WoolNet: Finding and Visualising Paths in Knowledge Graphs

Cristóbal Torres Gutiérrez[1], Aidan Hogan[1]

[1]IMFD & DCC, University of Chile, Santiago, Chile

Abstract

We present WoolNet: an online system for exploring paths in a knowledge graph. Specifically, given two or more entities requested by a user, the system finds and visualises paths that connect these entities, forming a topical subgraph. Upon requesting entities, a multi-source path search algorithm is run against an in-memory index of the knowledge graph, with paths displayed as they are found. Running the algorithm for longer (up to a maximum of two minutes) results in the system finding and displaying more paths. Features are provided to filter paths by length and node degree. The system we present and evaluate currently illustrates the use of WoolNet for exploring paths in the Wikidata knowledge graph.

Keywords

knowledge graphs, paths, exploratory search

1. Introduction

Techniques for graph-based data management have enjoyed renewed interest in recent years, particularly in the context of knowledge graphs [1]. In this setting, graphs provide a flexible model useful for integrating diverse data at large scale, where nodes represent entities, and edges represent relationships between these entities. This idea has given rise to both enterprise knowledge graphs (published by AirBnB, eBay, Facebook, Google, etc.) [1], as well as open knowledge graphs such as Wikidata [2].

A key feature of graph-based data models is the ability to find both direct and indirect connections between entities via paths in the graph. Graph query languages such as Cypher [3], G-CORE [4], GQL [5], Gremlin [6] and SPARQL [7] – and thus the engines supporting them (e.g., [8, 9, 10, 11], among many others) – incorporate a range of features for querying graphs, including finding shortest paths, or finding paths matching regular expressions (via regular path queries and variations thereof) [12]. However, many potential users of knowledge graphs may not be expert in the use of such languages and systems.

We are thus interested in enabling non-expert users to understand what connects two or more entities of interest in that knowledge graph for use-cases involving exploratory search [13] (e.g., for identifying new potential collaborations, potential conflicts of interest or collusion, relations between diseases and drugs, etc.). In this setting, a connection is formed between

AMW 2024: 16th Alberto Mendelzon International Workshop on Foundations of Data Management, September 30th–October 4th, 2024, Mexico City, Mexico

ctorresg@dcc.uchile.cl (C. Torres Gutiérrez); ahogan@dcc.uchile.cl (A. Hogan)
https://ctorresg.cl/ (C. Torres Gutiérrez); https://aidanhogan.com/ (A. Hogan)
0009-0000-9840-9282 (C. Torres Gutiérrez); 0000-0001-9482-1982 (A. Hogan)

© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

a pair of entities via a path in the graph. Paths between the same pair of entities may share nodes, as may paths between different pairs of entities. Thus a topical subgraph can be formed consisting of the paths between one or more pairs of entities of interest to the user.

To support such users, we propose *WoolNet*: an online system for finding and visualising paths in knowledge graphs. Having indexed a knowledge graph offline, the system enables users to search for two or more entities of interest. Upon hitting search, the back-end of the system begins to find paths in the knowledge graph between each pair of entities. As paths are found, they are sent to the front-end and displayed graphically to the user. A public demo of our system is available online at https://woolnet.dcc.uchile.cl.

In the design and development of WoolNet, we address two technical challenges: *performance* and *usability*. In order to support finding and visualising paths in real-time over large-scale knowledge graphs, performance is key, for which we propose in-memory indexes and a (to the best of our knowledge, novel) multi-source path finding algorithm. In terms of usability, WoolNet keeps the base user interaction simple, but further provides more advanced features for filtering paths that are not of interest. We then evaluate the performance and usability of WoolNet in the context of the Wikidata knowledge graph.

2. Related Work

Popular graph query languages [7, 6, 3, 5] and graph databases [8, 9, 10, 11] provide a range of features for querying paths. Such features require the user to specify their request as part of a structured query, which may be beyond the expertise of many users.

WoolNet rather targets a broader user base, featuring a more accessible user interaction inspired by two systems with similar functionality: RelFinder [14] and WiSP [15]. The system further permits finding paths between three or more nodes, which is not typically supported by graph databases and query languages.

The closest system to WoolNet is RelFinder [14], which permits finding and visualising paths between entities in DBpedia. However, the back-end of RelFinder relies on evaluating SPARQL basic graph patterns of fixed and increasing size to find paths of 1, 2, 3, etc.; note that property paths are not applicable here as they do not support returning the paths themselves, only the nodes connected by paths. This approach is further complicated by the need to query paths in both directions; for paths of length k, this would require 2^k unions of basic graph patterns with k triple patterns each, or k joins of unions of 2 triple patterns. Initial experiments delegating such queries to the public Wikidata Query Service [16] often generated timeouts after one minute for paths of length 2, and would not support finding and visualising paths interactively as they were found. We thus ruled out this approach.

