The unit-capacity constrained permutation problem

Pascale Bendotti\textsuperscript{a,b}, Pierre Fouilhoux\textsuperscript{b,*}, Safia Kedad-Sidhoum\textsuperscript{b}

\textsuperscript{a}EDF R&D, D\'epartement OSIRIS, 1 avenue du G\'en\'eral de Gaulle, 92141 Clamart cedex, France
\textsuperscript{b}Sorbonne Universit\'es, Universit\'e Pierre et Marie Curie, LIP6 CNRS UMR 7606, 4 place Jussieu 75005 Paris, France

Abstract

The Unit-capacity Constrained Permutation Problem (UCPP) is to find a sequence of moves for pieces over a set of locations. From a given location, a piece can be moved towards a location with a unit-capacity constraint, \textit{i.e.} the latter location must be free of its original piece. Each piece has a specific type and at the end every location must contain a piece of a required type. A piece must be handled using a specific tool incurring a setup cost whenever a tool changeover is required. The aim of the UCPP is finding a sequence of moves with a minimum total setup cost. This problem arises in the Nuclear power plant Fuel Renewal Problem (NFRP) where locations correspond to fuel assemblies and pieces to fuel assembly inserts. We first study the complexity of the UCPP and show that the UCPP is NP-hard. We exhibit some symmetry and dominance properties and propose a dynamic programming algorithm to solve the problem. Using this algorithm, we prove that the UCPP is polynomial when two tools and two types are considered. Experimental results showing the efficiency of the algorithm for some instances coming from the NFRP are presented.

\textit{Keywords:} Combinatorial optimization, OR in energy, Complexity theory, Dynamic programming, Dominance and symmetry properties.

\*Corresponding author. Correspondence to: Universit\'e Pierre et Marie Curie, Laboratoire LIP6, Bo\^{i}te courrier 169, 4 Place Jussieu, 75252 Paris cedex 05, France Tel: +33 1 44 27 87 96; fax: +33 1 44 27 70 00
\textit{Email addresses:} pascale.bendotti@edf.fr (Pascale Bendotti), pierre.fouilhoux@lip6.fr (Pierre Fouilhoux), safia.kedad-sidhoum@lip6.fr (Safia Kedad-Sidhoum)

1. Introduction

We consider a set of \( p \) pieces that have to be moved over a set of locations. A location contains at most one piece. Each piece possesses a specific type among \( q \) different types, denoted by \( \{1, \ldots, q\} \). In the initial assignment, each piece is assigned to an initial location and, in the final assignment, each location must contain a piece of a given type. Therefore, for each location whenever the type of a piece in the initial assignment is different from that required in the final assignment, the piece should change location. Finding a sequence of moves becomes an issue when pieces are handled with a unit-capacity constraint.

In fact each move involves only one piece at a time, and assigns a piece to another location provided the piece originally there has previously been moved. It is assumed that each piece can be moved only once and that any location can be reached from any other location. Furthermore, to handle a piece of type \( t \in \{1, \ldots, q\} \), a specific tool is used among the \( r \) possible tools among \( \{1, \ldots, r\} \). Therefore, pieces that can be handled in turn by using the same tool are gathered into batches, and each batch of pieces will incur a setup cost. The Unit-capacity Constrained Permutation Problem (UCPP) is to find a sequence of moves with a unit-capacity constraint for pieces to have the required type, such that the total setup cost is minimized.

The time it takes to pick-up, move and drop-off a piece is assumed to be constant. We further assume that the setup cost is the same for every tool and is equal to one. It follows that the setup cost is equal to the number of tool changeovers. Moreover, we assume that each instance of the problem supports at least one feasible solution.

To illustrate the problem, consider an instance with four locations and three pieces. Figure 1a shows the location (represented by a square) of each piece in the initial assignment. The three pieces are of two different types \( a \) and \( b \) (symbolized by circles). The pieces of type \( a \) can be moved using tool 1 (circle with full line), while the piece of type \( b \) can be moved using tool 2 (circle with
dashed line). Figure 1b shows the type of piece required for each location in the final assignment. One of the two pieces of type a can be moved to location 4. However, the piece of type b cannot be moved to location 3, as the original piece is still there.

Therefore, the piece of type a initially at location 3 should first be moved using tool 1 to location 4, thus making it possible to move the piece of type b to location 3 using tool 2. Finally the piece of type a at location 1 can be moved to its final location 2 using tool 1. Such a sequence of moves is illustrated in Figure 2. It is worth noting that two pieces cannot exchange their location, as one piece would need to be moved twice using an intermediate location, which is not permitted. Note that there should be at least one location free of piece for any sequence of moves to be feasible.

The UCPP arises in the nuclear power plant fuel renewal context. The fuel assemblies are located in the core of the reactor. Depending on the length of an
operating cycle that ranges from 12 to 18 months, a fraction typically one third
or one quarter of the fuel assemblies is replaced in the core. During a refueling
outage, some spent fuel assemblies are then replaced with fresh ones while the
remaining fuel assemblies have been used from one cycle up to two or three. Each
fuel assembly is equipped with an insert. In particular, specific requirements
apply depending on the location of the inserts in the core and translate into
constraints on types of inserts. The Nuclear power plant Fuel Renewal Problem
(NFRP) aims at performing the permutation of the inserts over the spent, fresh
and remaining fuel assemblies. As an insert consists of a cluster of long rods, it
should be handled with both great precision and care. For these reasons, only
one insert can be moved at a time by a spent fuel mast bridge, i.e. the fuel
handling system. A specific tool is required to handle a subset of types of inserts.
Moreover, the bridge carries only one tool at a time. This makes the constraint
on the capacity quite restrictive as not only one insert can be moved at a time,
but also the targeted location should be free of insert, in particular if it contains
one. In the whole refueling operation, the refueling machine lifts every fuel
assembly out of the reactor core and moves it from the reactor building through
a transfer container to the fuel building. Installing new fuel assemblies in the
core is performed using the same process in reverse. From a practical point of
view, the NFRP is solved before the fuel assemblies are moved out of the reactor
core. The fuel assemblies will be located in the fuel building according to the
moves involved in the solution of the NFRP. This makes the assumption on a
constant time to move an insert reasonable. Consequently, the NFRP aims at
minimizing the number of tool changeovers which is of paramount importance
in the nuclear power plant fuel renewal context.

The NFRP reduces to the UCPP where the locations correspond to the fuel
assemblies, and the pieces to the fuel assembly inserts and is to find an optimal
handling sequence for the spent fuel mast bridge to complete the permutation
of inserts for the renewal of the fuel assemblies.

For the nuclear power plants operated by Electricité de France, a heuristic
developed at EDF R&D in the seventies is used [Guigou and Mauget (1987)];
Mauget (1987)). It is nowadays used for the refueling of the 58 nuclear reactors operated by EDF. The heuristic assumes only one final location per insert. This restriction may lead to a sub-optimal solution. The assumption is already too restrictive for the operating reactors. It is also the case for the coming units of the European Pressurized Reactors (EPR) as new types of pieces with several final locations must be considered. In terms of practical setting, there is a need for more accurate solving approaches to tackle realistic instances of the NFRP.

To this end a first step is to perform a complexity analysis of the problem. The motivation of this article is to rely on a complexity analysis of the UCPP problem in deciding whether some exact polynomial methods can be derived. Moreover we propose a structural analysis of the problem using a dynamic programming approach to discuss some particular practical cases.

The organization of the article is as follows: Section 2 proposes a literature review about the UCPP together with a description of the contributions of this article. Section 3 provides some notations and some definitions that will be used in Section 4 to set the complexity of the UCPP. Some particular cases are exhibited. In Section 5 some dominance properties are introduced to reduce the solution space. Section 6 shows that the UCPP instances feature some symmetries. Section 7 presents a dynamic programming algorithm exploiting symmetries and dominance properties. Using this algorithm, a particular case of the UCPP is proved to be polynomial. Finally, Section 8 presents some experimental results while Section 9 concludes the article.

2. Literature review and contributions

The UCPP tends to remind us of combinatorial problems dealing with robots in production scheduling. However, the unit-capacity constraint involved in the UCPP translates into the use of a single robot to move all the pieces. Furthermore unlike many production scheduling problems where the robot carries a subset of tools (see Crama (1997)), the unit-capacity constraint implies that the robot can carry only one tool at a time.
The UCPP can be seen as a particular *Pickup and Delivery problem* (PDP) \cite{Savelsbergh:1995}. In the PDP, a vehicle must satisfy a set of transportation requests where a request is defined by a pick-up point, a corresponding delivery point and a demand to be carried between these two locations. Hence the UCPP seems to be closely related to the PDP using a single vehicle with a unit-capacity constraint \cite{Gribkovskai:2008} and, more specifically, to the *swapping problem* \cite{Anily:1992}. The swapping problem can be seen as a particular PDP where every pick-up point of a request is also a delivery point of another request. In the preemptive (non-preemptive) swapping problem, a piece can (cannot) be temporarily stored at intermediate vertices on the vehicle’s route before reaching its final destination and can (cannot) be reloaded later \cite{Bordenave:2009, Erdogan:2010, Anily:2011, Bordenave:2012}. The UCPP is then related to the non-preemptive version of the swapping problem. However the UCPP deeply differs from these two former problems. For a pick-up-and-delivery operation, the vehicle would frequently have to handle two pieces at a time. In the swapping problem, it is allowed to swap pieces between two locations. Neither the former nor the latter operation is possible in the UCPP. Another difference lies in the objective function: while in both the PDP and the swapping problem the cost matrix corresponds to the distance between locations, and is assumed to satisfy the triangular inequalities, in the UCPP the total setup cost is minimized, and the total distance traveled not considered.

