Building a curated motif database from multiple sources

Benjamin Jean-Marie Tremblay*

25 May 2026

Abstract

In practice, a motif scan rarely relies on a single public database. This vignette walks through curating one consolidated motif set from several public sources, using *Homo sapiens* motifs pulled out of three `MotifDb` sub-collections (JASPAR2022, HOCOMOCOv11-core, CIS-BP) as the working example. We import, inspect cross-source redundancy with `compare_motifs_lite()`, deduplicate via graph-based clustering in `merge_similar_lite()`, trim low-information flanks with `trim_motifs()`, annotate the survivors with the existing metadata via `to_df()` / `update_motifs()`, and export the whole curated library as a single MEME file ready for downstream use.

Contents

1 Introduction **1**

2 Loading motifs from several sources **2**

2.1 Sidebar: importing raw files directly 3

3 Inspecting cross-source redundancy **3**

4 Deduplicating via graph clustering **4**

5 Visualising the cluster structure **7**

6 Trimming uninformative flanks **8**

7 Adding the metadata back **9**

8 Export as a single MEME file **10**

9 Session info **11**

10 References **13**

1 Introduction

No single public motif database covers every transcription factor in every organism, and no single source is perfectly consistent: each ships its own naming conventions, its own matrix-type conventions (PCM, PPM, or PWM), and its own intra-source redundancies (several matrices for the same factor, often with slight quality differences). Anyone doing applied motif scanning sooner or later ends up pulling from several sources and reconciling them more or less by hand. This vignette gathers the building blocks that `universalmotif` provides for that reconciliation into a single end-to-end recipe, demonstrated here on a few hundred *Homo sapiens* motifs drawn from three `MotifDb` sub-collections.

*benjamin.tremblay@uwaterloo.ca

The previous vignettes cover the underlying processes in isolation: for single-motif import and export, see the motif manipulation vignette; for the mechanics of motif-vs-motif scoring and p-value calculation, see the comparisons and p-values vignette; for an end-to-end ChIP-seq analysis that would use a curated motif library, see the ChIP-seq workflow vignette. This vignette sits one step upstream of the ChIP-seq one: it produces the kind of consolidated MEME file that a real analyst would feed into `scan_sequences_lite()` or `enrich_motifs_lite()`.

The recipe proceeds in eight steps:

1. Pull motifs from three sources via `MotifDb`.
2. Mark the originating source onto each motif.
3. Inspect intra- and inter-source redundancy with `compare_motifs_lite()`.
4. Deduplicate via graph-based clustering with `merge_similar_lite()`.
5. Visualise the cluster structure with `motif_tree_lite()`.
6. Trim uninformative flanks with `trim_motifs()`.
7. Annotate the survivors with metadata via `to_df()` / `update_motifs()`.
8. Export the curated set as a single MEME file with `write_meme()`.

2 Loading motifs from several sources

`MotifDb` ships a large indexed collection that has already been parsed from a wide variety of source formats. For a single demonstration we narrow it to *Homo sapiens* and pull three sub-collections that, between them, cover a fair slice of human TFs: `jaspar2022`, `HOCOMOCOv11-core-A`, and `cisbp_1.02`.

```
suppressPackageStartupMessages(library(MotifDb))

hs <- suppressWarnings(query(MotifDb, "hsapiens"))
sources <- c("jaspar2022", "HOCOMOCOv11-core-A", "cisbp_1.02")
per.source <- suppressWarnings(lapply(sources, function(s)
  convert_motifs(query(hs, s))))
names(per.source) <- sources
sapply(per.source, length)
#>         jaspar2022 HOCOMOCOv11-core-A         cisbp_1.02
#>                692                181                313
```

