A Dynamic Programming Approach for the Job Sequencing and Tool Switching Problem

Emma Legrand[1][0009−0000−3836−1782], Vianney Coppé[3][0009−0000−3836−1782], Daniele Catanzaro[2][0000−0001−9427−1562], and Pierre Schaus[1][0000−0002−3153−8941]

[1] Department of Computing Science Engineering, Université Catholique de Louvain, Place Sainte Barbe 2, 1348 Louvain-la-Neuve, Belgium
`emma.legrand, pierre.schaus@uclouvain.be`

[2] Center for Operations Research and Econometrics, Université Catholique de Louvain, Voie du Roman Pays 34, 1348, Louvain-la-Neuve, Belgium.
`daniele.catanzaro@uclouvain.be`

[3] Hexaly, 251 Boulevard Pereire, 75017 Paris, France

Abstract. We present a new dynamic programming-based exact solution algorithm for the *Job Sequencing and Tool Switching Problem* (JS-TSP), a combinatorial optimization problem originating from manufacturing systems and encompassing the Traveling Salesman Problem as a special case. We propose a new family of lower bounds for the optimal solution to the problem, which are provably tighter than existing bounds in the literature and enhance both solution quality and pruning efficiency. We propose the use of *A** and its anytime variants to explore the solution space of the problem as well as a specific data structure, called *FreeTools*, both to keep track of the state information and to compute incremental costs throughout the implicit search efficiently. Extensive computational experiments show that the presented approach brings significant performance improvements over state-of-the-art methods for the JS-TSP, including branch-and-bound and integer linear programming formulations.

Keywords: combinatorial optimization · job sequencing · tool switching · branch and bound · A* · dynamic programming.

1 Introduction

Consider a flexible machine $\mathcal{M}$ that can be configured with any subset of maximum c tools from a given set $T = \{0, 1, 2, \ldots, m-1\}$ of $m \geq 3$ tools. We refer to c as the *capacity* of $\mathcal{M}$. In addition, consider a set $J = \{1, 2, \ldots n\}$ of $n \geq 3$ jobs, that must be processed sequentially on $\mathcal{M}$. Each job $j \in J$ requires a subset $t(j) \subseteq T$ of tools and is such that $|t(j)| \leq c$. Whenever a job must be processed on $\mathcal{M}$, one needs to decide which tool, from among those currently present on $\mathcal{M}$, is going to stay and which, instead, needs to be replaced. This operation is commonly referred to as *tool switching*. For example, by referring

Table 1: Example of an instance with 9 tools and 6 jobs with a capacity of 4.

Jobs	1	2	3	4	5	6
Tools	0	1	1	0	5	3
	1	3	5	4	6	5
	2	4	6	6	7	
			8	8		
Magazine capacity: 4						

Table 2: Example of an optimal solution for the instance 1 for which the optimal value is equal to 11.

Jobs	1	2	4	3	5	6
Tools	$\underline{0}$	$\textcircled{0}$	0	$\underline{1}$	6	$\underline{3}$
	$\underline{1}$	1	$\underline{6}$	6	$\underline{7}$	5
	$\underline{2}$	$\underline{3}$	$\underline{8}$	8	5	
		$\underline{4}$	4	$\underline{5}$		
Magazine capacity: 4						

to the job sequence reported in Table 1, two tool switches are involved in the transition from job 3 to job 4, in particular, tools 6 and 8 are going to stay in $\mathcal{M}$ while tools 1 and 5 must be replaced by tools 0 and 4. The *Job Sequencing and Tool Switching Problem* (JS-TSP) consists of determining the job sequence that minimizes the total number of tool switches required to process the jobs in J. For example, Table 2 shows the job sequence that minimizes the overall number of tool switches for the instance of the JS-TSP shown in Table 1. The optimal value for this instance is equal to 11. The tools underlined are those that give rise to a tool switch, while the circled tools have the special property of being not immediately necessary to process a specific job j in the sequence, but needed for processing the jobs subsequent to j. This property, firstly introduced by [16], gives rise to a greedy algorithm, commonly referred to as the *Keep Tool Needed Soonest* (KTNS) algorithm, that computes in polynomial-time the optimal tool switches for a fixed job sequence.

The first applications of the JS-TSP appeared in the literature already in 1966 [2]. The problem, however, was formalized only in 1987 by Tang and Denardo [16], who first proved also its general $\mathcal{NP}$-hardness. Specifically, the authors observed that if the start and stop states of the job processing sequence are represented by a dummy job having index 0 and $t(0) = \emptyset$, then any instance of the JS-TSP that satisfies $|t(j)| = c$, for all $j \in J \setminus \{0\}$, can be seen as an instance of the (symmetric) *Traveling Salesman Problem* (TSP) [1,8] in which the weight associated with the edge (i, j) of the graph is equal to $|t(j)| - |t(i) \cap t(j)|$. Tang and Denardo also proposed the first *Integer Linear Programming* (ILP) formulation for the problem known in the literature, which unfortunately proved to be rather disappointing in practice, mostly due to the poor lower bound provided by its linear programming relaxation [13]. This fact motivated the scientific community to search for improved exact solution algorithms of practical use. Laporte et al. [13] proposed the first ILP formulation able to improve upon Tang and Denardo's one as well as a *Branch-and-Bound* (B&B) algorithm based on dynamic programming. The ILP formulation solved only instances of JS-TSP containing 10 jobs and tools within the reference time of 1 hour. Ghiani et al. [9] proposed an alternative (nonlinear) formulation for the problem able to improve upon Laporte et al.'s one when the ratio between the minimum number of tools required by jobs and the magazine capacity is between 60% and 90%. Bessiere et al. [4] studied a similar problem using Constraint Programming, and Catanzaro et al. [5] proposed new ILP formulations, with the best outperforming earlier ones but struggling to solve instances with 15 jobs and 20 tools within the reference time. At present, the state-of-the-art exact solution algorithm for the JS-TSP is still Laporte et al.'s B&B algorithm, which proves capable of solving instances of the problem having up to 25 jobs and 25 tools within the reference time. This method combines a depth-first recursive enumeration of job sequences with combinatorial lower bounds on the optimal solution to the problem, which proved generally poorer than the ones proposed by Catanzaro et al. [5] but that are very fast to compute. Here, we extend the result discussed in [13] by:

– introducing a new family of combinatorial lower bounds for the optimal solution to the JS-TSP, which are provably tighter than those described in [13];
– proposing the use of A* and its anytime variant to explore the solution space of the problem; in particular, we introduce a specific data structure, called *FreeTools*, to keep track of the state information during the implicit search and to compute the incremental cost of partial job sequences efficiently.

