

softwareRisk: Computation of Node and Path-Level Risk Scores in Scientific Models

The R package **softwareRisk** leverages the network-like architecture of scientific models together with software quality metrics to identify risky paths, which are defined by the complexity of its functions and the extent to which errors can cascade along and beyond their execution order. It operates on **tbl_graph** objects representing call dependencies between functions (callers and callees). By leveraging the **sensobol** package (Puy et al. 2022), **softwareRisk** also supports variance-based uncertainty and sensitivity analyses to evaluate how the identification of risky function-call paths varies under alternative assumptions about the relative importance of function complexity, coupling and structural position within the software.

Workflow

We first load the packages needed for the analysis.

```
library/softwareRisk)
library(tidygraph)
```

Prepare the required datasets

softwareRisk draws on the representation of the model's source code as a directed call graph $G = (V, E)$ in which each node $v_i \in V$ is a function or subroutine and each directed edge $e_{ij} = (v_i, v_j) \in E$ is a function call. It also assumes that each function will have a cyclomatic complexity value. Therefore the analyst should have two different datasets before starting the analysis:

1. A spreadsheet listing the set of directed edges as an edge list, with one row per function call. The first column may contain the caller function (source node, v_i , "from") and the second column the callee function (target node, v_j , "to"):
2. A spreadsheet listing the cyclomatic complexity values for each function in the model.

Let us create these datasets to illustrate their format:

```
# Dataset 1: calls (edge list) -----

calls_df <- data.frame(
  from = c("clean_data", "compute_risk", "compute_risk", "calc_scores", "plot_results"),
  to   = c("load_data", "clean_data", "calc_scores", "mean", "compute_risk")
)

calls_df
#>      from      to
#> 1 clean_data load_data
#> 2 compute_risk clean_data
#> 3 compute_risk calc_scores
#> 4 calc_scores      mean
#> 5 plot_results compute_risk

# Dataset 2: cyclomatic complexity (node attributes) -----
```

```
cyclo_df <- data.frame(
  name = c("clean_data", "load_data", "compute_risk", "calc_scores", "mean", "plot_results"),
  cyclo = c(6, 3, 12, 5, 2, 4)
)

cyclo_df
#>           name cyclo
#> 1 clean_data      6
#> 2 load_data       3
#> 3 compute_risk    12
#> 4 calc_scores     5
#> 5 mean           2
#> 6 plot_results    4
```

The analyst can then merge them into a `tbl_graph`:

```
# Merge into a tbl_graph -----

graph <- tbl_graph(nodes = cyclo_df, edges = calls_df, directed = TRUE)

graph
#> # A tbl_graph: 6 nodes and 5 edges
#> #
#> # A rooted tree
#> #
#> # Node Data: 6 x 2 (active)
#>   name      cyclo
#>   <chr>    <dbl>
#> 1 clean_data      6
#> 2 load_data       3
#> 3 compute_risk    12
#> 4 calc_scores     5
#> 5 mean           2
#> 6 plot_results    4
#> #
#> # Edge Data: 5 x 2
#>   from to
#>   <int> <int>
#> 1     1  2
#> 2     3  1
#> 3     3  4
#> # i 2 more rows
```

The function `read_call_graph` performs this merge while validating the input, so that common preparation errors (misspelled columns, functions present in the edge list but missing from the complexity table, duplicated names, non-numeric complexity values) fail early with an informative message. It accepts data frames or paths to `.csv` files and is the recommended entry point for models written in languages other than R (e.g., Fortran, C, Python), whose edge lists and complexity values are extracted with external tools:

```
# Merge and validate with read_call_graph -----

graph <- read_call_graph(edges = calls_df, metrics = cyclo_df)

graph
```

```

#> # A tbl_graph: 6 nodes and 5 edges
#> #
#> # A rooted tree
#> #
#> # Node Data: 6 x 2 (active)
#>   name      cyclo
#>   <chr>     <dbl>
#> 1 clean_data      6
#> 2 load_data       3
#> 3 compute_risk    12
#> 4 calc_scores     5
#> 5 mean            2
#> 6 plot_results    4
#> #
#> # Edge Data: 5 x 2
#>   from to
#>   <int> <int>
#> 1     1  2
#> 2     3  1
#> 3     3  4
#> # i 2 more rows