Another related system is WiSP (Weighted Shortest Paths) [15], which supports finding paths between Wikidata entities. However, WiSP only finds a single path between a single pair of entities, with a focus on trying to find the most "interesting" path between two entities. Interesting paths are identified based on a combination of path length and graph metrics, where, for example, shorter paths passing through lower-degree nodes are considered more notable.

3. Preliminaries

We provide some brief preliminaries used throughout the paper.

We define a *knowledge graph* G to be a set of triples, where each triple (s, p, o) denotes a directed labelled edge of the form $s \xrightarrow{p} o$. We denote by $V(G)$ the set of *nodes* (or *vertices*) of G; i.e., $V(G) = \{x \mid \exists p, y : (x, p, y) \in G \text{ or } (y, p, x) \in G\}$. We denote by $L(G)$ the set of *edge labels* (or *predicates*) of G; i.e., $L(G) = \{p \mid \exists x, y : (x, p, y) \in G\}$. Given a node $v \in V(G)$, we denote by $E(v, G)$ the incident edges of v in G, i.e., $\{(x, p, y) \in G \mid x = v \text{ or } y = v\}$. We denote by $N(v, G)$ the neighbours of v in G, i.e., $N(v, G) = V(E(v, G)) \setminus \{v\}$.

We define a 1-length path between $u \in V(G)$ and $v \in V(G)$ in G as a sequence $\langle(x, p, y)\rangle$ such that $(x, p, y) \in G$, and $\{u, v\} = \{x, y\}$. We define a k-length path between u and v in G as a sequence of k distinct triples $\langle(x_1, p_1, y_1), \ldots, (x_k, p_k, y_k)\rangle$ such that:

- $u \in \{x_1, y_1\}$ and $v \in \{x_k, y_k\}$, and
- for all $1 \leq i \leq k$ it holds that $(x_i, p_i, y_i) \in G$, and
- for all $1 < i \leq k$, it holds that $\{x_{i-1}, y_{i-1}\} \cap \{x_i, y_i\} \neq \emptyset$.

Given a knowledge graph G, and a set of nodes $\{v_1, \ldots, v_n\}$ ($n \geq 2$) representing entities of interest selected by a user, the goal of WoolNet is to interactively identify and visualise paths in G between pairs of nodes v_i and v_j, with $1 \leq i < j \leq n$. Rather than visualise individual paths, WoolNet visualises the graph – considered a topical subgraph of the knowledge graph – consisting of the union of all edges of relevant paths.

4. Technical Overview of WoolNet

We now provide a technical overview of WoolNet. We start by describing the indexing technique used for the knowledge graphs. We then describe our multi-source graph search algorithm used to find paths between multiple entities. We present the design of the front-end and the user interaction it provides. We end by describing some features we added to filter the paths found and displayed to generate a more concise topical subgraph.

Graph indexing Our aim is to find and visualise paths connecting entities in large knowledge graphs. Our demo – described later – over Wikidata involves a graph of 100 million nodes and 716 million edges. Given the need to navigate significant parts of the knowledge graph in order to find paths, and the fact that such navigation needs to be conducted at runtime while the user waits for such paths to be reported, efficient methods to access and perform lookups on the graph are essential. In particular, given a node $v \in V(G)$, the key operation we need to implement efficiently is that of finding all incident edges $E(v, G)$.

To implement this operation as efficiently as possible, which in turn will increase the rate at which paths are found and returned to users, we developed and evaluated a number of static in-memory index structures. In the interest of space, we will present the most efficient – both in terms of time and space – currently used in the online system. We assume that the nodes and edge labels of the knowledge graph G use an alphabet of integers from the set $\{1, \ldots, |V(G) \cup L(G)|\}$, applying dictionary encoding if necessary. We then build an adjacency

list Adj consisting of an in-memory multi-dimensional integer array of size $|V(G)|$ structured as a trie. Consider for example the graph $G = \{(1,2,3),(1,2,5),(3,4,5),(5,6,1)\}$. The index $\texttt{Adj}_G$ constructed for G will then be as follows:

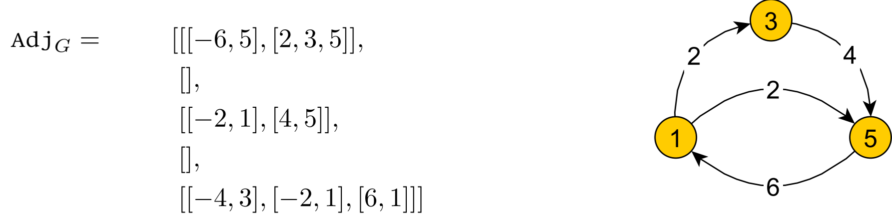


Figure 1: On the left the graph G as $\texttt{Adj}_G$ using the index representation, on the right an image of the graph G

This representation is a three-dimensional array where the initial index corresponds to the node label; this array has length equal to the maximum node label in the graph. Each index of this three-dimensional array resolves to a two-dimensional array, with an array for each unique edge label incident for that particular node (with edges in both directions): the first element of each such array indicates the edge label, while proceedings elements indicate the nodes linked by that label. Here, for example, the third element of $\texttt{Adj}_G$ indicates that node 3 has two incident edges: an incoming edge labelled 2 (we negate edge labels to indicate the inverse direction) pointing to node 1, and an outgoing edge labelled 4 pointing to node 5. Given a node v, we can then retrieve the incident edges of v (i.e., $E(v, G)$) in time $O(1)$ via an array lookup on $\texttt{Adj}_G$. Because there do not exist nodes with the values 2 and 4, their slots are empty.

