An Efficient Tabu Search Algorithm for the Tool Indexing Problem

Deepti Mohan
Diptesh Ghosh

Indian Institute of Management
Ahmedabad
Abstract

In this paper, we look at the tool indexing problem in which a single copy of each tool is allowed in the tool magazine. We develop problem specific methods to search the neighborhood efficiently and design a tabu search algorithm based on them. Computational experiments show that our algorithm is competent.
1 Introduction

Modern manufacturing setups often use automatic machining centers in manufacturing. In such automatic machining centers, several tools are available to perform various operations on a job sequentially. These machining centers use a tool changing mechanism to feed tools for the operations. To hold the tools and make them easily accessible at a job setup without manual intervention, automatic machining centers employ an automatic tool changer (ATC), also called a turret magazine. An ATC can be visualized as a disc that has multiple pockets called slots located equidistantly along its circumference and can rotate in either direction. The tools are arranged in the slots on the tool magazine. Whenever a tool is required by the machining center, the ATC rotates to bring the slot containing the tool to a fixed index position, from which the tool is picked, used, and returned after use by a tool change arm. Human intervention is not required in these operations, and hence such tool changers help reduce the non-productive time by changing the tools very quickly. Tool changers can be either drum type changers, used when the number of tools required is less than 30, or chain type changers when the number of tools required are larger than 30. Chain type tool changers can have up to 150 slots to store tools. The processing time of a job on a machining center includes actual machining time as well as non-machining time such as time required for the tooling operation described above. Profits, however, are generated only during the machining time, and so it is essential to perform the tooling operation as efficiently as possible. It is estimated that tooling accounts for 25% to 30% of fixed as well as variable costs in automated machining environments (Gray et al., 1993). During the tooling operation, selecting the next tool requires the ATC to rotate by an amount equal to the angular distance between relative positions of the two tools. This time elapsed during the rotation of ATC from one tool to the next tool is known as indexing time. It is obvious that the indexing time can be minimized by arranging the tools in the slots in an efficient manner. In this work the problem of minimizing the tool indexing time is addressed by finding an optimal allocation of tools to the slots. This problem is referred to as the tool indexing problem.

Tool indexing problems are of two types. In the first type, the turret magazine is allowed to have exactly one copy of each tool, and in the second type multiple copies of a tool may be present in the turret magazine. In this work, the first type of tool indexing problems is addressed.
2 Problem formulation

Recall that the tool indexing problem is one of finding an optimal policy of allocating tools to slots in the ATC. For ease of exposition, it is assumed that the number of tools required for a machining center’s operation is equal to the number of slots in the ATC.

Consider a turret magazine with $N$ slots, and an allocation of tools in the magazine. Also suppose that the turret rotates with an angular velocity of $\omega$. The time required to bring a tool in slot $l$ to the index position immediately after the tool change arm has replaced a tool in slot $k$ is given by $d(k, l) = (2\pi \omega / N) \min |l - k|, |N - |l - k||$, that can be rewritten as $d(k, l) = \min |l - k|, |N - |l - k||$ units where one unit is given by $(2\pi \omega / N)$.

Given a tool sequence for the machining center, the frequency $f(i, j)$ of tool $j$ immediately succeeding tool $i$ in the sequence for all pairs of tools in the magazine can be computed. Obviously $f(i, i) = 0$. These $f$ values contain all the necessary information about a tool sequence required by the tool indexing problem. Given an allocation of tools to slots represented by $\Pi = (\pi_1, \pi_2, \ldots, \pi_k, \ldots, \pi_N)$ in which tool $\pi_k$ occupies slot $k$, the total indexing time is given by $z(\Pi) = \sum_{k=1}^{N} \sum_{l=k+1}^{N} f(\pi_k, \pi_l) d(k, l)$ units. So, the tool indexing problem is one of obtaining a permutation $\Pi^*$ that minimizes $z(.)$.

This can be achieved through a mathematical programming formulation.

Define indices $i$ and $j$ on the set of tools in the magazine, and indices $k$ and $l$ on the set of slots in the magazine. Binary variables $x_{ik}$ take a value of 1 if tool $i$ is in slot $j$ and 0 otherwise. Then the tool indexing problem is defined as

\[
\text{Minimize} \quad \sum_{k=1}^{N-1} \sum_{l=k+1}^{N} \sum_{i=1}^{N} \sum_{j=1}^{N} f(i, j) . d(k, l) . x_{ik} . x_{jl} \quad \quad \frac{\text{cost of the allocation}}{1.2}
\]

Subject to

\[
\sum_{i=1}^{N} x_{ik} = 1 \text{ for all } i \quad \quad \frac{\text{each tool occupies exactly one slot}}{1.3}
\]

\[
\sum_{i=1}^{N} x_{ik} = 1 \text{ for all } k \quad \quad \frac{\text{each slot contains exactly one tool}}{1.4}
\]

\[
x_{ik} \in \{0, 1\} \text{ for all } i, \text{ for all } k \quad \quad \frac{\text{binary variables}}{1.5}
\]

This is clearly a quadratic assignment problem (QAP).
The QAP was formulated by Koopmans and Beckmann (1957) in the context of locating a small number of economic activities. Even though a more general version of the QAP was introduced by Lawler (1963), the Koopmans-Beckmann formulation is the most generally used version of the QAP. QAP is NP-hard (Burkard et al., 1998) and has been used to model various practical problems like hospital layout (Elshafei, 1977), blackboard wiring (Steinberg, 1961), single-row equidistant facility layout (Sarker et al., 1998), gray pattern (Taillard, 1995; Drezner et al., 2015), balanced graph partitioning problem (Rendl and Wolkowicz, 1995; Andreev and Räcke, 2004), maximum clique problem (Pardalos et al., 1994), and room allocation (Ciriani et al., 2004). Exact algorithms have been proposed for the problem, but they fail to reach an optimal solution within reasonable computation time for large size tool indexing problems. Current exact algorithms can solve mostly instances of problem size up to 40 (Drezner et al., 2015). Hence, heuristics-based solutions are a good alternative for solving large problem instances. An interested reader can refer to Burkard et al. (1998) and Loiola et al. (2007) for a discussion on QAP.

There are a few sources in the literature that solve the tool indexing problem through heuristics. Dereli and Filiz (2000) address the problem using a genetic algorithm. Baykasoğlu and Ozsoydan (2016) and Baykasoğlu and Ozsoydan (2017) solve a more complicated version of the problem by employing shortest path algorithm and simulated annealing algorithm respectively. Atta et al. (2019) develop a harmony search algorithm for solving the tool indexing problem.

In this work, neighborhood search based algorithms are proposed to solve the tool indexing problem. Methods to search solution neighborhoods efficiently that reduce the complexity of such searches by two orders of magnitude from naive search are proposed. These methods are then used to present a tabu search algorithm to solve the tool indexing problem.

3 Searching solution neighborhoods

Neighborhood of a solution is defined as a set of solutions that can be obtained from the solution by executing a set of pre-specified operations call moves. Recall that the solution representation in this problem is a permutation of the tools. This is possible since exactly one copy of each tool in the turret magazine is allowed and the number of slots in the magazine is assumed to be equal to the number of tools used. Two neighborhoods that are common to use for such “permutation” problems are the SWAP neighborhood and the INSERT neighborhood.

In the SWAP neighborhood, neighboring solutions or neighbors are generated by interchanging the positions of two tools in the solution. For example, suppose our current solution has eight tools (numbered 1, 2, ..., 8) in the permutation (3, 1, 4, 2, 5, 8, 6, 7). A SWAP neighbor may be obtained
for example, by swapping tools 2 and 7, to yield (3, 1, 4, 7, 5, 8, 6, 2). Another neighbor can be obtained by swapping tools 1 and 8 to yield (3, 8, 4, 2, 5, 1, 6, 7). Given a solution in an instance with \( N \) tools, then there are \( \binom{N}{2} \) solutions in the SWAP neighborhood, i.e., the size of the SWAP neighborhood is \( \mathcal{O}(N^2) \). Since the effort required to evaluate each solution na"ïvely is \( \mathcal{O}(N^2) \), the effort required to perform a naïve neighborhood search of a SWAP neighborhood is \( \mathcal{O}(N^4) \).

In the INSERT neighborhood, neighbors are generated by removing a tool from its position and inserting it in some other position in the solution. Consider for example the solution represented by the permutation (3, 1, 4, 2, 5, 8, 6, 7). Neighbors of this solution can be generated by removing, say, tool 1 from its current position and inserting it between tools 2 and 5 to obtain the solution (3, 4, 2, 1, 5, 8, 6, 7). It could also be inserted between tools 8 and 6 to obtain the neighbor (3, 4, 2, 5, 8, 1, 6, 7). Given a solution in an instance with \( N \) tools, there are \( N(N - 1) \), i.e., \( \mathcal{O}(N^2) \) INSERT neighbors of the solution, so that \( \mathcal{O}(N^4) \) effort is required to search the INSERT neighborhood of a solution na"ïvely.

Clearly an \( \mathcal{O}(N^4) \) time search is prohibitively expensive, and hence a method is now described to reduce this search effort by two orders of magnitude. A similar procedure that reduce the complexity of swap and insert neighborhood search to \( \mathcal{O}(n^2) \) have been independently obtained in Palubeckis (2021).

In the following, the presence of a frequency matrix \( F = [f(i, j)] \) representing the tool sequence required, and a distance matrix \( D = [d(k, l)] \) representing the distances between pairs of slots in the tool magazine is assumed. Notice that both matrices \( F \) and \( D \) are symmetric.

