Migration from Sequence to Schedule in Total Earliness and Tardiness Scheduling Problem

Mohammad Namakshenas · Mohammad Mahdavi Mazdeh

Abstract To be competitive, services must be delivered with high punctuality. To schedule punctual services, the classical scheduling theory offers to minimize total earliness and tardiness of services. In this study, we developed an optimal fully polynomial time algorithm to transform a given sequence, permutation of jobs, into its minimum cost schedule counterpart, timing of jobs, for the single machine scheduling with the objective total earliness and tardiness. We provide the fundamental necessary and sufficient optimal properties for the objective of sub-sequences of jobs, called clusters. The algorithm first decomposes a sequence into several clusters, and then it applies a recursive scheme to join the clusters and to generate their minimum cost schedule counterparts. The complexity status sheds light on the competitiveness of our algorithm in comparison with other state-of-the-art algorithms in the literature.

Keywords Scheduling · Single machine · Total earliness and tardiness · Exact algorithm

1 Introduction

The total tardiness and earliness scheduling problem has been receiving significant attention in the literature. One reason is a great variety products and services should be produced or delivered at exactly the right time. Thus jobs are expected to be punctual as well as early delivery is measured negatively. The total tardiness and earliness scheduling problem is essentially motivated by the Just-In-Time (JIT) production system in which all jobs’ ideal completion times are exactly at their due dates. The formal description of the problem is as follows. Consider a set $J = \{1, \ldots, n\}$ of independent jobs have processing times, $p_j$, $\forall j \in J$, that...

M. Namakshenas
School of Industrial Engineering, Iran University of Science and Technology, Tehran, Iran
E-mail: m_namakshenas@ind.iust.ac.ir

M. Mahdavi Mazdeh
School of Industrial Engineering, Iran University of Science and Technology, Tehran, Iran
E-mail: mazdeh@iust.ac.ir
must be processed on a single machine. Each job \( j \in J \) is also characterized by its distinct due date \( d_j \). All jobs are ready at time zero and a job cannot be preempted once started. The objective is to decide the optimal start times, \( s_j \), \( \forall j \in J \), which minimize the total earliness and tardiness criteria or equivalently the total absolute deviation of \( s_j + p_j \), \( \forall j \in J \), from \( d_j \), \( \forall j \in J \). According to the 3-field notation, we identify the problem as 1||\( \sum_{j \in J} E_j + \sum_{j \in J} T_j \) in which \( E_j = \max \{0, d_j - (s_j + p_j)\} \) and \( T_j = \max \{0, (s_j + p_j) - d_j\} \), \( \forall j \in J \). For the notational simplicity, we abbreviate term \( \sum_{j \in J} E_j + \sum_{j \in J} T_j \) as TE. The problem with equal processing times, \( p_j = p \), \( \forall j \in J \), is also characterized as 1||\( p \) | TE.

The problem with distinct processing times 1||TE is hard to solve: Wan and Yuan [15] proved that problem 1||TE is NP-hard in the strong sense (NPₜₐₜ) by a unary pseudo-polynomial reduction from the 3-Partition problem. In contrast to regular objective functions, non-decreasing in \( s_j \), the non-regular objective TE has an additional layer of complexity, i.e., finding the optimal position of idle times among jobs. Conway et al. [1] showed that there exists an optimal schedule without having idle times among jobs for any scheduling problem in form 1 | \( r_j = 0 \) | regular. In the theory of scheduling, the term sequencing is often used to define the permutation of jobs while the term scheduling refers to the timing which is usually described by the start times of jobs. Hence, an optimal schedule generates its optimal sequence counterpart, but not necessarily the other way around. We put forth two valid questions on scheduling classes 1||regular and 1||non-regular:

1. Can an optimal sequence and an optimal schedule be computed in separate, possibly (pseudo)-polynomial, oracles?
2. Is it possible to (pseudo)-polynomially compute a minimum cost schedule given an arbitrary sequence?

We postulate that a “YES” answer to the first question should pave promising ways to develop completely new and efficient algorithms since it offers a decomposable framework. We will show the answer to the both is indeed positive for 1 | \( p_j = p \) | TE; however, we leave them as general open questions for the single machine literature.