This representation is used because (1) using native arrays instead of lists, objects or other structures requires less memory and enables faster operations; (2) with this trie-like structure, less duplication is required; (3) although using the node label as the index may leave "gaps", it is faster to get the neighbors of a node with a specific identifier than using a dictionary (particularly in cases such as Wikidata, where node labels are numeric in nature, with a Q prefix; in this case, the neighbours for Q42 are retrieved from $\texttt{Adj}_G[42]$).

Path finding Given a knowledge graph G, and a set of nodes $\{v_1, \ldots, v_n\} \subseteq V(G)$, with $n \geq 2$, our goal is to find paths between all pairs of nodes v_i and v_j, with $1 \leq i < j \leq n$.

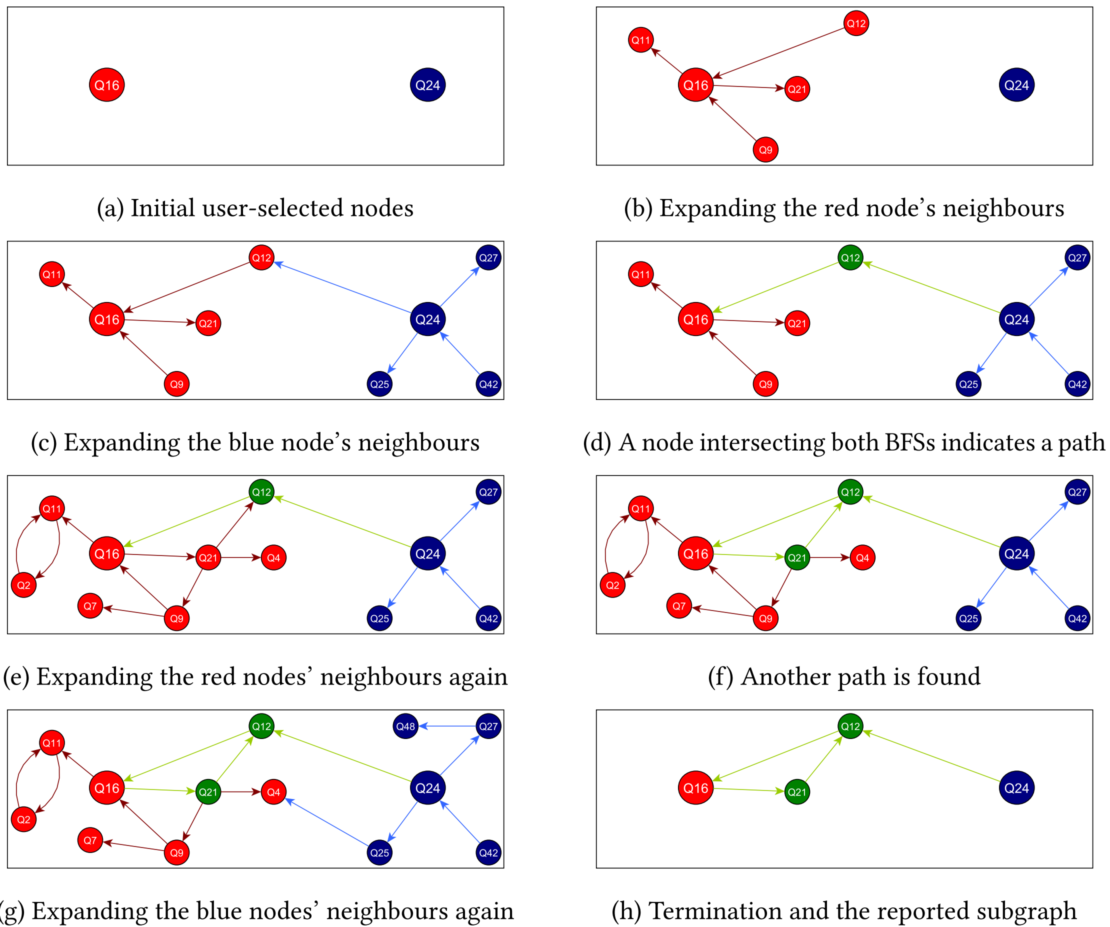
Rather than finding paths between each pair of nodes, requiring $O(n^2)$ independent searches that do not leverage information found previously (which would be necessary if we used existing graph database technology as our back-end), we propose the multi-source path finding (MSPF) algorithm sketched in Algorithm 1. Given the user-selected nodes $\{v_1, \ldots, v_n\}$, and a MAX_LENGTH parameter, we maintain a queue to which we initially add the user-selected nodes. We will conduct a BFS from each such node, expanding the neighbours of each node in turn, and then the neighbours of neighhours, etc. From each initial node, we expand $\lceil \frac{\text{MAX_LENGTH}}{2} \rceil$ hops. When a node is found in the intersection of two or more BFSs, it witnesses a path between the initial nodes of those BFSs. We further maintain meta-data for each node indicating the BFS in which it was first found, its distance from the initial (user-selected) node in that BFS, and its minimum distance from the initial node in any other BFS. If we find two neighbours with

