An Efficient Approach to Data Transfer Scheduling for Long Range Space Exploration

Emmanuel Hébrard, Christian Artigues, Pierre Lopez, Arnaud Lusson, Steve A. Chien, Adrien Maillard, Gregg R. Rabideau

To cite this version:

Emmanuel Hébrard, Christian Artigues, Pierre Lopez, Arnaud Lusson, Steve A. Chien, et al.. An Efficient Approach to Data Transfer Scheduling for Long Range Space Exploration. The 31st International Joint Conference on Artificial Intelligence (IJCAI-ECAI 2022), Jul 2022, Vienna, Austria. pp.4635-4641, 10.24963/ijcai.2022/643 . hal-03747736

HAL Id: hal-03747736
https://laas.hal.science/hal-03747736

Submitted on 8 Aug 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An Efficient Approach to Data Transfer Scheduling for Long Range Space Exploration

Emmanuel Hebrard[1] , **Christian Artigues**[1] , **Pierre Lopez**[1] , **Arnaud Lusson**[1] , **Steve Chien**[2] , **Adrien Maillard**[2] , **Gregg Rabideau**[2]
[1]LAAS-CNRS, Université de Toulouse, CNRS, France
[2]Jet Propulsion Laboratory, California Institute of Technology, CA, USA
{hebrard,artigues,lopez,lusson}@laas.fr, {steve.a.chien,adrien.maillard,gregg.r.rabideau}@jpl.nasa.gov

Abstract

Long range space missions, such as Rosetta, require robust plans of data-acquisition activities and of the resulting data transfers. In this paper we revisit the problem of assigning priorities to data transfers in order to maximize safety margin of onboard memory. We propose a fast sweep algorithm to verify the feasibility of a given priority assignment and we introduce an efficient exact algorithm to assign priorities on a single downlink window. We prove that the problem is NP-hard for several windows, and we propose several randomized heuristics to tackle the general case. Our experimental results show that the proposed approaches are able to improve the plans computed for the real mission by the previously existing method, while the sweep algorithm yields drastic accelerations.

1 Introduction

Scientific instruments for deep space exploration spacecrafts become more and more sophisticated and consequently produce more and more data that must be sent to Earth. Data management is highly critical as the onboard memory is limited and communication with Earth is often a bottleneck [Vallat *et al.*, 2017; Pérez-Ayúcar *et al.*, 2018].

The Rosetta case is representative of most high profile missions supported by the Deep Space Network where concurrent uploads and downloads make the optimization of data transfers a critical component. ESA and JPL designed an efficient algorithm for solving the memory dumping problem of the Rosetta spacecraft [Chien *et al.*, 2021] in the case where a data production plan is already designed and the decision process concerns transfer priorities. Conversely, for the Philae probe (Rosetta's lander), the transfer priorities were decided a priori, and the experiments were scheduled in consequence [Simonin *et al.*, 2015]. In both cases, data is produced into several memory buffers and the goal is to avoid data loss, which occurs when data is produced into a full buffer. More precisely, we want to maximize the minimum margin, where the margin is the percentage of a buffer's capacity left free, in order to design robust plans where unexpected changes of the dump or fill rates can be absorbed by the margins.

In this paper we consider the same case as Rabideau *et al.* where the data production plan is known and the problem consists in planning memory dumps [Rabideau *et al.*, 2017]. This problem is recurrent in space missions, see e.g. [Oddi *et al.*, 2002a; Oddi *et al.*, 2002b; Righini and Tresoldi, 2010]. Under this assumption, the fill rate of each memory buffer over the planning horizon is part of the input. Data can only be dumped when the spacecraft is visible from Earth. This is materialized by consecutive disjoint downlink windows. Data dumping is a semi-automatized process. For each downlink window a priority has to be assigned to each buffer, then, the transfers follow this priority ordering.

Note that another way to control the dumping process is to deactivate a buffer at a given time-point. This can only be done once per buffer per downlink window, and no reactivation is possible in that window. The buffer then stops data dumping independently of the buffer priority. However, it must be underlined that assigning precise times to events is incompatible with the robustness objective. Indeed, because fill and dump rates may not be known with precision, a buffer might be deactivated when its usage differs from what motivated that decision. Such uncertainties might lead to waste of the bandwidth. Hence, as Rabideau *et al.*, we consider only decisions on buffer priority because such plans better cope with uncertainties. We thus tackle the problem of computing a priority assignment that maximizes the minimum margin.

In Section 2, we define the problem, establish that it is NP-complete in general, and we give an efficient algorithm to simulate data transfers given a priority assignment. Then in Section 3 we provide a polynomial algorithm for a single downlink window. From this algorithm, we derive a heuristic for the case with multiple windows in Section 4. Finally, in Section 5 we report the results of a set of experiments on data instances from the Rosetta mission and we compare the proposed heuristic with the one actually used during the mission.

