Scheduling Data Transfers with Priorities for Space Missions

Julien Rouzot[1,2], Christian Artigues[1], Clément Carbonnel[3], Philippe Garnier[2], Emmanuel Hebrard[1], Pierre Lopez[1], and Bertrand Simon[4]

[1] Univ Toulouse, CNRS, LAAS, Toulouse, France
surname.name@laas.fr
[2] Univ Toulouse, CNRS, CNES, OMP, IRAP, Toulouse, France
surname.name@irap.omp.eu
[3] University of Montpellier, CNRS, LIRMM, Montpellier, France
clement.carbonnel@lirmm.fr
[4] UGA, CNRS, INRIA, Grenoble INP, LIG, Grenoble, France
bertrand.simon@cnrs.fr

Abstract. In space missions, scientific data collected by various instruments must be stored onboard before being downlinked to Earth during designated communication windows. For many long range missions, the available bandwidth is shared according to a priority assigned to each memory buffer during such downlink window. The overlapping Memory Dumping Problem (oMDP) consists in finding the priority assignment that minimizes the highest memory peak. This problem has been shown to be weakly NP-hard and has so far only been addressed with heuristic methods. In this paper, we complete the complexity analysis by proving that the problem is strongly NP-hard in the general case, and we propose the first exact method to solve the oMDP. We present a constraint programming approach combining new global constraints and a heuristic branching strategy, and show that our method is competitive with state-of-the-art heuristics while being more generic and able to produce optimality proofs on small instances.

Keywords: Constraint Programming · Scheduling · Data Transfers.

1 Introduction

In deep space missions, scientific instruments operate semi-independently, generating valuable scientific data that must be transmitted to Earth through tightly constrained downlink windows. Each instrument typically writes its data in a dedicated onboard buffer, which introduces a key challenge: coordinating data transfers in such a way that buffer overflows are avoided. This paper addresses the scheduling of these data transfers with the goal to avoid data loss.

In the case of the Rosetta mission [3], the data produced by the different instruments is stored on distinct buffers that are dumped only during *downlink windows*, where the spacecraft is visible from ground stations of the Deep Space

Network (DSN) [8]. Data transfers must be scheduled so that no buffer exceeds its capacity during the mission, which could result in the loss of critical data. More precisely, since actual data production and transfer rates are subject to uncertainties, the objective is to minimize the highest memory level across all buffers over the entire time horizon. This objective is motivated by the need to enhance robustness in the case of operational uncertainties. By minimizing the highest memory level across all instrument buffers, the scheduling plan gains greater tolerance to unexpected fluctuations on the actual data production and transfer rates. A lower memory peak effectively creates a safety margin within each buffer, reducing the risk of overflow. During a downlink window, bandwidth is shared via a priority-based Round-Robin scheme. Buffers with the best priority evenly share the available bandwidth; any unused capacity is then allocated to lower priorities, until all buffers are empty or no bandwidth remains.

We propose an exact method based on constraint programming to solve the problem of assigning priorities that minimize the highest memory peak, under the assumption that the fill rates of each buffer are known along the mission, as they are determined by the science observations plan [3]. This problem is known as the overlapping Memory Dumping Problem [12]. The oMDP was previously shown to be at least weakly NP-hard in [7] when both the number of memory buffers and the number of downlink windows are unbounded. We refine the complexity analysis by proving that the problem is strongly NP-hard in the general case. We also introduce a CP approach, the first exact method for the oMDP, previously tackled only with heuristics in [12] and [7].

We first formally introduce the oMDP in Section 2 and present the related works in Section 3. We provide an extensive complexity analysis in Section 4. We present our CP model with new global constraints and a heuristic branching strategy to address the oMDP in Section 5 and we finally highlight the performance of our approaches in Section 6, before concluding in Section 7.

2 The overlapping Memory Dumping Problem

We consider the oMDP, illustrated in Figure 1, where we have a set of buffers $\mathcal{B} = \{1, \ldots, n\}$. For each buffer $i \in \mathcal{B}$, the memory level is $U_i(t) : \mathbb{R}^+ \mapsto \mathbb{R}^+$ with t between 0 and h, where h is the time horizon. Each buffer has a finite capacity C_i, and the memory level must remain below this limit at all times. Data can be transmitted only during a set of downlink windows in $\mathcal{W} = \{1, \ldots, m\}$, that start at w_j^{start} and end at w_j^{end}. The dump rate—i.e. bandwidth—available for each window j is denoted δ_j. The memory peak $r_{i,j}$ for window j is the ratio of the highest memory level of buffer i during the window j and its capacity C_i. The objective is to minimize the highest peak: $\min\ rmax = \max_{i \in \mathcal{B}, j \in \mathcal{W}} (r_{i,j})$.

We assume that each buffer i has a piecewise constant fill rate function $f_i : \mathbb{R}^+ \mapsto \mathbb{R}^+$ determined by the science observation plan, where each nonzero segment corresponds to a distinct observation. Transfers are controlled through priorities, updated only at the beginning of each downlink window. For each buffer i, priorities $p_{i,j}$ are within the domain $[1, \ldots, n]$, where 1 is the *best* and