distances in two distinct BFSs summing to less than MAX_LENGTH, we report a path. In practice, we must further store meta-data about the nodes from which we reached the current node, and via which triples, in order to backtrack and report novel edges in the paths; it is also possible for multiple paths to be reported when multiple triples constitute a path of the same length. Crucially, paths are reported immediately as they are found. In terms of optimisation, on line 10 of the algorithm, we construct the neighbours using the Adj index discussed previously.

Algorithm 1: Multi-source path finding (MSPF)

1. **Parameter:** Integer MAX_LENGTH;
   Data: Knowledge graph G, Nodes $\{v_1, \ldots, v_n\}$;
2. **Init:** Queue Q; Map M;
3. **for** $1 \leq i \leq n$ **do**
4.   Q.enqueue(v_i);
       /* M stores tuples of the form $\langle b, d, d' \rangle$ where b identifies the current BFS, d the distance from the initial node of the current BFS, and d' the minimum distance from the initial node of any other BFS */
5.   M.put$(v_i, \langle i, 0, \infty \rangle)$;
6. **while** *Q is not empty* **do**
7. $u \leftarrow$ Q.dequeue$()$;
8. $\langle b_u, d_u, d'_u \rangle \leftarrow$ M.get(u);
9.   **if** $d_u \leq \lceil \frac{MAX_LENGTH}{2} \rceil$ **then**
10.   **for** $w \in N(u, G)$ **do**
11.   **if** *M.hasKey*(w) **then**
12. $\langle b_w, d_w, d'_w \rangle \leftarrow$ M.get(w);
13.   **if** $b_u = b_w$ **then**
14.   **if** $d_u + d'_w + 1 \leq$ *MAX_LENGTH* **then**
15.   **Report** path;
16.   M.put$(u, \langle b_u, d_u, \min\{d'_u, d'_w + 1\} \rangle)$;
17.   M.put$(w, \langle b_w, d_w, \min\{d'_w, d'_u + 1\} \rangle)$;
18.   **else**
19.   **if** $d_u + d_w + 1 \leq$ *MAX_LENGTH* **then**
20.   **Report** path;
21.   M.put$(u, \langle b_u, d_u, \min\{d'_u, d_w + 1\} \rangle)$;
22.   M.put$(w, \langle b_w, d_w, \min\{d'_w, d_u + 1\} \rangle)$;
23.   **else**
24.   Q.enqueue(w);
25.   M.put$(w, \langle b_u, d_u + 1, \infty \rangle)$

An example of the algorithm is shown in Figure 2. The algorithm is configured with a MAX_LENGTH parameter, which in this case we assume to be 3. We start from the initial user-selected nodes (Q16 in red, and Q24 in blue, in Figure 2a), and expand from one of these nodes to visit its neighbours (in Figure 2b, the red node Q16 is expanded). The algorithm continues, this time expanding the neighbours of the next node (the blue node Q24 per Figure 2c). As we expand nodes, we look for intersections at the frontier of the search, and check the meta-data



(a) Initial user-selected nodes

(b) Expanding the red node's neighbours

(c) Expanding the blue node's neighbours

(d) A node intersecting both BFSs indicates a path

(e) Expanding the red nodes' neighbours again

(f) Another path is found

(g) Expanding the blue nodes' neighbours again

(h) Termination and the reported subgraph

Figure 2: Example of using the MSPF algorithm to find paths between two nodes.

on intersecting nodes to ensure that they satisfy the maximum length constraint, in which case we backtrack to materialise and report the path (see intersecting node Q12 witnessing a path in Figure 2d). The search continues until we have expanded $\lceil \frac{\mathit{MAX_LENGTH}}{2} \rceil$ hops from all of the initial nodes (per Figure 2e, where we expand the red node's neighbours, finding a new path as seen in Figure 2f, and then expanding the blue node's neighbours in Figure 2g, terminating with the subgraph shown in Figure 2h; we highlight that in Figure 2g, the node Q4 intersects both BFSs, but its paths are rejected as they exceed the maximum length constraint). The algorithm generalises straightforwardly to the case of having three or more initial nodes.