```

Building the call graph automatically from R code

For models written in R, no manual preparation is needed: the function `call_graph_fun` builds the call graph directly from the source code. It collects the functions defined in an installed package (`pkg` argument) or in a directory of .R scripts (`dir` argument), detects the calls among them by static analysis (`codetools::findGlobals`), and computes the cyclomatic complexity of each function by walking its abstract syntax tree, following McCabe (1976). The code is never executed. To illustrate, we write a small model to a temporary directory and build its graph:

```

# Write a small R model to a temporary directory -----

td <- file.path(tempdir(), "toy_model")
dir.create(td, showWarnings = FALSE)

writeLines(c(
  "load_data <- function(x) x",
  "clean_data <- function(x) load_data(x)",
  "calc_scores <- function(x) if (length(x) > 0) mean(x) else 0",
  "compute_risk <- function(x) calc_scores(clean_data(x))",
  "plot_results <- function(x) compute_risk(x)"
), file.path(td, "model.R"))

# Build the call graph -----

auto_graph <- call_graph_fun(dir = td)
#> Call graph: 5 functions, 4 calls.

auto_graph
#> # A tbl_graph: 5 nodes and 4 edges
#> #
#> # A rooted tree
#> #

```

```

#> # Node Data: 5 x 2 (active)
#>   name      cyclo
#>   <chr>    <dbl>
#> 1 load_data      1
#> 2 clean_data      1
#> 3 calc_scores     2
#> 4 compute_risk     1
#> 5 plot_results     1
#> #
#> # Edge Data: 4 x 2
#>   from to
#>   <int> <int>
#> 1     2  1
#> 2     4  3
#> 3     4  2
#> # i 1 more row

```

The resulting `tbl_graph` carries the `cyclo` node attribute and feeds directly into the pipeline described below. The same works for any installed package, e.g., `call_graph_fun(pkg = "softwareRisk")` analyzes `softwareRisk` itself. Two limitations of static analysis are worth keeping in mind: calls built at run time (e.g., through `do.call` with a constructed name, `get`, or method dispatch) cannot be detected, and a function passed as a value (e.g., to `lapply`) is treated as called.

Once this is done, the data is prepared for `softwareRisk`.

Analysis

Here we illustrate the functions of `softwareRisk` by using the built-in data `synthetic_graph`. It consists of five entry nodes, 35 middle nodes and 15 sink nodes. Each entry node calls between two and five middle nodes and each middle node calls one to three sink nodes, thus simulating realistic code architecture. The synthetic example reproduces the characteristic right-tailed distribution of cyclomatic complexity found in real software systems, with many low-complexity functions and few highly complex ones (Landman et al. 2016).

```

# Load the data -----
data("synthetic_graph")

# Print it -----

synthetic_graph
#> # A tbl_graph: 55 nodes and 122 edges
#> #
#> # A directed acyclic simple graph with 1 component
#> #
#> # Node Data: 55 x 2 (active)
#>   name cyclo
#>   <chr> <dbl>
#> 1 E1      7
#> 2 M15     10
#> 3 M14     39
#> 4 M3      12
#> 5 M10     13
#> 6 E2     43

```

```

#> 7 M25      17
#> 8 M26      3
#> 9 E3       6
#> 10 M5      8
#> # i 45 more rows
#> #
#> # Edge Data: 122 x 2
#>   from    to
#>   <int> <int>
#> 1     1     2
#> 2     1     3
#> 3     1     4
#> # i 119 more rows

```

The next step is to compute the in-degree and betweenness centrality of each node, calculate its risk score and identify all simple paths through the directed function-call graph. All this is done with the `all_paths_fun` function. The in-degree and betweenness of the nodes are calculated internally by functions in the `igraph` package.

Definition of software risk

Risk is computed using a weighted power-mean aggregation of normalized cyclomatic complexity, in-degree and betweenness. The power parameter p controls how attributes combine: values of $p < 1$ emphasize functions where several risk factors co-occur, while values of $p > 1$ increasingly focus on the largest individual contributor. When $p = 1$, the formula reduces to a simple weighted sum (additive risk).

$$r_{(v_i)}^{(p)} = \left(\alpha \tilde{C}_{(v_i)}^p + \beta \tilde{d}_{(v_i)}^{in p} + \gamma \tilde{b}_{(v_i)}^p \right)^{1/p}, \quad p \in [-1, 2]. \quad (1)$$

where the tilde $\sim$ refers to normalization, C denotes cyclomatic complexity, d^{in} refers to in-degree and b denotes betweenness. The weights α , β and γ reflect the relative importance of complexity, coupling and network position and can be defined by the analyst, with the constraint that $\alpha + \beta + \gamma = 1$. High r values indicate functions that are complex and/or highly interconnected and hence potential points of structural vulnerability.

Path-level risk scores

The risk scores computed at the function level are then aggregated at the path level as

$$P_k = 1 - \prod_{i=1}^{n_k} (1 - r_{k(v_i)}), \quad (2)$$

where $r_{k(v_i)}$ is the risk of the i -th function in path k and n_k is the number of functions in that path. P_k is at least as large as the maximum individual function risk and monotonically increases as more functions on the path become risky, approaching 1 when several functions have high risk scores. High P_k scores thus identify not only vulnerable paths, but also paths whose potential failure can have a larger cascading effect into other parts of the system through their shared high-centrality functions.

