

Threadle User Guide (v1.2)

How to use Threadle with register data

Current versions: Threadle 1.2.0, threadleR 0.5.1



Carl Nordlund

Threadle User Guide (v1.2)

How to use Threadle with register data

Current versions: Threadle 1.2.0, threadleR 0.5.1

Carl Nordlund



Linköping University Electronic Press

Linköpings universitet, SE-581 83 Linköping, Sweden

Linköping 2026

Version 1.2, Linköping University Electronic Press, augusti 2026, Redaktör: Edvin Erdtman

© 2026 Carl Nordlund

 Förutom där något annat anges är detta verk licensierat under en Creative Commons Erkännande Icke-Kommerciell 4.0 Internationell licens. För att visa licensvillkoren gå till:
<https://creativecommons.org/licenses/by-nc/4.0/deed.sv>

ISBN: 978-91-8118-713-7 (PDF)

DOI: <https://doi.org/10.3384/9789181187137>

Contents

1	Introduction	6
1.1	Why Threadle? Why not igraph?	6
1.2	How Threadle addresses these challenges.....	6
1.3	The Threadle ecosystem	7
1.4	Funding and acknowledgements	8
1.5	How to read this guide	8
2	Core concepts.....	9
2.1	Nodesets and Networks.....	9
2.2	Layers: 1-mode and 2-mode relationships	9
2.3	The Pseudo-projection of 2-mode ties.....	10
2.4	Node attributes.....	11
2.5	File formats	11
2.5.1	Mutable vs. packed layers	12
3	Installation and Setup	13
3.1	Using Threadle on Mona.....	13
3.2	Using Threadle outside Mona.....	14
3.2.1	Installing Threadle.....	14
3.2.2	Installing and configuring threadleR	14
4	Building a Network from Register data	15
4.1	Overview of the pipeline.....	15
4.2	Step 1: SQL query to R data frame.....	15
4.3	Step 2: Writing temporary text file	17
4.4	Step 3: Importing temporary data files into Threadle	17
4.4.1	Creating the Nodeset and the Network	17
4.4.2	Adding a Layer	18
4.4.3	Importing the Edge/Affiliation file	18
4.5	Step 4: Saving the Threadle file(s).....	19
4.6	Verification and clean-up.....	20
5	Extending an existing network.....	21
5.1	Adding a new layer	21
5.2	Adding new node attributes.....	21
5.3	Packing finished layers.....	22
5.4	Saving an updated network.....	22
6	Querying a Threadle network.....	23
6.1	Inspecting a Network	23
6.2	Node queries.....	23
6.3	Neighbor and alter queries	23
6.4	Edge queries	24
6.5	Degree.....	24
6.6	Components	25
6.7	Shortest paths.....	25

6.8	Density	26
6.9	Filtering and subnetworks	26
7	Working with results in R.....	27
7.1	A note on data exports and microdata regulations	27
7.2	Getting node measures into R.....	27
7.3	Sampling node measures inside Threadle.....	28
7.4	Projecting 2-mode layers into equivalent 1-mode layers	28
8	Practical considerations on Mona	29
8.1	Working directories.....	29
8.2	File storage	29
8.3	Memory management	29
8.4	Long-running operations (imports, random walkers etc)	29
8.5	Verbosity.....	30
8.6	Reproducibility.....	30
8.7	Session hygiene	30
9	Reference	31
9.1	threadleR function reference	31
9.1.1	Session management	31
9.1.2	Structure management	31
9.1.3	Layer management	32
9.1.4	Data import and export	33
9.1.5	Node operations.....	34
9.1.6	Edge and affiliation operations	34
9.1.7	Node queries	35
9.1.8	Analysis and transformations.....	36
9.1.9	Utilities	37
9.2	Threadle CLI command quick reference	38
9.3	File format reference.....	41
9.3.1	Nodeset TSV format.....	41
9.3.2	Network TSV format	41
9.3.3	Edge list format (1-mode).....	42
9.3.4	Affiliation list format (2-mode)	42
9.3.5	Threadle binary (.bin) format.....	42
9.4	Glossary	43

1 Introduction

Threadle is a high-performance software system for building, storing, querying and analyzing large, complex network datasets. It was developed explicitly for working with Swedish full-population administrative register data, i.e. the kind of data held on the SCB Mona microdata server, where the scale and structural complexity of social networks exceed the practical limits of existing general-purpose tools.

1.1 Why Threadle? Why not igraph?

igraph is an excellent and widely-used network analysis library, and for many purposes it remains the right tool to use. However, it runs into fundamental limitations when working with population-scale register data:

- **No native multilayer support**
igraph treats a network as a single graph. Representing multiple relational domains (co-workers, co-residents, schoolmates, family members) either requires maintaining separate graph objects and managing node identity across them manually, or attaching type attributes to each edge to indicate what kind of relation it represents.
- **2-mode (bipartite) data must be projected**
igraph has no native representation for bipartite (2-mode) networks as first-class objects. To store persons and workplaces in igraph, both are stored as the same type of object, where a node attribute indicates whether the node is a person or a workplace. To query whether two individuals share a workplace, these bipartite constructs must first be converted into their 1-mode representation, an operation that materializes all pairwise co-affiliation edges and can expand memory requirements by several orders of magnitude. For a dataset with 10 million people and a million of possible affiliations, such a projection is simply not feasible in standard RAM.
- **No native node attribute storage**
Node attributes in igraph are stored as separate vectors attached to the graph, residing in the same flexible, yet inefficient, memory space as R. At population scale, this become very memory-inefficient.
- **Performance limits**
Whereas igraph is very fast and memory-efficient for single-layer binary networks, for which it is designed for, it runs into significant memory bottlenecks when used for multilayer, mixed-mode, and attributed large networks.

1.2 How Threadle addresses these challenges

Threadle was designed from the ground up for the characteristics of networks derived from register data:

- **Native multilayer architecture**
A Threadle Network can hold multiple relational layers simultaneously, each with its own properties (1-mode or 2-mode, directed or undirected, binary or valued), all layers sharing the same Nodeset. Most queries and analyses can be done on individual layers, all layers, or a set of these.

- **Pseudo-projection for 2-mode data**

Threadle stores 2-mode (bipartite) relations as true hyperedges: a person is affiliated to a workplace, not connected to every co-worker individually. Queries such as “are these two people connected?” or “who are this person’s co-workers?” work directly on the hyperedge structure, as if the data were projected, without ever materializing the projection. This saves enormous amounts of memory, at the expense of typically insignificant extra amounts of compute.

- **Native typed node attributes**

Node attributes (e.g., age, sex, region, income category, and so on) are stored directly on nodes in the Nodeset, typed (Int, Float, Char, Bool, String), and available for filtering, summarization, and importing. Once a node attribute is defined, memory is only spent when an attribute value is assigned to a particular node. For the String attribute type, would-be duplicate values are recycled: if several nodes have “Carl” as their ‘name’ node attribute, the string is only stored once in memory, with the node attribute values instead pointing to this same location.

- **Designed for scale**

Threadle is implemented in .NET/C# with very memory-efficient data structures optimized for networks with millions of nodes, billions of 1-mode edges, and the equivalent of several trillions of projected 2-mode edges.

1.3 The Threadle ecosystem

Threadle consists of three components:

- **Threadle.Core**

This contains the underlying data model, processing engine, analytical functions, utilities, and file I/O library

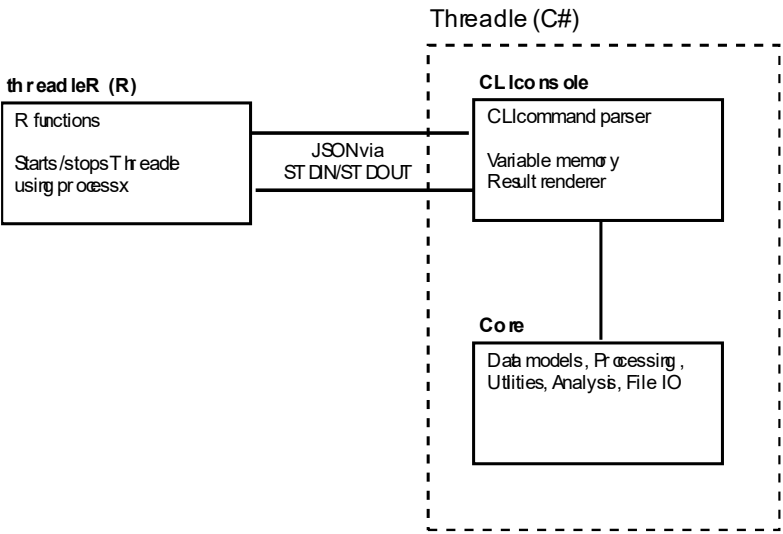
- **Threadle.CLIconsole**

A command-line interface that exposes Core’s functionality through a scripting language. Communication with the CLIconsole is either in human-readable text, or as JSON packages. On Mona, the CLIconsole runs invisibly in the background

- **threadleR**

An R package that communicates with CLIconsole and exposes all Threadle functionality through familiar R function calls. Communication with the CLIconsole is here done with JSON packages.

