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), sofwareRisk 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 set of directed edges as an edge list, with one row per function call. The first column may contian the caller function (source node, v_i, "from") and the second column the calle 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
```

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

Analysis

Here we illustrate the functions of `softwareRisk` by using the build-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. The risk score for node v_i is computed as

$$r_{(v_i)} = \alpha\,\tilde{C}_{(v_i)} + \beta\,\tilde{d}^{\text{in}}_{(v_i)} + \gamma\,\tilde{b}_{(v_i)}\,, \tag{1}$$

where the tilde˜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. High r values indicate functions that are complex and/or highly interconnected and hence potential points of structural vulnerability.

Path-level risk scores are defined as

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

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. The equation above behaves like a saturating OR-operator: 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 $\alpha = 0.6$, $\beta = 0.3$ and $\gamma = 0.1$ to adopt a definition of risk that prioritizes defect-proneness and the likelihood of unexpected behaviours, thus relegating propagation potential as secondary. Other weights are possible, with the only constrain that $\alpha + \beta + \gamma = 1$.

```
# 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
#>      <int> <list>     <chr>               <dbl>           <dbl> <list>
#>  1       1 <chr [6]>  E1 → M14 → M23 → M~     5           0.900 <dbl>
#>  2       2 <chr [4]>  E1 → M14 → M23 → S~     3           0.793 <dbl>
#>  3       3 <chr [5]>  E1 → M10 → M29 → M~     4           0.773 <dbl>
#>  4       4 <chr [6]>  E2 → M14 → M23 → M~     5           0.939 <dbl>
#>  5       5 <chr [4]>  E2 → M14 → M23 → S~     3           0.874 <dbl>
#>  6       6 <chr [5]>  E2 → M25 → M29 → M~     4           0.868 <dbl>
#>  7       7 <chr [3]>  E2 → M25 → S15          2           0.725 <dbl>
#>  8       8 <chr [5]>  E3 → M25 → M29 → M~     4           0.781 <dbl>
#>  9       9 <chr [3]>  E3 → M25 → S15          2           0.545 <dbl>
#> 10      10 <chr [6]>  E3 → M5 → M10 → M2~     5           0.799 <dbl>
#> # i 199 more rows
#> # i 4 more variables: gini_node_risk <dbl>, risk_slope <dbl>,
#> #   risk_mean <dbl>, risk_sum <dbl>
```

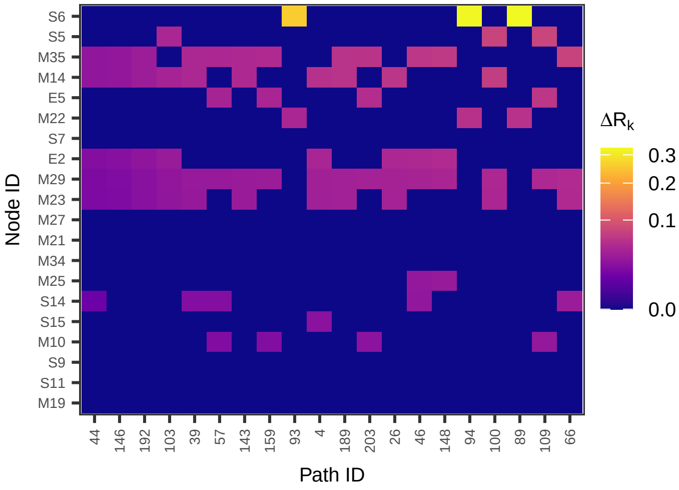
The output of `all_paths_fun` is a list with two slots, `nodes` and `paths`.

- `$nodes`: it contains the name of the nodes and their relevant metrics (cyclomatic complexity, in-degree, out-degree, betweenness and risk score).
- `$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 chose 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.

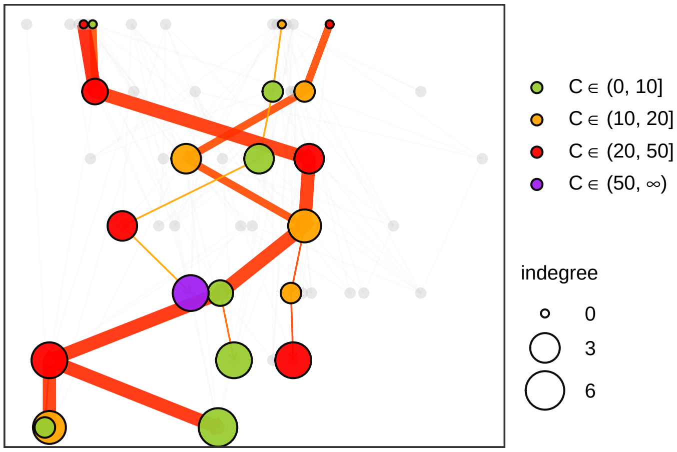
```r
path_fix_heatmap(all_paths_out = output, n_nodes = 20, k_paths = 20)
#> $delta_tbl
#> # A tibble: 400 x 3
#>    node  path_id deltaR
#>    <fct> <fct>    <dbl>
#>  1 S6    44       0
#>  2 S6    146      0
#>  3 S6    192      0
#>  4 S6    103      0
#>  5 S6    39       0
#>  6 S6    57       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.

```r
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)
```

