Path Finding Algorithms & Application in Discovering Meaningful Connections between Resources in the Web of Data

ABSTARCT:

Path Finding is well known issue in graph theory. Path Finding is a route plotted between two points starting point and destination point by a computer application. It is kind of solving a maze in real time. Several algorithms have been proposed to find the path in huge labyrinth. Some of the most well known methods such as Dijkstra's Algorithm, Depth First Search, Breadth First Search, A* etc. And we would discuss that semantically annotated paths lead to meaningful, significant connection and association among the plenty of resources in huge online datasets such as web of data. We would convert the Linked structured data in to the graph theory problem. Graph path finding algorithms are key in finding the path in structured data by making use of optimum computational resources. Applying these graph path-finding algorithms to inter-related data makes easier to resolve complex queries such as semantics of the relations between resources.

PROJECT PLAN:

In this paper we will be studying contribution of graph theory techniques, path finding algorithms and optimized algorithm among those based on their time complexity used to solve a search. Next, we will be discuss a new approach for detecting paths in Linked Data that considers the meaning of the connections and also deals with scalability. Approach combines preliminary processing and indexing of the datasets is used for finding paths between two resources in large-scale network of data in seconds. Pre-processing involves covert the Linked Dataset to list of triples and group them with subject and load those documents into index.

INTRODUCTION

Huge amounts of Linked data are converted to lists of triples and group them in documents for subject and load them into an index. The related documents will be searched on keyword searching process. The nodes in graph represent indexes. Relation among the documents is given as edges. The edges can be weighted based on how related the documents are, highly related or important nodes will have highest probability of leading to path. If the resources are linked to as many as possible other resources, the highly linked resources are hubs. The relationship among the nodes is represented as undirected labeled and weighted edges. If every document is related to all other documents then the graph will be complete graph. If the documents are not related to each other there will not be an edge between them.

Path finding refers to finding a path between two nodes in a graph [2]. Several algorithms have been proposed to solve this issue in graphs. The two most common algorithms are Dijkstra's and A* algorithms. The two most common algorithms are Dijkstra's and A*. The former finds a path by selecting nodes with the shortest distance to the source. This distance is calculated using the weight of the edges, resulting in the optimal path. The latter extends Dijkstra's algorithm with a minimal approximated distance, based on a provided heuristic, between a node and the end node. This allows the algorithm to evaluate fewer nodes, which increases its performance.

Linked Data is a method of constructing structured data so that it can be interlinked/connected and become more meaningful [1]. It is designed on standard Web technologies such as HTTP. Directly, It does not useful pages for human readers, it shares machine-readable information. This enables data from different sources to be connected and queried. Universal Resource indicator is a string of characters used to identify a resource. Such identification enables interaction with the resources over a network. The Resource Description Model or Resource Description Format used as a general method for conceptual description or modeling of information that is on web. RDF allows us to apply path-finding algorithms on the Linked data cloud to find a path between resources over a network. Applying the path algorithms on linked data has advantage of links are annotated which introduces semantics and gives the interpretation to make traverse among the nodes that gives the meaningful path.

A related study in the biomedicine domain about path finding demonstrates how graph-theoretic algorithms can be used for mining relational paths [4]. In their paper, they have proposed a scalable path finding algorithm that works on RDF data to find complex relationships between biological entities such as genes, compounds, pathways and diseases. Path finding was applied on metabolic graph by searching for one or more paths with lowest weight. The weights assigned to each compound represent the number of reactions in which it participates. They have shown that the average distance between pairs of metabolites is significantly larger in the weighted graph than in a raw unfiltered graph, suggesting that irrelevant data could be left out.

To demonstrate the ability of path finding algorithm, they have described an application everything is connected [3]. This application is able to find a path between Facebook user and any concept on DBPedia such as persons, places, locations, things etc. The found path is explained to the end user as a short story, which explains the relation between the searched concept and the search input. For this demo application, they

have indexed DBPedia so that path finding algorithm uses the DBPedia indexes to find the paths. When the path is found a story is composed of found path in runtime. For this story, demo application searches Google to find images on all nodes and present in the found path. YouTube to find movies on these nodes and Wikipedia to retrieve abstracts on the nodes. All these elements are used to compose the story at runtime. Furthermore users will not only be presented a ranking of relevant resources but a full and motivated explanation of why a certain resource is being considered relevant. There is a meaningful semantics for the entire path to discover an each resource.
The reference papers helped to understand the characteristics of the Linked data and partition the resources in to groups based on their semantics.