The overall architecture and organization of these three components are as follows:



On Mona, researchers interact exclusively with threadleR from within R. The CLIconsole is launched and managed automatically by threadleR. Direct use of the console is neither required nor possible on Mona.

If installing Threadle CLIconsole on a local machine, Threadle can be run directly as a console/terminal application. Communication (input and output) with the CLIconsole is then in human-readable (non-JSON) text, where the user enters command on the command line and receives output on the console/terminal. However, as the CLIconsole uses a rather simple scripting language, and as threadleR exposes all the CLIconsole commands, it is recommended to use threadleR from R also on local machines.

1.4 Funding and acknowledgements

Threadle was developed within the *The Complete Network of Sweden* research environment (netreg.se), funded by the Swedish Research Council (Vetenskapsrådet), grant 2024-01861. Threadle is developed and maintained by [Carl Nordlund](#), at the [Institute for Analytical Sociology](#), Linköping University, Sweden. The R library threadleR was developed by Yukun Jiao, graduate from the Master’s program in Computational Social Science at Linköping University, Sweden.

For more information, questions, and would-be support on Threadle, including suggestions on future extensions to Threadle, please contact carl.nordlund@liu.se

1.5 How to read this guide

This guide is written primarily for Mona researchers. All worked examples use R and threadleR. All threadleR functions are indicated like `this()`, and the full list of all threadleR functions are found in Section 9. For reference, the full CLI command reference is also provided in Section 9, but Mona users do not need to use CLI commands directly.

2 Core concepts

2.1 Nodesets and Networks

Every Threadle workflow involves two fundamental structures: a Nodeset and a Network.

A **Nodeset** is the set of nodes (also known as actors, vertices, entities), i.e. the set of all individuals (or other entities) in your dataset. Each node is identified by a unique unsigned integer, which in a register context is typically an anonymized personal identity number (PersonLopNr). The Nodeset is also where node attributes reside, such as demographic variables, derived node-centric measures, group memberships, and so on. As such, a Nodeset can be seen as a standard data table consisting of a set of individuals and their variable values.

A **Network** is the structure that contains one or more relational layers, each holding either a set of 1-mode relations or 2-mode affiliations. A Network does not duplicate the nodes: rather, each Network must refer to exactly one existing Nodeset. This means that a single Nodeset can be used by multiple Network structures.

In practice, for most register workflows, you will have one Nodeset and one Network per project, saved together as a pair of files. That said, Threadle can keep multiple Nodeset and Network structures simultaneously in memory, making it possible to immediately explore how the composition of one's alters in primary school relates to family formation patterns later in life.

2.2 Layers: 1-mode and 2-mode relationships

A layer is a set of relations within a Network, typically representing a particular kind of relationship. Each layer has a name and a set of fixed properties defined when the layer is created:

- **Mode**
Either **1** (unipartite: edges between nodes) or **2** (bipartite: affiliations between nodes and hyperedges).
- **Directionality (1-mode only)**
directed (asymmetric, e.g. “sent a message to”) or **undirected** (symmetric, e.g. “co-authored with”)
- **Value type (1-mode only)**
binary (edge exists or not) or **valued** (edge has a numeric weight, e.g. number of interactions)
- **Self-ties (1-mode only)**
Whether a node can be connected to itself

As all 2-mode affiliations are binary and symmetric by default, 2-mode layers do not use the latter three properties above.

In a register data context, typical 1-mode layers capture direct dyadic relationships between individuals. The typical example is kinship: parent-child ties, sibling ties, or marriage ties, all of which are recorded in the Swedish multi-generational register (FlerGen) and the civil status register (Civilandr). Note that kinship could encompass several types of ties, e.g. parent-child, siblings, spouses etc., and it is a deliberate research design decision whether to combine these into a single kinship layer or separate them into distinct relational layers of marriages, parent-

child, siblings, and so on. Separating them allows layer-specific queries (e.g. “who are this person’s siblings?”) but increases the number of layers to manage.

Typical 2-mode layers capture affiliations: individuals connect to workplaces (via the LISA register), residential buildings/properties (via GeoYYY), school classes, and so on. These are stored as hyperedges: each workplace or property is a named hyperedge, each with a collection of individuals affiliated to it.

Whereas 2-mode (bipartite) data represent many-to-many relationships, e.g. when an individual can be affiliated with several organizations, and an organization can have several individuals affiliated to it, 2-mode data in Mona are often one-to-many relationships. An individual typically has one workplace, one residential building, one neighborhood, at least per year. Threadle is, however, optimized for many-to-many bipartite relationships: it is thus perfectly fine that an individual is affiliated to multiple hyperedges (affiliations) in each 2-mode layer, for instance if a layer represents workplace affiliations stretching over a longer period.

A Network must have at least one layer before edges or affiliations can be added. When importing edges/affiliations from file, or adding them programmatically through threadleR code, the specific layer must exist (or have been created) beforehand.

2.3 The Pseudo-projection of 2-mode ties

The most important concept for researchers coming from igraph is Threadle’s handling of 2-mode data.

In a classical bipartite projection, you take a 2-mode network (individuals affiliated with organizations) and compute the corresponding 1-mode network. This is done by matrix multiplication, yielding an edge between every pair of individuals who share at least one organization, with edge weights equal to the number of shared organizations. With millions of individuals and hundreds of thousands of organizations, this projection creates billions if not trillions of edges and is not feasible to store in RAM.

Threadle takes a different approach. It stores the raw 2-mode structure, i.e. individuals affiliated with named hyperedges (organizations, school classes, households etc.), and answers 1-mode queries directly from that structure. When you thus ask “are person A and B connected?” and a 2-mode layer is involved in that query, Threadle checks whether they share at least one hyperedge, without building the projected graph. When you ask “who are person A’s co-workers?”, Threadle finds all persons affiliated to the same workplaces as person A.

This means that functions like `th_check_edge()`, `th_get_edge()`, `th_get_node_alter()`, `th_get_random_alter()`, and `th_get_random_edge()` work seamlessly on both 1-mode and 2-mode layers, querying 2-mode layers as if they were projected into their 1-mode equivalents. Whereas `th_get_edge()` returns the value of the edge on a 1-mode layer (which is 1 for ties in a binary layer), `th_get_edge()` on a 2-mode layer returns the number of affiliations (i.e. hyperedges) the two nodes share, which is exactly the valued projection weight, without materializing the full projected graph.

When you genuinely need a projected 1-mode layer from a 2-mode layer, Threadle provides `th_project_two_mode()` to do this. However, all querying and sampling tasks are already constructed with pseudo-projection: when determining the shortest path between two nodes, allowing for such paths to go through multiple layers, the `th_shortest_path()` function works seamlessly and equally well irrespective of whether the layers are 1-mode or 2-mode. As an analyst, you don’t have to consider whether a layer is 1-mode or 2-mode in that case.

2.4 Node attributes

Node attributes are typed variables stored directly in the Nodeset. Before setting or importing attribute data for nodes, the attribute must first be defined with a name and type using `th_define_attr()`. The exception is when loading a Nodeset from file, or importing additional attributes via `th_import_node_attributes()`: in both cases, the node attributes specified in the file's header are created automatically (see section 4.4.1 and 5.2 respectively).

Supported attribute types are:

Type	Description	Example values
Int	Integer (64-bit)	45, 0, -3
Float	Floating-point number (64-bit)	3.14, 0.5, -28.34
Char	Single character	m, f, o
Bool	Boolean	TRUE, FALSE
String	Text string	Norrköping, Carl, crocheting

Attributes can be set on nodes individually using `th_set_attr()`, or bulk-imported from a flat file using `th_import_node_attributes()`.

The flat file format for node attributes is a tab-separated file where the header row defines attribute names and their types using `attributename:Type` notation. The first column contains node IDs; its head cell is ignored. This file format is virtually identical to the Nodeset TSV file format (see section 9.3.1), the difference being that the top-left cell for the Nodeset TSV files contain the optional, more descriptive name of the Nodeset.