It is important to note that our MSPF algorithm intends to return the subgraph induced by the corresponding paths found between the nodes, i.e., the union of the sets of triples forming all such paths. We conjecture that the algorithm is *sound* in the sense that no triple is returned that does participate in such a path, and *complete* in the sense that all such triples are (eventually) returned by the algorithm. However, Algorithm 1 would require modification to ensure that all relevant *paths* are reported (rather than the union of the triples that form those paths).

User interface In Figure 3 we provide a screenshot illustrating the three principal components of the user interface taken from the demo over Wikidata described in more detail later.[1] For clarity, we present an example showing paths between two entities: Alan Turing and Edsger W. Dijkstra. On the left pane, users can search for an entity of interest using an auto-completion dialogue. For example, upon typing "dijkstra", the user is displayed a list of possible entities that includes Edsger W. Dijkstra. The user can select from the options shown to add an entity to the system. Once the user has selected two or more entities, the user can hit SEARCH, and in the main pane, the entities of interest will be displayed. In the back-end, the path-finding algorithm is invoked, and will begin to report paths. As these paths are received, they will be added to the graph displayed. Each node is represented visually with a label and an image (if available). Edges are displayed with direction and with their edge label. Intermediate nodes can be clicked in order to highlight the paths through them; they can also be clicked and dragged to move them. Their default position on the canvas is selected by force-directed layout. From the graph visualised in Figure 3, a user can learn that Turing and Dijkstra were both human, computer scientists and male; that they both spoke English (being the native language of Turing); that Dijkstra won the Turing Award named after Turing; and that Dijkstra was educated at the University of Cambridge, at which Turing worked.

Filtering paths Some queries may generate a graph with many edges making it difficult to view particular connections of interest. Hence we set some constraints on paths, and provide user-controlled filters to narrow down the subgraph.

In terms of constraints, we set MAX_LENGTH to 3 based on initial experience, where 3 is sufficient to find some connection between the vast majority of pairs of nodes, whereas paths of length 4 tend to generate irrelevant connections. We also set an overall timeout of 2 minutes for finding paths. We further opted to not expand neighbours of high-degree nodes. For example, in Figure 3, numerous nodes can be inserted either between Dijkstra and English or Turing and English while satisfying the maximum length constraint, including papers written by both in English, other people they are connected to that speak English, etc. Thus if a path intersects on a high-degree node, we still include such a path (as shown in Figure 3), but we will not expand that node's neighbours. As seen later, this heuristic further reduces memory overhead, while also increasing the paths reported per unit of time. We note that these three decisions are largely motivated by our practical goal to reduce the average cost (particularly in memory) of a request, and thus to be able to host a stable service on the Web via commodity hardware.

In terms of filters, at the bottom of the main pane we provide two sliders for filtering paths/edges from the visualisation. The first slider filters paths according to their maximum length. The second filters paths that pass through an intermediate node whose degree is above 10^n, where $1 \leq n \leq 8$. The idea behind this latter filter is that paths passing through high-degree nodes (such as English or human in Figure 3) are less likely to be of interest to the user [15]. An example of the usage of filters is provided in Figure 4 for a search between Alan Turing, Jensen Huang and Edsger W. Dijkstra: Figure 4a shows the search for paths between the initial entities without using any filter, Figure 4b uses a filter to only show paths with length 2 of less, Figure 4c filters nodes with a degree of more than 10^5, and Figure 4d uses both filters.

[1]Please note that the screenshot has been graphically edited for clarity.

