INFORMS Journal on Computing
Vol. 16, No. 2, Spring 2004, pp. 133–151
ISSN 0899-1499 | EISSN 1526-5528 | 04 | 1602 | 0133

DOI 10.1287/ijoc.1030.0036

An Ejection Chain Approach for the Generalized Assignment Problem

Mutsunori Yagiura, Toshihide Ibaraki
Department of Applied Mathematics and Physics, Graduate School of Informatics,
Kyoto University, Kyoto 606-8501, Japan {yagiura@i.kyoto-u.ac.jp, ibaraki@i.kyoto-u.ac.jp}

Fred Glover
Leeds School of Business, University of Colorado at Boulder, Campus Box 419,
Boulder, Colorado 80309-0419, USA, fred.glover@colorado.edu

We propose a tabu search algorithm for the generalized assignment problem, which is one of the representative combinatorial optimization problems known to be NP-hard. The algorithm features an ejection chain approach, which is embedded in a neighborhood construction to create more complex and powerful moves. We also incorporate an adaptive mechanism for adjusting search parameters, to maintain a balance between visits to feasible and infeasible regions. Computational results on benchmark instances of small sizes show that the method obtains solutions that are optimal or that deviate by at most 0.16% from the best known solutions. Comparisons with other approaches from the literature show that, for instances of larger sizes, our method obtains the best solutions among all heuristics tested.

Key words: generalized assignment problem; ejection chain; adaptive parameter adjustment; metaheuristics; tabu search; local search
History: Accepted by Michel Gendreau; received June 1999; revised June 2001; accepted January 2003.

1. Introduction

Metaheuristic algorithms are widely acknowledged to be powerful tools to deal with hard combinatorial optimization problems. Most metaheuristics are based on local search, in which the design of an appropriate neighborhood is crucial. An *ejection chain* is an embedded neighborhood construction that compounds simple moves to create more complex and powerful moves. Ejection chains generalize the alternating path constructions of graph theory (Berge 1962, Edmonds 1965) and also generalize the well-known Lin and Kernighan algorithms (Kernighan and Lin 1970, Lin and Kernighan 1973), which were successfully applied to graph partitioning and traveling salesman problems. Recent applications of the ejection chain approach include Laguna et al. (1995), Pesch and Glover (1997), Rego (1998a, b), and Rego and Roucairol (1996).

In this paper, we propose an ejection chain approach under the framework of *tabu search* (TS) for the *generalized assignment problem* (GAP), which is known to be NP-hard (Sahni and Gonzalez 1976). GAP seeks a minimum cost assignment of n jobs to m agents subject to a resource constraint for each agent. Among various heuristic algorithms developed for GAP are: A combination of the greedy method and local search by Martello and Toth (1981, 1990); a tabu search and simulated annealing approach by Osman (1995); a genetic algorithm by Chu and Beasley (1997); variable depth search methods by Amini and Racer (1995) and Racer and Amini (1994); a tabu search approach by Laguna et al. (1995) (which is proposed for a generalization of GAP); a set partitioning heuristic by Cattrysse et al. (1994); a relaxation heuristic by Lorena and Narciso (1996); a GRASP and MAX-MIN ant system combined with local search and tabu search by Lourenço and Serra (2002); a linear relaxation heuristic by Trick (1992); and so on. Many exact algorithms have also been proposed (e.g., Nauss 2003, Savelsbergh 1997). A simpler version of an ejection chain approach has also been proposed for the GAP in Laguna et al. (1995). Our ejection chain is based on the idea described in Glover (1997).

In our previous papers (Yagiura et al. 1998, 1999), we proposed variable depth search algorithms for GAP having the following two features. The first is the alternating use of shift and swap neighborhoods, which we call *SSS* (Shift and Subsequent Swaps) *probe* in Yagiura et al. (1998). The shift neighborhood is defined to be the set of solutions obtainable by changing the assignment of one job, while the swap neighborhood is the set of solutions obtainable by exchanging the assignments of two jobs. The second feature adopts the strategic oscillation component of tabu search by allowing the search to penetrate into the infeasible region. This alleviates the difficulty of

searching solely within the feasible region, which we encounter in those instances where the feasible region is very small or consists of many separate small regions. In fact, the problem of judging the existence of a feasible solution for GAP is already known to be NP-complete. The strategy of inducing the search to enter the infeasible region was also embodied in the earlier tabu search GAP approach of Laguna et al. (1995). Based on the success of our previous algorithms, these features are also incorporated in our new algorithm in a more sophisticated manner.

An ejection chain move considered in this paper is a sequence of shift moves, in which every two successive moves share a common agent. The ejection chain neighborhood is the set of solutions obtainable by such ejection chain moves. The length of an ejection chain move is the number of shift moves in the sequence. Both the shift and swap neighborhoods are subsets of the ejection chain neighborhood, since a shift (respectively, swap) move is an ejection chain move of length one (respectively, two). However, the size of the ejection chain neighborhood can become exponential unless intelligently controlled. Therefore, in our implementation, we consider only a special subset of alternatives that is divided into three neighborhoods called *shift*, *double shift*, and *long chain*, where a double shift move is an ejection chain move of length two, and a long chain move is an ejection chain of any length. These neighborhoods are further suppressed to manageable sizes by using heuristic rules based on *Lagrangian relative cost*, as will be explained in detail in subsequent sections. The sizes of shift, double shift, and long chain neighborhoods are $O(mn)$, $O(n\max\{m, \log n\})$, and $O(n^2)$, respectively. We also show that the expected size of the long chain neighborhood is $O(n^{(3/2)+\varepsilon})$ for an arbitrarily small positive ε under a simplified random model. It is observed in our experiments that this expected size accurately represents reality. In our algorithm, these three neighborhoods are used alternately to form an improving phase, which is called *EC* (*ejection chain*) *probe* in this paper.

As in our previous algorithms, we allow searching into the infeasible region, but infeasible solutions are penalized according to the degree of infeasibility. The search balance between the feasible and infeasible regions is controlled by the parameters in the penalty functions. Preliminary experimentation disclosed that the performance of the proposed algorithm can be significantly influenced by the values of parameters. We therefore incorporate an adaptive adjustment mechanism for determining their appropriate values.

Extensive comparisons with other existing heuristics have been conducted using benchmark instances known as types C, D, and E. For problems in a size range that can be handled by an exact branch-and-bound method, our algorithm obtains solutions that are optimal or that deviate by at most 0.16% from the best obtained by the exact method. For larger problems, our algorithm outperforms all other methods tested, obtaining better solutions to all problem instances.

2. Generalized Assignment Problem

Given n jobs $J = \{1, \ldots, n\}$ and m agents $I = \{1, \ldots, m\}$, we undertake to determine a minimum cost assignment subject to assigning each job to exactly one agent and satisfying a resource constraint for each agent. Assigning job j to agent i incurs a cost of c_{ij} and consumes an amount a_{ij} of a resource, whereas the total amount of the resource available at agent i is b_i. An assignment is a mapping $\sigma\colon J \to I$, where $\sigma(j) = i$ means that job j is assigned to agent i. For convenience, we define a 0-1 variable x_{ij} for each pair of $i \in I$ and $j \in J$ by

$$x_{ij} = 1 \quad \Longleftrightarrow \quad \sigma(j) = i.$$

Then the GAP is formulated as follows:

$$\begin{aligned} \text{minimize} \quad & cost(\sigma) = \textstyle\sum_{i\in I}\sum_{j\in J} c_{ij}x_{ij} \\ \text{subject to} \quad & \textstyle\sum_{j\in J} a_{ij}x_{ij} \le b_i, \quad \forall\, i \in I \\ & \textstyle\sum_{i\in I} x_{ij} = 1, \quad \forall\, j \in J \\ & x_{ij} \in \{0,1\}, \quad \forall\, i \in I \quad \text{and} \quad \forall\, j \in J. \end{aligned} \tag{1}$$

GAP is known to be NP-hard (e.g., Sahni and Gonzalez 1976), and the (supposedly) simpler problem of judging the existence of a feasible solution for GAP is NP-complete, since the partition problem (Garey and Johnson 1979) can be reduced to this problem with $m = 2$.

As we will show some computational results in §3 to report the behavior of our algorithm, we first explain the instances used for the experiments. There are five types of benchmark instances called types A, B, C, D, and E (Chu and Beasley (1997), Laguna et al. 1995). Out of these, we use the three types C, D, and E, since the other two are too easy to see differences among the tested algorithms. Instances of these types are generated as follows:

Type C: a_{ij} are random integers distributed uniformly on $[5, 25]$, c_{ij} are random integers distributed uniformly on $[10, 50]$, and $b_i = 0.8\sum_{j\in J} a_{ij}/m$.

Type D: a_{ij} are random integers distributed uniformly on $[1, 100]$, $c_{ij} = 111 - a_{ij} + e_1$, where e_1 are random integers distributed uniformly on $[-10, 10]$, and $b_i = 0.8\sum_{j\in J} a_{ij}/m$.

Type E: $a_{ij} = 1 - 10\ln e_2$, where e_2 are random numbers distributed uniformly on $(0, 1]$, $c_{ij} = 1000/a_{ij} - 10e_3$, where e_3 are random numbers distributed uniformly on $[0, 1]$, and $b_i = 0.8\sum_{j\in J} a_{ij}/m$.

Note that types D and E are usually harder than type C, since c_{ij} and a_{ij} are inversely correlated. We tested the following three sets of problem instances.

SMALL: Total of 60 instances of type C with n up to 60. These are taken from the OR-Library (http://mscmga.ms.ic.ac.uk/jeb/orlib/gapinfo.html). Their optimal values are also available from the OR-Library.

MEDIUM: Total of 18 instances of types C, D, and E with n up to 200. Among them, types C and D instances were taken from the OR-Library, and we generated type E instances, which are available at http://www-or.amp.i.kyoto-u.ac.jp/~yagiura/gap/.

LARGE: Total of 27 instances of types C, D, and E with n up to 1,600, all of which we generated. These instances are available at the above site.

3. Basic Components of the Tabu Search Algorithm

We call the tabu search algorithm developed in this paper *Algorithm TS*. We describe its basic components in this section.

3.1. Local Search, Search Space, and Basic Neighborhoods

Local search (LS) starts from an initial solution σ and repeats replacing σ with a better solution in its *neighborhood* $N(\sigma)$ until no better solution is found in $N(\sigma)$. The resulting solution σ is *locally optimal* in the sense that no better solution exists in its neighborhood.

Two types of neighborhoods, a shift neighborhood N_{shift} and a swap neighborhood N_{swap}, are usually used, where

$$N_{\text{shift}}(\sigma) = \{\sigma' \mid \sigma' \text{ is obtained from } \sigma \text{ by changing the assignment of one job}\},$$

$$N_{\text{swap}}(\sigma) = \{\sigma' \mid \sigma' \text{ is obtained from } \sigma \text{ by exchanging the assignments of two jobs}\}.$$

Other neighborhoods used in our TS approach will be explained later.

When the search visits the infeasible region, we evaluate the solutions by an objective function penalized by infeasibility. Here we use the following function:

$$pcost(\sigma) = cost(\sigma) + \sum_{i \in I} \alpha_i p_i(\sigma), \tag{2}$$

where

$$p_i(\sigma) = \max\left\{0, \left(\sum_{j \in J,\, \sigma(j)=i} a_{ij}\right) - b_i\right\}.$$

The parameters α_i (>0) can be given as fixed constants or can be adaptively controlled during the search; see §5 for details. A locally optimal solution under *pcost* may not always be feasible; however, we can increase the probability of obtaining feasible solutions by using a large value of α_i.

The local search procedure using a neighborhood N (e.g., $N = N_{\text{shift}}$ or N_{swap}) and an evaluation function f (e.g., $f = cost$ or $pcost$), starting from an initial solution σ (not necessarily feasible), is formally described as follows.

Algorithm LS (N, f, σ).

Step 1. If there is a solution $\sigma' \in N(\sigma)$ such that $f(\sigma') < f(\sigma)$, set $\sigma := \sigma'$ and return to Step 1. Otherwise go to Step 2.

Step 2. ($f(\sigma) \le f(\sigma')$ holds for all $\sigma' \in N(\sigma)$.) Output σ and stop.

We implement Step 1 as follows. Solutions in $N(\sigma)$ are scanned according to a prespecified order, and the first improved solution found is immediately accepted as the next solution. This strategy is commonly used in implementing LS algorithms and is called the *first admissible move strategy*.

Local search is often applied to a number of randomly generated initial solutions, and the best among the locally optimal solutions obtained is the output. This is called *random multi-start local search*.

3.2. Ejection Chain

In this section, we explain the ejection chain neighborhood. To form an ejection chain, *ejection moves* and *trial moves* are alternately executed. An ejection move is to remove a job j from the agent $\sigma(j)$ to generate an incomplete solution, where job j remains free. Such a solution is called the *reference structure*. Another type of ejection move is to shift a job to the agent from which another job has just been ejected in the previous ejection move. This type of ejection move is applied to reference structures. Finally, a trial move is to assign the free job into an agent to make a complete solution. For example, if we eject a job j (as an ejection move) and add it into another agent (as a trial move) immediately, this is a shift move. That is, a shift move is a special case of an ejection chain move.

The neighborhood is divided into three types of neighborhoods, called *shift*, *double shift*, and *long chain*. The shift neighborhood was already explained in §3.1. The other two are explained in §3.2.2–§3.2.4. As part of searching the long chain and double shift neighborhoods, we use a subgradient phase, which is described in §3.2.1. Then the whole framework of the search in the ejection chain neighborhood is summarized in §3.2.5.