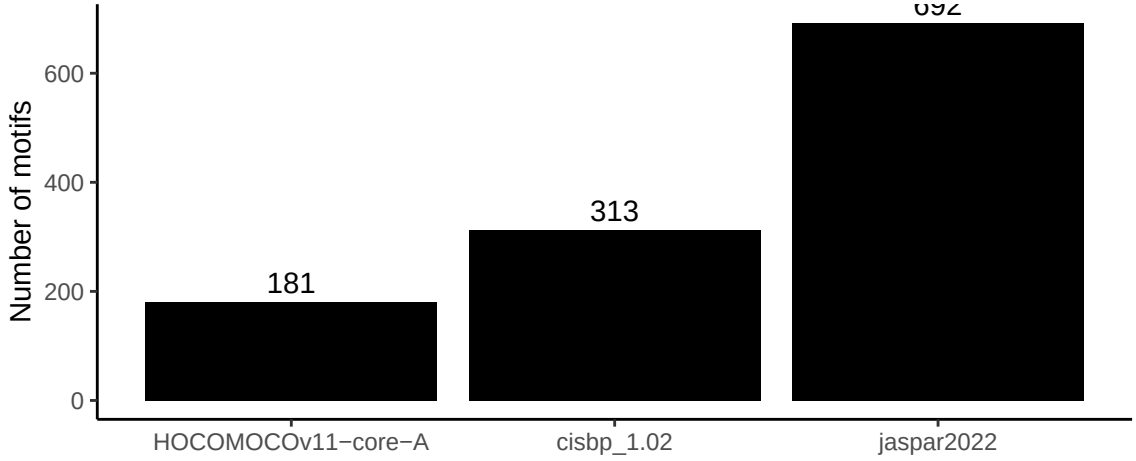
We annotate each motif's `family` slot with the originating source. Doing this here, before the lists are concatenated, means the source survives every subsequent operation (clustering, trimming, MEME export) and gives `motif_tree_lite()` a per-motif colouring column to use later. `family` is the natural slot for this because `MotifDb` rarely populates it, and because `motif_tree_lite(linecol = "family")` is the function's default.

```
for (s in sources) {
  for (i in seq_along(per.source[[s]]))
    per.source[[s]][[i]]["family"] <- s
}
all.motifs <- do.call(c, per.source)
length(all.motifs)
#> [1] 1186
```

As a quick visual sanity check, we can plot the per-source counts:

```
src.df <- data.frame(source = sources,
                     count  = sapply(per.source, length),
                     stringsAsFactors = FALSE)
ggplot(src.df, aes(x = .data$source, y = .data$count)) +
  geom_col(fill = "black") +
  geom_text(aes(label = .data$count), vjust = -0.4) +
```

```
  labs(x = NULL, y = "Number of motifs") +
  theme_bw() +
  theme(panel.grid         = element_blank(),
        panel.border       = element_blank(),
        axis.line.x.bottom = element_line(colour = "black"),
        axis.line.y.left   = element_line(colour = "black"))
```



2.1 Sidebar: importing raw files directly

The rest of this vignette assumes the motifs were sourced through `MotifDb`. If instead you have raw downloaded files sitting on disk, the import step is just a one-liner per format, and the rest of the workflow is identical. The chunk below is purely for illustration.

```
jaspar.motifs   <- read_jaspar("JASPAR2024_CORE_vertebrates.txt")
hocomoco.motifs <- read_meme("HOCOMOCOv11_core_HUMAN_mono_meme.txt")
cisbp.motifs    <- read_cisbp("CIS-BP_2.00_Homo_sapiens.txt")
transfac.motifs <- read_transfac("transfac_matrix.dat")
uniprobe.motifs <- read_uniprobe("uniprobe_results.txt")
all.motifs      <- c(jaspar.motifs, hocomoco.motifs, cisbp.motifs,
                     transfac.motifs, uniprobe.motifs)
```