[ignored]	gender:Char	age:Int	region:String
100001	m	45	Stockholm
100002	f	38	Göteborg
100003	o	52	Malmö

If no type annotation is provided in the header, String is assumed. Rows whose node ID does not exist in the Nodeset are skipped by default (left-join behavior), but the optional `admissingnodes` argument in `th_import_node_attributes()` allows for adding newly mentioned nodes to the Nodeset that the attributes are being imported to.

Calculated attributes, such as degree centrality and component membership, are also stored as node attributes. Functions like `th_degree()` and `th_components()` write their results directly to the Nodeset as named attributes, which can then be exported or used for filtering.

2.5 File formats

Threadle uses its own file formats for saving and loading Nodesets and Networks:

Extension	Description	Use case
.tsv	Tab-separated text, human readable	Inspection, debugging, importing Nodesets
.tsv.gz	Gzipped TSV	Readable but compressed

<code>.bin</code>	Threadle binary format, compact, not human readable	Recommended for production
<code>.bin.gz</code>	Gzipped binary	When disk space is critical

The `.bin` format is recommended for production use on Mona. Threadle binary files are already compact: a 10-million node network with 16 node attributes and several layers containing millions of edges and affiliations typically occupy 1-2 GB on disk. Disk space is generally not a constraint on Mona, and `.bin` loads and saves significantly faster than `.bin.gz`. Use `.bin.gz` only when disk space is genuinely critical.

A Network file always contains a reference to the Nodeset file that it is using. When you load a Network file, its referenced Nodeset is loaded automatically (and automatically assigned to a variable as well): you should not load the Nodeset separately before the Network.

2.5.1 Mutable vs. packed layers

By default, layers are stored in a mutable format: edges can be added and removed at will. Once a layer is finalized and no further modifications are expected, it can be packed using `th_pack()`, which converts it to a more compact, immutable representation. Packing saves significant RAM for large layers. A packed layer can be unpacked again with `th_unpack()`, if modifications are later needed. All queries and analytical functions work equally well on packed and unpacked layers alike.

3 Installation and Setup

3.1 Using Threadle on Mona

Since June 2026, Threadle is pre-installed on the Mona server. Researchers access Threadle exclusively through the threadleR R package, from where the Threadle client is started and controlled. The CLIconsole application thus runs silently in the background without having to be installed or configured manually by a Mona user.

Although threadleR is (currently) not in the official CRAN repository, threadleR is available as an installable package on Mona. To start using threadleR, install threadleR once from within your Mona R environment, as well as the two libraries that threadleR uses: processx (for managing the Threadle process), and jsonlite (for machine-based communication between threadleR and the Threadle client).

```
install.packages(c("processx", "jsonlite", "threadleR"))
```

The install package step only needs to be done once.

When you want to work with Threadle functionality and use threadleR, start your R scripts with the following:

```
library(processx)
library(jsonlite)
library(threadleR)
```

```
th_start_threadle()
```

To verify that Threadle is available and ready:

```
# This should return TRUE
th_is_available()
```

At the end of your session, stop the Threadle process cleanly:

```
th_stop_threadle()
```

Although Threadle runs as a child process to your R session, meaning that Threadle will be shut down when the R session stops, it is still very much recommended to close down Threadle cleanly in the way above.

Note that stopping Threadle (`th_stop_threadle()`) will delete all Nodeset and Network data that is currently in your Threadle working space, even though you will still have ‘dead’ references to these structures in your R environment. Although you can start a new Threadle process again, using `th_start_threadle()`, that will then be a new empty Threadle instance.

3.2 Using Threadle outside Mona

Researchers who want to use Threadle on their own machine (Windows, macOS, or Linux) need to install both Threadle itself and threadleR.

3.2.1 Installing Threadle

There are two installation options available at www.threadle.dev:

- **Setup installer** (recommended for Windows):
Download and run the setup installer suitable for your Windows machine. This installs Threadle and automatically adds it to your system PATH, so that threadleR can find it without any additional configuration.
- **Standalone executables** (Windows, macOS, Linux):
Download the suitable pre-built binary directly and store it in a suitable location. In this case, Threadle is not added to PATH, and you will need to provide the path to the executable when starting threadleR (see below).

Alternatively, build from source via the [GitHub repository](#).

3.2.2 Installing and configuring threadleR

threadleR is currently not on CRAN. Install it from GitHub using the remotes package (which has to be installed if you don't have it already):

```
install.packages("remotes")
remotes::install_github("YukunJiao/threadleR")
```

When installing outside Mona, the two dependencies – processx and jsonlite – will be installed automatically.

If you installed Threadle using the setup installer (and it is on your PATH), start threadleR without any additional configuration:

```
library(threadleR)
th_start_threadle()
```

If you downloaded the standalone executable instead, provide the path explicitly when starting the Threadle process (works on macOS and Linux):

```
library(threadleR)
th_start_threadle(path = "/path/to/threadle")
# or "C:/path/to/threadle.exe" on Windows
```

Verify that Threadle is available:

```
th_is_available()
```

4 Building a Network from Register data

4.1 Overview of the pipeline

Threadle can only read files on disk, with no possibility to connect to databases directly. This limitation is for security reasons. The workflow for building a network from register data therefore involves writing intermediate flat files that Threadle imports. This is straightforward in practice and fits naturally into an R scripting workflow.

The recommended pipeline is to first build the Nodeset from a comprehensive register query, saving the resulting data frame to file in the format of a TSV Nodeset, which is then loaded into Threadle. A Network object is then created using this Nodeset, and relevant relational layers are created. The data for these relational layers are then imported, confined to the existing Nodeset. This avoids the `addmissingnodes` pitfall (see Section 4.4.3) and keeps the node universe clearly defined from the start.

1. SQL query (population + attributes) -> R data frame
2. Write nodeset TSV file with typed header (`lopnr`, `gender:Char`, `age:Int`, ...) to "nodeset.tsv" or similar
3. `th_start_threadle()` + `th_sync_wd()`
4. `ns <- th_load_file("ns", file = "nodeset.tsv", type = "nodeset")`
5. `net <- th_create_network("net", nodeset = ns)`
6. `th_add_layer()` for each relational domain
7. For each layer:
 - a. SQL query -> R data frame -> write edge/affiliation TSV
 - b. `th_import_layer(net, layername=..., file=..., addmissingnodes=FALSE)`
8. `th_save_file(ns, file="nodeset.bin")`
9. `th_save_file(net, file="network.bin")`
10. `th_stop_threadle()`

For subsequent sessions, the pipeline shortens to:

1. `net <- th_load_file("net", file = "network.bin", type = "network")`
2. [query, analyze, extend network and nodeset as needed]
3. `th_save_file(net)` and possibly also the 'net_nodeset' nodeset
4. `th_stop_threadle()`

4.2 Step 1: SQL query to R data frame

The first SQL query should define the full node universe, such as all residents in Sweden for a particular year, and combine it with the node attributes you need. A natural starting point could be the GeoYYYY tables (which records geographical data for the entire Swedish population), joined with background data from `bakgrundsdata` and year-specific labor market variables from `LISAYYYY_Individ`.

The result of this query becomes the nodeset TSV file: the first column is the anonymized personal identity number [`PersonLopNr`], and subsequent columns are named with the `name:Type` notation described in section 2.4.

```
# Init library for database access
library(RODBC)

# Create SQL connection (replace with your database)
conn <- odbcConnect("P0123")

# SQL query: Nodeset + attributes for the year 2017
query_ns <- "SELECT a.PersonLopNr,
  a.Kommun as 'kommun:Int',
  a.BostrutaX as 'BostrutaX:Int',
  a.BostrutaY as 'BostrutaY:Int',
  a.BostDESO as 'BostDESO:String',
  b.FodelseAr as 'birthyear:Int',
  b.Kon as 'gender:Char',
  b.LandKod as 'birthcountry:Int',
  c.DispInk04 as 'DispInk04:Int',
  NTILE(10) OVER (ORDER BY c.LoneInk) AS 'loneinkdecile:Int'
FROM Geo2017 a
INNER JOIN Bakgrundsdata b ON a.PersonLopNr = b.PersonLopNr
LEFT JOIN LISA2017_Individ c ON a.PersonLopNr = c.PersonLopNr"

# Query and store result in a dataframe
nodeset_df <- sqlQuery(conn, query_ns)

# SQL query: Parent-child ties
query_pc <- "SELECT PersonLopNr,
  LopNrFar as LopNrParent
FROM Flergen
WHERE LopNrFar IS NOT NULL
UNION ALL
SELECT PersonLopNr,
  LopNrMor as LopNrParent
FROM Flergen
WHERE LopNrMor IS NOT NULL"

# Query and store result in dataframe
parentchild_df <- sqlQuery(conn, query_pc)

# SQL query: Residential affiliations (in 2017)
query_res <- "SELECT PersonLopNr, LopNrFastighet FROM Geo2017"

# Query and store result in dataframe
res_df <- sqlQuery(conn, query_res)

# SQL query:Workplaces (except your special no-workplace id)
query_work <- "SELECT PersonLopNr,
```

```
KU1CfarLopNr
FROM LISA2017_Individ
WHERE KU1CfarLopNr<>[no-workplace-ID]"
```

```
# Query and store result in dataframe
workplace_df <- sqlQuery(conn, query_work)
```

4.3 Step 2: Writing temporary text file

In the second step, we save the relational data we obtained from the SQL databases into temporary files. In the code below, these files are placed in a folder called ‘/temp’.