3.1 Searching SWAP neighborhoods efficiently

The number of neighbors in a SWAP neighborhood is \( \mathcal{O}(N^2) \). Each of these neighbors must be searched, and thus the only way in which the search can be made more efficient is to compute the cost of a neighbor in an incremental manner from the cost of the current solution or the cost of another neighbor computed before. Consider a current solution represented by the permutation \( \Pi_c = (\pi_1, \pi_2, \ldots, \pi_p, \ldots, \pi_q, \ldots \pi_N) \) and suppose it is required to compute the cost of the neighbor \( \Pi_{pq} = (\pi_1, \pi_2, \ldots, \pi_q, \ldots, \pi_p, \ldots \pi_N) \) obtained by interchanging tools \( \pi_p \) and \( \pi_q \) in \( \Pi_c \). Instead of computing \( z(\Pi_{pq}) \) ab initio, the cost can be computed as \( z(\Pi_{pq}) = z(\Pi_c) - \sum_{k=1}^{N} f(p, k) d(p, k) + \sum_{k=1}^{N} f(q, k) d(q, k) \). Once the cost \( z(\Pi_c) \) is computed and scored, the cost of all its SWAP neighbors can be computed in linear time, making the effort of computing the costs of all SWAP neighbors of \( \Pi_c \) \( \mathcal{O}(N^3) \). This method was proposed in Levitin and Rubinovitz (1993).
Now, consider another method of reducing the effort of searching the SWAP neighborhood of $\Pi_c$.

Let $\Pi_{pq}$ be the SWAP neighbor of $\Pi_c$ obtained by swapping the positions of $\pi_p$ and $\pi_q$ in $\Pi_c$. This swap can be obtained by performing the following steps. (In steps 1, 2, and 3, ‘*’ represents an empty position, and $f(*) = 0$ for all tools in the solution.)

1. Obtain a partial solution $\Pi^1 = (\pi_1, \pi_2, \ldots, \pi_{p-1}, *, \pi_{p+1}, \ldots, \pi_q, \ldots \pi_N)$ by removing tool $\pi_p$ from $\Pi_c$. Let $Rem(p) = z(\Pi_c) - z(\Pi^1)$.

2. Obtain a partial solution $\Pi^2 = (\pi_1, \pi_2, \ldots, \pi_{p-1}, *, \pi_{p+1}, \ldots, \pi_{q-1}, *, \pi_{q+1}, \ldots \pi_N)$ by removing tool $\pi_q$ from $\Pi^1$. Let $Rem(q) = z(\Pi^1) - z(\Pi^2)$.

3. Obtain a partial solution $\Pi^3 = (\pi_1, \pi_2, \ldots, \pi_{p-1}, *, \pi_{p+1}, \ldots, \pi_{q-1}, \pi_p, \pi_{q+1}, \ldots \pi_N)$ by adding back tool $\pi_p$ in position $q$ in $\Pi^2$. Let $Add(p, q) = z(\Pi^3) - z(\Pi^2)$.

4. Obtain $\Pi_{pq} = (\pi_1, \pi_2, \ldots, \pi_{p-1}, \pi_q, \pi_{p+1}, \ldots, \pi_{q-1}, \pi_p, \pi_{q+1}, \ldots \pi_N)$ by adding back tool $\pi_q$ in position $p$ in $\Pi^3$. Let $Add(q, p) = z(\Pi_{pq}) - z(\Pi^3)$.

Then $z(\Pi_{pq}) - z(\Pi_c) = Add(p, q) + Add(q, p) - Rem(p, p) - Rem(q, q)$. Figure 1 illustrates the above steps for obtaining SWAP neighbor $\Pi_{14}$ from current solution $\Pi = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

Figure 1: Illustration of the execution of swap move between tools in positions 1 and 4 to generate the neighbor $\Pi_{14}$ from $\Pi$. 

![Figure 1](image-url)
Now let a matrix $A = [a(i,j)]$ be defined in which $a(i,j) = \sum_{k=1}^{N} f(\pi_k, \pi_i) \cdot d(k,j)$.

The diagonal element $a(p, p) = \sum_{k=1}^{N} f(\pi_k, \pi_p) \cdot d(k, p)$ in $A$ is the sum of the coefficients of all the terms in the cost function $z(\Pi_c)$ containing the terms $x_p$. So clearly

$$Rem(p) = a(p,p).$$

Similarly, $a(q, q)$ is the sum of the coefficients of all the terms in $z(\Pi_c)$ containing the terms $x_q$ and would be the reduction in cost if tool $\pi_q$ was removed from $\Pi_c$. But since in $\Pi^1$ tool $\pi_p$ had already been removed,

$$Rem(q) = a(q,q) - f(\pi_p, \pi_q) \cdot d(p,q).$$

Now $a(p, q) = \sum_{k=1}^{N} f(\pi_k, \pi_p) \cdot d(k, q)$ is the sum of the coefficients of all terms that would be added to the function $z(\Pi_c)$ if an additional copy of tool $\pi_p$ interacted with all other tools in the magazine. This equals $Add(p, q)$ since the distance between the second copy of tool $\pi_p$ and the tool in position $q$ (originally tool $\pi_q$ in $\Pi_c$ and tool $\pi_p$ in $\Pi^2$) is zero. So

$$Add(p, q) = a(p,q).$$

$Add(q, p)$ exceeds $a(q, p)$ by the term $f(\pi_q, \pi_p) d(q,p)$ due to the presence of tool $\pi_p$ in position $q$ in $\Pi^3$. So $Add(q, p) = a(q, p) + f(\pi_q, \pi_p) d(q,p) = a(q, p) + f(\pi_p, \pi_q) d(p,q)$, since both $F$ and $D$ are symmetric.

Thus

$$z(\Pi_{pq}) - z(\Pi_c) = a(p, q) + a(q, p) - a(p, p) - a(q, q) + 2 \times f(\pi_p, \pi_q) \cdot d(p,q), \quad (2)$$

which can be computed in constant time given the matrix $A$. This implies that if the matrix $A$ is available, then the effort of searching the SWAP neighborhood of $\Pi_c$ is $\mathcal{O}(N^2)$.

Let $\phi(q) = q$ if $1 \leq q \leq N$ and $= q - N$ if $N + 1 \leq q \leq 2N$. Consider matrix elements $a(p, q), a(p, \phi(q + 1))$, and $a(p, \phi(q + 2))$. Let $\Delta(p, q) = a(p, \phi(q + 1)) - a(p,q)$, and let $\Delta^2(p, q) = \Delta(p, \phi(q + 1)) - \Delta(p, q) = a(p, \phi(q + 2)) - 2a(p, \phi(q + 1)) + a(p, q)$. Algebraically, $\Delta^2(p, q)$ can be computed as

$$\Delta^2(p, q) = \begin{cases} 2f(\pi_p, \pi_{\phi(q + 1)}) - 2f(\pi_p, \pi_{\phi(q + 1 + k)}) & \text{if } n = 2k, \text{ i.e., even} \\ 2f(\pi_p, \pi_{\phi(q + 1)}) - f(\pi_p, \pi_{\phi(q + k)}) - f(\pi_p, \pi_{\phi(q + 1 + k)}) & \text{if } n = 2k + 1, \text{ i.e., odd} \end{cases}$$

in constant, i.e., $\mathcal{O}(1)$ effort.

In matrix $A$, for any row $p$, the values of $a(p, p) = \sum_{k=1}^{N} f(\pi_k, \pi_p) \cdot d(k, p)$ and $a(p, \phi(p + 1)) = \sum_{k=1}^{N} f(\pi_k, \pi_p) \cdot d(k, \phi(p + 1))$ can both be computed with linear, i.e. $\mathcal{O}(N)$ effort. Once $a(p,p)$
is known, then each of the values of \( a(p, k) \) where column \( k \) is at an even distance from column \( p \) can be computed in constant time using the expression for \( \Delta^2(\ldots) \). Similarly, since the value of \( a(p, \phi(p + 1)) \) is known, each of the values of \( a(p, k) \) where column \( k \) is at an odd distance from column \( p \) can be computed in constant time using the expression for \( \Delta^2(\ldots) \). Thus row \( p \) of matrix \( A \) can be computed with \( \mathcal{O}(N^2) \) effort, and hence all the elements of matrix \( A \) can be computed with \( \mathcal{O}(N^2) \) effort.

The following Pseudo-code formalizes the method for searching the SWAP neighborhood of a solution with \( \mathcal{O}(N^2) \) effort.

\[
\text{Algorithm 1: Pseudo-code for searching SWAP neighborhood of a solution.}
\]

```
Input: Frequency matrix \( F = [f(i, j)], i, j \in \{1, \ldots, N\} \), distance matrix \( D = [d(k, l)], k, l \in \{1, \ldots, N\} \), current Solution \( \Pi_c = [\pi(i)] \), cost function \( z(.) \).

Output: The best SWAP neighbour \( \Pi_{best} \) of \( \Pi_c \).

1. \( \Delta^2(p, q) \leftarrow \Delta^2(p, q) \) computed using Equation (3) for all \( p \) and \( q \); 
2. \( \text{foreach } p \in \{1, \ldots, N\} \) do 
3. \( a(p, p) \leftarrow \sum_{k=1}^{N} f(\pi_k, \pi_p) \cdot d(k, p); \)
4. \( a(p, \phi(p + 1)) \leftarrow \sum_{k=1}^{N} f(\pi_k, \pi_p) \cdot d(k, \phi(p + 1)); \)
5. \( \Delta(p, p) \leftarrow a(p, \phi(p + 1)) - a(p, p); \)
6. \( \text{foreach } q \in \{p + 1, \ldots, N\} \) do 
7. \( \Delta(p, \phi(q)) \leftarrow \Delta(p, \phi(q - 1)) + \Delta^2(p, \phi(q - 1)); \)
8. \( a(p, \phi(q + 1)) \leftarrow a(p, \phi(q)) + \Delta(p, \phi(q)); \)
9. end 
10. end 
11. \( z(\Pi_{pq}) \leftarrow z(\Pi_c) + a(p, q) + a(q, p) - a(p, p) - a(q, q) + 2 \times f(\pi(q), \pi(p)) \cdot d(q, p) \) for all \( p \) and \( q \); 
12. \( \Pi_{best} \leftarrow \Pi_{pq} \) with best \( z(\Pi_{pq}) \); 
13. return \( \Pi_{best} \); 
```

The following example first illustrates the relationship given in Equation 3 and then proceeds to illustrate Algorithm 1.