3.2.1. Subgradient Phase. We construct ejection chains by exploiting the information from a

Lagrangian relaxation problem of (1):

$$\begin{aligned} L(v) = \min\ & \textstyle\sum_{i\in I}\sum_{j\in J}(c_{ij}-v_j)x_{ij} - \sum_{j\in J} v_j \\ \text{s.t.}\ & \textstyle\sum_{j\in J} a_{ij}x_{ij} \le b_i, \quad \forall\, i \in I \\ & 0 \le x_{ij} \le 1, \quad \forall\, i \in I \text{ and } \forall\, j \in J, \end{aligned} \tag{3}$$

where $v = (v_1, v_2, \ldots, v_n) \in R^n$ is a Lagrangian multiplier vector given to the constraint $\sum_{i\in I} x_{ij} = 1$, $j \in J$, and

$$c_{ij}(v) = c_{ij} - v_j$$

is called the *Lagrangian relative cost*. Problem (4) decomposes into m real-valued knapsack problems and thus can be solved in $O(mn)$ time (a linear-time algorithm can be found in, e.g., Martello and Toth 1990). For any v, $L(v)$ gives a lower bound on the objective value of problem (1), and the Lagrangian dual problem is to find a $v \in R^n$ that maximizes the lower bound $L(v)$. Any optimal solution v^* to the dual of the linear programming (LP) relaxation of (1),

$$\begin{aligned} \max\ & \textstyle\sum_{j\in J} v_j - \sum_{i\in I} b_i u_i \\ \text{s.t.}\ & v_j - a_{ij}u_i \le c_{ij}, \quad \forall\, i \in I \text{ and } \forall\, j \in J, \\ & u_i \ge 0, \quad \forall\, i \in I, \end{aligned} \tag{4}$$

is an optimal solution to the Lagrangian dual (Fisher 1981). However, computing such v^* by solving (4) is expensive for large scale instances.

A fast method for finding a near-optimal v is the *subgradient method* (Fisher 1981, Held and Karp 1971), which is based on the subgradients,

$$s_j(v) = 1 - \sum_{i\in I} x_{ij}, \qquad \forall\, j \in J.$$

This approach generates a sequence $v^{(0)}, v^{(1)}, v^{(2)}, \ldots$, where $v^{(0)}$ is defined arbitrarily, and $v^{(k)}$ $(k \ge 1)$ are updated by, e.g.,

$$v_j^{(k+1)} = v_j^{(k)} + \lambda \frac{UB - L(v^{(k)})}{\|s(v^{(k)})\|^2} s_j(v^{(k)}), \qquad j \in J,$$

where UB is an upper bound on the objective value of problem (1), and λ is a step-size parameter satisfying $0 < \lambda \le 2$. In our implementation, λ is initially set to two and halved whenever the lower bound does not increase in *chkevery_sg* consecutive iterations. The iteration is terminated when the lower bound is not improved in the last *maxitr_sg* iterations or λ becomes smaller than *minstep_sg*. (The values *chkevery_sg*, *maxitr_sg*, and *minstep_sg* are program parameters.) Then the vector $\tilde{v}$ when the best lower bound is attained during this iteration is chosen for the use in our algorithm.

In each subgradient phase, the current best upper bound is used as UB, and the previous best v is used as the initial $v^{(0)}$ except for the first subgradient call, where we use $v^{(0)} := (0, 0, \ldots, 0)$. Note that the subgradient phase is called only after a feasible solution is found by algorithm TS (because otherwise UB is not available). Once a feasible solution is found, the subgradient phase is called whenever the current best upper bound UB is updated, provided that the best lower bound was found in the last call of the subgradient phase. That is, the subgradient phase will not be called again once it fails to update the best lower bound. (Actually, the number of calls to the subgradient phase is only a few times in most cases. It reached ten times in only two cases among 225 runs, in which each of 45 instances from MEDIUM and LARGE with up to $n = 1{,}600$ and $m = 80$ were solved five times.) The subgradient phase used in our algorithm is formally described as follows for a given upper bound UB and an initial vector $v^{(0)}$.

ALGORITHM SUBGRADIENT (UB, $v^{(0)}$).

Step 1. If $v^{(0)} = (0, 0, \ldots, 0)$, let c_j^{**} be the second smallest c_{ij} among all $i \in I$ for each $j \in J$, and let $v_j^{(0)} := c_j^{**}$. Let $\tilde{v} := v^{(0)}$, $\lambda := 2$, *counter* $:= 0$, and $k := 0$.

Step 2. Let $v_j^{(k+1)} := v_j^{(k)} + \lambda(UB - L(v^{(k)}))s_j(v^{(k)})/\|s(v^{(k)})\|^2$, for all $j \in J$.

Step 3. If $L(v^{(k+1)}) > L(\tilde{v})$, let $\tilde{v} := v^{(k+1)}$ and *counter* $:= 0$; otherwise let *counter* $:=$ *counter* $+ 1$.

Step 4. If *counter* > 0 and *counter* $\equiv 0 \pmod{\textit{chkevery\_sg}}$ hold, then let $\lambda := \lambda/2$.

Step 5. If *counter* $\ge$ *maxitr_sg* or $\lambda <$ *minstep_sg*, then output $\tilde{v}$ and stop; otherwise let $k := k + 1$ and return to Step 2.

We set *chkevery_sg* $= 20$, *maxitr_sg* $= 300$, and *minstep_sg* $= 0.005$ in our experiments.

3.2.2. Long Chain Neighborhood. The long chain neighborhood is defined as follows. We first eject a job j_0 from agent $\sigma(j_0)$ (i.e., an ejection move), which causes the amount of resource available at agent $\sigma(j_0)$ to increase. In the resulting reference structure, job j_0 remains free. Let *avail* be the resulting amount of resource, and let j_1 be the job whose shift into $\sigma(j_0)$ is most profitable among the jobs satisfying $a_{\sigma(j_0), j_1} \le$ *avail* (where the profit will be defined later by function (6)). Job j_1 is then shifted into agent $\sigma(j_0)$. That is, the ejection move of j_1 is triggered by the ejection of j_0. At this point, we evaluate some solutions obtainable by trial moves of assigning job j_0 into some agents (e.g., $\sigma(j_1)$). The next ejection move is applied to the previous reference structure, not to the solutions generated by the trial moves. Since the amount of resource available at agent $\sigma(j_1)$ increases, a job j_2 is chosen by a similar rule and is shifted into $\sigma(j_1)$, i.e., the ejection of j_2 is triggered by the ejection of j_1. These steps are then repeated until the stopping criterion is satisfied.

In this process, we use the following functions, where $c_{ij}(\tilde{v}) = c_{ij} - \tilde{v}_j$ is the relative cost associated with the $\tilde{v}$ computed by the subgradient phase of §3.2.1 ($\tilde{v} = 0$ is used before the first call to the subgradient phase):

$$avail(j) = \begin{cases} a_{\sigma(j),j} - p_{\sigma(j)}(\sigma), \\ \quad \text{if } a_{\sigma(j),j} > p_{\sigma(j)}(\sigma) \\ a_{\sigma(j),j}, \quad \text{otherwise} \end{cases} \tag{5}$$

$$score(i,j) = -c_{ij}(\tilde{v}) \tag{6}$$

$$J' = \{k \in J \mid \exists h \in J \text{ s.t. } a_{\sigma(k),h} \le avail(k) \text{ and } \sigma(h) \ne \sigma(k)\} \tag{7}$$

$$best\_score(j) = \max\{score(\sigma(j),k) \mid k \in J', \sigma(k) \ne \sigma(j) \text{ and } a_{\sigma(j),k} \le avail(j)\} \tag{8}$$

$$B(j) = \{k \in J' \mid score(\sigma(j),k) = best\_score(j), \sigma(k) \ne \sigma(j) \text{ and } a_{\sigma(j),k} \le avail(j)\}. \tag{9}$$

The function $avail(j)$ defines the resource made available by the ejection of job j. Note that this may be different from the amount of resource actually available at agent $\sigma(j)$. The function $score(i,j)$ defines the attractiveness of the shift move of job j into agent i (where larger scores are better). We use $score(i,j) = -c_{ij}(\tilde{v})$ instead of the direct cost $-c_{ij}$ because it is known that the relative cost $c_{ij}(\tilde{v})$ provides more reliable information about the cost increase when the current solution is modified by changing the assignment of job j to agent i. (For example, the optimal solution remains the same even a modification of c_{ij} to $c_{ij} + \gamma_j$ (γ_j is a constant) is added for all agents i and jobs j; however, the above strategy is affected by such changes if $score(i,j) = -c_{ij}$ is used, but not if $score(i,j) = -c_{ij}(\tilde{v})$ is used, since the Lagrangian multiplier $\tilde{v}_j$ has the effect of removing the influence from such γ_j.) J' is the set of jobs j whose ejection enables at least one move to agent $\sigma(j)$ from another agent. The function $best\_score(j)$ keeps the maximum $score(\sigma(j),k)$ over jobs $k \in J'$ that are not assigned to agent $\sigma(j)$ and whose resource requirements are not more than $avail(j)$. Finally, $B(j)$ is the set of jobs that attain $best\_score(j)$. (There are other ways to define *avail* and *score*, and we tested some of them. Readers who are interested in these details may refer to the discussion in Appendix 1.)

The algorithm to generate solutions in the long chain neighborhood from the current solution σ is formally described as follows, where S is the set of jobs already chosen for j_0, and l is the length of the current ejection chain. Job j_0 is chosen in Step 2 and the most profitable agent i^* to receive j_0 is found in Step 3. Then ejection chain moves of various lengths starting from j_0 are output by repeating Steps 4 through 6. These steps are executed for all $j_0 \in J'$.

Algorithm LONG_CHAIN(σ).

Step 1. Let $S := \varnothing$.

Step 2. If $S = J'$, stop; otherwise randomly choose a $j_0 \in J' \backslash S$, let $S := S \cup \{j_0\}$, and $\sigma' := \sigma$. (Job j_0 is ejected from $\sigma(j_0)$.)

Step 3. Let i^* be the agent i that minimizes $c_{ij_0} + \alpha_i \max\{0, (\sum_{j \in J, \sigma(j)=i} a_{ij}) + a_{ij_0} - b_i\}$ among all agents $i \in I \backslash \{\sigma(j_0)\}$, and let $l := 0$.

Step 4. If $B(j_l) \backslash \{j_k \mid k \le l\} = \varnothing$, return to Step 2; otherwise let $l := l + 1$ and proceed to Step 5.

Step 5. Randomly choose $j_l \in B(j_{l-1}) \backslash \{j_k \mid k \le l-1\}$ and let $\sigma'(j_l) := \sigma(j_{l-1})$ (an ejection move of job j_l). Then execute the following Steps (a) and (b) (two trial moves).

(a) Let $\sigma'(j_0) := \sigma(j_l)$ (j_0 is inserted into $\sigma(j_l)$), and output σ'.

(b) Let $\sigma'(j_0) := i^*$ (j_0 is inserted into i^*), and output σ'.

Step 6. Return to Step 4.

The generation mechanism of algorithm LONG_CHAIN is illustrated in Figure 1. In this figure, a large rectangle represents an agent and small rectangles in it represent assigned jobs. The height of a rectangle of job j (respectively, agent i) represents a_{ij} (respectively, b_i). The current solution σ is illustrated in the top left corner, and other reference structures are given in the left column. The (complete) solutions generated by trial moves from these reference structures are shown in the right column.

The set of solutions output in Step 5(a) (respectively, Step 5(b)) is denoted $N_{\text{long\_cyc}}(\sigma)$ (respectively, $N_{\text{long\_path}}(\sigma)$), and the set $N_{\text{long\_cyc}}(\sigma) \cup N_{\text{long\_path}}(\sigma)$ is denoted $N_{\text{long}}(\sigma)$. For a solution $\sigma' \in N_{\text{long}}(\sigma)$, the sequence of shift moves executed to generate σ' from σ is called the *long-chain move*, and its length is defined to be the number of such shift moves.