Extensive computational experiments show that the solution algorithm presented in the next sections significantly outperforms state-of-the-art methods for the JS-TSP, including the B&B and ILP formulations discussed in [5,13].

The article is organized as follows. In Section 2, we briefly review, for the sake of completeness, Laporte et al.'s B&B algorithm, the KTNS algorithm, and known lower bounds for the optimal solution to the problem; in addition, we present a new lower bound which is provably tighter than the previous ones. In Section 3, we introduce an alternative version of the KTNS algorithm, called *Lazy KTNS*, and a specific representation of a state of the search space. The Lazy KTNS allows to improve the efficiency of computing the incremental cost of partial job sequences throughout the implicit search, while the specific state representation allows to avoid the exploration of redundant nodes in the search space. In Section 4, we analyze the results obtained when comparing the performance of the new exact solution algorithm versus state-of-the-art methods on a set of benchmark instances of the problem, including Laporte et al.'s B&B algorithm and the ILP formulations discussed above. Finally, in Section 5, we provide some preliminary conclusions and outline possible future research directions.

2 Brief recalls from the literature on the JS-TSP

Starting from an initially empty job sequence S, Laporte et al.'s B&B algorithm [13] enumerates all the permutations of J by augmenting S in a depth-first fashion, i.e., by appending one job at a time to S at each node of the search space. Thus, each node of the search space is defined by the sequence S. Let z^* denote the optimal solution to an instance of the JS-TSP and, for a given

Algorithm 1: Evaluate a job sequence S according to the *Keep Tool Needed Soonest* (KTNS) policy.

```
1  Function KTNS
       Input  : $S \rightarrow$ a (partial) job sequence encoded as a list of integers
       Output: cost $\rightarrow$ an integer encoding the cost of $S$
       // computes $\mathsf{next}[i][t] \leftarrow \min\{\{j \in [i..|S|] \mid t \in t(S[j])\} \cup \{|S|+1\}\}$
2      $\mathsf{next}[i][t] \leftarrow |S|+1 \quad \forall (i,t) \in [1..|J|] \times [1..|S|]$ ;
3      for $i$ from $|S|-1$ to 1 do
4          for $t \in T$ do
5              if $t \in t(S[i])$ then
6                  $\mathsf{next}[i][t] \leftarrow i$ ;
7              else
8                  $\mathsf{next}[i][t] \leftarrow \mathsf{next}[i+1][t]$ ;
9      $t_1 \leftarrow t(S[1])$ ;                 // tools currently set on the machine
10     $\mathsf{cost} \leftarrow |t_1|$ ;
11     for $i$ from 2 to $|S|$ do
12         $t_2 \leftarrow t(S[i])$;
13         $\mathsf{cost} \leftarrow \mathsf{cost} + |t_2 \setminus t_1|$ ;
14         $t_1 \leftarrow t_1 \cup t_2$;
15         while $|t_1| > c$ do
16             $t \leftarrow \arg\max_{k \in t_1}\{\mathsf{next}[i][k]\}$ ;
17             $t_1 \leftarrow t_1 \setminus \{t\}$;
18     return cost ;
```

partial sequence S, let R denote the subset of the remaining jobs to process, i.e., $R = J \setminus S$. Then, Laporte et al.'s B&B algorithm attempts to prune a generic node of the search space by computing a lower bound $lb = \hat{z} + \tilde{z}$ for z^*, where $\hat{z}$ is a lower bound on the number of tool switches associated with S and $\tilde{z}$ is a lower bound on the number of tool switches associated with R, respectively. The choice of the next job to appended to S is determined by the following simple heuristic: if S is empty, the job must have (i) the greatest number of tools, and (ii) these tools must figure from among the most frequently used ones. If $S \neq \emptyset$, the next job is the one sharing the most tools with the last job appended.

Given a (partial) job sequence S, one can compute $\hat{z}$ by using the KTNS algorithm outlined in Algorithm 1. Specifically, the KTNS algorithm relies on the upcoming jobs to decide the current configuration of $\mathcal{M}$. The algorithm first initializes a matrix that computes, for each tool and each position, the soonest next position in the sequence where this tool appears (see lines 2- 8 of Algorithm 1). Then, given the set of tools currently set in $\mathcal{M}$, the algorithm starts with the tools required by the first job $S[1]$. New tools required by the next job induce a cost of 1; and after adding the tools of the next job, the tools less urgently required in the future are removed based on their values in the precomputed `next` table to keep the capacity constraint satisfied. The time complexity of the KTNS algorithm is $\mathcal{O}(n^2 \cdot m)$.