We refer to the work of Garey et al. [5] as a serious attempt to develop a parsimonious and polynomial algorithm. The algorithm transforms a given sequence into a minimum cost schedule in \( O(n \log(n)) \) for 1||TE (a “YES” answer to the second question for 1||TE). Although the algorithm is robust and fast; however, its expressiveness, mainly due to its dependence on the traditional tree-based heap data structure [2], leads to a verbose algorithm. They proved that 1||TE is NP-hard in the ordinary sense (NPₜₐₜ) by reduction from 2-Partition; the explicit result remains consistent with the strongly NP-hardness of the problem [15] according to the complexity hierarchy, i.e., \( \text{NP} = \text{NP} \subseteq \text{NP} \subseteq \text{NP} \subseteq \text{NP} \neq \text{NP} \) [4].

Developing a time-tabling scheme and embedding it into an enumerative search algorithm was another significant work conducted by Davis and Kanet [3]. Their algorithm prunes the permutation space by detecting semi-active schedules, i.e., no shift applied on any job of a schedule can improve the objective value. They proved that any semi-active schedule dominates the set of all schedules for 1||TE, thereby produces a minimum cost schedule. This property was essentially helpful to reduce the permutation space as much as possible; however, they neither analyzed nor reported the exact worst-case running time of the algorithm.
Szwarc and Mukhopadhyay [11] developed a fundamentally new algorithm on top of a property in which a given sequence is divided into a number of sub-sequences, called clusters. The main feature of their algorithm is that the durations of idle times between the detected clusters remain unchanged, thereby increasing its efficiency. Nevertheless, this idea has a downside so that to preserve a schedule as semi-active, all start times should be pushed forward in every iteration. Our proposed algorithm only updates the necessary changes on the current cluster. The worst-case running time of Szwarc and Mukhopadhyay [11] algorithm is the polynomial function of number of clusters and jobs, which practically outperforms the $O(n \log(n))$ running time of [5] algorithm in most computational cases.

It might be helpful to look into some other researches developed lower bounds and greedy rules. For example, Kim and Yano [7] proved some dominance properties of problem $1 \mid \text{TE} \mid \text{min}$ in which two overlapping jobs finished at their distinct due dates. They developed a conflict resolution procedure for overlapping jobs to obtain tight lower bounds. The problem with equal processing times was investigated by Verma and Dessouky [14]; they formulated it as a time-indexed integer problem. They tightened the model by propagating some dominance properties on the start time of each job; hence, they reduced the scheduling domain.

A body of researches has avoided considering idle times either by restricting problems into non-idle schedules or a common due dates for all jobs (see for example [6], [8], [10], [13], [12], and [16]). Mason et al. [8] used a pair-wise interchange procedure to improve their initial heuristic solutions and Tanaka et al. [12] developed a dynamic programming procedure to efficiently schedule a set of jobs in a reasonable amount of time. The complexity analyses of both works were not reported.

The following lines provide a summary of contributions in this study. We develop a new compact algorithm whose time complexity only polynomially depends on the number of jobs and clusters to transform an input sequence into the minimum cost schedule for $1 \mid \text{TE} \mid \text{min}$. Our proposed algorithm is compact since it manipulates the algorithmic verbosity into single lemma, i.e., an optimality condition of a set function on $\text{TE}$ over a cluster of jobs. Our proposed algorithm treats the schedule of only one cluster regardless of the schedules of other clusters; hence, it is not necessary to push the entire schedule like in the algorithm of Szwarc and Mukhopadhyay [11].

2 Algorithm Design

2.1 Case 1. Distinct Processing Times

**Lemma 1** Given a fixed sequence $\langle 1, 2, \ldots, n \rangle$, there exists an optimal schedule in which no idle time is placed between jobs $j$ and $j + 1$ if $d_{j+1} - d_j \leq p_{j+1}$.

**Proof** Refer to Szwarc and Mukhopadhyay [11].

**Definition 1** (First-order cluster) A partial sequence $\pi$ is called a First-order Cluster if $d_{j+1} - d_j \leq p_{j+1}$, $\forall j \in \pi$.