As only tool changeover (setup) cost minimization is considered, the UCPP falls into the single-machine scheduling problem with sequence-dependent setup cost category \cite{Allahverdi:2008}, which can be reformulated as a Traveling Salesman Problem. It is worth noting that \cite{VanderVeen:1996} considered a special case of a sequence-dependent scheduling problem, referred as the $K$-group case, where jobs can be divided into $K$ groups depending on which resource they require. The authors proposed a polynomial-time algorithm to solve this problem. Furthermore, \cite{Psaraftis:1980} used a dynamic programming approach for sequencing a given set of jobs on a single machine for the
$K$-group case. For a problem of $m$ jobs per group, the algorithm’s running time is shown to grow as $K^2(m + 1)^K$. This is a polynomial function of $m$, but exponential function of $K$ if $K$ is not fixed. Such an algorithm could be of practical interest for applications where $K$ is small. The UCPP can be seen as a single-machine scheduling problem where the pieces are the jobs to process and the required resource the tool. Recall that in the UCPP, a piece can only be assigned to a given location if the piece originally there has already been moved. Therefore, precedence constraints should be considered and the algorithms proposed in [Van der Veen and Zhang (1996)] are not relevant for the UCPP.

In [Bendotti et al. (2015)], the authors present a necessary and sufficient condition to check whether an instance of the UCPP is feasible or not. In this latter case, they propose to use additional locations containing no piece, referred as Steiner locations. The resulting problem is called the UCPP with Feasibility Recovery (UCPPFR). Using a compact encoding for solutions and integer linear programming tools, they solve real instances coming from the NFRP.

In this article, we study the complexity of the UCPP with a constant setup cost for every tool. A generalized version of the UCPP using a general setup cost function is clearly NP-hard as it admits the Traveling Salesman Problem as a particular case. However, when the objective is to minimize the number of tool changeovers, the complexity of permuting pieces with a unit-capacity constraint is an open question. A reduction from a particular case of the well-known Shortest Common Supersequence problem ([Garey and Johnson (1979)]) enables us to prove that the UCPP is NP-hard. Some classes of instances where the UCPP can be solved in polynomial time are presented. For solving the general case, we provide a dynamic programming algorithm based on some symmetry and dominance properties of the feasible instances. This algorithm provides optimal solutions for some classes of NFRP instances.
3. Problem statement

In this section we introduce some notations and properties for the UCPP.

A set $V$ of locations is considered. There are $p$ pieces initially allotted at some locations of $V$. The initial (resp. final) assignment is given by function $\tau^0$ (resp. $\tau^f$) as follows. For every location $i \in V$ if a piece is at $i$ in the initial assignment, $\tau^0(i)$ denotes its type and is set to $-$ otherwise; similarly if a piece is required at $i$ in the final assignment, $\tau^f(i)$ denotes its required type and is set to $-$ otherwise. By convention, the type is set to $-$ whenever the location $i$ is free of piece. Without loss of generality, it is assumed that for every $i \in V$, $\tau^f(i)$ is distinct from $\tau^0(i)$. If they were the same, the piece would already be in an allowable location and no move would in fact be required.

We consider a partition $V = (N, U, C)$ where $N = \{i \in V : \tau^0(i) = -\}$, $U = \{i \in V : \tau^f(i) = -\}$ and $C = V \setminus (N \cup U)$. It is worth noting that locations of $N$ (resp. $U$) can be seen as “new” locations (resp. “used” locations) with no piece in the initial (resp. final) assignment, while locations of $C$ can be seen as “renewable” locations with a piece in both the initial and final assignments.

An instance of the UCPP is shown in Figure 3 to illustrate these notations. This instance with eight locations and five pieces is fully given by its initial and final assignments. The five pieces are of three different types $a$, $b$ and $c$ (symbolized by circles). The pieces of type $a$ can be moved using tool 1 (circles with full line), while the pieces of types $b$ and $c$ can be moved using tool 2 (circles with dashed line). There are two pieces of type $a$ at locations 1 and 5 in the initial assignment; moreover pieces of type $a$ are required at locations 2 and 6 in the final assignment, i.e. $\tau^0(1) = \tau^0(5) = \tau^f(2) = \tau^f(6) = a$. The other two types $b$ and $c$ then correspond to the values $\tau^0(2) = \tau^f(3) = b$ and $\tau^0(4) = \tau^0(6) = \tau^f(7) = \tau^f(8) = c$. There are three new locations $N = \{3, 7, 8\}$ that are free of piece in the initial assignment, i.e. $\tau^0(3) = \tau^0(7) = \tau^0(8) = -$; three used locations $U = \{1, 4, 5\}$ that do not require any piece in the final assignment, i.e. $\tau^f(1) = \tau^f(4) = \tau^f(5) = -$; and two renewable locations $C = \{2, 6\}$. 

We define a move as a pair of distinct locations \((i, j)\), \(i, j \in V\) such that there is a piece located at \(i\) whose type is that required at \(j\). The set of moves is denoted by \(A = \{(i, j) \mid i \in U \cup C, j \in N \cup C, i \neq j \text{ and } \tau^0(i) = \tau^f(j)\}\).

For convenience purposes, we introduce a graph model to represent a UCPP instance. Let \(G = (V, A)\) be a directed graph defined as follows. Each vertex \(v\) in the set \(V\) corresponds to a given location and each arc then corresponds to a move for the UCPP instance. Indeed, an arc \((i, j)\) belongs to \(A\) if and only if vertex \(i\) contains a piece of a given type in the initial assignment and vertex \(j\) requires a piece of the same type in the final assignment. Moreover, each arc of \((i, j) \in A\) is associated with the tool \(\Omega(i) \in \{1, \ldots, r\}\) required to move the piece located at vertex \(i\).

For example, by referring to the instance shown in Figure 3, the corresponding graph model is illustrated in Figure 4. The possible moves are shown by the arcs. The fact that the pieces of type \(a\) (resp. \(b\) and \(c\)) are handled by tool \(o_1\)
(resp. \(o_2\)) is represented using solid lines (resp. dashed lines). Note that both initial and final assignments could be recovered from the arcs by looking at their heads and tails. Note also that a vertex with no outgoing arc in \(A\) contains no piece in the initial assignment. Similarly, a vertex with no incoming arc in \(A\) contains no piece in the final assignment.

We define a permutation as a sequence \(\sigma = ((i_1,j_1),\ldots,(i_p,j_p))\) of distinct moves of \(A\) to handle pieces from their initial location to another location requiring their type. Consequently a permutation \(\sigma\) corresponds to a solution of the UCPP. Note that a permutation contains exactly \(p\) moves since a piece is moved exactly once.

The following property characterizes the sequence of moves that corresponds to a permutation.

**Property 1.** A sequence \(\sigma = ((i_1,j_1),\ldots,(i_p,j_p))\) of \(p\) distinct moves of \(A\) is a permutation if and only if

i) \(\forall i \in U \cup C, there is exactly one move (i_k,j_k), k \in \{1,\ldots,p\}\) with \(i_k = i\),

ii) \(\forall j \in N \cup C, there is exactly one move (i_k,j_k), k \in \{1,\ldots,p\}\) with \(j_k = j\),

iii) \(\forall k \in \{1,\ldots,p\}\) with \(j_k \in C\), there is \(l \in \{1,\ldots,k-1\}\) with \(i_l = j_k\), i.e. location \(j_k\) is free of piece after the first \(k - 1\) moves.

Given a permutation \(\sigma\), for every move \((i,j)\) of \(\sigma\), location \(i\) must contain a piece in the initial assignment and its type is required at location \(j\) in the final assignment. Moreover location \(j\) should be free of piece before moving the piece located at \(i\). Property i) follows from the fact that a piece must be moved exactly once and property ii) that a location of \(N \cup C\) must contain exactly one piece in the final assignment. Property iii) indicates that a piece located at \(i\) can be moved to \(j\) when \(j \in C\) provided \(j\) is free of piece. There are two cases: either \(j\) is free of piece in the initial assignment (i.e. \(j \in N\)), or the piece located at \(j\) in the initial assignment has previously been moved (i.e. \(j \in C\)).

Let \(\Omega(t) \in \{1,\ldots,r\}\) be the tool required to handle a piece of type \(t\), \(t \in \{1,\ldots,q\}\). Given a permutation \(\sigma = ((i_1,j_1),\ldots,(i_p,j_p))\), the total setup cost along a permutation is the number of tool changeovers incurred to perform the
moves, i.e. $1 + \sum_{k=1}^{p-1} \delta(t^0(i_k+1))$, where $\delta_a$ is a function equals 0 if $a = b$ and 1 otherwise.

For example, by referring to the instance shown in Figure 3 the sequence $((2,3),(1,2),(6,7),(5,6),(4,8))$ is a permutation that corresponds to the sequence of types $(b,a,c,a,c)$ and then to the sequence of tools $(o_2,o_1,o_2,o_1,o_2)$ for a total setup cost of five tool changeovers. There exists another permutation $((4,8),(2,3),(6,7),(1,2),(5,6))$ that corresponds to $(c,b,c,a)$ and to the tool sequence $(o_2,o_1)$ incurring only two tool changeovers.

An instance of the UCPP is fully given by $I = (V,p,q,r,\tau^0,\tau^f,\Omega)$. Hence the UCPP on instance $I$ is equivalent to finding a permutation that minimizes the total setup cost. Since $\tau^0$ and $\tau^f$ can be deduced from graph $G = (V,A)$, an instance of the UCPP can also be fully described by $I = (p,q,r,G,\Omega)$.