2.1 Lower bounds on the optimal solution to the JS-TSP

We present now three possible combinatorial lower bounds on z^* that can be obtained by specifying a way to compute $\tilde{z}$. We consider in particular three alternative bounds, denoted by $\tilde{z}_1$, $\tilde{z}_2$, and $\tilde{z}_3$, the first two of which can be derived by [6] and [13], respectively, while the third one is new. We start by observing that if $\mathcal{M}$ is set up with c tools at a given state of the job processing sequence, then any tool t which is not currently present in $\mathcal{M}$ and that is needed to process any of the remaining jobs in $R = J \setminus S$ will necessarily entail a tool switch. The following lower bound captures this rationale:

$$\tilde{z}_1 = \left| \bigcup_{i \in R} t(i) \right| - \min \left\{ c, \left| \bigcup_{j \in S} t(j) \right| \right\}. \tag{1}$$

Example 1. Consider the JS-TSP instance shown in Figure 1 and assume that $\mathcal{M}$ has already processed the job sequence $S = \langle 1, 3, 4 \rangle$. Consider the set $R = J \setminus S = \{2, 5, 6\}$. Then,

$$\tilde{z}_1 = \left| \bigcup_{i \in R} t(i) \right| - \min \left\{ c, \left| \bigcup_{j \in S} t(j) \right| \right\} = 6 - \min\{4, 7\} = 2.$$

The second lower bound, i.e., $\tilde{z}_2$, is obtained by considering a complete undirected weighted graph $G = (V, E)$ having $V = R$ and edge weight $w_{ij} = \max\{|t(i) \cup t(j)| - c, 0\}$ for each $e = (i, j) \in E$. Let $T = (V, E_T)$ denote a *Minimum Spanning Tree* (MST) of G, and let l denote the last job in S. The length of this spanning tree plus the cost of the cheapest edge to connect it to l is the lower-bound $\tilde{z}_2$:

$$\tilde{z}_2 = \sum_{(i,j) \in E_T} w_{ij} + \min_{i \in R} w_{li}. \tag{2}$$

Proof. A lower bound on the cost induced by scheduling j immediately after i is given by: $w_{ij} = |t(j) \setminus t(i)| - (c - |t(i)|)$. Here, $|t(j) \setminus t(i)|$ represents the new tools required for j, but up to $c - |t(i)|$ of these tools may already have been set due to the previous jobs scheduled before i. Simplifying, this formula yields $w_{ij} = |t(j) \setminus t(i)| + |t(i)| - c = |t(i) \cup t(j)| - c$. Note that this cost is symmetric, as $w_{ij} = w_{ji}$. Now, observe that the length of the shortest Hamiltonian path on G constitutes a lower bound for z^*. In turn, the length of the MST constitutes a lower bound on the length of the shortest Hamiltonian path as the MST relaxes the degree constraint on the internal vertices of the Hamiltonian path. Thus, adding the cost of the cheapest edge connecting the last job of S to any vertex of T provides a lower bound for z^*. $\square$

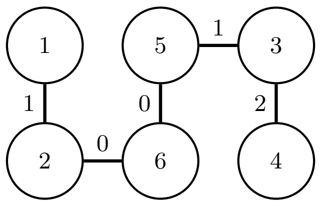

Fig. 1: Example of a Minimum Spanning Tree at the root node.

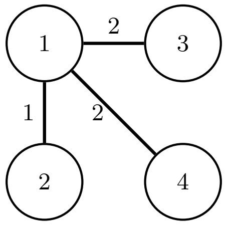

Fig. 2: Example of a Minimum Spanning Tree where $R = \{1, 2, 3, 4\}$.

It is worth noting here that, in general, it is not possible to determine which lower bound between $\tilde{z}_1$ and $\tilde{z}_2$ is the tightest, as shown in the following example.

Example 2. Consider again the JS-TSP instance shown in Figure 1 and assume that $\mathcal{M}$ has not processed any job yet (i.e., $S = \emptyset$, $R = J$). In this case, we have that $\tilde{z}_1 = 9 - \min\{4, 0\} = 9$. To compute $\tilde{z}_2$, it is sufficient to compute the length of the MST for $G = (V, E)$ having $V = J$ with $\min_{i \in R} w_{li} = 0$ as there is no processed job. For example, a MST T for G is shown in Figure 1 and its length is equal to $w_T = 1 + 0 + 0 + 1 + 2 = 4$. Hence, we have that $\tilde{z}_1 > \tilde{z}_2$. Now, consider the case in which $S = \langle 5, 6 \rangle$ and $R = \{1, 2, 3, 4\}$. Then, we have that $\tilde{z}_1 = 8 - 4 = 4$. By considering the MST for $G = (V, E)$, with $V = R$, shown in Figure 2, we have that $w_T = 1 + 2 + 2 = 5$, $\min_{i \in R} w_{li} = w_{62} = 0$ and that $\tilde{z}_2 = 5$. In this case, $\tilde{z}_1 < \tilde{z}_2$ holds.

A third lower bound $\tilde{z}_3$ can be obtained by combining the rationale at the core of $\tilde{z}_1$ and $\tilde{z}_2$. As for $\tilde{z}_2$, an MST $T = (V, E_T)$ is first computed on the graph of the remaining jobs with the same edge weights. Let $\{e_1, \ldots, e_{k-1}\} \subseteq E_T$ denote a set of $k - 1$ edges of the MST. Removing these edges induces partitioning the remaining jobs R into k connected components denoted as $C_1, \ldots, C_k$. Then, the following quantity

$$\tilde{z}_3 = \sum_{i=1}^{k-1} w_{e_i} + \sum_{i=1}^{k} \max\left(0, |\cup_{j \in C_i} t(j)| - c\right) + \min_{i \in R} w_{li} \tag{3}$$

is a lower bound for z^*. The proof of the validity of $\tilde{z}_3$ is omitted here. Still, it can be easily derived from that of $\tilde{z}_2$ by aggregating the nodes in a component and by replacing the length of a spanning tree in each component with $\tilde{z}_1$. Intuitively, the proof of this bound follows the reasoning used in establishing the second lower bound. In this case, the key difference is that an MST node may represent a set of jobs. Since $\tilde{z}_1$ is a lower bound for a set of jobs, it can also be used as a lower bound on the cost associated with this node.

