

A software tool for Bayes net development

Alan M. Kalet

May 9, 2015

1 Introduction

In this article I outline the details of a software tool designed to give users a simple graphical interface for extracting dependency network topologies from dependency layered ontologies. What is important about this tool and how it is different from other software like Hugin or Netica is that it derives knowledge directly from a dependency layered ontological knowledge base and therefore does not require significant user input towards developing a topological structure. There are many features which this tool does *not* have, such as the ability to create conditional probability tables or perform any sort of learning algorithms on data. However, topologies created with this tool can be exported and read into other systems such as Hugin for those tasks, making this tool complementary to other software systems for Bayes net development.

2 Goals and architecture

I developed a user interface prototype, the “Bayesian Network Domain Explorer” (BNDE), with a few key goals. The first goal is to allow for users to reasonably navigate an ontology class-subclass hierarchy and select concepts of interest. Secondly, the software needs to generate and display the resulting networks and information about those networks, as built from the ontological knowledge base and user selections. Lastly, the software has to provide user controlled network pruning and downloading of resulting network topologies in useful formats for further use. In accordance with these goals, I also seek *platform independence*, that is, to limit the software’s dependence on operating system, hardware, or other device types.

To meet these goals I use the `shiny` package for the R statistical programming language. There are several advantages to using `shiny` for this

software. **Shiny** is a web application framework for **R** and can be run on a local server tied to **R** processes [1]. This allows for fairly simple development of web-based user interface tools while retaining access to powerful computational **R** libraries. Web-based also means that many widgets, buttons, layouts, graphical methods, javascript and css stylesheets already available via open source projects such as bootstrap [2] and other webtools can be utilized. Because shiny applications are web-based, they are also fairly platform independent. Shiny applications work on almost all common contemporary web browsers, allowing users to apply their local browser settings, configurations, and schemes independent of the application. Shiny also uses a reactive programming model which effectively performs real-time updating of graphical output (or any other reactive functionality) based on user input.

The essential model of reactivity in shiny applications is shown in Figure 1. A thorough overview of reactivity is given in RStudio’s article on reactivity [3]. I present a shortened version for context. Client-side inputs are tied to server-side outputs (or any other descendant server-side reactive functions). When the input changes, the descendant functions are invalidated. After all invalidations are finished, the outputs (graphical display, dynamic UI, etc) are flushed and redrawn using the new user inputs. This event scheduling occurs on the order of milliseconds. The shiny server checks for input changes frequently—effectively updating changes with high responsiveness. If an underlying server-side function needs to run for some time, there can be delays in re-drawing the display. Fortunately, there are many available features in the development set to provide client-side user feedback during server-side computing processes.

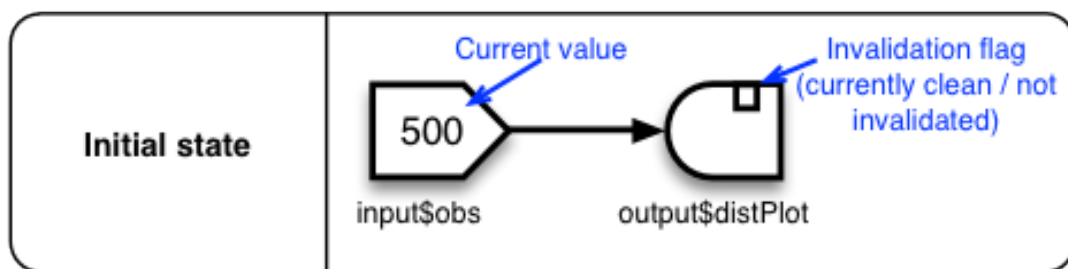


Figure 1: Shiny reactivity-initial state with input (`input$obs`) and a valid output (`output$distPlot`). The output `distPlot` gets invalidated, flushed, and re-executed upon changes to `input$obs`

In order to meet the desired specifications above I employed the `rrdf`, `RHugin`, `shinysky`, `shinyBS`, and `shinyIncubator` packages as well as a

set of developer defined functions. The core functionality relies on a few computational tasks:

1. parse description logic ontology
2. search for dependency paths between nodes
3. prune the network nodes

Description logic ontologies can be read into **R** as a set of rdf triples, however, they are represented as a Java class. To parse the ontology into a more easily searched matrix of dependency triples, sparql queries of the form below are employed to extract all the terms in the ontology which have dependence on other terms. All terms in the new triple sets (for each type of dependency) are SELECTed and collated to form a 3xN matrix.

```
CONSTRUCT {?object2 ro:dependsOn3 ?object1}
WHERE {?object1 rdfs:subClassOf ?restriction.
?restriction owl:onProperty ro:dependsOn3.
?restriction owl:someValuesFrom ?object2.
?restriction ?restrictionPredicate ?object2.}
```

From the larger ontology, the software now has a subset of triples to operate on. Here, a level-order non-binary search method (breadth-first search) is used to traverse the set of triples to discover pathways between user defined subset of terms. Marrying these core operations to the user interface requires assigning a series of reactive elements. A broad overview of the architecture which links UI inputs/outputs between the server, client, and client local filesystem is shown in Figure 2. The user interface (UI) is tied to **R** processes running on the server—gathering inputs and sending back outputs for displays or requests for uploads/downloads. There are ontologies loaded on the server filesystem, but options are available for users to upload their own ontology to the server and access it using the UI running in the client's local browser.

Because the core functionality of this tool is to perform logical deductive reasoning among dependent terminologies and express their topological connections, it can technically be applied among non-probabilistic variables as well, such as those found in computer programs. Therefore, if the application is given an ontology of its own functional dependencies among variables and reactive elements, it can describe itself! Figure 3 is a directed graph network representing all chained dependencies among the various inputs, outputs, and functions used in the BNDE to generate a network display. This was generated by uploading a dependency layered ontology representing the BNDE.

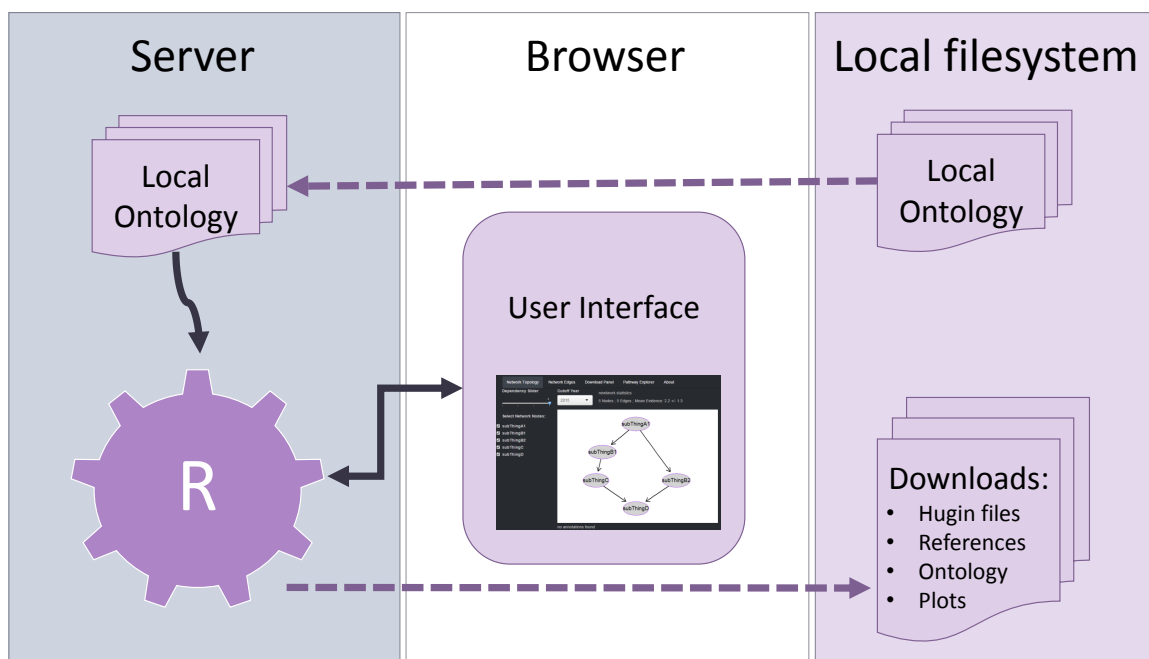


Figure 2: Dynamic information flow between the server, client, and local filesystem in the BNDE