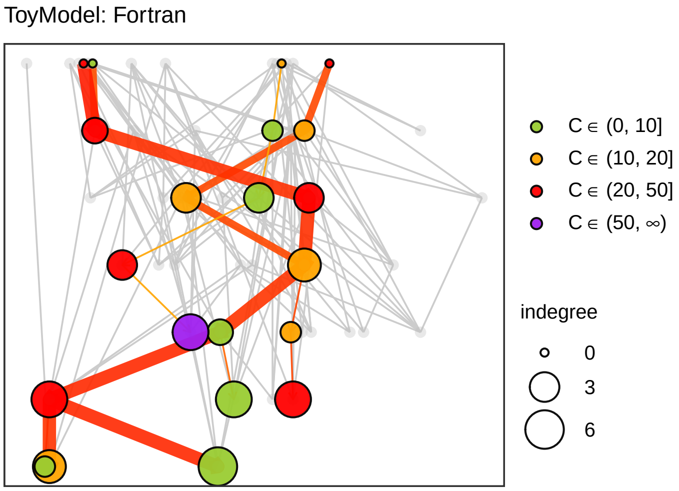
ToyModel: Fortran



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:

```r
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)
```



Uncertainty and sensitivity analysis

`softwareRisk` also enables the analyst to perform uncertainty and sensitivity analyses of risk score calculations by leveraging the sensobol package (Puy et al. 2022). By systematically varying the relative contribution of cyclomatic complexity and structural metrics such as in-degree and betweenness, the package allows users to evaluate how sensitive node- and path-level risk scores are to modelling assumptions. This approach makes it possible to assess the robustness of the identification of high-risk paths under alternative definitions of risk.

The relative weights assigned to cyclomatic complexity and structural metrics (in-degree and betweenness) are parameterized by the coefficients α, β, and γ, which are randomly sampled and internally transformed into normalized weights (α_{raw}, β_{raw}, and γ_{raw}) that sum to one.

Uncertainty and sensitivity analyses are implemented through the function `uncertainty_fun`:

```
# 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
#>      name
#>    <char>
#> 1:     E1
#> 2:    M15
#> 3:    M14
#> 4:     M3
#> 5:    M10
#>                                                              uncertainty_analysis
#>                                                                            <list>
#> 1: 0.03389831,0.04358354,0.02033898,0.02773498,0.04745763,0.05649718,...[2048]
#> 2:       0.2474083,0.1956474,0.3198736,0.2317179,0.2819272,0.1860685,...[2048]
#> 3:       0.3543718,0.3739882,0.3269088,0.3141830,0.4427872,0.4340059,...[2048]
#> 4:       0.1190252,0.1122161,0.1285580,0.1024348,0.1555242,0.1287104,...[2048]
#> 5:       0.3393575,0.3138678,0.3750430,0.3478581,0.3206560,0.2893171,...[2048]
#>    sensitivity_analysis
#>                   <list>
#> 1:    <data.table[6x3]>
#> 2:    <data.table[6x3]>
#> 3:    <data.table[6x3]>
#> 4:    <data.table[6x3]>
#> 5:    <data.table[6x3]>
#>
#> $paths
#>    path_id                         path_str  hops
#>      <int>                           <char> <num>
#> 1:       1 E1 → M14 → M23 → M29 → M31 → S15     5
#> 2:       2             E1 → M14 → M23 → S15     3
#> 3:       3       E1 → M10 → M29 → M31 → S15     4
#> 4:       4 E2 → M14 → M23 → M29 → M31 → S15     5
#> 5:       5             E2 → M14 → M23 → S15     3
#>                                                    uncertainty_analysis
#>                                                                  <list>
#> 1: 0.9257106,0.9177837,0.9390559,0.9351443,0.9143014,0.9025447,...[2048]
#> 2: 0.7303364,0.6921096,0.7880574,0.6902704,0.8078236,0.7221158,...[2048]
#> 3: 0.8739811,0.8528306,0.9041735,0.8980060,0.8258811,0.8012193,...[2048]
#> 4: 0.9413505,0.9402631,0.9466475,0.9462448,0.9399195,0.9375586,...[2048]
#> 5: 0.7871077,0.7762923,0.8144586,0.7432827,0.8652713,0.8219545,...[2048]
```

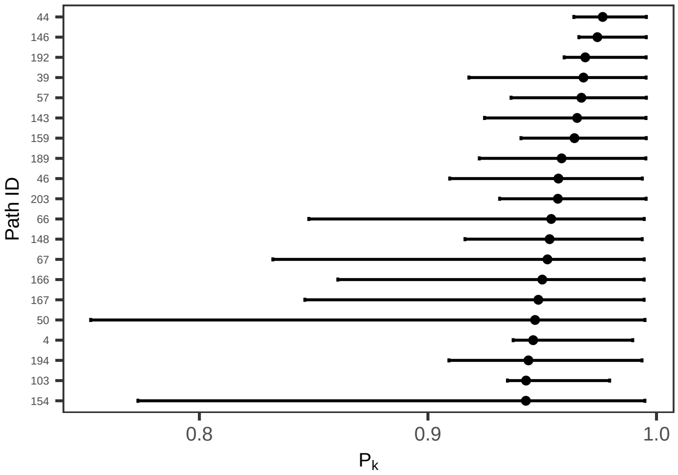
The output is a list with two slots:

- `$nodes`: a `name` column with the name of the node, an `uncertainty_analysis` column with a list of all the node-level risk scores after randomizing α, β and γ, and a `sensitivity_analysis` column with a data.table informing of the results of the sensitivity analysis.
- `$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 an `uncertainty_analysis` with a vector of path-level risk scores after the uncertainty analysis.

The `sensitivity_analysis` column can be unlisted and inspected as any other `data.table`. See the `sensobol` package for information on first-order (S_i) and total-order (T_i) indices (Puy et al. 2022).

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)
#> `height` was translated to `width`.
```



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.

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.