Note that when $k = |R| - 1$, each node corresponds to a component and therefore $\tilde{z}_3 = \tilde{z}_2$ since all the edges of the spanning tree contribute to the cost. When $k = 1$, there is only one component R and $\tilde{z}_3 = \tilde{z}_1 + \min_{i \in R} w_{li}$. The lower bound $\tilde{z}_3$ is a family of bounds since (i) several minimum spanning trees might

Table 3: Example of the computation of the lower bound defined in Equation (4) for the MST in Figure 2.

C_i	C_j	$\tilde{z}_3(C_1) + \tilde{z}_3(C_2) + w_e$	$\lvert \cup_{j \in (C_1 \cup C_2)} t(j) \rvert - c$	$\tilde{z}_3(C_i \cup C_j)$
1	2	$0+0+1=1$	1	1
$\{1,2\}$	3	$1+0+2=3$	4	4
$\{1,2,3\}$	4	$4+0+2=6$	4	6

exist for G and (ii) up to $2^{|R|-1}$ components exist for a given minimum spanning tree. Since we cannot easily guess the best combination of spanning tree and components to get the tightest possible value for $\tilde{z}_3$, we use a heuristic to define the component along the computation of the Kruskal algorithm without the computational overhead. At each step, Kruskal adds an edge e with weight w_e and thus connects two components C_1 and C_2. Let $\tilde{z}_3(C)$ denote the contribution to the bound for a component C. Then, we can set

$$\tilde{z}_3(C_1 \cup C_2) = \max(\tilde{z}_3(C_1) + \tilde{z}_3(C_2) + w_e, \lvert \cup_{j \in (C_1 \cup C_2)} t(j) \rvert - c). \tag{4}$$

By doing so, the bound $\tilde{z}_3$ that we compute is at least as tight as $\tilde{z}_2$ and is obtained with the same time complexity, assuming constant-time computation for union and size operations on sets. This assumption holds for most challenging open instances with fewer than 64 tools, which can be represented as a bitset within a single memory word.

Example 3. Consider the instance of the JS-TSP shown in Figure 1 and the MST shown in Figure 2. Assume that $\mathcal{M}$ has already processed (in this order of appearance) the set of jobs $S = \langle 5, 6 \rangle$ inducing an overall number of tool switches equal to $\hat{z} = 4$. To compute $\tilde{z}_3$, we apply recursively Equation (4). The result of this recursion is shown in Table 3 and the value obtained is $\tilde{z}_3 = 6 > \tilde{z}_2 = 5 > \tilde{z}_1 = 4$.

3 A graph search algorithm

In this section, we present a new dynamic programming-based exact solution algorithm for the JS-TSP that exploits both strategies to reduce the number of nodes explored in the search space and a lazy version of the KTNS algorithm that allows to compute incrementally the cost of a partial sequence so as to keep track of the set of tools that can be reused for free later in the sequence.

3.1 A preliminary observation on the search space

The number of nodes explored in the search space can be significantly reduced by identifying, during the implicit search, partial sequences that lead to equivalent solutions. The next lemma provides an example of how this situation may occur.

Lemma 1. *Consider two non-empty sequences S_1 and S_2 (i) defined on the same set of jobs (S_1 is a permutation of S_2), (ii) such that their last job is identical $last(S_1) = last(S_2)$ and such that (iii) this last job requires c tools, then any sequence on $R = J \setminus S_1 = J \setminus S_2$ induces the same overall cost to complete either S_1 or S_2.*

The example provided in Lemma 1 is unlikely to occur frequently in practice, as more challenging instances typically do not involve jobs that saturate a machine's capacity. Nonetheless, it may be valuable to explore more complex state representations that facilitate efficient testing of whether two partial sequences — defined on the same subset of jobs and ending with the same job — can lead to equivalent solutions.

Example 4. Consider three jobs: $t(1) = \{1\}$, $t(2) = \{2\}$, and $t(3) = \{3, 4\}$, out of n jobs to be scheduled on a machine with capacity $c = 4$. Now, take the two partial sequences $S_1 = \langle 1, 2, 3\rangle$ and $S_2 = \langle 2, 1, 3\rangle$. It is easy to see that the same sub-sequence can be used to complete S_1 and S_2 optimally with the remaining $n - 3$ jobs, as the tools $\{1, 2\}$ can remain on the machine at step 3 in both cases.

Building on this idea, we present a lazy version of the KTNS algorithm in the next section.

3.2 A lazy version of the KTNS algorithm

The *lazy version of KTNS* (Lazy-KTNS for short) computes incrementally the cost of a partial sequence and keeps track of the set of tools that can be reused for free later in the sequence. The decision to keep an unnecessary tool on the machine at each step is deferred until it eventually becomes required on a further step. If the KTNS algorithm can be seen as a forward-looking greedy, the Lazy-KTNS algorithm can be seen as a backward-looking one.

Lazy-KTNS algorithm uses a specific data structure — called *FreeTools* and presented in Algorithm 2 — to collect the set of tools that can be used "for free". Using a tool from this set means deciding it has remained on the machine since its last appearance in the partial sequence until the current time slot, as the standard KTNS algorithm would have done.

