Partial computation of the inverse of a large, sparse matrix

François-Henry Rouet

08/09/09

Context of my study

Context

- Collaboration with researchers in Astrophysics from the CESR (Centre d'Etude Spatiale des Rayonnements).
- Their application (data analysis) involves the computation of the diagonal of the inverse of a large, sparse matrix (largest problem of size 148000).
- This computation is by far the most expensive part of their application ($\simeq$ 10000s).

Aspects of my work

- Performance analysis:
  - Comparison of different techniques for computing A^{-1}.
  - Efficient computation of $A = B^T B$ (MA49).
  - Analysis of the whole application.
- Tackling an open problem about right-hand sides permutations for the computation of a set of entries in A^{-1} $\leftarrow$ main topic of this talk.

Plan

- Computing a set of entries of A^{-1}.
- Influence of right-hand sides permutations; preliminary experiments.
- Overview of the different approaches proposed.
- Focus on the structural approach: shape of an optimal solution.
- Constructive heuristics.
- Perspectives.

Computing a set of entries in A^{-1}

- The approach used relies on:
  - a traditional solution phase: $a^{-1}_{i,j} = (A^{-1}e_j)_i$
  - the use of a direct solver: once one has factorized A (e.g. $A = L\,U$), $a^{-1}_{i,j}$ can be obtained by:

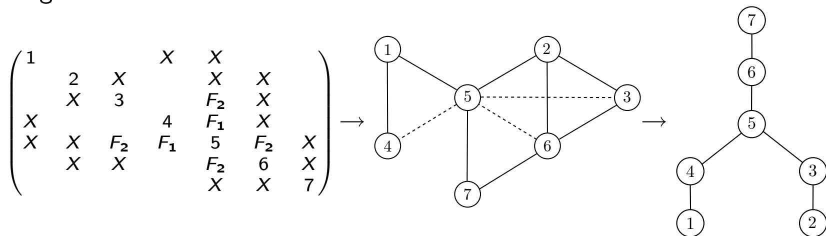
$$\begin{cases} y = L^{-1}e_j \\ a^{-1}_{i,j} = (U^{-1}y)_i \end{cases}$$

Computing a set of entries in A^{-1}

- The approach used relies on:
  - a traditional solution phase: $a^{-1}_{i,j} = (A^{-1}e_j)_i$
  - the use of a direct solver: once one has factorized A (e.g. $A = L\,U$), $a^{-1}_{i,j}$ can be obtained by:

$$\begin{cases} y = L^{-1}e_j \\ a^{-1}_{i,j} = (U^{-1}y)_i \end{cases}$$

- The computation can be significantly improved using a theorem by Gilbert and Liu (see next slide) which takes advantage of the sparsity of the right-hand sides and the elimination tree of A:

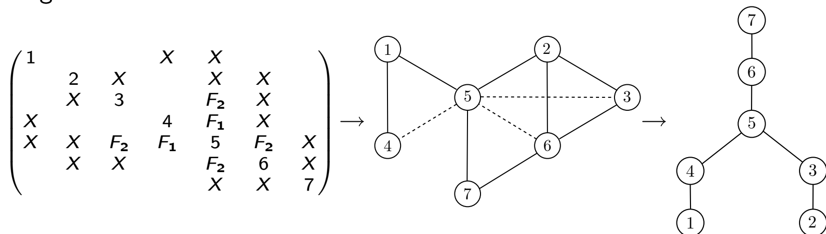


Computing a set of entries in A^{-1}

- The approach used relies on:
  - a traditional solution phase: $a^{-1}_{i,j} = (A^{-1}e_j)_i$
  - the use of a direct solver: once one has factorized A (e.g. $A = L\,U$), $a^{-1}_{i,j}$ can be obtained by:

$$\begin{cases} y = L^{-1}e_j \\ a^{-1}_{i,j} = (U^{-1}y)_i \end{cases}$$

- The computation can be significantly improved using a theorem by Gilbert and Liu (see next slide) which takes advantage of the sparsity of the right-hand sides and the elimination tree of A:



- This approach has been implemented in MUMPS during Tz. Slavova's PhD.

