Revisiting sparse matrix coloring and bicoloring

Alexis Montoison, Guillaume Dalle

November, 17^{th} 2025
JuMP-dev
Auckland

Introduction

- Automatic Differentiation (AD) is at the core of modern scientific computing and nonlinear optimization.
- Solvers such as Ipopt, Knitro, Uno, and MadNLP require Jacobians and Lagrangian Hessians at every iteration in order to compute search directions.
- These derivative matrices are large but sparse, and exploiting this sparsity is key to efficient AD and linear algebra.
- Coloring and bicoloring provides an elegant way to reduce the number of AD passes needed to recover sparse Jacobians and Hessians.

Jacobian computation via automatic differentiation

- Consider $c : \mathbb{R}^n \to \mathbb{R}^m$ with Jacobian $J_c(x) = \partial c(x)$.
- Forward-mode AD computes Jacobian-vector products: $u \mapsto J_c(x)u$.
- Reverse-mode AD computes vector-Jacobian products: $v \mapsto v^\top J_c(x)$.
- The full Jacobian can be reconstructed column-wise or row-wise.

- If columns have disjoint non-zeros, they can be recovered together.

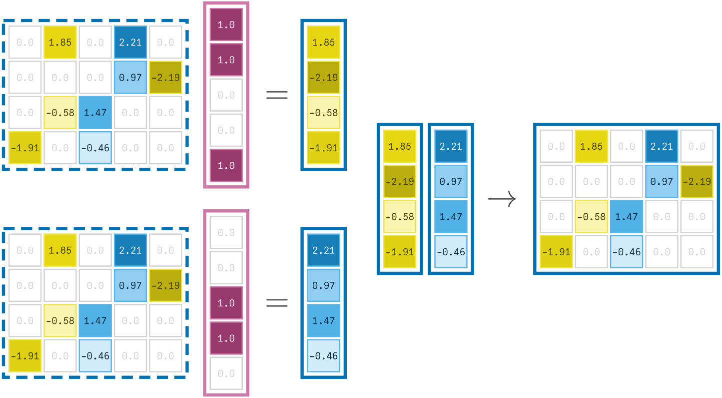


Figure – Materializing a Jacobian with forward-mode AD : (left) compressed evaluation of orthogonal columns (right) decompression to Jacobian matrix.

- If rows have disjoint non-zeros, they can be recovered together.

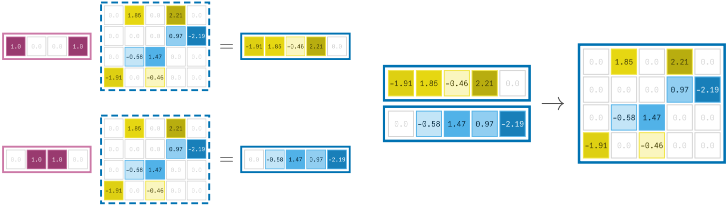
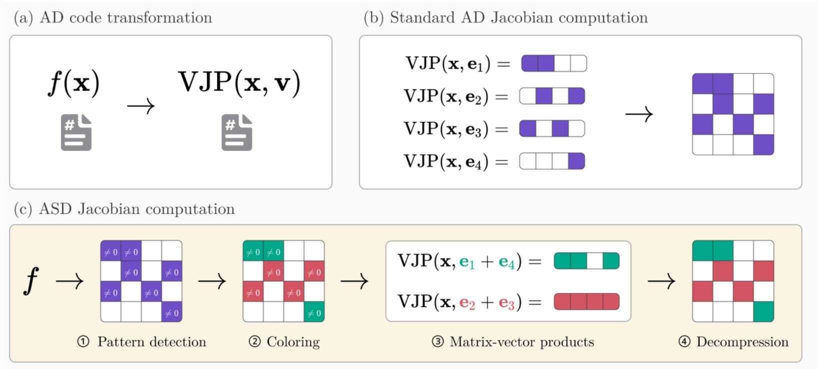


Figure – Materializing a Jacobian with reverse-mode AD : (left) compressed evaluation of orthogonal rows (right) decompression to Jacobian matrix.

Automatic sparse differentiation



Graph coloring

- This grouping problem can be reformulated as graph coloring.
- The goal is to minimize the number of AD evaluations.
- Graph coloring determines independent sets of columns (or rows).
- Complexity scales with the number of colors instead of n or m.

Graph coloring

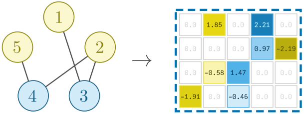


Figure – Optimal graph coloring.

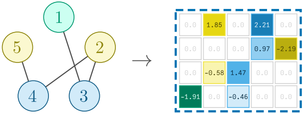


Figure – Suboptimal graph coloring (vertex 1 could be colored in yellow).

