Contents

1 Hot restart

In the last section we discussed a somewhat fine-grained method for assigning nonzeros to either A_r and A_c, as we make N independent choices (one for each nonzeros) and reuse some of the partitioning information (namely, whether a row or a column is cut or uncut).

It is possibile, however, to take a coarser approach and decide for more than one nonzero at a time: the "hot restart" heuristic does precisely so, making choices for a subset of a row or a column of A: in particular, we do not only take into account whether a row/column is cut or uncut, but also reuse information on which subsets of rows and column were assigned to the same partition.

The outline of this heuristic is the following:

```
(1) Compute the priority vector $v$
(2) Use $v$ with the Overpainting algorithm and compute $A_c$ and $A_r$
```

Algorithm 1: Hot restart

The general idea is that we compute te vector v, which is simply a permutation of the sequence of integers $0, \dots, m+n-1$: the numbers $0, \dots, m-1$ represent the m rows of A, while $m, \dots, m+n-1$ represent the n columns of the matrix.

This priority vector is necessary for the true core of this heuristic, which is the overpainting algorithm and it is described as follows:

```
Input: Vector of priorities $v$ (a permutation of the vector $[0, \dots, m+n-1]$)
Output: $A_r$,$A_c$

  for $i = 0, \dots, m+n-1$ do
    Mark all the unmarked nonzeros in row/column $i$
    if $v_i < m$ then
      Assign the marked nonzeros to $A_r$
    else
      Assign the marked nonzeros to $A_c$
    end if
  end for
```

Algorithm 2: Overpainting

To better explain the functioning of this algorithm, we start iterating through the elements of v: if $0 \leq i < m$, i represents a row, and therefore we assign all of the nonzeros (not previously assigned) of i to A_r, otherwise i represents a column and we assign its nonzeros to A_c; in any case, all of the nonzeros considered in this step are going to be together in the next partioning.

Algorithm 2 is just one of the possible ways of describing the same idea. In another, we could consider (again) individually each nonzero a_{ij} and look up v and see whether $i < j$ (where $<$ is to be intended as "i precedes j" and not as the comparison of the values) or the other way around; in the first case, the row has more priority and we assign a_{ij} to A_r, otherwise we assign it to A_c.

Alternatively, we could iterate through v *backwards* and forget about the marking of the nonzeros: whenever we consider the integer i, we assign all of its nonzeros together (to A_r or A_c, depending on the identity of i); in each subsequent assignment we are free to reassign nonzeros, and it will happen exactly twice for each of them. The assignment that will stick to each nonzero will be then the last one performed, which represent the row/column that precedes the other in v. This is the variant that we implemented and that gave the name of the algorithm: each nonzero is "painted" and then "painted over" a second time.

Note that this overpainting algorithm is completely deterministic: A_r and A_c are uniquely determined by the vector v and therefore the heuristic part of it lies entirely in the choice of this priority vector, as described in the next Section.

2 Computation of the priority vector

At the beginning of the previous section, we claimed that the hot restart heuristic can take into account a discrete amount of information from the previous partitioning, and this happens precisely during the computation of the priority vector. As it is not known exactly which information is good to employ and which one is to be discarded, we took a structured approach and explored a wide variety of possibilities during the generation of v.

In particular, we defined several *generating schemes*, which can be summarized as any path in the directed graph shown in Figure 1. Each one of these generating schemes has three phases[1]:

1. **Use of previous partitioning**: as our final goal is to build an iterative algorithm, it is quite straightforward that we have to re-employ some of the information obtained with the previous iteration. However, in order to understand its effectiveness, a comparison also with a full restart might be convenient. Therefore, we distinguish between:
   - partition aware (`pa`): the set $[0, \ldots, m + n - 1]$ is divided in two, uncut rows/columns before cut rows/columns (so with an higher priority). The next phases of the heuristic are performed in each of these two subset independently, and they are never mixed again.
   - partition oblivious (`po`): the set $[0, \ldots, m + n - 1]$ is kept in one piece.
2. **Sorting**: as v expresses priority, we want to have an effective way of ordering the rows/columns. A possible method is to look at the number of nonzeros and use that as criterium for sorting the rows/columns. Therefore we distinguish between:
   - `unsorted`: we keep the array from the previous phase untouched, and consider as all of the row/columns are tied.
   - `sorted`: we perform a sorting, which can be either in ascending order (`asc`) or descending order (`desc`). Then, we can decide to move all row/columns that have exactly 1 nonzero at the back of the block, thus giving them the lowest priority. From a theoretical point of view, this is always convenient because by assigning such nonzeros to the row/column where they are not the only element, never yields a communication cost. To denote this process we make a comparison with typography and call these row/columns *widows*: if the heuristic perform this move, we denote it as `w`, otherwise as `nw`.
3. **Tie-breaking**: we can find a finder method of arranging the elements of v, especially where, according to the previous phase, there were elements tied. We use three methods for tie-breaking:
   - `simple`: no actual tie-breaking is performed. The vector is left as-is from the previous phase.
   - `random`: the tied elements are shuffled randomly.
   - `mix`: we try to get, as much as possible, that rows and columns are equally distributed. This is a move toward balancing of the number of nonzeros of A_r and A_c. In order to achieve such distribution, there are two main ways: alternation (`alt`) and spread (`spr`); suppose there are k rows and $2k$ columns tied and we start with a row: with the first distribution we have that the first $2k$ elements are rows and column alternated, and then k columns $(r, c, r, c, r, c, \ldots, c, c, c, c)$, whereas with the second one we get k sequences of row, column, column $(r, c, c, r, c, c, \ldots, r, c, c)$. We still have to decide whether to start this distribution with a `row`, or with a column (`col`).