For the nodeset file, the column names in the SQL result already carry the name:Type annotation, so `write.table()` with headers produces the correctly formatted TSV file for Nodesets directly:

```
write.table(nodeset_df, file = "temp/nodeset.tsv",
  sep = "\t", row.names = FALSE, quote = FALSE, na = "")
```

For edge/affiliation files, you can add column names:

```
# Write the (1-mode) parent-child edgelist data frame
write.table(parentchild_df, file = "temp/parentchild.txt", sep
  = "\t", row.names = FALSE, quote = FALSE, col.names = TRUE)
```

```
# Write the (2-mode) residential affiliation data frame
write.table(res_df, file = "temp/res.txt", sep = "\t",
  row.names = FALSE, quote = FALSE, col.names = TRUE)
```

```
# Write the (2-mode) workplaces affiliation data frame
write.table(workplace_df, file = "temp/workplaces.txt", sep
  = "\t", row.names = FALSE, quote = FALSE, col.names = TRUE)
```

Note on Node IDs: Node IDs in Threadle must be unsigned integers. Anonymized register IDs (PersonLopNr) satisfy this requirement directly. Hyperedge names in 2-mode layers are strings, so workplace IDs and property IDs can be used as-is.

4.4 Step 3: Importing temporary data files into Threadle

4.4.1 Creating the Nodeset and the Network

Since the nodeset TSV file already contains all nodes and their attributes, the Nodeset is created by loading the file directly rather than creating an empty nodeset and importing attributes separately.

`th_load_file()` takes a variable name as its first argument: this is the name by which the structure is known inside the Threadle backend. The return value is an R handle to that structure. Use short, descriptive names without spaces.

```
# First create the Nodeset by loading it from file
ns <- th_load_file("ns", file = "temp/nodeset.tsv", type =
  "nodeset")
# Then, create a Network that uses this nodeset
net <- th_create_network("net", nodeset = ns, name = "Swe2017")
```

The optional name argument to `th_create_network()` sets a human readable label stored in the file header; the first argument (“net”) is the backend variable name.

4.4.2 Adding a Layer

Before importing edges, the layer must exist. Define its properties at creating time: these cannot be changed later without removing and recreating the layer.

```
# A 1-mode undirected binary layer for parent-child ties
th_add_layer(net, layername = "parentchild", mode = 1,
  directed = FALSE, valuetype = "binary")

# A 2-mode layer for residence affiliations
th_add_layer(net, layername = "residences", mode = 2)

# A 2-mode layer for workplaces
th_add_layer(net, layername = "workplaces", mode = 2)
```

Note that other kinship tie types – siblings, married couples, cousins etc – could each have their own 1-mode layer, or be combined into a single kinship layer, all depending on the research question. Separate layers allow layer-specific queries; a combined layer simplifies multi-relational queries across all kinship types at once.

4.4.3 Importing the Edge/Affiliation file

```
# Import 1-mode edge list (parent-child relations)
th_import_layer(net, layername = "parentchild", file =
  "temp/parentchild.txt", format = "edgelist", header = FALSE,
  addmissingnodes = FALSE)

# Import 2-mode affiliation list (residences)
th_import_layer(net, layername = "residences", file =
  "temp/res.txt", format = "edgelist", header = FALSE,
  addmissingnodes = FALSE)

# Import 2-mode affiliation list (workplaces)
th_import_layer(net, layername = "workplaces", file =
  "temp/workplaces.txt", format = "edgelist", header = FALSE,
  addmissingnodes = FALSE)
```

On `admissingnodes`: Because the Nodeset was build first from the full population register, all expected `PersonLopNr` values are already present. Setting `admissingnodes = FALSE` (the default) means that any `nodeId` appearing in an edgelist file but not in the Nodeset will be silently skipped, which is the correct behavior here. Only set `admissingnodes = TRUE` when deliberately building a Nodeset from scratch with edgelist imports rather than from a population query.

For non-standard file layouts, column positions can be overridden using zero-based column indices. For 1-mode layers use `node1col`, `node2col` (and `valuecol` for valued layers); for 2-mode layers use `nodecol` and `affcol`:

```
# Non-standard column order: source in col 1, target in col 3,
# value/weight in col 6
th_import_layer(net, layername = "vallayer", file =
  "valedgelist.txt", format = "edgelist", header = TRUE,
  node1col = 1, node2col = 3, valuecol = 6,
  admissingnodes = FALSE)

# Non-standard 2-mode file: node ID in col 2, affiliation name in
# col 0
th_import_layer(net, layername = "workplace_alt", file =
  "workplace_alt.txt", format = "edgelist", header = TRUE,
  nodecol = 2, affcol = 0, admissingnodes = FALSE)
```

4.5 Step 4: Saving the Threadle file(s)

Now when Threadle holds both the Nodeset with all the attributes, and the Network with layers that points to that nodeset, it is time to save these structures in Threadle's own internal format. To do this, the Nodeset must be saved first, followed by the Network. This is to make sure that the Network indeed points to a Nodeset that has already been saved.

We save these files in a separate folder, here called `'/th_data'`:

```
th_save_file(ns, file = "th_data/nodeset.bin")
th_save_file(net, file = "th_data/network.bin")
```

The `.bin` format is recommended: it is fast to save and load, and population-scale files remain manageable in size. Use `.bin.gz` only if disk space is a genuine concern.

It is also possible to save the Nodeset in human-readable TSV format, and the Network in machine-readable `.bin` format. That is a suitable option if you want to retain the possibility to open the nodeset and its attributes as a data frame in R. However, it is always possible to save a previously loaded Nodeset in any format, including TSV format.

Note that the `th_save_file()` function is smart: based on the type of reference provided (`ns`: Nodeset, `net`: Network) and the file ending used (`.bin`, `.tsv`, `.bin.gz`, `.tsv.gz`), the function saves it in the correct format.

Finally, once having saved your Threadle data files, you can terminate the Threadle process:

```
th_stop_threadle()
```

4.6 Verification and clean-up

Having created a full-population Threadle network with its Nodeset and Network files, you can now start a new R script. Clear out the existing R session and write a new script, at the same file location as you create script had above:

```
library(threadleR)
th_start_threadle()
th_sync_wd()

net <- th_load_file("net", file = "th_data/network.bin",
  type = "network")
```

The associated Nodeset is loaded automatically alongside the Network, automatically named to 'net_nodeset'. Note that this will generate two structures (and two references): one to the Network (referenced by 'net' in R) and one to the Nodeset (referenced by 'net_nodeset' in R, which is named as <networkname>_nodeset).

Use `th_info()` to confirm nodeset properties and attributes and what layers are present:

```
# Should output details about the different layers and an
# estimate of the RAM memory they take up
th_info(net)

# Should output details about the Nodeset and its attributes
# and an estimate of the RAM memory they take up
th_info(net_nodeset)
```

The `th_pack()` function can be used to pack individual layers more, reducing their RAM memory at the cost of becoming immutable. They can also be packed immediately upon loading, using the optional pack argument in `th_load_file()`.

Once you have verified that the Threadle network creation worked as intended, you can then delete the temporary files that were used, i.e. the ones stored in the '/temp' folder in the example code above.

5 Extending an existing network

Once a network has been built and saved, it can be loaded and extended in later sessions: adding new layers from new register years, supplementing with additional relational domains, or attaching newly derived node attributes.

To load a previously saved network, we use the same code as in the last example:

```
library(processx)
library(jsonlite)
library(threadleR)
th_start_threadle()
th_sync_wd()

net <- th_load_file("net", file = "th_data/network.bin",
  type = "network")
```

5.1 Adding a new layer

Adding a new layer follows the same pipeline as the initial build:

1. SQL
2. Save as temporary data frame
3. `th_add_layer()` with suitable properties
4. `th_import_layer()` to load the saved data frame file into Threadle
5. Delete the temporary data frame

Always use `admissingnodes = FALSE` when extending an existing network: the node universe is already defined and should not be silently expanded by new register data.