RELATED WORK

Path finding is a well-known problem in graph theory. Numerous algorithms are being defined. Below are the well known algorithms which are being used are DFS, BFS, Dijkstra's and A* algorithm.

Depth-first search (**DFS**): DFS is an algorithm for traversing any graph data structures. One starts at the root /source (selecting some arbitrary node as the root in the case of a graph) and involves exhaustive searches of all the nodes by going ahead before backtracking.

Rules in DFS:

- When the vertex is found, primarily visit all its adjacent unvisited vertices mark as visited.
- If no adjacent vertex is found, mark the vertex as visited. Repeat the rules.

How it works: DFS algorithm applied on below example to find path from source vertex 'S'.

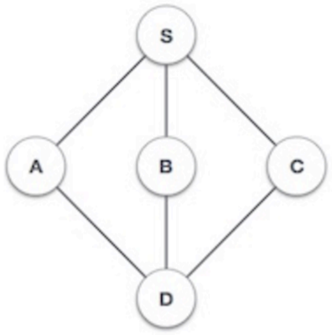

Figure 1.1

Output: DFS order of searching starts at 'S'. Below green colored nodes are found and red colored nodes are visited.

S→A
S→A→D
S→A→D→B
S→A→D→C
S→A→D
S→A
S

Breadth-first search (**BFS**): BFS is an algorithm for traversing or searching any graph data structures. It starts at the root/source node and explores the neighbor nodes first, before moving to the next level neighbors.
Rules in BFS:

- When the vertex is found, primarily mark it as visited and find all its adjacent unvisited vertices.
- If no adjacent vertex is found, mark the vertex as visited. Repeat the rules.

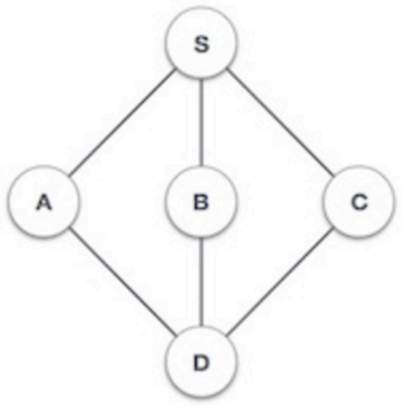

Figure 1.2

Output: BFS order of searching starts at 'S'. Below green colored nodes are found and red colored nodes are visited.

S→A→B→C
A→B→C→D
B→C→D
C→D
D

Dijkstra's algorithm: Dijkstra's algorithm is a greedy algorithm for finding the shortest paths between nodes in a graph.
Steps in Dijkstra's:

- Initialize the cost to reach the nodes to infinity and cost 'zero' to source node.
- Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the unvisited set.
- At the current node, consider all of its unvisited neighbors and calculate their tentative distances. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one. Mark the current node as visited and remove it from the unvisited set.
- Repeat till the destination node visited.

Lets consider simple application where we need to find shortest path between different cities. This problem will be modeled as graph a problem; cities are represented with nodes and routes among them with edges and distances among the cities as a cost on edges. Now let's see how a Dijkstra's algorithm is used to solve; consider we have modeled the problem into graph as:

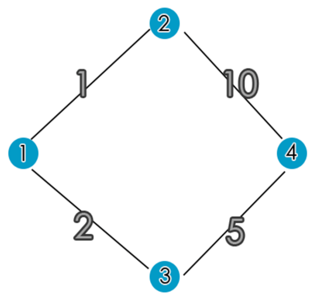

Figure 1.3

Output: Dijkstra's order of searching starts at '1' found and halts when it finds the destination '4'.

