Six Degrees of Separation

Pichayut (Petch) Jirapinyo, Linfeng Yang

Abstract

Six Degrees of Separation is the idea that any two random people in the world are at most six degrees apart in terms of social connections from each other. We hypothesize that there exists a similar property for proper nouns. With abundant information resources on the Internet such as Wikipedia, we should be capable of studying the connections between proper nouns. Nevertheless, the problem is a challenging search problem, as information on the Internet is undoubtedly large and full of irrelevant information. In this project, we attempt to use different search algorithms and heuristics to tackle the issue of finding paths that connect two proper nouns together.

1. Introduction

Six Degrees of Separation is the idea that any two random people in the world are at most six degrees apart in terms of social connections from each other. We wonder whether there is a similar property for any random pair of proper nouns in the real-world knowledge base. For instance, there could be multiple paths connecting *Bangkok* to *Beijing*. A more direct path could be that they are both Asian cities, whereas a more interesting path could be that *Bangkok* is the capital city of *Thailand*, which sent delegates to the *2008 Summer Olympics*, which was held in *Beijing*. We are interested in using data obtained from the Internet to uncover hidden connections between two proper nouns in interesting ways. The goal of the project is to come up with an algorithm and a set of heuristics that would make the search most efficient but at the same time yield interesting results.

Specifically, our program takes as inputs two proper nouns in natural language and the number of paths connecting the two proper nouns the user wants to search for. Then, to resolve the ambiguity between objects sharing the same name, the program prints out a short description of each of the objects and asks the user to choose which of the objects she is actually interested in. Subsequently, the program starts running a pre-chosen search algorithm with pre-chosen heuristics to determine a path between the two objects. A path is defined to be a series of objects that connect to each other via some type of relationship, starting from the start object and ending the goal object. The connection between two nodes has the relationship type, e.g. "capital city of", "sent delegates to", or "was held in", as its value. The program prints out paths connecting the two objects and the relationship types of all the objects along the paths.

The project is interesting in two ways. First, it could be a fun, interactive game or tool. We believe that there are oftentimes some unknown but interesting facts that two objects have in common. For example, not many people at Harvard know that both *Harvard University* and the *California's Great America* theme park in San Francisco are of the same total land area. The well-known physicists *Galileo Galilei* and *Stephen Hawking* were both born on January 8th. Second, the project could be used as part of a system for related topic discovery. By using this program, we can tell how related two random words are through some aggregating function, such as the average length of the paths between the two objects or the number of paths with six degrees or fewer. It is possible to use a similar system to this project to recommend related topics the user might be interested in, while she is browsing on some webpage. As an ambitious, final

step of this project, we will attempt to make it into a website for the general public. This last step would involve much more optimization and caching.

In addition to the fact that we found the project interesting, we also think that the problem space of the project is computationally challenging. First, we are doing search on real world, half-user-generated and half-regulated data, which means that our search problem could be very large and full of irrelevant information. Branching factor is an issue in our search space, as each node can connect to thousands of other nodes. Besides, as we solicit data in real-time, expanding a node is very costly. Lastly, with online, real-world data, there are many information-based and graph-based heuristics we can use. Information-based heuristics are those that involve understanding the nature of the actual content, whereas graph-based heuristics are those that infer some information about each of the nodes through the structure of the graph.

The rest of the paper is organized as follows. In Section 2, we explain related works and our research into the problem space. Section 3 explains the process of obtaining data and the nature of the data itself. Section 4 describes the basic algorithms we implemented in the project and the desired properties of search algorithms. Heuristics we used, as well as other fine adjustments we made to the search algorithms and the search space, are explained in Section 5. Experiment results are shown in section 6, before we mention possible future works in Section 7.