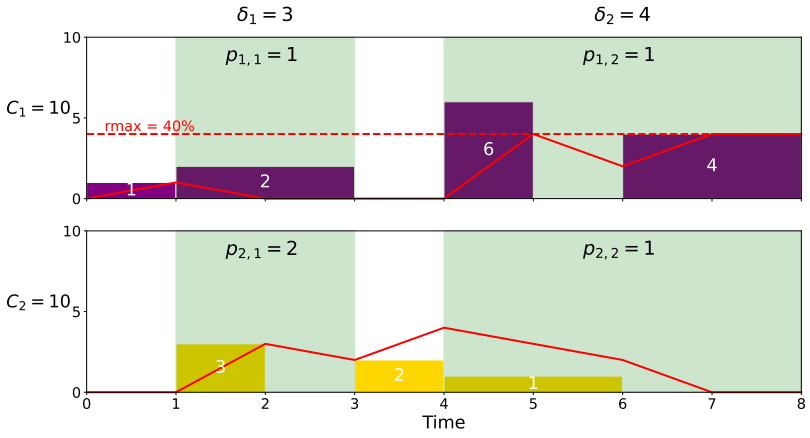

Fig. 1: *Evolution of the buffers memory level $U_i(t)$ (red), according to their fill rates (violet and yellow), the dump rates (green), and a priority assignment.*

n the *worst*. The transfer policy follows a simple priority-based Round-Robin scheme for sending atomic data packets. More precisely, whenever bandwidth is available, the system transfers a single packet from one of the non-empty buffers with the highest priority. If all such buffers are empty, it proceeds to the next priority pool, continuing this process until a non-empty buffer is found or all priority pools have been checked. Since buffers can share the same priority, ties are broken using a circular queue, meaning that among buffers with equal priority, preference is given to the one that transferred data least recently.

Although the downlink procedure is discrete, it is shown in [14] that the transfer rate functions resulting from a priority policy p can be tightly approximated as piecewise linear functions, as the size of the data block are extremely small in front of the fill rates and the dump rates. In [7], a procedure based on the work presented in [14] is proposed to compute the transfer rates and the memory level functions resulting from the fill rate functions and a priority assignment, assuming that buffers are filled and dumped continuously.

Definition 1 (Transfer Rates [7]). *Given a downlink window j and a priority assignment p, the transfer rate functions $g_i^p(t)$ can be computed recursively by considering the buffers from the best priority rank to the worst. Let $\Omega = \{i_1, \ldots, i_k\}$ be the set of buffers with best priority, ordered by ascending memory level, breaking ties with ascending fill rate at time t. The available bandwidth Δ_j for buffers in Ω is initially set to δ_j. Then, the transfer rate $g_{i_1}^p(t)$ of buffer i_1 at time t is $\Delta_j/|\Omega|$ if its memory is not empty (an equal share of the bandwidth) and $\min(f_{i_1}(t), \Delta_j/|\Omega|)$ otherwise (as fast as it is filled, but no more than an equal share). Then, for $\ell \in [2, k]$, the transfer rate $g_{i_\ell}^p(t)$ is computed similarly, after setting Δ_j to $\Delta_j - g_{i_{\ell-1}}^p(t)$ and Ω to $\Omega \setminus \{i_{\ell-1}\}$. Finally,*

if $\Delta_j > 0$, transfer rates are similarly computed for lower-priority buffers, rank by rank, until bandwidth is exhausted or all buffers are processed.

Definition 2 (Memory Level). *Given the fill and transfer rate functions f_i and g_i^p, and the initial memory level for each buffer $U_i(0)$, the memory level $U_i^p(t)$ is the integral over time of the data production minus the data transfer:*

$$U_i^p(t) = U_i(0) + \int_0^t \left(f_i(t') - g_i^p(t')\right)\, dt'$$

Definition 3 (oMDP). *Given the buffer set $\mathcal{B}$, the downlink windows $\mathcal{W}$, the fill rate functions f_i, and the initial memory level $U_i(0)$ for each buffer i, what is the priority assignment p that minimizes:*

$$rmax = \max_{i \in \mathcal{B}} \left(\max_{t \in [0,h]} \left(U_i^p(t)\right) / C_i \right)$$

3 Related Work

In [12], the authors introduced the oMDP and presented the fast DownlinkCount heuristic to solve the problem. This first heuristic approach for oMDP is based on the following assumption: buffers that overflow first if no data dump is performed should be given a better priority. More precisely, for a given downlink window and initial memory levels at the start of this window, DownlinkCount *counts* the number of windows before an overflow occurs, when $\delta_j = 0,\ j \in \mathcal{W}$. Then, the priorities are distributed to the buffers, starting with the best priority for the buffer with the lowest downlink count. In case of ties, the same priority is assigned. This process is repeated until all priorities are assigned. This heuristic has a very low computing cost as we only need to sum the fill rate events until the overflow (or the end of the instance) is reached, for each buffer, for a given downlink window. This simple heuristic yields surprisingly good solutions on its own and was used by Rabideau et al. in their IterativeLeveling method, consisting of iteratively reducing the overflow limit. This method was used for the Rosetta science operations planning tool, ASPEN [2].

Even though the Rosetta mission is over, efficient algorithms for scheduling data transfers via priority assignments remain valuable for future missions. The oMDP has been further studied in [7], where the authors introduced a fast and exact Simulation procedure to compute memory level functions based on a given priority assignment. They also proposed a polynomial-time algorithm, SingleWindow, to determine the optimal priority assignment for a single downlink window, as well as an LNS-based heuristic (RepairDescent) for solving the multiple-window problem. In this paper, we build on this work, so let us present the Simulation algorithm. Algorithm 1 computes the function U_i as defined in Definition 2, for a given downlink window j.

This Simulation procedure starts by setting the current and the next time point to the start time of the downlink window (lines 1). The function **MakeEventList** creates a list of events, including the start or the end of each fill rate event, ordered by time (line 2). Each event e is characterized by a *buffer*

Algorithm 1 SIMULATION: Computes the memory level functions.

Require: A window j, the dump rate δ_j, the start and end dates w_j^{start} and w_j^{end}, $\mathcal{B}$, $f_i(t)_{t\in[w_j^{start},w_j^{end}]}$, the priorities p_j, the initial memory level $U_i(w_j^{start})$.