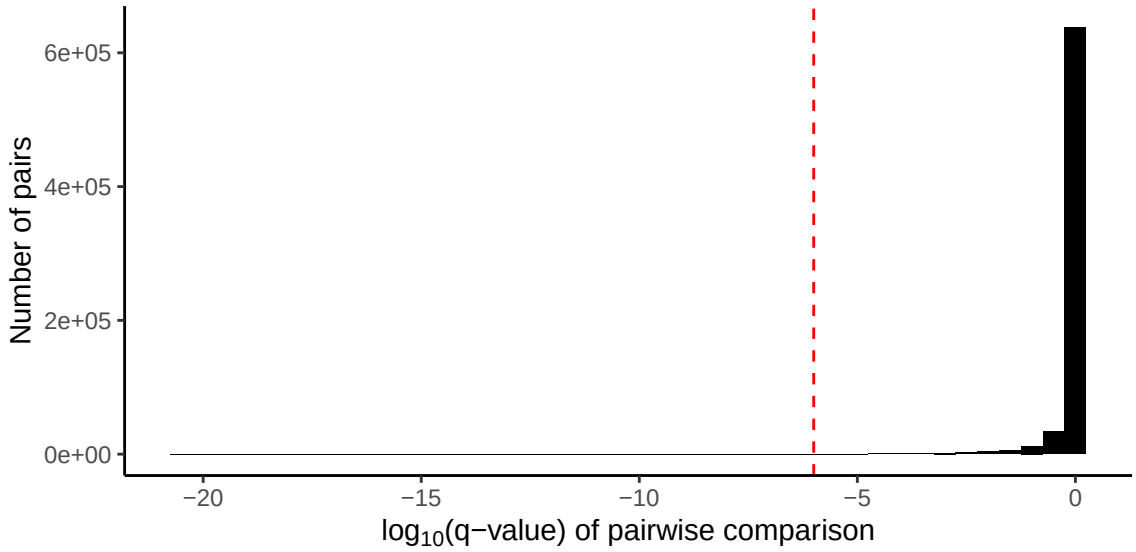
3 Inspecting cross-source redundancy

Before deduplicating, it is worth quantifying how much redundancy is actually in the combined set. `compare_motifs_lite()` in matrix mode returns a square q-value matrix (one q-value per ordered pair, after multiple-testing correction). With several hundred motifs the result is a 1,186-row matrix that holds every pairwise q-value. Two threads finish the comparison in about four seconds on my laptop.

```
qmat <- compare_motifs_lite(all.motifs,
                            matrix.out = "qvalue",
                            nthreads   = 2)
dim(qmat)
#> [1] 1186 1186
qvals <- qmat[upper.tri(qmat)]
length(qvals)
#> [1] 702705
summary(qvals)
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>   0.000   1.000   1.000   0.911   1.000   1.000
```

The pairwise q-value distribution is the visual cue for picking a sensible deduplication threshold. On a log10 axis it spans many orders of magnitude, with a long left tail of genuinely redundant pairs (within-TF or cross-source duplicates) and a dense pile of unrelated pairs heaped at q = 1.

```
log10q <- log10(pmax(qvals, .Machine$double.xmin))
ggplot(data.frame(log10q = log10q), aes(x = .data$log10q)) +
  geom_histogram(binwidth = 0.5, fill = "black", colour = NA) +
  geom_vline(xintercept = log10(1e-6), linetype = "dashed",
             colour = "red") +
  labs(x = expression(log[10] * "(q-value) of pairwise comparison"),
       y = "Number of pairs") +
  theme_bw() +
  theme(panel.grid         = element_blank(),
        panel.border       = element_blank(),
        axis.line.x.bottom = element_line(colour = "black"),
        axis.line.y.left   = element_line(colour = "black"))
```



The dashed line marks q = 1e-6, the threshold we will use for the deduplication in the next section. Picking that threshold is something of a judgement call. Set it too loose and graph transitivity starts collapsing unrelated motifs into giant components (with this dataset, a threshold of q = 1e-3 produces a single large cluster of hundreds of motifs, swallowing much of the zinc-finger and homeodomain families, because their pairwise scores share just enough small q-values to form one connected component). Set it too tight and very few motifs are merged at all. For our purposes here, q = 1e-6 seems to strike a reasonable balance, collapsing a good deal of redundancy without bridging unrelated families.

```
mean(qvals < 1e-6)
#> [1] 0.001159804
sum(qvals < 1e-6)
#> [1] 815
```