2 Overlapping Memory Dumping

In the overlapping Memory Dumping Problem (oMDP), we are given m downlink windows, where $[s_j, e_j[$ stands for the time interval in which the downlink j is available, and δ_j for the dump rate for transfers of downlink j. Moreover, there are n memory buffers, where C_i stands for the capacity of buffer

i; $r_i(j)$ for the maximum handover usage of buffer i at the end of downlink window j; and $f_i : \mathbb{R} \mapsto \mathbb{R}^+$ for the piece-wise constant fill rate function of buffer i over time. Let ν be the number of inflection points over all fill rate functions.

The transfers can be controlled by setting a priority function over the buffers defining a *ranking*. Data is transferred in standard-sized packets. At every step, one of the buffers of highest priority among those who have at least one packet of data in memory is selected via round-robin to transfer it.

Let $U_i : \mathbb{R} \mapsto \mathbb{R}$ be the quantity of data on buffer i over time and $g_i : \mathbb{R} \mapsto \mathbb{R}$ be the transfer rate out of buffer i over time.

In between downlink window, all transfer rates are null and the memory usage growth is the fill function, for every buffer i. In particular, for any time t preceding the start of the first downlink s_1, we have $g_i(t) = 0$ and $U_i(t) = \int_0^t f_i(x)\,dx$.

During a downlink window with dump rate δ, however, the effective transfer rate at time t depends on the priority function P, and on the usages and fill rates at time t.

We consider the decision problem of finding a priority ranking for every downlink, such that the peak usage of any buffer (given its fill rate functions, and the data transfer system described above) is less than its capacity, i.e., without *data loss*. The objective function is actually to maximize the minimum margin, $\min\{M(i) \mid i \in B\}$, where $M(i)$ is defined as one minus the ratio between the peak usage and the capacity of buffer i. However, it can be achieved by binary search using an algorithm for the decision problem above.

Example 1. *Figure 1 shows a plan with two downlinks, both with dump rate* 1*, and 3 buffers all of capacity* 1*. Fill rates are figured by colored rectangles, e.g.,* $f_1(t) = \frac{1}{2}$ *for* $t \in [0,1]$*. In the first downlink, all buffers have equal priority* $P_1(1) = P_1(2) = P_1(3) = 1$*, in the second, buffer priorities are given by their index (*$P_1(i) = i$*, i.e., buffer 3 has the highest priority then 2 then 1). The resulting buffer usage functions* $U_i(t)$ *are displayed in black line. The minimum margin is* $\frac{1}{6}$*.*

2.1 Simulation of a Dumping Plan

As shown by Simonin *et al.*, it is possible to compute the data transfers efficiently, that is, with a complexity polynomial in the number of changes in the fill rate functions, as opposed to the number of packets of data [Simonin *et al.*, 2015]. Here we use a similar method, and crucially, we show that one does not need to know the exact priority function to compute the usage of a given memory buffer over time and that a polynomial number of time points have to be considered. This algorithm is denoted `Simulation` in the remainder of the paper.

Lemma 1. *Given the set of buffers of strictly higher priorities and the set of buffers of equal priority, the usage at any time of a given buffer can be computed in polynomial time.*

Proof. Consider first the problem of computing the instant transfer rate $g_i(t)$ at time t of a buffer i when the usages $U(t)$ are known. Given a set of buffers Ω, let $f_\Omega(t) = \sum_{i\in\Omega} f_i(t)$ be the sum of the instant fill rates for buffers in Ω.

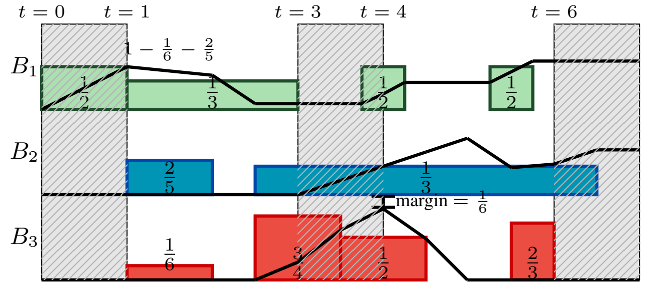


Figure 1: Illustration of Example 1. White areas indicate visibility.