Concepts are prefixed to indicate their superclass (user interface input/output `GUI_in:/GUI_out:`, or developer/reactive functions `DFN:/RFN:`). The `RFN:Z` function generates the network graph object used by several components of the software including a graph display function (`GUI_out:bnPlot`) at the bottom which generates the graph display output. At the time of writing, the BNDE ontology is available in the server-side local ontology selection list, that any brave of heart transparency seeking enthusiast might peruse to understand better how the software application works.

3 User interface development

Much of the design for the BNDE software was guided by the fact that the author of the software is also a potential future user. The layout of the tool is limited to some extent by screen space in general, regardless of device. The basic design is intended meet the goals of network development, i.e. the information to be displayed is mainly classes, networks, and some widgets for network pruning. However, this leaves little screen real-estate for other planned features. Therefore, this software could not be programmed as a flat-page site. This informed the choice to use a tabbed environment, where

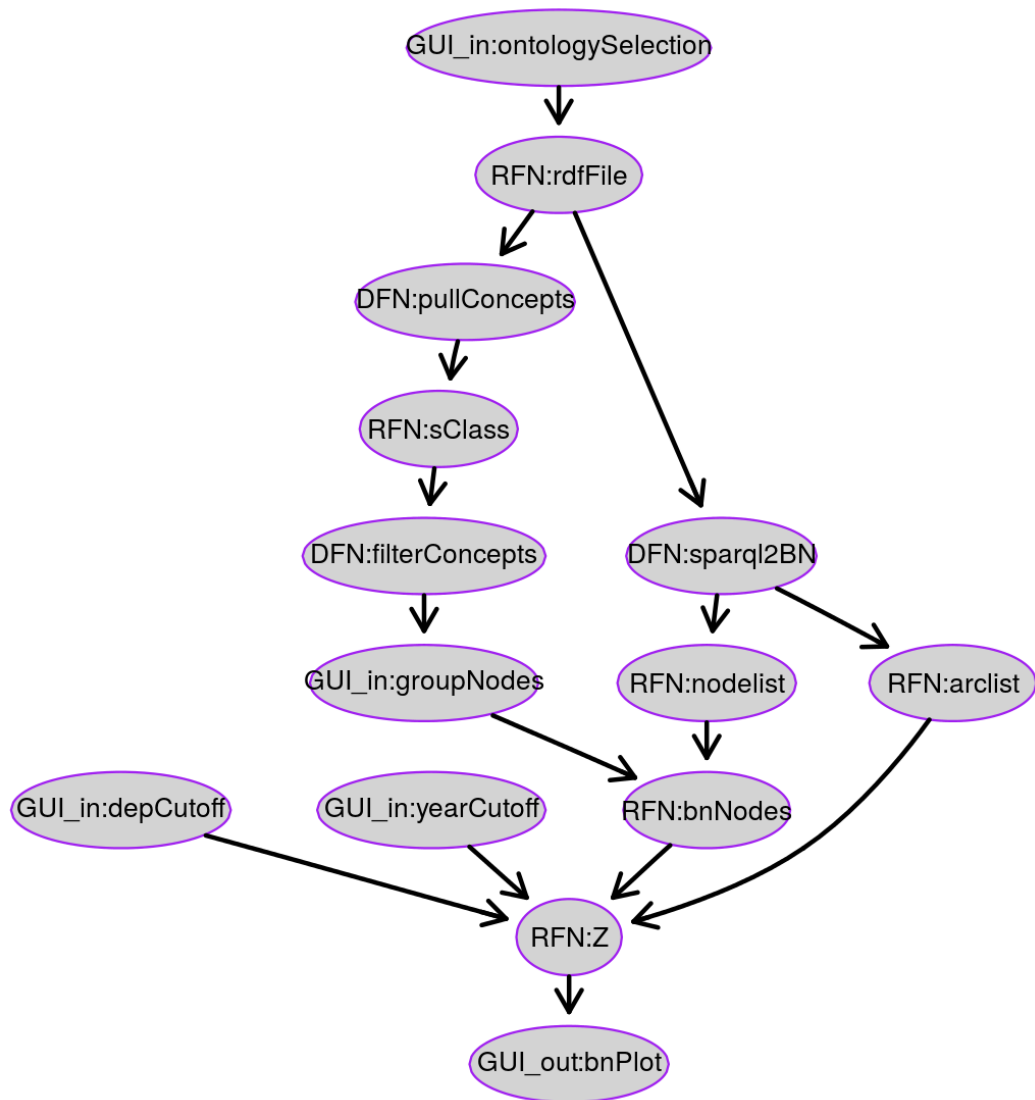


Figure 3: Functional dependency graph among variables in the BNDE used to generate network displays

additional features could be placed into other tabs. Having a tabbed page also lets users stay on one site while working and not have to reload any pages or refresh any information.

While some parameters can be static, others are functionally tied to network properties in a dynamic way. For example, the displayed node names

can be longer than the size of the node circle and become difficult to read. So the node `width` parameter is set as a function of the length (in characters) of the node name. Css stylesheets are used to maintain legibility of text, as well as minimal, gentle color schemes. Contrast is used to improve understanding of presented information. I found it important to keep required user actions as simple as possible, and provide *useful* feedback wherever issues arise.

4 Software features

The primary use of this software is to semi-automate the construction of dependency networks (Bayesian networks) from a domain ontology knowledge base. The domain ontology specifies what concepts are dependent on others, so it is not necessary to have *a priori* knowledge of what they are. When the user selects a set of concepts of interest, the tool will automatically create the network nodes and arcs in a graph object and display the structure of the network, among other things (described in more detail below). This eases the development of Bayesian networks in that the user need not reconstruct a network from scratch for every use case, nor search the literature or interview domain experts to establish an appropriate network structure. The interface is also interactive, thus, changes in concept selection, dependency level, and other parameters result in real-time updating of the graphics and other reactive features.