4. Complexity analysis

In this section we set the complexity of the UCPP. To this end, we first introduce a problem derived from the well-known shortest common supersequence problem (Garey and Johnson (1979)). Indeed, minimizing the setup cost of a UCPP instance is to find a particular sequence whose characters correspond to the tools required to handle the pieces.

4.1. Shortest Common Supersequence

Given a finite sequence $S$ over an alphabet $\Lambda$, a sequence $S'$ is a supersequence (resp. subsequence) of $S$ if some characters in $S'$ can be deleted (resp. added) such that the resulting sequence is equal to $S$. Given a finite set $R = \{S^1,S^2,\ldots,S^k\}$ of sequences, a common supersequence (CS) of $R$ is a sequence which is a supersequence of every $S^i$, $i \in \{1,\ldots,k\}$. For example, $12221$ is a CS of $\{1222,121,221\}$.

Given a finite alphabet $\Lambda$, a finite set of sequences $R$ over $\Lambda$ and a positive integer $\lambda$, the Shortest Common Supersequence Problem (SCSP) is to find whether there exists a sequence $S$ with $|S| \leq \lambda$ which is a CS of $R$. Maier
first showed that the SCSP is NP-complete for a 5-character alphabet, and Räihä and Ukkonen (1981) that the SCSP is NP-complete for a 2-character alphabet.

We will show that a solution of the UCPP is similar to a solution of this SCSP where the characters correspond to the tools required to handle the pieces. However in a UCPP solution, the setup cost is not related to minimizing the size of the tool sequence, but rather the number of tool changeovers. Therefore to analyze the UCPP complexity we introduce the following special case of the SCSP where any two consecutive elements are distinct.

A sequence $S = (x_0, \ldots, x_{l-1})$ over an alphabet $\Lambda$ is defined as a $D$-sequence if each character is different from the next, i.e. $x_i \neq x_{i+1}$, for $i = 0, \ldots, l - 2$.

We consider a special case of the SCSP defined as follows: given a finite alphabet $\Lambda$, a finite set $R$ of D-sequences over $\Lambda$ and a positive integer $\lambda$, the Shortest Common $D$-Supersequence Problem (SCDSP) is to find whether there exists a D-sequence $S$ with $|S| \leq \lambda$ which is a CS of $R$. We refer to the resulting CS as a Common $D$-Supersequence (CDS).

It is easy to see that SCDSP is polynomial for sequences over an alphabet of size 2, as the longest sequence over a set $R$ is almost its CDS. In Fleischer and Woeginger (2012), it is shown that the SCDSP is NP-complete for sequences over an alphabet of size 4. It can be deduced from a result of Darte (2000) that the SCDSP is still NP-complete for alphabet of size 3. A shorter proof of this result can be found in Lagoutte and Tavenas (2015).

4.2. Complexity result

We can now prove that, given $r \in \mathbb{N}$, the SCDSP over an alphabet with $r$ characters reduces to the UCPP with $r$ tools. Given a UCPP instance and a positive integer $\lambda$, the decision problem of the UCPP is to find whether there exists a permutation using $\lambda$ tool changeovers.

**Lemma 1.** Given $r \in \mathbb{N}$, the SCDSP over an alphabet of size $r$ reduces to the decision problem of the UCPP.
Proof. Consider an instance $I$ of the SCDSP given by the alphabet $\Lambda = \{1, \ldots, r\}$, a set $R = \{S^1, S^2, \ldots, S^k\}$ of $k$ sequences over $\Lambda$ and a positive integer $\lambda$. We denote by $S^i$ the sequence $(x^i_0 \ldots x^i_{l^i-1})$, $i \in \{1, \ldots, k\}$. W.l.o.g. we assume that $l^i > 2$ for every $i \in \{1, \ldots, k\}$. An instance $J = (p,q,r,G,\Omega)$ of the UCPP is derived from the SCDSP instance $I$ as follows: for each sequence $S^i$ of $R$, $i \in \{1, \ldots, k\}$, we create $l^i+1$ vertices $u^i_0, u^i_1, \ldots, u^i_{l^i}$, as the component parts of set $V$. Figure 5 illustrates the resulting UCPP instance for a given sequence $R^i$.

In the initial assignment, pieces are located at vertices $u^i_0, \ldots, u^i_{l^i-1}$ and $u^i_l$ is free of piece. There is a bijection between the set of types and the set of pieces.

The set of types realizes a bijection with the set of pieces, indeed $p = q = \sum_{i=1}^{k} l_i$. For every $i \in \{1, \ldots, k\}$ and $j \in \{0, \ldots, l_i-1\}$, an arc $(u^i_j, u^i_{j+1})$ associated with character $x^i_j$ in the sequence $S^i$ of $R$ is defined in $G$. The resulting instance $J$ can clearly be constructed from the SCDSP instance in polynomial time and space. Note that by construction, the resulting graph $G$ is exactly the union of the vertex-disjoint paths $\mu_i = (((u^i_0, u^i_1), (u^i_1, u^i_2), \ldots, (u^i_{l_i-1}, u^i_{l_i})), i \in \{1, \ldots, k\}$, so that each of them is associated with a sequence $S^i$.

We now prove that there exists a CDS of size at most $\lambda$ for $I$ if and only if there exists a permutation $\sigma$ of cost at most $\lambda$ for $J$.

Suppose that there exists a permutation $\sigma$ of cost $\rho \leq \lambda$. Let $S = (y_0, \ldots, y_{\rho-1})$ be the sequence of characters corresponding to the $\rho$ consecutive tools used in $\sigma$. Sequence $S$ is then a D-sequence over the alphabet $\{1, \ldots, r\}$.

By construction, a permutation $\sigma$ must contain every subsequence

$$((u^i_{l_i-1}, u^i_{l_i}), (u^i_{l_i-2}, u^i_{l_i-1}), \ldots, (u^i_0, u^i_1)), i \in \{1, \ldots, k\},$$

whose arcs are associated with the sequence $(x^i_{l_i-1}, \ldots, x^i_0)$ which is the mirror
sequence of \( S_i \). Consequently, the mirror sequence \( S^{-1} = (y_{p-1}, \ldots, y_0) \) is a D-supersequence of \( R \) with \( |S^{-1}| = |S| = \rho \).

Conversely, assume that there exists a CDS \( S = (y_0, \ldots, y_{\rho-1}) \) of \( R \) with \( |S| \leq \lambda \) for \( I \). We construct a permutation \( \sigma \) by selecting arcs following supersequence \( S \). The first step of this construction is to partition the set of vertices containing a piece into \( |S| - 1 \) subsets: each of these subsets will correspond to moves using the same tool. Such a partition can be obtained according to the following algorithm. Let \( V_0, \ldots, V_{|S|-1} \) be empty vertex sets that will be filled with the vertices of \( V \).

\begin{verbatim}
For each \( i \in \{1, \ldots, k\} \) Do
  Let \( y_m \) be the rearmost character in \( (y_0, \ldots, y_{|S|-1}) \) with \( y_m = x_0^i \)
  \( V_m \leftarrow V_m \cup \{u_0^i\} \)
  For each \( j \in \{1, \ldots, l_i - 1\} \) Do
    Let \( y_{m'} \) be the rearmost character in \( (y_0, \ldots, y_{m-1}) \) with \( y_{m'} = x_j^i \)
    \( V_{m'} \leftarrow V_{m'} \cup \{u_j^i\} \)
    \( m \leftarrow m' \)
  End For
End For
\end{verbatim}

This algorithm is clearly valid since sequences \( S_i, i \in \{1, \ldots, k\} \), are sub-sequences of \( S \). At the end of the algorithm, every vertex of \( V \) containing a piece in the initial assignment is assigned to one of the subsets \( V_m, m \in \{0, \ldots, |S| - 1\} \).

The second step of the construction provides permutation \( \sigma \) by sorting the arcs using partition \( V_0, \ldots, V_{|S|-1} \). Let \( \text{succ}(u) \) be the unique successor of \( u \in V \) in \( G \). Let \( \sigma_m, m \in \{0, \ldots, |S| - 1\} \), be an arbitrary ordered sequence of the arcs of \( \{(u, \text{succ}(u)), u \in V_m\} \). Therefore, \( \sigma = (\sigma_{|S|-1}, \ldots, \sigma_0) \) contains every path \( \mu_i, i \in \{1, \ldots, k\} \) and is then a permutation of cost \( |S| \).

**Corollary 1.** The UCPP with 3 tools is NP-hard.

**Proof.** Since checking whether an assignment sequence is a permutation can be done in polynomial time, the UCPP is clearly in NP. Lagoutte and Tavenas (2015) have shown that the SCDS is NP-complete for a 3-character alphabet, then from Lemma \( \square \) the UCPP with 3 tools is NP-hard. \( \square \)
This complexity result relies on the particular NP-hard case of UCPP instances with an equal number of pieces and types \((p = q)\). Given a particular class \(C\) of UCPP instances, let us consider the instances obtained from \(C\) by keeping only one location per type: if the resulting instances are such that \(p = q\) with at least three tools, then the UCPP is still NP-hard on \(C\). We can then remark that most of the UCPP instances will correspond to NP-hard cases. We will focus in Section 4.3 to instances with particular structures.

4.3. Particular cases

The following additional definitions are useful to further characterize instances of the UCPP.

A type is substitutable (resp. non-substitutable) if there exist several locations (resp. there exists only one location) in \(V\) with a piece of this type. Hence given a location \(v \in U \cup C\) with a piece of a non-substitutable type, the piece located at \(v\) has a predetermined destination (a “one-to-one assignment”).