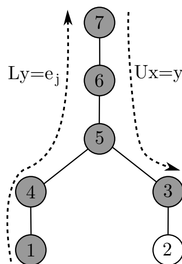
Computing a set of entries of A^{-1}

The solution used relies on a fundamental result derived from a theorem by Gilbert-Liu, which shows which nodes of the elimination tree of A have to be accessed to compute a particular entry:

Theorem

To compute a particular entry $a_{i,j}^{-1}$ in A^{-1}, the only factors which have to be loaded are the L factors on the path from node j up to the root node, and the U factors going back from the root to node i.

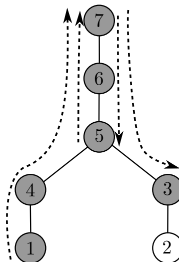
Example: traversal of the tree for the computation of $a_{3,1}^{-1}$



Computing a set of entries of A^{-1}

- It is interesting to compute several entries at the same time and not independently because nodes in common are loaded only once.

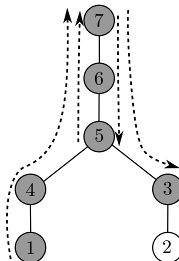
Example: when computing $a^{-1}_{3,1}$ and $a^{-1}_{5,5}$, accesses to 5,6,7 are spared



Computing a set of entries of A^{-1}

- It is interesting to compute several entries at the same time and not independently because nodes in common are loaded only once.

Example: when computing $a_{3,1}^{-1}$ and $a_{5,5}^{-1}$, accesses to 5,6,7 are spared

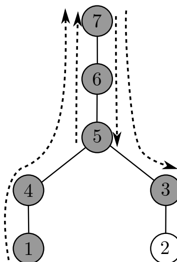


- This is critical in an out-of-core context: an access to a node = an access to the hard disk.

Computing a set of entries of A^{-1}

- It is interesting to compute several entries at the same time and not independently because nodes in common are loaded only once.

Example: when computing $a^{-1}_{3,1}$ and $a^{-1}_{5,5}$, accesses to 5,6,7 are spared



- This is critical in an out-of-core context: an access to a node = an access to the hard disk.
- In pratical applications: when one wants to compute a large number of entries of the inverse, the set of associated right-hand sides is divided into several blocks. $\Rightarrow$ is there a way to form the blocks such that the number of accesses is minimized ?

Preliminary experiments

How do topological orders (e.g. post-order) behave ? The ratio number of accesses over the lower bound is measured:

Matrix	10% diagonal	10% off-diag
CESR(46799)	1.01	1.28
af2356	1.02	2.09
boyd1	1.03	1.92
ecl32	1.01	2.31
gre1107	1.17	1.89
saylr4	1.06	1.92
sherman3	1.04	2.51
grund/bayer07	1.05	1.96
mathworks/pd	1.09	2.10
stokes64	1.05	2.35

$\Rightarrow$ topological orders provide good results for the diagonal case, but are not efficient enough for the general case; hence the need for a detailed study.

Possible approaches

Several approaches have been suggested:

- Transformation of the problem into a hypergraph partitioning problem (first for the diagonal case, then extended to the general case).

Possible approaches

Several approaches have been suggested:

- Transformation of the problem into a hypergraph partitioning problem (first for the diagonal case, then extended to the general case).
- An approach relying on a succession of local improvements, starting from an initial guess.

Possible approaches

Several approaches have been suggested:

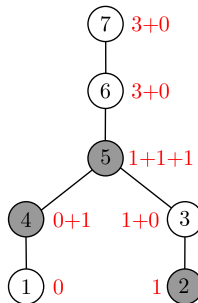
- Transformation of the problem into a hypergraph partitioning problem (first for the diagonal case, then extended to the general case).
- An approach relying on a succession of local improvements, starting from an initial guess.
- A constructive approach: what is the shape of an optimal solution ? How to build it ?