1: $t_{now} \leftarrow t_{next} \leftarrow w_j^{start}$ ▷ Current and next breakpoint
2: $events \leftarrow$ **MakeEventList** $\left(f_i(t)_{t\in[w_j^{start},w_j^{end}]}\right)$ ▷ Ordered by ascending time
3: $k \leftarrow 0$ ▷ Current event index
4: **while** $t_{now} < w_j^{end}$ **do** ▷ Loop through all events
5: $t_{now} \leftarrow t_{next}$ ▷ Update current breakpoint
6: **while** $t_{now} = t_{next}$ **do**
7: $i \leftarrow events[k].buffer$
8: $f_i(t_{now}) \leftarrow events$[k].*value* ▷ Update current fill rates
9: $k \leftarrow k + 1$ ▷ Next event
10: $t_{next} \leftarrow events[k].time$ ▷ Next breakpoint
11: $g^{p_j}(t_{now}) \leftarrow$ **ComputeTransferRates**$(U(t_{now}), f(t_{now}), \delta_j, p_j)$
12: **for** $i \in B$ **do**
13: $t_{empty} \leftarrow$ **GetEmptyTime**$(t_{now}, U_i(t_{now}), f_i(t_{now}), g^{p_j}(t_{now}))$
14: **if** $t_{empty} < t_{next}$ **then**
15: $t_{next} \leftarrow t_{empty}$ ▷ Insert new breakpoint if buffer i becomes empty
16: **JumpTo**$(t_{next}, U(t_{now}), f(t_{now}), g^{p_j}(t_{now}))$
17: **return** $U(t)_{t\in[w_j^{start},w_j^{end}]}$

index (e.buffer), a *time* (e.time) and a *value* (e.value). They correspond to the *breakpoints* that must be evaluated chronologically to update the memory levels. While events correspond to the current breakpoint, the fill rate for the corresponding buffer is updated (line 8). Then, current transfer rates resulting from the memory levels, fill rates, dump rate, and priority assignment p at $t = t_{now}$ are computed in function **ComputeTransferRates** (line 11) as explained in Definition 1. For a given downlink window, these transfer rates do not change unless an observation starts or ends (i.e. a fill rate changes) or a buffer becomes empty (as it will redistribute its share of the bandwidth). While fill rate breakpoints are known, breakpoints due to buffers becoming empty must be computed dynamically. **GetEmptyTime** returns $t_{now} + U_i(t_{now}) / \left(g_i^p(t_{now}) - f_i(t_{now})\right)$ if $g_i^p(t_{now}) > f_i(t_{now})$, and returns $+\infty$ otherwise. In Algorithm 1, a new event is inserted if a buffer becomes empty before t_{next}. The memory level is then updated at the next time point inside function **JumpTo** that sets $U_i(t_{next})$ to $U_i(t_{now}) + (t_{next} - t_{now}) \times (f_i(t_{now}) - g_i^p(t_{now}))$ for each buffer i. This process is repeated until the end of the current downlink window is reached. With k the number of fill rate events, n the number of buffers, this algorithm simulates a priority assignment in $O(k \log k + n^2 k \log n)$-time.

4 Complexity

D-oMDP (the decision variant of oMDP) was shown to be weakly NP-hard in [7], when both the number of memory buffers and the number of downlink windows

are not bounded by a constant, and polynomial for a single downlink window. In this section, we show that the problem is strongly NP-hard.

Theorem 1. *D-oMDP is strongly NP-hard.*

Proof. We use a reduction from the NP-complete problem 3-Partition [4], which takes as input a set of $3n$ numbers $A = \{a_1, \ldots, a_{3n}\}$ with $\sum_{i=1}^{3n} a_i = nT$ with $\frac{T}{4} < a_i < \frac{T}{2}$ and asks whether there exists a partition of A into n triplets, each summing to T. The reduction uses $4n$ buffers and $3n$ downlink windows. Window i has duration a_i and dump rate n. For $1 \leq i \leq 3n$, buffer i has capacity a_i, starts at $U_i(0) = 0$ and has a fill rate of 2 during the downlink window i. All remaining buffers have capacity $2T$, start with a memory level of T and have a fill rate of 1 during each downlink window.

Assume we have a solution to the reduced instance. Note that the buffers capacities sum to $3nT$, the maximal data that can be dumped is n^2T and the total data that is produced is $2nT + nT + n^2T$, so all buffers must be full at the end, from which we have the following Lemma:

Lemma 1. *Every buffer must be full at the end of the plan.*

In window i, buffer i must dump exactly a_i data. Otherwise, if it dumps less than a_i, it would exceed its capacity at the end of window i and if it dumps more it would contradict Lemma 1. This means it must be given the best priority tied with exactly $n-1$ non-empty buffers. These buffers must be among buffers $3n+1$ to $4n$, because non-empty buffers from 1 to $3n$ would never re-fill and hence that would contradict Lemma 1. In other words, at window j, the usage of exactly one buffer among $3n+1$ to $4n$ increases by exactly a_j. Therefore, the reduction is satisfiable if and only if there exists a partition of A into n sets, each summing to T. These sets must contain exactly 3 elements as $\frac{T}{4} < a_i < \frac{T}{2}$.

5 CP Model

We propose to address the oMDP with constraint programming. As presented in Section 3, we must calculate the transfer rates for each buffer at each breakpoint (where a fill rate or the dump rate changes) to compute the memory level function according to a priority assignment. We also have to dynamically add new breakpoints when buffers become empty as the bandwidth is redistributed to the other buffers in this case, and thus the transfer rates change. Therefore, modeling the simulation process with classical constraints is extremely challenging and it is very likely that time would have to be discretized in order to obtain an implementable model (by approximating the problem in this way and creating a high number of memory variables). Instead, we encapsulate the simulation in a global constraint, and we only track the memory levels at the beginning of each window, and the maximum memory levels during each window.