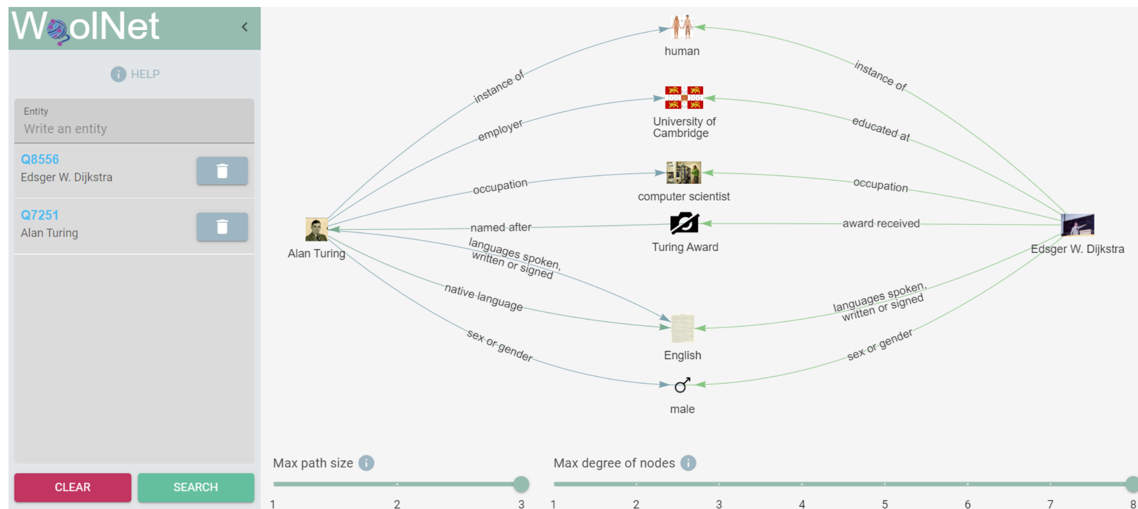
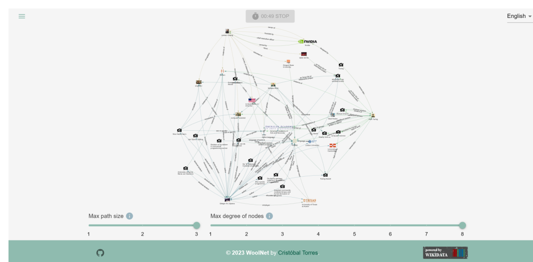
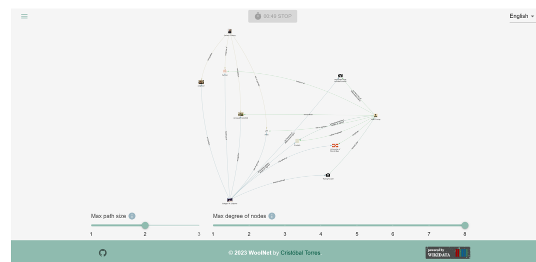


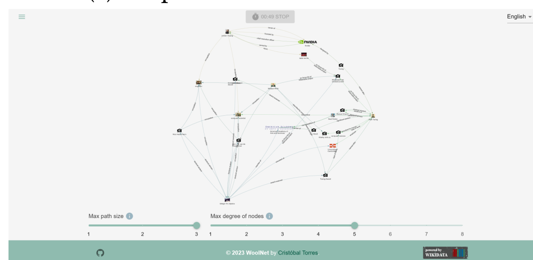
Figure 3: User interface of WoolNet showing paths found between Alan Turing and Edsger W. Dijkstra in Wikidata.



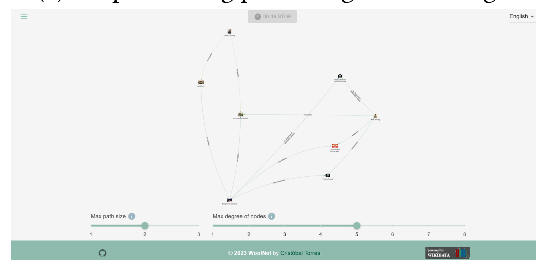
(a) Graph without the use of filters



(b) Graph filtering paths longer than 2 edges



(c) Graph filtering high-degree nodes



(d) Graph applying both filters

Figure 4: Different filters applied to a search between three nodes.

5. Evaluation

We provide an overview of the evaluation of the system in terms of performance (both space and time) and usability.

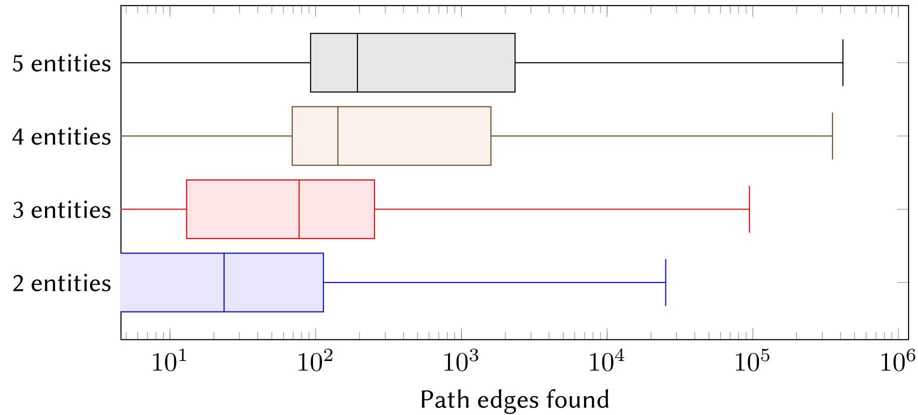


Figure 5: Number of path edges found in 60 seconds for differing numbers of entities in the search.

Setup Experiments were run on a machine with 59.6 GB of RAM, a 12-core Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz running Devuan. The index and graph search were implemented in Java (v. 11.0.18). We index a truthy version of Wikidata, considering only triples with Q-code nodes and truthy P-code edge labels. This graph contains 715,906,922 edges and 99,609,308 nodes. Based on preliminary experiments, we do not dictionary encode the graph, but rather use the numeric codes of Wikidata directly as IDs. Processing the dump in order to filter unused triples and load the `Adj` index required 4.8 hours, occupying 26.9 GB in memory.