Let Γ be a set of buffers with priority over buffer i. If at least one buffer in Γ contains some data at time t, no buffer outside Γ may dump any data. Otherwise, buffers in Γ globally dump in downlink window j at rate $g_\Gamma(t) = \min(\delta_j, f_\Gamma(t))$ where δ_j is the dump rate of the downlink j, which leaves a residual dump rate of $\delta_j - g_\Gamma(t)$. *In example 1, at time* $t = 5.5$*, buffer 2 thus dumps at the full residual rate* $\delta_2 - f_3(t) = 1/3$ *since it is not empty, and buffer 1 cannot dump data.*

In other words, the set of high-priority buffers takes all the available bandwidth (up to its total fill rate if they are all empty) and the exact priority function within Γ is irrelevant to lower priority buffers.

Now, let $g_i(t, \Omega, \Delta_j(t))$ denote the transfer rate from buffer i if the effective dump rate available to buffers in Ω at time t is $\Delta_j(t)$ and all buffers in Ω have the same priority as i. It can be computed recursively as follow:

- $\Delta_j(t)$ is initialized with the residual rate $\delta_j - g_\Gamma(t)$. If Ω contains empty buffers, let ℓ be one with minimum fill rate. If $f_\ell(t) < \Delta_j(t)/|\Omega|$, then $g_i(t, \Omega, \Delta_j(t)) = f_\ell(t)$. This value can be subtracted to the effective dump rate and ℓ be ignored in order to compute, for all $i \neq \ell$, $g_i(t, \Omega, \delta_j) = g_i(t, \Omega \setminus \{\ell\}, \delta_j - g_\ell(t, \Omega, \delta_j))$.
- Otherwise, every buffer in Ω could transfer at a rate greater than or equal to $\Delta_j(t)/|\Omega|$, and therefore they transfer at that rate through the round-robin policy.

In Example 1, at time $t = 1$ *and* $\Omega = \{1, 2, 3\}$ *we initially have* $\Delta_j(t) = 1$*. Buffer 3 is empty and* $f_3(t) = 1/6 < 1/3$*, so we set* $\Delta_j(t) = 1 - 1/6 = 5/6$ *and* $\Omega = \{1, 2\}$*. Buffer 2 is empty and* $f_2(t) = 2/5 < 5/12$*, so we set* $\Delta_j(t) = 5/6 - 2/5 = 13/30$*, which is the transfer rate of buffer 1.*

Therefore, if we know the usage $U_i(t)$ of each buffer i at time t, their fill rates $f_i(t)$ and the current dump rate δ_j, we can compute all the transfer rates at time t by applying the method above, starting with the batch Ω of buffers of highest priority, until either all buffers have been explored or the residual bandwidth at time t is null. This takes $O(n \log n)$ time.

Moreover, notice that fill and dump rates are piece-wise constant and hence the transfer rates are also piece-wise constant, and the usage rates are piece-wise linear. Therefore, we only need to compute those rates at the inflection points, i.e., when one of the following events happens: *the dump rate changes* or *the fill rate of a buffer changes* or *a buffer becomes empty*.

Changes in dump rates occur when downlink windows starts and ends, and are thus known in advance as are changes in fill rates. Moreover, given the fill and dump rates at time t, the transfer rate of each buffer can be computed as shown above, and hence it is easy to know when a buffer would become empty at the current rate. Such "empty buffer" events need to be inserted only when they are relevant, i.e., when they come earlier than the next rate-change event. Moreover, at most n such events can occur before the next rate-change event. Therefore, there are $O(n(\nu + m))$ events to explore in total.

For each event, the memory usages are updated w.r.t. the previous event, and dump and fill rates are updated in $O(n)$ time. The complexity of an iteration is dominated by the computation of the round-robin dump rate g_i for every buffer i. As explained above, this can be done in $O(n \log n)$ by sorting the buffers and applying the proposed recursion. Moreover, the $\Theta(\nu + m)$ events known "a priori" must be sorted, which costs $\Theta((\nu + m) \log(\nu + m))$ and the worst case time complexity is in $O((\nu + m)(\log(\nu + m) + n^2 \log n))$. □

2.2 Problem Complexity

We consider here the decision problem PRIORITYALLOC: "Is there a priority assignment for each downlink window such that there is no data loss?".

Theorem 1. *The problem* PRIORITYALLOC *is NP-complete.*

Proof. It is in NP by Lemma 1, since verifying that a priority allocation entails no data loss is polynomial. To show hardness, we use a reduction from PARTITION, which, given a set $A = \{a_1, \ldots, a_m\}$ of positive reals with $\sum_{j=1}^{m} a_j = 1$, asks if there is a set $S \subseteq A$ such that $\sum_{a_j \in S} a_j = \frac{1}{2}$.

We build an instance $\pi(A)$ of PRIORITYALLOC with m downlink windows, such that window j has duration a_j and dump rate 2, and $m + 2$ buffers $1, \ldots, m + 2$. Buffers $m + 1$ and $m + 2$ have capacities $\frac{3}{2}$, while buffer j has capacity a_j for $j \in [1, m]$. The fill rates f_{m+1} and f_{m+2} are equal to 1 for a duration 1 before the first downlink window, then they are equal to 1 during every downlink and null otherwise. Moreover, f_j is equal to 2 during window j and is null otherwise.