4.1 Ontology selection

In order to use this tool the user must select or upload an ontology. The main sidebar panel has options to do both. Here, a dropdown menu has been provided and a few preloaded ontologies can be selected from. Uploaded ontologies will appear in this selectable list of available ontologies after successful upload. A simple ontology ("testontology.owl") is available for experimenting with software features. These ontologies contain object properties that define the dependencies between classes.

Once an ontology is selected, the software will automatically read the class-subclass hierarchy and recreate it in the main sidebar panel as a folder-tree. It is generated from the ontology at the time of loading by using a recursive function that traverses the ontology's class-subclass tree to recreate the class hierarchy. This folder tree structure is an instance of (javascript) `jstree` and is therefore interactive and selectable. Though a few ontologies are available locally, the uploading feature gives users the flexibility to use these tools on any ontology constructed with the dependency layer protocol

e.g. object properties of “dependsOn” type.

4.2 Network topology

With an ontology loaded and a folder-tree built, a user can select among the various concepts shown in the folder tree. Holding the 'ctrl' key selects for multiple concepts. Selection of a concept or category of concepts will add it and any sub-concepts to the checkbox list just to the right of the folder tree. Some ontologies can be too large to view easily in a folder display, so this list is here to keep track of all the things are have selected so far. The checkbox group is also selectable, so network nodes can be removed and added from the graph without re-navigating the folder tree.

If dependencies among the selected concepts exist, then the network graph belonging to this set of concepts will be immediately computed and displayed in this panel. If there are no dependencies among the selected terms, nothing will be graphed or displayed, except an error message (see section 4.6). Similarly, if a selected node has no edge connection to any other terms in the set, it will not be displayed or included in the graph structure. If the dependency properties in the ontology have numeric tags specifying the strength of dependency, (“dependsOn3”, for example) then some statistical metrics describing the network (nodes and edges) will be computed and displayed above the graph.