Note that pieces of a same type \(t \in \{1, \ldots, q\}\) share the same possible destinations. Consequently, in the corresponding graph model, the subset of arcs whose tails correspond to a piece of type \(t\) in the initial assignment induces a biclique of the graph \(G\) (a graph \(H = (W^+ \cup W^-, F)\) is a biclique if \(F\) is the set of all arcs \((i, j)\) with \(i \in W^+\) and \(j \in W^-\)).

A UCPP instance of \(p\) pieces and \(|V|\) locations will be referred as a \((r, q)\)-instance where \(r\) is the number of tools and \(q\) is the number of types. Note that when \(q = p\) each piece has a distinct non-substitutable type.

We focus on the following particular cases of the UCPP.

- \((r, q = p)\)-instances

  The graph of a feasible \((r, q = p)\)-instance is a set of vertex-disjoint paths. Consequently, from the proof of Lemma \[\text{the UCPP on } (r \geq 3, q = p)\)-instances is NP-hard. The UCPP on \((r = 2, q = p)\)-instances clearly reduces to the SCDSP for sequences over an alphabet of size 2 which can be solved in polynomial time. It is worth noting that the heuristic in operation at EDF for the NFRP solves such instances to optimality when \(r = 2\).
• \((r, q)\)-instances with only paths of size 1

We now consider UCPP instances where every vertex contains (resp. requires) a piece in the initial (resp. final) assignment and does not require (resp. contain) a piece in the final (resp. initial) assignment. Their corresponding graphs then feature only paths of size 1. More precisely this graph is the set of 3 vertex-disjoint bicliques \(H_t = (W_t^+ \cup W_t^-, F_t), t \in \{1, \ldots, q\}\), where \(W_t^+\) (resp. \(W_t^-\)) is the set of vertices containing a piece of type \(t\) in the initial assignment (resp. requiring a set of type \(t\) in the final assignment). Such an instance admits a trivial optimal permutation obtained as follows: an optimal permutation is an arbitrary sequence of \(q\) subsequences \(\sigma_t, t \in \{1, \ldots, q\}\), so that every \(\sigma_t\) is an arbitrary order on an arbitrary matching of \(H_t\) i.e. a set of \(|W_t^+|\) vertex-disjoint arcs of \(F_t\). Thus an optimal permutation can be computed in linear time.

Such instances correspond to the special case of the NFRP where all assemblies in the core are replaced with fresh ones. In practice only a fraction of the fuel assemblies is replaced.

• \((r = 2, q = 2)\)-instances

In this special case, there are only two substitutable types which require distinct tools. In Section 7.3 we will prove that a dynamic programming algorithm based on dominance properties can solve this case in \(O(p)\). This case is worth investigating when considering that all pieces are with a substitutable type and each type requires the use of a different tool. In the NFRP, there has been so far only one substitutable type. However there are some symmetries in the geometry of the core and in the use of the inserts that make this specific case relevant from a practical point of view.

5. Dominance properties

In the previous sections, some specific features of the UCPP instances enable us to exhibit interesting polynomial cases. Since such instances come from the NFRP, the idea is to analyze their underlying structural properties and use them to reduce the set of solutions to explore. In this section, some dominance
properties are introduced.

Let \( S \) be the set of all permutations. A subset \( S' \subset S \) of permutations is dominant if for each \( \sigma \in S \) there exists \( \sigma' \in S' \) with a cost less or equal to that of permutation \( \sigma \). We introduce two sets of dominant permutations.

- **E-permutations**

  Consider a permutation \( \sigma = (\Sigma_1, \ldots, \Sigma_\rho) \) where \( \Sigma_k, k \in \{1, \ldots, \rho\} \), is a subsequence of consecutive moves such that every move of \( \Sigma_k \) is using the same tool. We define an **E-permutation** as a permutation \( \sigma \) when for every \( k, l \in \{1, \ldots, \rho\}, k + 1 < l \) with \( \Sigma_k \) and \( \Sigma_l \) using the same tool, then for every \( (i, j) \in \Sigma_l \), location \( j \) is still containing a piece after the completion of \( \Sigma_k \). In fact E-permutations can be seen as permutations where pieces are moved as early as possible in the sequence. A tool changeover is then considered only if no additional move using the current tool is possible.

**Property 2.** The set of E-permutations is dominant.

**Proof.** Let \( \sigma = (\Sigma_1, \ldots, \Sigma_\rho) \) be a permutation which is not an E-permutation. By definition there exists a move \( (i, j) \in \Sigma_l \) with \( k, l \in \{1, \ldots, \rho\}, k + 1 < l \) such that \( \Sigma_k \) and \( \Sigma_l \) are using the same tool and \( j \) is free of piece after \( \Sigma_k \). Consequently move \( (i, j) \) can be shifted from \( \Sigma_l \) to the last position of \( \Sigma_k \) and the resulting sequence, denoted by \( \sigma' \), is still a permutation. Moreover there are two cases: either \( \Sigma_l \) contains at least two moves then \( \sigma' \) and \( \sigma \) incur the same cost, or \( \Sigma_l \) contains only one move then \( \sigma' \) incurs one less tool changeover than \( \sigma \). By iteratively performing such shifting operations, an E-permutation can then be derived with a cost less or equal to the cost of \( \sigma \). \( \square \)

For example, by referring to the instance presented in Figure 3, permutation \(((4,8),(6,7),(5,6),(2,3),(1,2))\) is not an E-permutation and incurs four tool changeovers while \(((4,8),(6,7),(2,3),(5,6),(1,2))\) is an E-permutation incurring only two tool changeovers.

- **CU-permutations**

  A permutation \( \sigma = ((i_1,j_1), \ldots, (i_p,j_p)) \) is a **CU-permutation** if for every
\[ k, l \in \{1, \ldots, p\} \text{ with } i_k \in C, i_l \in U \text{ and } \tau^0(i_k) = \tau^0(i_l) \text{ then } k < l. \]

In a CU-permutation the moves of the pieces with a given type will be performed from locations of \( C \) before locations of \( U \). Moving a piece from a location of \( C \) makes the location ready to receive a new piece.

**Property 3.** The set of CU-permutations is dominant.

**Proof.** Let \( \sigma = ((i_1, j_1), \ldots, (i_p, j_p)) \) be a permutation which is not a CU-permutation. Then there exist \( k, l \in \{1, \ldots, p\} \) with \( k > l \) such that \( i_k \in C, i_l \in U \) and \( \tau^0(i_k) = \tau^0(i_l) \). Note that by definition, \( \tau^f(j_k) = \tau^0(i_k) \) and \( \tau^f(j_l) = \tau^0(i_l) \), and thus \( (i_k, j_l) \) \( \text{ and } (i_l, j_k) \) belong to the move set \( A \). Define \( \sigma' \) as the sequence of moves obtained from \( \sigma \) by exchanging moves \((i_k, j_k)\) by \((i_l, j_k)\) and \((i_l, j_l)\) by \((i_k, j_l)\). By construction sequence \( \sigma' \) clearly satisfies property i) and ii) of Property 1. Moreover property iii) is also satisfied by \( \sigma' \). Move \((i_k, j_l)\) (resp. \((i_l, j_k)\)) can actually be performed since \( j_l \) (resp. \( j_k \)) is free of piece after the first \( l - 1 \) (resp. \( k - 1 \)) moves in \( \sigma \). Since \( i_k \in C \), then there exists \( m \) with \( k < m \leq p \) such that \( j_m = i_k \). Thus \((i_m, j_m)\) appears after move \((i_l, j_k)\) in \( \sigma' \).

In the sequence \( \sigma' \) location \( i_k \) is actually free of piece before receiving the piece coming from \( i_m \). Moreover \( i_l \in U \) is not involved in another move in \( \sigma' \). By Property 1, since the rest of the sequence \( \sigma' \) is unchanged with respect to \( \sigma \), \( \sigma' \) is a permutation with the same cost. By iteratively performing such exchange operations, a CU-permutation with the same cost can then be derived. \( \square \)

For example, by referring to the instance presented in Figure 3, permutation \( ((4,8), (6,7), (2,3), (5,6), (1,2)) \) is not a CU-permutation, while \( ((6,7), (4,8), (2,3), (5,6), (1,2)) \) is a CU-permutation. Recall that \( 4 \in U \) and \( 6 \in C \), and both vertices contain a piece of the same type in the initial assignment, then 6 should appear before 4 in a CU-permutation.

6. Breaking symmetries of UCPP instances

We now exhibit a symmetry relation between permutations whose equivalence classes contain permutations with the same cost. This property will reduce
the size of the permutations set to be considered to find an optimal permutation for the UCPP.

6.1. \( \tau \)-symmetry

Consider a permutation \( \sigma = ((i_1,j_1), \ldots, (i_p,j_p)) \) of a UCPP instance. We define an operation \( T \) as a function that associates permutation \( \sigma \) to the sequence of pairs of types for pieces located at each location \( i_k, 1 \leq k \leq p \), i.e., \( T(\sigma) = ((\tau^0(i_1), \tau^f(i_1)), \ldots, (\tau^0(i_p), \tau^f(i_p))) \). Recall that by definition \( \tau^0(i_l) = \tau^f(j_l) \) for \( l \in \{1, \ldots, p\} \).

Given two permutations \( \sigma = ((i_1,j_1), \ldots, (i_p,j_p)) \) and \( \sigma' = ((i'_1,j'_1), \ldots, (i'_p,j'_p)) \), \( \sigma \) and \( \sigma' \) are \( \tau \)-symmetric if \( T(\sigma) = T(\sigma') \). By construction such permutations have the same cost.