The size of $N_{\text{long}}(\sigma)$ is $O(n^2)$ in the worst case, since the maximum length of a long chain is n and, hence, the total number of repetitions of Steps 4 through 6 in the complete execution of LONG_CHAIN is $O(n^2)$. However, our experimentation confirms that the average length of a long chain is usually much shorter than n, which means that $|N_{\text{long}}(\sigma)|$ is much smaller than $O(n^2)$. In other words, the iteration of Steps 4 through 6 may stop with an l much smaller than n, since $B(j_l) \backslash \{j_k \mid k \le l\} = \varnothing$ usually holds for small l (i.e., all jobs in $B(j_l)$ are already used in the current long chain move). This is indicated in the computational results of Table 1. The left two columns of Table 1 show the lengths of long chains for benchmark instances with up to $n = 1{,}600$. From the table, we can observe that the average length

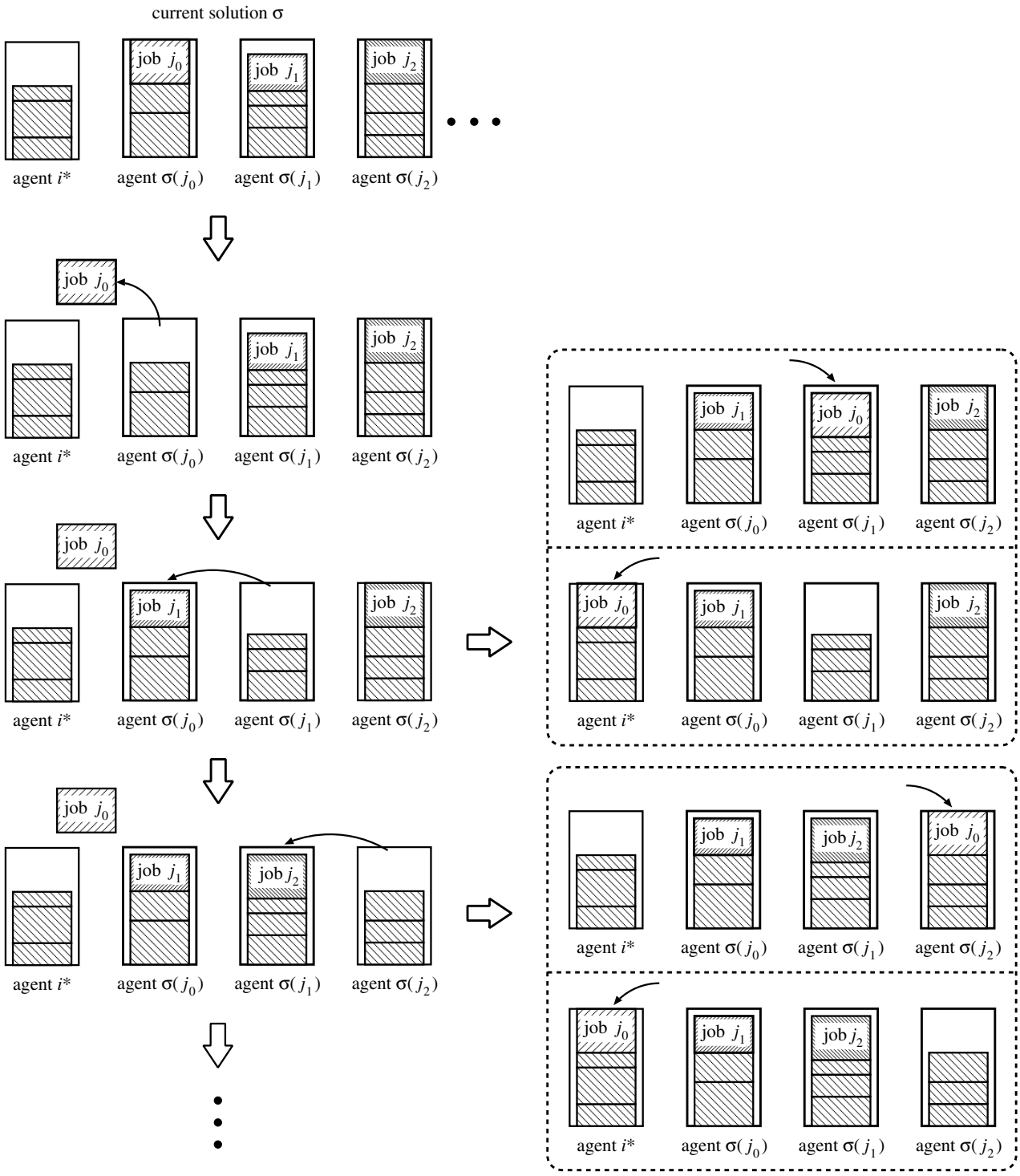

Figure 1 An Illustrative Example of Long Chain Moves

is around $\sqrt{n}$ or smaller. Furthermore, we can theoretically show that, under a simple random model, the expected length of a long chain is bounded from above by $n^{(1/2)+\varepsilon}$ for an arbitrary small $\varepsilon > 0$ (see Appendix 2 for details), which implies $|N_{\text{long}}(\sigma)| = O(n^{(3/2)+\varepsilon})$. This analysis gives intuitive support to the results in Table 1.

3.2.3. Efficient Implementation of Algorithm LONG_CHAIN. The most expensive computation in

Table 1 The Length of Long Chain Moves and the Number of Scans in *jlist*

			Length of a long chain		$l^{(j)}_{\max}$ of INITIALIZE_B		$l^{(j)}_{\max}$ ($l^{(j)}_{\max}$/*secondsh_max*) of DOUBLE_SHIFT	
Type	n	m	Average	Max.	Average	Max.	Average	Max.
C	100	5	5.8	21	6.8	15	15.0 (2.1)	26 (3.7)
C	100	10	7.2	25	4.4	12	15.2 (1.5)	25 (2.5)
C	100	20	5.9	26	3.1	9	21.3 (1.1)	28 (1.4)
C	200	5	5.9	26	9.8	28	23.0 (2.9)	41 (5.1)
C	200	10	9.1	33	7.0	19	19.7 (2.0)	34 (3.4)
C	200	20	10.7	39	4.6	12	25.3 (1.3)	34 (1.7)
C	400	10	11.2	40	9.5	26	26.6 (2.7)	46 (4.6)
C	400	20	15.1	55	6.7	19	30.6 (1.5)	45 (2.2)
C	400	40	15.4	53	5.0	12	45.1 (1.1)	53 (1.3)
C	900	15	15.4	59	15.5	37	40.3 (2.7)	66 (4.4)
C	900	30	20.6	73	9.6	24	46.5 (1.6)	65 (2.2)
C	900	60	23.6	80	6.0	16	67.7 (1.1)	81 (1.4)
C	1,600	20	18.7	72	22.0	54	56.0 (2.8)	91 (4.5)
C	1,600	40	24.8	94	12.7	29	61.9 (1.5)	86 (2.1)
C	1,600	80	11.2	39	5.5	23	91.7 (1.1)	107 (1.3)
D	100	5	8.9	37	2.8	13	10.1 (1.4)	26 (3.7)
D	100	10	8.7	37	2.2	10	11.2 (1.1)	22 (2.2)
D	100	20	10.6	45	1.9	9	18.7 (0.9)	31 (1.6)
D	200	5	12.1	49	3.5	20	13.0 (1.6)	41 (5.1)
D	200	10	13.7	55	2.4	14	12.7 (1.3)	29 (2.9)
D	200	20	15.4	64	2.1	10	21.3 (1.1)	38 (1.9)
D	400	10	18.8	76	2.9	21	14.6 (1.5)	43 (4.3)
D	400	20	22.8	84	2.3	18	22.8 (1.1)	45 (2.2)
D	400	40	21.3	82	2.0	11	39.3 (1.0)	60 (1.5)
D	900	15	27.0	114	3.2	26	21.0 (1.4)	67 (4.5)
D	900	30	31.5	136	2.4	26	33.8 (1.1)	74 (2.5)
D	900	60	35.4	138	2.0	16	59.4 (1.0)	91 (1.5)
D	1,600	20	38.4	179	3.2	30	26.9 (1.3)	79 (4.0)
D	1,600	40	39.6	165	2.4	24	44.6 (1.1)	89 (2.2)
D	1,600	80	51.1	209	2.0	21	80.8 (1.0)	135 (1.7)
E	100	5	6.1	25	4.7	28	13.9 (2.0)	40 (5.7)
E	100	10	7.3	32	3.1	14	14.5 (1.5)	30 (3.0)
E	100	20	7.3	36	2.2	10	22.7 (1.1)	30 (1.5)
E	200	5	6.6	28	7.1	37	20.3 (2.5)	49 (6.1)
E	200	10	9.6	40	4.3	28	17.2 (1.7)	43 (4.3)
E	200	20	9.0	40	3.2	25	25.4 (1.3)	49 (2.5)
E	400	10	10.9	42	7.1	20	20.7 (2.1)	62 (6.2)
E	400	20	9.4	43	5.1	14	29.1 (1.5)	53 (2.6)
E	400	40	13.2	58	3.6	12	45.9 (1.1)	75 (1.9)
E	900	15	10.6	45	10.3	29	26.5 (1.8)	86 (5.7)
E	900	30	13.2	56	7.5	21	41.7 (1.4)	80 (2.7)
E	900	60	17.0	120	4.6	16	68.6 (1.1)	109 (1.8)
E	1,600	20	10.8	62	13.8	41	37.6 (1.9)	94 (4.7)
E	1,600	40	15.8	73	8.9	30	52.1 (1.3)	83 (2.1)
E	1,600	80	22.5	213	5.6	18	92.0 (1.2)	119 (1.5)

algorithm LONG_CHAIN is the calculation of $B(j)$. This set function can be prepared at the beginning of algorithm LONG_CHAIN, since it is independent of the sequence $(j_0, j_1, \ldots, j_l)$. Its time complexity is $O(n^2)$ if implemented naively.

To justify the importance of improving the computation of B, we note that the rest of LONG_CHAIN usually requires much less time than $O(n^2)$ for the following reasons. The time needed to generate a solution σ' in $N_{\text{long}}(\sigma)$ is O(1). (As it is enough to evaluate the difference $pcost(\sigma) - pcost(\sigma')$ in algorithm TS, we output only $pcost(\sigma')$ instead of the entire sequence σ'.) Therefore, the time complexity of algorithm LONG_CHAIN, excluding the computation of B, is proportional to $|N_{\text{long}}(\sigma)|$, which is $O(n^{3/2})$ in practice as discussed in §3.2.2.

Now, we adopt the following data structure to speed up the computation of $B(j)$. Let $a_i^{(k)}$ be the kth value from the smallest among $a_{i1}, a_{i2}, \ldots, a_{in}$. That is, $a_i^{(1)}, a_i^{(2)}, \ldots, a_i^{(n)}$ are obtained by sorting $a_{i1}, a_{i2}, \ldots, a_{in}$ in nondecreasing order. We define $a_i^{(0)} = 0$ for convenience. For simplicity, in the following definition of

jlist, we assume $score(i, j) \neq score(i, j')$ holds for all $j, j' \in J$ with $j \neq j'$. Then the list *jlist* is defined for $i \in I$, $0 \leq k \leq n$, and $1 \leq l \leq \min\{k, jlist\_max\}$ as follows, where *jlist_max* ($\leq n$) is a prespecified parameter:

$$jlist(i, k, l) = j \Leftrightarrow score(i, j) \text{ is the } l\text{th from the largest among the jobs in } \left\{j' \in J \mid a_{ij'} \leq a_i^{(k)}\right\}.$$

In other words, for a fixed pair i and k, $jlist(i, k, 1)$, $jlist(i, k, 2), \ldots$ is the list of jobs j, sorted in nonincreasing order of $score(i, j)$, whose sizes a_{ij} are not more than $a_i^{(k)}$. This list can be computed in $O(mn \cdot jlist\_max)$ time and space. Note that this computation is needed only when the Lagrangian multiplier vector $\tilde{v}$ is updated, as *jlist* is independent of σ, and the number of updates of $\tilde{v}$ is usually quite small, as discussed in §3.2.1. Therefore, the computation of *jlist* is not very expensive.

Now $B(j)$ is computed by the following algorithm, where Steps 1 through 4 compose the first phase to determine set J' of (7) and Steps 5 through 10 compose the second phase to compute B.

Algorithm INITIALIZE_B(σ).

Step 1. Let $j := 0$ and $J' := \varnothing$.

Step 2. If $j = n$, exit to Step 5; otherwise let $j := j + 1$ and $i := \sigma(j)$.

Step 3. Let k be the largest index for which $a_i^{(k)} \leq avail(j)$ holds.

Step 4. If $\{jlist(i,k,1), \ldots, jlist(i,k,\min\{k,jlist\_max\})\} \cap \{h \in J \mid \sigma(h) \neq i\} \neq \varnothing$ holds, then let $J' := J' \cup \{j\}$. Return to Step 2.

Step 5. Let $S := \varnothing$ and $B(j) := \varnothing$ for all $j \in J$.

Step 6. If $S = J'$, stop; otherwise choose $j \in J' \backslash S$, and let $S := S \cup \{j\}$ and $i := \sigma(j)$.

Step 7. Let k be the largest index for which $a_i^{(k)} \leq avail(j)$ holds.

Step 8. Let l be the minimum index such that $jlist(i, k, l) \in \{h \in J' \mid \sigma(h) \neq i\}$ holds. Then let $b\_score := score(i, jlist(i, k, l))$.

Step 9. If $l > \min\{k, jlist\_max\}$ or $score(i, jlist(i, k, l)) < b\_score$ holds, return to Step 6; otherwise proceed to Step 10.

Step 10. Let $h := jlist(i, k, l)$. If $h \in J'$ and $\sigma(h) \neq i$ holds, then let $B(j) := B(j) \cup \{h\}$. Let $l := l + 1$ and return to Step 9.

From the condition in Step 3, the condition of Step 4 becomes equivalent to the condition of J' in (7) (provided $jlist\_max \geq k$). Similarly, from the condition in Step 7, the *b_score* computed in Step 8 is equal to $best\_score(j)$ of (8). Thus the jobs with the score of $best\_score(j)$ are listed consecutively in *jlist* (e.g., $jlist(i, k, l-2) < jlist(i, k, l-1) = jlist(i, k, l) = jlist(i, k, l+1) = best\_score(j) < jlist(i, k, l+2)$). All such jobs are scanned in Steps 9 and 10, and appropriate job indices are added to $B(j)$.