The *FreeTools* data structure encodes cardinality constraints on the set of active tools, coming from the limited capacity c of the machine. It is represented as a sequence of sets of tools $\langle F_1, \ldots, F_{size}\rangle$ with $size < c$. The generic F_k represents the set of tools removed from $\mathcal{M}$ at a previous step when the capacity slack was exactly k. Furthermore, the invariant on this sequence is that the sets are pairwise disjoint, and the tools in F_k were set on the machine at a time slot after the tools in $F_1, \ldots, F_{k-1}$. The implicit *FreeTools* semantic of the cardinality constraints on the tools that can be re-used freely reads as the conjunction of the following constraints:

– At most one tool from F_1 can be used for free, and
– At most two tools from $F_1 \cup F_2$ can be used for free,

Algorithm 2: Class FreeTools

```
F[k] : set ← $\emptyset$ , $\forall$ k $\in$ [1..c − 1] ;
size ← 0
Method contains(t: int): int
    foreach k from 1 to size − 1 do
        if t $\in$ F[k] then
            return k;
    return −1;
Method add(k: int, free: set): int
    F[k] ← F[k] $\cup$ free ;
    size ← max(size, k + 1) ;
Method remove(k: int, t: int): int
    F[k] ← F[k] \ {t} ;
    F[k − 1] ← F[k − 1] $\cup$ F[k] ;
    foreach l from k to size − 1 do
        F[l] ← F[l + 1] ;
    F[size − 1] ← $\emptyset$ ;
    size ← size − 1;
Method removeFrom(k: int): int
    foreach l from k + 1 to size − 2 do
        F[k] ← F[k] $\cup$ F[l] ;
        F[l] ← $\emptyset$ ;
    size ← k + 1;
```

- ...
- At most *size* tools from $F_1 \cup \cdots \cup F_{size}$ can be used for free.

To ensure that this property is maintained, a tool used for free must call the method `remove`. When calling this method for a tool t present in F_k, the subsequent sets are left-shifted to reflect the update on the cardinality constraint, as can be seen in the body of the `remove` method.

Algorithm 3 outlines the Lazy-KTNS algorithm. Similarly to its non-lazy version described in Algorithm 1, Lazy-KTNS computes the optimal cost for a fixed partial job sequence S. In addition, Lazy-KTNS makes heavy use of the $FreeTools$ data structure. The tools in $t(i) \setminus t(i-1)$ are the new ones to be scheduled. Each such tool t can be used for free if it appears in $FreeTools$; if not, it must incur a cost of 1. In case it appears in $FreeTools$, then $FreeTools$ must be updated to lazily decide to keep a tool previously on $\mathcal{M}$ from a past job till step i. For example, suppose that we found it in F_k. Then, the slack is reduced by 1 for all the jobs scheduled after. This amounts to shifting all the sets after F_k by one position to the left since those tools were added afterward. This is the operation `remove` in Algorithm 2. The tools that can be used later for free

Algorithm 3: Evaluate a job sequence S using Lazy-KTNS

```
1  Function Lazy-KTNS
       Input  : $S \rightarrow$ a (partial) job sequence encoded as a list of integers
       Output: cost $\rightarrow$ an integer encoding the cost of $S$
       Data   : next $\rightarrow$ a $|J| \times |S|$ matrix that keeps track of which tool is
                needed first in the current sequence $S$
                $t_1$, $t_2 \rightarrow$ support sets of integers used to encode subsets of tools
2      $cost \leftarrow |t(S[1])|$ ;
3      $F \leftarrow$ new $FreeTools()$ ;
4      for $i$ from 2 to $|S|$ do
5          $new \leftarrow t(S[i]) \setminus t(S[i-1])$ ;  // tools not present on the machine
6          for $t \in new$ do
7              $k \leftarrow F$.contains($t$) ;
8              if $k > 0$ then          // can $t$ be reused from previous jobs?
9                  $F$.remove($k, t$) ;
10             else
11                 $cost \leftarrow cost + 1$ ;
12         $slack \leftarrow c - |t(S[i])|$ ;
13         if $slack > 0$ then              // slots available to the machine?
14             if $slack < F$.size then
15                 $F$.removeFrom($slack$) ;
16             if $new \neq \emptyset$ then
17                 $F$.add($slack, t(S[i-1]) \setminus t(S[i])$) ;
18         else
19             $F$.reset() ;
20     return cost ;
```

in F must still be added. The quantity *slack* denotes the number of remaining slots $c - |t(S[i])|$. The tools that are not directly reused in $t(S[i-1]) \setminus t(S[i])$ are the ones that can possibly remain set for later use. By calling `add`, those are stored in the internal set F_{slack} of *FreeTools* to represent that *slack* of them could have stayed set. Also, all the internal sets F_k with $k > slack$ (if any) are added to F_{slack} and then deleted by calling the method `removeFrom`.

The time complexity of the Lazy-KTNS algorithm for processing a partial sequence of size $|S|$ is $\mathcal{O}(|S| \cdot c^2)$. Specifically, for each new job, at most c new tools are added. Each method of the *FreeTools* data structure executes in $\mathcal{O}(c)$, assuming bitset representations for the internal sets of tools and a number of tools that allows constant-time operations on the bitset operations. When the Lazy-KTNS algorithm is implemented in Laporte et al.'s B&B algorithm in place of the usual KTNS algorithm, the time complexity down a branch can be reduced to $\mathcal{O}(n \cdot c^2)$ from $\mathcal{O}(n^2 \cdot m)$ thanks to the reuse of the state of `F`.

Example 5. Consider again the JS-TSP instance shown in Figure 1 and assume that $\mathcal{M}$ has processed a partial sequence $S = \langle 4, 2, 6 \rangle$ and the next job is 5. The current state of the *FreeTools* data structure is $[\{0, 6, 8\}, \{1, 4\}, \{\}]$. After adding the job 5, as F contains the tool 6, it is taken for free, and all sets are shifted by one to the left (line 9). So $\{0, 8\}$ can no longer be taken for free. All other tools induce a cost (line 11). Then, any tools that were previously on the machine but are no longer there are added to F, as there is still an empty space on the machine (line 17). The new current state of *FreeTools* is $[\{1, 3, 4\}, \{\}, \{\}]$.

3.3 On the equivalence between states

FreeTools can help to characterize the equivalence between two partial sequences.