For example, by referring to the instance shown in Figure 3, consider permutations \( \sigma_1 = ((2,3), (6,7), (4,8), (1,2), (5,6)) \) and \( \sigma_2 = ((2,3), (6,7), (4,8), (5,6), (1,2)) \) which are both CU-E optimal permutations with two tool changeovers. Since \( T(\sigma_1) = T(\sigma_2) = ((b,a), (c,a), (c,-), (a,-), (a,-)) \), then \( \sigma_1 \) and \( \sigma_2 \) are \( \tau \)-symmetric.

6.2. Using \( \tau \)-symmetry

Since two \( \tau \)-symmetric permutations have the same cost, we will explore only one permutation per class. In fact we consider sequences of pairs of types instead of sequences of moves.

Given a permutation \( \sigma = ((i_1,j_1), \ldots, (i_p,j_p)) \) it is straightforward to obtain the associated sequence of pairs of types \( T(\sigma) \). However given a sequence of pairs of types \( s = ((t^0_1, t^f_1), (t^0_2, t^f_2), \ldots, (t^0_p, t^f_p)) \), it could happen that no permutation \( \sigma \) such that \( T(\sigma) = s \) exists. Theorem \( \overline{1} \) will provide a characterization for the sequences of pairs of types that correspond to a permutation.

Given a set \( M \) of pairs of types and a type \( t \), let us define the sum

\[
\Pi(M, t) = |\{j \in N : \tau^f(j) = t\}| + |\{(t^0, t^f) \in M : t^f = t\}| - |\{(t^0, t^f) \in M : t^0 = t\}|
\]
Note that in the case where \( M \) corresponds to a partial permutation \( \sigma \) of \( |M| \) moves, \( \Pi(M, t) \) is equal to the number of free locations requiring a piece of type \( t \) after completing the moves. More specifically this sum consists of three terms:

- the first term \( \left| \{ j \in N : \tau^f(j) = t \} \right| \) corresponds to the number of locations originally free and requiring a piece of type \( t \),

- the second (resp. third) term \( \left| \{(t^0, t^f) \in M : t^f = t \} \right| \) (resp. \( \left| \{(t^0, t^f) \in M : t^0 = t \} \right| \)) corresponds to the number of pieces that have already been moved from (resp. to) a location requiring a piece of type \( t \).

Given a sequence of pairs of types \( s = ((t^0_1, t^f_1), (t^0_2, t^f_2), \ldots, (t^0_p, t^f_p)) \), we define the set of pairs of types \( M_k(s) = \{(t^0_i, t^f_i), (t^0_k, t^f_k), \ldots, (t^0_p, t^f_p)\} \) for every \( k \in \{1, \ldots, p\} \) and set \( M_0(s) = \emptyset \). The characterization is given by the following theorem.

**Theorem 1.** A sequence of pairs of types \( s = ((t^0_1, t^f_1), (t^0_2, t^f_2), \ldots, (t^0_p, t^f_p)) \) corresponds to a permutation if and only if \( \Pi(M_{k-1}(s), t^0_k) > 0 \) for every \( k \in \{1, \ldots, p\} \).

**Proof.** Consider a permutation \( \sigma = ((i_1, j_1), \ldots, (i_p, j_p)) \) and its corresponding sequence of pairs of types \( T(\sigma) \). Let \( k \in \{1, \ldots, p\} \). If \( k = 1 \), then \( \Pi(M_{k-1}(s), t^0_k) = \Pi(\emptyset, \tau^0(i_1)) \). This holds since there is initially at least one location free of piece that requires a piece of type \( \tau^0(i_1) \), i.e. \( \Pi(M_{k-1}(s), t^0_k) > 0 \) for \( k = 1 \). If \( k > 1 \) it is worth noting that by property iii) of Property 1 \( \Pi(i_k) \) is free of piece after the first \( k - 1 \) moves. Hence there is at least one location free of piece that requires a piece of type \( \tau^0(i_k) \), then \( \Pi(M_{k-1}(s), t^0_k) > 0 \).

Conversely consider a sequence of pairs of types \( s = ((t^0_1, t^f_1), \ldots, (t^0_p, t^f_p)) \) with \( \Pi(M_{k-1}(s), t^0_k) > 0 \) for every \( k \in \{1, \ldots, p\} \). The claim is that a permutation \( \sigma = ((i_1, j_1), \ldots, (i_p, j_p)) \) can be constructed such that \( t^0_k = \tau^0(i_k) \) and \( t^f_k = \tau^f(i_k) \) for every \( k \in \{1, \ldots, p\} \). This can be shown by induction on \( k \). For \( k = 1 \), since \( \Pi(\emptyset, t^0_1) = \left| \{ j \in N : \tau^f(j) = t^0_1 \} \right| > 0 \), then a move \((i_1, j_1)\) can be chosen arbitrarily in \( A \) so that \( \tau^0(i_1) = t^0_1 \), \( \tau^f(i_1) = t^f_1 \) and \( \tau^0(j_1) = - \). Assume that given \( k \) with \( 1 < k \leq p \), there exists a partial permutation \( \sigma_{k-1} = ((i_1, j_1), \ldots, (i_{k-1}, j_{k-1})) \) associated to the subsequence of pairs...
of types after \( k - 1 \) moves. Since \( \Pi(M_{k-1}(s), t^0_k) > 0 \), there exists a location \( j_k \) with \( \tau_f(j_k) = t^0_k \) free of piece after these \( k - 1 \) moves. Therefore, since the number of pieces of each type in the initial and final assignments is the same, there is a move \((i_k, j_k)\) distinct from the moves of \( \sigma_{k-1} \) and such that \( \tau_0(i_k) = t^0_k \) and \( \tau_f(i_k) = t^f_k \). Consequently \(((i_1, j_1), \ldots, (i_{k-1}, j_{k-1}), (i_k, j_k))\) is a partial permutation of \( k \) distinct moves. It follows that the result holds for \( k = p \). □

Following the sketch of the latter proof Algorithm II will construct a permutation from a sequence of pairs of types in \( O(p^2) \).

**Algorithm 1**

**Input:** a sequence of pairs of types \( s = ((t^0_1, t^f_1), \ldots, (t^0_p, t^f_p)) \).

that satisfies Theorem I

**Output:** a permutation \( \sigma = ((i_1, j_1), \ldots, (i_p, j_p)) \) such that \( T(\sigma) = s \).

\( \sigma \leftarrow () \)

for \( k \) from 1 to \( p \) do,

Choose (arbitrarily) a move \((i_k, j_k) \notin \sigma\) such that \( \tau_0(i_k) = t^0_k, \tau_f(i_k) = t^f_k \) and location \( j_k \) free of piece after \( k - 1 \) moves.

\( \sigma \leftarrow (\sigma, (i_k, j_k)) \).

end for

7. Dynamic programming algorithm

In this section a dynamic programming algorithm will construct a sequence of pairs of types leading to an optimal permutation.

7.1. Description of the algorithm

The key idea of the dynamic programming algorithm is to relax the order of the sequence of pairs of types, thus considering a subset of pairs of types instead of a sequence. A *state* \((k, M, (t^0, t^f))\) of the dynamic programming algorithm consists of an integer \( k \in \{1, \ldots, p\} \), a subset \( M \) of \( k - 1 \) pairs of types and
an additional pair \((t^0, t^f)\). In fact a state \((k, M, (t^0, t^f))\) corresponds to the set of sequences of \(k\) pairs of types obtained by ordering the \(k - 1\) pairs of \(M\) and ending with the latter pair \((t^0, t^f)\).

Let \(C = \{(\tau^0(i), \tau^f(i)) \mid \exists (i, j) \in A\}\) be the candidate set. Such a set contains all the pairs of types to which a move can correspond. The algorithm starts by enumerating states \((1, \emptyset, (t^0, t^f))\), for every \((t^0, t^f)\) in \(C\). The corresponding cost \(g\) is \(g(1, \emptyset, (t^0, t^f)) = 1\).

Given all the previously generated states for a given \(k \in \{1, \ldots, p - 1\}\), the step \(k\) of the dynamic programming algorithm is to generate states with \(k + 1\) moves using Algorithm 2.

Algorithm 2 Step \(k\) of the dynamic programming algorithm

for every previously generated state \((k, M', (t'^0, t'^f))\) do

Compute the set \(L\) of remaining candidate pairs of types

i.e. \(L \leftarrow \{(t^0, t^f) \in C \setminus M\text{ such that } \Pi(M, t^0) > 0\}\)

for every \((t^0, t^f) \in L\) do

\(M \leftarrow M' \cup \{(t'^0, t'^f)\}\)

Generate state \((k + 1, M, (t^0, t^f))\)

end for

end for

Moreover, the cost \(g\) of every generated state is calculated recursively as follows:

\[
g(k + 1, M, (t^0, t^f)) = \min_{(\tilde{t}^0, \tilde{t}^f) \in M} \left\{ \sigma_{\Pi}^{\Omega(t^0)} + g(k, M \setminus \{(t^0, \tilde{t}^f)\}, (\tilde{t}^0, \tilde{t}^f)) \right\}.
\]

By exploring every solution, this dynamic programming algorithm leads, for \(k = p\), to a sequence \(s^*\) of pairs of types minimizing cost \(g\) and satisfying Theorem 1. Consequently there exists a permutation \(\sigma^*\) with \(g(s^*)\) tool changeovers, that can be obtained from \(s^*\) using Algorithm 1 and we have the following result.