Any solution of $\pi(A)$ satisfies the four following properties:

1. *Every buffer must be full at the end of the last window*. Indeed, the cumulative amount of data produced is 6 while a most 2 can be dumped. Moreover, the capacities of buffers $m + 1$ and $m + 2$ add up to 3 and the capacities of all other buffers add up to 1.
2. *Buffer j cannot have priority strictly higher than both $m + 1$ and $m + 2$ in window j*. Indeed, if j had strict priority over $m + 1$ and $m + 2$, then all data produced on this buffer would be dumped and it would be empty at $t = e_m$, which contradicts Property (1).
3. *In window j, for any $k < j$, the priority of k must be low enough so that no data is dumped from this buffer*. This is a direct consequence of Property (1) since no more data will be produced on buffer k.
4. *Either $m + 1$ or $m + 2$ must have a strictly lower priority than j, and none can be of strictly higher priority in window j*. Indeed, if neither $m + 1$ or $m + 2$ is ranked lower than j, then it cannot get more than $1/3$ of the bandwidth. Therefore, it cannot dump more than $2a_j/3$ in window j and the residual data ($4a_j/3$) exceeds its capacity. Moreover, if either $m + 1$ or $m + 2$ has strict priority over j, it will take the whole bandwidth since they both contain data at all times. Therefore, buffer j would not dump any data and hence exceeds its capacity.

Therefore, on window j, there are only two feasible ranking:

1. $P(m + 1) = P(j) > P(m + 2) \geq P(k)\ \forall k \in [1, j - 1]$
2. $P(m + 2) = P(j) > P(m + 1) \geq P(k)\ \forall k \in [1, j - 1]$

In the first case, buffers $m + 1$ and j share the bandwidth equally (i.e., the dump rate is 1 for each) and at the end of window j, the data on buffers j and $m + 2$ has increased by a_j. The usage of all other buffers remains the same.

The second case is symmetric, and at the end of window j, the data on buffers j and $m + 1$ has increased by a_j.

In other words, there is an equivalence between the choice among these two rankings and the choice of a partition for element a_j. Therefore, given a solution S of an instance A of PARTITION, we can build a solution of the instance $\pi(A)$ of PRIORITYALLOC where, on window j, the priority is $P(m + 1) = P(j) > P(m + 2) > P(k)\ \forall k \in [1, j - 1]$ if $a_j \in S$ and $P(m + 2) = P(j) > P(m + 1) > P(k)\ \forall k \in [1, j - 1]$ if $a_j \in S$ otherwise. It can be verified that this solution does not exceed any buffer's capacity.

Consider now a solution of the instance $\pi(A)$. On each window j, the usage of exactly one of buffers $m + 1$ and $m + 2$ grows by a_j. Since the final usage of both buffers is $\frac{1}{2}$, they form an equi-partition of A and we can conclude that the instance A of PARTITION is satisfiable. □

This reduction requires m buffers and m downlink windows, and hence does not preclude that an efficient algorithm exists if either parameter is fixed. In the next section we show that the problem is polynomial for a single window.

3 Polynomial Algorithm for a Single Window

Let $M_{\Gamma \prec \Omega}(i, j) = \left(C_i - \max_{t \in [e_{j-1}, e_j[}(U_i(t))\right) / C_i$ be the usage margin of i in the interval $[e_{j-1}, e_j[$, with Γ the set of buffers with priority over i and Ω the set of buffers with same priority as i. The usage $U_i(t)$ of buffer i at all times t can be computed efficiently using the algorithm `Simulation` with a priority assignment compatible with the definition of Γ and Ω, since the exact priority within Γ is irrelevant. It follows that $M_{\Gamma \prec \Omega}(i, j)$ can also be computed efficiently.

The four following lemmas lead to a method (Algorithm 1) to decide if there exists a priority assignment of the buffers such that given their usage at time e_{j-1}, every buffer has a margin of at least *obj* in $[e_{j-1}, e_j[$, with $e_0 = 0$ and $s_{m+1} = \infty$.

