritical path are drawn in bold

arcs $H = \bigcup_{j=2}^{n}\{(j-1, j), (n+j-1, n+j)\}$, and "skew" arcs $S = \bigcup_{j=z+3}^{n}\{(n+j-z-2, j)\}$. Each node $j \in K$ represents the processing of job $\pi(j)$ on machine 1, event $A_{\pi(j)}$, and has weight $a_{\pi(j)}$, whereas node $n+j \in L$ represents the processing of job $\pi(j)$ on machine 2, event $B_{\pi(j)}$ and has weight $b_{\pi(j)}$. Each arc has zero weight. It is clear that job completion time $A_{\pi(j)}$ found by (1) equals the length of the longest path to the node $j \in K$ (including $a_{\pi(j)}$) in $G(\pi)$, and $B_{\pi(j)}$ found by (2) equals the length of such a path to node $n+j \in L$. The makespan $C_{max}(\pi)$ equals the length of the longest path (critical path) in $G(\pi)$.

Consider a critical path p in $G(\pi)$ interpreted as a sequence of graph nodes (see Fig. 2). Obviously, p has to start from node $1 \in K$ and end in $2n \in L$. This path contains subsequences of nodes from K and L in an alternate order. Denote the number of these subsequences by $2w$, $w \geq 1$. Then p can be written in the form of $p = (p_1^K, p_1^L, \ldots, p_w^K, p_w^L)$, where p_l^K is a sequence of nodes from K, and p_l^L is a sequence of nodes from L, $l = 1, \ldots, w$. Let $p_l^K = (e_l, e_l+1, \ldots, f_l)$, $1 \leq e_l \leq f_l \leq n$. The path from the last node of p_l^K (i.e. the node f_l) to the first node of p_l^L goes through the vertical arc $(f_l, n+f_l) \in V$. The path from the last node of p_l^L to the first node of p_{l+1}^K (i.e. the node e_{l+1}) goes through the skew arc $(n+g_l, e_{l+1}) \in S$, where $g_l = e_{l+1} - z - 2$ for $l = 1, \ldots, w-1$. Then $p_l^L = (n+f_l, n+f_l+1, \ldots, n+g_l)$, $1 \leq f_l \leq g_l \leq n$. Therefore, it has to be $1 = e_1 \leq f_1 \leq g_1 < e_2 \leq f_2 \leq g_2 < \ldots < e_w \leq f_w \leq g_w = n$. Using the above denotations we can write the formula on $C_{max}(\pi)$ as follows

$$C_{max}(\pi) = \sum_{l=1}^{w}\Big(\sum_{j=e_l}^{f_l} a_{\pi(j)} + \sum_{j=f_l}^{g_l} b_{\pi(j)}\Big). \qquad (4)$$

In Fig. 2, the critical path $p = (1, 2, 3, 13, 14, 15, 16, 9, 10, 20)$ can be decomposed into $2w = 4$ subsequences $p_1^K = (1, 2, 3)$, $p_1^L = (13, 14, 15, 16)$, $p_2^K = (9, 10)$, $p_2^L = (20)$. Then, $e_1 = 1$, $f_1 = 3$, $g_1 = 6$, $e_2 = 9$, $f_2 = 10$, $g_2 = 10$.

Note that the sequence of nodes p_l^K corresponds to a sequence of jobs $\pi(e_l), \pi(e_l+1), \ldots, \pi(f_l)$ performed consecutively on machine 1, whereas the sequence p_l^L corresponds to a sequence of jobs $\pi(f_l), \pi(f_l+1), \ldots, \pi(g_l)$ performed consecutively on machine 2. If $e_l < f_l$, we will

[1] Constraints (d) and (e) are redundant since (d) implies the first part of (e) and vice versa. These constraints may follow either from industrial needs (the physical buffer construction) or theory (arbitral assumptions made in order to simplify the solution method). We use constraints in their real form since relations betweeen non/permutation schedules and different service rules have not been presented here.

[2] If $z = 0$, the job must remain on machine 1 until machine 2 becomes available.