Theorem 2. The dynamic programming algorithm produces a sequence of pairs of types corresponding to an optimal permutation using Algorithm 1.

---

22
7.2. Pruning rules using dominance properties

Unfortunately the dynamic programming algorithm enumerates an exponential number of states in the general case. The dominance properties introduced in the previous section will be used to prune some useless states. The following rules will be applied at every step of the algorithm to reduce the size of the remaining candidate pairs of types sets $L$ after the third line of Algorithm 2. Consider the following two pruning rules.

CU-dominance rule : for each $t^0 \in \{1, \ldots, L\}$ do
if there exists $(t^0, t^f) \in L$ such that $t^f \neq -$ then
$L \leftarrow L \setminus \{(t^0, -) \in L\}$
end if
end for

E-dominance rule : if there exists $(t^0, t^f) \in L$ such that $\Omega(t^0) = \Omega(t^f)$ then
for each $t^0 \in \{1, \ldots, L\}$ with $\Omega(t^0) \neq \Omega(t^f)$ do
$L \leftarrow L \setminus \{(t^0, t^f) \in L\}$
end for
end if

Using the CU-dominance rule, for every type a step will always consider moves from locations of $C$ before any move from locations of $U$. Using the E-dominance rule, for every type a step will always consider moves using the current tool before moves involving a tool changeover. Using theses two pruning rules at each step of the dynamic programming algorithm the resulting permutation is a CU-E-permutation. Consequently, from Properties 3 and 2 a permutation computed by this algorithm is optimal.

7.3. A polynomial case

In this section, we use the dynamic programming algorithm to derive a polynomial case of the UCPP. Since the UCPP is NP-hard for $(r \geq 3, q = p)$-instances and is polynomial for $(r = 2, q = p)$-instances, an important question
is to state the complexity of the UCPP for \((r = 2, q = 2)\)-instances. This case is of practical interest for the NFRP. Theorem 3 states the complexity of this case.

**Theorem 3.** The UCPP is polynomial for \((r = 2, q = 2)\)-instances.

**Proof.** We will prove that the dynamic programming algorithm using CU- and E-dominance pruning rules produces exactly two states per step for the UCPP instances with two tools and two types.

Let us use \(t \in \{1, 2\}\) to denote these two types. Then the algorithm starts exploring a candidate set \(C\) of at most four pairs of types: pairs \((1, 2)\) and \((1, -)\) involving tool 1 and pairs \((2, 1)\) and \((2, -)\) involving tool 2. From the CU-dominance rule, pair \((1, 2)\) (resp. \((2, 1)\)), if it exists, is used prior to pair \((1, -)\) (resp. \((2, -)\)). It follows that the first step only produces two states, one corresponding to the move of a piece with tool 1 and the other with tool 2. Let us now consider the arborescence of the generated states corresponding to, w.l.o.g., the move with tool 1 as root. From the E-dominance rule, this arborescence is a path: indeed the current tool should be used until no more move could be performed using it. Hence, the dynamic programming algorithm only produces at most two states per step. Consequently an optimal solution can be computed in \(O(p)\). \(\square\)

The solution of this special case features a very particular structure that can be described precisely. The optimal cost can be given as follows. Let \(p_t, t \in \{1, 2\}\) be the number of pieces of type \(t\). Let \(\eta_t, t \in \{1, 2\}\) be the number of locations which are initially free of pieces and require a piece of type \(t\), i.e. \(\eta_t = |\{i \in V \mid \tau^0(i) = - \text{ and } \tau^f(i) = t\}|\). Since this instance supports at least one feasible solution, \(\eta_1 > 0\) and \(\eta_2 > 0\). The optimal cost for an \((r = 2, q = 2)\)-instance is then \(c^* = \min(c_1, c_2)\), where \(c_t = \left\lceil \frac{\eta_t}{\eta_1 + \eta_2} \right\rceil + \left\lceil \frac{p_t - \eta_t}{\eta_1 + \eta_2} \right\rceil + \left\lceil \frac{p - p_t}{\eta_1 + \eta_2} \right\rceil\), \(t \in \{1, 2\}\).
8. Experimental Results

In this section, some computational results using the dynamic programming algorithm to solve instances of the UCPP are presented. The algorithm has been implemented in C++. The experiments were carried out on a 8 core Intel i7-2600K running at 3.40GHz equipped with 16 GB RAM, and operating system Linux Debian GNU Sid 64bits. The results reported are only with respect to instances that were solved within a 3600 second time limit and 16 GB RAM.

8.1. Instances

The UCPP instances are realistic NFRP instances with respect to the fuel operation management rules both present and future. For comparison purpose we use the same generation scheme as that described in Bendotti et al. (2015).

We considered instances with a number of pieces $24 \leq p \leq 216$ with an increment of 12. The fraction $f$ of renewable locations is such that $n = (1 + f)p$. Moreover, $|C| = (1 - f)p$ and $|N| = |U| = fp$. In practice, for NFRP instances, $f$ is either $\frac{1}{3}$, $\frac{1}{4}$ or sometimes $\frac{1}{6}$. The number of tools $r$ ranges mostly from 2 to 3 and for some instances is 4. The number of pieces handled by the same tool (pieces of the same type) varies from $\frac{1}{6}p$ to $\frac{2}{3}p$ ($\frac{1}{6}p$ to $\frac{1}{3}p$). The following generation scheme is used. For given $p$, $q$, $r$, we first randomly generate the initial type of pieces $\tau^0$ dispatched over $V$ depending on the proportion of each type, the tools required, and the fraction of renewable locations $f$. The required type of pieces $\tau^f$ is generated at random such that each type of piece appears with the same cardinality as in $\tau^0$. For the following tests, we ensure the feasibility of every instance using a polynomial algorithm given in Bendotti et al. (2015).

We refer to an $(r, q)$-instance using the label described in Bendotti et al. (2015). Recall that $q = q_s + q_{ns}$ where $q_s$ (resp. $q_{ns}$) denotes the number of substitutable (resp. non-substitutable) types. The other fields refer to the characteristics as follows $p-r-q_s-q_{ns}-f$. For example, 72-3-2-11-$\frac{1}{3}$ stands for a $(3, 13)$-instance with 72 pieces, 3 tools, 13 types (2 substitutable types and
11 non-substitutable types), and a fraction of renewable locations equal $\frac{1}{3}$.

Note that the corresponding instance with all the same characteristics but non-substitutable types would be an $(r = 3, q = 2)$-instance, and would have the following label 72-3-2-0-$\frac{1}{3}$.

In the NFRP, there has been so far only one substitutable type. In this article, the aim is to solve instances with more than one substitutable type. For this purpose we derive new instances from the realistic ones (with $q_s = 1$) taking benefits from some symmetries in the geometry of the core and in the use of the inserts. Consequently the instances considered in this article correspond only to random realistic instances (denoted by RND in Bendotti et al. (2015)). For each of these instances (with $q_s = 1$), we derive a new instance where a subset of pieces with a non-substitutable type are converted into a subset of pieces with a new substitutable type, thus leading to instances with two substitutable types ($q_s = 2$). Such a conversion is a relaxation of the 1-to-1 assignment corresponding to a non-substitutable type. Instances with three substitutable types can be derived similarly, e.g. instance 36-2-1-18-$\frac{1}{3}$ is with $q_s = 1, q_{ns} = 18$ and the number of pieces with a substitutable type is $p - q_{ns} = 18$, while instance 36-2-2-6-$\frac{1}{3}$ is with $q_s = 2, q_{ns} = 6$ and the number of pieces with a substitutable type is $p - q_{ns} = 30$. There are 12 pieces out of 30 which are of the second substitutable type in 36-2-2-6-$\frac{1}{3}$, whereas they are of a non-substitutable type in 36-2-1-18-$\frac{1}{3}$. It follows that the difference between the two sets of instances shows up in the ratio between the number of pieces of substitutable types and non-substitutable types. Note that in the case of instances with one substitutable type, the number of pieces with a non-substitutable type exceeds the number of pieces with a substitutable type.

In the next subsection, our experimental results are presented in Tables 1 to 5. The first column of the tables presents the label of the instance. The other entries of the tables give some statistical results about the dynamic programming algorithm:
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>Optimal cost (number of tool changeovers),</td>
</tr>
<tr>
<td>$n_s$</td>
<td>Maximal number of states for a given step (in thousand of states),</td>
</tr>
<tr>
<td>$n_t$</td>
<td>Total number of states (in thousand of states),</td>
</tr>
<tr>
<td>CPU</td>
<td>Total CPU time (in seconds).</td>
</tr>
</tbody>
</table>

### 8.2. Efficiency of pruning rules

In a preliminary experimental work, a standard dynamic programming algorithm that does not account for the $\tau$-symmetry has been implemented. This algorithm could not solve instances with $p = 24$ pieces within a space limit of 16 GB RAM and a time limit of one hour. The following experimental results show clearly the efficiency of the proposed dynamic programming algorithm derived from the $\tau$-symmetry property.

Table 1 is dedicated to show the effectiveness of the pruning rules derived from the dominance properties presented in Section 5. The idea is to compare the number of states produced by the proposed dynamic programming algorithm when first using no dominance rule, and then using $CU$- and then $E$ dominance rule, and finally using both of them. The table shows the experimental results over a large set of small size instances relative to $p = 24$. The CPU times are not reported as they are all less than a second.

Both $CU$- and $E$-dominance rules have clearly a deep impact on the number of generated states for the dynamic programming algorithm. The average values of $n_s$ (resp. $n_t$) are reduced by 53\% (resp. 53.7\%) using $CU$-dominance rule. Similarly the average values of $n_s$ (resp. $n_t$) are reduced by 79.9\% (resp. 80.3\%) using $E$-dominance rule.