1

1→2 and 1→3

1→2→4 versus 1→3→4

1→3→4

A* Algorithm: A* is similar to Dijkstra's, but A*examines fewer nodes to find the shortest path than Dijkstra's algorithm. It uses probability theory, and heuristic search algorithms to reduce and prioritize the search space between a source vertex and a destination vertex [7]. It is widely used path finding algorithm due to its performance.

Steps of A*:

- A* algorithm uses heuristic function h(n)to guide the direction to find shortest path. Where h(n) is the heuristic estimated cheapest cost from last node in the path to goal node. Each time algorithm examines the lowest value of f(n)=h(n)+g(n); g(n) is actual distance from source node to any vertex node(n, the last node in path). Algorithm uses priority queue to perform the repeated selection of minimum costs to expand.
- At each step of the algorithm, the node with the lowest *f* (*x*) value is removed from the queue, the *f* and *g* values of its neighbors are updated accordingly, and these neighbors are added to the queue.

Below is the example for how the A* search works using f(n):

Source node is: "Arad" and goal node is: "Bucharest". Numeric values in red are calculated heuristic distance (h(n)) from node to goal node and Numeric values in blue in the graph are calculated value of f(n). The cheapest value f(n) node will be considered in shortest path and move along its neighbor's until we reach goal node.

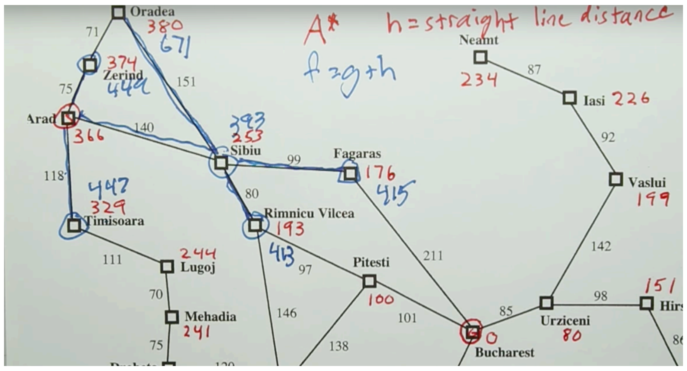

Figure 1.4

Output: Arad→Sibiu→Fagaras→Bucharest.

Analysis:

Algorithm	Time Complexity	Space Complexity
BFS	O (V+E)	O (V)
DFS	O (V+E)	O (V)
Dijkstra's	O (|E|log (|V|))	O (|V|)
A*	O (|E|)	O (|V|)

Table 1.0: Analysis of various algorithms considered.

BFS Applications and DFS:

- Testing whether graph is connected
- Computing a spanning forest of graph
- Computing, for every vertex in graph, a path with the minimum number of edges
- Computing a cycle in graph
- To check if the given graph is bipartite
- Computing the diameter of a tree (largest of all shortest-path distance in the tree)
- But, BFS and DFS are suitable to find shortest path when the edges are not weighted

Difference between A* and Dijkstra's:

- A* and Dijkstra's can be applied, if the graph has positive edge weight
- Dijkstra's is special case of A* where heuristic is zero
- Heuristic with A* makes the path finding more efficient

Here, we have modeled the problem of finding path between linked resources as undirected graph with edge weights non-zero positive using semantic similarity. Since, the edge weights are non-zero BFS and DFS are not suitable algorithms to be applied in this case. Therefore, we can *go with either A* or Dijkstra's,* as the graph is very large in real time using heuristic makes *A* efficient compare to Dijkstra's*. *Therefore, we will be using A* algorithm with a simple heuristic* in our paper.

PROPOSED ALGORITHM

The proposed algorithm takes a start and destination resources as an input and returns the possible path between them. It has two parts:

1. Pre-processing
2. Graph Browsing

Introduced approach finds path in Linked data cloud making use of above explained A* algorithm. For this, they have indexed the resources to speed up the search process.

Pre-processing:
In this step, the source Linked data is made into list of triples and group in document according to subjects and load into an index to speed up retrieving resources. The index contains references (URIs) of all the resources we consider in our dataset. An index is an efficient method to instantly retrieve a resource given a match pattern. We would receive hundreds of instruction per second in real time to achieve the performance; we need to optimize the data structure and load into index.