5.2 Adding new node attributes

Given that we have a Nodeset 'net_nodeset' loaded in Threadle, we can add new node attributes from file, as shown in the pseudocode example below:

```
income_df <- sqlQuery(conn, "SELECT PersonLopNr,
  Sun2000niva AS 'Sun2000niva:Int'
  FROM LISA2017_Individ")
write.table(income_df, file = "incomeattr.txt", sep = "\t",
  row.names = FALSE, quote = FALSE, col.names = TRUE)
th_import_node_attributes(net_nodeset, file = "incomeattr.txt",
  admissingnodes = FALSE)
```

Derived attributes (degree, component membership) computed by Threadle are automatically added to the Nodeset and can be exported back to R by saving the Nodeset as a TSV file.

5.3 Packing finished layers

Once the construction of a layer is complete and ready for analysis, i.e. where no more edges will be added, the layer can be packed to save RAM:

```
th_pack(net, layername = "parentchild")
th_pack(net, layername = "residences")
th_pack(net, layername = "workplaces")
```

A packed layer can subsequently be unpacked using – you guessed it – `th_unpack()`.

5.4 Saving an updated network

Once a Nodeset and/or a Network has been updated, with new attributes, layers, nodes, edges etc., just save the structures again:

```
th_save_file(net_nodeset, file = "nodeset.bin")
th_save_file(net, file = "network.bin")
```

6 Querying a Threadle network

6.1 Inspecting a Network

```
th_info(net)      # Layers, layer properties, edge counts, memory
th_info(net_nodeset) # Nodes, attributes
th_preview(net)   # First few nodes/edges from each layer
th_preview(net_nodeset) # First few nodes and their attributes
th_i()           # Inventory of all currently existing structures
```

6.2 Node queries

For many of the node queries, either the Nodeset can be specified, or a Network structure for which the Nodeset it uses will be queried).

Do note that all the queries below can be assigned to R variables:

```
# Total number of nodes (either a nodeset or a network)
th_get_nbr_nodes(net)

# Retrieve node IDs (paginated, default limit is 1000)
th_get_all_nodes(net, offset = 0, limit = 100)

# Pick a random node
th_get_random_node(net)

# Get a single attribute value for a node
th_get_attr(net, nodeid = 12345, attrname = "birthyear")

# Get an attribute for multiple nodes at once
th_get_attrs(net, nodes = c(1234,2345,3456), attrname = "gender")

# Summary statistics for an attribute
th_get_attr_summary(net, attrname = "age")
# Returns mean, median, SD, min, max, Q1, Q3 for numeric types
# Frequency distribution for Char/String
# count/missing for all types
```

6.3 Neighbor and alter queries

These functions work on both 1-mode and 2-mode layers, where the automatic pseudo-projection is silently used for the 2-mode layers.

```
# All alters of a node in a specific layer
th_get_node_alters(net, nodeid = 12345, layernames = "kinship")
```

```
# All alters across multiple layers (deduplicated by default)
th_get_node_alters(net, nodeid = 12345,
  layernames = "workplaces;kinship;residences",
  unique = TRUE)

# For directed layers, specify direction:
th_get_node_alters(net, nodeid = 12345,
  layernames = "transfers",
  direction = "out") # "in", "out", or "both"

# Sample a random alter: useful in random walk and sampling
# workflows
th_get_random_alter(net, nodeid = 12345, layernames = "kinship")

# For 2-mode layers: get the hyperedges a node is affiliated with
th_get_node_hyperedges(net, layername = "workplaces",
  nodeid = 12345)

# For 2-mode layers: get nodes affiliated to a specific hyperedge
th_get_hyperedge_nodes(net, layername = "workplaces",
  hypername = "org_12345")
```

6.4 Edge queries

```
# Does an edge (or shared affiliation) exist?
th_check_edge(net, layername = "workplace",
  node1id = 12345, node2id = 23456)

# Edge value (for 2-mode layers: returns count
# of shared affiliations)
th_get_edge(net, layername = "workplace",
  node1id = 12345, node2id = 23456)

# Get all edges in a 1-mode layer (paginated)
th_get_all_edges(net, layername = "kinship",
  offset = 0, limit = 1000)

# Get all hyperedges in a 2-mode layer (paginated)
th_get_all_hyperedges(net, layername = "workplaces",
  offset = 0, limit = 1000)
```

6.5 Degree

`th_degree()` computes degree for all nodes in a layer and stores the result as a named node attribute. This is the recommended approach for large networks.

```
# Degree for all nodes (stored as node attribute 'deg_kinship')
th_degree(net, layername = "kinship", attrname = "deg_kinship",
  direction = "both")

# Inspect the degree distribution
th_get_attr_summary(net, attrname = "deg_kinship")

# Calculate degree for a single node
th_get_degree(net, nodeid = 12345, layernames = "kinship")

# Degree across multiple layers (each counted once by default)
th_get_degree(net, nodeid = 12345, layernames =
  "workplaces;kinship", unique = TRUE)
```

6.6 Components

```
# Compute component membership (stored as node attribute)
th_components(net, layername = "kinship",
  attrname = "comp_kinship")

# How many components, and what are their sizes?
th_get_attr_summary(net, attrname = "comp_kinship")

# Extract a subnet for a single component
# th_filter() and th_subnet() require a backend variable as
# their first argument
sub_ns <- th_filter("sub_ns", net,
  attrname = "comp_kinship",
  cond = "eq",
  attrvalue = "2")
sub_net <- th_subnet("subnet", net, nodeset = sub_ns)
th_get_nbr_nodes(sub_net)
```

Note on `th_filter()`: the `attrvalue` argument is always passed as a string, even for numeric attributes. Available conditions are “eq” (equal to), “ne” (not equal to), “gt” (greater than), “lt” (less than), “ge” (equal or greater than), “le” (equal or less than), “isnull” (true if null, `attrvalue` argument ignored), and “notnull” (true if not null, `attrvalue` argument ignored).

Note on `th_filter()` and `th_subnet()` signatures: Both functions take a backend variable name (a quoted string) as their first argument, followed by the source structure. The result is both registered in the Threadle backend under that name and returned as an R handle.

6.7 Shortest paths

```
# Shortest path between two specific nodes
th_shortest_path(net, node1id = 12345, node2id = 23456)
```

```
# Restrict to specific layers
th_shortest_path(net, node1id = 1234, node2id = 23456,
  layernames = "kinship;workplaces")
```

6.8 Density

```
# Exact density for a layer
th_density(net, layername = "kinship")

# Approximate density using sampling (recommended for very
# large layers, and especially 2-mode layers)
th_density(net, layername = "workplaces", samplesize = 200)
```

6.9 Filtering and subnetworks

```
# Create a sub-network: all individuals in Stockholm
# First argument is the backend variable name for the new
# nodeset
sthlm_ns <- th_filter("sthlm_ns", net,
  attrname = "region",
  cond = "eq",
  attrvalue = "Stockholm")

# Create a subnetwork from that nodeset
# First argument is the backend variable name for the new network
sthlm_net <- th_subnet("sthlm_net", net, nodeset = sthlm_ns)

th_get_nbr_nodes(sthlm_net)
th_density(sthlm_net, layername = "kinship")
```

Filtered sub-nodeset and sub-network are deep copies, completely independent of the original structures. The sub-network contains only edges between nodes in the filtered set.

7 Working with results in R

7.1 A note on data exports and microdata regulations

Threadle on Mona operates on microdata – individual-level register data – which is subject to very strict confidentiality and security rules. Exporting individual-level data, including network data (edges, nodes etc) out of Mona is strictly forbidden under standard Mona access conditions.

As all Threadle files on Mona are expected to contain microdata, such files (like the `nodeset.bin` and `network.bin` files we created in the examples earlier) may NEVER LEAVE THE MONA PLATFORM.

This means that functions such as `th_export_layer()` (which exports a layer to an edgelist) and `th_export()` (which exports a complete network to GEXF format for Gephi) are not appropriate for use with register data on Mona, except for analysis steps that remain within the Mona environment. Gephi is not available on Mona, so there is little point exporting any data to Gephi (GEXF) format.

The correct approach is to use Threadle to compute aggregate, non-individual measures – degree distributions, component statistics, inter-group distances – and store those results in R data structures. Aggregate findings that contain no individual-level information can then be exported from Mona in the usual way, as long as all traditional conditions are fulfilled. Consult with the MONA microdata group if unsure.

7.2 Getting node measures into R

The most natural way to bring computed node-level measures (degree centrality, component membership, and other attributes stored on the Nodeset) into R is to read the nodeset TSV file directly as a data frame. Since the nodeset in this format is written with proper headers, it should read cleanly with `read.csv2()`, `read.table()` etc.