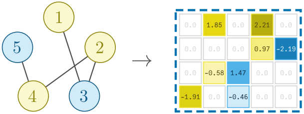


Figure – Infeasible graph coloring (vertices 2 and 4 are adjacent on the graph, but share a color)

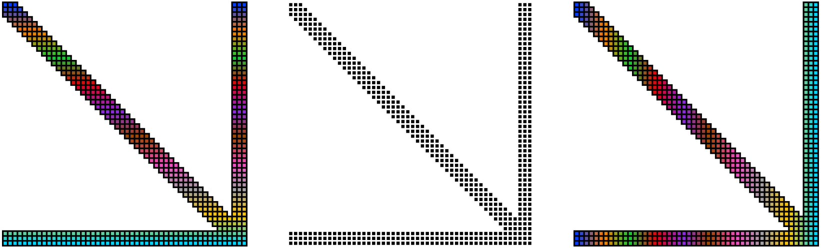
.

Hessian computation via automatic differentiation

- Consider $f : \mathbb{R}^n \to \mathbb{R}$ with Hessian $H_f(x) = \partial \nabla f(x)$.
- Forward-over-reverse mode AD computes Hessian-vector products: $w \mapsto H_f(x) w$.
- We can exploit symmetry to recover non-zeros from the lower or upper triangle.
- Star coloring and acyclic coloring are the most common symmetric colorings.

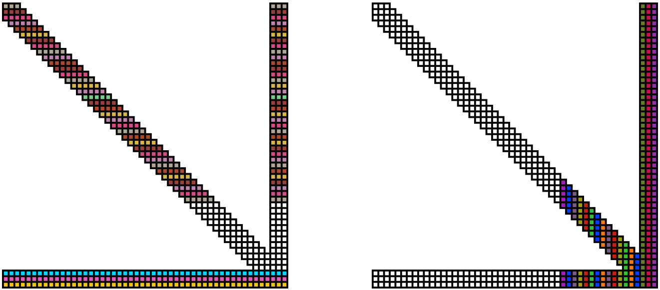
Bicoloring

- Some Jacobians contain dense substructures.
- Standard unidirectional coloring can become inefficient.
- Bicoloring jointly colors rows and columns.
- This combines forward- and reverse-mode AD for better performance.



- Row (left) and column (right) coloring of an arrowhead matrix (center), both requiring the same number of colors as the matrix dimension (50 in this case).

Unidirectional coloring versus bicoloring



- Bicoloring of an arrowhead matrix, requiring 10 colors for the rows (left) and 10 colors for the columns (right).

Coloring and bicoloring on a rectangle matrix

Figure – Row coloring (left) and column coloring (right) of a rectangle matrix, requiring the same number of colors as the matrix dimensions (respectively 6 and 12 in this case).

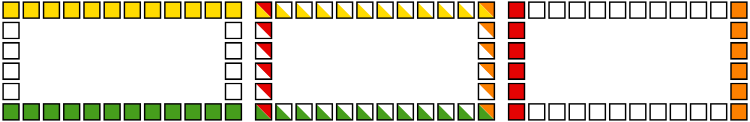


Figure – Bicoloring of a rectangle matrix, requiring only 2 colors for the rows (left) and 2 colors for the columns (right). In the central figure, each nonzero coefficient is colored using its row's color and its column's color.

Bicoloring in nonlinear least-squares problems

- Gauss-Newton subproblem : $\min_{d \in \mathbb{R}^n} \|J(x_k)d + F(x_k)\|^2$ with $J \in \mathbb{R}^{m \times n}$.

- Column coloring + forward-mode AD is efficient when $m \gg n$.

- Dense rows (e.g., normalization constraints) make column coloring inefficient : entire row must be recovered.

- Row coloring + reverse-mode AD is inefficient if $m \gg n$ (many row colors).

- **Bicoloring** : recover sparse columns with forward-mode, few dense rows with reverse-mode $\rightarrow$ improved performance.

Bicoloring in equality-constrained optimization

- Consider $\min_{x \in \mathbb{R}^n} f(x) \quad \text{subject to} \quad c(x) = 0$
- $f : \mathbb{R}^n \to \mathbb{R}$, $c : \mathbb{R}^n \to \mathbb{R}^m$, and the Jacobian $J_c(x) \in \mathbb{R}^{m \times n}$
- Row coloring + reverse-mode AD is efficient when $n \gg m$.
- Dense columns (variables affecting many constraints) make row coloring inefficient.
- Column coloring + forward-mode AD is an alternative but may be suboptimal.
- **Bicoloring** : recover sparse rows with reverse-mode, dense columns with forward-mode $\to$ better overall efficiency.

How to perform a bidirectional coloring ?