In this example, we set $p = 1$ (additive risk), $\alpha = 0.6$, $\beta = 0.3$ and $\gamma = 0.1$, thus prioritizing defect-proneness and the likelihood of unexpected behaviours and relegating propagation potential as secondary.

```

# Run the function -----
output <- all_paths_fun(graph = synthetic_graph,

```

```

alpha = 0.6,
beta  = 0.3,
gamma = 0.1,
complexity_col = "cyclo",
weight_tol = 1e-8)

# Print the output -----

output
#> $nodes
#> # A tibble: 55 x 6
#>   name cyclomatic_complexity indeg outdeg btw risk_score
#>   <chr>          <dbl> <dbl> <dbl> <dbl>      <dbl>
#> 1 E1              7     0     4  0      0.0610
#> 2 M15             10     3     3 14.5     0.236
#> 3 M14             39     2     3 12      0.485
#> 4 M3              12     1     3  2.5     0.157
#> 5 M10             13     3     2 34.8     0.289
#> 6 E2              43     0     3  0      0.427
#> 7 M25             17     3     4 25      0.319
#> 8 M26              3     2     3  7      0.114
#> 9 E3               6     0     4  0      0.0508
#> 10 M5              8     1     6  7.25    0.122
#> # i 45 more rows
#>
#> $paths
#> # A tibble: 209 x 10
#>   path_id path_nodes path_str      hops path_risk_score path_cc gini_node_risk
#>   <int> <list>      <chr>    <dbl>          <dbl> <list>          <dbl>
#> 1     1 1 <chr [6]> "E1 \u2192 M~    5      0.900 <dbl>          0.253
#> 2     2 2 <chr [4]> "E1 \u2192 M~    3      0.793 <dbl>          0.277
#> 3     3 3 <chr [5]> "E1 \u2192 M~    4      0.773 <dbl>          0.243
#> 4     4 4 <chr [6]> "E2 \u2192 M~    5      0.939 <dbl>          0.136
#> 5     5 5 <chr [4]> "E2 \u2192 M~    3      0.874 <dbl>          0.0946
#> 6     6 6 <chr [5]> "E2 \u2192 M~    4      0.868 <dbl>          0.132
#> 7     7 7 <chr [3]> "E2 \u2192 M~    2      0.725 <dbl>          0.0841
#> 8     8 8 <chr [5]> "E3 \u2192 M~    4      0.781 <dbl>          0.253
#> 9     9 9 <chr [3]> "E3 \u2192 M~    2      0.545 <dbl>          0.269
#> 10    10 10 <chr [6]> "E3 \u2192 M~    5      0.799 <dbl>          0.285
#> # i 199 more rows
#> # i 3 more variables: risk_slope <dbl>, risk_mean <dbl>, risk_sum <dbl>

```

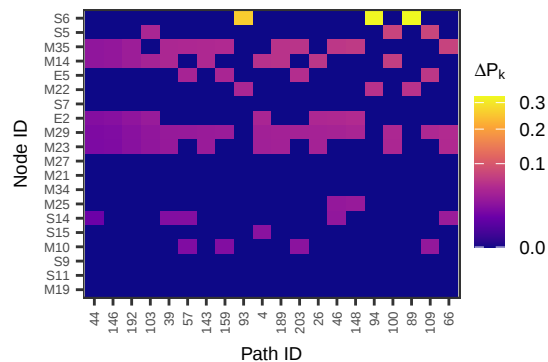
The output of `all_paths_fun` is a list with two slots, `nodes` and `paths`.

- `$nodes`: it contains the name of the nodes and their relevant metrics (cyclomatic complexity, in-degree, out-degree, betweenness and risk score).
- `$paths`: it describes each simple path in the call graph and includes the ordered sequence of nodes forming the path, the number of hops (i.e., the number of edges traversed along the path), the path-level risk score and the vector of cyclomatic complexity values for the nodes along the path (`path_cc` column). In addition, two distributional metrics are reported: the Gini coefficient of node-level risk along the path (`gini_node_risk` column), which quantifies the inequality of risk contributions across nodes within a path, and the `risk_slope`, which captures the direction and magnitude of change in node-level risk from the beginning to the end of the path.

Plotting

`softwareRisk` provides functions to inspect the results of the analysis. The function `path_fix_heatmap` allows the analyst to choose the top n nodes and n paths in terms of their risk score and observe how much the risk score of the riskiest paths would decrease if the selected high-risk nodes were made perfectly reliable. This analysis identifies nodes that act as chokepoints for risk propagation, highlights paths dominated by single high-risk functions and reveals which refactoring actions would yield the greatest reductions in path-level risk.