As the nodeset for a full country population is large, this data frame could be very large, but it should be straightforward to work with this using standard R tools, such as `dplyr`, `data.table` etc. for aggregation, modelling, and further descriptives.

Note: after running analyses such as `th_degree()` or `th_components()`, save the nodeset as a TSV before reading it into R, so the derived attributes are written to the text file. Saving as `.bin` does not update the `.tsv`:

```
# ... doing analyses on existing network, storing node results
# as new attributes for each node in the 'ns' Nodeset.
# Save the nodeset in tsv (human-readable format):
th_save_file(ns, file = "nodeset.tsv")
# If the original format of the nodeset was .bin, save it again
# as .bin, just to make sure that the network structures pointing
# to this nodeset will still treat it as a .bin format nodeset.
th_save_file(ns, file = "nodeset.bin")
```

7.3 Sampling node measures inside Threadle

For exploring distributions without loading the full nodeset into R, Threadle's random sampling functions provide an efficient alternative:

```
# Sample degree values for 10k random nodes
sampled_degrees <- replicate(10000, {
  nodeid <- th_get_random_node(net)
  th_get_attr(net, nodeid = nodeid, attrname = "deg_kinship")
})

# Summarize in R
summary(as.numeric(sampled_degrees))
hist(as.numeric(sampled_degrees), main="Degree distribution
      (kinship, sampled)")
```

This keeps all individual-level data inside Threadle and produces only aggregate R objects.

7.4 Projecting 2-mode layers into equivalent 1-mode layers

When a materialized 1-mode projection is needed for a specific analysis (for example, an external algorithm that requires a standard adjacency structure like an edgelist):

```
# Valued projection: edge weight = number of shared affiliations
th_project_two_mode(net, layername = "workplaces",
  method = "count")

# Binary projection: 0: no shared affiliations, 1: shared
th_project_two_mode(net, layername = "workplaces",
  method = "binary")

# Newman-normalized projection (normalizes by hyperedge size)
th_project_two_mode(net, layername = "workplaces",
  method = "newman")
```

The original 2-mode layer is preserved when using `th_project_two_mode()` and a new 1-mode layer is created with a derived name. Note that at population scale, a materialized projection may create an enormous number of edges and consume substantial RAM. So, use this with care, and prefer the automatic pseudo-projection where possible.

8 Practical considerations on Mona

8.1 Working directories

It is a good idea to always synchronize Threadle's working directory with R's at the start of a session:

```
th_sync_wd()
```

Use `th_dir()` to verify the directory contents as seen by Threadle. All file paths in Threadle commands can be relative to the working directory.

8.2 File storage

Store flat files (intermediate TSV exports from SQL) and Threadle files (`.bin`, `.tsv`, and their Gzipped versions) within your allocated Mona project folder. As these files contain microdata, only store these files where you are allowed to do so in Mona. A clear directory structure helps:

```
project/
  data/          # Raw TSV and text files from SQL
  threadle/      # nodeset.bin, network.bin etc
  output/        # Aggregate R results for further analysis
```

8.3 Memory management

Use `th_info()` to check how much RAM your current structures are consuming – the output shown there are calculated estimates for the Nodeset and each Network layer. This is useful for planning whether packing or subnetworking is needed before running memory-intensive analyses.

- Pack layers that are completely built (`th_pack()`) before running analyses – this converts mutable layers to compact immutable storage and can substantially reduce RAM usage.
- Load Network files with `pack = TRUE` when opening a network that you do not plan to modify.
- Use `.bin` for all Threadle files. At population scale, `.bin` files are already compact and `.bin` loads and saves significantly faster than `.bin.gz`.

8.4 Long-running operations (imports, random walkers etc)

Importing population-scale layers (tens of millions of affiliations) into Threadle from file takes time. Structure your build script so that the full import-and-save sequence runs as a single uninterrupted script.

The default timeout for `threadleR` to wait for a response from Threadle is 1800 seconds (30 minutes). For very large imports and other time-consuming operations (such as the `rwfpt` random walker), you might have to increase this before starting:

```
options(threadle.timeout = 7200)    # 2 hours
```

```
th_start_threadle()
```

8.5 Verbosity

Threadle and threadleR can be quite verbose in its responses. If you do a lot of iterative operations to modify and create a network at the node and edge level, this could result in quite a lot of output and unnecessary json communication from Threadle. It is possible to toggle how verbose Threadle (and thus threadleR) is with the `th_setting()` function:

```
th_setting(name = "verbose", value = FALSE)
```

8.6 Reproducibility

Set a random seed to any number of your choosing before any stochastic operations (picking random nodes, random walkers, random network generation, sampling etc):

```
th_random_seed(seed=6031769)
```

8.7 Session hygiene

Always stop Threadle cleanly at the end of a session:

```
th_stop_threadle()
```

Note that this will still keep any would-be references to Threadle objects in R (i.e. what you see in the environment tab in RStudio). These references will now be ‘dead’, as there are no corresponding Threadle structures available anymore. It is thus a good idea to also delete these references in R when you stop the Threadle process.

9 Reference

9.1 threadleR function reference

The tables below list all exported functions in threadleR (version 0.5.1), separated by type of function. Run `getNamespaceExports("threadleR")` or `ls("package:threadleR")` for the definite live list in your installed version.

Note that the references below only contain the function and a description, excluding the arguments and their default values. For exact argument names and defaults, use `?th_function_name` in R.

9.1.1 Session management

Function	Description
<code>th_start_threadle()</code>	Launch the Threadle CLI process. Optionally provide <code>path</code> if the executable is not on your system <code>PATH</code> .
<code>th_stop_threadle()</code>	Terminate the Threadle process cleanly. Always call this at the end of a session.
<code>th_sync_wd()</code>	Synchronize Threadle's working directory to R's current <code>getwd()</code> . Call this at the start of every session before using relative file paths.
<code>th_is_available()</code>	Check whether the Threadle executable can be found. Useful for verifying a new installation.
<code>th_i()</code>	Print an inventory of all structures (nodesets, networks) currently loaded in the Threadle backend.
<code>th_cmd()</code>	Low-level escape hatch for sending raw CLI commands not yet wrapped by a threadleR function.
<code>th_stage_examples_to_wd()</code>	Copy bundled example files into the current working directory so you can run the package examples locally.

9.1.2 Structure management

Function	Description
<code>th_create_nodeset()</code>	Create an empty nodeset in the Threadle backend. The first argument is a quoted

	backend variable name. Optionally pre-populate with a given number of nodes.
<code>th_create_network()</code>	Create a new network built on an existing nodeset. The first argument is a quoted backend variable name. The network initially has no layers.
<code>th_load_file()</code>	Load a nodeset or network from a <code>.tsv</code> , <code>.tsv.gz</code> , <code>.bin</code> , or <code>.bin.gz</code> file. The first argument is a quoted backend variable name; specify <code>type = "nodeset"</code> or <code>type = "network"</code> . Use <code>pack = TRUE</code> to load layers in compact immutable form when no further edge additions are planned — this saves substantial RAM. When loading a network, the associated nodeset is loaded automatically.
<code>th_save_file()</code>	Save a nodeset or network to file. Saving as <code>.bin</code> does not update any <code>.tsv</code> file — if you need the TSV refreshed (e.g. to read updated attributes back into R), save to <code>.tsv</code> explicitly as a separate call.
<code>th_load_examples()</code>	Load one of the bundled example networks (<code>"mynet"</code> or <code>"lazega"</code>) for exploration and testing.
<code>th_delete()</code>	Delete a structure from the Threadle backend and free its memory.
<code>th_delete_all()</code>	Delete all currently loaded structures.
<code>th_info()</code>	Print metadata about a structure: node count, layer names, edge counts per layer, attribute definitions, and memory usage. Use this to monitor RAM consumption before running memory-intensive analyses.
<code>th_preview()</code>	Print the first few nodes and edges from a structure — useful for a quick sanity check after loading.

9.1.3 Layer management

Function	Description
----------	-------------

<code>th_add_layer()</code>	Add a new relational layer to a network. Properties (mode, directionality, value type) are set at creation and cannot be changed later without removing and recreating the layer.
<code>th_remove_layer()</code>	Remove a layer and all its edges from a network.
<code>th_clear_layer()</code>	Remove all edges from a layer while keeping the layer definition.
<code>th_pack()</code>	Convert a layer to compact immutable storage, freeing significant RAM. Use once a layer is finalized and no further edge additions are planned. If no <code>layername</code> is given, packs all layers.
<code>th_unpack()</code>	Convert a packed layer back to mutable storage so edges can be added or removed again.