Lower-bound of the number of accesses

First, we need an estimation of the minimum number of accesses:

- It requires the notion of number of requests of a node:

$$\text{nb_requests}(i) = \sum_{j \in children(i)} \text{nb_requests}(j) + \text{req}(i)$$

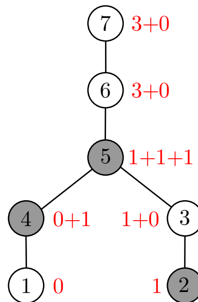


Lower-bound of the number of accesses

First, we need an estimation of the minimum number of accesses:

- It requires the notion of number of requests of a node:

$$\text{nb_requests}(i) = \sum_{j \in children(i)} \text{nb_requests}(j) + \text{req}(i)$$



- Then, the lower bound is given by:

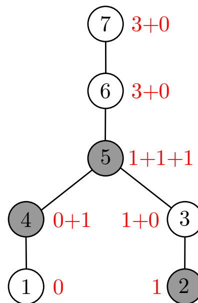
$$\sum_{all\ nodes} \text{size}(node) \left\lceil \frac{\text{nb_requests}(node)}{\text{RHS_block_size}} \right\rceil$$

Lower-bound of the number of accesses

First, we need an estimation of the minimum number of accesses:

- It requires the notion of number of requests of a node:

$$\text{nb_requests}(i) = \sum_{j \in children(i)} \text{nb_requests}(j) + \text{req}(i)$$



- Then, the lower bound is given by:

$$\sum_{all\ nodes} \text{size}(node) \left\lceil \frac{\text{nb_requests}(node)}{\text{RHS_block_size}} \right\rceil$$

- This lower bound is not always reachable, but is a good objective.

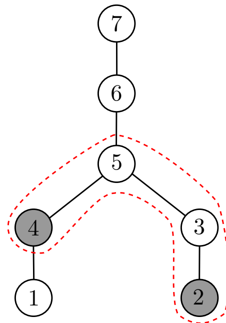
Shape of an optimal solution

During this study, a necessary and sufficient condition was proven. Here we provide only a (weaker) sufficient condition.
We need the notion of encompassing tree of a block of requested entries:

$$T_S^e = \bigcup_{s_i \in S} P(s_i, f_S),$$

where f_S is the lowest common ancestor of the nodes in S.

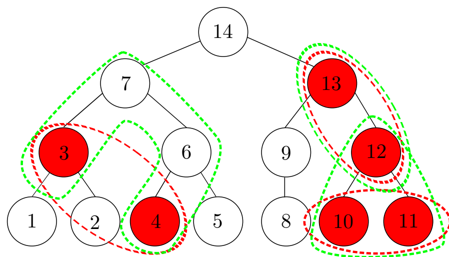
Example: with $a_{4,4}^{-1}$ and $a_{2,2}^{-1}$, the encompassing tree is $\{5, 4, 3, 2\}$.



Shape of an optimal solution

Sufficient condition for reaching the lower-bound: the encompassing trees of the blocks of requested entries do not intersect, or intersect only in one node.

Example:

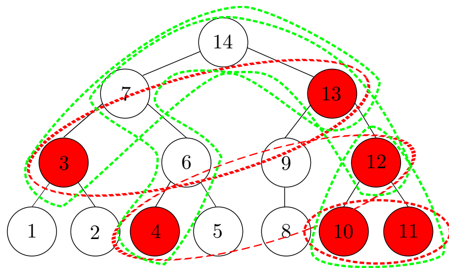


This partitioning reaches the lower bound.

Shape of an optimal solution

Sufficient condition for reaching the lower-bound: the encompassing trees of the blocks of requested entries do not intersect, or intersect only in one node.

Example:

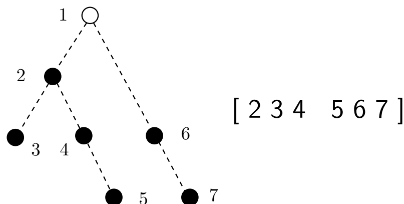


With $4 \leftrightarrow 13$, it would not be the case anymore.

Constructive heuristics

Now that we know what a good solution looks like, we try to build it. We propose the following heuristics:

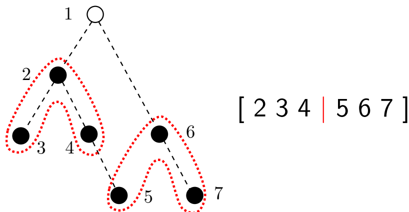
- Collect the requested nodes following a depth-first traversal of the tree (which is likely to gather nodes which are close).