Performance To test the path finding rate, we selected 10,000 nodes from Wikidata, randomly sampled with weights proportional to the degree of the node. We weighted the random sample as users are more likely to enter high-degree nodes, where the vast majority of nodes in Wikidata have a low degree and are relatively obscure. We then selected 100 sets of 2, 3, 4 and 5 nodes. Running each set for 60 seconds, we measure how many edges are found for relevant paths. Figure 5 presents the results, where we see that with more entities specified by the user, we can find more paths per unit time. The median number of path edges found in one minute varies from 24 to 194, with a median memory usage peaking at 7 GB for a single query. Applying the restriction that avoids expanding neighbours of nodes with degree greater than 100,000 increases the median number of path edges found in one minute to the range of 34 to 1,009, with a greatly reduced ($\times$70) median memory usage of 100 MB per query.

Usability In order to gain initial insights into the usability of the system, we created an online survey that asked user to perform two tasks: one is a guided navigation (where the entities for the search are provided), hinting in this case that they may find an interesting path and the other is a navigation where the user chooses the entities to search.

They then responded to a System Usability Scale (SUS) survey [17], rating ten claims about the system on a Likert scale from 1 (disagree) to 5 (agree). The SUS score ranges from 0–100, where a score of 68 is considered average [18]. A Spanish version of the survey was shared on a forum of students and professors at the University of Chile, receiving 82 responses. An English

Table 1
SUS results (m: mean, s: standard deviation).

Claim to evaluate	*ES(82)*		*EN(15)*		*ALL(97)*	
	m	*s*	*m*	*s*	*m*	*s*
I think that I would like to use this system frequently.	3.04	1.21	3.07	1.44	3.04	1.24
I found the system unnecessarily complex.	1.83	0.99	2.27	1.28	1.90	1.05
I thought the system was easy to use.	4.34	0.76	3.87	1.19	4.27	0.85
I think that I would need the support of a technical person to be able to use this system.	1.37	0.69	1.73	1.03	1.42	0.76
I found the various functions in this system were well integrated.	4.26	0.87	3.47	1.13	4.13	0.95
I thought there was too much inconsistency in this system.	1.45	0.77	2.33	0.90	1.59	0.85
I would imagine that most people would learn to use this system very quickly.	4.22	1.03	4.07	1.10	4.20	1.04
I found the system very cumbersome to use.	1.66	0.92	2.33	1.05	1.76	0.97
I felt very confident using the system.	4.28	0.86	4.13	0.92	4.26	0.87
I needed to learn a lot of things before I could get going with this system.	1.29	0.66	2.13	1.36	1.42	0.86
SUS Score	81.34	13.35	69.50	17.53	79.51	14.62

version was shared on the public Wikidata mailing list, receiving 15 responses.

Table 1 presents the mean and standard deviation of the responses for each claim of the SUS questionnaire, along with the SUS score. Considering all 97 respondents, the overall SUS score was 79.51 ± 14.62, which is considered better than average usability [18]. While users appreciate the speed at which paths are found, and the way that some such paths reveal interesting connections between entities of interest, they also commented that sometimes too many paths are displayed, making it difficult to read the graph.

6. Discussion

WoolNet is an online system that enables users to find, filter and visualise paths between entities of interest in a knowledge graph. The system can be used for exploratory search, where some use-cases we envisage include finding potential collaborations between researchers or affiliations, identifying potential conflicts of interest, studying vectors of corruption, etc. We showed that by using our proposed indexing scheme and multi-source path finding algorithm, WoolNet is capable of finding tens or hundreds of paths per minute (in the median case) on a real-world knowledge graph consisting of 100 million nodes and 716 million edges. An evaluation with 97 users indicates that WoolNet's usability is better than average.

The final system applies a number of practical heuristics to reduce the average cost of each request, such as limiting the maximum path length, avoiding the expansion of high-degree nodes, etc.; these heuristics could affect certain applications, and testing them in more detail would be an interesting subject for future work. Another next step would be to conduct a topology-based analysis of the performance of the algorithm considering the graph density, potentially using synthetic graphs, and also to explore adaptations to our algorithm that would allow for returning full paths between all pairs of nodes according to a particular semantics.

The online system (https://woolnet.dcc.uchile.cl) provides an end-to-end system. In terms of limitations and potential future improvements, WoolNet assumes the knowledge graph to be static, where in a future version we hope to support updates. It would also be of interest to explore the potential of multi-threading to improve performance, and to be able to configure the system to use popular graph databases as a back-end. In some cases, WoolNet can generate a great many paths, making it difficult to visually distinguish individual connections; we would like to provide users more powerful features to filter paths, and to explore measures and other heuristics that can help to filter paths that are unlikely to be of interest to a user.