2. Background

2.1 Related Works

There are many existing online systems that involve representing data using nodes and infer implicit information through the connections between those nodes and the structure of the resulted graph. Google's PageRank algorithm uses the graph structure to represent the connections between websites, and assumes that nodes with many incoming links are interesting and trustworthy[i]. The transitive trust system, which attempts to infer how much Person A should trust Stranger C, based on how much Person A trust Person B and how much Person B trust Person C, also uses many graph-based algorithms, such as MaxFlow, ShortestPath, PageRank, and RandomWalk[ii]. Closer to our problem space is the related topic sidebar on the website of the Harvard library's Hollis Catalog, which makes nouns into nodes in the graph and connects them based on relevancy[iii]. Nevertheless, we have not seen any system similar to ours, which attempts to find connections between any two proper nouns through graph search.

2.2 Wikipedia and Natural Language Processor

Originally, we planned to use Wikipedia[iv] as our source of data, as it is an arguably richest free source of information on the Internet. Our plan of attack was to scrape a respective wikipedia page and then use an open-sourced natural language processor to try to understand the sentence structure. However, we decided to not use this approach due to various reasons as explained in Section 2.3.

2.3 Freebase

Nonetheless, we decided to use the Freebase API instead. Equipped with free API, Freebase is a structured database based on Wikipedia and user-generated contents.[v]

Freebase treats all articles as nodes, and nodes can connect to each other via some descriptive links. For instance, the node */en/coldplay*, which represents the English band Coldplay, connects to the node */guid/9202a8c04000641f800000000839684a*, which represents the song Viva la Vida, via the link */music/track/recorded_by*, which represents "recorded by." Freebase API queries can be done via HTTP using Freebase-specific Metaweb Query Language (MQL), which resembles JSON. We decided to opt for Freebase instead of Wikipedia due to many reasons. First, no natural language processors will be good enough for us to extract the relationship between two objects the way Freebase provides us, meaning that using Wikipedia, we might not be able to make sense of the path we find. Second, since everything on Freebase is an object, it provides us a much easier way to deal with ambiguity between objects of the same name than using a natural language processor. Lastly, we believe that a Freebase query is much faster than running a natural language processor on a scraped Wikipedia page. We explain the data Freebase provides us in details in the following section.

3. Search Space

3.1 Vertices

Everything on Freebase, including articles, community portals, links, images, properties, user data, and Freebase's internal properties, is represented as a node. Each node has a type property, which lets us know which type(s) that particular node belongs to. We prune out unnecessary nodes, based on the types. Each node also has a short description and a human-readable text, which we use for interacting with our user.

3.2 Edges

There are two types of links on Freebase. The first type of link (we call "object link") connects two objects together with an encoded explanation of how they are related stored on the link. The Coldplay-Viva la Vida example from the previous section illustrates this type of link. All objects and links are in encoded version, e.g. */guid/9202a8c04000641f800000000839684a* representing *Viva la Vida*, and hence we have to get the human-readable version of the link and the object. The second type of link (we call "property link") connects an object to a property treated as an object, with a value on the link. For example, */en/harvard_university* points to the object */education/educational_institution/founded* with a value on the edge of 1636. Both types of links are directed.

To make Freebase work for our search space, we come up with three categories of links for an object of interest, A. Obtained from the "object link", "TYPE 1" links go out from A to some other object nodes. Similarly to TYPE 1 links in that it is obtained from "object link", "TYPE 2" links come from some other object nodes into A. We can see that "TYPE 1" and "TYPE 2" edges are perfectly invert of each other. To use "property link", we query all the objects that share the same value under all properties as A. For instance, we query all objects that link to */education/educational_institution/founded* with a value of 1636, and get all educational institutions that were founded in 1636. We call this type of link "TYPE 3".

There are 3 types of links. TYPE 1 and TYPE 2 are normal links that bring us to nodes that are likely to be closer to the current node in the search space (e.g. From "China" to

"Beijing"). TYPE 3 is more interesting in terms of that it can bring us to some wild and completely irrelevant nodes (e.g. From "Harvard University" to an amusement park called "California's Great America", just because they have the same area size). This is important to jump around the search space quickly. For a comparison of local search, following TYPE 1 and 2 links is like hill climbing and following TYPE 3 is like having a random restart.