Constructive heuristics

Now that we know what a good solution looks like, we try to build it. We propose the following heuristics:

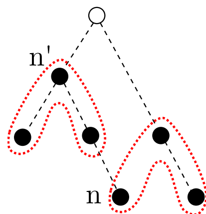
- Collect the requested nodes following a depth-first traversal of the tree (which is likely to gather nodes which are close).



Constructive heuristics

Now that we know what a good solution looks like, we try to build it. We propose the following heuristics:

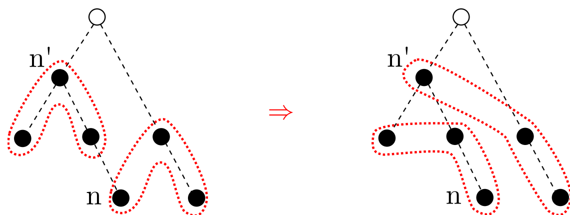
- Collect the requested nodes following a depth-first traversal of the tree (which is likely to gather nodes which are close).
- Apply a preventive idea: try to gather a node and its ancestors (or nodes close to its ancestors) in the previous block.
  Let n' be the highest ancestor of n in the previous block.
  Case I: there is no node in $S \backslash \text{Subtree}(n')$.



Constructive heuristics

Now that we know what a good solution looks like, we try to build it. We propose the following heuristics:

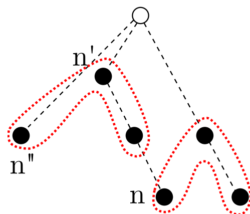
- Collect the requested nodes following a depth-first traversal of the tree (which is likely to gather nodes which are close).
- Apply a preventive idea: try to gather a node and its ancestors (or nodes close to its ancestors) in the previous block.
  Let n' be the highest ancestor of n in the previous block.
  Case I: there is no node in $S \backslash \mathrm{Subtree}(n')$. $\Rightarrow$ exchange n and n'.



Constructive heuristics

Now that we know what a good solution looks like, we try to build it. We propose the following heuristics:

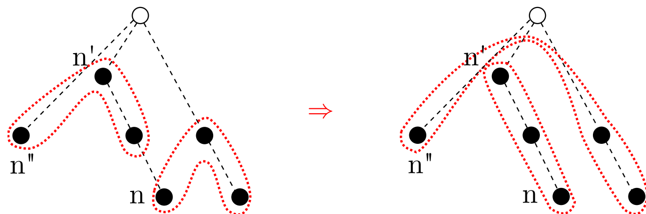
- Collect the requested nodes following a depth-first traversal of the tree (which is likely to gather nodes which are close).
- Apply a preventive idea: try to gather a node and its ancestors (or nodes close to its ancestors) in the previous block.
  Let n' be the highest ancestor of n in the previous block.
  Case II: n'' is the highest node in $S \setminus \text{Subtree}(n')$.



Constructive heuristics

Now that we know what a good solution looks like, we try to build it. We propose the following heuristics:

- Collect the requested nodes following a depth-first traversal of the tree (which is likely to gather nodes which are close).
- Apply a preventive idea: try to gather a node and its ancestors (or nodes close to its ancestors) in the previous block.
  Let n' be the highest ancestor of n in the previous block.
  Case II: n'' is the highest node in $S \backslash \text{Subtree}(n')$. $\Rightarrow$exchange n and n''.



Experiments

Computation of randomly chosen 10% of the diagonal:

Matrix	Constructive heuristics
CESR(46799)	1.01
af2356	1.01
boyd1	1.03
ecl32	1.01
gre1107	1.09
saylr4	1.08
sherman3	1.05
grund/bayer07	1.07
mathworks/pd	1.06
stokes64	1.04

Roughly the same results than a post-order. Is it possible to extend it to the non-diagonal case ?

Take the cost of the heuristic construction into account!

Conclusion

Conclusions

- This combinatorial problem is interesting and significant gains can be expected.
- Several approaches can be considered.
- The method presented here already shows promising results.

Perspectives

Several extensions and improvements can be studied:

- In-core case.
- Multiple entries per RHS.
- Parallel environment.
- Non-diagonal case for the heuristics.
- Compressed representations of the problem.
- ...

Conclusion

Thank you for your attention !

Any questions ?