Let us consider the time complexity of this algorithm. Steps 3 and 7 can be executed in $O(\log n)$ time by using binary search if the values a_{ij} ($j = 1, 2, \ldots, n$) are sorted beforehand for all $i \in I$ (which requires $O(mn \log n)$ time). Step 4 is executed by incrementing l from $l = 1$ until $jlist(i, k, l) \in \{h \in J \mid \sigma(h) \neq i\}$ or $l > \min\{k, jlist\_max\}$. The execution of Step 8 is similar. Let $l_{\max}^{(j)}$ be the value of $l - 1$ for the iteration of job j just before we return to Step 6 from Step 9. Then the time complexity of Step 4 and Steps 8 through 10 is $O(l_{\max}^{(j)})$. In the worst case, these steps may require time proportional to the number of jobs assigned to agent i, which is $O(n)$. However, our experiments show that this part of the computation is usually very small, as shown in the middle two columns of Table 1. We can also show that the expectation of $l_{\max}^{(j)}$ is $m/(m-1)$ if we assume $c_{ij} \neq c_{ij'}$ for all $j \neq j' \in J$, $J' = J$, and a random σ (see Appendix 3). Although the values in Table 1 are larger than this expectation, the average values for type D instances are constants around two to three and seem independent of their sizes. The values for types C and E instances are larger, but they do not increase much even for large n. In this sense, the above analysis provides intuitive support to the results in the table. As the computation time of algorithm INITIALIZE_B is $O(n \log n + \sum_{j=1}^{n} l_{\max}^{(j)})$, the worst-case time complexity is $O(n^2)$ and the expected time complexity is $O(n \log n)$.

Since J' of (7) and $B(j)$ of (9) may not be correctly computed if $jlist\_max < n$ holds, we set $jlist\_max = n$ in our experiments in §6. However, as is apparent from Table 1, these are usually correctly computed even if *jlist_max* is set much smaller than n. It is recommended to use a small value of *jlist_max* for large instances, to save memory required for *jlist*.

3.2.4. Double Shift Neighborhood. In this section, we explain that the double shift neighborhood can also be efficiently computed by using the data structure of §3.2.3. A move in this neighborhood is realized by two shift moves. Let *secondsh_max* ($\leq n$) be the parameter to specify the maximum number of second shift moves to consider. First, we eject a job j_0 from agent $i_0 = \sigma(j_0)$ (an ejection move) and compute the maximum k satisfying $a_{i_0}^{(k)} \leq avail(j_0)$. Then for each $l = 1, 2, \ldots, secondsh\_max$, we assign job $j_1 = jlist(i_0, k, l)$ to i_0 as the second shift move (i.e., ejection moves triggered by the ejection of j_0); job j_0 is then inserted into agent $\sigma(j_1)$ or into the agent with the minimum *pcost* (i.e., trial moves).

The algorithm to generate all solutions in the double shift neighborhood is formally described as follows.

Algorithm DOUBLE_SHIFT(σ).

Step 1. Let $j := 0$ and let π: $J \to J$ be a random permutation.

Step 2. If $j = n$, stop; otherwise let $j := j + 1$, $j_0 := \pi(j)$, and $i_0 := \sigma(j_0)$. (At this point, job j_0 is ejected from i_0.)

Step 3. Let i^* be the agent that minimizes $c_{ij_0} + \alpha_i \max\{0, (\sum_{j \in J,\, \sigma(j)=i} a_{ij}) + a_{ij_0} - b_i\}$ among all agents i except i_0. Let k be the largest value such that $a_{i_0}^{(k)} \leq avail(j_0)$. Let $l := 1$ and $counter := 1$.

Step 4. If $l > \min\{k, jlist\_max\}$ or $counter > secondsh\_max$, return to Step 2; otherwise let $j_1 := jlist(i_0, k, l)$, $i_1 := \sigma(j_1)$, and proceed to Step 5.

Step 5. If $i_1 = i_0$, let $l := l + 1$ and return to Step 4; otherwise proceed to Step 6.

Step 6. Let $\sigma' := \sigma$ and $\sigma'(j_1) := i_0$ (an ejection move of job j_1). Then execute the following Steps (a) and (b) (two trial moves).

(a) Let $\sigma'(j_0) := i_1$ (j_0 is inserted into i_1), and output σ'.

(b) Let $\sigma'(j_0) := i^*$ (j_0 is inserted into i^*), and output σ'.

Step 7. Let $l := l + 1$, $counter := counter + 1$ and return to Step 4.

Denote the set of solutions output in Step 6(a) (respectively, Step 6(b)) by $N_{\text{double\_cyc}}(\sigma)$ (respectively, $N_{\text{double\_path}}(\sigma)$), and let $N_{\text{double}}(\sigma) = N_{\text{double\_cyc}}(\sigma) \cup N_{\text{double\_path}}(\sigma)$. We call $N_{\text{double\_cyc}}$ a *cyclic double shift neighborhood*. Note that $N_{\text{double\_cyc}}(\sigma) \subseteq N_{\text{swap}}(\sigma)$ generally holds, and, if $secondsh\_max$ and $jlist\_max$ are set to n, then $N_{\text{double\_cyc}}(\sigma) = N_{\text{swap}}(\sigma)$ holds.

The time complexity of DOUBLE_SHIFT(σ) is $O(n^2)$ in the worst case; however, its practical running time is much smaller than $O(n^2)$ as the number of returns from Step 5 to Step 4 is usually not very large. Let $l_{\max}^{(j)}$ be the value of $l - 1$ just before we return to Step 2 from Step 4 in the iteration of job j. The right two columns of Table 1 show the average and the maximum of $l_{\max}^{(j)}$ and $l_{\max}^{(j)}/secondsh\_max$. The latter gives the ratio of the total number of scans to the number of necessary scans. We can also show that the expectation of $l_{\max}^{(j)}$ is $secondsh\_max \cdot m/(m-1)$ if we assume that $c_{ij} \neq c_{ij'}$ for all $j \neq j' \in J$ and solution σ is randomly generated (see Appendix 3). This analysis gives intuitive support for the results in Table 1.

Our experiments also show that $N_{\text{double}}(\sigma)$ is almost as powerful as $N_{\text{swap}}(\sigma)$, although we set $secondsh\_max = \max\{m, \log_2 n\}$. We set $secondsh\_max$ to this value because Step 3 of algorithm DOUBLE_SHIFT already requires $O(\max\{m, \log_2 n\})$ time. As a result, the size of $N_{\text{double}}(\sigma)$ becomes $O(n \max\{m, \log_2 n\})$ and the expected time complexity of DOUBLE_SHIFT is $O(n \max\{m, \log_2 n\})$, although its worst case time complexity is $O(n^2)$. As the time complexity $O(n^2)$ of scanning all solutions in $N_{\text{swap}}(\sigma)$ (both in the worst case and in practice) appears to be too large, we use $N_{\text{double}}(\sigma)$ instead of $N_{\text{swap}}(\sigma)$ in our algorithm TS.

3.2.5. EC Probe. The coordination of three neighborhoods, shift, double shift, and long chain, is called the *EC probe*. The solution is improved first by LS with shift neighborhood, second by LS with double shift neighborhood, and then by a long chain move, where the first admissible move strategy is used. The EC probe is repeated until no improvement is found. Note that we return to LS with shift neighborhood immediately whenever an improved solution is found in the long chain neighborhood, since the entire LS with long chain neighborhood is too expensive. (Note, however, that we check all solutions in $N_{\text{long}}(\sigma)$ if no improved solution is found.) The algorithm is formally described as follows, where σ is a given initial solution.

ALGORITHM EC_PROBE(σ).
Repeat the following Steps a, b, and c cyclically, until σ is no longer improved.

(a: Improve by shift neighborhood) Let $\sigma := \text{LS}(N_{\text{shift}}, pcost, \sigma)$.

(b: Improve by double shift neighborhood) Let $\sigma := \text{LS}(N_{\text{double}}, pcost, \sigma)$.

(c: Improve by a long chain) If a solution $\sigma' \in N_{\text{long}}(\sigma)$ with $pcost(\sigma') < pcost(\sigma)$ exists, then let $\sigma := \sigma'$.

The solution obtained by EC probe is locally optimal with respect to shift, double shift, and long chain neighborhoods.

4. Tabu Search

In this section, we explain how we construct the tabu search algorithm TS. The search starts from a randomly generated solution, which is then improved by EC probe. Then the best non-tabu shift (which is non-improving) is tried. After such a shift move, the solution is improved first by the cyclic double shift neighborhood $N_{\text{double\_cyc}}$ and then by EC probe. These steps are repeated until the computation time exceeds *timelim* (a prespecified parameter). The reason for the use of cyclic double shift neighborhood here is because it avoids a short cycling of returning to the solution before the shift move (since the number of jobs assigned to each agent does not change by a move in $N_{\text{double\_cyc}}$). This combination is motivated by the success of the procedure SSS probe (Yagiura et al. 1998), in which N_{shift} and N_{swap} are alternately used. To understand the idea of these approaches, here we quote a paragraph from Yagiura et al. (1998):

> Consider first the behavior of local search in which $N_{\text{shift}} \cup N_{\text{swap}}$ is always used as the neighborhood. In this case, if the current solution σ is close to locally optimal, shift moves rarely occur, since the agent i to which a job is shifted usually becomes infeasible and its penalty $p_i(\sigma)$ increases a large amount. As swap moves do not change the number of jobs assigned to each agent, this means that the number of jobs of each

agent is usually fixed at an early stage of the search and will never be changed until a locally optimal solution is reached. In other words, local search with neighborhood $N_{\text{shift}} \cup N_{\text{swap}}$ may not have enough search power, even if the size $|N_{\text{shift}} \cup N_{\text{swap}}|$ is fairly large.

The algorithm is formally described as follows. The set T represents the tabu list consisting of those solutions to which the moves are forbidden. The Lagrangian multiplier vector $\tilde{v}$ is used in *score* of (6) and affects the behavior of EC_PROBE. During the algorithm, the α_i parameters in penalty function (2) are adaptively controlled. An algorithm to control α_i will be described later in §5. In the TS algorithm, σ denotes the current solution; σ_{prev} the solution found in the previous iteration; σ^* the best feasible solution found during the search; and *best* the cost of σ^*.

Algorithm TS.

Step 1. Randomly generate an initial solution σ, let $\sigma_{\text{prev}} := \sigma$, $best := \infty$, $\tilde{v} := (0, 0, \ldots, 0)$, $chksg := 1$, $T := \varnothing$, and initialize *jlist*.

Step 2. Initialize parameters α_i for all $i \in I$.

Step 3. Let $\sigma :=$ EC_PROBE(σ) ($\tilde{v}$ and the α_i's are used in EC probe).

Step 4. If the computation time exceeds *timelim*, output σ^* and stop; otherwise proceed to Step 5.

Step 5. Update parameters α_i for all $i \in I$.

Step 6. If $pcost(\sigma) < pcost(\sigma_{\text{prev}})$ holds, let $\sigma_{\text{prev}} := \sigma$ and $T := \varnothing$; otherwise let $\sigma := \sigma_{\text{prev}}$. If $N_{\text{shift}}(\sigma) \setminus T = \varnothing$ holds, return to Step 1. Let σ' be the solution that minimizes *pcost* among $N_{\text{shift}}(\sigma) \setminus T$. Then let $T := T \cup \{\sigma'\}$ and $\sigma := \sigma'$.

Step 7. Let $\sigma :=$ LS($N_{\text{double\_cyc}}$, *pcost*, σ).

Step 8. If a solution $\sigma' \in N_{\text{long}}(\sigma)$ with $pcost(\sigma') < pcost(\sigma)$ exists, then let $\sigma := \sigma'$.

Step 9. If σ is updated in Steps 7 and 8, return to Step 3; otherwise return to Step 4.

Remark. Although not explicitly described in the above algorithm, we execute the following steps whenever a feasible solution is found (which is possible in Steps 1, 3, 6, 7, and 8): If the feasible solution σ satisfies $cost(\sigma) < best$, do the following (a) and (b).

(a) Let $best := cost(\sigma)$ and $\sigma^* := \sigma$ (i.e., σ^* keeps the best feasible solution found during the search).

(b) If $chksg = 0$ holds, skip this step. Let $v :=$ SUBGRADIENT($cost(\sigma^*)$, $\tilde{v}$). If $L(v) > L(\tilde{v})$ holds (L was defined in (4)), let $\tilde{v} := v$ and update *jlist*; otherwise let $chksg := 0$.

Since it takes time to check whether a solution is included in T if implemented naively, the tabu list T in Step 6 is implemented as follows. We use an $m \times n$ table Tstatus(i, j), whose initial values are set to 0 for all $i \in I$ and $j \in J$. Let *num_itr* be the number of calls of Step 6 during the search, and *prev_reset* be the value of *num_itr* when T is set empty most recently. For convenience, we define $i_{\sigma\to\sigma'}$ and $j_{\sigma\to\sigma'}$ for two solutions σ and σ' with $\sigma' \in N_{\text{shift}}(\sigma)$ as follows: $j_{\sigma\to\sigma'}$ is the job that is shifted and $i_{\sigma\to\sigma'}$ is the agent to which job $j_{\sigma\to\sigma'}$ is added in the move from σ to σ' (i.e., σ' is obtained from σ by shifting job $j_{\sigma\to\sigma'}$ to agent $i_{\sigma\to\sigma'}$). Whenever a solution σ' is added into T in Step 6, we set Tstatus($i_{\sigma\to\sigma'}$, $j_{\sigma\to\sigma'}$) := *num_itr*. Then a solution $\sigma' \in N_{\text{shift}}(\sigma)$ is in T if and only if Tstatus($i_{\sigma\to\sigma'}$, $j_{\sigma\to\sigma'}$) $\geq$ *prev_reset*. Thus we can check in O(1) time whether a solution is in T or not.

In Step 6, we return to σ_{prev} if $pcost(\sigma) > pcost(\sigma_{\text{prev}})$ holds to intensify the search from σ_{prev}. We also tested the following two strategies in this case of Step 6:

- The search always moves to the best non-tabu shift move (i.e., the search does not return to σ_{prev}).
- A random solution in $N_{\text{shift}}(\sigma)$ is tried instead of the solution with the minimum *pcost*.