We can see that all edges on our search space are virtually bidirectional, since any TYPE 1 edge always has a corresponding TYPE 2 edge that goes in the opposite direction and TYPE 3 edges are undirected by their nature.

For each edge, we derive a "relationship type" (e.g. TYPE 1 + */location/location/contains*) and a corresponding, human-readable "relationship text" (e.g. *contains*) which explain the connection between the two nodes, based on Freebase's link properties and our 3-TYPE scheme.

3.3 Supporting Routines

The backend of our interface that talks to Freebase involves sections that resolve ambiguity between objects of an identical name, take care of timed-out requests, get rid of incomplete dataset, prune out unimportant nodes based on types, and translate Freebase links into our 3-TYPE link scheme.

It is important to note that after basic pruning of irrelevant types, some nodes still have thousands of edges. For instance, *United States* "contains" all states, all cities, famous places, and other notable objects. Thus, basic pruning does not help with the issue of large branching factor.

4. Basic Search Algorithms

This problem is essential about searching, thus choosing the fit basic search algorithm is crucial. We defined the search space in Section 3. In this section, we introduce various algorithms we have implemented.

4.1 Algorithm Properties

First thing first, we need to choose desired properties for our algorithm. Based on that, we can then design algorithms to satisfy those properties. We can also use the properties as metrics to evaluate each algorithm.

4.1.1 Interestingness

Interestingness means how interesting the paths found are. For example, suppose an user asked paths between "Mao Zedong" (the first chairman of People's Republic of China) and "Harry Potter". After waiting for a while, the application simply gave the following output:

```
Mao - Gender: Male
Barack Obama - Gender: Male
```

which is probably not terribly interesting and the deeply pissed user would probably never turn back to use our application again. On the other hand, the following path may look (arguably) more appealing:

```
Mao Zedong - Place of death: Beijing
Beijing - Inversion of Subjects: The Interior
The Interior - Inversion of Books In This Genre: Mystery
Supernatural - Inversion of TV programs of this genre: Mystery
Human - Found in fictional universe: Supernatural
Harry Potter - Species: Human
```

There is no good measurement for interestingness other than human judgment and it is hard to guide the search towards more interesting results. Nonetheless, there are several basic things we can do, such as avoid using gender as a node along the path. We will talk about this in details in Section 5.

4.1.2 Performance

Performance is the time it takes to get desired number of solution paths. For Freebase, expanding a node takes at least 1 second. This is the minimum cost we need to pay to connect Freebase and to ask for results. If the node has many neighbors, the time will be much longer. Thus the cost of expanding is high. To improve performance, we want to expand as few nodes as possible.

4.1.3 Completeness

In this particular problem, for any search algorithms, there are two sides to consider for the completeness. The first side is that if there is at least one path between two entries, the algorithm should find the path. The second side is that if we let the algorithm run long enough, it should find all the paths. There is a trade-off between completeness and speed of the algorithm.

We certainly do not care about the second side. Given the size of the search space, for any given two nodes, there are a great number of paths. Not only it is almost impossible to find all of them, but also most of our users would not bother to see that many results. On the other hand, it is easy to argue either way for the first side. To speed up our searching process, based on some heuristic we may decide to prone some paths which are unlikely to lead us to the goal node in short time. This violates the second side of completeness and we need to make the trade-off decision. Nonetheless, in most of the cases, there should be many paths between two nodes. There is a large probability that even if we decide not to follow some paths, we can still find enough solution paths that the user asks. Thus we decide to suffer some potential completeness of our algorithm for much better performance.