4 Deduplicating via graph clustering

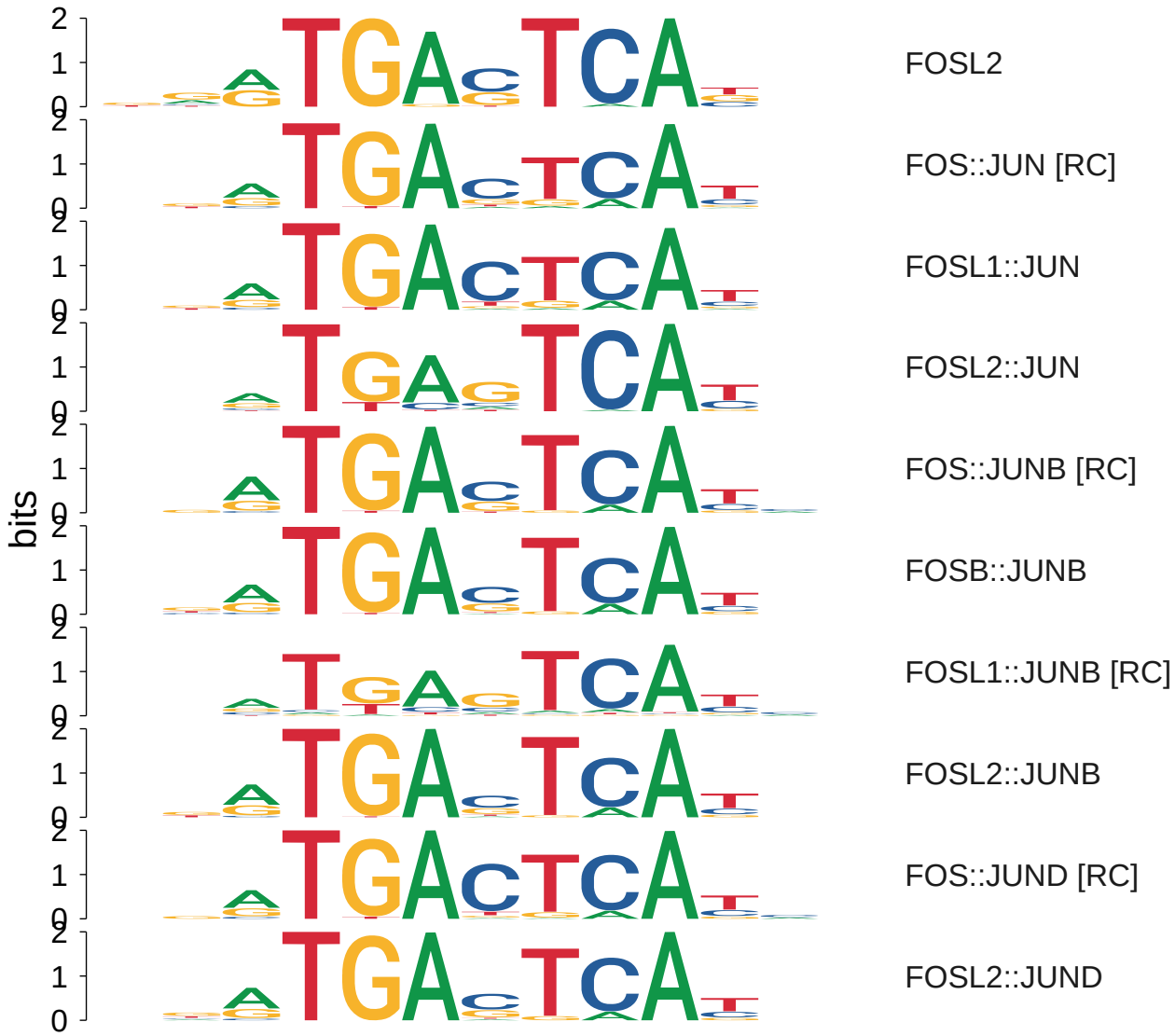
`merge_similar_lite()` builds a graph whose edges link any two motifs that pass the q-value threshold, finds the connected components, and returns one representative motif per cluster. We use it twice: once with `return.clusters = TRUE` to get the per-motif cluster assignments (so we can confirm them visually), and once without, to get the final merged set of motifs.

```
clusters <- merge_similar_lite(all.motifs,
                               qvalue          = 1e-6,
                               return.clusters = TRUE,
                               nthreads        = 2)
```

To judge whether the q-value ought to be tightened further, we can take a look at the largest cluster:

```
big.cluster <- names(sort(table(clusters$cluster),
  decreasing = TRUE)[1])
big.cluster.motifs <- all.motifs[
  clusters$motif.i[clusters$cluster == big.cluster]]
if (length(big.cluster.motifs) > 10)
  big.cluster.motifs <- big.cluster.motifs[1:10]

view_motifs_lite(big.cluster.motifs,
  names.pos = "right", show.positions = FALSE)
```



It is well worth taking the time to go carefully back and forth between examining the clusters and adjusting the q-value threshold, since the final merge is really the most consequential decision in curating a motif

database. In our case we will carry on with this threshold.

```
merged <- merge_similar_lite(all.motifs,
                             qvalue   = 1e-6,
                             nthreads = 2)
length(all.motifs)
#> [1] 1186
length(merged)
#> [1] 796
```