call the sequence of jobs $\pi(e_l), \pi(e_l+1), \ldots, \pi(f_l)$ the *block of type* 1; if $f_l < g_l$, we will call the sequence of jobs $\pi(f_l), \pi(f_l+1), \ldots, \pi(g_l)$ the *block of type* 2. Blocks of type 1 and 2, which we also call blocks on machines 1 and 2 (respectively), correspond to the known already notion "block of jobs" (see e.g. Grabowski et al., 1983, 1986). Next, we introduce the new notion – *block of type* 0. The block of type 0 is the sequence of jobs $\pi(g_l), \pi(g_l+1), \ldots, \pi(e_{l+1})$ associated with the pair of nodes $(n+g_l, e_{l+1})$ linked to each other by a skew arc, $l = 1, \ldots, w-1$. The philosophy of skew block follows from the paper by Smutnicki (1986). Let k be the number of blocks in the critical path p. We index all of them according to their occurrence in p. To simplify notations, we define a new sequence $u = (u_0, \ldots, u_k)$, such that $u_0 < u_1 < \ldots < u_k$ and $\{u_0, u_1, \ldots, u_k\} = \{e_1, f_1, g_1, \ldots, e_w, f_w, g_w\}$. By definition, we have $u_0 = 1$ and $u_k = n$. Thus, the l-th block is represented by the sequence $\mathscr{B}_l = (\pi(u_{l-1}), \pi(u_{l-1}+1), \ldots, \pi(u_l))$ and let t_l denote the type of this block, $l = 1, 2, \ldots, k$. Note that the last job in $\mathscr{B}_l$ is simultaneously the first in $\mathscr{B}_{l+1}$, $l = 1, 2, \ldots, k-1$. The number of jobs in $\mathscr{B}_l$ is equal to $u_l - u_{l-1} + 1$, and by the definition of u we have $|\mathscr{B}_l| \geq 2$, $\sum_{l=1}^{k} |\mathscr{B}_l| = n+k-1$. Using the block notion we can re-write the formula on $C_{max}(\pi)$

$$C_{max}(\pi) = \sum_{l\in\{1,\ldots,k\}:t_l=1} \sum_{j\in\mathscr{B}_l} a_j + \sum_{l\in\{1,\ldots,k\}:t_l=2} \sum_{j\in\mathscr{B}_l} b_j. \quad (5)$$

In Fig. 2, we have four blocks: $\mathscr{B}_1 = (1,2,3)$ of type 1, $\mathscr{B}_2 = (3,4,5,6)$ of type 2, $\mathscr{B}_3 = (6,7,8,9)$ of type 0, and $\mathscr{B}_4 = (9,10)$ of type 1. Then, $u = (1,3,6,9,10)$, $k = 4$.

3 Algorithm

We apply the tabu search technique (*TS*) to solve the problem stated. Currently, *TS* is the best one from among local search methods designed for finding a near optimal solution of many combinatorial optimisation problems (Glover, 1989,1990). The basic version of this approach refers to some elements called the *move, neighbourhood, initial solution, searching strategy, memory, aspiration criteria, stopping rules*. Although these elements have many different forms (that depend on the problem solved), we outline them only in the form suitable for our needs. The *move* is a small deterministic perturbation of the given permutation allowing us to obtain another, usually very close permutation. The whole subset of moves, which can be performed from a given permutation, generates the collection of descendant permutations called the *neighbourhood*. In our case, *TS* starts from an initial permutation. At each iteration the neighbourhood of the current permutation is searched in order to find the neighbour with the lowest makespan. Next, the move that leads to this neighbour is performed and the resulting permutation becomes the new current – to initiate the next iteration. To avoid cycling and becoming trapped in a local optimum as well as to add robustness to the search, *TS* uses a recency-based short-term memory of the search history called the *tabu list*. This list recorded, for a chosen span of time, either *forbidden* moves or their *attributes*, to prevent future moves that would "undo" the effects of previous moves. A forbidden move aspires to be performed if it leads to the makespan better than the best already known one. [More effective *TS* methods also include some advanced elements i.e. *long-term memory* functions, in order to achieve certain goals of the search *intensification* and *diversification* (Glover and Laguna, 1993)]. The stopping rule traditionally consists of setting a limit on the execution time, number of iterations, number of iterations without improving the makespan and/or criterion performance.

3.1 Moves and the neighbourhood

Among many types of moves considered in the literature (for problems where the solution is represented by a permutation) we have selected *insertion moves* as the most promising for our aim. Briefly, the insertion move operates on a permutation and removes a job placed at a position in this permutation and puts it in another position. Precisely, let $v = (x, y)$ be a pair of positions in the permutation π, $x, y \in \{1, \ldots, n\}$, $x \neq y$. With respect to the permutation π, the pair $v = (x, y)$ defines the move that consists in removing job $\pi(x)$ from position x and inserting it in position y. Thus move v generates a new permutation π_v from π in the following manner

$$\pi_v = (\pi(1), \ldots, \pi(x-1), \pi(x+1), \ldots, \pi(y), \pi(x), \pi(y+1), \ldots, \pi(n)) \quad \text{if } x < y,$$

$$\pi_v = (\pi(1), \ldots, \pi(y-1), \pi(x), \pi(y), \ldots, \pi(x-1), \pi(x+1), \ldots, \pi(n)) \quad \text{if } x > y.$$