Theorem 1. *Consider two non-empty partial sequences S_1 and S_2 such that:*

1. *S_1 and S_2 are defined on the same set of jobs (S_1 is a permutation of S_2),*
2. *their last job is identical, i.e., $last(S_1) = last(S_2)$,*
3. *the Lazy-KTNS algorithm applied to these sequences results in an identical* FreeTools *state,*

then any sequence S_{tail} on $R = J \setminus S_1 = J \setminus S_2$ induces the same cost to complete S_1 or S_2.

Proof. Let $i = |S_1| = |S_2|$, the length of the sequences. Let S_{tail} denote the sequence completing R. By examining the implementation of Algorithm 3 and assuming its correctness, we observe that the decisions taken by the algorithm for the concatenated sequences $S_1 \cdot S_{tail}$ and $S_2 \cdot S_{tail}$ are identical starting from index $i + 1$. This fact holds because these decisions depend only on the previous job $S[i - 1]$ and the state of the free tools at that point. By assumption, both $S[i - 1]$ and the *FreeTools* state are the same for S_1 and S_2. Therefore, the cost to complete either sequence is identical. $\square$.

Example 6. Consider the JS-TSP instance shown in Figure 1. Let denote $S_1 = \langle 2, 6, 1 \rangle$ and $S_2 = \langle 6, 2, 1 \rangle$, two sets of jobs that $\mathcal{M}$ have already processed (in this order of appearance). These two sets are defined on the same set of jobs, and the last job is identical: 1. The *FreeTools* state for each one is $[\{3, 4, 5\}, \{\}, \{\}]$. Thus, the cost to complete either sequence is identical.

3.4 A graph search algorithm based on A*

The *A** graph search algorithm [11] can be applied to solve instances of the JS-TSP by using a state representation that includes the *FreeTools* data structure, the set of remaining jobs to be scheduled, and the last scheduled job. By Theorem 1, equivalent partial sequences can be identified and merged, thereby reducing redundant exploration of the search space. In this framework, each state corresponds to a node in the graph, each arc is labeled with the job to

Table 4: Dataset Properties.

	Dataset A	Dataset B
n	$\{8, 9, 15, 20, 25\}$	$\{10, 15\}$
m	$\{15, 20, 25\}$	$\{10, 20\}$
c	$\{5, 10, 15, 20\}$	$\{4, 5, 6, 7, 8, 10, 12\}$
min	$\{2, 3, 5, 10, 15\}$	$\{2\}$
max	$\{5, 10, 15, 20\}$	$\{4, 6\}$

be appended and its associated additional cost, and each path represents a sequence of scheduled jobs. The objective is to find the shortest path in this layered graph, ending at a goal node that is reached when all jobs have been scheduled. The admissible heuristic $h(S)$ used by $A*$ is the lower-bound $\tilde{z}_3$. The cost-so-far value $g(S)$ represents the length (or cost) of the shortest path to the current node. The $A*$ algorithm systematically explores the search space by expanding states in the order of their estimated total cost $f(S) = g(S) + h(S)$. The first complete solution encountered is guaranteed to be optimal. In our experiments, we tested several anytime variants of $A*$ that have been successfully applied to other combinatorial optimization problems (see, for instance, [7,12,15]). These variants include *Iterative Beam Search* (IBS) [14], *Anytime Weighted A** (AWA) [18], and *Anytime Column Search* (ACS) [17]. Additionally, we evaluated the decision diagram-based B&B approach [3] and the DDO solver of [10]. However, this direction was abandoned because the bounds obtained by merging states were significantly weaker compared to the specialized bounds.

4 Computational experiments

In this section, we report on the results of applying the graph search algorithm to two benchmark datasets. Dataset A is used to compare our approach with the approach described in [13] as the dataset used in the article. Dataset B is used to compare our approach with that of [5]. Table 4 summarizes the most important characteristics of each dataset that may impact both the time complexity and difficulty of solving the problem to optimality. In particular, an instance is defined by n representing the number of jobs, m the number of tools, c the capacity, and min and max, respectively, the minimum and maximum number of tools contained in a job. Each instance of one of these datasets corresponds to a specific combination of these parameters, but not all combinations exist in the dataset, and 10 instances have the same combination.

The computational experiments reported in this section run an Intel(R) Xeon(R) CPU E5-2687W with 128GB of RAM, with a timeout set to 3600 seconds. The implementation was done in Java and executed with Java 8. The ILP model [5] is modeled in Mosel and solved with FICO Xpress v9.5.0. The source code and instances are available[4].

Our experiments were driven by several key questions:

[4] https://github.com/emmalegrand/JS-TSP

- What is the impact of the different lower bounds $\tilde{z}_1$, $\tilde{z}_2$, $\tilde{z}_3$ (on the cost of the remaining jobs) on the performance of a B&B algorithm that uses the KTNS algorithm to compute the lower bound $\hat{z}$ for the partial sequence?
- After selecting the best bound based on the previous answer, what is the impact of replacing the KTNS algorithm with the Lazy-KTNS algorithm on the execution time of the B&B algorithm?
- What is the impact of using an A* like graph search algorithm instead of a tree search?
- How do the B&B algorithm with Lazy-KTNS and the graph search algorithms compare to the state-of-the-art ILP formulation?

In the next subsections, we address each of these questions.