Graph-Browsing:
Once we prepared the indexing, perform the path finding algorithm given a source and destination point. Algorithm iterates over resources the edges are verified and would lead a path. The output path will be a source and destination node along with the nodes that are connecting them. The algorithm will iterate over the resources that may contribute towards the path, the links between the resources are verified to make a decision whether it is an acceptable or not. This assures the efficiency of the path and discards the trivial paths. To ensure the meaningful paths to be returned, have to optimal use of the semantics properties of the data. The execution of the algorithm is analyzed in three steps:

a) Initialization
b) Iterations
c) Termination

Initialization:
We start by fetching all the children for the start node, named source and the destination node, named target. We define a global set that has references to all resources. For Example, in table 1.1, green colored nodes are source and destination. All the red color nodes are children/connecting nodes of source and target that are stored with references to original node as in example table 1.2.

Resources
: Paris
: Barack Obama
: France
: Eiffel Tower
: United States

Table 1.1

: Paris
France: capital (label on link)
Eiffel Tower: Monument

Table 1.2: Paris is parent/source node; France and Eiffel Tower are children/connecting to nodes to source node.

: Barack Obama
United states: president of

Table 1.3: Barack Obama is parent/source node; United States is a children/predicate nodes to source node.

We transform these data structures to an adjacency matrix, which represents the link between the resources. Adjacency matrix lets us to perform A* algorithm on it. We store the resources in a list, the index position of the resource in the list correspond with rows and columns position in adjacency matrix. For Example.

Resources=(0= Paris, 1= Barack Obama, 2=France, 3=Eiffel Tower, 4=United States)

Adjacency matrix: In this case, we get a symmetrical sparse matrix, as most of the cells are 0. Most of the cells are 0 because there is no direct link between most resources. Note that we do not distinguish forward and backward links, this has the benefit of resulting in a symmetrical matrix. For Example: France is linked to Paris as ”has capital” and the inverse link ”is capital of” is equally important. Only when there is a parent-child connection or vice-versa a cell gets value 1. To avoid self-loops we set the links between the same resources to 0.

	0	1	2	3	4
0	0	0	1	1	0
1	0	0	0	0	1
2	1	0	0	0	0
3	1	0	0	0	0
4	0	1	0	0	0

Table 1.4: Row and column 0 show a link with row and column 2 and 3, which correspond in the list resources with Paris, France and Eiffel Tower respectively.

Iterations:
Adjacency matrix is been created in the initialization, we prefer the A* algorithm, because of the increased performance. Thus apply A* algorithm on matrix to find the path. Primarily, assign the weight on link using semantic measures. This measure is introduced to optimize the quality of the links [1].

degree (node)=sum (node links)
weight (parent node, child node)= log (degree (parent)+degree (child))

This semantic weight measure is perfectly suited as a metric for weighting the paths. It was introduced to optimize the quality of the links. Rare nodes, nodes with low probability that a random walk returns to the same node, lead to better and more interesting paths. It was prove that considering weight as the sum of the degrees of each node is a valid measure.

A* algorithm needs the heuristic for calculating the distance between nodes, which allows the sorting of the links in order of probability of leading to a path, without having to calculate the actual distance, resulting into a performance gain. As a suitable heuristic they have chosen Jaccard distance which complementary for Jaccard coefficient [8]. It measures the dissimilarity between the data sets and it is one of the most efficient methods for calculating semantic dissimilarity. The Jaccard distance is 0 when two nodes have similar features, it is symmetric, and obeys the triangle inequality. If the two nodes have many other children nodes in common then they are highly related to each other. That means we can find a path between them. We obtain the Jaccard coefficient by dividing the difference of the sizes of the union and the intersection of two sets by the size of the union.

$$\text{Jaccard}_{\text{sim}}\,(\text{node A, node B}) = (||\text{node A} \cap \text{node B}||) \,/\, (||\text{node A U node B}||)$$

$$\text{Jaccard}_{\text{coefficient}}\,(\text{node A, node B}) = (|\text{AUB}|-|\text{A intersection B}|)/|\text{AUB}|$$

Once the weights and heuristic is defined for each node, we can apply algorithm to find path using adjacency matrix. If no path is found in the first iteration, find the next level children add to resources set, will be used in next generation. The child resources added in each iteration form a generation. If we have found a path algorithm halts.