All permutations π_v, which can be obtained by applying moves v from a given move set U, create the *neighbourhood* $\mathscr{N}(U, \pi) = \{\pi_v : v \in U\}$ of this permutation. Obviously, the proper selection of U is the first step on the road to success. The set U should be neither too big nor too small (compare the fairly detailed discussion of this subject in the paper by Nowicki and Smutnicki 1996). The "biggest" insertion neighbourhood is generated by the move set $D = \{(x, y) : y \notin \{x-1, x\},\ x, y \in \{1, 2, \ldots n\}\}$ of the cardinality $(n-1)^2$ (see e.g. Taillard, 1990).

We propose a small promising move set created on the basis of blocks of jobs and some *elimination properties* formulated below. Let us consider, for block $\mathscr{B}_l$ of type 0, the set of moves $W_l(\pi)$ which can be performed inside this block, i.e. in the positions $u_{l-1}+1, \ldots, u_l-1$. For the block of type 1 – consider the set of moves $W_l(\pi)$ which can be performed inside this block as well as on the first position u_{l-1}. By symmetry, for the block of type 2 – consider the set of moves $W_l(\pi)$ which can be performed inside this block as well as the last position u_l. Precisely, each set $W_l(\pi)$ is defined by the formulae

$$W_l(\pi) = \begin{cases} \{(x,y) : x, y \in \{u_{l-1}+1, \ldots, u_l-1\}\} & \text{if } t_l = 0, \\ \{(x,y) : x, y \in \{u_{l-1}, \ldots, u_l-1\}\} & \text{if } t_l = 1, \\ \{(x,y) : x, y \in \{u_{l-1}+1, \ldots, u_l\}\} & \text{if } t_l = 2, \end{cases} \quad (6)$$

for $l = 1, \ldots, k$. All these moves create the set $W(\pi) = \bigcup_{l=1}^{k} W_l(\pi)$. The following property provides the basis for elimination.

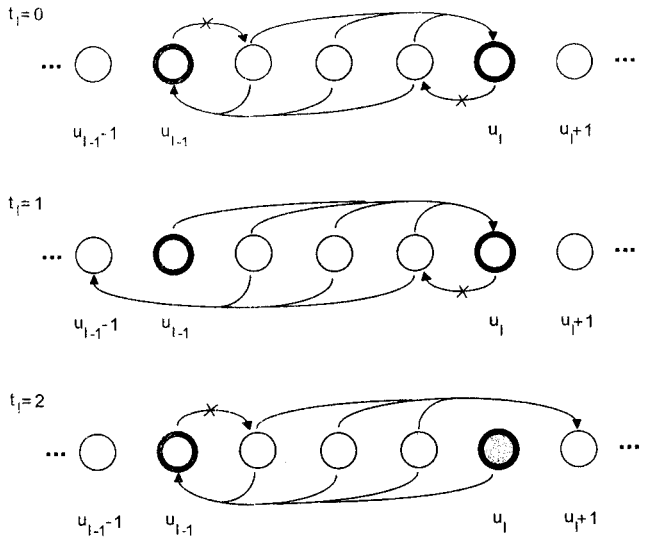

Fig. 3. Moves to the left and right for various types of blocks

Property. *For any move* $v \in W(\pi)$ *we have* $C_{\max}(\pi_v) \geq C_{\max}(\pi)$.

The proof of the Property follows from the following sequence of observations: (1) all graphs have the same structure, but various weigths of nodes; (2) the critical path in $G(\pi)$ is also a path (not necessarily critical) in $G(\pi_v)$; (3) this path can be decomposed into subsequences, each of which can be obtained from a block in π alternating jobs inside this block; (4) due to formulae (5) the length of the path is equal to $C_{\max}(\pi)$. The Property states that moves from $D \setminus W(\pi)$ are "more interesting" than those from $W(\pi)$, taking into account the possibility of an immediate improvement of the makespan after the move made. Computational experiments have shown that the move set $D \setminus W(\pi)$ is still too big. Next, we provide a subset of $D \setminus W(\pi)$, which has a significantly lower number of elements and allow us to find better makespans in a shorter time.