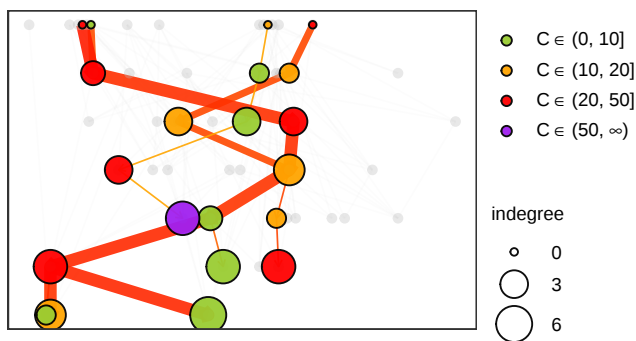
```
path_fix_heatmap(all_paths_out = output, n_nodes = 20, k_paths = 20)
#> $delta_tbl
#> # A tibble: 400 x 3
#>   node path_id deltaR
#>   <fct> <fct>   <dbl>
#> 1 S6    44      0
#> 2 S6   146      0
#> 3 S6   192      0
#> 4 S6   103      0
#> 5 S6    39      0
#> 6 S6    57      0
#> 7 S6   143      0
#> 8 S6   159      0
#> 9 S6    93    0.260
#> 10 S6     4      0
#> # i 390 more rows
#>
#> $plot
```



`softwareRisk` also allows to plot the call graph with the top risky paths highlighted. This is done with the function `plot_top_paths_fun`. The top ten most risky paths are highlighted in colour. The thickness of the edge shows how frequently an edge participates in the top 10 most risky paths. The color of the edge (from orange to red) indicates the mean risk of paths including that edge.

```
plot_output <- plot_top_paths_fun(graph = synthetic_graph,
                                  all_paths_out = output,
                                  model.name = "ToyModel",
                                  language = "Fortran",
                                  top_n = 10,
                                  alpha_non_top = 0.05)
```

ToyModel: Fortran

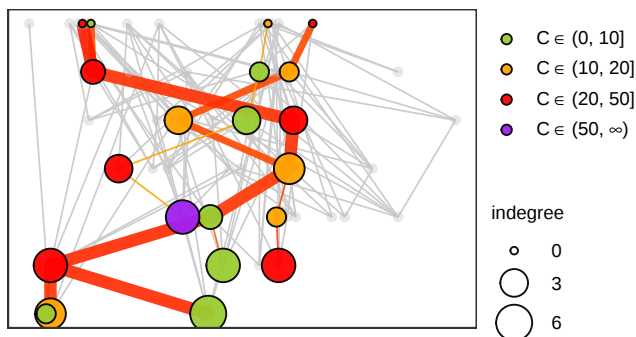


The color of the nodes maps onto the cyclomatic complexity categories defined by Watson and McCabe (1996) (0-10 low risk; 10-20 moderate complexity, 20-50 complex, high risk; > 50 very complex, untestable).

The `alpha_non_top` argument controls the transparency of the paths that are not identified as top. For small or sparse models, it can be set to `alpha_non_top = 1` to better visualize the full call graph:

```
plot_output <- plot_top_paths_fun(graph = synthetic_graph,
  all_paths_out = output,
  model.name = "ToyModel",
  language = "Fortran",
  top_n = 10,
  alpha_non_top = 1)
```

ToyModel: Fortran



Path-aware node criticality

The node risk score captures how error-prone a function is in isolation, but not how many risky execution chains depend on it. The function `node_exposure_fun` summarizes, for each node, its participation in the enumerated entry-to-sink paths: the number of paths containing the node, the *risk load* (the sum of P_k over those paths) and the corresponding shares. Comparing the rank of a node by risk load with its rank by risk score separates *chokepoints* (functions that many risky paths route through) from *hotspots* (functions that are complex in themselves):

```
# Compute node exposure -----

exposure <- node_exposure_fun(output)

exposure
#> # A tibble: 55 x 9
#>   name risk_score n_paths path_share risk_load risk_load_share mean_path_risk
#>   <chr>      <dbl>   <int>    <dbl>    <dbl>      <dbl>          <dbl>
```

```
#> 1 M29      0.393      77      0.368      66.0      0.441      0.857
#> 2 M35      0.486      63      0.301      53.7      0.359      0.852
#> 3 M31      0.206      60      0.287      51.0      0.341      0.850
#> 4 M23      0.391      36      0.172      31.2      0.209      0.868
#> 5 M25      0.319      39      0.187      31.2      0.209      0.800
#> 6 E3       0.0508     39      0.187      27.8      0.186      0.713
#> 7 E2       0.427      29      0.139      25.3      0.169      0.871
#> 8 S14      0.308      30      0.144      23.0      0.154      0.767
#> 9 M14      0.485      26      0.124      23.0      0.153      0.883
#> 10 E1      0.0610     27      0.129      20.7      0.139      0.768
#> # i 45 more rows
#> # i 2 more variables: rank_risk_load <int>, rank_risk_score <int>
```

A node ranking much higher on `rank_risk_load` than on `rank_risk_score` is a chokepoint rather than a hotspot: refactoring it pays off because of *where* it sits, not because of *what* it contains.