Lemma 2. *$i = k$ or $k \notin \Omega$ or $M_{\emptyset \prec \Omega}(i, j) \geq M_{\{k\} \prec \Omega \setminus \{k\}}(i, j)$.*

Algorithm 1: An Algorithm for a Single Window

```
Algorithm: SingleWindow
Data: A downlink window j, a target margin obj
Result: Returns ∅ and sets the priority ranking P_j on B for
        downlink window j, if there is one that achieves the
        target margin. Otherwise, returns a minimal set of
        buffers that would exceed the target margin even if
        all other buffers were given less priority.
p ← 0
B ← {1, ..., n}
while B ≠ ∅ do
    Ω ← B
1   repeat
        Ω' ← Ω
2       Ω ← {i | i ∈ Ω & M_{B\Ω≺Ω}(i, j) ≥ obj}
    until Ω' = Ω
3   if Ω = ∅ then return B
4   foreach i ∈ Ω do P_j(i) ← p
5   B ← B \ Ω
    p ← p + 1
```

Proof. If given a strictly higher priority k can only take more bandwidth, and hence the margin of i can only decrease. □

Lemma 3. *If $M_{\emptyset \prec \Omega}(i, j) < M$, then the priority of i cannot be the lowest of all buffers in Ω, if the minimum margin is M.*

Proof. By lemma 2, the margin can only decrease if any buffer except i is given more priority. □

Lemma 4. *If for every $i \in \Omega$, we have $M_{\emptyset \prec \Omega}(i, j) < M$, then there is no solution with margin M or larger.*

Proof. By lemma 3, no buffer in Ω can be of lowest priority among all buffers in Ω, which is a contradiction. □

Lemma 5. *If $i \in \Omega \implies M_{\Gamma \prec \Omega}(i, j) \geq M$, then there is a solution with margin at least M if and only if there is a solution with margin at least M for the set of buffers Γ.*

Proof. "If": Suppose that there is a solution for Γ with margin $\geq M$. Give the same priority (strictly lower than that of any buffer in Γ) to all buffers in Ω. Since $M_{\Gamma \prec \Omega}(i, j) \geq M$ for all $i \in \Omega$, this solution extends to a solution of $\Gamma \cup \Omega$ with margin $\geq M$. "Only if": Trivial. □

Algorithm 1 first tries to determine which buffers can be of lowest priority. In Loop 1 the margin of all remaining buffers are computed conditional to all having the lowest priority. By lemma 3, the buffers whose margin is below obj cannot be of lowest priority. Therefore, we remove those buffers with margin less than M from the candidate set for lowest priority Ω and iterate. If eventually no buffer can have a margin equal to or larger than M, we can conclude that there is no solution by lemma 4. Otherwise, in Line 4, we set the priority of the buffers in Ω, and we can ignore these buffers by lemma 5.

In the worst case, i.e., when the priority has to be a total order and a single buffer is removed from Ω at each iteration of Loop 1, Algorithm 1 requires $O(n^2)$ calls to Simulation.

Algorithm 2: A heuristic for multiple downlinks

```
Algorithm: Descent
Data: An oMDP instance
      X = (C_1, ..., C_n, [s_1, e_1[, ..., [s_m, e_m[, f), initial
      memory usage U, a step parameter ε
ub ← RelaxBandwidth(X)
lb ← 0
while lb < ub do
    clear handover constraints
1   P, lb ← DownlinkCount(X, lb)
    M ← Repair(X, P, U, lb + ε)
    if M > lb then lb ← M
    else ub ← lb
return lb
```

Notice that, besides the margin, we can check handover constraints (specific bound on the usage of a buffer at the end of a downlink) with the same method. Moreover, we also only need to know the sets of buffers of higher and equal priority. In other words, on the downlink window $[s_j, e_j[$, we can change the test at Line 2 to: $M_{B \setminus \Omega \prec \Omega}(i, j) \geq obj$ & $U_i(e_j) \leq r_i(j)$ and Algorithm 1 remains correct.

4 A Heuristic for Priority Assignment

Here we consider the general (NP-complete) problem, with buffer usage carrying over multiple downlink windows.

We propose a heuristic (Algorithm 2) which implements a greedy "descent": a seed priority assignment is computed using the heuristic DownlinkCount [Rabideau *et al.*, 2017] in Line 1, and iteratively "repaired" via Algorithm 3 to achieve a strictly higher target margin (by a step $\epsilon = 10^{-5}$).

It simulates the current priority assignment until reaching downlinks j at which the target margin is exceeded. Then, at line 2 it calls Algorithm 1 to check whether there exists a priority assignment P_j that achieves the expected margin. If there is such an assignment, it is run by Simulation at Line 1 and we advance to downlink window $j + 1$. If the last window is reached, an improving solution has been found. Otherwise, if downlink j does not have a priority assignment without data loss, Algorithm 1 returns a set B of buffers guaranteed to exceed their target margin given the current usage at the end of downlink $j - 1$, even if they are given (globally) the highest priority. Therefore, the only way to avoid data loss in this downlink is to reduce the residual load for at least one of these buffers at the end the previous downlink.