The memory variables $mem_{i,j}$ represent the memory level of buffer i at the beginning of a window j. Each window is composed of the non-visibility part

where the dump rate is zero and the actual downlink window where the bandwidth is δ_j. Thus, the memory variables serve as indicators of the memory level at both the start of window j and the end of window $j-1$ when $j > 1$. For each buffer, we keep track of the memory peaks $r_{i,j}$ during window j with the peak variables. As the constraint programming solver we use does not handle continuous variables, we have to round memory and peak values. In the case of memory variables, we round to the upper integer. In the case of peaks, since they are a ratio between 0 and 1, we use a parameter r^{ub} to control the rounding, so the actual peak is $r_{i,j} \,/\, r^{ub}$. Therefore, increasing r^{ub} improves the precision.

In the oMDP, our only decision variables are the priority variables $p_{i,j}$ for each buffer and window. The objective and constraints are as follows:

$$\min \quad rmax = \left[\max_{i \in \mathcal{B},\, j \in \mathcal{W}} (r_{i,j}) \right] \tag{1}$$

s.t.

$$mem_{i,0} = U_i(0) \qquad i \in \mathcal{B} \tag{2}$$

$$\textsc{PriorityTransfer}\,(ins_j, mem_j, mem_{j+1}, r_j, p_j) \qquad j \in \mathcal{W} \tag{3}$$

$$\textsc{DenseRanking}\,(p_j) \qquad j \in \mathcal{W} \tag{4}$$

$$\textsc{PrioritySymmetry}\,(ins_j, mem_j, p_j) \qquad j \in \mathcal{W} \tag{5}$$

In our model, the instance is denoted ins and contains all the information about its parameters (observation events, downlink windows, etc.). The dropping of index i (for instance, mem_j or p_j) means that all buffers in window j are taken into account. The instance is split into m single window instances during a preprocessing step, and ins_j is a tuple gathering the parameters of the j-th window. More precisely, $ins_j = \{j, \delta_j, w_j^{start}, w_j^{end}, \mathcal{B}, f(t)_{t \in [w_j^{start}, w_j^{end}]}\}$. The objective is to minimize the highest memory peak (1). We initialize the first memory variables to the initial memory levels (2). We propose a new global constraint PriorityTransfer that maintains and filters the domains of the memory, peak and priority variables (3). We also introduce two new global constraints to eliminate symmetric solutions, one based on the rules of a dense ranking to order the priorities in a unique way (4), and one taking advantage of the oMDP structure (5).

5.1 PriorityTransfer Constraint

PriorityTransfer constraint is responsible for maintaining consistency between the domains of the memory variables $mem_{i,j}$, $mem_{i,j+1}$, the peak variables $r_{i,j}$, and the priority variables $p_{i,j}$.

Definition 4 (PriorityTransfer).
Let $U(t)_{t \in [w_j^{start}, w_j^{end}]} = \textsc{Simulation}\,(ins_j, p_j, mem_j)$*, then the constraint* $\textsc{PriorityTransfer}\,(ins_j, mem_j, mem_{j+1}, r_j, p_j)$ *is satisfied if and only if:* $mem_{i,j+1} = U_i\left(w_j^{end}\right) \wedge r_{i,j} = \max_{t \in \left[w_j^{start}, w_j^{end}\right]} (U_i(t)) \leq rmax,\ \forall i \in \mathcal{B}$

PriorityTransfer consists of three propagators. Propagator Simulation Check handles the simplest case, when $mem_{i,j}$ and $p_{i,j}$ are fixed, as we can compute the values of $mem_{i,j+1}$ and $r_{i,j}$ with the Simulation algorithm. We know $U(t)$ for $t \in \left[w_j^{start}, w_j^{end}\right]$, by running Simulation (ins_j, p_j, mem_j). Thus, the memory level for the next window is $mem_{i,j+1} = U_i(w_j^{end})$ and $r_{i,j} = \max_{t \in \left[w_j^{start}, w_j^{end}\right]}(U_i(t))$, for each buffer i. The worst-case time complexity of this propagator is the same as Simulation, $O(k \log k + n^2 k \log n)$ with k the number of fill rate events. This propagator is sufficient for the constraint correctness but performs only minimal filtering on the domains of the variables, as it only updates the memory for the next window when all variables are fixed for the current one. To improve filtering, we developed another propagator, SimulationLowerBound, that also uses Simulation. This propagator considers a *best-case scenario* for buffer i, to compute a lower bound on $mem_{i,j+1}$ and $r_{i,j}$. After each Simulation run, we update the lower bounds of $mem_{i,j+1}$ and $r_{i,j}$. If $\min(r_{k,j}) > rmax$, the solver fails and performs backtracking. The time complexity of this propagator is $O(n(k \log k + n^2 k \log n))$, as we need to run Simulation for each buffer.

Theorem 2 (Best-case scenario). *From the perspective of a buffer i, given the domains of $mem_{i',j}$ and $p_{i',j}$ for all buffers $i' \in \mathcal{B}$ and a single window j, the minimal memory level peak and memory level at the end of the window will be reached if: memory levels at the start of window j are minimal:* $\min(mem_{i',j}), \forall i' \in \mathcal{B}$*; Priority of buffer i during window j is the best:* $\min(p_{i,j})$*; Priorities of other buffers during window j are the worst:* $\max(p_{i',j}), \forall i' \neq i \in \mathcal{B}$.

Proof. For a given share of the bandwidth allocated to buffer i, increasing the initial memory level $mem_{i,j}$ can only increase its memory level during the window. Increasing the initial memory level of any other buffer $i' \neq i$ can never increase the bandwidth allocated to buffer i, but it can decrease it. In the same manner, giving buffer i a higher (i.e. worse) priority or giving another buffer a lower (i.e. better) priority can only lead to decrease i in the buffers relative priority order and thus decreasing the bandwidth allocated to buffer i. Therefore, the minimal peak and end memory level will be reached if the three conditions of Theorem 2 stand.

Although oMDP is NP-hard, it has been demonstrated that an optimal solution can be computed in polynomial time if we consider a single downlink window [7]. The algorithm to decide whether there exists a priority assignment achieving a given peak usage is called SingleWindow. We can extend this algorithm to take into account the priority domains to compute this solution and filter some inconsistent values.