The “Mean Evidence” metric is the mean value of the set of dependency arcs strengths. The “Evidence” term used comes from a medical context and represents a ranking system used to describe the strength of the results measured in a clinical trial or research study, though this value could represent some other type of dependency more generally. For an ontology that

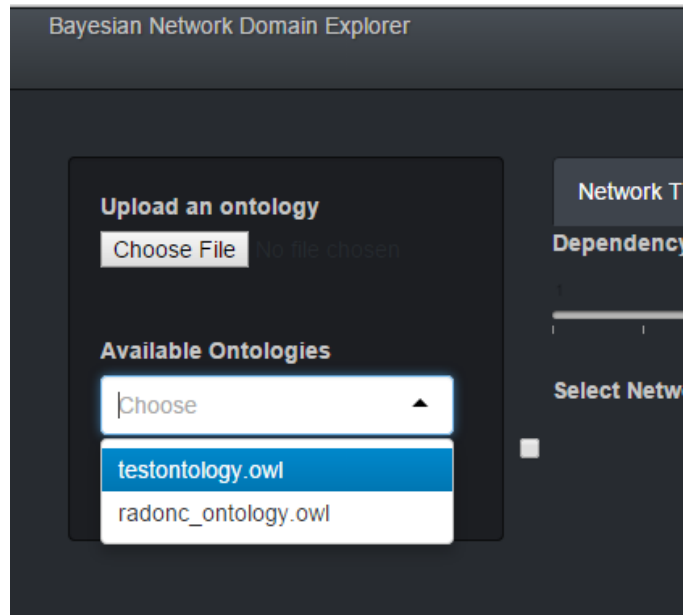


Figure 4: BNDE upload button and dropdown menu for ontology selection.