4.1.4 Search Depth

In tree search algorithms, optimality means the algorithm should return solutions with the lowest costs. In our case, we can consider the solution with shorter path (i.e. degree) to be better. However, we do not think the length of the path has anything to do with the interestingness of the solution. As we can see from section 4.1.1, longer path can be more interesting than shorter path. On the other hand, short path can be interesting. Consider the following case:

```
J. R. R. Tolkien - Inversion of Screenplay by: The Lord of the Rings film
trilog
The Lord of the Rings film trilogy - Notable filming locations: New
Zealand
```

This solution is short but captures the relationship between the two nodes well. However, the following longer version makes less sense:

```
J. R. R. Tolkien - Country of nationality: United Kingdom
United Kingdom - Inversion of Basin countries: Loch Awe
Loch Awe - Inversion of Contains: Argyll and Bute
Argyll and Bute - Inversion of Contains: Scotland
Scotland - Inversion of Country of origin: Rab C. Nesbitt
Rab C. Nesbitt - Same value of 'Number of seasons' (8): NZ Performance Car
TV
NZ Performance Car TV - Country of origin: New Zealand
```

It is kind of "cool" that the application can find that "Rab C. Nesbitt" and "NZ Performance Car TV" (both are TV programs) have the same "number of seasons", but it probably does not matter much to a user who is interested in "Tolkien" and "New Zealand". In general, short paths can be view as either "precise" or "boring", and long paths can be either "surprising" or "irrelevant". Since the search depth has little to do with interestness, we decide not to consider this as a requirement for our search algorithms.

4.1.5 Decision

The ideal algorithms for this problem should give interesting solutions in reasonable amount of time. Interestness and performance are our major concerns. We consider completeness and search depth to be less important.

4.2 Algorithms

4.2.1 Limited Depth First Search (DFS)

The first algorithm we implemented was DFS. Given the branching factor and the size of the search space, we expect DFS to perform poorly. Once the algorithm picked a node to expand in the first level, it may never come back to other nodes in the first level, thus the penalty of picking the wrong path is large. Given the large branching factor, we use depth limit search instead of the pure DFS and arbitrarily set the limit to 6 or 7 (root node has depth 0). We expect the average length of solutions given by DFS to be large. Not surprisingly, we use a search stack to track all the nodes waiting to be expanded

4.2.2 Breath First Search (BFS)

The average length of solutions given by BFS should be much smaller than that of DFS. We think most nodes should be connected through at least several short paths. In these cases, we expect BFS to work much better than DFS. But for nodes are not connected through short paths, BFS will have a huge penalty since the branching factor is large. We use a search queue to track all the nodes waiting to be expanded.

4.2.3 Iterative Deepening Search (IDS)

IDS performs similar to BFS but with much less memory usage. We do not find memory size to be a limit by far thus we did not end up implementing it.

4.2.4 Best First Search (BeFS)

BeFS uses heuristics to rank the order of nodes to expand. It does not make much sense to talk about BeFS without considering heuristics. With good heuristics, we expect BeFS to perform well. Different from DFS and BFS, BeFS can go both deeper and broader at the same time thus the path lengths of solutions vary a lot. This can give us diverse solutions, which tend to be more interesting. We consider this as an advantage of BeFS. We use a list to track all the nodes and sort all of them whenever we have new nodes inserted into the list.

4.3 Direction

4.3.1 Uni-direction

Uni-directional search is simple and standard. We ask user to give a starting point and a goal proper noun. Then we start from the starting node and try to find paths to the goal node. This is straightforward to implement.

4.3.2 Bi-direction

Bi-directional search is more interesting. It fits perfectly into our search problem in two ways:

1. We have two nodes provided by the user. We can search from both of them and try to find a path to each other.

2. All of the edges in our search space are bi-directional. We can search from either direction (start to goal or goal to start) without affecting the completeness of the result. If the edges are only uni-directional, we can only search from start to goal. Searching backwards will not generate the same path. Then it will be no longer possible to do bi-directional search.

We implemented bi-directional search for all the three search algorithms. We start from both of the initial nodes provided by the user and try to find a path to each other. There are two search trees, the start-side tree *T1* and the goal-side tree *T2*. One thing to point out is that we use only a single thread for our application. The thread expands nodes from *T1* and *T2* alternatively -- one from *T1*, then one from *T2*, then one from *T1*, and so on.