Lemma 2. *Given a window j, a buffer i and a value $v \in D(p_{i,j})$, if buffer i exceeds the current rmax when running Simulation with: $p_{i,j} = v$ and $\forall\, 'i \in \mathcal{B}, i' \neq i$, $p_{i',j} = \max(p_{i',j})$; $\forall\ i' \in \mathcal{B}$; $U_{i'}(w_j^{start}) = \min(mem_{i',j})$, then the assignment $p_{i,j} = v$ is inconsistent.*

Proof. Giving a better priority to any buffer $i' \neq i$ can never result in increasing the bandwidth share allocated to buffer i. In the same way, increasing the memory level of any buffer can only increase the peak usage of buffer i. Therefore, if the scenario in Lemma 2 violates the constraint $r_{i,j} \leq rmax$, the only way to increase the bandwidth allocated to buffer i, and thus reducing its peak, is to assign it a strictly better priority than v.

This leads to the constraint's third propagator, SingleWindow. In this propagator, memory levels are initialized to $\min(mem_{i,j})$ and the Simulation algorithm is run successively with the priority assignment p', that corresponds to the priority variables set to their current upper bound, and the memory levels set to their lower bounds. By Lemma 2, the current priority value of each overflowing buffer is inconsistent and can be removed. We repeat this process until no buffer overflows, or one priority domain is empty. In Rouzot's thesis [13], we provide the pseudo code for this propagator.

5.2 Symmetry Breaking

Dense Ranking A solution of oMDP is a complete assignment of the priority variables for each downlink window. The priorities correspond to a *ranking* between the different buffers. The use of integer variables to represent the priority group for all buffers is straightforward, but the downside is the existence of equivalent solutions. For instance, for 3 buffers, $p_{1,j} = 1, p_{2,j} = 1, p_{3,j} = 2$ is equivalent to $p_{1,j} = 1, p_{2,j} = 1, p_{3,j} = 3$ (i.e. it represents the same ranking). To break these symmetries, we force the variables to respect the rules of a *dense ranking*: at least one element is equal to 1 (i.e. best rank 1 must be assigned); for any $k \in \{2, \ldots, n\}$, if k is assigned, then rank $k-1$ is also assigned (i.e. consecutive ranks must be assigned). To enforce these rules, we introduce a new global constraint called DenseRanking, defined as follows:

Definition 5 (DenseRanking).
Let $\mathcal{X} = \{x_1, \ldots, x_n\}$, then the constraint DenseRanking$(\mathcal{X})$ is satisfied if and only if: Min$(\mathcal{X}) = 1 \wedge \forall i \in \{1, \ldots, n\}, \forall k \in \{2, \ldots, k\}$, $x_i = k \implies \exists i' \in \{1, \ldots, n\}$ such that $x_{i'} = k - 1$

Our propagation algorithm for the DenseRanking constraint enforces only bound consistency and operates on a set of n variables whose domains are subsets of $\{1, \ldots, n\}$ with consecutive integer values. The pseudo-code for finding a bound support for the DenseRanking constraint is presented in Algorithm 2, and is used in Algorithm 3, which is the propagation algorithm.

In DenseRankingBoundSupport, variables are sorted in increasing order of their lower bounds (line 1). Then, we try to build a valid dense ranking—i.e. a support for the current domains of the variables. The variable k tracks the current rank being assigned in the construction of the support (line 3). A binary heap is initialized—empty at first—to maintain candidate variables ordered by ascending upper bounds. For each rank k, we add to the heap all variables whose domain includes k (line 7). At least one variable must be selected to cover each

Algorithm 2 DenseRankingBoundSupport.

Require: $\mathcal{X} = \{x_1, \ldots, x_n\}$
1: $x_1, \ldots, x_n \leftarrow$ **SortLB**$(\mathcal{X})$ ▷ Sort the variables by ascending lower bounds
2: $heap \leftarrow$ **InitBinaryHeap**$(ordering = asc\_ub)$
3: $k \leftarrow 1$ ▷ Current rank
4: $i \leftarrow 1$
5: **while** $i \leq n$ **do**
6: **while** $\min(x_i) \leq k$ **do**
7: $heap$.**Insert**(x_i) ▷ Insert variables with k in their domain
8: $i \leftarrow i + 1$
9: **if** $heap$.**IsEmpty**() **then** ▷ No variable to cover the current rank
10: **return** `false`
11: **else**
12: $heap$.**RemoveRoot**() ▷ Variable with smallest UB takes rank k
13: **while** $\max(heap.\textbf{Root}()) = k$ **do**
14: $heap$.**RemoveRoot**() ▷ Variables x_i with $\max(x_i) = k$ takes rank k
15: $k \leftarrow k + 1$
16: **return** `true`

Algorithm 3 DenseRankingBC

Require: $\mathcal{X} = \{x_1, \ldots, x_n\}$
1: **for** $x \in \mathcal{X}$ **do**
2: **AdjustLB**$(x, \mathcal{X})$
3: **AdjustUB**$(x, \mathcal{X})$
4: **if** $\min(x) > \max(x)$ **then**
5: **return** `Fail`

rank. Since we assume domains with consecutive values bounded by the lower and upper bounds of the variable, the variable with the smallest upper bound must be selected (line 12). Indeed, variables with larger upper bounds can still cover future ranks, while those with smaller upper bounds may not be able to do so, risking discontinuities in the dense ranking. We prove that this strategy is optimal in Lemma 3. After assigning a variable to rank k, we remove from the heap all variables whose upper bound is equal to k (line 14), as they cannot be used to cover any higher ranks. If no variable in the heap can be assigned to the current rank and some variables are still to be assigned (line 9), it means that the current domains are inconsistent. This process is repeated with rank $k + 1$ until failure or until all variables have been assigned to a rank. In the latter case, a support for the current domains has been successfully found. The time complexity of Algorithm 2 is $O(n \log n)$: sorting the variables initially takes $O(n \log n)$, and in the main loop, at most n insertions and n root removals are performed sequentially, each in $O(\log n)$.