For a problem with \( N = 8 \) and current solution \( \Pi_c \) where tool \( i \) is in position \( i \) i.e., \( \pi(i) = i \) for \( i \in \{1, 2, \ldots, 8\} \), the entries of first row of matrix \( A = a(i, j) \) computed using the definition \( a(i, j) = \sum_{k=1}^{N} f(\pi_k, \pi_i) \cdot d(k, j) \) is given below.

\[
\begin{align*}
a(1, 1) &= f(1, 2) \times 1 + f(1, 3) \times 2 + f(1, 4) \times 3 + f(1, 5) \times 4 + f(1, 6) \times 3 + f(1, 7) \times 2 + f(1, 8) \times 1 \\
a(1, 2) &= f(1, 2) \times 0 + f(1, 3) \times 1 + f(1, 4) \times 2 + f(1, 5) \times 3 + f(1, 6) \times 4 + f(1, 7) \times 3 + f(1, 8) \times 2 \\
a(1, 3) &= f(1, 2) \times 1 + f(1, 3) \times 0 + f(1, 4) \times 1 + f(1, 5) \times 2 + f(1, 6) \times 3 + f(1, 7) \times 4 + f(1, 8) \times 3 \\
a(1, 4) &= f(1, 2) \times 2 + f(1, 3) \times 1 + f(1, 4) \times 0 + f(1, 5) \times 1 + f(1, 6) \times 2 + f(1, 7) \times 3 + f(1, 8) \times 4 \\
a(1, 5) &= f(1, 2) \times 3 + f(1, 3) \times 2 + f(1, 4) \times 1 + f(1, 5) \times 0 + f(1, 6) \times 1 + f(1, 7) \times 2 + f(1, 8) \times 3 \\
\end{align*}
\]
\[ a(1, 6) = f(1, 2) \times 4 + f(1, 3) \times 3 + f(1, 4) \times 2 + f(1, 5) \times 1 + f(1, 6) \times 0 + f(1, 7) \times 1 + f(1, 8) \times 2 \]
\[ a(1, 7) = f(1, 2) \times 3 + f(1, 3) \times 4 + f(1, 4) \times 3 + f(1, 5) \times 2 + f(1, 6) \times 1 + f(1, 7) \times 0 + f(1, 8) \times 1 \]
\[ a(1, 8) = f(1, 2) \times 2 + f(1, 3) \times 3 + f(1, 4) \times 4 + f(1, 5) \times 3 + f(1, 6) \times 2 + f(1, 7) \times 1 + f(1, 8) \times 0 \]

Using the above expressions, \( \Delta \) values can be written as

\[ \Delta(1, 1) = a(1, 2) - a(1, 1) = -f(1, 2) - f(1, 3) - f(1, 4) - f(1, 5) + f(1, 6) + f(1, 7) + f(1, 8) \]
\[ \Delta(1, 2) = a(1, 3) - a(1, 2) = f(1, 2) - f(1, 3) - f(1, 4) - f(1, 5) - f(1, 6) + f(1, 7) + f(1, 8) \]
\[ \Delta(1, 3) = a(1, 4) - a(1, 3) = f(1, 2) + f(1, 3) - f(1, 4) - f(1, 5) - f(1, 6) - f(1, 7) + f(1, 8) \]
\[ \Delta(1, 4) = a(1, 5) - a(1, 4) = f(1, 2) + f(1, 3) + f(1, 4) - f(1, 5) - f(1, 6) - f(1, 7) - f(1, 8) \]
\[ \Delta(1, 5) = a(1, 6) - a(1, 5) = f(1, 2) + f(1, 3) + f(1, 4) + f(1, 5) - f(1, 6) - f(1, 7) - f(1, 8) \]
\[ \Delta(1, 6) = a(1, 7) - a(1, 6) = -f(1, 2) + f(1, 3) + f(1, 4) + f(1, 5) + f(1, 6) - f(1, 7) - f(1, 8) \]
\[ \Delta(1, 7) = a(1, 8) - a(1, 7) = -f(1, 2) - f(1, 3) + f(1, 4) + f(1, 5) + f(1, 6) + f(1, 7) - f(1, 8) \]

Substituting these expressions in place of \( \Delta \), \( \Delta^2 \) value reduces to an expression with just two elements of frequency matrix as below.

\[ \Delta^2(1, 1) = \Delta(1, 2) - \Delta(1, 1) = 2f(1, 2) - 2f(1, 6) \]
\[ \Delta^2(1, 2) = \Delta(1, 3) - \Delta(1, 2) = 2f(1, 3) - 2f(1, 7) \]
\[ \Delta^2(1, 3) = \Delta(1, 4) - \Delta(1, 3) = 2f(1, 4) - 2f(1, 8) \]
\[ \Delta^2(1, 4) = \Delta(1, 5) - \Delta(1, 4) = 2f(1, 5) \]
\[ \Delta^2(1, 5) = \Delta(1, 6) - \Delta(1, 5) = 2f(1, 6) - 2f(1, 2) \]
\[ \Delta^2(1, 6) = \Delta(1, 7) - \Delta(1, 6) = 2f(1, 7) - 2f(1, 3) \]

Clearly, the expressions for \( \Delta^2 \) takes the form as given in Equation 3 which makes it computable in constant time. This property of the \( \Delta^2 \) values is exploited to compute the entries of matrix \( A \) in \( \mathcal{O}(N^2) \). To illustrate this, a concrete tool sequence for the example is considered; the \( F \) matrix for the example problem is taken as

\[
F = \begin{bmatrix}
  f(1, 1) & f(1, 2) & f(1, 3) & \ldots & f(1, 8) \\
  f(2, 1) & f(2, 2) & f(2, 3) & \ldots & f(2, 8) \\
  \vdots & \ddots & \ddots & \ddots & \vdots \\
  f(8, 1) & f(8, 2) & f(8, 3) & \ldots & f(8, 8)
\end{bmatrix}
\quad =
\begin{bmatrix}
  0 & 1 & 3 & 4 & 1 & 0 & 1 & 2 \\
  1 & 0 & 2 & 5 & 2 & 1 & 0 & 4 \\
  3 & 2 & 0 & 1 & 2 & 4 & 1 & 0 \\
  4 & 5 & 1 & 0 & 2 & 1 & 1 & 3 \\
  1 & 2 & 2 & 2 & 0 & 1 & 5 & 1 \\
  0 & 1 & 4 & 1 & 1 & 0 & 2 & 3 \\
  1 & 0 & 1 & 1 & 5 & 2 & 0 & 4 \\
  2 & 4 & 0 & 3 & 1 & 3 & 4 & 0
\end{bmatrix}
\]
Using this example problem, each step of the algorithm is explained below. First, all entries of $\Delta^2$ matrix are computed using Equation 3 as

\[
\Delta^2(1, 1) = 2f(1, 2) - 2f(1, 6) = 2 \times 1 - 2 \times 0 = 2 \\
\Delta^2(1, 2) = 2f(1, 3) - 2f(1, 7) = 2 \times 3 - 2 \times 1 = 4 \\
\vdots \\
\Delta^2(1, 5) = 2f(1, 6) - 2f(1, 2) = 2 \times 0 - 2 \times 1 = -2 \\
\Delta^2(1, 6) = 2f(1, 7) - 2f(1, 3) = 2 \times 1 - 2 \times 3 = -4 \\
\vdots 
\]

Evaluating entries of the first row of matrix $A$ is illustrated below (same procedure is applied to compute the elements of matrix $A$ row by row). The first two elements in row 1 of matrix $A$ is computed as

\[
a(1, 1) = f(1, 2) \times 1 + f(1, 3) \times 2 + f(1, 4) \times 3 + f(1, 5) \times 4 + f(1, 6) \times 3 + f(1, 7) \times 2 + f(1, 8) \times 1 = 1 \times 1 + 3 \times 2 + 4 \times 3 + 1 \times 4 + 0 \times 3 + 1 \times 2 + 2 \times 1 = 27 \\
a(1, 2) = f(1, 2) \times 0 + f(1, 3) \times 1 + f(1, 4) \times 2 + f(1, 5) \times 3 + f(1, 6) \times 4 + f(1, 7) \times 3 + f(1, 8) \times 2 = 1 \times 0 + 3 \times 1 + 4 \times 2 + 1 \times 3 + 0 \times 4 + 1 \times 3 + 2 \times 2 = 21. 
\]

Now $\Delta(1, 1)$ can be calculated as $\Delta(1, 1) = a(1, 2) - a(1, 1) = 21 - 27 = -6$. Once $\Delta^2(1, 1)$ and $\Delta(1, 1)$ are known, $\Delta(1, 2)$ can be obtained as $\Delta(1, 2) = \Delta(1, 1) + \Delta^2(1, 1) = -6 + 2 = -4$. The third element in the first row of matrix $A$ can now be calculated using $\Delta(1, 2)$ as $a(1, 3) = a(1, 2) + \Delta(1, 2) = 21 + -4 = 17$. Again, $\Delta(1, 3) = \Delta(1, 2) + \Delta^2(1, 2) = -4 + 4 = 0$ and $a(1, 4) = a(1, 3) + \Delta(1, 3) = 17 + 0 = 17$. This way, all entries in the first row of matrix $A$ can be computed.