To find a path in bi-directional search, instead of comparing the current node against the goal node, we find the "pivot node" instead. A pivot node is a node that is in both search trees. Once we find a pivot node, we can use the path going through it as a solution path. Thus, when we see a new node, we should check it against all the nodes in the other search tree. To make this checking process faster, we use two hash tables to store all the nodes in *T1* and *T2*. This in most cases reduces the checking time from linear to constant. Since the time for expanding nodes dominates the overall running time, we consider the overhead of goal checking to be acceptable.

We expect bi-directional search to perform much better than uni-directional search. The total number of nodes in the search tree should be much smaller since we start from both directions and each search tree can have only half of the search depth comparing with uni-directional search. We set the depth limit for all three algorithms to be 3, thus

the maximum path length is 6. For bi-directional search, the maximum number of nodes in both search trees is only b^3 where b is the average branching factor, much smaller comparing with b^6 in uni-directional search for depth limit 6.

5. Search Techniques

Given the special properties of our search space (large number of nodes, large branching factors, high time cost of node expanding), we proposed several search techniques to improve performance and interestingness.

5.1 Look-forward

We implemented 1-step look forward, i.e. whenever we get a node from Freebase, we first check the node against the goal before inserting it into the search stack / queue. With this, our search algorithm can see much more nodes than it actually expands. This is particularly useful in our case since expanding any node is expensive time-wisely.

For bi-directional search, look-forward is more interesting. As mentioned above, we constantly check a new node against the other tree's search tree. We future expand this concept and call this **vision**. Vision includes all the nodes this tree has seen so far. Not only we check nodes before inserting them into the search stack / queue, we also insert them into the vision as well. The two trees are constantly checking against visions to each other. Our search tree can see much further than it actually goes.

The overhead of 1-step look-forward is very small (we need to get all these nodes anyway). Again, the dominant time here is the cost of getting data back from Freebase. In practice, look-forward performs very well. The number of nodes in vision is always much larger than the number of nodes expanded.

5.2 Loop Detection & De-duplication

Because our graph contains loops, we need to somehow detect loops. Also we do not want to expand the same node twice so we need some methods to detect duplications.

To do this, we store all the nodes we have expanded in a set *S*. Before picking a node *N* from the search stack / queue and expanding it, we first check it against *S*. If it is in *S*, we skip *N* without expanding it. This works nicely in most cases except with only one problem. Consider the case in DFS where the search algorithm first expand node *N1* at depth *k*. It follows all the descendants of *N1* until reaching the depth limit *d*. These descendants have a maximum of depth *d-k* counting from *N1*. Now it returns to a higher level and find *N1* again at depth *k-1*. It should expand *N1* again since this time it can see more descendants of *N1* since now the maximum of depth for all the descendants is *d-k+1*. This is only a problem in DFS. We handle it such that if we see the node again in a shallower level, we shall expand it again.

5.3 Link Cut

5.3.1 Method I: For Freebase Result

As mentioned in Section 3, we do basic pruning when we first obtain data in order to get rid of irrelevant nodes, such as images, community portals, Freebase's user information, and Freebase's internal properties. Nevertheless, many nodes, such as country nodes, still have more than a thousand of connections left. Thus, we use a

series of heuristics to cut down the number of nodes to be passed into the search algorithm.

First, when we get all the neighbors of a node, we scan through all the "relationship type" (from Section 3.2), and downplay the relationship types that are too abundant (defined as more than 30 connections). For example, the relationship type */location/location/contains* has more than 1,000 connections for *United States*, implying that this relationship type is likely to not be very important (as opposed to the relationship type */location/country/capitals*, which contains only 1 connection). For these abundant relationship types, we only look ahead to see if there are any interesting nodes that we are looking for (in case of Bidirectional search), and get rid of all other nodes. This heuristic certainly makes the overall algorithm less optimal as there are some paths that are cut from the system, but it also makes the results more interesting, as we downplay the significance of the hub nodes.