Termination:

A stop condition prevents the algorithm from running infinitely when no path found. When there is no path between source and destination, then no path will be found, the algorithm stops after certain arbitrary amount time or iterations that no of iterations depends on the dataset and target application.

Algorithm:

Input data:

Start: source

Destination: target

Result:

Path between source and target

```
Path_Algorithm:
adjacency_matrix = initialize (start, destination)
iteration = 0
path = False
stop_condition = not path and iteration < MAX
while stop_condition:
path = iterate (adjacency_matrix)
iteration += 1
termination (path)
```

Algorithm 1: The algorithm iterates over the adjacency matrix until the stop condition is met.

OPTIMIZATION

One of the main issues with in this approach is that time to create the adjacency matrix increases exponentially and the required memory space for the large graphs. If we notice that is due to the adjacency matrix becoming too large for the large data sets. To avoid the adjacency matrix becomes too large to process, if we could ensure a limited number of resources to check in each iteration and still have a way to increasing the probability of finding a valid path in each iteration the problem will be solved. In order to increase probability of finding a path with each new iteration, we estimate which resources are the most important and drop those who are not. Important nodes have the highest probability of leading to path and thus have links to many other important nodes.

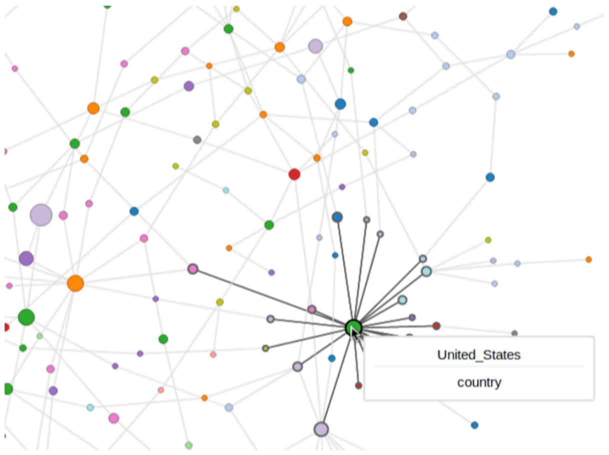

Figure 1.5: The relations between found paths expose frequently returning resources such as United States.

Thus, we link resources to as many as possible related resources; resources that are linked to a lot of resources are called hubs. All relations in linked data have an inverse relation that is equally important. Here, important nodes are frequently visited nodes. Once of the centrality measure is the Page Rank algorithm [5]. A page-ranking model that reflects the importance of a particular page as a function of how many actual visits it receives by real users. A page ranking model that reflects the importance of a particular page as a function of how many actual visits it receives by real users is called the intentional surfer model. Using centrality we can find reduced rank approximation to the adjacency matrix. We would use these highly ranked nodes in the iterations.

RESULTS AND EVALUATION

In order to assure this approach, Perform a sample search using the above algorithm by storing data about retrieved paths: source, destination, all the hops of the path with the meaning of the links between them and the execution time. We check the average length of found paths and we measure the fraction of paths found within various time frames. A found path is relevant if it occurs within a tolerable time for the users.
Based on the context and the size of the dataset this time may vary. The paper has discussed the hit rate, distribution of execution time and path lengths for a test set containing 10000 random path calculations randomly among 200 DBPedia resources (popular cities, countries or brands). Considered the stop condition for the algorithm on a path length of 12.

Meaningfulness: Given a dataset with popular cities, countries or brands, country related information. Input as source: Barack Obama, destination: Paris. The found paths should not be trivial, for example Paris and Barack Obama could have been linked because Barack Obama lives in the White House in Washington DC. Both Paris and Washington DC are cities and this would be a very short and relevant path.