Matrix $A$ obtained by following this procedure for each row is given below.

\[
A = \begin{bmatrix}
a(1, 1) & a(1, 2) & a(1, 3) & \ldots & a(1, 8) \\
a(2, 1) & a(2, 2) & a(2, 3) & \ldots & a(2, 8) \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
a(8, 1) & a(8, 2) & a(8, 3) & \ldots & a(8, 8)
\end{bmatrix}
= \begin{bmatrix}
27 & 21 & 17 & 17 & 21 & 27 & 31 & 31 \\
34 & 31 & 26 & 25 & 26 & 29 & 34 & 35 \\
27 & 30 & 29 & 26 & 25 & 22 & 23 & 26 \\
23 & 24 & 33 & 42 & 45 & 44 & 35 & 26 \\
26 & 28 & 32 & 30 & 30 & 28 & 24 & 26 \\
35 & 35 & 31 & 29 & 21 & 21 & 25 & 27 \\
34 & 35 & 38 & 33 & 34 & 33 & 30 & 35
\end{bmatrix}
\]

The cost of each SWAP neighbor of $\Pi_c$ can now be computed using Equation 2. For instance, cost
of the SWAP neighbor $\Pi_{12}$ is

$$z(\Pi_{12}) = z(\Pi_c) + a(1, 2) + a(2, 1) - a(1, 1) - a(2, 2) + 2 \times f(1, 2).d(1, 2).$$

Cost of the current solution evaluated using the cost function $z(.)$ is 123. So,

$$z(\Pi_{12}) = 123 + 21 + 34 - 27 - 31 + 2 \times 1 \times 1 = 123 - 1 = 122.$$

### 3.2 Searching INSERT neighborhoods efficiently

As with the SWAP neighborhood, the number of neighbors in the INSERT neighborhood of a solution is $O(N^2)$. The remainder of this section will show a method of reducing the effort of searching the INSERT neighborhood of $\Pi_c = (\pi_1, \pi_2, \ldots, \pi_p, \ldots, \pi_q, \ldots \pi_N)$.

Consider an INSERT neighbor $\Pi_{pq} = (\pi_1, \pi_2, \ldots, \pi_{p-1}, \pi_{p+1}, \ldots, \pi_q, \pi_p, \pi_{q+1}, \ldots \pi_N)$ obtained by removing the tool at position $p$ in $\Pi_c$ and inserting it between the tools at positions $q$ and $q + 1$ in $\Pi_c$. In this exposition, it is assumed that $q = p + r > p$, i.e., position $q$ is $r$ slots to the right of position $p$. The treatment when $q < p$ can be done in a largely similar manner and is not elaborated here.

Since $q$ is $r$ slots to the right of $p$, the INSERT neighbor can be obtained by performing $r$ swap moves between adjacent elements, i.e., first swap tools $\pi_p$ and $\pi_{p+1}$ to obtain one intermediate solution, say $\Pi^1$, then swap tools $\pi_{p+1}$ and $\pi_{p+2}$ in $\Pi^1$ to obtain a solution $\Pi^2$ and so on until tools $\pi_{q-1}$ and $\pi_q$ in $\Pi^{r-1}$ are swapped to obtain $\Pi_{pq}$. Note that tools $\pi_{p+1}$ in $\Pi^1$, $\pi_{p+2}$ in $\Pi^2$ etc. are all the same, i.e., tool $\pi_p$ in $\Pi_c$. For example, if $\Pi_c = (1, 2, 3, 4, 5, 6, 7, 8)$, then $\Pi_{27} = (1, 3, 4, 5, 6, 2, 7, 8)$. The solution $\Pi_{27}$ is obtained through intermediate solutions $\Pi^1 = (1, 3, 2, 4, 5, 6, 7, 8)$, $\Pi^2 = (1, 3, 4, 2, 5, 6, 7, 8)$, $\Pi^3 = (1, 3, 4, 5, 2, 6, 7, 8)$, and $\Pi^4 = (1, 3, 4, 5, 6, 2, 7, 8)$. Then the difference $z(\Pi_{pq}) - z(\Pi_c)$ when $q = p + r$ can be computed as

$$z(\Pi_{pq}) - z(\Pi_c) = (z(\Pi_{pq}) - z(\Pi^{r-1})) + (z(\Pi^{r-1}) - z(\Pi^{r-2})) + \ldots + (z(\Pi^1) - z(\Pi_c)).$$

Each of the compound terms on the right-hand side computes the increase in cost if two adjacent tools are swapped; for example the term $(z(\Pi^1) - z(\Pi_c))$ computes the increase in cost if the tool at position $p$ is swapped with the tool at position $p + 1$. Figure 2 illustrates the above procedure for obtaining INSERT neighbor $\Pi_{15}$ from current solution $\Pi = \{1, 2, 3, 4, 5, 6, 7, 8\}$ through intermediate solutions $\Pi^1$ to $\Pi^3$.

In order to compute the increase in cost due to swap move efficiently, a matrix $S = [s(p, q)]$ is constructed where $s(p, q) = \sum_{k=1}^{q} f(p, k)$. Note that $s(p, q + 1) = s(p, q) + f(p, q + 1)$, and can be computed in constant time once $s(p, q)$ is known. Since $s(p, p) = f(p, p) = 0$, each row of the $S$ matrix can be computed in $O(N)$ time, which implies that the $S$ matrix can be computed in $O(N^2)$ time.
Next suppose that the $S$ matrix is available. Then the increase in the cost of the solution obtained after the change over the cost of $\Pi_c$ can be computed as follows. The expression for cost depends on whether $N$ is even or odd and whether $p$ and $q$ are closer in the clockwise or anticlockwise direction from position 1. The computation for the case when $N$ is an even number given by $N = 2K$, and $q \leq K$ has been shown in this exposition. Expressions for other cases can be arrived through similar approach.

Suppose that tool at position $p$ is swapped with the tool at position $p+1$ in solution $\Pi_c$. Note that $p < q < K < N$. Due to this move, distance from tool $\pi_p$ to tools $\pi_1, \pi_2, \ldots, \pi_{p-1}$ increases by one while distance from tool $\pi_{p+1}$ to tools at these positions decreases by one and hence the associated increase in cost is $\{f(p, 1) + f(p, 2) + \ldots + f(p, p-1)\} - \{f(p+1, 1) + f(p+1, 2) + \ldots + f(p+1, p-1)\}$. The distance from tool $\pi_p$ to tools $\pi_{p+2}, \pi_{p+3}, \ldots, \pi_{p+K}$ decreases by one while distance from tool $\pi_{p+1}$ to tools at these positions increases by one and hence the associated increase in cost is $\{f(p+1, p+2) + f(p+1, p+3) + \ldots + f(p+1, p+K)\} - \{f(p, p+2) + f(p, p+3) + \ldots + f(p, p+K)\}$. The distance from tool $\pi_p$ to tools $\pi_{p+K+1}, \pi_{p+K+2}, \ldots, \pi_N$ increases by one while distance from tool $\pi_{p+1}$ to tools at these positions decreases by one and hence the associated increase in cost is $\{f(p, p+K+1) + f(p, p+K+2) + \ldots + f(p, N)\} - \{f(p+1, p+K+1) + f(p+1, p+K+2) + \ldots + f(p+1, N)\}$. The distance from tool $\pi_p$ to tools $\pi_N$ decreases by one while distance from tool $\pi_{p+1}$ to tools at these positions increases by one and hence the associated increase in cost is $\{f(p+1, N)\} - \{f(p, N)\}$. The distance from tool $\pi_p$ to tools $\pi_N$ increases by one while distance from tool $\pi_{p+1}$ to tools at these positions decreases by one and hence the associated increase in cost is $\{f(p, N)\} - \{f(p+1, N)\}$.

Figure 2: Illustration of execution of insert move in which tool 1 is inserted between tools 5 and 6 to generate $\Pi_{15}$ from $\Pi$.
Thus the increase in cost due to the swap move is given by

\[
z\left(\Pi^1\right) - z(\Pi_c) = \{f(p, 1) + f(p, 2) + \ldots + f(p, p - 1)\} \\
- \{f(p + 1, 1) + f(p + 1, 2) + \ldots + f(p + 1, p - 1)\} \\
+ \{f(p + 1, p + 2) + f(p + 1, p + 3) + \ldots + f(p + 1, p + K)\} \\
- \{f(p, p + 2) + f(p, p + 3) + \ldots + f(p, p + K)\} \\
+ \{f(p, p + K + 1) + f(p, p + K + 2) + \ldots + f(p, N)\} \\
- \{f(p + 1, p + K + 1) + f(p + 1, p + K + 2) + \ldots + f(p + 1, N)\}
\] (4)

This can be rewritten in terms of the \( S \) matrix entries as

\[
z\left(\Pi^1\right) - z(\Pi_c) = s(p, p - 1) - s(p + 1, p - 1) + \{s(p + 1, p + K) - s(p + 1, p + 1)\} \\
- \{s(p, p + K) - s(p, p + 1)\} + \{s(p, N) - s(p, p + K)\} \\
- \{s(p + 1, N) - s(p + 1, p + K)\}
\] (5)

This gives \( z\left(\Pi^1\right) = z\left(\Pi_{p(p+1)}\right) \) in constant time.