[1]the verbatim words are the labels used to represent that element in both Figure 1 and the Tables with the numerical results.

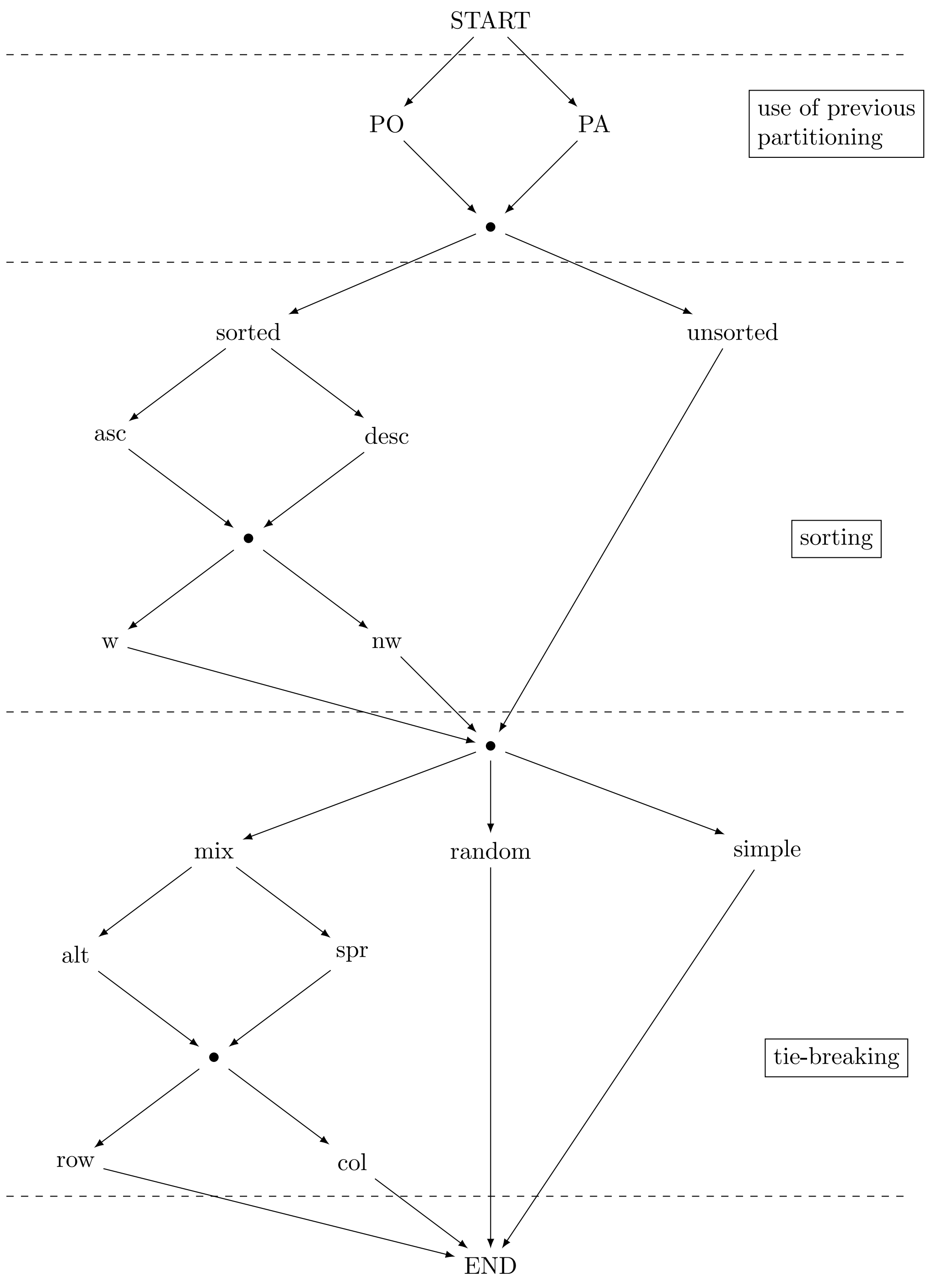

Figure 1: Directed graph that represents the family of heuristics used (any path from START to END). Dummy nodes (the ones without any label) were added in order to reduce the number of edges and ease legibility.