The cost/hit rate analysis of the given algorithm over test set considered above resulted hit rate 95%, considering the relatively small number of resources that had to actually checked compared to the size of the entire dataset. Checking a resource means retrieving the resource from the index and identifying the linked resources. The number of resources checked is less than 6000 in most of the cases, these results indicate that popular concepts on DBPedia are well interlinked and form a dense graph. In this case, the most of the resources are densely connected; therefore, the mean path length is about 4 hops.

Execution time: The time complexity of A* algorithm depends on the complexity for the evaluation of the heuristic [6]. The evaluation of our heuristic, the Jaccard, is linear to the number of children for a resource. If we apply the optimization of Adjacency matrix, we would retain the linear time execution.

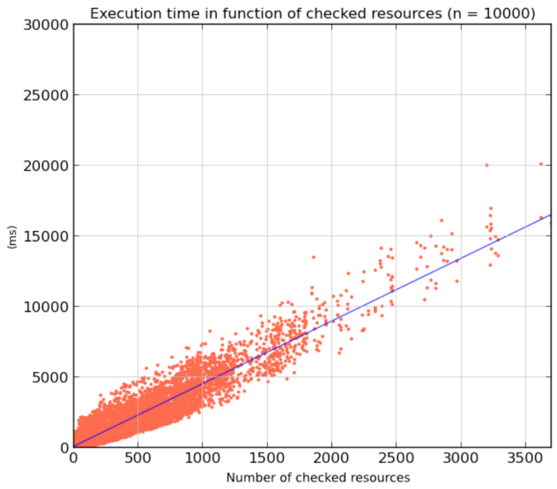

Figure: Execution time (y) is approximately a linear function of the checked resources (x); y ≈ 4.4x +k

CONCLUSIONS

Path finding algorithms are especially embedded in applications such as navigation systems. In this paper we have elaborated on a more generic approach of path finding using Linked Data and an optimized A* algorithm in web-data. We tried different searching algorithm and discussed their pros and cons. Another contribution is the choice of heuristics and considering the effect they have on the result set, determining how they lead to improved performance. As a heuristic for the A*, we have chosen the Jaccard distance which keeps our execution time to be linear. In order to achieve interesting paths we have optimized the weights to take into account the semantic properties of Linked Data. We introduced a reduction step of the matrix based on node centrality. This reduction step reduces the rank of the adjacency matrix, without throwing away many possible solutions. We have demonstrated that using Linked data in combination of indexing opens door to us to find path on large datasets in the tolerable time.

RESOURCES

1. Laurens De Vocht, Sam Coppens, Ruben Verborgh, Miel Vander Sande, Erik Mannens, Rik Van de Walle, **Discovering Meaningful Connections between Resources in the Web of Data**, 2013

2. B. Cherkassky, A. Goldberg, and T. Radzik. **Shortest paths algorithms: theory and experimental evaluation**. Mathematical programming, 1996

3. Miel Vander Sande, Ruben Verborgh, Sam Coppens, Tom De Nies, Pedro Debevere, Laurens De Vocht, Pieterjan De Potter, Davy Van Deursen, Erik Mannens, and Rik Van de Walle, **Everything is connected: Using Linked Data for Multimedia Narration of Connections between Concepts**, 2012

4. B. He, J. Tang, Y. Ding, H. Wang, Y. Sun, J. H. Shin, B. Chen, G. Moorthy, J. Qiu, P. Desai, and D. J. Wild. **Mining relational paths in integrated biomedical data**. PLoS ONE, 6(12): e27506, 12 2011

5. C. H. Q. Ding, X. He, P. Husbands, H. Zha, and H. D. Simon. **PageRank: Hits and a unified framework for link analysis. In SDM**, 2003

6. F. Nah. **A study on tolerable waiting time: how long are web users willing to wait? Behavior & Information Technology**, 23(3): 153–163, 2004

7. P. Hart, N. Nilsson, and B. Raphael**. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics**, 4:100–107, 1968

8. S. Kulkarni and D. Caragea. **Computation of the semantic relatedness between words using concept clouds**. In KDIR, pages 183–188, 2009