Next the tool at position \( p + 1 \) is swapped with the tool at position \( p + 2 \) in solution \( \Pi^1 \). As a result, tools in positions \( p, p + 1, p + 2 \) in solution \( \Pi^2 \) are \( \pi_{p+1}, \pi_{p+2}, \) and \( \pi_p \), respectively. Due to this move, distance from tool \( \pi_p \) to tools \( \pi_1, \pi_2, \ldots \pi_{p-1} \) increases by one while distance from tool \( \pi_{p+2} \) to tools at these positions decreases by one and hence the associated increase in cost is

\[
\{f(p, 1) + f(p, 2) + \ldots + f(p, p - 1)\} - \{f(p + 2, 1) + f(p + 2, 2) + \ldots + f(p + 2, p - 1)\}
\]

The distance from tool \( \pi_p \) to tools \( \pi_{p+3}, \pi_{p+4}, \ldots, \pi_{p+K+1} \) decreases by one while distance from tool \( \pi_{p+2} \) to tools at these positions increases by one and hence the associated increase in cost is

\[
\{f(p + 2, p + 3) + f(p + 2, p + 4) + \ldots + f(p + 2, p + K + 1)\} - \{f(p, p + 3) + f(p, p + 4) + \ldots + f(p, p + K + 1)\}
\]

The distance from tool \( \pi_p \) to tools \( \pi_{p+K+2}, \pi_{p+K+3}, \ldots, \pi_N \) increases by one while distance from tool \( \pi_{p+2} \) to tools at these positions decreases by one and hence the associated increase in cost is

\[
\{f(p, p + K + 2) + f(p, p + K + 3) + \ldots + f(p, N)\} - \{f(p + 2, p + K + 2) + f(p + 2, p + K + 3) + \ldots + f(p + 2, N)\}
\]

Thus the increase in cost due to the swap move is

\[
z\left(\Pi^2\right) - z\left(\Pi^1\right) = \{f(p, 1) + f(p, 2) + \ldots + f(p, p - 1)\} \\
- \{f(p + 2, 1) + f(p + 2, 2) + \ldots + f(p + 2, p - 1)\} \\
+ \{f(p + 2, p + 3) + f(p + 2, p + 4) + \ldots + f(p + 2, p + K + 1)\} \\
- \{f(p, p + 3) + f(p, p + 4) + \ldots + f(p, p + K + 1)\} \\
+ \{f(p, p + K + 2) + f(p, p + K + 3) + \ldots + f(p, N)\} \\
- \{f(p + 2, p + K + 2) + f(p + 2, p + K + 3) + \ldots + f(p + 2, N)\}
\] (6)
and this can be expressed in terms of the $S$ matrix entries as

$$z(\Pi^2) - z(\Pi^1) = \{s(p, p - 1) - s(p + 2, p - 1)\} + \{s(p + 2, p + K + 1) - s(p + 2, p + 2)\}$$
$$- \{s(p, p + K + 1) - s(p, p + 2)\} + \{s(p, N) - s(p, p + K + 1)\}$$
$$- \{s(p + 2, N) - s(p + 2, p + K + 1)\}$$

(7)

Similarly, expressions for $z(\Pi^3) - z(\Pi^2)$, $z(\Pi^4) - z(\Pi^3)$, ..., $z(\Pi_{pq}) - z(\Pi^{r-1})$ can be obtained in constant time.

The important thing to be noted here is that the intermediate solutions $\Pi^1$, $\Pi^2$, ..., $\Pi^{r-1}$ are INSERT neighbors $\Pi_{p(p+1)}$, $\Pi_{p(p+2)}$, ..., $\Pi_{p(q-1)}$ respectively of $\Pi_c$. Thus by computing $z(\Pi_{pN})$ (the cost of INSERT neighbor $\Pi_{pN}$) through intermediate solutions as described above, $z(\Pi_{p(p+1)})$, $z(\Pi_{p(p+1)})$, ..., $z(\Pi_{p(p+N-1)})$ are also obtained.

The following pseudo-code formalizes the method for searching the INSERT neighborhood of a solution with $O(N^2)$ effort.

**Algorithm 2:** Pseudo-code for searching INSERT neighborhood of a solution.

**Input:** Frequency matrix $F = [f(i, j)]$, $i, j \in \{1, \ldots, N\}$, distance matrix $D = [d(k, l)]$, $k, l \in \{1, \ldots, N\}$, current Solution $\Pi_c = [\pi(i)]$, cost function $z(.)$.

**Output:** The best INSERT neighbor $\Pi_{best}$ of $\Pi_c$.

1. $s(p, q) \leftarrow \sum_{k=p}^{q} f(p, k)$ for all $p$ and $q$;
2. **foreach** $p \in \{1, \ldots, N\}$ **do**
3.  
4.  **foreach** $q \in \{p + 1, \ldots, N\}$ **do**
5.  
6.  $costdiff \leftarrow$ computed using entries of matrix $S$;
7.  
8.  $z(\Pi_{pq}) \leftarrow insolcost + costdiff$;
9.  
10. **end**
11. **end**
12. $\Pi_{best} \leftarrow \Pi_{pq}$ with best $z(\Pi_{pq})$;
13. **return** $\Pi_{best}$;

This method is illustrated by taking the same example problem as used in Section 3.1. The entries of matrix $S$ are first computed as

$s(1, 1) = f(1, 1) = 0$
$s(1, 2) = f(1, 1) + f(1, 2) = 1$
$s(1, 3) = f(1, 1) + f(1, 2) + f(1, 3) = 4$
\[ s(1, 4) = f(1, 1) + f(1, 2) + f(1, 3) + f(1, 4) = 8 \\
\vdots \\
\[ s(1, 8) = f(1, 1) + f(1, 2) + \ldots + f(1, 8) = 12 \\
\vdots \\
to obtain

\[
S = \begin{bmatrix}
s(1, 1) & s(1, 2) & s(1, 3) & \ldots & s(1, 8) \\
s(2, 1) & s(2, 2) & s(2, 3) & \ldots & s(2, 8) \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
s(8, 1) & s(8, 2) & s(8, 3) & \ldots & s(8, 8)
\end{bmatrix} = \begin{bmatrix}
0 & 1 & 4 & 8 & 9 & 9 & 10 & 12 \\
1 & 1 & 3 & 8 & 10 & 11 & 11 & 15 \\
3 & 5 & 5 & 6 & 8 & 12 & 13 & 13 \\
4 & 9 & 10 & 12 & 13 & 14 & 17 & \\
1 & 3 & 5 & 7 & 7 & 8 & 13 & 14 \\
0 & 1 & 5 & 6 & 7 & 7 & 9 & 12 \\
1 & 1 & 2 & 3 & 8 & 10 & 10 & 14 \\
2 & 6 & 6 & 9 & 10 & 13 & 17 & 17
\end{bmatrix}
\]

Consider insert move \( \Pi_{4,5} \) where tool 4 is inserted between tool 5 and tool 6 in \( \Pi_c \). The incremental cost of this move, \( \Delta(4, 5) \), can be computed using the entries of matrix \( S \) as follows. Since the positions of only tool 4 and tool 5 change while rest of the tools are in the same position before and after the move, \( \Delta(4, 5) \) is the sum of incremental cost due to change in position of tool 4 and that of tool 5. Before the insert operation, distance of tool 4 to each of the tools from 1 to 8 is 3, 2, 1, 0, 1, 2, 3, 4, respectively, while that of tool 5 to each of these tools is 4, 3, 2, 1, 0, 1, 2, 3. After the insert operation, distance of tool 4 to these tools changes to 4, 3, 2, 0, 1, 1, 2, 3, respectively, while that of tool 5 to these tools changes to 3, 2, 1, 1, 0, 2, 3, 4, respectively. Thus

\[
\Delta(4, 5) = (4f_{41} + 3f_{42} + 2f_{43} + f_{45} + f_{46} + 2f_{47} + 3f_{48}) - (3f_{41} + 2f_{42} + f_{43} + f_{45} + 2f_{46} + 3f_{47} + 4f_{48}) + (3f_{51} + 2f_{52} + f_{53} + f_{54} + 2f_{56} + 3f_{57} + 4f_{58}) - (4f_{51} + 3f_{52} + 2f_{53} + f_{54} + f_{56} + 2f_{57} + 3f_{58})
\]

or \( \Delta(4, 5) = (f_{41} + f_{42} + f_{43}) - (f_{46} + f_{47} + f_{48}) - (f_{51} + f_{52} + f_{53}) + (f_{56} + f_{57} + f_{58}) \) \( \) (9)

The sum of frequency terms in each parenthesis in the above equation can be expressed as the difference between two entries of matrix \( S \) as given below

\[
\Delta(4, 5) = s(4, 3) - \{s(4, 8) - s(4, 5)\} - s(5, 3) + \{s(5, 8) - s(5, 5)\}
\]

(10)
or

\[
\Delta(4, 5) = 10 - (17 - 12) - 5 + (14 - 7) \\
= 10 - 5 - 5 + 7 \\
= 7
\]  

(11)

Thus \( z(\Pi_{45}) = z(\Pi_c) + \Delta(4, 5) = 123 + 7 = 130 \).

### 4 Neighborhood Search algorithms

Neighborhood search based algorithms like local search, simulated annealing, and tabu search, are some of the widely used heuristic algorithms to solve hard problems. A local search algorithm and a tabu search algorithm, which incorporates the SWAP and INSERT neighborhood search procedures discussed in the above sections, were developed for the tool indexing problem. These are detailed in this section.

#### 4.1 Local search algorithm

Local search algorithm starts with an initial solution which is set as the current solution. The algorithm proceeds by searching the neighborhood of the current solution and choosing the best possible solution among the neighbors and marking that solution as the current solution. If the search fails to find any solution better than the current solution, the local search terminates, else the current solution is replaced by the best solution in the neighborhood and the search continues by exploring the new neighborhood. The current solution gets updated by increasingly better solutions as the search progresses and thus the current solution at the time when the local search terminates is taken as an approximation to the optimal solution to the problem. The Pseudo-code for the local search algorithm is given in Algorithm 3.

#### 4.2 Tabu search algorithm

Local search often gets trapped in inferior quality local optima and terminates, thereby resulting in a poor quality solution. Tabu search is a neighborhood search based meta-heuristic search technique equipped with features to steer local search away from local optima. Introduced and formalized in Glover (1989, 1990), tabu search is employed for solving computationally hard optimization problems. It has memory structures that allows worsening moves when the local search is stuck in
Algorithm 3: Pseudo-code for the local search algorithm

Input: Frequency matrix $F = [f(i,j)]$, $i,j \in \{1, \ldots, N\}$, distance matrix $D = [d(k,l)]$, $k,l \in \{1, \ldots, N\}$, starting solution $\Pi_c = [\pi(i)]$, cost function $z(.)$.