Lemma 3. *Given the domains of variables $\mathcal{X}$, Algorithm 2 always finds a bound-consistent support for* DenseRanking*, if such a support exists.*

Proof. When building the support for $\mathcal{X}$ in DenseRankingBoundSupport, we select at least one variable x that contains k, with the smallest upper bound to cover each rank k. Let us choose any other variable x' that also contains k instead of x. If choosing x' leads to a valid support, it is possible to switch the values of x and x' as $[k, \max(x)]$ is included in $D(x')$ and x can take value k. Thus, choosing the variable with the smallest upper bound at each step is the best strategy. It follows that if no variable can take current rank k, there is no assignment of $\mathcal{X}$ with the current domains such as the DenseRanking constraint is respected.

In Algorithm 3, the bounds of variable domains are updated by eliminating inconsistent lower and upper bounds values with Algorithm 2. As we are asserting bound consistency only, this can be done in $O(n \log n)$-time by performing a dichotomic search on both bounds with procedures AdjustLB and AdjustUB. In AdjustLB, the domain of variable x is reduced to a subset of $D(x)$ with consecutive values starting from $\min(x)$. The procedure searches for the maximum value of m such that no support can be found for $\mathcal{X}$ with $D(x) = [\min(x), \ldots, m]$. When such m is found, the lower bound of x is set to $m+1$, as all values in $[\min(x), \ldots, m]$ are not bound consistent for DenseRanking. Similarly, in AdjustUB, the procedure search for the minimum value of m such that no support exists for $\mathcal{X}$ with $D(x) = [m, \ldots, \max(x)]$, and the upper bound of the variable is set to $m - 1$. This is repeated for all variables $x \in \mathcal{X}$, so the time complexity of our propagator DenseRankingBC is $O(n^2 \log^2 n)$, as DenseRankingBoundSupport is called $O(n \log n)$ times.

Note that DenseRanking constraint can be enforced with the global constraint AtLeastNValues$(\mathcal{X}, n)$ [1], based on SoftAllDiff [11]. This constraint forces the set of variables $\mathcal{X}$ to take at least n distinct values. By setting n to Max$(\mathcal{X})$, we ensure that all values in $\{1, \ldots, \text{Max}(\mathcal{X})\}$ will be assigned to at least one variable. However, this combination of constraints is sometimes not as strong as DenseRanking. Let us demonstrate it with a simple example: Let $\mathcal{X} = \{x_1, x_2, x_3\}$ with $D(x_1) = \{1, 2, 3\}$, $D(x_2) = \{2, 3\}$, $D(x_3) = \{2, 3\}$. As rank 1 is mandatory for respecting the rules of a dense ranking, x_1 must take the value 1 and thus $x_1 = 2$ and $x_1 = 3$ are not consistent and those values should be discarded. If we run the arc consistency for AtLeastNValues$(\mathcal{X}, N)$ and $N = \text{Max}(\mathcal{X})$, $x_1 = 2$ and $x_2 = 3$ will not be filtered even though they are inconsistent for the dense ranking. In the other hand, as we are not providing an algorithm for arc consistency on DenseRanking, the decomposition may sometimes filter values missed by DenseRankingBC.

Priority Symmetry Symmetry also arises when a group of buffers, having priority over the bandwidth, holds enough data to fully utilize the available dump rate during the downlink window. In this case, the worst-priority buffers receive no bandwidth, regardless of their relative priority order. Let v be the worst priority value among a subset of buffers S with fixed priorities for a given downlink window j. Let us assume that the buffers in S are never all empty during the downlink window j, thus no share of the overall bandwidth will be

redistributed to buffers with a priority strictly greater (i.e. worse) than v. In that case, any priority value v' such as $v' > v$ (i.e., a worse priority) is equivalent to the priority value $v + 1$. This observation being made, we can infer a filtering procedure for the priority variables, that is embedded in constraint PrioritySymmetry. Every time the domain of a priority variable $p_{i,j}$ changes, we run: Simulation(ins_j, p_j, mem_j) with $p_j = \{\max(p_{i,j}) \mid \forall\, i \in \mathcal{B}\}$, $mem_j = \{\min(mem_{i,j}) \mid \forall\, i \in \mathcal{B}\}$. During the simulation, we keep track of whether each buffer has been allocated bandwidth given the current priority assignment. Let v be the worst priority value among the buffers that received bandwidth. Since any priority assignment strictly greater than v would result in the buffer not receiving bandwidth, we can remove all values strictly greater than $v+1$ from the priority domains for the current downlink window j, as they represent equivalent priority assignments. This propagator is as costly as Simulation that runs in $O(k \log k + n^2 k \log n)$-time, with k the number of observation events (see Section 3).

5.3 Search

In NP-hard problems such as the oMDP, the way we explore the search tree may significantly impact the quality of the solution, given a limited time. As PriorityTransfer strongly relies on the current $rmax$ upper bound to trigger backtracking with SimulationLowerBound and SingleWindow propagators, we must find good solutions quickly. To that end, we developed a search method based on the DownlinkCount heuristic for variable and value selection.

In our search, decisions are made starting from the earliest downlink windows, proceeding to the next only when all priority variables of the current window are fixed. Within a single window, we use DownlinkCount (see Section 3) for both variable and value selection. Instead of setting the buffer capacity as the overflow limit, we introduce a random limit based on the current $rmax$ upper bound. More precisely, the overflow limit for DownlinkCount is drawn uniformly between $0.5 \times \max(rmax)$ and $\max(rmax)$. Adding randomness to the decision process is helpful, so the solver explores new branches of the search tree after each restart, thereby accelerating the resolution. For variable selection, we begin with the variable that has the best priority in the solution produced by DownlinkCount. Choosing the buffers with the best priority first is more likely to trigger propagation through the DenseRanking constraint and the PrioritySymmetry propagator. For value selection, we follow the priority order given by DownlinkCount if the value is present in the domain of the variable. If the priority exceeds the upper bound of the variable, we assign the upper bound; if the priority is lower than the lower bound, we assign the lower bound.