However, the performance of Algorithm TS as described above seems to be slightly better than these two variants.

Note that the shifting operation that facilitates diversification in our proposed algorithm shares features in common with "kick" moves in iterated local search (ILS) (Johnson 1990; Martin et al. 1991, 1992). The use of such diversifying moves was proposed in tabu search (including reference to associated strategies involving concepts of "move distance" and "influence") somewhat before they were incorporated into so-called large step methods (see, e.g., Glover 1990), but there are papers in which algorithms that utilize this form of diversification are given the generic ILS label (e.g., Lourenço and Zwijnenburg 1996). An important differentiation between TS and ILS concerns not only the broader context given by TS for applying such ideas but also the inclusion of the strategic principles that make use of adaptive memory, as we have incorporated into our present approach to guide our ejection chain process.

5. Adaptive Control of the Penalty Weights

In this section, we explain the adaptive mechanism for controlling the penalty weights α_i in Steps 2 and 5 of algorithm TS. We incorporate this mechanism in TS because its performance highly depends on their parameter values. Throughout this section, we assume $b_i > 0$ for all $i \in I$.

Initialization. The initial α_i values in Step 2 of TS are determined by solving the following quadratic programming problem:

$$\text{(QP)} \quad \text{minimize} \quad \sum_{i\in I} y_i^2$$

$$\text{subject to} \quad y_i = \sum_{(i',j)\in S_i} (c_{ij} - c_{i'j} + \alpha_i a_{ij} - \alpha_{i'} a_{i'j}), \quad \forall i \in I$$

$$\alpha_i \geq 0, \quad \forall i \in I,$$

where set S_i is defined by

$$S_i = \{(i', j) \mid i' \in I,\ i' \neq i,\ j \in J \text{ and } a_{i'j}/b_{i'} < a_{ij}/b_i\}.$$

If $(i', j) \in S_i$, agent i' is more desirable than i with respect to the relative value of the resource requirement for job j. By solving (QP), we expect that the average increase in the cost for a shift move is balanced with the average decrease in the penalty. In our implementation, (QP) is solved by a simple Gauss-Seidel method (e.g., p. 247 of Bertsekas 1995), whose completion time was almost negligible (about five seconds even for the instances with $n = 1{,}600$ and $m = 80$, which are the largest among those tested) in our experiment. If the optimal solution of (QP) satisfies $\alpha_i = 0$ for all $i \in I$, we set $\alpha_i := \varepsilon$ for all $i \in I$ using a small positive ε.

Update. In Step 5 of algorithm TS, we update α_i by the following rule. Here we use functions

$$q_i^{\text{inc}}(\sigma) = p_i(\sigma)/b_i,$$

$$q_i^{\text{dec}}(\sigma) = \begin{cases} -1, & \text{if } p_i(\sigma) = 0 \\ 0, & \text{otherwise.} \end{cases}$$

Case 1 (if no feasible solution was found after the last execution of Step 2 or 5). The α_i values are increased for all $i \in I$ by

$$\alpha_i := \begin{cases} \alpha_i(1 + \Delta \cdot q_i^{\text{inc}}(\sigma)), & \alpha_i > 0 \\ \Delta \cdot q_i^{\text{inc}}(\sigma) \min_{h \in I}\{b_h \alpha_h \mid b_h \alpha_h > 0\}/b_i, & \text{otherwise,} \end{cases}$$

where

$$\Delta = \begin{cases} step\_size\_inc / \max_{i \in I} |q_i^{\text{inc}}(\sigma)|, \\ \quad \text{if } \max_{i \in I} |q_i^{\text{inc}}(\sigma)| > 0 \\ 0, \quad \text{otherwise} \end{cases}$$

($step\_size\_inc > 0$ is a prespecified parameter).

Case 2 (otherwise). All α_i are decreased. The rule to update α_i is the same as Case 1, except that $q^{\text{dec}}(\sigma)$ and $step\_size\_dec$ (a prespecified parameter satisfying $0 < step\_size\_dec < 1$) are used instead of $q^{\text{inc}}(\sigma)$ and $step\_size\_inc$.

The meaning of these rules is explained as follows. In the functions $q_i^{\text{inc}}(\sigma)$, the $p_i(\sigma)$ terms are divided by b_i to normalize among agents. If no feasible solution is found in the last iteration of EC probe (Case 1), we increase α_i by $\alpha_i \cdot \Delta \cdot q_i^{\text{inc}}(\sigma)$ for all $i \in I$ with $\alpha_i > 0$, where the $q_i^{\text{inc}}(\sigma)$ terms are the weights to emphasize overloaded agents, and Δ is used to specify the maximum amount of changes among agents to $step\_size\_inc$. For every $i \in I$ with $\alpha_i = 0$, α_i is set to a positive value that is small enough compared to other agents with $\alpha_i > 0$. The opposite case (Case 2) is similarly explained. In our preliminary experiments, we found that the performance of TS is robust with parameters $step\_size\_inc$ and $step\_size\_dec$. In §6, we use $step\_size\_inc = 0.01$ and $step\_size\_dec = 0.1$.

Remark. It is possible to set the initial values of α_i as constants (e.g., $\alpha_i := 1$ for all $i \in I$) and omit the solving of (QP). However, if the initial values of α_i are not appropriate, it may take a very long time until their appropriate values are attained, since their values are updated slowly. Note that such a situation can easily be constructed, essentially without changing the problem instance, by multiplying the values of a_{ij} and b_i by M_i for all $i \in I$ and $j \in J$, where M_i are large constants (e.g., $M_i = 1{,}000{,}000$). Then an appropriate value of α_i is $1/M_i$ times that of the original instance for each $i \in I$. Even in such a situation, it is observed in our experiment that the initial α_i values obtained by (QP) are quite reasonable, although they are usually slightly smaller than appropriate values (i.e., we cannot obtain feasible solutions before a few update steps are made).

6. Computational Results

In this section, algorithm TS is evaluated on the benchmark instances of §2. All the algorithms were coded in C and run on a Sun Ultra 2 Model 2300 workstation (two UltraSPARC II 300 MHz processors with 1 GB of memory), where the computation was executed on a single processor. In §6.1, TS is applied to small instances whose exact optimal values are known. In §6.3, TS is compared with an exact branch-and-bound algorithm proposed by Nauss (2003). We then compare TS with other existing heuristic algorithms in §6.4.

6.1. Results for Problem Set SMALL

When applied to instances in SMALL, algorithm TS obtained optimal solutions for all the tested instances within a few seconds, as shown in Table 2. There

Table 2 **Computation Time of TS in Seconds for SMALL Problem Instances (Exact Optimal Solutions Were Obtained in All Cases)**

			Time (sec)		
Type	n	m	Min.	Average	Max.
C	15	5	0.00	0.07	0.21
C	20	5	0.00	0.06	0.16
C	25	5	0.01	0.09	0.22
C	30	5	0.04	0.16	0.36
C	24	8	0.01	0.48	2.59
C	32	8	0.08	0.43	1.38
C	40	8	0.02	0.29	0.69
C	48	8	0.16	0.97	4.10
C	30	10	0.05	0.65	2.08
C	40	10	0.01	0.88	5.07
C	50	10	0.06	0.87	2.93
C	60	10	0.08	0.54	2.53

are five instances for each size, and each instance was solved five times using different random seeds. For each size, the minimum, average, and maximum among the results of 25 (=5 × 5) runs are shown.

From the table, we observe that the average computation time to reach the optimal solution is always less than one second. This suggests that these instances are too small to be used for comparing algorithms on current computers.

6.2. Speeds of Different Computers

Before presenting the computational results in §6.3 and §6.4, we give a rough comparison of the machines, Sun Ultra 2 Model 2300 (UltraSPARC II, 300 MHz), Dell XPS D300 (Pentium II, 300 MHz), Sun Ultra 1 Model 170E (UltraSPARC, 167 MHz), and Silicon Graphics Indigo (R4000, 100 MHz), on which the algorithms were run. Table 3 gives CPU, clock frequency, and benchmark values of SPECint95, SPECfp95, and Mflop/s, respectively. The values of SPECint95 and SPECfp95 are taken from the SPEC site (Standard Performance Evaluation Corporation, http://www.specbench.org/) (except for the data of SPECfp95 for the Dell XPS D300, which is in Nauss 2003), and Mflop/s are taken from Dongarra (1999). In the table, the row "TS run" shows the inverse of the average computation time needed for algorithm TS to obtain prespecified target solutions, where the time for the Sun Ultra 2 is normalized to one. For comparison purposes, we also show the data for a Gateway GP6-350 (Pentium II, 350 MHz). From the results in the row "TS run," we observe that SPECint95 gives more reliable information about the computation time of TS on different computers than does SPECfp95. Based on these, we give rough estimates on the speeds of algorithm TS on these computers in the row "estimate," where the speed of the Sun Ultra 2 is normalized to one, and a larger value means that it is faster.

6.3. Comparison with an Exact Algorithm

We now compare algorithm TS with the exact branch-and-bound algorithm developed by Nauss (2003) (denoted B&B) on problem instances of set MEDIUM. This exact method is known to be highly effective for problem instances in this size range and gives us an opportunity to see how close to optimality our TS approach is for instances of larger sizes. The results of B&B were provided by R. M. Nauss and run on a different computer, a Dell XPS D300 (Pentium II, 300 MHz).

Table 4 shows the minimum, average, and maximum cost; the number of runs for which the best solution was found; and the average computation time to find the best solution (this average was taken for those runs in which the best cost was found) among five runs of algorithm TS. For each run of TS, the parameter *timelim* was set to 3,000 (respectively, 6,000) seconds for instances with $n = 100$ (respectively, 200). Algorithm B&B was applied three times for each instance, where the time limit of each run was set to 1,200 seconds, and the best incumbent value of the previous run was used as the initial upper bound in the second and third runs. (This strategy was selected by Nauss as an effective one for the B&B procedure.) The best cost, the total computation time to find the best solution, the total computation time until the algorithm stops by confirming optimality, and whether the optimality was confirmed or not are shown in the table. The mark "∗" indicates the best cost values attained by these two algorithms. We also show the lower bound (denoted LB) obtained by solving the Lagrangian dual problem (i.e., maximize $L(v)$ of (4)), using the subgradient method, in which the variables x_{ij} are constrained to be 0 or 1. (Note that the values of $L(v)$ are usually non-integer, and they are rounded up in the table.) By the integrality constraints, problem (4) becomes m 0-1 knapsack problems, which are known to be NP-hard; however, the obtained lower bound becomes usually better than the one obtained from the LP relaxation of problem (1). (We used a simple dynamic programming algorithm to solve the 0-1 knapsack problems to compute the LB in Tables 4, 5, and 7–9.) Note that the values in column LB are different from (usually much larger than) those obtained in the subgradient phase of algorithm TS, since the relaxation problem (4) used in TS is actually an LP

Table 3 Comparison of Different Computers

	Sun Ultra 2 Model 2300	Dell XPS D300	Sun Ultra 1 Model 170E	Silicon Graphics Indigo	Gateway GP6-350
CPU	UltraSPARC II	Pentium II	UltraSPARC	R4000	Pentium II
Clock	300 MHz	300 MHz	167 MHz	100 MHz	350 MHz
SPECint95	12.3	11.9∗	6.26		13.4–13.9∗
SPECfp95	20.2	8.15	9.06		10.8–11.2∗
Mflop/s			70	15	
TS run	1		0.55		1.37
Estimate	1	1	0.55	0.1	1.37

Note. Data with "∗" are those for the Intel SE440BX motherboard.

Table 4 Comparison with the Branch-and-Bound Method of Nauss (2003)

				TS (5 runs)					B&B			
				Cost			No. of	Avg. time		Time to	Time to	
Type	n	m	LB	Min.	Avg.	Max.	best	to best	Cost	best	confirm	Opt?
C	100	5	1,930	1,931*	1,931.0	1,931	5	0.6	1,931*	0.3	7.1	Yes
C	100	10	1,400	1,402*	1,402.0	1,402	5	3.0	1,402*	15.5	33.2	Yes
C	100	20	1,242	1,243*	1,243.0	1,243	5	22.5	1,243*	39.7	69.5	Yes
C	200	5	3,455	3,456*	3,456.0	3,456	5	3.7	3,456*	36.4	45.8	Yes
C	200	10	2,804	2,806*	2,806.0	2,806	5	403.8	2,806*	631.1	1,081.0	Yes
C	200	20	2,391	2,391*	2,391.0	2,391	5	301.8	2,392	926.0	—	No
D	100	5	6,350	6,353*	6,353.0	6,353	5	649.2	6,353*	170.7	174.7	Yes
D	100	10	6,342	6,349*	6,351.8	6,354	1	2,440.7	6,349*	1,023.7	—	No
D	100	20	6,177	6,206	6,210.6	6,214	1	1,591.9	6,196*	2,122.9	—	No
D	200	5	12,741	12,743*	12,743.2	12,744	4	3,564.8	12,745	102.1	—	No
D	200	10	12,426	12,440	12,441.6	12,443	1	5,829.9	12,436*	1,309.9	—	No
D	200	20	12,230	12,277	12,278.6	12,281	3	1,757.7	12,264*	3,464.4	—	No
E	100	5	12,673	12,681*	12,681.0	12,681	5	97.3	12,681*	44.8	46.1	Yes
E	100	10	11,568	11,577*	11,577.0	11,577	5	31.5	11,577*	58.3	125.4	Yes
E	100	20	8,431	8,436*	8,438.4	8,439	1	1,144.2	8,436*	1,660.5	1,770.1	Yes
E	200	5	24,927	24,930*	24,930.0	24,930	5	20.0	24,930*	61.5	69.5	Yes
E	200	10	23,302	23,307*	23,307.0	23,307	5	866.8	23,307*	357.6	990.2	Yes
E	200	20	22,377	22,379*	22,379.0	22,379	5	627.8	22,379*	988.2	1,126.0	Yes

Note. Algorithm TS was run on a Sun Ultra 2 Model 2300 with time limits of 3,000 and 6,000 seconds for $n = 100$ and 200, respectively. Algorithm B&B was run on a Dell XPS D300 with time limit of 3,600 ($=1{,}200 \times 3$) seconds.

relaxation of problem (1). (As the values of LB are very close to optimal, readers may expect that some optimal solutions (e.g., those in class SMALL and type C with $n = 200$ and $m = 20$) were actually found during the subgradient phase. However, this was not the case, i.e., all optimal solutions reported in this paper were found during the local search phase.)