**Definition 2** (Semi-active schedule) A schedule is said to be semi-active if no shift of a job provides a lower cost schedule without changing any job sequence.
In what follows, we assume that a sequence of jobs is given as an input and any changes in the sequence is not permitted. We denote \( \pi_c = (1, \ldots, n_c) \) sequence of cluster \( c \) or partial sequence \( c \), in which index \( c \) indicates the number of cluster initiating from cluster set \( C \), i.e., \( \forall c \in C = \{1, \ldots, \phi \} \) and \( \max \{ C \} = n \). Hence, \( \langle \pi_1, \ldots, \pi_\phi \rangle \) forms a complete sequence. We also denote \( s_j \in \pi_c \) as the start time of job \( j \) of cluster \( \pi_c \) and \( \text{TE}(\pi_c) \) as the objective evaluated for \( \pi_c \). The following lemma shows the pivotal step in the transition from a sequence to a schedule of a cluster.

**Lemma 2** There exists a semi-active schedule for cluster \( \pi_c \) if

\[
\text{med}\{A\} \quad \text{where } med\{A\} \text{ is the median of set } A.
\]

**Proof** From the definition of a cluster, no idle time between any consecutive jobs in \( \pi_c \) are allowed. Hence, \( \text{TE}(\pi_c) \) can be redefined based on any job in \( \pi_c \), e.g., job 1, and it follows that \( s_j \leftarrow s_1 + \sum_{j'=1}^{j-1} p_{j'} \), \( \forall j \in \pi_c \setminus \{1\} \). Moreover, we obtain \( \text{TE}(\pi_c) = \sum_{j \in \pi_c} |s_1 - (d_j - \sum_{j'=1}^j p_{j'})| \). It directly follows that \( \text{TE}(\pi_c) \) is a 1-norm function with single variable \( s_1 \), and the median of set \( \{d_j - \sum_{j'=1}^j p_{j'}, \; j \in \pi_c\} \) necessarily minimizes \( \text{TE}(\pi_c) \) [9] (i). For checking the sufficiency, let \( S = \{s_1 - (d_j - \sum_{j'=1}^j p_{j'}), \; j \in \pi_c\} \) and set function \( f : 2^{|S|} \rightarrow \mathbb{R}_+ \) such that \( f(\emptyset) = 0 \). We deduce that \( f = \sum_{\xi \in S} |\xi| \) over set \( S \) holds the following property,

\[
f(M_1) + f(M_2) \geq f(M_1 \cap M_2) + f(M_1 \cup M_2), \; \forall M_1, M_2 \subseteq S,
\]

since the left and the right hand sides contain the independent and non-separable modules. Hence, \( f \) is sub-modular (ii). From (i) and (ii), the results follow directly.

\( \square \)

According to the definition of a cluster and Lemma 2, the start time of other jobs in \( \pi_c \) is immediately computed as \( s_j \in \pi_c \leftarrow s_1 + \sum_{j' \in \pi_c} p_{j'} \), \( \forall j \in \pi_c \). Algorithm 1 reproduces the core result of Lemma 2 in function \( \text{fix\_start}(\pi_c) \).

**Algorithm 1** \( \text{fix\_start}(\pi_c) \): Applying Lemma 2

1: **Input:** \( \pi_c \)

2: \( s_1 \in \pi_c \leftarrow \max \left\{ 0, \text{med}\left\{d_j - \sum_{j'=1}^j p_{j'}, \; j \in \pi_c\right\} \right\} \)

3: \( s_n \in \pi_c \leftarrow s_1 + \sum_{j \in \pi_c \setminus \{n_c\}} p_j \)

4: **return** \( s_1 \in \pi_c, s_n \in \pi_c \)

**Lemma 3** (Second-order cluster) Consider two cluster \( \pi_c \) and \( \pi_{c+1} \) to which Lemma 2 are applied separately and some jobs of \( \pi_c \) are overlapping and some jobs of
\(\pi_{c+1}\), i.e., \(s_{n_c} + p_{n_c} \geq s_1\). There exists a semi-active schedule if two cluster form a new cluster \(\pi' = (\pi_c, \pi_{c+1})\) with \(s_1 \in \pi' \leftarrow \max \{0, \text{med} \{d_{j} - \sum_{j'=1}^{j} p_{j'}, j \in \pi'\}\}\).

We shall call \(\pi'\) a Second-order cluster.

Proof To preserve the order of the input sequence, the final partial schedule should process cluster \(\pi_c\) followed by cluster \(\pi_{c+1}\). It directly follows that any idle time between two clusters does not reduce TE. Hence, joining two clusters in \(\pi' \leftarrow (\pi_c, \pi_{c+1})\) and applying Lemma 2 on \(\pi'\) yield a semi-active schedule.