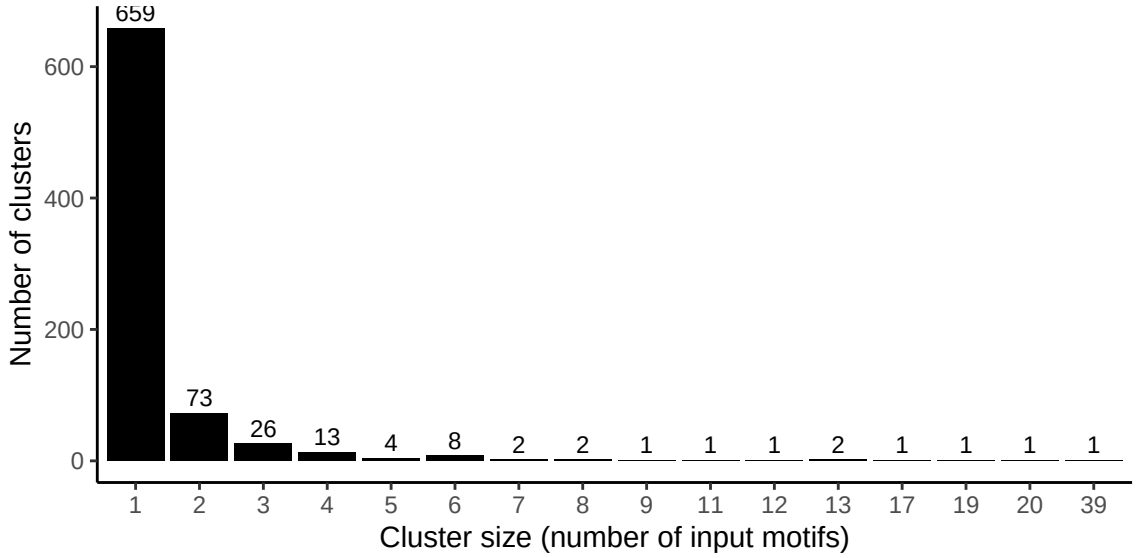
`merge_similar_lite()` autogenerates compound names for the merged motifs by joining every contributing name with `+`, which again becomes unwieldy once several motifs collapse into a single cluster. A more compact alternative is to keep the first one or two contributing names and roll the rest up into a count suffix, so that the lineage stays visible without overflowing either the table or the MEME header. Singletons simply keep their original name:

```
contrib.names <- tapply(clusters$name, clusters$cluster,
                        function(n) as.character(n))
for (i in seq_along(merged)) {
  nms <- contrib.names[[i]]
  if (length(nms) == 2L)
    merged[[i]]["name"] <- paste(nms, collapse = "+")
  else if (length(nms) >= 3L)
    merged[[i]]["name"] <- sprintf("%s+%s+%dmore",
                                   nms[1], nms[2], length(nms) - 2L)
}
## `clusters` is a universalmotif_df: one row per input motif, with
## the motif objects in the `motif` column plus the `motif.i` and
## `cluster` bookkeeping columns. Show just the assignment columns:
print(head(as.data.frame(clusters)[, c("name", "motif.i", "cluster")], 8),
      row.names = FALSE)
#>      name motif.i cluster
#>     FOXF2       1       1
#>     FOXD1       2       2
#>      IRF2       3       3
#>  MAX::MYC       4       4
#>     PPARG       5       5
#>      PAX6       6       6
#>      PBX1       7       7
#>      RORA       8       8
```

The reduction is substantial, with roughly a third of the input motifs collapsing into shared clusters. The distribution of cluster sizes shows the long tail of singletons (motifs unique enough to stay on their own), together with a moderate body of size-2 and size-3 clusters (the common within-TF and cross-source redundancies):

```
cluster.sizes <- table(clusters$cluster)
size.dist     <- as.data.frame(table(cluster.sizes),
                               stringsAsFactors = FALSE)
names(size.dist) <- c("cluster.size", "n.clusters")
size.dist$cluster.size <- as.integer(size.dist$cluster.size)
ggplot(size.dist, aes(x = factor(.data$cluster.size),
                      y = .data$n.clusters)) +
  geom_col(fill = "black") +
  geom_text(aes(label = .data$n.clusters), vjust = -0.4, size = 3) +
  labs(x = "Cluster size (number of input motifs)",
```

```
       y = "Number of clusters") +
  theme_bw() +
  theme(panel.grid         = element_blank(),
        panel.border       = element_blank(),
        axis.line.x.bottom = element_line(colour = "black"),
        axis.line.y.left   = element_line(colour = "black"))
```



A cluster drawing on two or three different sources is direct evidence of cross-database redundancy, which is precisely the kind of duplication that a single-source library would quietly miss. We can tally the sources per cluster:

```
clusters$source <- vapply(all.motifs[clusters$motif.i],
                          function(m) m["family"], character(1))
sources.per.cluster <- tapply(clusters$source, clusters$cluster,
                              function(s) length(unique(s)))
table(sources.per.cluster)
#> sources.per.cluster
#>   1   2   3
#> 734  57   5
```