Output: $\Pi_{Best}$, a good solution to the tool indexing problem

1. $\Pi_{Best} \leftarrow \Pi_c$;
2. $\Pi_{Curr} \leftarrow \Pi_c$;
3. $Improving \leftarrow TRUE$;
4. while $Improving$ do
5.     $Improving \leftarrow FALSE$;
6.     foreach SWAP (INSERT) neighbour $\Pi_{p,q}$ of $\Pi_{Curr}$ do
7.         if $z(\Pi_{p,q}) < z(\Pi_{Best})$ then
8.             $\Pi_{Best} \leftarrow \Pi_{p,q}$;
9.             $Improving \leftarrow TRUE$
10.        end
11.     end
12.     $\Pi_{Curr} \leftarrow \Pi_{Best}$;
13. end
14. return $\Pi_{Best}$;

local optima and forbids certain moves that is likely to bring the search to already visited search spaces. In its basic form, tabu search has a list of forbidden moves called tabu list, the number of iterations for which the moves remain forbidden called the tabu tenure, an aspiration criterion that can override the tabu status of a move, and a termination criterion that determines when to end the tabu search. Tabu search implementation of Taillard (1991) is adapted for the indexing problem. Memory structures used in the standard tabu search algorithm followed by their adaptation and implementation details for swap and insert based tabu search have been described below.

**Tabu List** Tabu search works by maintaining a list of moves called a tabu list that aids in discouraging return to solutions recently chosen by the search. For this, tabu list should include the reverse of the moves executed by the search at the end of each iteration and forbid these moves in the subsequent iterations. If a move is present in the tabu list, the move is said to be tabu active.

Taillard (1991) defines tabu list as follows. Swap move $swap(p,q)$ is tabu if it assigns tool $\pi[p]$ as well as tool $\pi[q]$ to a position it has occupied in any of the last $s$ iterations. To implement this definition of tabu list, $(i,j)$ is defined as a move that places tool $i$ in position $j$. Given a solution represented by $\Pi$, swap move $swap(p,q)$ on $\Pi$ constitutes moving tool $\pi[p]$ to position $q$, and moving tool $\pi[q]$ to position $p$ and are indicated by $(\pi[p], q)$ and $(\pi[q], p)$, respectively. As per the definition of tabu list in Taillard (1990), The reverse of $swap(p,q)$ is composed of $(\pi[p], p)$
and \((\pi[q], q)\). Hence, if the SWAP neighbor chosen at the end of an iteration is \(\Pi_{p,q}\), then moves \((\pi[p], p)\) and \((\pi[q], q)\) are added to the tabu list. A swap move \(\text{swap}(p, q)\) is set as ‘tabu’ if both the moves \((\pi[p], q)\) and \((\pi[q], p)\) are tabu active.

In an insert move \(\text{insert}(p, q)\), the positions of \(|q-p|+1\) tools gets changed. Forbidding the reverse of all these moves might over constrain the search. Hence only two of these position changes; move of tool \(\pi[p+1]\) to position \(p\) and move of tool \(\pi[p]\) to position \(q\). The reverse of these moves are \((\pi[p], p-1)\) and \((\pi[q], p)\). However, forbidding one or both of these moves is not enough to prevent cycling back to earlier solutions. Hence, in addition to these moves, the cost of the solution is also added to the tabu list. Thus if either of the moves \((\pi[p], p-1)\) or \((\pi[q], p)\) of \(\text{insert}(p, q)\) are tabu active or if the cost of \(\Pi_{p,q}\) is tabu active, then the move \(\text{insert}(p, q)\) is marked tabu.

**Tabu Tenure** Tabu tenure (denoted as \(s\)) is the duration (in terms of the number of iterations) for which a move remains in the tabu list. At the end of iteration \(i\), moves that were added in iteration \(i-s\) or earlier, are removed from the tabu list. A smaller value may cause frequent cycling while a larger value may be too restrictive in exploring good quality solutions. Following Taillard (1991), tabu tenure \(s\) is kept as a random number between \(S_{\text{min}}\) and \(S_{\text{max}}\) and changes it after every \(2S_{\text{max}}\) iterations which is done in order to have some probability to perform some iterations with \(s = S_{\text{max}}\). Based on computational experiments, \(S_{\text{min}}\) and \(S_{\text{max}}\) were set as \(0.9N\) and as \(1.1N\) respectively for swap based tabu search and \(N\) and as \(3N\) respectively for insert based tabu search.

**Aspiration criterion** If a move marked as tabu produces a solution which meets a predetermined condition called the aspiration criteria, the tabu status of the move is overridden. The classical aspiration function overrides tabu status on a swap move or an insert move if it leads to a solution better than the best solution found so far. This has been implemented in the tabu search. Tabu status is also overridden for the best solution in the neighborhood when all the moves that constitute the neighborhood are tabu.

**Termination criterion** Tabu search is terminated when a pre-specified criterion such as a time limit, a maximum number of no improvement iterations, or a maximum number of search iterations is met. In the tabu search implementation, maximum number of iterations is used as the termination criteria and it is set to be \(N^2\).

The psuedocode for SWAP tabu search (\(\text{SwapTS}\)) and INSERT tabu search (\(\text{InsertTS}\)) developed based on the above description is given in Algorithm 4 and Algorithm 5, respectively.
Algorithm 4: Pseudo-code for swap based tabu search algorithm

**Input:** Frequency matrix $F = [f(i, j)], i, j \in \{1, \ldots, N\}$, distance matrix $D = [d(k, l)], k, l \in \{1, \ldots, N\}$, starting solution $\Pi_c = [\pi(i)]$, cost function $z(.)$.

**Output:** $\Pi_{Best}$, a good solution to the tool indexing problem

1. $\Pi_{Best} \leftarrow \Pi_c$;
2. $\Pi_{Curr} \leftarrow \Pi_c$;
3. $S_{min} \leftarrow 0.9N$;
4. $S_{max} \leftarrow 1.1N$;
5. MAXITER $\leftarrow N^2$;
6. tabulist $\leftarrow \emptyset$;
7. iter $\leftarrow 1$;
8. while iter $< MAXITER$ do
   9. if iter is equal to 1 or a multiple of $2S_{max}$ then
      | TABUTENURE $\leftarrow$ a random number between $S_{min}$ and $S_{max}$;
   10. end
   11. $\Pi_{BestNbr} \leftarrow \Pi_{Curr}$;
   12. foreach SWAP neighbour $\Pi_{p,q}$ of $\Pi_{Curr}$ do
      | moveAllowed $\leftarrow$ FALSE;
      | moveAspiring $\leftarrow$ FALSE;
      | if $(\pi[p], q) \notin$ tabulist AND $(\pi[q], p) \notin$ tabulist AND $z(\Pi_{p,q}) < z(\Pi_{BestNbr})$ then
      | | moveAllowed $\leftarrow$ TRUE;
      | end
      | if moveAllowed = FALSE AND $z(\Pi_{p,q}) < z(\Pi_{Best})$ then
      | | moveAspiring $\leftarrow$ TRUE;
      | end
      | if moveAllowed = TRUE OR moveAspiring = TRUE then
      | | $\Pi_{BestNbr} \leftarrow \Pi_{p,q}$;
      | | tabumove1 $\leftarrow (\pi[p], p)$;
      | | tabumove2 $\leftarrow (\pi[q], q)$;
      | end
   13. end
   14. $\Pi_{Curr} \leftarrow \Pi_{BestNbr}$;
   15. if $z(\Pi_{Curr}) < z(\Pi_{Best})$ then
      | $\Pi_{Best} \leftarrow \Pi_{Curr}$;
   16. end
   17. add tabumove1 and tabumove2 to tabulist for the next TABUTENURE iterations;
   18. iter $\leftarrow$ iter + 1;
19. end
20. return $\Pi_{Best}$;
5 Computational Experiments

The four algorithms described so far for the tool indexing problem: SWAP neighborhood search (SwapNS) and INSERT neighborhood search (InsertNS) based on Algorithm 3, and SWAP tabu search (SwapTS) and INSERT tabu search (InsertTS) based on Algorithm 4 and Algorithm 5, respectively, were coded in C++. For evaluating the performance of these algorithms, instances used in the literature for the tool indexing problem and related literature were identified. Based on the source from which the instances have been taken/adapted, these are classified as ‘single’, ‘bk’, ‘anjos’, and ‘sko’ instances. Since the instances in these sets are mostly small-sized and few in number, a set of large-sized random problem instances have also been generated called the ‘large’ instances for comparing the performance of our algorithms among themselves.

Each instance is denoted as ‘name-X-Y-Z’ where name is the instance set name (single/bk/anjos/sko), X is the number of slots assumed to be on the ATC, Y is the number of tools used for operations, and Z is the instance number that distinguishes instances with the same problem size (X and Y values are the same) but different frequency matrices. For instance, anjos-100-75-2 refers to the anjos instance with 100 slots and 75 tools, and frequency matrix denoted as 2. For each instance, an algorithm performed the search 51 times, each time with a different starting solution (Martı and Reinelt (2011)).

Against each of bk, sko, and anjos instances, the cost and time reported in Ghosh (2016) for their TS algorithm (which is a tabu search algorithm based on $O(n^3)$ delta computation technique for swap move costs) is also displayed. Wherever our algorithm found a better objective function value than that reported by the TS algorithm, the corresponding cost value is displayed in bold font.