6 Experimental Results

For long-range missions by the ESA (European Space Agency), science observation planning is divided into four steps [5,10]. First, the Long Term Plans (LTP)

are established well in advance by the Science Ground Segment in collaboration with the Science Working Team. These high-level plans each cover a few months of the mission. They are then refined into Medium Term Plans (MTP) of a month, Short Term Plans (STP) of a week, and finally Very Short Term Plans (VSTP), which is the smallest and final planning level, spanning half a week. In the case of Rosetta, each new smaller plan corresponds to a new instance to solve, requiring orchestration of data transfers to minimize buffer overflow risks. Proving optimality for non-trivial MTP and LTP instances is extremely challenging, and at these stages, good solutions are often sufficient. However, having an optimality guarantee for STP and VSTP instances is far more valuable.

For their experiments, [7] used 4 real instances, that are MTP of the Rosetta mission with 16 buffers, 64 to 94 downlink windows, and around 10,000 data production events each (see [3] for Rosetta observation planning). They also generated random instances based on these real scenarios. However, these instances often feature long non-visibility periods, during which high memory peaks can occur, making trivial lower bound[5] on $rmax$ easy to match. To address this issue and thus produce instances corresponding to potentially more difficult contexts, we generated new instances from scratch by introducing random events, shortening non-visibility periods, and ensuring all buffers produce substantial data.

Our model is implemented using the OR-Tools original CP solver [9]. For all experiments, we approximate the peaks to a tenth of a percent ($r^{ub} = 10^3$) and always round up peak values. All experiments are run on a Xeon E5-2695 v3 @ 2.30 GHz CPU with 10 GB of RAM, with a time limit of one hour per experiment. Our source code and instances are available in our Git repository[6].

First, we analyze the efficiency of our global constraints in reducing the search space on small randomly generated instances. We then highlight the impact of our search heuristic in quickly finding good solutions for real Rosetta instances. Finally, we present a broader statistical analysis on both Rosetta instances and randomly generated instances of various sizes, comparing our approach against the state-of-the-art: IterativeLeveling [12] and RepairDescent [7].

Table 1 presents the average search effort required to prove optimality on 20 small instances, each with 4 buffers and 4 downlink windows. We evaluate the impact of disabling constraints or propagators by comparing performance across different model configurations. Specifically, we report the average solution time, the number of branches explored by the solver, and the number of optimal solutions. The first row corresponds to the baseline, where only the SimulationCheck propagator is enabled. In the subsequent rows, a check mark (✓) indicates which additional propagators or constraints are active.

We observe a clear improvement when using our global constraints compared to the baseline, for the resolution time (560 times faster), the number of branches explored (2400 times fewer) and the number of optimal solutions (20 vs. 16). We remark the importance of symmetry breaking with DenseRanking

[5] A way to find such lower bound is FullTransfer [7], an algorithm that allocates all bandwidth to each buffer during downlink windows.

[6] `https://gitlab.laas.fr/roc/julien-rouzot/transfer-scheduling-csp`

Constraints				Resolution time	Number of branches	Optimal solutions
DR	PS	SW	SLB			
				14 min	168.2 M	16
	✓	✓	✓	3.5 min	10.2 M	19
✓		✓	✓	53.3 s	2.6 M	20
✓	✓		✓	1.7 s	0.14 M	20
✓	✓	✓		1.6 s	0.11 M	20
✓	✓	✓	✓	1.5 s	0.07 M	20

Table 1: *Mean resolution time, number of branches explored, and number of optimal solutions for 20 small instances for different combinations of global constraints and propagators enabled. DR: DenseRanking, PS: PrioritySymmetry, SW: SingleWindow, SLB: SimulationLowerBound.*

and PrioritySymmetry constraints, as the resolution time and the number of branches raise significantly when they are disabled. On the other hand, SimulationLowerBound and SingleWindow propagators seem to have the most limited impact on the resolution time, but help pruning the search space.

We also compare our CP method against IterativeLeveling heuristic [12] and RepairDescent heuristic [7] on 480 generated scenarios in Figure 2.

The synthetic instances include 240 small plans (4 to 16 buffers, 4 to 16 downlinks) and 240 larger plans (8 to 24 buffers, 20 to 80 downlinks). We present the proportion of optimal solutions for 240 small instances with varying numbers of buffers and windows. The darker zone at the bottom of each bar represents the proportion of instances where either IterativeLeveling or RepairDescent can prove optimality by reaching the lower bound provided by FullTransfer. As intended by our instance generation process, very few instances are trivially proved optimal by the heuristics. In contrast, our CP model succeeds in proving optimality much more frequently, even when the FullTransfer lower bound is unattainable. However, providing non-trivial optimality proofs becomes challenging when the number of buffers and windows increases.

In Rouzot's thesis [13], we provide additional experiments. We first evaluate the impact of our search strategy against two strategies on all four Rosetta real-world instances. The *random* search is used as baseline and the popular generic strategy *min-dom* selects the variable with the smaller domain, following the first-fail philosophy [6]. The first two instances (MTP011, MTP012) are solved to optimality in a few seconds with our search strategy *downlink-count*, while we reach the one hour time limit for the two other strategies. For the two other instances (MTP013, MTP014), we can find much better solutions with *downlink-count* in a minute than with *random* or *min-dom* in an hour. These results highlight the importance of an efficient branching strategy for solving the oMDP.