The $CU$-dominance rule seems to be more effective when $f$ gets larger. This is expected as the $CU$-dominance would have no effect when either $U$ or $C$ is empty. From the considered values of $f$, it follows that $|C| > |U|$. Therefore the larger $U$, the more effective the $CU$-dominance rule. The $E$-dominance rule is also more effective when $f$ gets larger. Compared to the $CU$-dominance, it seems to be more effective when $q$ increases as well. Recall the $E$-dominance rule...
along with the \(CU\)-dominance rule enabled us to show that the \((r = 2, q = 2)\)-instances can be solved in linear time and space (see Section 7.3).

Beyond the individual effect of the dominance rules, their combined use is very effective as the corresponding results (see last two columns with label \(CU-E\) in Table 1) show a significant improvement: The average values of \(n_s\) (resp. \(n_t\)) are reduced by 92.9\% (resp. 93.4\%).

<table>
<thead>
<tr>
<th>(p-r-q_s-q_n-f)</th>
<th>(c)</th>
<th>(n_s)</th>
<th>(n_t)</th>
<th>(n_s)</th>
<th>(n_t)</th>
<th>(n_s)</th>
<th>(n_t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>24-3-3-0-(\frac{1}{3})</td>
<td>5</td>
<td>1.71</td>
<td>17.53</td>
<td>0.10</td>
<td>1.22</td>
<td>0.73</td>
<td>6.50</td>
</tr>
<tr>
<td>24-3-3-0-(\frac{1}{3})</td>
<td>6</td>
<td>1.99</td>
<td>21.50</td>
<td>0.20</td>
<td>2.45</td>
<td>1.45</td>
<td>13.82</td>
</tr>
<tr>
<td>24-3-3-0-(\frac{1}{3})</td>
<td>7</td>
<td>0.39</td>
<td>5.50</td>
<td>0.07</td>
<td>1.01</td>
<td>0.35</td>
<td>3.83</td>
</tr>
<tr>
<td>24-4-4-0-(\frac{1}{3})</td>
<td>7</td>
<td>70.77</td>
<td>517.45</td>
<td>3.22</td>
<td>24.81</td>
<td>33.90</td>
<td>221.90</td>
</tr>
<tr>
<td>24-2-2-4-(\frac{1}{3})</td>
<td>4</td>
<td>1.77</td>
<td>18.28</td>
<td>0.21</td>
<td>2.42</td>
<td>0.18</td>
<td>1.92</td>
</tr>
<tr>
<td>24-2-2-5-(\frac{1}{3})</td>
<td>5</td>
<td>3.52</td>
<td>38.50</td>
<td>0.78</td>
<td>9.01</td>
<td>0.60</td>
<td>6.96</td>
</tr>
<tr>
<td>24-2-2-3-(\frac{1}{3})</td>
<td>6</td>
<td>0.33</td>
<td>4.65</td>
<td>0.09</td>
<td>1.29</td>
<td>0.18</td>
<td>1.99</td>
</tr>
<tr>
<td>24-3-2-4-(\frac{1}{3})</td>
<td>5</td>
<td>2.53</td>
<td>26.25</td>
<td>0.30</td>
<td>3.51</td>
<td>0.65</td>
<td>5.92</td>
</tr>
<tr>
<td>24-3-2-5-(\frac{1}{3})</td>
<td>7</td>
<td>5.04</td>
<td>55.16</td>
<td>1.15</td>
<td>13.33</td>
<td>2.01</td>
<td>20.02</td>
</tr>
<tr>
<td>24-3-2-3-(\frac{1}{3})</td>
<td>7</td>
<td>0.42</td>
<td>5.95</td>
<td>0.12</td>
<td>1.71</td>
<td>0.37</td>
<td>4.12</td>
</tr>
<tr>
<td>24-2-1-12-(\frac{1}{3})</td>
<td>4</td>
<td>45.88</td>
<td>334.79</td>
<td>15.19</td>
<td>104.66</td>
<td>2.02</td>
<td>21.42</td>
</tr>
<tr>
<td>24-2-1-14-(\frac{1}{3})</td>
<td>5</td>
<td>54.23</td>
<td>411.91</td>
<td>37.06</td>
<td>278.68</td>
<td>3.68</td>
<td>34.74</td>
</tr>
<tr>
<td>24-2-1-13-(\frac{1}{3})</td>
<td>6</td>
<td>16.24</td>
<td>138.82</td>
<td>11.40</td>
<td>96.06</td>
<td>7.66</td>
<td>53.75</td>
</tr>
<tr>
<td>24-2-1-12-(\frac{1}{3})</td>
<td>5</td>
<td>77.08</td>
<td>543.69</td>
<td>28.78</td>
<td>193.00</td>
<td>7.48</td>
<td>53.58</td>
</tr>
<tr>
<td>24-2-1-14-(\frac{1}{3})</td>
<td>7</td>
<td>62.88</td>
<td>457.75</td>
<td>43.72</td>
<td>322.77</td>
<td>3.30</td>
<td>23.55</td>
</tr>
<tr>
<td>24-2-1-13-(\frac{1}{3})</td>
<td>9</td>
<td>29.01</td>
<td>197.12</td>
<td>21.48</td>
<td>148.24</td>
<td>7.84</td>
<td>43.05</td>
</tr>
<tr>
<td>24-3-1-12-(\frac{1}{3})</td>
<td>6</td>
<td>66.10</td>
<td>480.34</td>
<td>21.48</td>
<td>147.78</td>
<td>7.61</td>
<td>60.28</td>
</tr>
<tr>
<td>24-3-1-14-(\frac{1}{3})</td>
<td>7</td>
<td>79.79</td>
<td>601.01</td>
<td>54.13</td>
<td>403.31</td>
<td>15.33</td>
<td>120.69</td>
</tr>
<tr>
<td>24-3-1-13-(\frac{1}{3})</td>
<td>9</td>
<td>20.50</td>
<td>172.52</td>
<td>14.32</td>
<td>119.59</td>
<td>13.32</td>
<td>100.76</td>
</tr>
<tr>
<td>Average values</td>
<td></td>
<td>28.43</td>
<td>213.01</td>
<td>13.36</td>
<td>98.68</td>
<td>5.72</td>
<td>42.04</td>
</tr>
</tbody>
</table>

Table 1: Effectiveness of the pruning rules for a test set with \(p = 24\).
results relative to \((r, q)\)-instances with both substitutable and non-substitutable types.

<table>
<thead>
<tr>
<th>(p-r-q_s-q_n\cdot f)</th>
<th>(c)</th>
<th>CPU</th>
<th>(n_s)</th>
<th>(n_t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>36-2-1-18-(\frac{1}{4})</td>
<td>4</td>
<td>2.65</td>
<td>41.53</td>
<td>284.18</td>
</tr>
<tr>
<td>36-2-1-21-(\frac{1}{3})</td>
<td>5</td>
<td>17.16</td>
<td>218.46</td>
<td>1889.18</td>
</tr>
<tr>
<td>36-2-1-20-(\frac{1}{6})</td>
<td>6</td>
<td>60.98</td>
<td>1040.00</td>
<td>6554.47</td>
</tr>
<tr>
<td>36-3-1-18-(\frac{1}{4})</td>
<td>6</td>
<td>3.20</td>
<td>59.56</td>
<td>394.58</td>
</tr>
<tr>
<td>36-3-1-21-(\frac{1}{4})</td>
<td>7</td>
<td>53.43</td>
<td>679.47</td>
<td>6085.17</td>
</tr>
<tr>
<td>36-3-1-20-(\frac{1}{6})</td>
<td>8</td>
<td>36.40</td>
<td>2938.29</td>
<td>24692.67</td>
</tr>
<tr>
<td>48-2-1-24-(\frac{1}{4})</td>
<td>4</td>
<td>205.85</td>
<td>1928.97</td>
<td>14571.06</td>
</tr>
<tr>
<td>48-3-1-24-(\frac{1}{3})</td>
<td>6</td>
<td>208.94</td>
<td>2379.81</td>
<td>17006.31</td>
</tr>
</tbody>
</table>

Table 2: \((r, q)\)-instances with \(q_s = 1\)

We limit the experimental analysis to instances with two and three tools as we could not solve the corresponding instances with four tools within the CPU and memory limits. A first remark is that instances with \(q_s = 2\) considered in Table 3 are solved more efficiently than those with \(q_s = 1\) considered in Table 2. Referring to instance 36-2-2-6-\(\frac{1}{3}\), the computing time is 0.02 second, while referring to instance 36-2-1-18-\(\frac{1}{4}\) it is 2.65 seconds, which is consistent with the results presented in Table 1: in the case of instances with one substitutable type, the number of pieces with a non-substitutable type exceeds the number of pieces with a substitutable type. The experimental results show clearly that for these instances the number of generated states explodes. For example, the total number of states for instance 48-2-1-24-\(\frac{1}{4}\) is 14571059, whereas for 48-2-2-8-\(\frac{1}{3}\) it is 14017. The largest instance in term of number of pieces solved within the time and memory limits corresponds to \(p = 96\) when \(q_s = 2\), and to \(p = 48\) when \(q_s = 1\).