9.1.4 Data import and export

Function	Description
<code>th_import_layer()</code>	Import edges or affiliations into a layer from a flat file. Supports 1-mode edge lists (two node ID columns, optional value column) and 2-mode affiliation lists (node ID column, hyperedge name column). Column positions are zero-based and can be overridden. Set <code>admissingnodes = FALSE</code> (the default) when the nodeset is already fully defined — any node ID in the file that is not in the nodeset will be silently skipped.
<code>th_import_node_attributes()</code>	Import node attributes from a tab-separated file with a typed header (<code>attrname:Type</code>). As with layer import, rows whose node ID is absent from the nodeset are skipped by default.
<code>th_export_layer()</code>	Export a layer to a flat edge-list file. Not appropriate for register microdata on Mona — use only for aggregate or synthetic data.

<code>th_export()</code>	Export a network to GEXF format for use in Gephi. Not available on Mona.
--------------------------	--

9.1.5 Node operations

Function	Description
<code>th_add_node()</code>	Add a single node to a nodeset or network.
<code>th_remove_node()</code>	Remove a node and all its edges from a structure.
<code>th_define_attr()</code>	Define a new typed node attribute on a nodeset. Must be called before values can be set or imported. Supported types: <code>int</code> , <code>float</code> , <code>char</code> , <code>bool</code> , <code>string</code> .
<code>th_undefine_attr()</code>	Remove an attribute definition (and all stored values) from a nodeset.
<code>th_set_attr()</code>	Set an attribute value for a single node.
<code>th_remove_attr()</code>	Remove the attribute value for a single node (sets it to missing).
<code>th_get_attr()</code>	Retrieve the attribute value for a single node.
<code>th_get_attrs()</code>	Retrieve attribute values for a vector of nodes in a single call.
<code>th_get_attr_summary()</code>	Summary statistics for an attribute across all nodes. Returns mean, median, SD, min, max, quartiles for numeric types; a frequency table for <code>Char</code> and <code>String</code> ; counts and missing for all types.
<code>th_generate_attr()</code>	Assign random values to an attribute for all nodes — useful for generating synthetic test data.
<code>th_filter()</code>	Create a new sub-nodeset containing only nodes matching a condition on an attribute (e.g. <code>region == "Stockholm"</code>). The first argument is a quoted backend variable name for the new nodeset. The <code>attrvalue</code> is always passed as a string even for numeric attributes. Available conditions: <code>"eq"</code> , <code>"ne"</code> , <code>"gt"</code> , <code>"lt"</code> , <code>"ge"</code> , <code>"le"</code> , <code>"isnull"</code> , <code>"notnull"</code> .

9.1.6 Edge and affiliation operations

Function	Description
<code>th_add_edge()</code>	Add a single 1-mode edge between two nodes.
<code>th_remove_edge()</code>	Remove a single 1-mode edge.
<code>th_add_hyper()</code>	Add a named hyperedge to a 2-mode layer, optionally with an initial set of affiliated nodes.
<code>th_remove_hyper()</code>	Remove a hyperedge and all its affiliations.

<code>th_add_aff()</code>	Add an affiliation between a single node and a hyperedge in a 2-mode layer.
<code>th_remove_aff()</code>	Remove an affiliation between a node and a hyperedge.
<code>th_check_edge()</code>	Check whether an edge exists (1-mode) or whether two nodes share at least one affiliation (2-mode). Works directly on the stored hyperedge structure — no projection required.
<code>th_get_edge()</code>	Get the edge value for a 1-mode layer, or the number of shared affiliations for a 2-mode layer. The 2-mode result equals the valued projection weight without materializing the projection.
<code>th_get_all_edges()</code>	Retrieve all edges in a 1-mode layer as a paginated result.
<code>th_get_all_hyperedges()</code>	Retrieve all hyperedge names in a 2-mode layer as a paginated result.
<code>th_get_random_edge()</code>	Sample a random edge from a layer.

9.1.7 Node queries

Function	Description
<code>th_get_nbr_nodes()</code>	Return the total number of nodes in a structure.
<code>th_get_all_nodes()</code>	Retrieve all node IDs as a paginated result.
<code>th_get_random_node()</code>	Sample a random node ID. Useful as the starting point for random-walk and snowball sampling workflows.
<code>th_get_nodeid_by_index()</code>	Retrieve the node ID at a given index position (zero-based). Useful for iterating over all nodes when combined with <code>th_get_nbr_nodes()</code> .
<code>th_get_node_alterers()</code>	Retrieve all alters (neighbors) of a node across one or more layers. For 2-mode layers, returns all nodes sharing at least one affiliation — the pseudo-projection. Multiple layer names are separated by semicolons. Set <code>unique = TRUE</code> to deduplicate alters appearing in more than one layer.

<code>th_get_random_alter()</code>	Sample a single random alter of a node. Supports multi-layer sampling, directional filtering, and weighted or balanced selection.
<code>th_get_node_hyperedges()</code>	For a 2-mode layer, return all hyperedge names that a given node is affiliated to.
<code>th_get_hyperedge_nodes()</code>	For a 2-mode layer, return all nodes affiliated to a given hyperedge.
<code>th_get_degree()</code>	Return the degree of a single node, optionally across multiple layers.

9.1.8 Analysis and transformations

Function	Description
<code>th_degree()</code>	Compute degree for all nodes in a layer and store the result as a named node attribute on the nodeset. This is the recommended approach at population scale — the result is not returned to R but is available for summary, filtering, and export.
<code>th_components()</code>	Compute connected component membership for all nodes and store it as a named node attribute. Each unique value identifies one component; use <code>th_get_attr_summary()</code> to inspect the size distribution.
<code>th_density()</code>	Compute the density of a layer. For very large layers, use the <code>samplesize</code> argument to get an efficient approximation rather than an exact count.
<code>th_shortest_path()</code>	Find the shortest path between two specific nodes, optionally restricted to named layers. Returns the path length and the sequence of nodes.
<code>th_shortest_paths()</code>	Compute all-pairs shortest paths and aggregate results by a node attribute (e.g. mean distance between regions). The first argument is a quoted backend variable name for the result. Warning: runs in $O(N \times (N + E))$ time — only feasible for small networks or subnetworks.

<code>th_project_two_mode()</code>	Materialize a 2-mode layer into a 1-mode projection. Methods: "count" (edge weight = shared affiliations), "binary", or "newman" (normalized by hyperedge size). The original 2-mode layer is preserved. At population scale this can produce an enormous number of edges — prefer the pseudo-projection for querying wherever possible.
<code>th_symmetrize()</code>	Symmetrize a directed 1-mode layer using a specified aggregation method (max, min, average, sum, etc.).
<code>th_dichotomize()</code>	Convert a valued 1-mode layer to binary by applying a threshold condition.
<code>th_subnet()</code>	Create a new subnetwork containing only the nodes in a given nodeset (typically produced by <code>th_filter()</code>). The first argument is a quoted backend variable name for the new network. Only edges between nodes present in the nodeset are retained.
<code>th_generate()</code>	Generate a random network layer using a standard model (Erdős–Rényi, Watts–Strogatz, Barabási–Albert, or random 2-mode). Useful for null model comparisons.
<code>th_rwdistances()</code>	Compute inter-category distances using a random walker method. Results are aggregate (mean crossing times between attribute categories) and do not contain individual-level data — suitable for export from Mona. The first argument is a quoted backend variable name for the result.
<code>th_rwfpt()</code>	Compute inter-category distances using first-passage times. Like <code>th_rwdistances()</code> , results are aggregate and Mona-exportable. The first argument is a quoted backend variable name for the result.

9.1.9 Utilities

Function	Description
----------	-------------

<code>th_dir()</code>	List the contents of a directory as seen by Threadle. Use to verify that files are where Threadle expects them.
<code>th_set_workdir()</code>	Manually set Threadle's working directory. Prefer <code>th_sync_wd()</code> in most workflows.
<code>th_get_workdir()</code>	Return Threadle's current working directory.
<code>th_sync_wd()</code>	Set Threadle's working directory to match R's <code>getwd()</code> . Call at the start of every session.
<code>th_random_seed()</code>	Set the random seed for all stochastic Threadle operations (random walkers, random network generation, random sampling). Set this before any such operations for reproducible results.
<code>th_load_script()</code>	Execute a pre-written CLI script file. Useful for long-running build jobs that you want to keep outside your R script.
<code>th_setting()</code>	Toggle an internal Threadle setting by name.