Prioritizing fixes under a budget

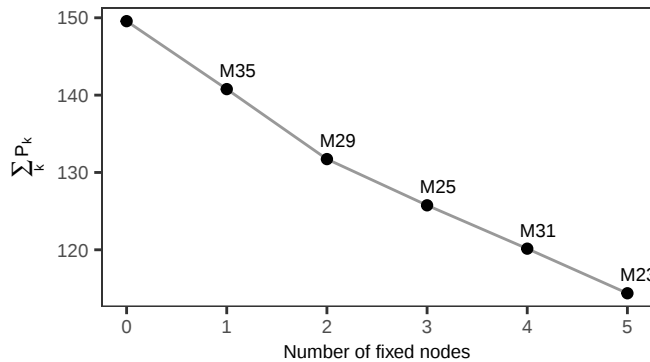
The heatmap above shows the effect of fixing single nodes on single paths. The function `fix_portfolio_fun` answers the budgeted question directly: given resources to refactor k functions, which k ? At each step it evaluates every candidate node, fixes the one whose repair (risk set to 0) most reduces the objective, and repeats. Two objectives are available: the total path risk $\sum_k P_k$ (default) and the risk of the riskiest path $\max_k P_k$. For the total-risk objective the improvement is monotone submodular, so the greedy selection is guaranteed to achieve at least $1 - 1/e \approx 63\%$ of the reduction of the optimal set of the same size (Nemhauser, Wolsey, and Fisher 1978).

```
# Greedy portfolio of five fixes -----

portfolio <- fix_portfolio_fun(output, budget = 5, objective = "total")

portfolio$portfolio
#> # A tibble: 5 x 5
#>   step node objective_after delta cum_reduction
#>   <int> <chr>          <dbl> <dbl>          <dbl>
#> 1     1 M35             141.  8.79           0.0588
#> 2     2 M29             132.  9.05           0.119
#> 3     3 M25             126.  5.97           0.159
#> 4     4 M31             120.  5.61           0.197
#> 5     5 M23             114.  5.77           0.235

portfolio$plot
```



The curve displays the diminishing returns typical of submodular objectives and helps locate the point where

additional refactoring stops paying off.

Uncertainty and sensitivity analysis

softwareRisk also enables the analyst to perform uncertainty and sensitivity analyses of risk and path score calculations by leveraging the **sensobol** package (Puy et al. 2022). By systematically varying the weights (α, β, γ) and the power parameter p , the package allows users to evaluate how sensitive node- and path-level risk scores are to different risk conceptualizations. This approach makes it possible to assess the robustness of the identification of high-risk paths under alternative definitions of risk.

Uncertainty and sensitivity analyses are implemented through the function **uncertainty_fun**. The user needs to define the order of the effects explored (**first**, **second** or **third**).

Internally, **uncertainty_fun** builds a Sobol' quasi-random design over four independent $U(0,1)$ draws. Three of them (**a_raw**, **b_raw**, **c_raw**) are normalised to sum to one, yielding the weights α , β and γ ; the fourth (**p_raw**) is mapped linearly to $p \in [-1, 2]$. Independent uniform inputs are required by the quasi-random sequence, hence the need for the raw draws; the sensitivity indices, however, are attributed to the transformed parameters and labelled **alpha**, **beta**, **gamma** and **p** in the output, so the results are directly interpretable in terms of the model parameters.

```
# Run uncertainty analysis -----

uncertainty_analysis <- uncertainty_fun(all_paths_out = output,
                                       N = 2^10,
                                       order = "first")

# Print the top five rows -----

lapply(uncertainty_analysis, function(x) head(x, 5))
#> $nodes
#> # A tibble: 5 x 3
#>   name uncertainty_analysis sensitivity_analysis
#>   <chr> <list>          <list>
#> 1 E1    <dbl [1,024]>      <sensobol>
#> 2 M15   <dbl [1,024]>      <sensobol>
#> 3 M14   <dbl [1,024]>      <sensobol>
#> 4 M3    <dbl [1,024]>      <sensobol>
#> 5 M10   <dbl [1,024]>      <sensobol>
#>
#> $paths
#> # A tibble: 5 x 6
#>   path_id path_str             hops uncertainty_analysis gini_index risk_trend
#>   <int> <chr>              <dbl> <list>          <list>      <list>
#> 1     1 "E1 \u2192 M14 \u2192~    5 <dbl [1,024]>      <dbl>      <dbl>
#> 2     2 "E1 \u2192 M14 \u2192~    3 <dbl [1,024]>      <dbl>      <dbl>
#> 3     3 "E1 \u2192 M10 \u2192~    4 <dbl [1,024]>      <dbl>      <dbl>
#> 4     4 "E2 \u2192 M14 \u2192~    5 <dbl [1,024]>      <dbl>      <dbl>
#> 5     5 "E2 \u2192 M14 \u2192~    3 <dbl [1,024]>      <dbl>      <dbl>
```