Lemma 3 immediately ensures that all partial schedules from clusters remain semi-active in the following recursive scheme. In what follows, we construct an algorithm with the least TE based on the results obtained from Lemmas 1 through 3. Algorithm 2 consists of two primary routines, i.e., the clustering routine (CLR) and the schedule-correcting routine (SCR). The SCR, which starts from while loop, first checks whether the currently corrected cluster is overlapping with the previous or the next cluster. If such a case happens, it joins two clusters, according to Lemma 3 and the product is a new cluster. In the next step, it checks any infeasibility caused by the overlapping jobs inside the new cluster and then produces a semi-active schedule counterpart. Hence, the algorithm runs recursively in the backward or the forward direction until the condition \(c \leq \phi\) is not satisfied in the while loop. Algorithm 2 does not distinguish the first and second-order clusters.

Function update list\((\pi, \phi)\) simply refers to performing push operations on items of a list. Precisely, if \(\phi\) reduces to \(\phi - 1\) and the SCR forces \(\pi_c \leftarrow (\pi_c, \pi_{c+1})\) and \(C \leftarrow \{1, \ldots, \phi - 1\}\) operations, then update list\((\pi, \phi)\) re-index the clusters since \(\pi_{c+1}\) is redundant. This operation can be efficiently done in \(O(1)\).

Lemma 4 Every semi-active schedule is optimal for \(1 || TE\).

Proof Refer to Davis and Kanet [3].

Corollary 1 Given an arbitrary sequence, Algorithm 2 yields the minimum cost schedule for \(1 || TE\).

Proof The outputs of the algorithm are the number of clusters and the corresponding schedule to each cluster. Recall that the number of possible clusters can be 1, \(n\), and \(1 < \phi < n\). Suppose that \(\phi = n\), the simplest case, then the only way that the algorithm produces \(n\) clusters is \(d_i - d_i > p, \forall i, j \in J\). If such a case happens, the algorithm outputs \(s_j \leftarrow d_j - p, j \in J\) without entering into the SCR. For case \(\phi < n\), according to Lemma 2 and 3, the SCR produces the final clusters both feasible and semi-active. According to Lemma 4, such a schedule is also globally optimal for \(1 || TE\).

Corollary 2 Algorithm 2 generates the minimum cost schedule for \(1 || TE\) in worst-case running time of \(O(\phi n)\).

Proof It is immediate the CLR is performed in a separate \(O(n)\) iteration. The schedule correcting routine escalates to the worst-case scenario when the maximum number of the median function in \(O(n)\) is evaluated. In the worst-case, the algorithm computes the median function \(2\phi - 1\) times to generate the final semi-active schedule. The results follow directly.
Algorithm 2 Recursive scheme to generate minimum cost schedule for $1 \mid || \text{TE}$