Tables 2 and 3 present instances with several values for the fraction \(f\) of the fuel assemblies that are renewed. We can remark that the instances are harder to solve when \(f\) is decreasing. The number of states that have to be
Table 3: \((r, q)\)-instances with \(q = 2\) and \(36 \leq p \leq 96\)

<table>
<thead>
<tr>
<th>(p-r-q_s-q_n-f)</th>
<th>(c)</th>
<th>CPU</th>
<th>(n_s)</th>
<th>(n_t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>36-2-2-6-(\frac{1}{3})</td>
<td>4</td>
<td>0.02</td>
<td>0.22</td>
<td>1.98</td>
</tr>
<tr>
<td>36-2-2-7-(\frac{1}{3})</td>
<td>5</td>
<td>0.09</td>
<td>1.01</td>
<td>11.19</td>
</tr>
<tr>
<td>36-2-2-5-(\frac{1}{6})</td>
<td>6</td>
<td>0.01</td>
<td>0.10</td>
<td>1.52</td>
</tr>
<tr>
<td>36-3-2-6-(\frac{1}{7})</td>
<td>5</td>
<td>0.01</td>
<td>0.11</td>
<td>1.86</td>
</tr>
<tr>
<td>36-3-2-7-(\frac{1}{7})</td>
<td>7</td>
<td>0.14</td>
<td>1.74</td>
<td>25.88</td>
</tr>
<tr>
<td>36-3-2-5-(\frac{1}{6})</td>
<td>8</td>
<td>0.06</td>
<td>0.47</td>
<td>9.27</td>
</tr>
<tr>
<td>48-2-2-8-(\frac{1}{3})</td>
<td>4</td>
<td>0.13</td>
<td>1.30</td>
<td>14.02</td>
</tr>
<tr>
<td>48-2-2-8-(\frac{1}{3})</td>
<td>5</td>
<td>0.59</td>
<td>6.80</td>
<td>92.91</td>
</tr>
<tr>
<td>48-2-2-6-(\frac{1}{6})</td>
<td>6</td>
<td>0.05</td>
<td>0.96</td>
<td>11.90</td>
</tr>
<tr>
<td>48-3-2-8-(\frac{1}{3})</td>
<td>5</td>
<td>0.03</td>
<td>0.44</td>
<td>8.14</td>
</tr>
<tr>
<td>48-3-2-8-(\frac{1}{3})</td>
<td>6</td>
<td>0.83</td>
<td>8.98</td>
<td>160.55</td>
</tr>
<tr>
<td>48-3-2-6-(\frac{1}{6})</td>
<td>7</td>
<td>0.26</td>
<td>3.44</td>
<td>52.27</td>
</tr>
<tr>
<td>60-2-2-9-(\frac{1}{3})</td>
<td>4</td>
<td>0.83</td>
<td>7.78</td>
<td>105.00</td>
</tr>
<tr>
<td>60-2-2-11-(\frac{1}{3})</td>
<td>5</td>
<td>15.51</td>
<td>122.68</td>
<td>1425.66</td>
</tr>
<tr>
<td>60-2-2-7-(\frac{1}{6})</td>
<td>6</td>
<td>0.32</td>
<td>3.57</td>
<td>50.63</td>
</tr>
</tbody>
</table>

Table 3: \((r, q)\)-instances with \(q = 2\) and \(36 \leq p \leq 96\)

explored depends on the number \(|C|\) of locations in \(C\), which increases when \(f\) decreases. Note that considering any \((r, q)\)-instance with \(q = 1\) and its corresponding instance with \(q = 2\), as described in Section 8.1, the cost of an optimal solution of the former is less or equal that of the latter. Recall the only difference between both instances is that the 1-to-1 assignment for a subset of seven pieces has been relaxed. The more pieces with a substitutable type, the more alternatives to find a solution with fewer tool changeovers. For instance an optimal solution with seven tool changeovers is obtained in Table 1 for instance 24-3-2-3-\(\frac{1}{6}\), while it is with nine tool changeovers for its corresponding instance 24-3-1-13-\(\frac{1}{6}\).

The proposed algorithm could solve such instances up to 48 pieces, as shown in Table 2. The algorithm proposed in Bendotti et al. (2015) outperforms this
dynamic programming approach since within the same CPU limit, instances up to 144 pieces are solved optimally. However, the method in [Bendotti et al. (2015)] fails to solve instances featuring large subsets of pieces with a substitutable type. For this latter case, the next section shows that our dynamic programming algorithm is able to solve them.

8.4. Solving \((r, q = r)\)-instances

Tables 4 to 5 show results relative to \((r, q = r)\)-instances for \(r = 3\) and \(r = 4\). Recall \((r = 2, q = 2)\)-instances are solved in linear time by the dynamic programming algorithm and the UCPP has been shown to be NP-hard for \((r = 3, q = p)\)-instances. It is interesting to focus on \((r = 3, q = 3)\)-instances to analyze the computational limits of the dynamic programming method. Interestingly Table 4 shows that \((r = 3, q = 3)\)-instances are efficiently solved to optimality. As for \((4, 4)\)-instances presented in Table 5, we notice an important increase in the number of states. This number is in fact exponential with respect to parameter \(p\). Note that for such instances we only present results relative to a fraction \(f = \frac{1}{3}\) since this case is easier and the number of states already exhibits an exponential increase with respect to \(p\).

It is worth noting that \((r = 3, p = 3)\)-instances were difficult to solve with the algorithm proposed in [Bendotti et al. (2015)], while they are very efficiently solved with the dynamic programming algorithm using \(CU\)-E pruning rules. Indeed the MIP approach from [Bendotti et al. (2015)] fails to solve \((r = 3, p = 3)\)-instances with \(p = 72\) pieces while the dynamic programming approach succeeds in solving the same subset of instances till \(p = 216\) pieces \((p > 216\) is not considered in the experiments). Such \((r = 3, p = 3)\)-instances with 216 pieces correspond to realistic instances of the NFRP problem that are of interest with respect to future fuel operation management rules.

9. Conclusions and Perspectives

In this article we propose a complexity analysis of the UCPP. We prove that the UCPP is NP-hard as soon as the number of tools reaches 3, and present
some particular cases where the problem is either NP-hard or polynomial. From a practical perspective, this complexity analysis allows us to characterize some classes of easy and difficult instances of the NFRP. We further analyzed the combinatorial structure of optimal solutions to the UCPP. We exhibited some symmetry and dominance properties that were major determinants to implement an efficient dynamic programming algorithm. In the particular case of \((r = 2, p = 2)\)-instances the proposed algorithm computes an optimal solution in linear time and space. Even though the complexity of the UCPP is NP-hard for 3-tool instances, experimental results clearly show that \((r = 3, q = 3)\)-
instances of the NFRP could still be solved efficiently. Furthermore, instances with more pieces with a substitutable type can be solved more efficiently. Such instances become of practical interest in the NFRP where more than one substitutable type can be considered.

In future works we could benefit from the symmetry and dominance properties to improve the performance of the integer linear program proposed in [Ben-dotti et al. (2015)]. This would be interesting in particular for \((r = 3, q)\)-instances where \(q\) is much less than \(p\) or equal three.

Another perspective will be to extend this approach to a more general problem where pieces can be moved several times.

**Acknowledgments:** The authors wish to thank Abdoul Bitar and Anisse Ismaili for their interesting discussions on the problem and their contributions to the experimental part.

<table>
<thead>
<tr>
<th>(p-r-q_n-f)</th>
<th>(c)</th>
<th>CPU</th>
<th>(n_s)</th>
<th>(n_t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>36-4-4-0-(\frac{1}{3})</td>
<td>7</td>
<td>0.05</td>
<td>0.79</td>
<td>12.32</td>
</tr>
<tr>
<td>48-4-4-0-(\frac{1}{3})</td>
<td>7</td>
<td>0.18</td>
<td>2.08</td>
<td>39.96</td>
</tr>
<tr>
<td>60-4-4-0-(\frac{1}{3})</td>
<td>6</td>
<td>0.72</td>
<td>4.97</td>
<td>122.98</td>
</tr>
<tr>
<td>72-4-4-0-(\frac{1}{3})</td>
<td>7</td>
<td>5.39</td>
<td>30.86</td>
<td>734.95</td>
</tr>
<tr>
<td>84-4-4-0-(\frac{1}{3})</td>
<td>7</td>
<td>11.54</td>
<td>49.24</td>
<td>1407.47</td>
</tr>
<tr>
<td>96-4-4-0-(\frac{1}{3})</td>
<td>7</td>
<td>44.20</td>
<td>159.80</td>
<td>4755.08</td>
</tr>
<tr>
<td>120-4-4-0-(\frac{1}{3})</td>
<td>7</td>
<td>112.44</td>
<td>299.47</td>
<td>10667.67</td>
</tr>
<tr>
<td>144-4-4-0-(\frac{1}{3})</td>
<td>7</td>
<td>382.40</td>
<td>819.78</td>
<td>32553.82</td>
</tr>
<tr>
<td>168-4-4-0-(\frac{1}{3})</td>
<td>7</td>
<td>466.44</td>
<td>762.56</td>
<td>36736.72</td>
</tr>
<tr>
<td>192-4-4-0-(\frac{1}{3})</td>
<td>7</td>
<td>1188.18</td>
<td>1660.49</td>
<td>86217.25</td>
</tr>
<tr>
<td>216-4-4-0-(\frac{1}{3})</td>
<td>7</td>
<td>985.55</td>
<td>1417.83</td>
<td>68510.43</td>
</tr>
</tbody>
</table>

Table 5: \((r = 4, q = 4)\)-instances for \(36 \leq p \leq 216\) and \(f = \frac{1}{3}\)
References


Gribkovskaia, I., Laporte, G., & Shyshou, A. (2008). The single vehicle routing prob-