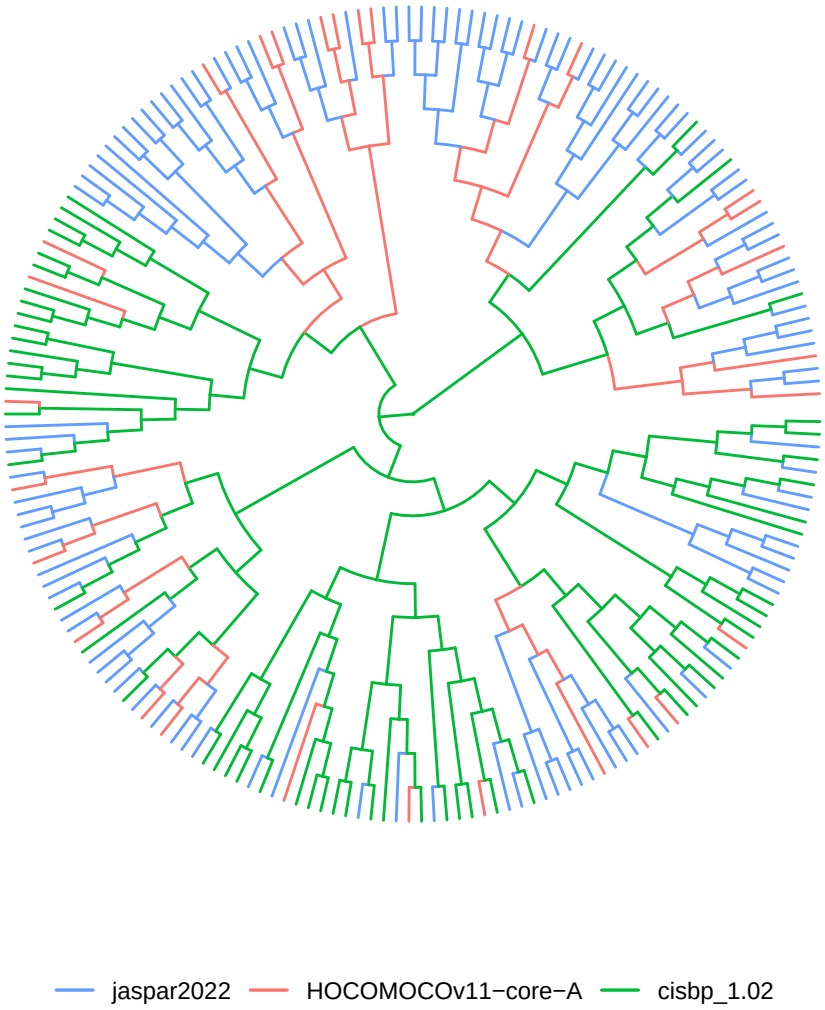
Many clusters contain motifs from more than one source; those multi-source clusters are exactly the cross-database redundancies that this whole workflow exists to collapse.

5 Visualising the cluster structure

`motif_tree_lite()` builds a hierarchical tree from the pairwise comparison and renders it via `ggtree`. On all 1,186 motifs the labels would be too crowded to read, so we plot a random 200-motif subsample instead, coloured by source. The clades typically mix all three sources, which mirrors the multi-source cluster finding from above.

```
set.seed(2026)
sub.idx <- sort(sample(length(all.motifs), 200))
sub     <- all.motifs[sub.idx]
clean.names <- make.unique(gsub("[^A-Za-z0-9_]", "_",
                                vapply(sub, function(m) m["name"],
                                       character(1))))
```

```
for (i in seq_along(sub)) sub[[i]]["name"] <- clean.names[i]
motif_tree_lite(sub, linecol = "family", labels = "none",
            layout = "circular", tiplogo = FALSE, progress = FALSE) +
  ggplot2::scale_color_discrete(breaks = sources) +
  ggplot2::theme(legend.position = "bottom",
                 legend.title    = ggplot2::element_blank())
```



6 Trimming uninformative flanks

PWM-style downstream scanning is sensitive to long low-information flanks, which dilute scores and inflate match counts. `trim_motifs()` removes columns whose information content falls below `min.ic` (default 0.25 bits) from the edges. The default "both" mode trims from both ends.

For a clean demonstration we pick the merged motif that loses the most columns to trimming. Computing the trim delta directly (the width before, minus the width after) is rather more reliable than trying to guess from the flank IC, because `trim_motifs()` walks inward from the edges and stops at the first column above `min.ic`, which is not always the column the eye would have picked out as "low IC".

```
trim.delta <- vapply(merged, function(m)
  ncol(m["motif"]) - ncol(trim_motifs(m)["motif"]),
  integer(1))
trim.candidate.idx <- which.max(trim.delta)
```

```
trim.before <- merged[[trim.candidate.idx]]
trim.before["name"] <- paste0(trim.before["name"], "  (untrimmed)")
trim.after  <- trim_motifs(merged[[trim.candidate.idx]])
trim.after["name"] <- paste0(trim.after["name"], "  (trimmed)")
view_motifs_lite(list(trim.before, trim.after), names.pos = "right")
```

We then apply the trim to every merged motif before exporting:

```
curated <- trim_motifs(merged)
range(sapply(merged,  function(m) ncol(m["motif"])))
#> [1]  6 35
range(sapply(curated, function(m) ncol(m["motif"])))
#> [1]  5 33
```

The width range shrinks, as expected: the merge step often pads clusters out with low-confidence flanking columns wherever the input motifs disagreed, and `trim_motifs()` is the natural way to clean those back up.