For each fixed x we consider at most one move to the right ($y > x$) and at most one to the left ($y < x$). Moves are associated with blocks. Let us take block $\mathscr{B}_l = (\pi(u_{l-1}), \ldots, \pi(u_l))$. We move the job $\pi(x)$, $u_{l-1} \leq x < u_l$ to the right on the smallest position y, such that $x < y \leq n$ and the Property cannot be applied to (x, y), i.e. $(x, y) \notin W_l(\pi)$. By symmetry, the job $\pi(x)$, $u_{l-1} < x \leq u_l$, is moved to the left on the greatest position y, such that $1 \leq y < x$ and $(x, y) \notin W_l(\pi)$. This definition allows *redundant* moves to be made (two moves $(x-1, x)$ and $(x, x-1)$ are redundant because provide the same permutation). Therefore, to avoid redundancy, one of these moves is removed. For illustration, the moves performed to the left and right as well as $W_l(\pi)$ for various type of blocks are shown in Fig. 3. According to the description given, for each block $\mathscr{B}_l$, $l = 1, \ldots, k$ we define the following set of moves to the right

$$ZR_l(\pi) = \begin{cases} \{(x, u_l) \quad : u_{l-1} < x < u_l\} & \text{if } t_l = 0, \\ \{(x, u_l) \quad : u_{l-1} \leq x < u_l\} & \text{if } t_l = 1, \\ \{(x, u_l + 1) : u_{l-1} < x < u_l\} & \text{if } t_l = 2. \end{cases} \quad (7)$$

and the set of moves to the left

$$ZL_l(\pi) = \begin{cases} \{(x, u_{l-1}) \quad : u_{l-1} < x < u_l\} & \text{if } t_l = 0, \\ \{(x, u_{l-1} - 1) : u_{l-1} < x < u_l\} & \text{if } t_l = 1, \\ \{(x, u_{l-1}) \quad : u_{l-1} < x \leq u_l\} & \text{if } t_l = 2. \end{cases} \quad (8)$$

Artificial moves $(x, n+1)$ and $(x, 0)$ are not performed. In the case "$b_l = 2$ and $t_l = 0$" we introduce an exclusion in the formula (7), setting $ZR_l(\pi) = \{(u_{l-1}, u_l)\}$. Finally, we propose the following set of moves

$$Z(\pi) = \bigcup_{l=1}^{k} (ZR_l(\pi) \cup ZL_l(\pi)).$$

From (7) and (8) we have $|ZR_l(\pi) \cup ZL_l(\pi)| \leq 2|\mathscr{B}_l| - 3$. Hence, $|Z(\pi)| \leq 2\sum_{l=1}^{k} |\mathscr{B}_l| - 3k = 2(n + k - 1) - 3k \leq 2n - 3$.

For the instance defined in Figs. 1 and 2 we have: for block $\mathscr{B}_1$ the moves $(2, 3)$ and $(1, 3)$, for $\mathscr{B}_2$ – $(5, 7)$, $(4, 7)$, $(4, 3)$, $(5, 3)$, and $(6, 3)$, for $\mathscr{B}_3$ – $(8, 9)$, $(7, 9)$, $(7, 6)$, and $(8, 6)$, for $\mathscr{B}_4$ – $(9, 10)$. There are 12 moves in the set $Z(\pi)$, whereas D contains 81 moves.

3.2 Search accelerator

We found that the calculation of the makespans for all neighbours is crucial for the search speed. Numerical results have shown that this consumes more than 90% of the total running time of *TS*. Therefore any acceleration of this calculation in desirable. Since the calculation of $C_{max}(\pi)$ needs a time $O(n)$ and since we have approximately $2n$ moves in $Z(\pi)$, the selection of the best move always requires the time of the order $O(n^2)$. We propose a method that allow us to perform these calculations in the time of order at least $O(nz)$ and at most $O(n^2)$.

Let us consider a permutation π and the corresponding graph $G(\pi)$. The main idea of the accelerator consists of avoiding superfluous, repetitive calculation, by finding suitable decomposiotion and aggregation calculation steps. To that end we need to find some other (besides A and B) auxiliary values, i.e. E_j, F_j, $j \in J$. Let E_j denote the length of the longest path between nodes j and $2n$, whereas F_j is the length of the longest path between the nodes $n + j$ and $2n$. These values are similar to A_j, B_j, and can be found in a time $O(n)$ using the following formula

$$F_{\pi(j)} = \max(F_{\pi(j+1)},\ E_{\pi(j+z+2)}) + b_{\pi(j)} \quad (9)$$

$$E_{\pi(j)} = \max(E_{\pi(j+1)},\ F_{\pi(j)}) + a_{\pi(j)} \quad (10)$$

for $j = n, n-1, \ldots, 1$, where $\pi(j) = n + 1$ if $j > n$, $E_{n+1} = 0 = F_{n+1}$.