For most of types C and E instances, both algorithms obtained the optimal solutions within a reasonable amount of time. (For the single problem from

Table 5 The Best Costs Obtained by the Tested Algorithms

Type	n	m	LB	TS	MLS	BVDS-I	BVDS-j	YYI	RA	LKGG	NI	RLS♭	CB♮
C	100	5	1,930	1,931*	1,931*	1,931*	1,931*	1,931*	1,938	1,931*	1,931*	1,942	1,931*
C	100	10	1,400	1,402*	1,408	1,402*	1,403	1,402*	1,405	1,403	1,403	1,407	1,403
C	100	20	1,242	1,243*	1,263	1,244	1,244	1,246	1,250	1,245	1,245	1,247	1,244
C	200	5	3,455	3,456*	3,460	3,456*	3,457	3,457	3,469	3,457	3,465	3,467	3,458
C	200	10	2,804	2,806*	2,835	2,809	2,808	2,809	2,835	2,812	2,817	2,818	2,814
C	200	20	2,391	2,392*	2,434	2,401	2,400	2,405	2,419	2,396	2,407	2,405	2,397
D	100	5	6,350	6,357*	6,399	6,358	6,362	6,365	—	6,386	6,415	6,476	6,373
D	100	10	6,342	6,358*	6,489	6,367	6,370	6,380	6,532	6,406	6,487	6,469	6,379
D	100	20	6,177	6,221*	6,414	6,275	6,245	6,284	6,428	6,297	6,368	6,358	6,269
D	200	5	12,741	12,746*	12,847	12,755	12,755	12,778	—	12,788	12,973	12,923	12,796
D	200	10	12,426	12,446*	12,679	12,480	12,473	12,496†	12,799	12,537	12,889	12,746	12,601
D	200	20	12,230	12,284*	12,602	12,440	12,318	12,335†	12,665	12,436	12,793	12,617	12,452
E	100	5	12,673	12,682	12,722	12,681*	12,682	12,685	12,917	12,687††	12,686‡	12,836	NA
E	100	10	11,568	11,577*	11,700	11,585	11,599	11,585	12,047	11,641††	11,590‡	11,780	NA
E	100	20	8,431	8,443*	8,845	8,499	8,484	8,490	9,004	8,522††	8,509‡	8,717	NA
E	200	5	24,927	24,930*	24,969	24,942	24,933	24,948	25,649	25,147††	24,958‡	25,317	NA
E	200	10	23,302	23,307*	23,503	23,346	23,348	23,340	24,717	23,567††	23,396‡	23,620	NA
E	200	20	22,377	22,391*	22,844	22,475	22,437	22,452†	24,117	22,659††	22,551‡	22,779	NA

Note. The meaning of the marks *, †, ††, ‡, ♭, and ♮ are explained as follows:
*The best cost among the tested algorithms.
†Results after 1,000 seconds on Sun Ultra 2 Model 2300 (300 MHz).
††Results after 20,000 seconds on Sun Ultra 2 Model 2300 (300 MHz).
‡Results after 5,000 seconds on Sun Ultra 2 Model 2300 (300 MHz).
♭Results on Sun Ultra 1 Model 170E, whose computation time is in Table 6.
♮Results in Chu and Beasley (1997), whose computation time is in Table 6.

Table 6 The Computation Time in Seconds of the Algorithms TS, CB, and RLS

			TS (Sun Ultra 2)		CB (Silicon Graphics Indigo)		RLS (Sun Ultra 1)	
Type	n	m	Time to best	Time limit	Time to best	Total time	Time to best	Total time
C	100	5	0.5	150.0	139.1	302.4	137.1	358.5
C	100	10	2.6	150.0	170.6	394.2	178.3	384.7
C	100	20	3.7	150.0	279.9	669.3	309.6	703.3
C	200	5	0.9	300.0	531.2	810.1	1,693.8	2,088.2
C	200	10	30.0	300.0	628.6	1,046.0	1,086.0	2,331.3
C	200	20	16.7	300.0	1,095.9	1,792.3	2,694.8	3,332.9
D	100	5	39.6	150.0	369.9	530.3	2,459.2	2,542.9
D	100	10	67.0	150.0	870.2	1,094.7	5,587.3	5,986.7
D	100	20	8.5	150.0	1,746.1	2,126.1	13,656.8	14,125.3
D	200	5	116.0	300.0	1,665.9	1,942.8	11,106.9	21,952.1
D	200	10	63.7	300.0	2,768.7	3,189.6	47,538.7	52,763.2
D	200	20	162.1	300.0	4,878.4	5,565.1	116,969.0	116,969.0
E	100	5	5.0	150.0	NA	NA	1,358.1	1,358.1
E	100	10	39.8	150.0	NA	NA	2,908.7	3,332.6
E	100	20	125.0	150.0	NA	NA	3,450.8	5,516.9
E	200	5	17.3	300.0	NA	NA	5,319.0	12,342.6
E	200	10	16.2	300.0	NA	NA	21,253.9	30,865.6
E	200	20	297.5	300.0	NA	NA	59,920.8	68,442.0

Note. Computers on which algorithms TS, CB, and RLS were run are named in parentheses in table boxheads. NA = not available.

Table 7 Best Costs of the Tested Algorithms for Larger Instances

Type	n	m	LB	TS	MLS	BVDS-I	LKGG	NI
C	400	10	5,596	5,597*	5,645	5,605	5,608	5,613
C	400	20	4,781	4,782*	4,868	4,795	4,792	4,793
C	400	40	4,244	4,244*	4,327	4,259	4,251	4,247
C	900	15	11,339	11,341*	11,459	11,368	11,362	11,365
C	900	30	9,982	9,985*	10,187	10,022	10,007	10,000
C	900	60	9,325	9,328*	9,533	9,386	9,341	9,340
C	1,600	20	18,802	18,803*	19,011	18,892	18,831	18,822
C	1,600	40	17,144	17,147*	17,457	17,262	17,170	17,165
C	1,600	80	16,284	16,291*	16,594	16,380	16,303	16,301
D	400	10	24,959	24,974*	25,330	25,032	25,145	26,004
D	400	20	24,561	24,614*	25,165	24,780	24,872	25,496
D	400	40	24,350	24,463*	—	24,724	24,726	25,405
D	900	15	55,403	55,435*	56,277	55,614	56,423	57,504
D	900	30	54,833	54,910*	56,101	55,210	55,918	57,630
D	900	60	54,551	54,666*	—	55,123	55,379	56,699
D	1,600	20	97,823	97,870*	99,358	98,248	100,171	101,122
D	1,600	40	97,105	97,177*	99,114	97,721	99,290	100,574
D	1,600	80	97,034	97,109*	—	98,146	98,439	100,471
E	400	10	45,745	45,746*	46,163	45,878	172,185	46,243
E	400	20	44,876	44,882*	45,628	45,079	137,153	45,492
E	400	40	44,557	44,589*	45,673	44,898	63,669	45,574
E	900	15	102,420	102,423*	103,512	102,755	463,142	104,932
E	900	30	100,426	100,442*	102,200	100,956	527,451	104,173
E	900	60	100,144	100,185*	102,395	100,917	479,650	105,494
E	1,600	20	180,642	180,647*	182,590	181,143	936,609	187,207
E	1,600	40	178,293	178,311*	181,029	179,036	1,026,259	187,451
E	1,600	80	176,816	176,866*	180,744	178,205	1,026,417	189,774

Note. Time limits were 3,000, 10,000, and 50,000 seconds for $n = 400$, 900, and 1,600, respectively.

Table 8 Detailed Results of Algorithm TS

Type	n	m	LB	Cost (error from LB) of 5 runs			No. of best found	Average time to best	Time limit
				Minimum (%)	Average (%)	Maximum (%)			
C	100	5	1,930	1,931 (0.052)	1,931.0 (0.052)	1,931 (0.052)	5/5	0.6	150
C	100	10	1,400	1,402 (0.143)	1,402.0 (0.143)	1,402 (0.143)	5/5	3.0	150
C	100	20	1,242	1,243 (0.081)	1,243.0 (0.081)	1,243 (0.081)	5/5	21.6	150
C	200	5	3,455	3,456 (0.029)	3,456.0 (0.029)	3,456 (0.029)	5/5	3.7	300
C	200	10	2,804	2,806 (0.071)	2,806.2 (0.078)	2,807 (0.107)	4/5	100.5	300
C	200	20	2,391	2,391 (0.000)	2,391.6 (0.025)	2,392 (0.042)	2/5	137.4	300
C	400	10	5,596	5,597 (0.018)	5,597.0 (0.018)	5,597 (0.018)	5/5	105.8	300
C	400	20	4,781	4,783 (0.042)	4,783.0 (0.042)	4,783 (0.042)	5/5	130.4	300
C	400	40	4,244	4,245 (0.024)	4,245.0 (0.024)	4,245 (0.024)	5/5	157.6	300
C	900	15	11,339	11,340 (0.009)	11,341.4 (0.021)	11,342 (0.026)	1/5	759.6	1,000
C	900	30	9,982	9,986 (0.040)	9,986.2 (0.042)	9,987 (0.050)	4/5	720.0	1,000
C	900	60	9,325	9,330 (0.054)	9,330.4 (0.058)	9,331 (0.064)	3/5	704.7	1,000
C	1,600	20	18,802	18,803 (0.005)	18,804.0 (0.011)	18,805 (0.016)	1/5	691.6	5,000
C	1,600	40	17,144	17,148 (0.023)	17,148.6 (0.027)	17,149 (0.029)	2/5	2,103.7	5,000
C	1,600	80	16,284	16,295 (0.068)	16,299.4 (0.095)	16,304 (0.123)	1/5	4,317.2	5,000
D	100	5	6,350	6,354 (0.063)	6,355.6 (0.088)	6,357 (0.110)	1/5	62.6	150
D	100	10	6,342	6,356 (0.221)	6,359.6 (0.278)	6,364 (0.347)	1/5	107.2	150
D	100	20	6,177	6,215 (0.615)	6,220.0 (0.696)	6,226 (0.793)	1/5	111.0	150
D	200	5	12,741	12,744 (0.024)	12,745.6 (0.036)	12,747 (0.047)	1/5	95.5	300
D	200	10	12,426	12,445 (0.153)	12,445.4 (0.156)	12,446 (0.161)	3/5	129.2	300
D	200	20	12,230	12,277 (0.384)	12,284.4 (0.445)	12,289 (0.482)	1/5	120.7	300
D	400	10	24,959	24,976 (0.068)	24,978.2 (0.077)	24,980 (0.084)	1/5	16.1	300
D	400	20	24,561	24,604 (0.175)	24,609.2 (0.196)	24,617 (0.228)	1/5	81.3	300
D	400	40	24,350	24,460 (0.452)	24,466.6 (0.479)	24,482 (0.542)	1/5	165.2	300
D	900	15	55,403	55,425 (0.040)	55,436.2 (0.060)	55,443 (0.072)	1/5	112.9	1,000
D	900	30	54,833	54,903 (0.128)	54,908.8 (0.138)	54,912 (0.144)	1/5	234.4	1,000
D	900	60	54,551	54,673 (0.224)	54,677.6 (0.232)	54,682 (0.240)	1/5	833.8	1,000
D	1,600	20	97,823	97,867 (0.045)	97,875.4 (0.054)	97,884 (0.062)	1/5	143.0	5,000
D	1,600	40	97,105	97,160 (0.057)	97,166.0 (0.063)	97,177 (0.074)	1/5	1,294.0	5,000
D	1,600	80	97,034	97,098 (0.066)	97,105.2 (0.073)	97,110 (0.078)	1/5	4,795.4	5,000
E	100	5	12,673	12,681 (0.063)	12,681.4 (0.066)	12,682 (0.071)	3/5	39.9	150
E	100	10	11,568	11,577 (0.078)	11,577.0 (0.078)	11,577 (0.078)	5/5	31.6	150
E	100	20	8,431	8,439 (0.095)	8,440.6 (0.114)	8,443 (0.142)	2/5	90.4	150
E	200	5	24,927	24,930 (0.012)	24,930.0 (0.012)	24,930 (0.012)	5/5	20.0	300
E	200	10	23,302	23,307 (0.021)	23,308.0 (0.026)	23,310 (0.034)	2/5	34.1	300
E	200	20	22,377	22,379 (0.009)	22,384.0 (0.031)	22,391 (0.063)	2/5	209.3	300
E	400	10	45,745	45,746 (0.002)	45,747.2 (0.005)	45,749 (0.009)	1/5	260.7	300
E	400	20	44,876	44,887 (0.025)	44,889.6 (0.030)	44,892 (0.036)	2/5	212.9	300
E	400	40	44,557	44,596 (0.088)	44,608.0 (0.114)	44,617 (0.135)	1/5	217.7	300
E	900	15	102,420	102,424 (0.004)	102,425.0 (0.005)	102,426 (0.006)	1/5	157.4	1,000
E	900	30	100,426	100,438 (0.012)	100,445.4 (0.019)	100,450 (0.024)	1/5	758.9	1,000
E	900	60	100,144	100,197 (0.053)	100,207.6 (0.064)	100,213 (0.069)	1/5	483.7	1,000
E	1,600	20	180,642	180,648 (0.003)	180,649.0 (0.004)	180,650 (0.004)	1/5	1,683.2	5,000
E	1,600	40	178,293	178,313 (0.011)	178,319.4 (0.015)	178,323 (0.017)	1/5	2,214.7	5,000
E	1,600	80	176,816	176,878 (0.035)	176,889.8 (0.042)	176,900 (0.048)	1/5	2,473.1	5,000

types C and E, for which B&B failed to confirm optimality, TS obtained a slightly better solution.) Thus, these outcomes indicate that algorithm TS, while not exact, is highly successful in finding optimal solutions. Type D instances seem harder for both algorithms. B&B could not verify optimality for most of these instances but lived up to its reputation for being highly effective by finding slightly better solutions than TS for three instances, while TS found a slightly better solution on one instance. Again, the very similar quality of the solutions obtained by both methods suggests that TS obtains near optimal solutions for these types of problem instances. Accordingly, we turned to the examination of substantially larger problems. These were beyond the scope of B&B to handle, and we therefore conducted our remaining tests by comparing with other heuristic methods. (To our knowledge, the code of B&B by Nauss is not applicable to instances with $n > 300$.)