The output is a list with two slots:

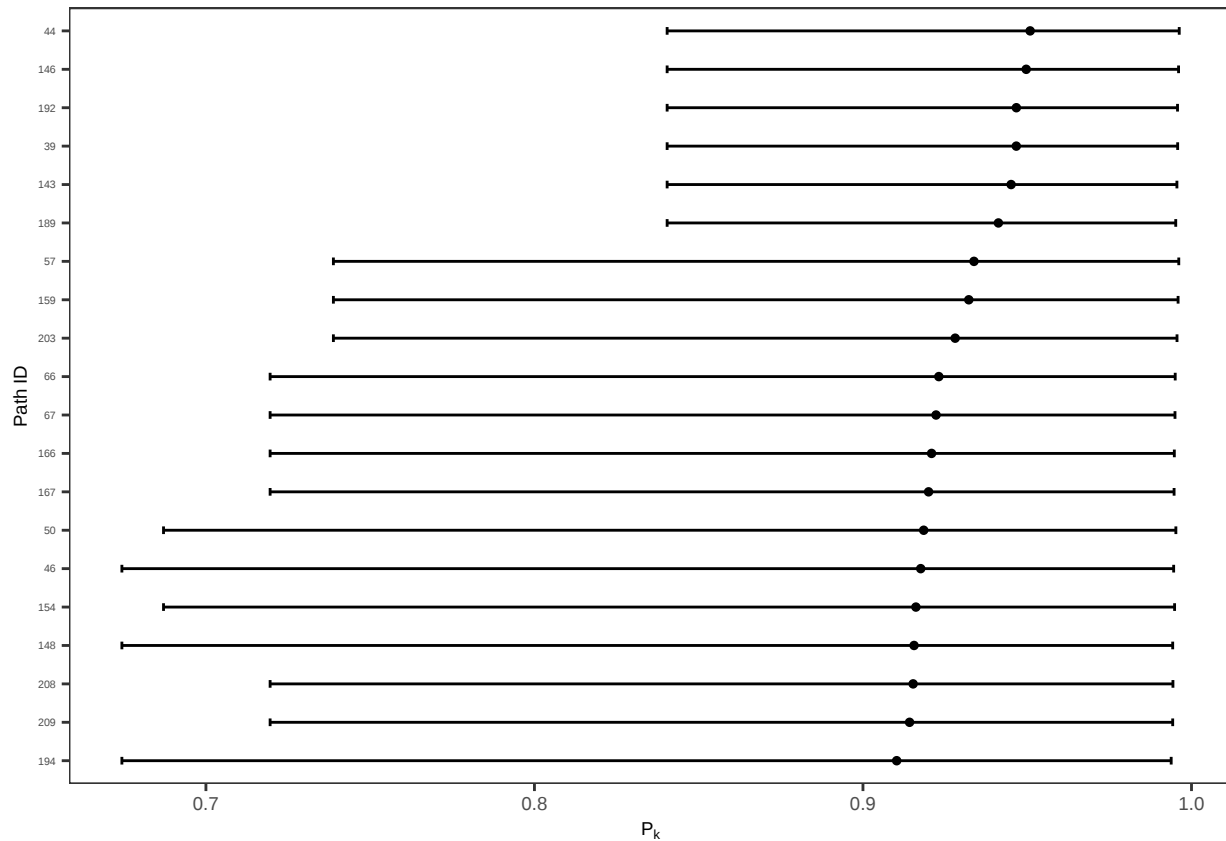
- **\$nodes**: a **name** column with the name of the node, an **uncertainty_analysis** column with a vector of N node-level risk scores after randomizing α , β , γ and p , and a **sensitivity_analysis** column with the results of the sensitivity analysis. Each element of **sensitivity_analysis** is a **sensobol::sobol_indices()** object whose **\$results** data frame reports first-order (S_i) and/or total-order (T_i) indices for the four parameters, labelled **alpha**, **beta**, **gamma** and **p**. See the **sensobol** package

for further details (Puy et al. 2022).

- `$paths`: a `path_id` column with the ID number of the path, a `path_str` column informing on the sequence of functions calls for that path, a `hops` column informing on the number of edges traversed along the path and three columns informing on the results of the uncertainty analysis (UA):
 - `uncertainty_analysis`: vector of path-level risk scores after the UA.
 - `gini_index`: vector of `gini_index` values after the UA.
 - `risk_trend`: vector of `risk_slope` values after the UA.