- Bicoloring and symmetric coloring share similarities.
- Bicoloring : Recover coefficients from rows or columns.
- Symmetric coloring : Recover coefficients from upper or lower triangle.
- Can we use star and acyclic symmetric colorings for bicoloring?

How to perform a bidirectional coloring ?

- Bicoloring on a Jacobian J can be seen as a symmetric coloring on $H = \begin{bmatrix} 0 & J^T \\ J & 0 \end{bmatrix}$.
- We can easily derive both **direct** (star) and **substitution** (acyclic) bicoloring.

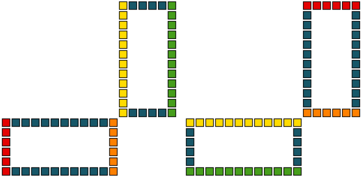


Figure – Symmetric coloring on H. Nonzeros are colored by the color of their columns on the left panel and by the color of their rows on the right panel.

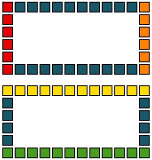


Figure – Bicoloring on J. Nonzeros are colored according to their column colors in the top panel and according to their row colors in the bottom panel.

Relation between neutral color and two-colored structures

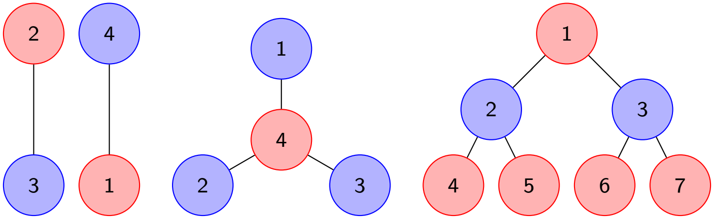


Figure – Variants of two-colored structures with trivial stars and trees (left), normal star (center) and normal tree (right).

- Diagonal entries take the color of their column, but are always zero under bicoloring.
- Normal trees require both colors for decompression.
- In normal stars, spoke colors are irrelevant for decompression.
- For trivial structures, the decompression color may be chosen arbitrarily from either vertex.

Content of SparseMatrixColorings.jl

SparseMatrixColorings.jl is a registered Julia package dedicated to coloring sparse Jacobians and Hessians.

```
pkg> add SparseMatrixColorings
julia> using SparseMatrixColorings
```

SparseMatrixColorings.jl implements algorithms from our research and the following articles :

- *What Color Is Your Jacobian? Graph Coloring for Computing Derivatives*, Gebremedhin et al. (2005)
- *New Acyclic and Star Coloring Algorithms with Application to Computing Hessians*, Gebremedhin et al. (2007)
- *Efficient Computation of Sparse Hessians Using Coloring and Automatic Differentiation*, Gebremedhin et al. (2009)
- *ColPack : Software for graph coloring and related problems in scientific computing*, Gebremedhin et al. (2013)
- *Revisiting sparse matrix coloring and bicoloring*, Montoison et al. (2025)

Content of SparseMatrixColorings.jl

The three main functions to perform a coloring are `coloring`, `ColoringProblem` and `GreedyColoringAlgorithm`.

```
using SparseMatrixColorings, SparseArrays

S = sparse([
  1 1 1 1 1 1 1 1 1 1
  1 0 0 0 0 0 0 0 0 1
  1 0 0 0 0 0 0 0 0 1
  1 0 0 0 0 0 0 0 0 1
  1 1 1 1 1 1 1 1 1 1
])

problem = ColoringProblem(; structure=:nonsymmetric,
                            partition=:bidirectional)

order = RandomOrder()

algo = GreedyColoringAlgorithm(order;
                               decompression=:direct,
                               postprocessing=true)

result = coloring(S, problem, algo)
```

Based on the result of `coloring`, you can easily recover a vector of integer colors with `row_colors`, `column_colors`, as well as the groups of colors with `row_groups` and `column_groups`.

```
julia> column_colors(result)
 1
 0
 0
 0
 0
 0
 0
 0
 0
 2
```

```
julia> column_groups(result)
 [1]
 [10]
```

Content of SparseMatrixColorings.jl

```
julia> row_colors(result)
 2
 0
 0
 0
 1
```

```
julia> row_groups(result)
 [5]
 [1]
```

```
julia> ncolors(result)
4
```

Content of SparseMatrixColorings.jl

The functions `compress` and `decompress` efficiently store and retrieve compressed representations of colorings for sparse matrices.

```
A = sparse([
  1  2  3  4  5  6  7  8  9  10
  11 0  0  0  0  0  0  0  0  14
  12 0  0  0  0  0  0  0  0  15
  13 0  0  0  0  0  0  0  0  16
  17 18 19 20 21 22 23 24 25 26
])
```

```
Br, Bc = compress(A, result)
2×10 Matrix{Int64}:
 17  18  19  20  21  22  23  24  25  26
  1   2   3   4   5   6   7   8   9  10

5×2 Matrix{Int64}:
  1  10
 11  14
 12  15
 13  16
 17  26
```