We also report the mean objective values for the three methods across instances with varying numbers of buffers and downlink windows. We observe that for few buffers and downlink windows, our CP model is competitive with RepairDescent, occasionally even outperforming it. In general, IterativeLevel-

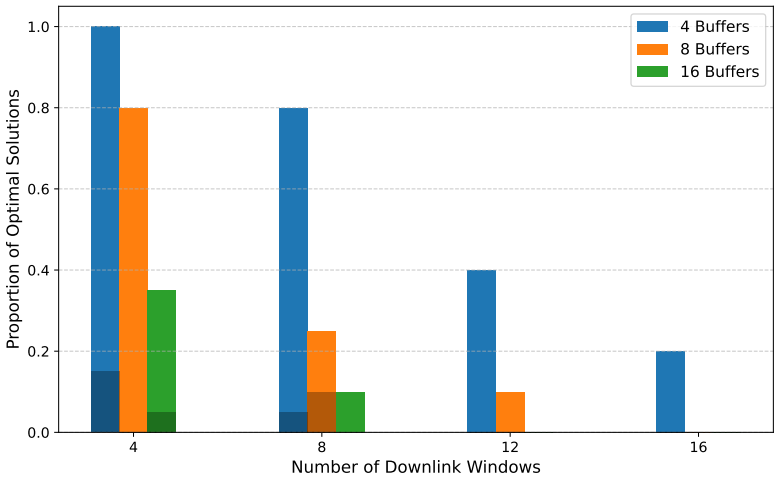

Fig. 2: *Proportion of optimal solutions found for 240 synthetic instances. The shaded areas in the bar plot correspond to the proportion of solutions for which the heuristics are able to prove optimality (by reaching a trivial lower bound).*

ING is dominated by the other two methods, except for the largest instances, where the CP model fails to scale.

7 Conclusion

In this paper, we introduced the first exact approach for solving the overlapping Memory Dumping Problem (oMDP) and provided an enhanced complexity analysis, demonstrating that the problem is strongly NP-hard in the general case. We proposed a constraint programming model incorporating novel global constraints and a search strategy to improve its efficiency. The generic DenseRanking constraint can be applied to any application in which *ranking* is involved. While our approach is slower than the heuristics of [12] and [7] on large instances, it outperforms state-of-the-art methods on smaller instances and is capable of producing non-trivial optimality proofs. Our results suggest that both approaches could be used synergistically for data transfer scheduling: the RepairDescent heuristic for strategic planning (LTP-MTP) and our CP model for tactical and operational planning (STP-VSTP). Furthermore, by leveraging an expressive framework such as constraint programming, our approach offers greater flexibility, allowing additional operational constraints to be integrated into the model.

Acknowledgements This work benefited from ANITI AI Cluster funded by the France 2030 program under Grant Agreement No. ANR-23-IACL-0002.

References

1. Bessière, C., Hebrard, E., Hnich, B., Kiziltan, Z., Walsh, T.: Filtering algorithms for the NValue constraint. Constraints **11**, 271–293 (2006)
2. Chien, S., Knight, R., Stechert, A., Sherwood, R., Rabideau, G.: Using iterative repair to increase the responsiveness of planning and scheduling for autonomous spacecraft. In: International Joint Conference on Artificial Intelligence (IJCAI 1999). Stockholm, Sweden (August 1999)
3. Chien, S., Rabideau, G., Tran, D., Troesch, M., Doubleday, J., Nespoli, F., Ayucar, M.P., Sitja, M.C., Vallat, C., Geiger, B., et al.: Activity-based scheduling of science campaigns for the Rosetta orbiter. In: Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-15. Buenos Aires, Argentina (July 2015)
4. Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., USA (1990)
5. Haddow, C., Whitehead, G., Adamson, K., Sousa, B.: Mission planning-establishing a common concept for esoc's missions. In: SpaceOps 2010 Conference Delivering on the Dream Hosted by NASA Marshall Space Flight Center and Organized by AIAA. p. 1969 (2010)
6. Haralick, R.M., Elliott, G.L.: Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence **14**(3), 263–313 (1980)
7. Hebrard, E., Artigues, C., Lopez, P., Lusson, A., Chien, S., Maillard, A., Rabideau, G.: An efficient approach to data transfer scheduling for long range space exploration. In: De Raedt, L. (ed.) Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22. pp. 4635–4641. International Joint Conferences on Artificial Intelligence Organization (7 2022)
8. Imbriale, W.A.: Large antennas of the deep space network. John Wiley & Sons (2005)
9. van Omme, N., Perron, L., Furnon, V.: OR-Tools user's manual. Tech. rep., Google (2014)
10. Pérez-Ayúcar, M., Ashman, M., Almeida, M., Sitja, M.C., Beteta, J.J.G., Hoofs, R., Kueppers, M., Yaseli, J.M., Merritt, D., Nespoli, F., et al.: The Rosetta science operations and planning implementation. Acta Astronautica **152**, 163–174 (2018)
11. Petit, T., Régin, J.C., Bessière, C.: Specific filtering algorithms for over-constrained problems. In: Principles and Practice of Constraint Programming: 7th International Conference, Paphos, Cyprus, Proceedings 7. pp. 451–463. Springer (2001)
12. Rabideau, G., Chien, S., Nespoli, F., Costa, M.: Managing spacecraft memory buffers with overlapping store and dump operations. In: Workshop on Scheduling and Planning Applications, International Conference on Automated Planning and Scheduling (SPARK, ICAPS 2016). London, UK (June 2016)
13. Rouzot, J.: Combinatorial Optimization for Space Exploration with Constraint Programming : Data Transfers, Scientific Observations, and Operations Scheduling. Ph.D. thesis, Université de Toulouse (2025), `https://www.theses.fr/2025TLSEI026`
14. Simonin, G., Artigues, C., Hebrard, E., Lopez, P.: Scheduling scientific experiments on the Rosetta/Philae mission. In: International Conference on Principles and Practice of Constraint Programming. pp. 23–37. Springer (2012)