The analyst can also plot the top n risky paths and their uncertainty with the function `path_uncertainty_plot`. The error bars encompass the minimum, mean and maximum P_k value for that path after the uncertainty analysis.

```
path_uncertainty_plot(ua_sa_out = uncertainty_analysis, n_paths = 20)
```



Robustness of the risk ranking

Every uncertainty draw corresponds to one plausible definition of risk (one sampled combination of α , β , γ and p). Ranking the paths within each draw therefore shows which paths are flagged as risky *regardless* of how risk is weighted and which ones enter the top- k only under specific assumptions. The function `rank_robustness_fun` computes, for each path (or node, with `what = "nodes"`), the probability of belonging to the top- k across draws, together with the median and the 5th and 95th percentiles of its rank. As a global summary it reports the mean Spearman correlation between the risk values of each draw and the consensus (mean) values: values close to 1 indicate that the ranking barely responds to the risk definition.

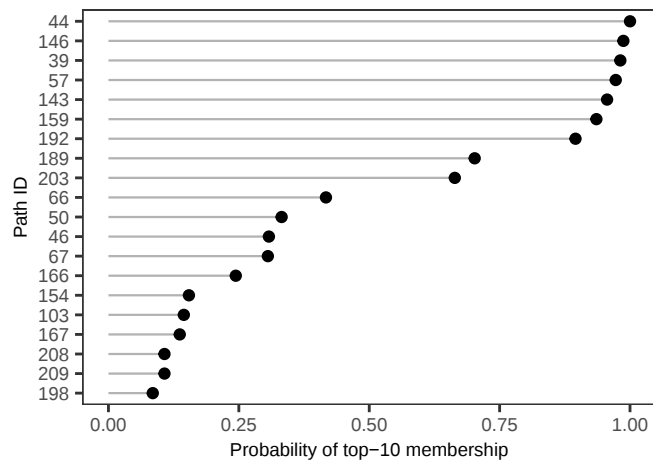
```
# Robustness of the top-10 path ranking -----
robustness <- rank_robustness_fun(uncertainty_analysis, top_k = 10, what = "paths")
```

```
robustness$summary
#> # A tibble: 209 x 7
#>       id path_str      mean_risk top_k_prob rank_median rank_q05 rank_q95
#>   <int> <chr>          <dbl>      <dbl>      <dbl>      <dbl>      <dbl>
#> 1    44 "E2 \u2192 M14 \u21~ 0.951        1          1          1          3
#> 2   146 "E2 \u2192 M14 \u21~ 0.950      0.987          2          1          4
#> 3    39 "E1 \u2192 M14 \u21~ 0.947      0.981          3          1          7
#> 4    57 "E5 \u2192 M7 \u219~ 0.934      0.973          7          2          9
#> 5   143 "E1 \u2192 M14 \u21~ 0.945      0.956          4          1         10
#> 6   159 "E5 \u2192 M7 \u219~ 0.932      0.936          7          5         12
#> 7   192 "E2 \u2192 M14 \u21~ 0.947      0.896          4          1         14
#> 8   189 "E1 \u2192 M14 \u21~ 0.941      0.702          6          1         20
#> 9   203 "E5 \u2192 M7 \u219~ 0.928      0.664          9          7         22
#> 10    66 "M13 \u2192 M23 \u2~ 0.923      0.417         12          6         19
#> # i 199 more rows

robustness$consensus_correlation
#> [1] 0.9692249
```

The companion function `rank_robustness_plot` displays the top- k membership probability of the paths most often flagged as risky:

```
rank_robustness_plot(robustness, top_n = 20)
```



Paths with a probability close to 1 are risky under essentially any weighting of complexity, coupling and structural position; intermediate probabilities identify paths whose criticality depends on the analyst's assumptions.