4.1 Testing the lower bounds

This experiment explores the impact of the lower bounds $\tilde{z}_1$, $\tilde{z}_2$, and $\tilde{z}_3$ on the performance of a B&B algorithm that uses the KTNS algorithm to compute the lower bound $\hat{z}$ for a partial sequence. The implementation using $\max(\tilde{z}_1, \tilde{z}_2)$ replicates Laporte et al.'s B&B algorithm [13]. In this test, we considered instances with a maximum of 15 jobs and focused on the time required to solve them optimally. In Figure 3, we observe that the performance of the B&B algorithm based on lower bound $\tilde{z}_3$ overcomes the others. The algorithm can optimally solve 100% of the instances with up to 15 jobs in at most 3600 seconds. Lower bound $\tilde{z}_1$ also achieves decent results but cannot reach 100%. The second lower bound, $\tilde{z}_2$, is not strong enough to prune a significant number of nodes in the tree, thus failing to solve around 30% of instances within the time limit. Nevertheless, in some cases, it proves stronger than $\tilde{z}_1$, which experimentally supports the idea of using the maximum of the two lower bounds as suggested in [13].

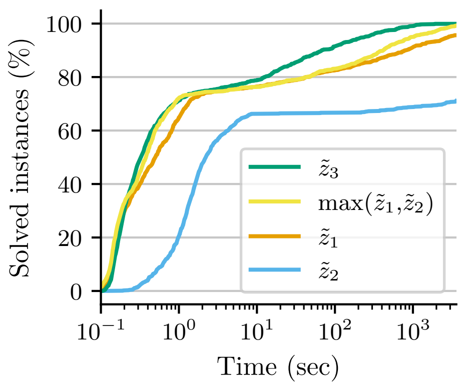

Fig. 3: Solved instances on dataset A regarding the time with $n = \{8, 9, 15\}$ for all lower bound defined in Section 2.1.

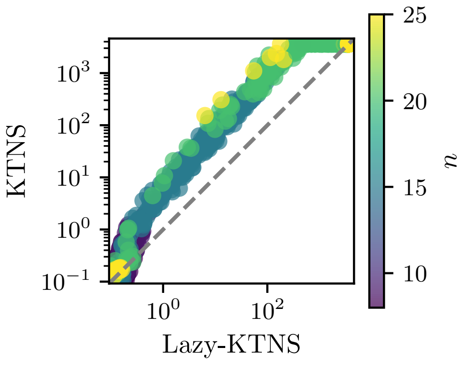

Fig. 4: Comparison of B&B with Lazy-KTNS and KTNS algorithms, regarding the time. Each dot represents an instance of the dataset A.

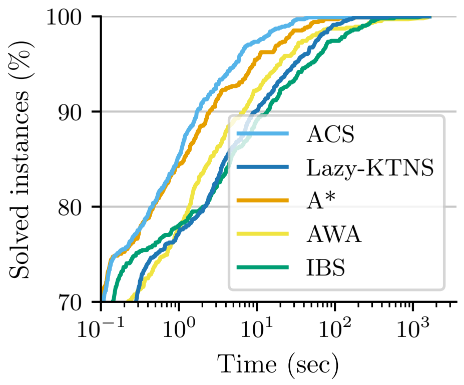

Fig. 5: Solved instances on dataset A regarding the time with $n = \{8, 9, 15\}$.

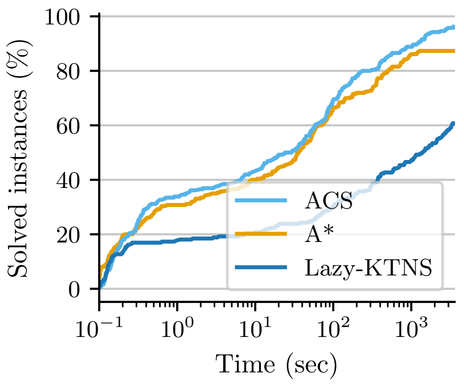

Fig. 6: Solved instances on dataset A regarding the time with $n = \{20, 25\}$.

4.2 Lazy-KTNS algorithm vs KTNS algorithm

Recall from Section 3 that the complexity of Lazy-KTNS algorithm, $\mathcal{O}(n \cdot c^2)$ is lower than $\mathcal{O}(n^2 \cdot m)$ because, in general, c^2 is less or equal than $n \cdot m$. This experiment proposes to compare KTNS and the Lazy-KTNS to compute the lower-bound $\hat{z}$ on the partial sequence while using the lower bound $\tilde{z}_3$ for the remaining jobs. The search spaces are thus identical; only the time spent in each node can vary. As can be observed on the right of Figure 4 illustrating the time to find and prove the optimal solution for each instance with the two versions, replacing KTNS with its lazy version speeds up the execution time for most of the instances, and up to an order of magnitude for the larger values of n.

4.3 B&B vs graph search

As explained in Theorem 1, two non-empty sequences S_1 and S_2 can lead to the same state information. Therefore, graph search, which avoids recomputing similar states, can be used instead of a traditional tree search. This experiment compares our best B&B tree search, denoted as the Lazy-KTNS algorithm, with various A* graph search variants: the standard A* [11], *Anytime Column Search* (ACS) [17], *Iterative Beam Search* (IBS) [14], and *Anytime Weighted A** (AWA) [18]. All these versions utilize the lower bound $\tilde{z}_3$ for the remaining jobs.

As shown in Figure 5 and Figure 6, Lazy-KTNS takes longer to find and prove the optimal solution, compared to its graph search alternatives. This is especially visible for larger instances in Figure 6, where Lazy-KTNS solves only 60% of the instances, while A* and ACS solve 80-90% of the instances. Figure 7 compares the number of explored nodes for each instance by Lazy-KTNS and ACS. It is easy to observe that the number of nodes explored in ACS is lower than in B&B for most instances. These results highlight that the gain from employing a state-caching strategy to avoid exploring similar states can be substantial.

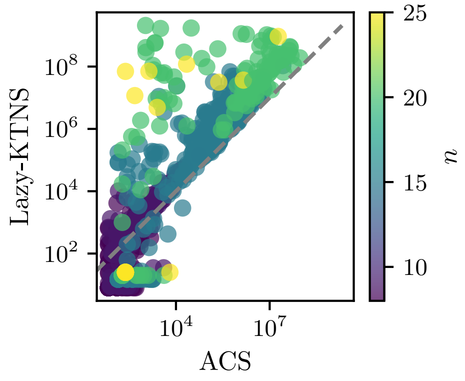

Fig. 7: Comparison of B&B with Lazy-KTNS and ACS, on dataset A, regarding the number of explored nodes.