Acknowledgments

We thank the anonymous users who took part in our survey for their time and feedback. We also thank the anonymous reviewers whose feedback helped to improve this work. This work was partly funded by ANID - Millennium Science Initiative Program - Code ICN17_002 and by FONDECYT Grant 1221926.

References

[1] A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. de Melo, C. Gutierrez, S. Kirrane, J. E. L. Gayo, R. Navigli, S. Neumaier, A. N. Ngomo, A. Polleres, S. M. Rashid, A. Rula, L. Schmelzeisen, J. F. Sequeda, S. Staab, A. Zimmermann, Knowledge graphs, ACM Comput. Surv. 54 (2022) 71:1–71:37. URL: https://doi.org/10.1145/3447772. doi:10.1145/3447772.

[2] D. Vrandecic, M. Krötzsch, Wikidata: a free collaborative knowledgebase, Commun. ACM 57 (2014) 78–85.

[3] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, A. Taylor, Cypher: An Evolving Query Language for Property Graphs, in: SIGMOD 2018, 2018. URL: https://doi.org/10.1145/3183713.3190657. doi:10.1145/3183713.3190657.

[4] R. Angles, M. Arenas, P. Barceló, P. A. Boncz, G. H. L. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. F. Sequeda, O. van Rest, H. Voigt, G-CORE: A Core for Future Graph Query Languages, in: Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018, ACM, 2018, pp. 1421–1432. URL: https://doi.org/10.1145/3183713.3190654. doi:10.1145/3183713.3190654.

[5] A. D. et. al., Graph pattern matching in GQL and SQL/PGQ, in: SIGMOD '22, 2022. URL: https://doi.org/10.1145/3514221.3526057. doi:10.1145/3514221.3526057.

[6] M. A. Rodriguez, The Gremlin graph traversal machine and language (invited talk), in: Proceedings of the 15th Symposium on Database Programming Languages, Pittsburgh, PA, USA, October 25-30, 2015, ACM, 2015, pp. 1–10. URL: https://doi.org/10.1145/2815072.2815073. doi:10.1145/2815072.2815073.
[7] W. W. W. Consortium, et al., Sparql 1.1 overview (2013).
[8] O. Erling, Virtuoso, a Hybrid RDBMS/Graph Column Store, IEEE Data Eng. Bull. 35 (2012) 3–8. URL: http://sites.computer.org/debull/A12mar/vicol.pdf.
[9] J. Webber, A programmatic introduction to Neo4j, in: SPLASH '12, 2012. URL: https://doi.org/10.1145/2384716.2384777. doi:10.1145/2384716.2384777.
[10] B. Thompson, M. Personick, M. Cutcher, The Bigdata® RDF Graph Database, in: Linked Data Management, Chapman and Hall/CRC, 2014, pp. 193–237.
[11] B. R. Bebee, D. Choi, A. Gupta, A. Gutmans, A. Khandelwal, Y. Kiran, S. Mallidi, B. McGaughy, M. Personick, K. Rajan, et al., Amazon Neptune: Graph Data Management in the Cloud, in: ISWC (P&D/Industry/BlueSky), 2018.
[12] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. L. Reutter, D. Vrgoč, Foundations of Modern Query Languages for Graph Databases, ACM Comput. Surv. 50 (2017). URL: https://doi.org/10.1145/3104031. doi:10.1145/3104031.
[13] M. Lissandrini, T. B. Pedersen, K. Hose, D. Mottin, Knowledge graph exploration: where are we and where are we going?, SIGWEB Newsl. 2020 (2020) 4:1–4:8. URL: https://doi.org/10.1145/3409481.3409485. doi:10.1145/3409481.3409485.
[14] P. Heim, S. Hellmann, J. Lehmann, S. Lohmann, T. Stegemann, RelFinder: Revealing Relationships in RDF Knowledge Bases, in: 4th International Conference on Semantic and Digital Media Technologies (SAMT), volume 5887 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 182–187. URL: https://doi.org/10.1007/978-3-642-10543-2_21. doi:10.1007/978-3-642-10543-2_21.
[15] G. Tartari, A. Hogan, WiSP: Weighted Shortest Paths for RDF Graphs, in: Fourth International Workshop on Visualization and Interaction for Ontologies and Linked Data (VOILA@ISWC), volume 2187 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2018, pp. 37–52. URL: https://ceur-ws.org/Vol-2187/paper4.pdf.
[16] S. Malyshev, M. Krötzsch, L. González, J. Gonsior, A. Bielefeldt, Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph, in: ISWC 2018, 2018.
[17] J. R. Lewis, The system usability scale: past, present, and future, International Journal of Human–Computer Interaction 34 (2018) 577–590.
[18] J. Sauro, J. R. Lewis, Quantifying the user experience: Practical statistics for user research, Morgan Kaufmann, 2016.