Content of SparseMatrixColorings.jl

```
julia> C = decompress(Br, Bc, result)
5×10 SparseMatrixCSC{Int64, Int64} with 26 stored entries:
  1   2   3   4   5   6   7   8   9  10
 11   ⋅   ⋅   ⋅   ⋅   ⋅   ⋅   ⋅   ⋅  14
 12   ⋅   ⋅   ⋅   ⋅   ⋅   ⋅   ⋅   ⋅  15
 13   ⋅   ⋅   ⋅   ⋅   ⋅   ⋅   ⋅   ⋅  16
 17  18  19  20  21  22  23  24  25  26
```

```
julia> decompress!(A, 2*Br, 3*Bc, result)
5×10 SparseMatrixCSC{Int64, Int64} with 26 stored entries:
   6   8  12  16  20  24  28  32   36   60
  66   ⋅   ⋅   ⋅   ⋅   ⋅   ⋅   ⋅    ⋅   84
  72   ⋅   ⋅   ⋅   ⋅   ⋅   ⋅   ⋅    ⋅   90
  78   ⋅   ⋅   ⋅   ⋅   ⋅   ⋅   ⋅    ⋅   96
 102  72  76  80  84  88  92  96  100  156
```

What actually matters for JuMP and MOI ?

- Jacobian coloring is unnecessary : expression trees already enable very efficient reverse-mode passes.
- But recent work on bicoloring introduces the idea of neutral colors in symmetric colorings and post-processing.
- These neutral colors become directly useful in MOI if we stop assuming a fully nonzero Hessian diagonal.

Ordering strategies matter

- MOI only supports the natural ordering of vertices.
- Many vertex orderings exist (random, largest first, <u>smallest last</u>, incidence degree, dynamic largest first, ...) and produce different colorings.
- We can precompute multiple colorings with different orderings as a preprocessing phase.
- Perfect elimination ordering is optimal for acyclic coloring on banded matrices or matrices with chordal-like sparsity.

Acyclic vs. star coloring

- MOI currently relies on acyclic coloring.
- Only the colors are kept; tree structures are discarded, limiting efficient preparation for decompression.
- Star coloring is cheaper to compute but yields more colors. We can alternate decompression and directional derivatives without storing all compressed Hessian columns.
- No need for `DataStructures.IntDisjointSets`: the forest structure in `SparseMatrixColorings.jl` already captures everything, and could replace the `DataStructures.jl` dependency in MOI.

- Neutral colors can be used in symmetric coloring for generic Hessian AD.

- Multiple-coloring preprocessing could improve robustness and reduce AD passes.

- Integration of SparseMatrixColorings.jl in MOI, potentially inside a new AD backend ?

Optimal coloring with JuMP / MOI

We implemented an optimal column / row coloring algorithm based on constraint programming in JuMP.

```julia
n = nb_vertices(bipartite_graph, Val(side))
model = Model(optimizer)

# one variable per vertex to color, removing some renumbering
    symmetries
@variable(model, 1 <= color[i=1:n] <= i, Int)

# one variable to count the number of distinct colors
@variable(model, ncolors, Int)
@constraint(model, [ncolors; color] in MOI.CountDistinct(n + 1))

# neighbors of the same vertex must have distinct colors
for i in vertices(bg, Val(other_side))
    neigh = neighbors(bg, Val(other_side), i)
    @constraint(model, color[neigh] in
        MOI.AllDifferent(length(neigh)))
end

# minimize the number of distinct colors
@objective(model, Min, ncolors)
optimize!(model)
```

Optimal coloring with JuMP / MOI

Still need to add a JuMP formulation for symmetric colorings.

```
using SparseMatrixColorings, JuMP, MathOptInterface, MiniZinc

coloring_problem = ColoringProblem(;
    structure=:nonsymmetric, partition=:column)

algo = OptimalColoringAlgorithm(
() -> MiniZinc.Optimizer{Float64}("highs");
    silent=false, assert_solved=false)

coloring(J, coloring_problem, algo)
num_colors = ncolors(result)

import ORTools_jll
path_cp_sat = joinpath(ORTools_jll.artifact_dir, "share",
    "minizinc", "solvers", "cp-sat.msc")

algo = OptimalColoringAlgorithm(()-> MiniZinc.Optimizer{Float64}
    (path_cp_sat); silent=false, assert_solved=false)

coloring(J, coloring_problem, algo)
num_colors = ncolors(result)
```

Downloading SparseMatrixColorings.jl

https://github.com/gdalle/SparseMatrixColorings.jl