In order to avoid branching on the possible ways of reducing this load, we instead add one randomly chosen handover constraint at Line 4. We pick a random buffer i in the set B and we set its maximum handover value $r_i(j-1)$ to the current usage $U_i(j-1)$ to which we subtract the gap we need to achieve the expected margin: the margin we obtain when buffer i is given highest priority with all other buffers in B (that is, $M_{\emptyset \prec B}(i, j)$) minus the expected margin obj. Then the previous window is solved again with this new constraint: it "backtracks" by decreasing the current downlink window j

Algorithm 3: A heuristic to repair a priority assignment

Algorithm: `Repair`

Data: An oMDP instance $X = (C_1, \dots, C_n, [s_1, e_1[, \dots, [s_m, e_m[, f)$, priority assignment P, current usage U, target margin obj

Result: Update the priority assignment P and return the corresponding margin $obj' > obj$, or $\emptyset$ if no improving priority assignment was found

$obj' \leftarrow 1$
$j \leftarrow 1$
while $j \leq m + 1$ **do**
1 `Simulation`($[e_{j-1}, e_j], X$)
$M \leftarrow \min\{\frac{C_i - U_i(t)}{C_i} \mid i \in \{1, \dots, n\}, t \in [e_{j-1}, e_j]\}$
if $M \geq obj$ **then**
$obj' \leftarrow \min(M, obj')$
$j \leftarrow j + 1$
else
repeat
2 $B \leftarrow$ SINGLEWINDOW($\{1, \dots, n\}, [s_j, e_j[$)
if $B \neq \emptyset$ **then**
if $j = 1$ **then return** $\emptyset$
3 $j \leftarrow j - 1$
pick a random buffer $i \in B$
4 $r_i(j) \leftarrow U_i(e_j) - (M_{\emptyset \prec B}(i, j) - obj)C_i$
until $B = \emptyset$
return obj'

at Line 3. If the current downlink window is the first ($j = 1$), then no such constraint can be posted and we fail.

Observe that Algorithm 2 converges since at Line 4, one maximum handover value strictly decreases. Therefore, eventually, either the last window will be solved with the expected margin, or the first window will be proven unsatisfiable (given the choice of handover constraints).

5 Experimental Evaluation

The software used for the real mission (DALLOC) is a proprietary software. Therefore, we could not directly compare our method to it and instead we re-implemented the best heuristic described by Rabideau *et al.*. This approach relies on a heuristic called DOWNLINKCOUNT (DC for short). This methods greedily assigns priorities based on when each buffer would exceed a given target margin. More precisely, it counts in how many downlinks would occur before exceeding the target if no data is dumped. At each downlink window $j \in [1, m]$, the transfers are computed starting from the current usage, and with a dump rate equal to 0. Then a priority assignment is extracted as defined above, and window j is simulated using the same procedure but with dump rate δ_j. It is important to notice that the choice of target margin can change the priority assignment. Therefore, in DALLOC, this heuristic is used within a binary search. The resulting algorithm is denoted ITERATEDLEVELING (IL for short). We do not compare with two other methods introduced by Rabideau *et al.* that were found to be less effective, in particular because they are not suited to the binary search method (IL).

	Margin				Avg. CPU
	`MTP1`	`MTP2`	`MTP3`	`MTP4`	
DALLOC's	40.4	46.5	17.8	29.8	14.6
Ours	46.4	68.1	17.8	30.8	0.018

Table 1: New and old implementations of DOWNLINKCOUNT.

In this section we first give some evidence that our implementation using a faster data transfer simulation algorithm is more efficient than DALLOC's. Then, we compare REPAIRDESCENTwith our implementation of ITERATEDLEVELING, using the same hardware and common routines.

We used the same data set as Rabideau *et al.*, constituted of four scenarios (`MTP1`, `MTP2`, `MTP3` and `MTP4`), corresponding to the whole activity sequence of Rosetta divided in four quarters. The whole sequence has over 40,000 data production events for 16 memory buffers and 324 downlink windows. Its duration is about three months and a half with a precision of one second. It turns out that the first three can be solved to optimality as a simple lower bound matches the results of our method or DALLOC's ITERATEDLEVELING. The last instance (`MTP4`) is not trivial. However, in order to get significant results, we generated a synthetic data set. We first added 48 new buffers, for a total of 64, as follows: For each memory buffer in the original data set, we created three "clones" whose capacities and fill rates were multiplied by a factor drawn uniformly randomly in $[0.8, 1.2]$ and whose activities were shifted in time drawn uniformly randomly in $[0s, 3600s]$. Then, for each value of n in $\{8, 12, \dots, 36\}$ we randomly chose n memory buffers, and generated one instance for a set of time intervals defined by the ends of two downlink windows x and y. We generated 14160 instances: each one is parameterized by the number of buffers n and the first and last considered downlinks are multiple of 5 $x, y \in \{0, \dots, 325\}$ such that $x + 10 \leq y$.