In the sequel, we will show the method of finding $C_{\max}(\pi_v)$ for a move $v = (x, y)$ to the right, associated with a block $\mathscr{B}_l$, i.e. $v \in ZR_l(\pi)$. For the sake of simplicity, we set $\beta = \pi_{(x,y)}$. Denote by A', B' the job completion times in the permutation β. Actually, we only need these values for $j = x, \ldots, y$, and this can be done in a time $O(y - x)$ using formulae analogous to (1)–(2),

$$A'_{\beta(j)} = \max(A'_{\beta(j-1)},\ B'_{\beta(j-z-2)}) + a_{\beta(j)} \quad (11)$$

$$B'_{\beta(j)} = \max(B'_{\beta(j-1)},\ A'_{\beta(j)}) + b_{\beta(j)}, \quad (12)$$

where $A'_{\beta(j)} = A_{\beta(j)}$, $B'_{\beta(j)} = B_{\beta(j)}$, for $j < x$. Next, we can express the makespan as follows

$$C_{\max}(\beta) = \max\{A'_{\beta(y)} + E_{\pi(y+1)},\ B'_{\beta(y)} + F_{\pi(y+1)},\ \max_{y+1 \le i \le y+z+2}(B'_{\beta(i-z-2)} + E_{\pi(i)})\}, \quad (13)$$

Formula (13) can be performed in the time $O(z)$ having known appropriate values of A' and B'. Since these values can be found in time $O(y-z)$, the total time of computations necessary to find $C_{\max}(\beta)$ is of the order $O(y-x+z)$. Moves to the left can be analysed analogically. Next, note that the value $y-x$ is of the order $O(|B_l|)$ for all moves $v \in ZR_l(\pi) \cup ZL_l(\pi)$. Since $|ZR_l(\pi) \cup ZL_l(\pi)| \approx 2\,|\mathscr{B}_l|$, the values $C_{\max}(\pi_v)$ for all $v \in ZR_l(\pi) \cup ZL_l(\pi)$ can be computed in a time $O(z\,|\mathscr{B}_l| + |\mathscr{B}_l|^2)$. Finally, for all moves from $Z(\pi)$ the time $O(\sum_{l=1}^{k}(z\,|\mathscr{B}_l| + |\mathscr{B}_l|^2))$ is needed. This complexity depends crucially on the distribution of blocks in the permutation. If the number of blocks k tends to n, this complexity is close to $O(nz)$. In the opposite case (single long block) this complexity is $O(n^2)$. Assuming the uniform distribution of blocks, i.e. $|\mathscr{B}_l| \approx n/k$, we have the evaluation of the expected search time $O(n^2/k)$ per neighbourhood. Experimental tests show that k is between 5 and 6 for $n = 50$, about 8 for $n = 100$, and 10 for $n = 200$. Thus, the speed of the algorithm has been accelerated k times in average.

3.3 Tabu list

Tabu list T is a cyclic list of the length $maxT$. T is a realisation of the short-term search memory and stores some attributes of each move performed and permutation visited (in our case represented by a single pair of jobs per permutation and move). The list T is initiated by $maxT$ zero elements. Each newly added pair replaces the oldest one. Let $v = (x, y)$ be a move performed from π. Then we add to T the pair $(\pi(x), \pi(x+1))$ if $x < y$, and $(\pi(x-1), \pi(x))$, otherwise. The move $v = (x, y)$ from a permutation β cannot be performed (has status tabu) if at least one pair $(\beta(j), \beta(x))$, $j = x+1, \ldots, y$ is in T if $x < y$, and at least one pair $(\beta(x), \beta(j))$, $j = y, \ldots, x-1$ is in T, otherwise.

3.4 Algorithm TS

The algorithm is designed using the general scheme reported by Nowicki and Smutnicki (1996a). Starting from the initial permutation π^0 we perform successive iterations, generating the sequence of permutations $\pi^0, \pi^1, \ldots$. At the i-th iteration ($i = 0, 1, \ldots$), for the given permutation π^i we search the neighbourhood defined by the move set $Z(\pi^i)$. Denote the makespan for π^i by $C^i = C_{max}(\pi^i)$ and the best makespan found up to iteration i by $\hat{C}^i = \min_{j \in \{0,1,\ldots,i\}} C^j$. For each move $v \in Z(\pi^i)$ we calculate the makespan using the accelerator and we test the tabu status of v using the list T. The move v to be performed is selected as the one with the minimal makespan among unforbidden moves from $Z(\pi)$ and those forbidden but with the makespan less than $\hat{C}^i$. Then we update T, perform the move selected by setting $\pi^{i+1} = \pi^i_v$, set $C^{i+1} = C_{max}(\pi^{i+1})$, $\hat{C}^{i+1} = \min(\hat{C}^i, C^{i+1})$, and we go to the next iteration.