7 Adding the metadata back

For a curated library to be genuinely useful, users will want to know, for each final motif, which sources contributed to it. The `universalmotif_df` interface (`to_df()` returns a data-frame view, and `update_motifs()` writes columns back into the motif list) is the natural way to add per-motif annotations in a tidy fashion.

We compute the contributing-source list per cluster, then attach it to the curated motifs together with a curation date:

```
src.by.cluster <- tapply(clusters$source, clusters$cluster,
                         function(s)
                           paste(sort(unique(s)), collapse = ";"))
n.by.cluster   <- tapply(clusters$source, clusters$cluster,
                         length)

curated.df <- to_df(curated, extrainfo = TRUE)
curated.df$dataSource    <- NULL  # MotifDb's single-source field;
                                  # superseded by the multi-source
                                  # db.sources column we are about
                                  # to add
curated.df$db.sources    <- as.character(src.by.cluster)
curated.df$n.contributors <- as.integer(n.by.cluster)
curated.df$curation.date <- "2026-05-25"
```

```
print(as.data.frame(head(curated.df[, c("name", "db.sources",
                                        "n.contributors",
                                        "curation.date")], 10)),
      row.names = FALSE)
#>       name db.sources n.contributors curation.date
#>      FOXF2 jaspar2022              1    2026-05-25
#>      FOXD1 jaspar2022              1    2026-05-25
#>       IRF2 jaspar2022              1    2026-05-25
#>   MAX::MYC jaspar2022              1    2026-05-25
#>      PPARG jaspar2022              1    2026-05-25
#>       PAX6 jaspar2022              1    2026-05-25
#>       PBX1 jaspar2022              1    2026-05-25
#>       RORA jaspar2022              1    2026-05-25
#>  RORA+RORC jaspar2022              2    2026-05-25
#>      RREB1 jaspar2022              1    2026-05-25
```

We round-trip back into motif objects with `update_motifs()`. The new columns land in the `extrainfo` slot of each motif, and are then preserved everywhere that `universalmotif` reads or writes its native representation.

```
curated <- to_list(update_motifs(curated.df), extrainfo = TRUE)
curated[[1]]["extrainfo"]
#>     db.sources n.contributors  curation.date
#>   "jaspar2022"            "1"   "2026-05-25"
```

The MEME format has no slot for arbitrary key-value metadata, so how much of the `extrainfo` survives is really limited by what the output format itself can carry. For analyses where that metadata is critical, my suggestion would be to write the curated library out as a `universalmotif_df` RDS file (which preserves everything) and treat the MEME export only as the format that consumers like FIMO and the MEME suite happen to need.

8 Export as a single MEME file

`write_meme()` writes the whole motif list out to a single file. The MEME header carries the `name`, `altname`, `nsites`, and `eval` of each motif, plus a global background.

```
out.file <- tempfile(fileext = ".meme")
write_meme(curated, file = out.file, overwrite = TRUE)
file.info(out.file)$size
#> [1] 416221
```

The first few lines of the file show the standard MEME preamble and the start of the first motif block:

```
cat(head(readLines(out.file), 25), sep = "\n")
#> MEME version 5
#>
#> ALPHABET= ACGT
#>
#> strands: + -
#>
#> Background letter frequencies
#> A 0.250000 C 0.250000 G 0.250000 T 0.250000
#>
#> MOTIF FOXF2 MA0030.1
#> letter-probability matrix: alength= 4 w= 11 nsites= 28 E= 0
#>  0.629630  0.148148  0.074074  0.148148
```

```
#>  0.464286  0.178571  0.178571  0.178571
#>  0.136364  0.500000  0.363636  0.000000
#>  0.259259  0.000000  0.740741  0.000000
#>  0.000000  0.000000  0.000000  1.000000
#>  1.000000  0.000000  0.000000  0.000000
#>  1.000000  0.000000  0.000000  0.000000
#>  1.000000  0.000000  0.000000  0.000000
#>  0.000000  0.925926  0.000000  0.074074
#>  1.000000  0.000000  0.000000  0.000000
#>  0.592593  0.148148  0.074074  0.185185
#>
#> MOTIF FOXD1 MA0031.1
#> letter-probability matrix: alength= 4 w= 7 nsites= 20 E= 0
```

We load it back with `read_meme()` to confirm that the file parses cleanly and that the length matches:

```
re.in <- read_meme(out.file)
length(re.in)
#> [1] 796
length(curated)
#> [1] 796
identical(sapply(re.in,   function(m) m["name"]),
          sapply(curated, function(m) m["name"]))
#> [1] TRUE
```

The names match, the motif matrices match, and the file is now ready for any downstream tool that reads MEME-format motif collections (FIMO, AME, CentriMo, or `read_meme()` itself for the next analysis).