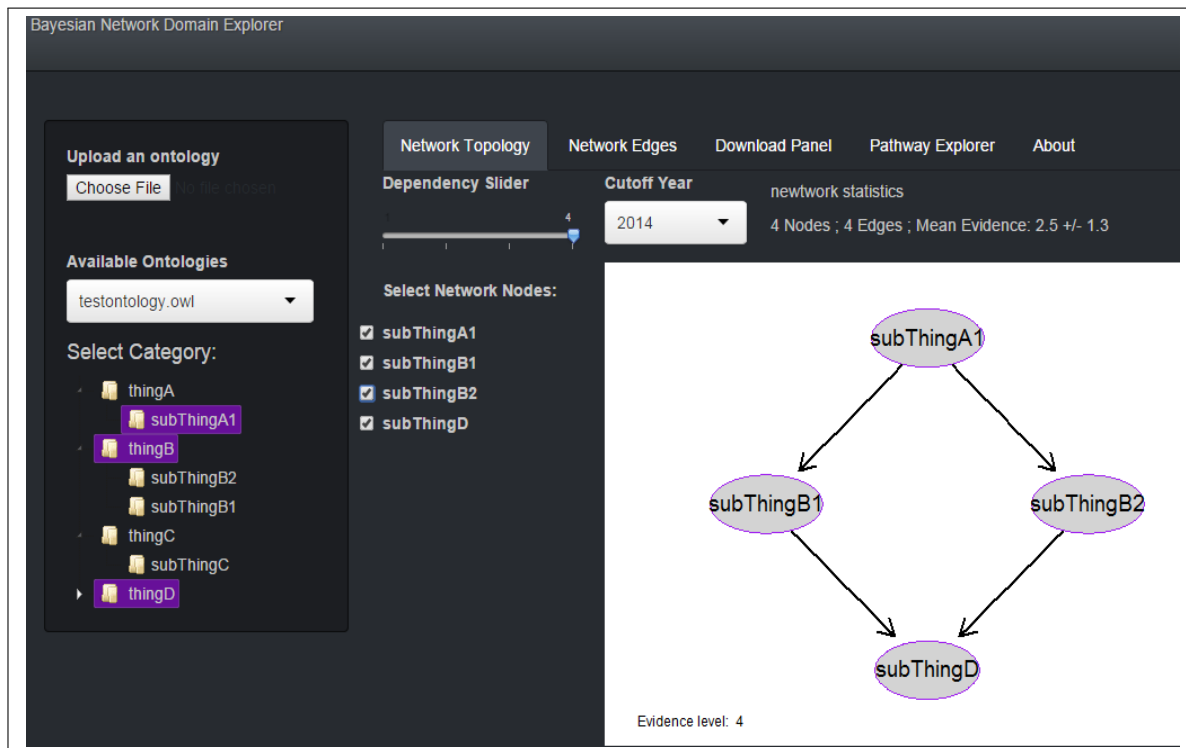


Figure 5: BNDE Network Topology tab. The selected ontology’s class-subclass structure is recreated (dynamic UI) as an interactive folder-tree appearing on the main sidebar panel (left).

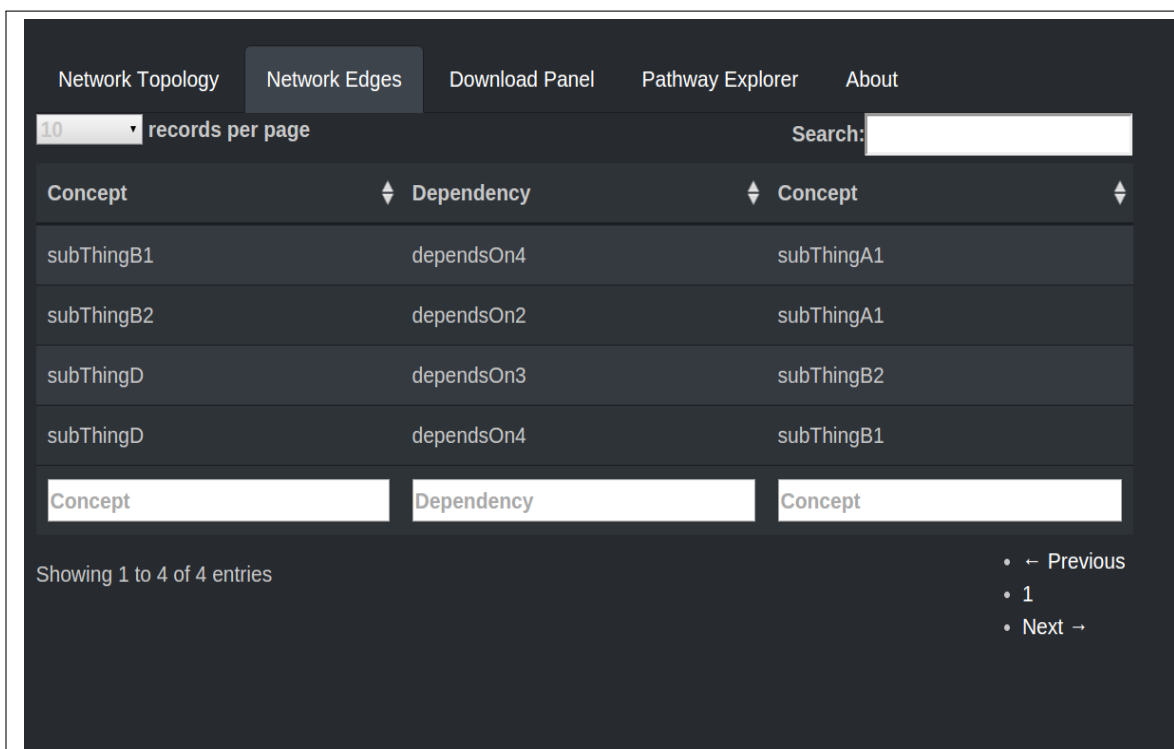
contains a layered dependency such as this, the ”Dependency Slider” can be changed to exclude nodes below the selected level. If no specific strength is provided by the ontology, all arcs are considered level 1 strength. Networks are updated in real-time as this value is changed. With respect to levels of evidence, one can use this slider to adjust or examine the strength of the overall network. As previously mentioned, because the software is reactive, changes in inputs (ontology choice, selected nodes, dependency slider, etc) will automatically result in re-computation and redrawing of the network display and any other dependent factors on other tab panels.