Table 9 Detailed Results of Algorithm TS

Type	n	m	LB	Cost (error from LB) of 5 runs			No. of best found	Average time to best	Time limit
				Minimum (%)	Average (%)	Maximum (%)			
C	100	5	1,930	1,931 (0.052)	1,931.0 (0.052)	1,931 (0.052)	5/5	0.6	3,000
C	100	10	1,400	1,402 (0.143)	1,402.0 (0.143)	1,402 (0.143)	5/5	3.0	3,000
C	100	20	1,242	1,243 (0.081)	1,243.0 (0.081)	1,243 (0.081)	5/5	22.5	3,000
C	200	5	3,455	3,456 (0.029)	3,456.0 (0.029)	3,456 (0.029)	5/5	3.7	6,000
C	200	10	2,804	2,806 (0.071)	2,806.0 (0.071)	2,806 (0.071)	5/5	403.8	6,000
C	200	20	2,391	2,391 (0.000)	2,391.0 (0.000)	2,391 (0.000)	5/5	301.8	6,000
C	400	10	5,596	5,597 (0.018)	5,597.0 (0.018)	5,597 (0.018)	5/5	103.4	3,000
C	400	20	4,781	4,782 (0.021)	4,782.4 (0.029)	4,783 (0.042)	3/5	1,956.3	3,000
C	400	40	4,244	4,244 (0.000)	4,244.6 (0.014)	4,245 (0.024)	2/5	1,602.0	3,000
C	900	15	11,339	11,340 (0.009)	11,340.4 (0.012)	11,341 (0.018)	3/5	5,837.2	10,000
C	900	30	9,982	9,984 (0.020)	9,984.6 (0.026)	9,985 (0.030)	2/5	5,186.7	10,000
C	900	60	9,325	9,328 (0.032)	9,329.0 (0.043)	9,330 (0.054)	1/5	9,259.0	10,000
C	1,600	20	18,802	18,803 (0.005)	18,803.2 (0.006)	18,804 (0.011)	4/5	19,484.3	50,000
C	1,600	40	17,144	17,147 (0.017)	17,147.4 (0.020)	17,148 (0.023)	3/5	19,102.4	50,000
C	1,600	80	16,284	16,291 (0.043)	16,292.4 (0.052)	16,294 (0.061)	2/5	25,493.9	50,000
D	100	5	6,350	6,353 (0.047)	6,353.0 (0.047)	6,353 (0.047)	5/5	649.2	3,000
D	100	10	6,342	6,349 (0.110)	6,351.8 (0.155)	6,354 (0.189)	1/5	2,440.7	3,000
D	100	20	6,177	6,206 (0.469)	6,210.6 (0.544)	6,214 (0.599)	1/5	1,591.9	3,000
D	200	5	12,741	12,743 (0.016)	12,743.2 (0.017)	12,744 (0.024)	4/5	3,564.8	6,000
D	200	10	12,426	12,440 (0.113)	12,441.6 (0.126)	12,443 (0.137)	1/5	5,829.9	6,000
D	200	20	12,230	12,277 (0.384)	12,278.6 (0.397)	12,281 (0.417)	3/5	1,757.7	6,000
D	400	10	24,959	24,974 (0.060)	24,976.4 (0.070)	24,979 (0.080)	1/5	2,611.8	3,000
D	400	20	24,561	24,604 (0.175)	24,609.0 (0.195)	24,616 (0.224)	1/5	81.6	3,000
D	400	40	24,350	24,456 (0.435)	24,461.2 (0.457)	24,464 (0.468)	1/5	1,385.9	3,000
D	900	15	55,403	55,425 (0.040)	55,433.4 (0.055)	55,436 (0.060)	1/5	114.2	10,000
D	900	30	54,833	54,903 (0.128)	54,908.8 (0.138)	54,912 (0.144)	1/5	241.0	10,000
D	900	60	54,551	54,656 (0.192)	54,666.6 (0.212)	54,680 (0.236)	1/5	1,844.0	10,000
D	1,600	20	97,823	97,867 (0.045)	97,872.4 (0.050)	97,877 (0.055)	1/5	153.8	50,000
D	1,600	40	97,105	97,160 (0.057)	97,166.0 (0.063)	97,177 (0.074)	1/5	1,300.7	50,000
D	1,600	80	97,034	97,097 (0.065)	97,103.0 (0.071)	97,110 (0.078)	2/5	6,002.2	50,000
E	100	5	12,673	12,681 (0.063)	12,681.0 (0.063)	12,681 (0.063)	5/5	97.3	3,000
E	100	10	11,568	11,577 (0.078)	11,577.0 (0.078)	11,577 (0.078)	5/5	31.5	3,000
E	100	20	8,431	8,436 (0.059)	8,438.4 (0.088)	8,439 (0.095)	1/5	1,144.2	3,000
E	200	5	24,927	24,930 (0.012)	24,930.0 (0.012)	24,930 (0.012)	5/5	20.0	6,000
E	200	10	23,302	23,307 (0.021)	23,307.0 (0.021)	23,307 (0.021)	5/5	866.8	6,000
E	200	20	22,377	22,379 (0.009)	22,379.0 (0.009)	22,379 (0.009)	5/5	627.8	6,000
E	400	10	45,745	45,746 (0.002)	45,746.0 (0.002)	45,746 (0.002)	5/5	863.6	3,000
E	400	20	44,876	44,882 (0.013)	44,883.4 (0.016)	44,885 (0.020)	1/5	2,665.3	3,000
E	400	40	44,557	44,579 (0.049)	44,584.6 (0.062)	44,589 (0.072)	1/5	2,602.0	3,000
E	900	15	102,420	102,422 (0.002)	102,423.0 (0.003)	102,424 (0.004)	1/5	4,701.0	10,000
E	900	30	100,426	100,438 (0.012)	100,440.6 (0.015)	100,443 (0.017)	2/5	5,255.6	10,000
E	900	60	100,144	100,177 (0.033)	100,181.2 (0.037)	100,185 (0.041)	1/5	8,139.7	10,000
E	1,600	20	180,642	180,647 (0.003)	180,647.4 (0.003)	180,648 (0.003)	3/5	19,142.6	50,000
E	1,600	40	178,293	178,311 (0.010)	178,313.2 (0.011)	178,316 (0.013)	2/5	35,026.0	50,000
E	1,600	80	176,816	176,856 (0.023)	176,862.8 (0.026)	176,869 (0.030)	1/5	49,790.3	50,000

6.4. Comparison with Other Heuristic Algorithms

Algorithm TS was compared with eight heuristic algorithms: (1) Random multistart local search (denoted MLS) (Yagiura et al. 1998, 1999); (2) two algorithms of branching variable depth search by Yagiura et al. (1998) (denoted BVDS-l and BVDS-j); (3) variable depth search by Yagiura et al. (1999) (denoted VDS); (4) variable depth search by Racer and Amini (1994) (denoted RA); (5) tabu search by Laguna et al. (1995) (denoted LKGG); (6) tabu search for the general purpose constraint satisfaction problem by Nonobe and Ibaraki (1998) (denoted NI); (7) a MAX-MIN ant system combined with local search and tabu search by Lourenço and Serra (2002) (denoted RLS). MLS, BVDS-l, BVDS-j, VDS, and RA were coded in the C language by us, while the codes of LKGG, NI, and RLS were sent from the authors. The codes of LKGG and NI are written in C, and that of RLS is written in FORTRAN 77. The parameters for BVDS-l, BVDS-j, and VDS are set to the values reported in Yagiura

et al. (1998). For MLS, the neighborhood $N_{\text{shift}} \cup N_{\text{swap}}$ is used. RA does not include any parameter. The parameters for LKGG and NI are set to the default values. The RLS codes include various types of algorithms, which can be combined by choosing appropriate options. Here we chose the option ASH + LS + TS as recommended in Lourenço and Serra (2002), and other parameters were set to the default values.

All algorithms were first applied to MEDIUM instances. Table 5 shows the best costs obtained by these algorithms within 150 seconds for $n = 100$ and 300 seconds for $n = 200$, respectively, unless otherwise stated. For algorithms VDS, LKGG, and NI, this time limit appears to be too short for some instances. In such cases, we allowed more computation time, as stated in the notes below the table. As algorithm RLS does not have the option to set a time limit, we ran RLS until it stopped. Its computation time is also shown in Table 6, where it was run on Sun Ultra 1 Model 170E. For comparison purposes, we also include in Table 5 the results of the genetic algorithm by Chu and Beasley (1997) (denoted CB), where the computation time, reported in Chu and Beasley (1997), is on a different workstation, a Silicon Graphics Indigo (R4000, 100 MHz). The results of CB for type E instances are not available and are denoted "NA" in the table. The computation time of CB is shown in Table 6. In Table 5, we also show the lower bounds (denoted LB) as in Table 4. In the table, each "$*$" mark represents that the best cost among those obtained by the tested algorithms is attained, and "—" means that no feasible solution was found.

Table 7 shows similar results for larger instances from set LARGE, where the time limits are set to 3,000 seconds for $n = 400$, 10,000 seconds for $n = 900$, and 50,000 seconds for $n = 1{,}600$. For these instances, we only tested algorithms BVDS-l, MLS, LKGG, and NI, since the results of BVDS-j and YYI are similar to those for BVDS-l, and RA and RLS are not competitive with the results of Table 5.

From Tables 5 and 7, we can observe the following:

1. Our algorithm TS obtained the best costs for most instances among all the tested algorithms; in particular, for all instances of larger sizes.

2. The differences between the best cost and LB are at most 1% for all the tested instances, which implies that the solution quality of TS is quite high.

We also show detailed results of the proposed TS in Tables 8 and 9 with different time limits to encourage future research. In the tables, the figures in parentheses are errors in % from LB, and "No. of Best Found" is the number of runs out of five in which the best cost is attained.

7. Conclusion

We have proposed a tabu search algorithm TS for the generalized assignment problem. It includes an ejection chain neighborhood, Lagrangian relative costs to reduce the neighborhood size, and a mechanism for adaptively controlling the balance between the feasible and infeasible regions visited by the search. Computational comparison of the proposed method with an exact algorithm shows that our TS obtains optimal or near optimal solutions for instances of small and medium sizes. In addition, comparisons with other existing heuristic algorithms on benchmark instances whose sizes are too large to handle by the exact method show that TS obtains solutions that are better than those obtained by all the other methods.

Appendix 1. Comments on Functions *avail* and *score*

(Note: We recommend reading Appendix 1 after reading §3.2.2–§3.2.4.) For the function *avail* of (5), we also tested the following two functions:

$$a_{\sigma(j),j}, \tag{A.1}$$

$$\max\{0, a_{\sigma(j),j} - p_{\sigma(j)}(\sigma)\}. \tag{A.2}$$

In function (A.1), all jobs whose resource requirement is not more than that of the ejected job become the candidates to be inserted, while in function (A.2), *avail(j)* represents the amount of resource actually available at agent $\sigma(j)$. The function (A.2) is better than (A.1) in most cases; however, if σ is far from feasible, i.e., $p_i(\sigma)$ is very large for all i, $N_{\text{long}}(\sigma) = \varnothing$ and $N_{\text{double}}(\sigma) = \varnothing$ may result. To avoid this, we combine (A.1) and (A.2), and adopt (5) in our algorithm TS.