5.1 Single instances

There is only one instance in the literature for the tool indexing problem without tool duplication. This is a small instance from Dereli and Filiz (2000) which has 10 tools and assumed to be used on a bidirectional ATC with 16 slots. They used a genetic algorithm and obtained a solution with objective function value of 13 rotations in 80 seconds. Our neighborhood search algorithms SwapNS and InsertNS arrived at the same cost function value of 13 in under 0.005 cpu seconds, and our tabu search algorithms hit this value within 0.160 cpu seconds. Two other instances were taken from Baykasoğlu and Dereli (2004) in which tool duplication is allowed on the tool magazine. One is a small instance with 8 tools and the other is a large instance with 50 tools. Number of slots on
the ATC is assumed to be 12 in the first example and 70 in the latter example. Since they have considered tool indexing problem with tool duplication, their objective function cannot be compared with ours. Performance of our algorithms against the single instances are given in Table 1.

Table 1: Performance comparison on single instances

<table>
<thead>
<tr>
<th>Instance</th>
<th>Our Swap Algorithms</th>
<th>Our Insert Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SwapNS</td>
<td>SwapTS</td>
</tr>
<tr>
<td></td>
<td>cost</td>
<td>time</td>
</tr>
<tr>
<td>single-16-10-01</td>
<td>13</td>
<td>0.003</td>
</tr>
<tr>
<td>single-12-8-01</td>
<td>53</td>
<td>0.002</td>
</tr>
<tr>
<td>single-70-50-01</td>
<td>404</td>
<td>1.058</td>
</tr>
</tbody>
</table>

5.2 BK instances

These are 30 instances from Baykasoğlu and Ozsoydan (2016) used for experiments on a version of the indexing problem in which duplication of tools in the tool magazine was allowed. They use three categories of benchmark problems: small, medium, and large. Their large category instances (bk-16-12-01 to bk-16-12-10) require 12 tools to be assigned to 16 slots. Each medium (bk-12-8-01 to bk-12-8-10) and small (bk-12-8-11 to bk-12-8-20) category instances have 8 tools to be assigned to 12 slots. Each instance in the paper corresponds to a cutting tool sequence that gives the order in which each tool has to be used. To render these instances suitable for the indexing problem without duplication, the tool sequence was converted into frequency matrix form. Table 2 presents the performance of TS algorithm and our swap and insert algorithms for these instances. Our algorithms were on average 2.4 to 97.3 times faster and hit the same cost when compared to the TS algorithm.

5.3 Anjos instances

These are instances are taken from Anjos et al. (2005) for the single row facility layout problem (SRFLP) and adapted for the tool indexing problem. Frequency matrix denoting the frequency of interaction between a pair of facilities in the SRFLP instance is copied as the frequency matrix denoting the number of times a pair of tools are used in consequent operations in the tool indexing problem instance. Number of slots is considered as 100 for the experiments. Results for anjos instances are presented in Table 3. Our InsertNS algorithm performed worse than TS algorithm in terms of cost in only 2 out of the 20 instances and was over 67 times faster than the latter. It
Table 2: Performance comparison on \(BK\) instances

<table>
<thead>
<tr>
<th>Instance</th>
<th>TS</th>
<th>Our Swap Algorithms</th>
<th>Our Insert Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cost</td>
<td>time</td>
<td>SwapNS</td>
</tr>
<tr>
<td>bk-12-8-01</td>
<td>48</td>
<td>0.268</td>
<td>48 0.003</td>
</tr>
<tr>
<td>bk-12-8-02</td>
<td>48</td>
<td>0.270</td>
<td>48 0.004</td>
</tr>
<tr>
<td>bk-12-8-03</td>
<td>50</td>
<td>0.266</td>
<td>50 0.003</td>
</tr>
<tr>
<td>bk-12-8-04</td>
<td>50</td>
<td>0.269</td>
<td>50 0.003</td>
</tr>
<tr>
<td>bk-12-8-05</td>
<td>48</td>
<td>0.272</td>
<td>48 0.002</td>
</tr>
<tr>
<td>bk-12-8-06</td>
<td>51</td>
<td>0.270</td>
<td>51 0.003</td>
</tr>
<tr>
<td>bk-12-8-07</td>
<td>54</td>
<td>0.273</td>
<td>54 0.003</td>
</tr>
<tr>
<td>bk-12-8-08</td>
<td>55</td>
<td>0.267</td>
<td>55 0.002</td>
</tr>
<tr>
<td>bk-12-8-09</td>
<td>46</td>
<td>0.267</td>
<td>46 0.004</td>
</tr>
<tr>
<td>bk-12-8-10</td>
<td>54</td>
<td>0.269</td>
<td>54 0.003</td>
</tr>
<tr>
<td>bk-12-8-11</td>
<td>19</td>
<td>0.270</td>
<td>19 0.002</td>
</tr>
<tr>
<td>bk-12-8-12</td>
<td>31</td>
<td>0.265</td>
<td>31 0.003</td>
</tr>
<tr>
<td>bk-12-8-13</td>
<td>25</td>
<td>0.267</td>
<td>25 0.002</td>
</tr>
<tr>
<td>bk-12-8-14</td>
<td>29</td>
<td>0.266</td>
<td>29 0.003</td>
</tr>
<tr>
<td>bk-12-8-15</td>
<td>38</td>
<td>0.264</td>
<td>38 0.002</td>
</tr>
<tr>
<td>bk-12-8-16</td>
<td>28</td>
<td>0.270</td>
<td>28 0.000</td>
</tr>
<tr>
<td>bk-12-8-17</td>
<td>24</td>
<td>0.266</td>
<td>24 0.003</td>
</tr>
<tr>
<td>bk-12-8-18</td>
<td>31</td>
<td>0.266</td>
<td>31 0.003</td>
</tr>
<tr>
<td>bk-12-8-19</td>
<td>31</td>
<td>0.267</td>
<td>31 0.001</td>
</tr>
<tr>
<td>bk-12-8-20</td>
<td>25</td>
<td>0.267</td>
<td>25 0.003</td>
</tr>
<tr>
<td>bk-16-12-01</td>
<td>117</td>
<td>0.580</td>
<td>117 0.007</td>
</tr>
<tr>
<td>bk-16-12-02</td>
<td>114</td>
<td>0.378</td>
<td>114 0.007</td>
</tr>
<tr>
<td>bk-16-12-03</td>
<td>118</td>
<td>0.584</td>
<td>118 0.009</td>
</tr>
<tr>
<td>bk-16-12-04</td>
<td>95</td>
<td>0.586</td>
<td>95 0.009</td>
</tr>
<tr>
<td>bk-16-12-05</td>
<td>121</td>
<td>0.582</td>
<td>121 0.005</td>
</tr>
<tr>
<td>bk-16-12-06</td>
<td>112</td>
<td>0.602</td>
<td>112 0.007</td>
</tr>
<tr>
<td>bk-16-12-07</td>
<td>107</td>
<td>0.591</td>
<td>107 0.005</td>
</tr>
<tr>
<td>bk-16-12-08</td>
<td>136</td>
<td>0.589</td>
<td>136 0.007</td>
</tr>
<tr>
<td>bk-16-12-09</td>
<td>136</td>
<td>0.624</td>
<td>136 0.005</td>
</tr>
<tr>
<td>bk-16-12-10</td>
<td>117</td>
<td>0.619</td>
<td>117 0.012</td>
</tr>
</tbody>
</table>

found cost better than that by \(TS\) algorithm in 10 out of the 20 \(anjos\) instances. On the other hand, our \(SwapNS\) algorithm output inferior cost in 13 and superior cost in 3 out of the 20 instances. Our algorithms were faster than the \(TS\) algorithm at the least by 2.5 times on average across \(anjos\) instances.

5.4 Sko instances

These instances are from Anjos and Yen (2009) used for SRFLP computational experiments. Duplicate problem instances with the same frequency matrix among these were discarded the remaining
Table 3: Performance comparison on *anjos* instances

<table>
<thead>
<tr>
<th>Instance</th>
<th>Our Swap Algorithms</th>
<th>Our Insert Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TS</td>
<td>SwapNS</td>
</tr>
<tr>
<td></td>
<td>cost time</td>
<td>cost time</td>
</tr>
<tr>
<td>anjos-100-60-1</td>
<td>54053 179.468</td>
<td>54053 2.543</td>
</tr>
<tr>
<td>anjos-100-60-2</td>
<td>31278 180.109</td>
<td>31285 3.302</td>
</tr>
<tr>
<td>anjos-100-60-3</td>
<td>23510 180.109</td>
<td>23514 2.779</td>
</tr>
<tr>
<td>anjos-100-60-4</td>
<td>11592 178.235</td>
<td>11610 2.907</td>
</tr>
<tr>
<td>anjos-100-60-5</td>
<td>15182 178.233</td>
<td>15181 3.414</td>
</tr>
<tr>
<td>anjos-100-70-1</td>
<td>42297 178.452</td>
<td>42370 4.105</td>
</tr>
<tr>
<td>anjos-100-70-2</td>
<td>51723 178.046</td>
<td>51723 4.210</td>
</tr>
<tr>
<td>anjos-100-70-3</td>
<td>43795 178.500</td>
<td>43820 3.363</td>
</tr>
<tr>
<td>anjos-100-70-4</td>
<td>27705 178.842</td>
<td>27733 24.523</td>
</tr>
<tr>
<td>anjos-100-70-5</td>
<td>134269 181.483</td>
<td>134348 4.666</td>
</tr>
<tr>
<td>anjos-100-75-1</td>
<td>66656 182.202</td>
<td>66658 4.455</td>
</tr>
<tr>
<td>anjos-100-75-2</td>
<td>111806 181.454</td>
<td>111806 5.480</td>
</tr>
<tr>
<td>anjos-100-75-3</td>
<td>38153 28.743</td>
<td>38153 4.95</td>
</tr>
<tr>
<td>anjos-100-75-4</td>
<td>106341 29.095</td>
<td>106341 4.314</td>
</tr>
<tr>
<td>anjos-100-75-5</td>
<td>47032 28.451</td>
<td>47032 4.517</td>
</tr>
<tr>
<td>anjos-100-80-1</td>
<td>54469 37.118</td>
<td>54469 4.160</td>
</tr>
<tr>
<td>anjos-100-80-2</td>
<td>52866 40.211</td>
<td>52866 3.805</td>
</tr>
<tr>
<td>anjos-100-80-3</td>
<td>95133 36.115</td>
<td>95133 4.603</td>
</tr>
<tr>
<td>anjos-100-80-4</td>
<td>100845 36.996</td>
<td>100845 4.039</td>
</tr>
<tr>
<td>anjos-100-80-5</td>
<td>36249 36.115</td>
<td>36249 4.517</td>
</tr>
</tbody>
</table>

seven instances were used for our experiments. To allow for comparison with the *TS* algorithm, a tool magazine with 100 slots was used when the number of tools were more than 60 and 60 slots otherwise. Table 4 presents the results for these instances. *InsertTS* algorithm beat *TS* algorithm in six out of the seven instances. Our Neighborhood Search algorithms and Tabu Search algorithms were faster than *TS* algorithm 57 times and 2.3 times on average, respectively.