1: Input: sequence re-indexed in a numerically increasing order
2: for $c \in J$ do
3: \hspace{1em} $\pi_c \leftarrow \langle \emptyset \rangle$
4: \hspace{1em} $c \leftarrow 1$
5: for $j \in J$ do
6: \hspace{2em} if $\pi_c := \langle \emptyset \rangle$ and $d_{j+1} - d_j \leq p_{j+1}$ then
7: \hspace{3em} $\pi_c \leftarrow \langle \pi_c, j, j+1 \rangle$
8: \hspace{2em} else if $d_{j+1} - d_j \leq p_{j+1}$ then
9: \hspace{3em} $\pi_c \leftarrow \langle \pi_c, j+1 \rangle$
10: \hspace{2em} else
11: \hspace{3em} $n_c \leftarrow |\pi_c|; c \leftarrow c + 1$
12: \hspace{1em} $\phi \leftarrow c; c \leftarrow 1$
13: for $c \leq \phi$ do
14: \hspace{2em} $s_1 \in \pi_c, s_n \in \pi_c \leftarrow \text{fix-start}(\pi_c)$
15: \hspace{2em} while $c \leq \phi$ do
16: \hspace{3em} if $c < \phi$ and $s_1 \in \pi_{c+1} - s_n \in \pi_c \leq \sum_{j \in \pi_c} p_j$ then
17: \hspace{4em} $\pi_c \leftarrow \langle \pi_c, \pi_{c+1} \rangle$
18: \hspace{4em} $\phi \leftarrow \phi - 1$
19: \hspace{4em} update_list($\langle \pi_1, \ldots, \pi_\phi \rangle \cup \phi$
20: \hspace{4em} if $s_n \in \pi_c - s_1 \in \pi_c < \sum_{j \in \pi_c \setminus \{n_c\}} p_j$ then
21: \hspace{5em} $s_1 \in \pi_c, s_n \in \pi_c \leftarrow \text{fix-start}(\pi_c)$
22: \hspace{2em} else if $c > 1$ and $s_1 \in \pi_c - s_n \in \pi_{c-1} \leq \sum_{j \in \pi_{c-1}} p_j$ then
23: \hspace{3em} $\pi_{c-1} \leftarrow \langle \pi_{c-1}, \pi_c \rangle$
24: \hspace{3em} $c \leftarrow c - 1; \phi \leftarrow \phi - 1$
25: \hspace{3em} update_list($\langle \pi_1, \ldots, \pi_\phi \rangle \cup \phi$
26: \hspace{3em} if $s_n \in \pi_c - s_1 \in \pi_c < \sum_{j \in \pi_c \setminus \{n_c\}} p_j$ then
27: \hspace{4em} $s_1 \in \pi_c, s_n \in \pi_c \leftarrow \text{fix-start}(\pi_c)$
28: \hspace{2em} else
29: \hspace{3em} $c \leftarrow c + 1$
30: \hspace{1em} $C \leftarrow \{1, \ldots, \phi\}$
31: return $\{s_1 \in \pi_c, \forall c \in C\}$

2.2 Case 2. Equal Processing Times

In this section, we settle on a “YES” answer to the first question raised in the Section 1 by proving an optimal sequencing rule for $1 \mid p_j = p \mid \text{TE}$. Then, we revise the results of the previous subsection to develop a faster algorithm than Algorithm 2 for $1 \mid p_j = p \mid \text{TE}$.

Lemma 5 There exists an optimal schedule for $1 \mid p_j = p \mid \text{TE}$, in which the jobs are ordered according to the Earliest Due Date (EDD) rule in which job $j$ follows job $i$ if $d_i \leq d_j$.

Proof Proof by contradiction. Suppose schedule $\pi$, which is not ordered w.r.t. EDD, is optimal. There must be two adjacent jobs $(j, i) \in \pi$, say job $j$ followed by job $i$, hence $s_i > s_j$, such that $d_i \leq d_j$. We perform a standard pairwise interchange on
jobs \( j \) and \( i \) and call the new schedule \( \pi' \) in which \( \langle i, j \rangle \in \pi' \). It is immediate that \( \pi' \) is feasible, since jobs \( i \) and \( j \) have equal processing times and do not overlap under this interchange. Exactly six different interchange cases are constructed based on the relative ordering of jobs \( i \) and \( j \) among \( d_i \) and \( d_j \). It can be easily verified that \( TE(\pi) = TE(\pi') \) for cases \( s_j < s_i \leq d_i < d_j \) and \( d_i < s_j \leq s_i \leq d_j \), and \( s_j \leq d_i < d_j \leq s_i \). This contradicts the optimality of \( \pi \) and completes the proof.

Without loss of generality, we re-index jobs of a cluster w.r.t. EDD rule in a numerically increasing order.

**Corollary 3** Given a first-order cluster \( \pi_c \) ordered by EDD, there exists a semi-active schedule for \( \pi_c \) if

\[
\begin{align*}
  s_{1 \in \pi_c} &\leftarrow \max\left\{0, (\gamma_1 + \gamma_2)/2\right\}, \\
  \gamma_1 &\leftarrow d_{\left\lceil (n_c+1)/2 \right\rceil} - \left\lceil (n_c + 1)/2 \right\rceil p, \\
  \gamma_2 &\leftarrow d_{\left\lfloor (n_c+1)/2 \right\rfloor} - \left\lfloor (n_c + 1)/2 \right\rfloor p.
\end{align*}
\]

(1)

**Proof** From Lemma 2, we have

\[
  s_{1 \in \pi_c} \leftarrow \max\left\{0, \text{med}\{d_j - jp, j \in \pi_c\}\right\},
\]

and it directly follows that set \( \{d_j -jp, j \in \pi_c\} \) is non-increasing since \( \pi_c \) preserves both the EDD rule and Lemma 1.