For the function *score*, we also tested the following five candidates:

$$-c_{ij}, \tag{A.3}$$

$$-c_{ij} + c_{\sigma(j),j}, \tag{A.4}$$

$$-c_{ij} + \max_{i' \in I} c_{i'j}, \tag{A.5}$$

$$-c_{ij} + \sum_{i' \in I} c_{i'j}/m, \tag{A.6}$$

$$-c_{ij} + c_{\sigma^*(j),j}.$$

Function (A.4) seems to be more meaningful than others, since the decrease in the cost for agent $\sigma(j)$ from which job j is ejected is also taken into account. However, the speedup technique in §3.2.3 is not applicable to function (A.4), since $B(j)$ depends on σ in this case. In our experiments, functions (6) and (A.3) appear to be slightly better than others. Between functions (6) and (A.3), function (A.3) may not work properly if there are some jobs whose costs are much smaller than others for all agents, because $B(j)$ for many j's are occupied by such jobs. (For example, if we add a dummy job $\tilde{j}$ with $c_{i\tilde{j}} = a_{i\tilde{j}} = 0$ for all $i \in I$. Then $B(j) = \{\tilde{j}\}$ holds for all $j \in J$, and algorithm LONG_CHAIN will not work. Functions (A.5) and (A.6)

would work in this specific case; however, they will not work if the dummy job $\tilde{j}$ has a very large assignment cost for only one agent. Algorithm DOUBLE_SHIFT will have a similar problem if many such dummy jobs exist.) Since such irregularity is overcome to some extent by function (6), we adopted (6).

Appendix 2. Expected Length of a Long Chain Move

As argued in §3.2.3, the length of long chain moves has a crucial effect on the performance of algorithm LONG_CHAIN. We derive here the expected length of long chain moves under the following simple random model: $B(j)$ of (9) for each j consists of a job randomly chosen from J. Let X be the random variable that represents the length of an ejection chain. Then, we can show the following:

THEOREM A.1. *Let X be defined as above. Then for any small constant $\varepsilon > 0$, $\mathrm{E}(X) \le n^{(1/2)+\varepsilon}$ holds for sufficiently large n.*

To prove this theorem, we use the following lemma.

LEMMA A.1. *Function $f(n,l) = (n/(n-l))^{n-l}\mathrm{e}^{-l}$ satisfies $f(n,\ n^{(1/2)+\varepsilon}) \le \mathrm{e}^{-n^{2\varepsilon}/2}$ for any $\varepsilon \in (0, 1/2)$, if $n \ge 3^{2/(1-2\varepsilon)}$ holds.*

PROOF OF LEMMA A.1. First, we claim that $\ln(1+x) \le x - x^2/2 + x^3/3$ holds for all $x \ge 0$. In fact, let

$$g(x) = x - \tfrac{1}{2}x^2 + \tfrac{1}{3}x^3 - \ln(1+x).$$

Then, $g(0) = 0$ and

$$g'(x) = 1 - x + x^2 - \frac{1}{1+x} = \frac{x^3}{1+x} \ge 0$$

hold for all $x \ge 0$. Now suppose $0 \le l < n$. Then we have

$$\begin{aligned}
\ln f(n,l) &= (n-l)\ln\left(1+\frac{l}{n-l}\right) - l \\
&\le (n-l)\left\{\frac{l}{n-l} - \frac{1}{2}\left(\frac{l}{n-l}\right)^2 + \frac{1}{3}\left(\frac{l}{n-l}\right)^3\right\} - l \\
&= -\frac{1}{2}\cdot\frac{l^2}{n-l} + \frac{1}{3}\cdot\frac{l^3}{(n-l)^2} \\
&= -\frac{l^2}{2n} - \frac{l^3(n-3l)}{6n(n-l)^2}.
\end{aligned}$$

As we can show that $n - 3n^{(1/2)+\varepsilon} \ge 0$ for $n \ge 3^{2/(1-2\varepsilon)}$, we get the above result by substituting l with $n^{(1/2)+\varepsilon}$. □

PROOF OF THEOREM A.1. We consider the process of growing a chain in the long chain move. Suppose that the job ejected in the last step was j_{l-1}. Then the process stops if the inserted job $j_l \in B(j_{l-1})$ has already been included in $J_{l-1} = \{j_1, j_2, \ldots, j_{l-1}\}$. Therefore, $X \ge l$ holds if $j_i \notin J_{i-1}$ for $i = 1, 2, \ldots, l-1$, each of which occurs with probability $|J_{i-1}|/n = (n-i+1)/n$. Hence,

$$\begin{aligned}
\Pr(X \ge l) &= \frac{(n-1)(n-2)\cdots(n-l+1)}{n^{l-1}} \\
&= \frac{(n-1)!}{n^{l-1}(n-l)!}
\end{aligned}$$

holds for all $l = 1, 2, \ldots, n$. Then, for $l = \mathrm{o}(n)$, by using Stirling's approximation, we have

$$\begin{aligned}
\Pr(X \ge l) &= \frac{\sqrt{2\pi(n-1)}(n-1)^{n-1}\mathrm{e}^{-(n-1)}(1+\mathrm{O}(n^{-1}))}{n^{l-1}\sqrt{2\pi(n-l)}(n-l)^{n-l}\mathrm{e}^{-(n-l)}(1+\mathrm{O}(n^{-1}))} \\
&= \left(\frac{n}{n-l}\right)^{n-l}\left(\frac{n-1}{n}\right)^{n-1}\mathrm{e}^{1-l}\sqrt{\frac{n-1}{n-l}}(1+\mathrm{O}(n^{-1})) \\
&= \left\{\left(\frac{n}{n-l}\right)^{n-l}\mathrm{e}^{-l}\right\}\left(1-\frac{1}{n}\right)^{n(1-1/n)} \\
&\quad \cdot \mathrm{e}\sqrt{1+\frac{l-1}{n-l}}\left(1+\mathrm{O}(n^{-1})\right).
\end{aligned}$$

As $\Pr(X \ge l)$ is monotonically decreasing with l, we have by Lemma A.1 that

$$\Pr(X \ge l) \le \mathrm{e}^{-n^{2\varepsilon'}/2}(1+\mathrm{o}(1))$$

holds for an arbitrary $\varepsilon' \in (0, 1/2)$ and all $l \ge n^{(1/2)+\varepsilon'}$, if $n \ge 3^{2/(1-2\varepsilon')}$ holds. Therefore, for $\varepsilon = 2\varepsilon'$ and sufficiently large n, we have

$$\begin{aligned}
\mathrm{E}(X) &= \sum_{l=1}^{n} l\cdot\Pr(X=l) \\
&= \sum_{l=1}^{n}\Pr(X \ge l) \\
&\le \sum_{l<n^{(1/2)+\varepsilon'}} 1 + \sum_{l\ge n^{(1/2)+\varepsilon'}} \mathrm{e}^{-n^{2\varepsilon'}/2}(1+\mathrm{o}(1)) \\
&\le n^{(1/2)+\varepsilon'} + \mathrm{o}(1) \\
&\le n^{(1/2)+\varepsilon}. \quad \square
\end{aligned}$$

Appendix 3. Expected Number of Scans in *jlist*

We show that the expected value of $l^{(j)}_{\max}$ in Step 9 of algorithm INITIALIZE_B(σ) in §3.2.3 is $m/(m-1)$ if we assume that $c_{ij} \ne c_{ij'}$ for all $j \ne j' \in J$, $J' = J$, and a randomly selected σ. We also show that the expected value of $l^{(j)}_{\max}$ of algorithm DOUBLE_SHIFT(σ) is $secondsh\_max \cdot m/(m-1)$ under the same assumption except that $J' = J$ is not assumed. These results are clear from the following well-known lemma.

LEMMA A.2. *Consider a sequence of Bernoulli trials, in which the success probability of each trial is p. Then the expected number of trials until k successes occur is k/p.*

In our case, the success probability is $(m-1)/m$ in both cases. The number of successes needed is one in the former case, and *secondsh_max* in the latter case. Thus the above statements immediately follow from Lemma A.2.

Acknowledgments

The authors are grateful to Manuel Laguna and Helena Lourenço, who provided them with their original codes, and to Robert Nauss and Koji Nonobe, who provided them with their computational results. They are also grateful to Kouichi Taji, W. David Kelton (Editor-in-Chief), and the anonymous referees for their valuable comments. This research was partially supported by Scientific Grant-in-Aid by the Ministry of Education, Culture, Sports, Science, and Technology of Japan.

References

Amini, M. M., M. Racer. 1995. A hybrid heuristic for the generalized assignment problem. *Eur. J. Oper. Res.* **87** 343–348.

Berge, C. 1962. *The Theory of Graphs and Its Applications*. Methuen, London, and Wiley, New York. (Translated by A. Doig.)

Bertsekas, D. P. 1995. *Nonlinear Programming*. Athena Scientific, Belmont, MA.

Cattrysse, D. G., M. Salomon, L. N. Van Wassenhove. 1994. A set partitioning heuristic for the generalized assignment problem. *Eur. J. Oper. Res.* **72** 167–174.

Chu, P. C., J. E. Beasley. 1997. A genetic algorithm for the generalized assignment problem. *Comput. Oper. Res.* **24** 17–23.

Dongarra, J. J. 1999. Performance of various computers using standard linear equations software. Technical report. Computer Science Department, University of Tennessee, Knoxville, TN 37996-1301, and Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831. http://www.netlib.org/benchmark/performance.ps.

Edmonds, J. 1965. Maximum matching and a polyhedron with 0, 1-vertices. *J. Res. National Bureau of Standards* **69B** 125–130.

Fisher, M. L. 1981. The Lagrangian relaxation method for solving integer programming problems. *Management Sci.* **27** 1–18.

Garey, M. R., D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York.

Glover, F. 1990. Tabu search—A tutorial. *Interfaces* **20**(1) 74–94.

Glover, F. 1997. Ejection chain moves for generalized assignment problems. Manuscript. Leeds School of Business, University of Colorado, Boulder, CO.

Held, M., R. M. Karp. 1971. The traveling salesman problem and minimum spanning trees: Part II. *Math. Programming* **1** 6–25.

Johnson, D. S. 1990. Local optimization and the traveling salesman problem. M. S. Paterson, ed. *Automata, Languages and Programming, Lecture Notes in Computer Science*, No. 443. Springer-Verlag, Berlin, 446–461.

Kernighan, B. W., S. Lin. 1970. An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.* **49** 291–307.

Laguna, M., J. P. Kelly, J. L. González-Velarde, F. Glover. 1995. Tabu search for the multilevel generalized assignment problem. *Eur. J. Oper. Res.* **82** 176–189.

Lin, S., B. W. Kernighan. 1973. An effective heuristic algorithm for the traveling salesman problem. *Oper. Res.* **21** 498–516.

Lorena, L. A. N., M. G. Narciso. 1996. Relaxation heuristics for a generalized assignment problem. *Eur. J. Oper. Res.* **91** 600–610.

Lourenço, H. R., D. Serra. 2002. Adaptive search heuristics for the generalized assignment problem. *Mathware Soft Comput.* **9** 209–234.

Lourenço, H. R., M. Zwijnenburg. 1996. Combining the large-step optimization with tabu-search: Application to the job-shop scheduling problem. I. H. Osman, J. P. Kelly, eds. *Meta-Heuristics: Theory & Applications*. Kluwer Academic Publishers, Boston, MA, 661–675.

Martello, S., P. Toth. 1981. An algorithm for the generalized assignment problem. *Proc. Ninth IFORS Internat. Conf. on Operational Res.*. Hamburg, Germany, July 1981. North-Holland, Amsterdam, The Netherlands, 589–603.

Martello, S., P. Toth. 1990. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Chichester, U.K.

Martin, O., S. W. Otto, E. W. Felten. 1991. Large-step Markov chains for the traveling salesman problem. *Complex Systems* **5** 299–326.

Martin, O., S. W. Otto, E. W. Felten. 1992. Large-step Markov chains for the TSP incorporating local search heuristic. *Oper. Res. Lett.* **11** 219–224.

Nauss, R. M. 2003. Solving the generalized assignment problem: An optimizing and heuristic approach. *INFORMS J. Comput.* **15**(3) 249–266.

Nonobe, K., T. Ibaraki. 1998. A tabu search approach to the CSP (constraint satisfaction problem) as a general problem solver. *Eur. J. Oper. Res.* **106** 599–623.

Osman, I. H. 1995. Heuristics for the generalized assignment problem: Simulated annealing and tabu search approaches. *OR Spektrum* **17** 211–225.

Pesch, E., F. Glover. 1997. TSP ejection chains. *Discrete Appl. Math.* **76** 165–181.

Racer, M., M. M. Amini. 1994. A robust heuristic for the generalized assignment problem. *Ann. Oper. Res.* **50** 487–503.

Rego, C. 1998a. Relaxed tours and path ejections for the traveling salesman problem. *Eur. J. Oper. Res.* **106** 522–538.

Rego, C. 1998b. A subpath ejection chain method for the vehicle routing problem. *Management Sci.* **44** 1447–1459.

Rego, C., C. Roucairol. 1996. A parallel tabu search algorithm using ejection chains for the vehicle routing problem. I. H. Osman, J. P. Kelly, eds. *Meta-Heuristics: Theory & Applications*. Kluwer Academic Publishers, Boston, MA, 661–675.

Sahni, S., T. Gonzalez. 1976. P-complete approximation problems. *J. Association Comput. Machinery* **23** 555–565.

Savelsbergh, M. 1997. A branch-and-price algorithm for the generalized assignment problem. *Oper. Res.* **45** 831–841.

Trick, M. A. 1992. A linear relaxation heuristic for the generalized assignment problem. *Naval Res. Logist.* **39** 137–151.

Yagiura, M., T. Yamaguchi, T. Ibaraki. 1998. A variable depth search algorithm with branching search for the generalized assignment problem. *Optim. Methods Software* **10** 419–441.

Yagiura, M., T. Yamaguchi, T. Ibaraki. 1999. A variable depth search algorithm for the generalized assignment problem. S. Voß, S. Martello, I. H. Osman, C. Roucairol, eds. *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*. Kluwer Academic Publishers, Boston, MA, 459–471.