Extracting and plotting Sobol' indices

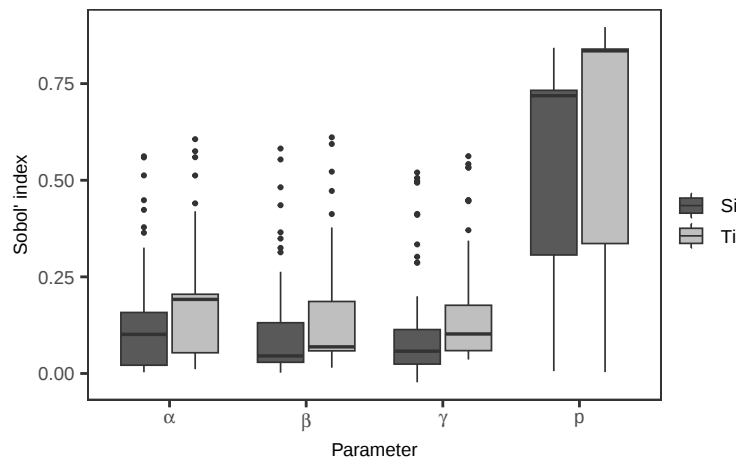
The `sensitivity_analysis` list-column in `$nodes` stores one `sensobol::sobol_indices()` object per node. The Sobol' indices for a given node are accessible via the `$results` slot of that object, which is a data frame with three columns: `original` (the index value), `sensitivity` ("Si" for first-order, "Ti" for total-order), and `parameters` ("alpha", "beta", "gamma", "p").

```
# Sobol' indices for the first node
si_node1 <- uncertainty_analysis$nodes$sensitivity_analysis[[1]]$results
si_node1
#> Index: <sensitivity>
```

```
#>      original sensitivity parameters
#>      <num>      <char>      <char>
#> 1: 0.10120387      Si      alpha
#> 2: 0.02898638      Si      beta
#> 3: 0.02429127      Si      gamma
#> 4: 0.71862389      Si      p
#> 5: 0.19167458      Ti      alpha
#> 6: 0.05858662      Ti      beta
#> 7: 0.05915430      Ti      gamma
#> 8: 0.83929594      Ti      p
```

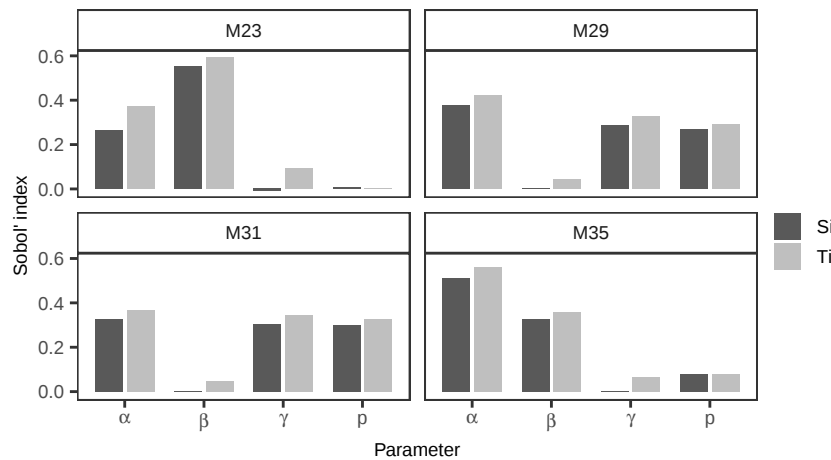
The function `sensitivity_plot_fun` wraps the extraction and visualization of these indices. With the default `nodes = NULL`, it shows the distribution of S_i and T_i for each parameter across all nodes, answering the system-level question of which assumption dominates the risk scores. A large gap between S_i and T_i for a parameter indicates important higher-order interactions with the other parameters.

```
sensitivity_plot_fun(uncertainty_analysis)
```



With `nodes` set to a character vector of node names, the indices of those nodes are displayed individually with their confidence intervals, one panel per node:

```
sensitivity_plot_fun(uncertainty_analysis,
  nodes = c("M29", "M35", "M23", "M31"))
```



References

- Landman, Davy, Alexander Serebrenik, Eric Bouwers, and Jurgen J. Vinju. 2016. “Empirical Analysis of the Relationship Between CC and SLOC in a Large Corpus of Java Methods and C Functions.” *Journal of Software: Evolution and Process* 28 (7): 589–618. <https://doi.org/10.1002/smr.1760>.
- McCabe, T. J. 1976. “A Complexity Measure.” *IEEE Transactions on Software Engineering* SE-2 (4): 308–20. <https://doi.org/10.1109/TSE.1976.233837>.
- Nemhauser, G. L., L. A. Wolsey, and M. L. Fisher. 1978. “An Analysis of Approximations for Maximizing Submodular Set Functions – I.” *Mathematical Programming* 14 (1): 265–94. <https://doi.org/10.1007/BF01588971>.
- Puy, Arnald, Samuele Lo Piano, Andrea Saltelli, and Simon A. Levin. 2022. “Sensobol: An R Package to Compute Variance-Based Sensitivity Indices.” *Journal of Statistical Software* 102 (5): 1–37. <https://doi.org/10.18637/jss.v102.i05>.
- Watson, Arthur H, and Thomas J McCabe. 1996. “Structured Testing : A Testing Methodology Using the Cyclomatic Complexity Metric.” NIST SP 500-235. Edited by Dolores R Wallace. 0th ed. Gaithersburg, MD: National Institute of Standards and Technology. <https://doi.org/10.6028/NIST.SP.500-235>.