### 5.5 Large instances

Since sufficient number of large-sized instances could not be found from the literature for comparing our algorithm performances with one another, a set of problem instances were created by randomly generating frequency matrices for three different problem sizes; \( N = 50 \), \( N = 75 \), and \( N = 100 \). For each problem size, 100 frequency matrices numbered 1 to 100 were generated such that the entries of matrices 1-25 and 51-75 take integer values between and including 0 and 20, whereas those in matrices 26-50 and 76-100 take integer values between and including 0 and 10. Further, matrices 51-100 were made sparse. The instances with frequency matrices numbered 1-25, 26-50, 51-75, and 76-100 were named as ‘large(A)’, ‘large(B)’, ‘large(C)’, and ‘large(D)’, respectively. Ta-
Table 4: Performance comparison on sko instances

<table>
<thead>
<tr>
<th>Instance</th>
<th>TS Swap Algorithms</th>
<th>Our Insert Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SwapNS</td>
<td>SwapTS</td>
</tr>
<tr>
<td></td>
<td>cost</td>
<td>time</td>
</tr>
<tr>
<td>sko-60-42-1</td>
<td>24408</td>
<td>38.281</td>
</tr>
<tr>
<td>sko-60-49-1</td>
<td>36582</td>
<td>38.078</td>
</tr>
<tr>
<td>sko-60-56-1</td>
<td>52974</td>
<td>38.030</td>
</tr>
<tr>
<td>sko-100-64-1</td>
<td>95337</td>
<td>182.062</td>
</tr>
<tr>
<td>sko-100-72-1</td>
<td>133607</td>
<td>182.561</td>
</tr>
<tr>
<td>sko-100-81-1</td>
<td>185852</td>
<td>183.078</td>
</tr>
<tr>
<td>sko-100-100-1</td>
<td>290496</td>
<td>185.000</td>
</tr>
</tbody>
</table>

Table 5 summarizes the performance of our algorithms on large instances. Tables 5a contains the summary of average cost reported by each algorithm for ‘large(A)’, ‘large(B)’, ‘large(C)’, and ‘large(D)’ instances while 5c displays the average cost reported by each algorithm for different problem sizes. The solution output by each algorithm is ranked and the summary of average rank for each instance type and each problem size is presented in Tables 5b and 5d, respectively.

Table 5: Performance summary on large instances

(a) Average cost for each instance type

<table>
<thead>
<tr>
<th>Instance type</th>
<th>Our Swap Algorithms</th>
<th>Our Insert Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SwapNS</td>
<td>SwapTS</td>
</tr>
<tr>
<td>largeA</td>
<td>430016.69</td>
<td>430003.11</td>
</tr>
<tr>
<td>largeB</td>
<td>202642.78</td>
<td>202633.36</td>
</tr>
<tr>
<td>largeC</td>
<td>101813.34</td>
<td>101799.41</td>
</tr>
<tr>
<td>largeD</td>
<td>47852.77</td>
<td>47836.82</td>
</tr>
</tbody>
</table>

(b) Average rank for each instance type

<table>
<thead>
<tr>
<th>Instance type</th>
<th>Our Swap Algorithms</th>
<th>Our Insert Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SwapNS</td>
<td>SwapTS</td>
</tr>
<tr>
<td>largeA</td>
<td>4.63</td>
<td>4.35</td>
</tr>
<tr>
<td>largeB</td>
<td>4.75</td>
<td>4.38</td>
</tr>
<tr>
<td>largeC</td>
<td>4.82</td>
<td>4.59</td>
</tr>
<tr>
<td>largeD</td>
<td>5.02</td>
<td>4.64</td>
</tr>
</tbody>
</table>

(c) Average cost for each problem size

<table>
<thead>
<tr>
<th>Problem size</th>
<th>Our Swap Algorithms</th>
<th>Our Insert Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SwapNS</td>
<td>SwapTS</td>
</tr>
<tr>
<td>50</td>
<td>59531.89</td>
<td>59520.46</td>
</tr>
<tr>
<td>75</td>
<td>206997.37</td>
<td>206987.64</td>
</tr>
<tr>
<td>100</td>
<td>499933.50</td>
<td>499912.06</td>
</tr>
</tbody>
</table>

(d) Average rank for each problem size

<table>
<thead>
<tr>
<th>Problem size</th>
<th>Our Swap Algorithms</th>
<th>Our Insert Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SwapNS</td>
<td>SwapTS</td>
</tr>
<tr>
<td>50</td>
<td>4.88</td>
<td>4.43</td>
</tr>
<tr>
<td>75</td>
<td>4.92</td>
<td>4.63</td>
</tr>
<tr>
<td>100</td>
<td>4.87</td>
<td>4.54</td>
</tr>
</tbody>
</table>

Our insert algorithms outperform their swap counterparts and the performance difference is more pronounced in problem sizes where number of tools is 75 or more. The same is also true when frequency matrices are sparse (‘large(C)’, and ‘large(D)’ instances). The best performing algorithm of all is InsertTS as it ranks better on average than other algorithms in all categories.
6 Conclusion

Tool indexing problem is an important layout problem in automated machining centers of production facilities and is described in Section 1. Section 2 explains how the problem can be formulated as a Quadratic Assignment Problem and discusses the literature on it. Section 3 describes how the problem characteristics can be exploited to develop efficient swap and INSERT neighborhood search procedures for the problem that bring down the search complexity from $O(N^4)$ time to $O(N^2)$ time, which is the best that can be achieved theoretically. A local search algorithm and a tabu search algorithm is designed for the problem based on our neighborhood search procedures and their implementation details are provided in Section 4. Based on computational experiments conducted using several instances from the literature, the results presented in Section 5 indicate that our algorithms are competent.
Algorithm 5: Pseudo-code for insert based tabu search algorithm

Input: Frequency matrix $F = [f(i,j)], i, j \in \{1, \ldots, N\}$, distance matrix $D = [d(k,l)], k, l \in \{1, \ldots, N\}$, starting solution $\Pi_c = [\pi(i)]$, cost function $z(.)$.

Output: $\Pi_{Best}$, a good solution to the tool indexing problem

1 $\Pi_{Best} \leftarrow \Pi_c$ ;
2 $\Pi_{Curr} \leftarrow \Pi_c$ ;
3 $S_{min} \leftarrow N$ ;
4 $S_{max} \leftarrow 3N$ ;
5 $MAXITER \leftarrow N^2$ ;
6 $\text{tabulist} \leftarrow \emptyset$ ;
7 $\text{iter} \leftarrow 1$ ;
8 while $\text{iter} < \text{MAXITER}$ do
9     if $\text{iter}$ is equal to 1 or a multiple of $2S_{max}$ then
10        $\text{TABUTENURE} \leftarrow$ a random number between $S_{min}$ and $S_{max}$ ;
11     end
12     $\Pi_{BestNbr} \leftarrow \Pi_{Curr}$ ;
13     foreach INSERT neighbour $\Pi_{p,q}$ of $\Pi_{Curr}$ do
14        $\text{moveAllowed} \leftarrow \text{FALSE}$ ;
15        $\text{moveAspiring} \leftarrow \text{FALSE}$ ;
16        if $(\pi[p],q) \notin \text{tabulist}$ AND $(\pi[q],p) \notin \text{tabulist}$ AND $z(\Pi_{p,q}) \notin \text{tabulist}$ AND
17            $z(\Pi_{p,q}) < z(\Pi_{BestNbr})$ then
18            $\text{moveAllowed} \leftarrow \text{TRUE}$ ;
19        end
20        if $\text{moveAllowed} = \text{FALSE}$ AND $z(\Pi_{p,q}) < z(\Pi_{Best})$ then
21            $\text{moveAspiring} \leftarrow \text{TRUE}$ ;
22        end
23        if $\text{moveAllowed} = \text{TRUE}$ OR $\text{moveAspiring} = \text{TRUE}$ then
24           $\Pi_{BestNbr} \leftarrow \Pi_{p,q}$ ;
25           $\text{tabumove1} \leftarrow (\pi[p], p - 1)$ ;
26           $\text{tabumove2} \leftarrow (\pi[p], p + 1)$ ;
27           $\text{tabucost} \leftarrow z(\Pi_{p,q})$ ;
28        end
29     end
30     $\Pi_{Curr} \leftarrow \Pi_{BestNbr}$ ;
31     if $z(\Pi_{Curr}) < z(\Pi_{Best})$ then
32        $\Pi_{Best} \leftarrow \Pi_{Curr}$ ;
33     end
34     add $\text{tabumove1, tabumove2, tabucost}$ to $\text{tabulist}$ for next $\text{TABUTENURE}$ iterations ;
35     $\text{iter} \leftarrow \text{iter} + 1$ ;
36 end
37 return $\Pi_{Best}$ ;
References