9.2 Threadle CLI command quick reference

The following table lists all Threadle CLI commands. Mona researchers do not need to use these directly: all functionality is available through the `threadleR` functions in section 9.1. The CLI reference is provided for completeness and for users running Threadle outside Mona. Full specifications for each command are available at www.threadle.dev/?page=cli

Command	Description
Session	
<code>exit</code>	Exit the CLI console
<code>about()</code>	Display version and metadata
<code>help()</code>	List all commands or get help on a specific command
<code>i()</code>	Inventory ¹ of loaded structure
Structure management	
<code>createnodeset()</code>	Create an empty nodeset
<code>createnetwork()</code>	Create a network from a nodeset

¹ This is of course a tribute to old-school text adventures, where the 'I' command shows the inventory one is currently carrying.

loadfile()	Load nodeset or network from file
savefile()	Save nodeset or network to file
delete()	Delete a structure
deleteall()	Delete all structures
info()	Metadata about a structure
preview()	Preview nodes/edges
Layer management	
addlayer()	Add a layer
removelayer()	Remove a layer
clearlayer()	Remove all edges from a layer
pack()	Convert layer to compact immutable storage
unpack()	Convert layer back to mutable storage
Data import/export	
importlayer()	Import edges from flat file
importnodeattributes()	Import node attributes from TSV
exportlayer()	Export layer to flat file
export()	Export to external format (GEXF)
Node operations	
addnode()	Add a node
removenode()	Remove a node
defineattr()	Define a node attribute
undefineattr()	Remove an attribute definition
setattr()	Set an attribute value
removeattr()	Remove an attribute value
getattr()	Get an attribute value
getattrs()	Get attribute values for multiple nodes
getattrsummary()	Summary statistics for an attribute
generateattr()	Generate random attribute values
filter()	Create filtered sub-nodeset
Edge and affiliation operations	
addedge()	Add a 1-mode edge
removeedge()	Remove a 1-mode edge
addhyper()	Add a hyperedge

<code>removehyper()</code>	Remove a hyperedge
<code>addaff()</code>	Add a node–hyperedge affiliation
<code>removeaff()</code>	Remove an affiliation
<code>checkedge()</code>	Check if edge or shared affiliation exists
<code>getedge()</code>	Get edge value
<code>getalledges()</code>	All edges in a 1-mode layer
<code>getallhyperedges()</code>	All hyperedge names in a 2-mode layer
<code>getrandomedge()</code>	Sample a random edge
Node queries	
<code>getnbrnodes()</code>	Node count
<code>getallnodes()</code>	All node IDs
<code>getrandomnode()</code>	Random node ID
<code>getnodeidbyindex()</code>	Node ID by index
<code>getnodealters()</code>	Alters of a node
<code>getrandomalter()</code>	Random alter
<code>getnodehyperedges()</code>	Hyperedges a node belongs to
<code>gethyperedgenodes()</code>	Nodes in a hyperedge
<code>getdegree()</code>	Degree for a single node
Analysis	
<code>degree()</code>	Degree for all nodes
<code>components()</code>	Component membership
<code>density()</code>	Layer density
<code>shortestpath()</code>	Shortest path between two nodes
<code>shortestpaths()</code>	All-pairs shortest paths
<code>projecttwomode()</code>	Materialize 2-mode projection
<code>symmetrize()</code>	Symmetrize a directed layer
<code>dichotomize()</code>	Dichotomize a valued layer
<code>subnet()</code>	Create subnetwork
<code>generate()</code>	Generate random network
<code>rwdistances()</code>	Random walker distance measure
<code>rwfpt()</code>	First-passage-time distance measure
Utilities	
<code>dir()</code>	List directory contents

setwd()	Set working directory
getwd()	Get working directory
randomseed()	Set random seed
loadscript()	Execute a CLI script
setting()	Toggle an internal setting

9.3 File format reference

9.3.1 Nodeset TSV format

The nodeset flat file is a tab-separated file. The first row is a header where each column (after the first) defines an attribute using `attributename:Type` notation. The first column contains node IDs; its header cell contains the more descriptive, human-readable name of the nodeset. Supported attribute types are: Int, Float, String, Bool, Char. If no type is specified, String is assumed. Do note that Threadle has no in-built functionality for subsequent typecasting, e.g. converting an attribute of type Int to String or vice versa, so make sure that the attribute types are accurate. Also: although String uses a clever cloning check to never store the same attribute value more than once, String takes up the most memory (32 bits plus the actual string), compared to the other attribute types (32 bits each).

people	gender:Char	age:Int	income_decile:Int	region:String
100001	m	45	7	Stockholm
100002	f	38	4	Göteborg
100003	o	52	9	Malmö

9.3.2 Network TSV format

A Network can also be stored as a human-readable tab-separated format, which then can be inspected in a text editor. Although it is possible to create this file from scratch in a text editor, this is better seen as a read-only format.

Starting off with an initial section where the name and the nodeset is specified, the Network TSV file then stores each layer separately. Note that lines starting with a hashtag (#) are not parsed.

```
# Network Metadata
Name: My network
NodesetFile: mynet_nodes.tsv
```

```
# Layer
LayerMode: 1
LayerName: friends
Directionality: directed
ValueType: binary
Selfties: false
```

```

1      2      4      8      17
2      16     17     22     26
...

```

```

# Layer
LayerMode: 2
LayerName: Workplaces
E1      1      2      7
E2      1      2      3
E3      3      7      8     16
...

```

9.3.3 Edge list format (1-mode)

Two columns: source node ID, destination node ID. For valued layers, a third column holds the edge weight. These default to the first 3 columns but can be set when importing. No header by default; set `header = TRUE` in `th_import_layer()` if one is present. Tab-separated by default, but can also be adjusted when importing.

```

100001 100002
100001 100003
100002 100004

```

9.3.4 Affiliation list format (2-mode)

Two columns: node ID, hyperedge name. Hyperedge names are strings. No header by default. Columns, value-separator, headers etc. can be set when importing.

```

100001 org_12345
100002 org_12345
100002 org_67890

```

9.3.5 Threadle binary (.bin) format

The .bin and .bin.gz formats are Threadle’s own compact binary formats. They are not human-readable and should not be edited externally. Use .tsv for human-readable exports. .bin is recommended for production storage: it is fast to load and save, and population-scale files remain manageable in size. Use .bin.gz only when disk space is genuinely critical.

The Magic Bytes (first 4 bytes) in a binary Threadle file is “TNTW” for networks, and “TNDS” for nodesets. This is followed by a single byte with the format version. The current binary file format version in Threadle is 3: would-be future extensions will however remain backward compatibility to previous file format versions.

For details about the binary formats, please consult the source code for the binary writers:

- Writing a Nodeset structure to binary format:
<https://github.com/carlnordlund/Threadle/blob/master/Threadle.Core/Utilities/FileSerializerBin.cs#L409>

- Writing a Network structure to binary format:
<https://github.com/carlnordlund/Threadle/blob/master/Threadle.Core/Utilities/FileSerializerBin.cs#L493>

For more details about the Threadle file formats (.bin and .tsv), please contact the developer (carl.nordlund@liu.se).

9.4 Glossary

Affiliation. A membership relation between a node and a hyperedge in a 2-mode layer. Stored directly; not derived from projection.

Bipartite network. A network with two distinct sets of nodes, where edges connect only across sets (e.g. persons and organizations). Represented in Threadle as a 2-mode layer.

Hyperedge. A named entity in a 2-mode layer representing a shared context — a workplace, school class, household, or event. Nodes are affiliated to hyperedges, not to each other directly.

Layer. A relational dimension within a Network, with fixed properties (mode, directionality, value type). A Network can contain multiple layers, mixing 1- and 2-mode layers with different properties.

Mutable/immutable layer. A mutable layer supports adding and removing edges. An immutable (packed) layer is stored more compactly but cannot be modified. Use `th_pack()` and `th_unpack()` to convert between them.

Network. Contains a collection of layers, each holding different kinds of edges. A Network references a specific Nodeset structure that must be in the Threadle workspace.

Nodeset. The universe of nodes in Threadle, each identified by a unique unsigned integer ID. Node attributes are stored in the Nodeset.

Pseudo-projection. Threadle's ability to query 2-mode data as if it were projected to 1-mode, without materializing the projected graph. Functions like `th_check_edge()`, `th_get_edge()`, and `th_get_node_alters()` use the pseudo-projection transparently.

Projection. A derived 1-mode network computed from a 2-mode network, where two nodes are connected if they share at least one affiliation. Threadle can materialize this with `th_project_two_mode()`, but pseudo-projection is usually sufficient for querying.

Unipartite network. A network where all nodes belong to the same set and edges connect nodes to other nodes. Represented in Threadle as a 1-mode layer.