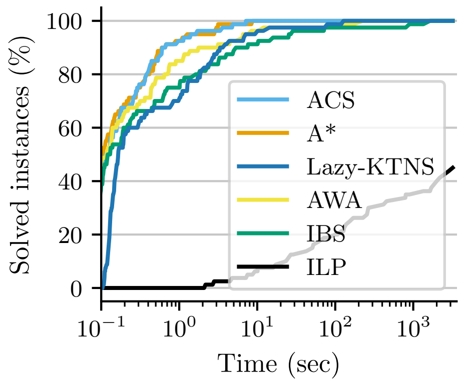

Fig. 8: Solved instances on dataset B regarding the time.

4.4 Comparison with ILP formulations

In this final experiment, we compare the different approaches with the best ILP formulation from [5]. As expected, and consistent with the findings in [5], Figure 8 demonstrates that the ILP formulation struggles to solve instances with 15 jobs within the 3600 second timeout. In contrast, the proposed approaches successfully solve these instances, regardless of the search algorithm used.

5 Conclusion

In this article, we proposed a novel approach for solving the JS-TSP. Our contributions include introducing a new family of lower bounds, denoted as $\tilde{z}_3$, which are provably tighter than existing bounds, and the development of the Lazy-KTNS algorithm, a computationally efficient variant of the KTNS algorithm. Additionally, we demonstrated the advantages of reformulating the traditional tree-based B&B algorithm into a graph search problem, leveraging the state-caching capabilities of modern search algorithms such as A* and its anytime variants. Our source code is publicly available, providing reproducible experiments for anyone interested in comparing their approach with ours.

For future work, we aim to explore alternative instantiations of the lower bound $\tilde{z}_3$ and compare them with the greedy one based on spanning tree construction. Additionally, we plan to investigate how a constraint programming solver performs on this problem and whether global constraints could be used or designed to solve it more efficiently. We believe the KTNS strategy could inspire strong constraint programming models for the JS-TSP.

References

1. Applegate, D.L., Bixby, R.E., Chvátal, V., Cook, W.J.: The traveling salesman problem: A compuational study. Princeton University Press, NJ (2006)
2. Belady, L.A.: A study of replacement algorithms for virtual storage computers. IBM Systems Journal **5**, 78–101 (1966)
3. Bergman, D., Cire, A.A., Van Hoeve, W.J., Hooker, J.N.: Discrete optimization with decision diagrams. INFORMS Journal on Computing **28**(1), 47–66 (2016)
4. Bessiere, C., Hebrard, E., Ménard, M.A., Quimper, C.G., Walsh, T.: Buffered resource constraint: Algorithms and complexity. In: Integration of AI and OR Techniques in Constraint Programming: 11th International Conference, CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proceedings 11. pp. 318–333. Springer (2014)
5. Catanzaro, D., Gouveia, L., Labbé, M.: Improved integer linear programming formulations for the job sequencing and tool switching problem. European journal of operational research **244**(3), 766–777 (2015)
6. Crama, Y., Oerlemans, A.G., Spieksma, F.C., Crama, Y., Oerlemans, A.G., Spieksma, F.C.: Minimizing the number of tool switches on a flexible machine. Springer (1996)
7. Fontaine, R., Dibangoye, J., Solnon, C.: Exact and anytime approach for solving the time dependent traveling salesman problem with time windows. European Journal of Operational Research **311**(3), 833–844 (2023)
8. Garey, M.R., Johnson, D.S.: Computers and Intractability: A guide to the theory of NP-Completeness. Freeman, New York, NY (2003)
9. Ghiani, G., Grieco, A., Guerriero, E.: Solving the job sequencing and tool switching problem as a nonlinear least cost hamiltonian cycle problem. Networks: An International Journal **55**(4), 379–385 (2010)
10. Gillard, X., Schaus, P., Coppé, V.: Ddo, a generic and efficient framework for mdd-based optimization. In: International Joint Conference on Artificial Intelligence (IJCAI-20) (2014)
11. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE transactions on Systems Science and Cybernetics **4**(2), 100–107 (1968)
12. Kuroiwa, R., Beck, J.C.: Solving domain-independent dynamic programming problems with anytime heuristic search. In: Proceedings of the International Conference on Automated Planning and Scheduling. vol. 33, pp. 245–253 (2023)
13. Laporte, G., Salazar-Gonzalez, J.J., Semet, F.: Exact algorithms for the job sequencing and tool switching problem. IIE transactions **36**(1), 37–45 (2004)
14. Libralesso, L., Bouhassoun, A.M., Cambazard, H., Jost, V.: Tree search for the sequential ordering problem. In: ECAI 2020, pp. 459–465. IOS Press (2020)
15. Libralesso, L., Focke, P.A., Secardin, A., Jost, V.: Iterative beam search algorithms for the permutation flowshop. European Journal of Operational Research **301**(1), 217–234 (2022)
16. Tang, C.S., Denardo, E.V.: Models arising from a flexible manufacturing machine, Part I: Minimization of the number of tool switches. Operations Research **36**(5), 767–777 (1987)
17. Vadlamudi, S.G., Gaurav, P., Aine, S., Chakrabarti, P.P.: Anytime column search. In: AI 2012: Advances in Artificial Intelligence: 25th Australasian Joint Conference, Sydney, Australia, December 4-7, 2012. Proceedings 25. pp. 254–265. Springer (2012)
18. Zhou, R., Hansen, E.A.: Multiple sequence alignment using anytime a*. In: AAAI/IAAI. pp. 975–977 (2002)