Because often the concept names used in ontologies are acronyms or jargon, it can be hard to read the network plot. For this reason, this software make the nodes of a displayed Bayesian network clickable. Clicking on nodes will produce their annotated definitions, NCIT definition, or SNOMED-CT definition, or all three if they exist. The node definition is printed just below the graph. If there are no definitions then a message is printed below to

indicate that nothing was found¹.

4.3 Network edge list

On the second tab in the tabset, a searchable, sortable, datatable is provided which displays the user constructed network's edge connections and each edge's dependency type. A screenshot of this panel is shown in Figure 6.



Concept	Dependency	Concept
subThingB1	dependsOn4	subThingA1
subThingB2	dependsOn2	subThingA1
subThingD	dependsOn3	subThingB2
subThingD	dependsOn4	subThingB1

Showing 1 to 4 of 4 entries

← Previous
• 1
• Next →

Figure 6: Edge Panel tab

4.4 Download handling

The download panel provides several options for saving user generated networks and obtaining more detailed information from the networks:

- Download a .net file that is compatible with the Hugin Expert² software. The Hugin software can accept a topology generated from this

¹Due to unresolved mismatching between server/client click-identification points, this is an unstable feature and remains in-progress

²Hugin Expert A/S, Aalborg, Denmark

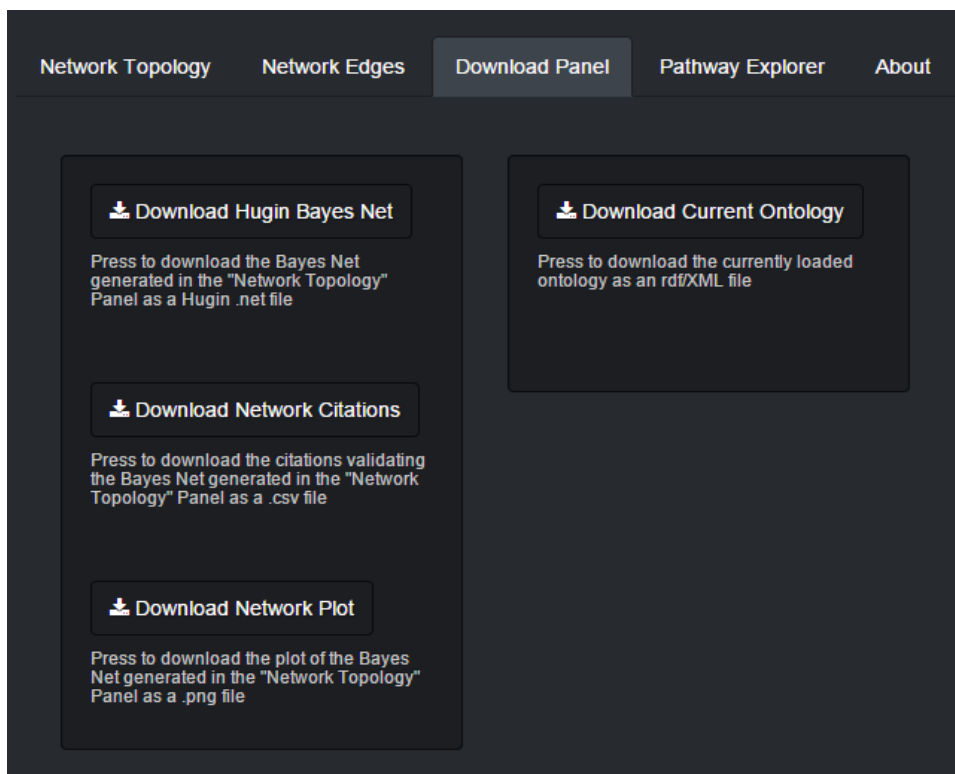


Figure 7: Dowload Panel in the BNDE

tool, and its facilities allow one to populate the conditional probability tables with user provided data or belief values to make a network fully computable.