If C^i is strictly less than $\hat{C}^{i-1}$ (the makespan improvement has appeared in the iteration $i-1$), we record, in the long-term memory represented by the cyclic list L of length $maxL$, the *profitable search region*. We assume that $\hat{C}^{-1} = \infty$. This region is characterised by the permutation π^i, current state of tabu list T, and list N containing $maxC$ best unforbidden moves from $Z(\pi^i) \setminus \{v\}$. Each profitable region can be visited at most $maxC$ times, to start the search into other directions represented by the moves stored there. This technique is called the *back jump on the search trajectory* (Nowicki and Smutnicki, 1996a). If $maxI$ successive iterations has been performed without improving the makespan, we make the back jump to the last profitable search region stored in L. The search is restarted from the appropriate permutation and tabu list stored by performing the best move from the list N of this region. Obviously, the list N is reduced by the move taken. If N becomes empty, this region is removed from the region list L. The algorithm ends its activity when the list L becomes empty.

4 Computer experiments

Several heuristic approaches exist in the literature to solve the problem stated. In most of the cases, they implement and adopt algorithms developed for similar problems, e.g. those with zero buffers, or unlimited capacity buffers. In their recent survey, Hall and Sriskandrayah (1996), refer to the wide computational study of various heuristics in the paper of Leisten (Leisten, 1990). Following the conclusions provided there, the best algorithms for the problem stated are: (GG) – the algorithm of Gilmore and Gomory described by Hall and Sriskandarayah (1996) for the two-machine zero-buffer flow-shop; (CDS) – the algorithm of Campbell et al. (1970) for the flow-shop with unlimited capacity buffers; (NEH) – the algorithm of Nawaz et al. (1983) for the same problem, (L) – the algorithm BFPSE of Leisten (Leisten, 1990) designed specially for the problem with finite buffers. For the two-machine problem ($m = 2$), CDS generates only single permutation and thus is equivalent to Johnson's well-known algorithm. Algorithms GG and CDS have the computational complexity $O(n \log n)$, whereas L has this complexity $O(n^2)$. The original computational complexity $O(n^3 m)$ of NEH (for the problem with unlimited buffers) was reduced to $O(n^2 m)$ in an efficient implementation of Taillard (1990). There are still some doubts, not clarified in the paper by Leisten (1990), associated with the interpretation of NEH. Note that there are two different techniques: (i) the discrimination of partial permutations is made with the use of the makespan calculated for the problem with unlimited buffers, (ii) the discrimination is made with the use of the makespan calculated for the problem with finite capacity buffers. Denote by NEH-B the extension of NEH obtained by introducing the technique (ii). Due to the application of Taillard's idea combined with our accelerator from the Sect. 3.2, NEH-B can be also implemented as the procedure with the computational complexity $O(n^2 m)$. Bearing in mind the idea presented in the paper by Nowicki and Smutnicki 1993, we can reduce additionally, approximately

Table 1. The performance of TS

	Mean [%]		Max [%]		C^{TS}=LB	To C^{TS}		Total
n	η^R	η^{TS}	η^R	η^{TS}	[%]	iter	CPU [s]	CPU [s]
10	1.78	.00	6.16	.00	100	61	.01	.01
20	1.20	.00	2.97	.00	100	165	.07	.07
30	1.21	.05	2.26	.45	84	234	.15	.51
40	1.07	.03	2.01	.37	82	276	.27	.92
50	.98	.02	1.88	.19	74	465	.57	1.85
60	.99	.02	2.37	.14	68	355	.60	2.86
70	.95	.01	1.45	.11	82	621	1.27	2.77
80	.96	.01	1.33	.13	72	1032	2.74	5.54
90	.90	.02	1.54	.13	68	732	2.24	6.31
100	.95	.01	1.78	.11	72	848	2.89	6.89
150	.63	.01	1.02	.05	76	1063	6.93	13.02
200	.63	.01	1.13	.04	64	1379	15.15	31.55