9 Session info

```
sessionInfo()
#> R version 4.6.0 RC (2026-04-17 r89917)
#> Platform: x86_64-pc-linux-gnu
#> Running under: Ubuntu 24.04.4 LTS
#>
#> Matrix products: default
#> BLAS:   /home/biocbuild/bbs-3.24-bioc/R/lib/libRblas.so
#> LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.12.0  LAPACK version 3.12.0
#>
#> locale:
#>  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
#>  [3] LC_TIME=en_GB              LC_COLLATE=C
#>  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
#>  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
#>  [9] LC_ADDRESS=C               LC_TELEPHONE=C
#> [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
#>
#> time zone: America/New_York
#> tzcode source: system (glibc)
#>
#> attached base packages:
#> [1] stats4    stats     graphics  grDevices utils     datasets  methods
#> [8] base
```

```
#>
#> other attached packages:
#>  [1] dplyr_1.2.1
#>  [2] ggtree_4.3.0
#>  [3] ggplot2_4.0.3
#>  [4] MotifDb_1.55.0
#>  [5] BSgenome.Athaliana.TAIR.TAIR9_1.3.1000
#>  [6] BSgenome_1.81.0
#>  [7] BiocIO_1.23.3
#>  [8] rtracklayer_1.73.0
#>  [9] GenomeInfoDb_1.49.1
#> [10] GenomicRanges_1.65.0
#> [11] Biostrings_2.81.3
#> [12] Seqinfo_1.3.0
#> [13] XVector_0.53.0
#> [14] IRanges_2.47.2
#> [15] S4Vectors_0.51.3
#> [16] BiocGenerics_0.59.7
#> [17] generics_0.1.4
#> [18] universalmotif_1.31.42
#>
#> loaded via a namespace (and not attached):
#>  [1] ggiraph_0.9.6               tidyselect_1.2.1
#>  [3] farver_2.1.2                S7_0.2.2
#>  [5] bitops_1.0-9                fastmap_1.2.0
#>  [7] RCurl_1.98-1.19             lazyeval_0.2.3
#>  [9] fontquiver_0.2.1            GenomicAlignments_1.49.0
#> [11] XML_3.99-0.23               digest_0.6.39
#> [13] lifecycle_1.0.5             tidytree_0.4.7
#> [15] magrittr_2.0.5              compiler_4.6.0
#> [17] rlang_1.2.0                 tools_4.6.0
#> [19] yaml_2.3.12                 data.table_1.18.4
#> [21] knitr_1.51                  htmlwidgets_1.6.4
#> [23] S4Arrays_1.13.0             labeling_0.4.3
#> [25] curl_7.1.0                  splitstackshape_1.4.8.1
#> [27] DelayedArray_0.39.3         RColorBrewer_1.1-3
#> [29] aplot_0.2.9                 abind_1.4-8
#> [31] BiocParallel_1.47.0         withr_3.0.3
#> [33] purrr_1.2.2                 grid_4.6.0
#> [35] gdtools_0.5.1               scales_1.4.0
#> [37] MASS_7.3-65                 dichromat_2.0-0.1
#> [39] tinytex_0.60                SummarizedExperiment_1.43.0
#> [41] cli_3.6.6                   rmarkdown_2.31
#> [43] crayon_1.5.3                treeio_1.37.0
#> [45] otel_0.2.0                  httr_1.4.8
#> [47] rjson_0.2.23                BiocBaseUtils_1.15.1
#> [49] ape_5.8-1                   parallel_4.6.0
#> [51] ggplotify_0.1.3             restfulr_0.0.17
#> [53] matrixStats_1.5.0           vctrs_0.7.3
#> [55] yulab.utils_0.2.4           Matrix_1.7-5
#> [57] fontBitstreamVera_0.1.1     jsonlite_2.0.0
#> [59] bookdown_0.47               patchwork_1.3.2
#> [61] gridGraphics_0.5-1          systemfonts_1.3.2
```

```
#> [63] tidyr_1.3.2                glue_1.8.1
#> [65] codetools_0.2-20           gtable_0.3.6
#> [67] UCSC.utils_1.9.0           tibble_3.3.1
#> [69] pillar_1.11.1              rappdirs_0.3.4
#> [71] htmltools_0.5.9            R6_2.6.1
#> [73] evaluate_1.0.5             lattice_0.22-9
#> [75] Biobase_2.73.1             Rsamtools_2.29.0
#> [77] cigarillo_1.3.0            fontLiberation_0.1.0
#> [79] ggfun_0.2.0                Rcpp_1.1.1-1.1
#> [81] SparseArray_1.13.2         nlme_3.1-169
#> [83] xfun_0.59                  MatrixGenerics_1.25.0
#> [85] fs_2.1.0                   pkgconfig_2.0.3
```

10 References