- Download the citations belonging to the network edges. This feature allows downloading of the entire set of references associated with the network edges in the network created. Ontologies whose dependencies contain annotations (e.g. references to a journal publication or other set of information which supports the dependency evidence between two concepts) are extractable via this option. In this way, users can obtain the list of references (read: justifications) for the structure of any particular network. As part of the way the algorithm creates networks in this system, some edges/references will appear in this list that might not appear in the network itself. This is because dependency pathways might be logically chained from distant nodes via deductive reasoning methods. In the citation list, the chained dependency is included.
- Download the graphical plot of the user generated network in portable

network graphics format.

- Download the current ontology in RDF/XML format. These files can be viewed or edited using the free, open source ontology editor tool Protégé or other similar software.

4.5 Exploring dependency paths

The pathway explorer panel includes a feature which allows for examination of the full dependency paths between any two nodes in a network. This exploratory feature may be useful to understand further some of the underlying knowledge within a set of domain concepts. For instance, in the figure below, examining the paths between "subThingA1" and "subThingD" reveals a dependency path which includes a concept node from outside the currently selected network, "subThingC".

The screenshot shows the Pathway Explorer interface. On the left, there are controls for uploading an ontology and selecting categories. The main panel displays a dependency graph with nodes: subThingA1, subThingB1, subThingB2, subThingC, and subThingD. Arrows indicate dependencies: subThingA1 depends on subThingB2 and subThingB1; subThingB1 depends on subThingC; subThingB2 depends on subThingD; and subThingC depends on subThingD. Below the graph is a table of dependency paths.

Dependency Paths	Path References
subThingB2 dependsOn3 subThingA1	ref. year0000
subThingD dependsOn2 subThingB2	ref. year0000
subThingB1 dependsOn1 subThingA1	reference
subThingC dependsOn4 subThingB1	ref. year0000
subThingD dependsOn1 subThingC	ref. year0000

Figure 8: Pathway explorer panel

Using this feature is straightforward. Users select one node from each of the dropdown lists (these are populated from the selected network concepts in the main panel) and if there are dependency pathways between these nodes,

their graphical representation will be computed and shown in this panel, as well as an edge list below the graph which includes any reference annotations. There is a download button here as well which provides a spreadsheet file of the dependencies and their annotations, for the case that one is interested in exploring a particular set of dependency paths and their reference sources outside this software.

4.6 Error handling

The BNDE software has been tested in a host of modern browsers including Chrome, Iceweasel, Firefox, and Opera. The software also works on mobile browsers, though screen size limits much of what can be performed with this application as it is not optimized for mobile devices. When known errors occur during a network request or other operations, user feedback is given in the form of custom error messages. These errors don't crash the system, they simply inform the user. The application runs as usual even after encountering these errors. The following is a list of some messages currently output and what they indicate:

Error messages:

Error: please select additional concepts to create a network

Explanation: This notice occurs when not enough dependent concepts are selected to generate a network

Error: no paths exist for these concepts

Explanation: This notice occurs when no dependency paths exist between any of the selected concepts, regardless of external constraints

Error: no paths exist for these conditions

Explanation: This notice occurs when dependency paths exist among the selected concepts, but due to constraints put on by the user (for example, the dependency slider setting) all the available nodes are excluded from the network.

Error: no paths exist among these concepts

Explanation: This notice occurs when no dependency paths can be found to exist between the two selected nodes

5 Summary

The software application described here is a valuable tool for leveraging dependency layered domain ontologies for building directed graphical net-

works in a fairly straightforward way. With this application, researchers can take advantage of computational elements to subset ontologies and extract networks and network information. One limitation of this software (in its current beta form) becomes obvious when attempting to select for highly interconnected networks containing many nodes (30+). The path search algorithm is not optimized for performance, and it can take several minutes of waiting for a larger network to be constructed. Given real-time updating, any small change to input parameters of a network restarts the entire construction process. Despite this limitation, it is expected that network development time is still significantly reduced compared to the task of manually researching and justifying dependency among a similarly large set of domain concepts. Additionally, both breath-first and depth-first search algorithms process on a time proportional to the order of the number of edges plus the number of nodes, which sets a fundamental lower computational bound.

References

References

- [1] R Development Core Team. shiny: Web application framework for r, 2014.
- [2] Mark Otto and Jacob Thornton, 2011.
- [3] RStudio, 2014. Accessed: 2014-09-30.