Table 2. The influence of buffer size on TS

	Buffer size														
	1	2	5	1	2	5	1	2	5	1	2	5	1	2	5
n	Mean η^R			Mean η^{TS}			Max η^R			Max η^{TS}			$C^{TS} = LB$ [%]		
10	1.78	.21	.08	.00	.01	.00	6.16	2.69	1.36	.00	.27	.00	100	98	100
20	1.20	.16	.02	.00	.00	.00	2.97	1.46	.34	.00	.16	.00	100	98	100
30	1.21	.21	.00	.05	.01	.00	2.26	2.44	.10	.45	.13	.00	84	96	100
40	1.07	.38	.04	.03	.00	.00	2.01	2.09	1.02	.37	.07	.17	82	98	98
50	.98	.22	.04	.02	.01	.00	1.88	2.04	.83	.19	.07	.06	74	88	98
60	.99	.32	.01	.02	.00	.01	2.37	1.53	.22	.14	.06	.05	68	96	98
70	.95	.29	.00	.01	.00	.00	1.45	2.21	.05	.11	.09	.05	82	94	96
80	.96	.35	.03	.01	.00	.00	1.33	1.58	.85	.13	.08	.04	72	94	94
90	.90	.33	.01	.02	.00	.00	1.54	1.12	.21	.13	.04	.00	68	96	100
100	.95	.30	.02	.01	.01	.00	1.78	1.58	.44	.11	.10	.06	72	92	96
150	.63	.22	.01	.01	.00	.00	1.02	.88	.23	.05	.04	.00	76	98	100
200	.63	.20	.00	.01	.00	.00	1.13	1.07	.06	.04	.00	.00	64	100	100

twice, the real running time. Finally, in the experiments we will use only algorithms GG, CDS, NEH, NEH-B, and L.

The proposed algorithm TS needs several tuning parameters. Any heuristic algorithm can be used to find the initial permutation; we start our TS from GG. Next, to reach a compromise between the cost of calculation and the makespan quality, we experimentally set the following parameters: $maxT = 8$, $maxL = 5$, $maxC = 4$, $maxI = 2,000$ if the profitable search region is visited the first time and $maxI = 100$ otherwise.

Several tests were carried out to evaluate the performance of TS. Since in the literature, there are not standard benchmarks for this problem, we generated test instances. Three tests were carried out to show: (A) general performance of TS, (B) the influence of the buffer size on TS and scheduling results, (C) the influence of the similarity between jobs on TS the scheduling results.

A. General performance of TS

For each n=10,20,30,40,50,60,70,80,90,100,150,200 the sample of 50 *hard* instances were generated. Thus, 600 instances were selected. Job processing times were chosen as random integers with uniform distribution on interval [1,100], whereas the buffer size was always set at one ($z = 1$). We considered the instance as hard if the relative deviation $\eta^R = 100\% \cdot (C^R - LB)/LB$ was greater than a threshold value F (we have F=0.75% for $10 \leq n \leq 100$ and F=0.5% for $150 \leq n \leq 200$), where C^R was the reference makespan and LB was a lower bound. The value C^R was equal to the best among makespans provided by algorithms GG, CDS, NEH, NEH-B, L, whereas LB was found by solving classic two-machine flow-shop problems with unlimited capacity buffers. This selection of hard instances was made consciously; for instances generated without this limitation on η^R, TS found the makespan equal the lower bound (i.e. optimal) just after a few iterations (20 in average) in more than 95% of cases.

For each n we calculated the mean and maximum values of η^R, the mean and maximum values of η^{TS}, the fraction (in %) of instances for which algorithm TS provided the makespan C^{TS} equal LB (thus optimal), the mean number of iterations and the mean running time of TS to find the best makespan C^{TS}, and the total CPU time up to the end of the algorithm run. The algorithm was run on a PC with the processor 586/120. Test results are given in Table 1.

Note that because of the test specificity no reference solution is optimal and obviously no starting solution for TS is optimal. TS provided solutions very close to optimal (the mean value of η^{TS} is lower than 0.02% for all instances tested) and in most of the cases optimal. The fraction of makespans C^{TS} that are equal LB (thus optimal) are as follows: 100% for instances with $n \leq 20$, 68-84% for $20 < n \leq 100$, and over 60% for $n > 100$. In fact, TS provided many more optimal solutions since it was frequently observed that C^{TS} differs from LB only by at most one unit; simultaneously, LB evaluated optimal makespans with some finite accuracy, and the analysis of some selected instances showed a weakness of the lower bound technique used. On average, the number of iterations of TS necessary

Table 3. The influence of job similarity on TS

n	Mean [%] η^R	Mean [%] η^{TS}	Max [%] η^R	Max [%] η^{TS}	C^{TS}=LB [%]	To iter	C^{TS} CPU [s]	Total CPU [s]
10	2.38	.01	8.70	.20	90	125	.03	.10
20	1.92	.02	6.81	.12	90	290	.18	.38
30	1.86	.01	6.07	.12	94	361	.24	.40
40	1.64	.02	3.77	.33	86	349	.36	.81
50	1.43	.01	2.54	.16	80	420	.57	1.55
60	1.41	.05	3.47	.45	74	800	1.58	3.28
70	1.22	.01	2.51	.16	78	449	1.04	2.10
80	1.27	.02	2.64	.23	72	1022	2.54	5.32
90	1.15	.02	2.46	.31	84	1069	3.63	5.54
100	1.17	.02	2.38	.11	74	931	3.48	7.67
150	.73	.01	1.68	.07	84	1202	8.34	12.51
200	.72	.01	1.48	.06	74	1109	11.60	28.48

to find the best makespan is below 1,000 in average, and this value increases very slowly with n. All solution times are acceptable for practical applications.