5.1 Re-implementation of DOWNLINKCOUNT

Table 1 shows the results published in [Rabideau *et al.*, 2017] for DOWNLINKCOUNT and the results of our implementation. We believe that the main difference is the algorithm used to play the downlinks given a priority assignment. We used Algorithm `Simulation`, and we can clearly see a speedup: even though the hardware may not be equivalent, it cannot account for the speedup of several orders of magnitude. We also observe that our version finds downlink plans with higher margin for `MTP1` and `MTP2`. This is because the results of Rabideau *et al.* include pre-assigned (high) priorities for some buffers due to some specific constraints of the actual mission. We are confident, however, that this implementation is truthful, and at least as effective as the one in DALLOC.

5.2 Comparison with the State of the Art

Now that we have established that our implementation is at least as efficient as the one in DALLOC,

	Margin				Avg. CPU
	`MTP1`	`MTP2`	`MTP3`	`MTP4`	
UB	46.4	72.5	54.8	53.4	
IL	46.4	72.5	54.8	48.5	0.116
RD	46.4	72.5	54.8	**52.8**	0.088

Table 2: Comparison with the state of the art on real scenarios.

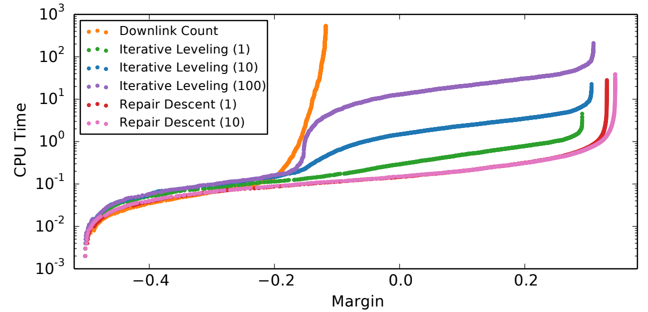
	≤ 12		16-20		24-28		≥ 32	
	M.	CPU	M.	CPU	M.	CPU	M.	CPU
DC	28.7	0.0	-41.5	0.1	-249.3	0.1	-460.7	0.1
IL	47.1	0.3	22.9	0.8	-22.4	1.4	-68.7	2.2
RD	49.6	0.1	27.4	1.1	-15.8	7.5	-60.7	31.3

Table 3: Comparison with the state of the art on synthetic scenarios.

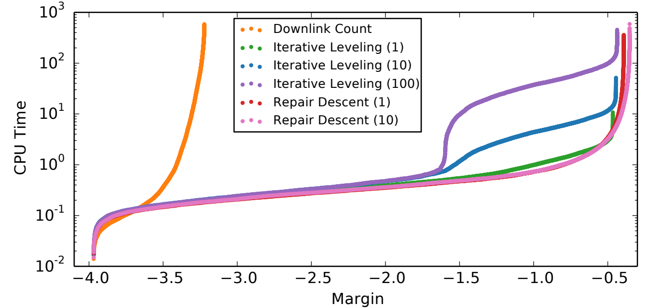
we can compare the algorithm proposed in this paper (REPAIRDESCENT) with the approach introduced by Rabideau *et al.* (ITERATEDLEVELING). Table 2 shows the results of both approaches (denoted respectively RD and IL) on the original instances. We report the minimum margin (in percent of the capacity) and the average CPU time. The three first instances are solved to optimality, witnessed by the simple upper bound (UB) consisting in allocating the full bandwidth to every buffer. Observe, moreover, that our implementation of DOWNLINKCOUNT also solves `MTP1` optimally, but not `MTP2` nor `MTP3`. This indicates that on `MTP1` in particular, the bottleneck on each downlink window is mainly due to a single memory buffer. Yet, finding this critical buffer might not be straightforward (such as in `MTP2` and `MTP3`), since priority decisions can have an impact far downstream. On `MTP4`, however, the bottleneck is likely due to a much more complex interaction between buffers, and no approach can reach the upper bound. Our method improve the minimum margin by 4.3 points over the solution found by ITERATEDLEVELING. CPU times are comparable and very low (under a second for all instances).

The real scenarios do not give us enough datapoints for a statistically significant comparison. Therefore we ran these approaches on the large dataset described above. Table 3 gives an overview of the results. We ignored the instances for which both REPAIRDESCENT and ITERATEDLEVELING find the same margin (32% of the data set). For the other scenarios we give the average minimum margin and the average CPU time for these two algorithms (denoted respectively RD and IL) as well as for the heuristic DOWNLINKCOUNT (DC), for four classes of instances: those with 12 buffers or fewer, with less 20 buffers, with 24 to 28 and with 32 buffers or more. We can see that our algorithm finds more robust transfer plans, by about 2.5 points for the first class, 4.5 points for the second class, 6.6 for the third and 8 points for last. It requires, however, much more CPU time on large instances.