Algorithm 3 is practically more efficient than Algorithm 2 since it distinguishes the first-order clusters by plugging Corollary 3 in \( O(1) \) iteration into the CLR. In other words, it fixes the start times of jobs in the first-order clusters before entering into the SCR.

**Corollary 4** Given a sequence sorted by EDD, Algorithm 3 yields an optimal schedule for \( 1 \mid p_j = p \mid TE \).

**Proof** It is immediate that Algorithm 3 preserves the input sequence, which is ordered by EDD rule, to the end. The rest of proof is trivial from Corollary 1.

**Corollary 5** Algorithm 2 generates the minimum cost schedule for problem \( 1 \mid p_j = p \mid TE \) in worst-case running time of \( O(\phi_1 n - \phi_1^2/2 + \phi_1/2) \) in which \( \phi_1 \) is the number of the first-order clusters.

**Proof** Like Algorithm 2, this algorithm handles the CLR in a separate \( O(n) \) oracle. Again, the worst-case scenario for the SCR happens when the maximum number of the median function is evaluated in a \( O(n) \) oracle. Recall that the CLR partitions all jobs into \( \phi_1 \) clusters of \( n_c \) jobs. Without loss of generality, consider a case in which the first and the second first-order clusters are overlapping. After correction, it overlaps with the third first-order cluster. This continues to the \( \phi_1 \)th cluster. Hence, the algorithm needs to evaluate the median of \( (n_1 + n_2), (n_1 + n_2 + n_3), \ldots, \sum_{c=1}^{\phi_1} n_c \) in the SCR. Hence, the worst-case running time is computed as \( O(\phi_1 n - \phi_1^2/2 + \phi_1/2) \).
Algorithm 3 Recursive scheme to generate minimum cost schedule for 1 | pj = p | TE

1: **Input:** list of sorted \{s_j ← d_j, ∀ j ∈ J\} based on EDD
2: for c ∈ J do
3: \( \pi_c ← \langle ∅ \rangle \)
4: c ← 1
5: for j ∈ J do
6: if \( \pi_c := \langle ∅ \rangle \) and \( d_{j+1} - d_j ≤ p \) then
7: \( \pi_c ← \langle \pi_c, j, j + 1 \rangle \)
8: else if \( d_{j+1} - d_j ≤ p \) then
9: \( \pi_c ← \langle \pi_c, j + 1 \rangle \)
10: else
11: \( n_c ← |\pi_c| \)
12: \( s_{1\in\pi_c} ← \max \left\{ 0, (\gamma_1 + \gamma_2)/2 \right\} \) ▶ \( \gamma_1 \) and \( \gamma_2 \) are computed w.r.t. Corollary 3.
13: c ← c + 1
14: \( \phi_1 ← c ; c ← 1 \)
15: while c ≤ \( \phi_1 \) do
16: if \( s_{n_c\in\pi_c} - s_{1\in\pi_c} < (n_c - 1)p \) then
17: \( s_{1\in\pi_c} ← \max \left\{ 0, \text{med}\{d_j - jp, j ∈ \pi_c\} \right\} \)
18: \( s_{n_c\in\pi_c} ← s_{1\in\pi_c} + (n_c - 1)p \)
19: if c < \( \phi_1 \) and \( s_{1\in\pi_c} - s_{n_c\in\pi_c} ≤ n_c p \) then
20: \( \pi_c ← \langle \pi_c, \pi_{c+1} \rangle \)
21: \( \phi_1 ← \phi_1 - 1 \)
22: update_list(\( \langle \pi_1, \ldots, \pi_{\phi_1} \rangle \), \( \phi_1 \))
23: else if c > 1 and \( s_{1\in\pi_c} - s_{n_{c-1}\in\pi_c} ≤ n_{c-1} p \) then
24: \( \pi_{c-1} ← \langle \pi_{c-1}, \pi_c \rangle \)
25: c ← c - 1 ; \( \phi_1 ← \phi_1 - 1 \)
26: update_list(\( \langle \pi_1, \ldots, \pi_{\phi_1} \rangle \), \( \phi_1 \))
27: else
28: c ← c + 1
29: C ← \{1, \ldots, \phi_1\}
30: return \( \{s_{1\in\pi_c}, ∀ c ∈ C\} \)

References