B. The influence of buffer size on TS

In this test, we used the same instances as those chosen for the test A. Next, each such instance was solved three times, for the size of the buffer $z = 1, 2, 5$, respectively. The results are given Table 2.

First, note that the deviation η^R decreases 3–6 times if we only increase the buffer size from one unit to two units. A further drastic decrease of η^R is observed for a greater buffer size, and already for $z = 5$ is very close to 0.02%. These results show clearly that the use of a buffer greater than 5 is unreasonable, which is with contrast to the experiments of Leisten (1990) where instances up to $z = 20$ were tested for $n \leq 50$. It should be pointed out that in more than 99.5% of cases the best results are provided by NEH-B, which is dominant compared to the rest of the algorithms, ie. GG, CDS, NEH and L, in almost all tests. Algorithm TS provides the makespan optimal or very close to optimal in all tests, the mean value of η^{TS} is close to zero. The fraction of instances for which TS found the makespan equal to LB (thus optimal) is 88–100% for $z = 2, 5$ and this fraction increases with increasing z. The results obtained suggest that a two-machine system with a buffer of size one has the optimal makespan which is very close to the two-machine system with the unlimited capacity buffer.

C. The influence of job similarity on TS

In order to carry out this test, we designed a special generator of instances. We assumed that the job set was composed of $n/2$ small jobs and $n/2$ big jobs. For small jobs we assumed job processing times as random integers with uniform distribution on interval [1,50], whereas for long jobs on interval we assume [1,150]. The buffer size is set at one. Similarly, we selected only these instances which had the relative deviation η^R greater than a threshold value F already given. One can intuitively feel that for these instances the buffer has a significant impact on the result of scheduling. Results are given in Table 3. Comparing Table 3 with Table 1 we conclude that instances C are harder than A for all heuristics used; both the mean and maximal value of η^R have increased. On the other hand, TS behaves equally well in these instances. It provides solutions frequently optimal or very close to optimal in a short running time.

5 Conclusions

The fast and easily implementable tabu search algorithm for the two-machine flow-shop problem with inter-machine buffer has been presented. The presented approach can be extended to more general problems, e.g. m-machine flow-shop with buffers, ready times, due dates and min-max cost criterion.

References

1. Campbell HG, Dudek RA, Smith ML (1970) Heuristic algorithm for the n job m machine sequencing problem. Management Science 16:B630–637
2. Glover F (1989) Tabu search. Part I. ORSA Journal of Computing 1:190–206
3. Glover F (1990) Tabu search. Part II. ORSA Journal of Computing 2:4–32
4. Glover F, Laguna M (1993) Tabu search. In: Reeves C (ed) Modern heuristic techniques for combinatorial problems. Blackwell, Oxford, pp 70–141
5. Grabowski J, Skubalska E, Smutnicki C (1983) On flow shop scheduling with release and due dates to minimize maximum lateness. Journal of the Operational Research Society 34:615–620
6. Grabowski J, Nowicki E, Zdrzałka S (1986) A block approach for single-machine scheduling with release dates and due dates. European Journal of Operational Research 26:278–285
7. Hall NG, Sriskandarayah C (1996) A survey of machine scheduling problems with blocking and no-wait in process. Operations Research 44:510–525
8. Leisten R (1990) Flowshop sequencing problems with limited buffer storage. International Journal of Production Research 28:2085–2100
9. Nawaz M, Enscore Jr EE, Ham I (1983) A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. OMEGA International Journal of Management Science 11:91–95
10. Nowicki E, Smutnicki C (1993) New results in the worst-case analysis for flow-shop scheduling. Discrete Applied Mathematics 46:21–41
11. Nowicki E, Smutnicki C (1996) A fast taboo search algorithm for the job shop problem. Management Science 42:797–813
12. Nowicki E, Smutnicki C (1996a) A fast taboo search algorithm for the permutation flow-shop problem. European Journal of Operational Research 91:160–175
13. Papadimitriou CH, Kanellakis PC (1980) Flowshop scheduling with limited temporary storage, Journal of the Association for Computing Machinery 27:533–549
14. Smutnicki C (1986) The block approach in the flow-shop problem with storage constraints (Polish), Zeszyty Naukowe Poltechniki Śląskiej: sAutomatyka 84:223–233
15. Taillard E (1990) Some efficient heuristic methods for flow shop sequencing. European Journal of Operational Research 47:65–74