These average results on CPU times might be deceiving, however. Figure 2 shows the average best margin X after Y seconds for the instances with at most 20 buffers and for more



(a) $\leq$ 20 buffers



(b) $>$ 20 buffers

Figure 2: Comparison with the state of the art on synthetic scenarios.

than 20 buffers. We can see that REPAIRDESCENT is actually *faster* than ITERATEDLEVELING to find solutions of similar quality. In other words, it takes more time to converge because it continues to optimize the plan for longer. We ran randomized versions of the algorithms. In the case of DOWNLINKCOUNT, each of the m priority decisions randomly uses a total ordering based on the two versions discussed earlier, based *time* or the *window* of the first data loss. The heuristic is then called 10000 times. For ITERATEDLEVELING, we simply use the randomized version of DOWNLINKCOUNT whereas REPAIRDESCENT is naturally randomized because handover constraints are chosen randomly. In both cases, the number of runs per iteration is shown besides the method name. From these graphs, it is clear that investing more time via randomization can pay off, however, REPAIRDESCENT is almost always the best choice.

6 Conclusion

We proposed an efficient algorithm to simulate data transfers under priorities, and overlapping data production and dumping. Then, we showed that deciding if there is a priority assignment without data loss is polynomial for a single downlink, but NP-complete for multiple downlinks. Finally, we proposed a heuristic for allocating priorities over multiple windows based on the exact algorithm for a single window. This approach significantly outperforms the state of the art for the overlapping Memory Dumping problem.

Acknowledgements

Part of this research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration (80NM0018D0004). Part of this work was supported by the AI Interdisciplinary Institute ANITI, funded by the French program "Investing for the Future – PIA3" under grant agreement no. ANR-19-PI3A-0004. We would like to thank the European Space Agency and in particular Miguel Perez Ayucar and Federico Nespoli for providing data.

References

[Chien *et al.*, 2021] Steve Chien, Gregg Rabideau, Daniel Q. Tran, Martina Troesch, Federico Nespoli, Miguel Perez Ayucar, Marc Costa Sitjà, Claire Vallat, Bernhard Geiger, Fran Vallejo, Rafael Andres, Nico Altobelli, and Michael Kueppers. Activity-based scheduling of science campaigns for the Rosetta orbiter. *Journal of Aerospace Information Systems*, 2021. doi: 10.2514/1.I010899.

[Oddi *et al.*, 2002a] Angelo Oddi, Amadeo Cesta, Nicola Policella, and Gabriella Cortellessa. Automating the generation of spacecraft downlink operations in mars express: analysis, algorithms and an interactive solution aid. Technical report, Italian National Research Council, 2002.

[Oddi *et al.*, 2002b] Angelo Oddi, Amadeo Cesta, Nicola Policella, and Gabriella Cortellessa. Scheduling downlink operations in mars express. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, 2002.

[Pérez-Ayúcar *et al.*, 2018] Miguel Pérez-Ayúcar, Mike Ashman, Miguel Almeida, Marc Costa Sitjà, Juan José García Beteta, Raymond Hoofs, Michael Kueppers, Julia Marín Yaseli, Donald Merritt, Federico Nespoli, and Eduardo Sánchez Suarez. The Rosetta science operations and planning implementation. *Acta Astronautica*, 152:163–174, 2018.

[Rabideau *et al.*, 2017] Greg Rabideau, Steve Chien, M. Galer, Federico Nespoli, and Marc Costa Sitjà. Managing spacecraft memory buffers with concurrent data collection and downlink. *Journal of Aerospace Information Systems*, 14(12):637–651, 2017.

[Righini and Tresoldi, 2010] Giovanni Righini and Emanuele Tresoldi. A mathematical programming solution to the Mars Express memory dumping problem. *IEEE Trans. Syst. Man Cybern. Part C*, 40(3):268–277, 2010.

[Simonin *et al.*, 2015] Gilles Simonin, Christian Artigues, Emmanuel Hebrard, and Pierre Lopez. Scheduling scientific experiments for comet exploration. *Constraints: An Int. J.*, 20(1):77–99, 2015.

[Vallat *et al.*, 2017] Claire Vallat, Nicolas Altobelli, Bernhard Geiger, Bjoern Grieger, Michael Kueppers, Claudio Muñoz Crego, Richard Moissl, Matthew G.G.T. Taylor, Claudia Alexander, Bonnie Buratti, and Mathieu Choukroun. The science planning process on the Rosetta mission. *Acta Astronautica*, 133:244–257, 2017.