5.3.2 Method II: For Node Expansion

We also do link cut when expanding nodes. Since the cost of expanding is high, we decide not to follow all of the links all the time. We set a quota to expand for each link type of each node. The first method mentioned in section 5.3.1 has already gotten rid of most of the links. Just in case we still have too many links of any type (more than the quota), we randomly choose a portion of them to put on the search stack / queue equal to the number of the quota.

We mentioned the value of each type in section 3.2. Type 1 and 2 links are normal differing only in search direction. We consider them to be equally important and assign them the same quota of 40. TYPE 3 is wild and we only assign it with quota 5.

5.3.3 Affects on Vision

In section 5.1 we mentioned the concept of search vision. Link cut shrinks the vision of our search algorithms. To keep the vision as big as possible, we did the following rules:

1. For method I, we still check all links against the other search tree's vision. In this case we will not miss a potential path. However, we do not insert all the links cut into the current search tree's vision, since this requires much more performance / memory overhead in our implementation.

2. For method II, since the total number of links is much smaller after link cut with method I, we check all links against the other search tree's vision and also insert all these links into the current search tree's vision even that we decide to cut them and not to put them into search stack / queue.

5.4 Ranking Heuristics

Ranking heuristics can give us a general guidance on the order to expand nodes. This is crucial since our search space is huge. A good search order can save the application a lot of time. One heuristic we come up with is the relevance of the node to the goal node. The more relevance it is, the more likely we can find a path to the goal by following this node's descendants in a short time.

5.4.1 Bing Heuristic

To measure the relevance, we combine the text of a node *N* with text of the goal node *G* together and search this query on the Microsoft search engine -- Bing. We use the total number of results as a general guidance. The rational behind it is: if the current node shows more times together with the goal node in many articles, it is probably relevant to the goal node.

However this shows only the one-level relevance. We still consider it to be a good guess. In practice, we find this heuristic tend to lead us to larger nodes such as country names. For example, the number of results of searching any arbitrary word (this is the goal word) along with "United States" is probably going to be larger than the number of results of searching this goal word along with a more relevant word. Thus this heuristic will decide to expand "United States" first. This in many cases can lead to solutions quickly without expanding many nodes. However the solutions may not be that interesting since there are many country names involved (e.g. this person and that person have the same nationality...)

5.4.2 Normalized Bing Heuristic

To overcome the issue of Bing Heuristic of favoring large nodes, we proposed another heuristic. Suppose B(*x*) is the function returning the number of results on Bing when search *x*. For any given node *N* and goal node *G*, Bing Heuristic returns:

$$B(N.text + G.text)$$

Normalized Bing Heuristic will instead return:

$$B(N.text + G.text) / B(N.text)$$

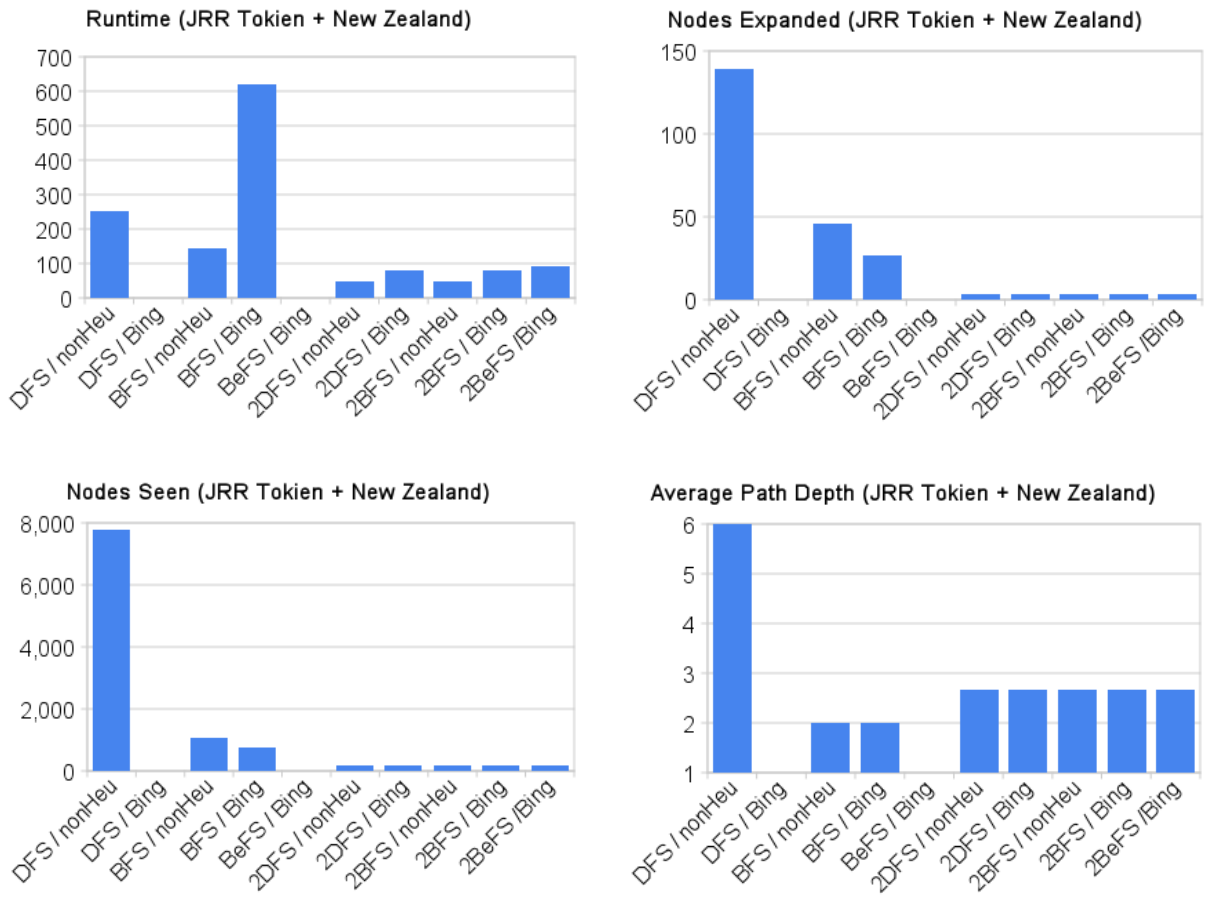
By dividing the number of results getting from Bing by only searching the node, we can get the ratio of relevance without favoring large nodes. However in practice, this does not work well since it almost always choose very small nodes (usually with very long names) and lead to no solution paths.

5.4.3 Heuristic Performance

Bing has an API through which we can query and scrape the result to get the number of results. However this process is also slow and there is a Query Per Second (QPS) limit per IP address. Even that we do queries in multiple threads, we have on average only QPS = 2. This makes Bing heuristic very expensive. In practice, even that using heuristics can help us expand fewer nodes before getting desired number of solution paths; the overall time may still be longer.

6. Experiments

The nature of our project does not allow us to run extensive experiments on different algorithms, as it takes approximately 1 second to get data from Freebase per node and another second to get a Bing heuristic value. However, to test different algorithms and heuristics, we used two pairs of objects - JRR Tokien to New Zealand (Easy) and Harvard University to Coca-Cola (Hard).



Figures 1 (top left), 2 (top right), 3 (bottom left), and 4 (bottom right)

6.1 Easy Experiment

For the first pair (JRR Tokien and New Zealand), we set the depth limit of all search algorithms to 6 and the number of paths we want to be 3. This pair is intended to be an easy case, as we know that the best solutions are of depth 2. The results are as shown in Figures 1, 2, 3, and 4. ("2" in front of the search algorithm name refers to "Bi-directional search", whereas "BeFS" refers to "Best-First Search".)

In terms of runtime (Figure 1), algorithms with the Bing heuristic does somewhere between 59% and 416% worse than those without. This is largely due to the fact that it takes about 1 second to retrieve data from Bing for each of the nodes seen. Other than that, we find that Bi-directional Depth-First Search, Breadth-First Search, and Best-First search algorithms are much faster than all Uni-directional search algorithms.

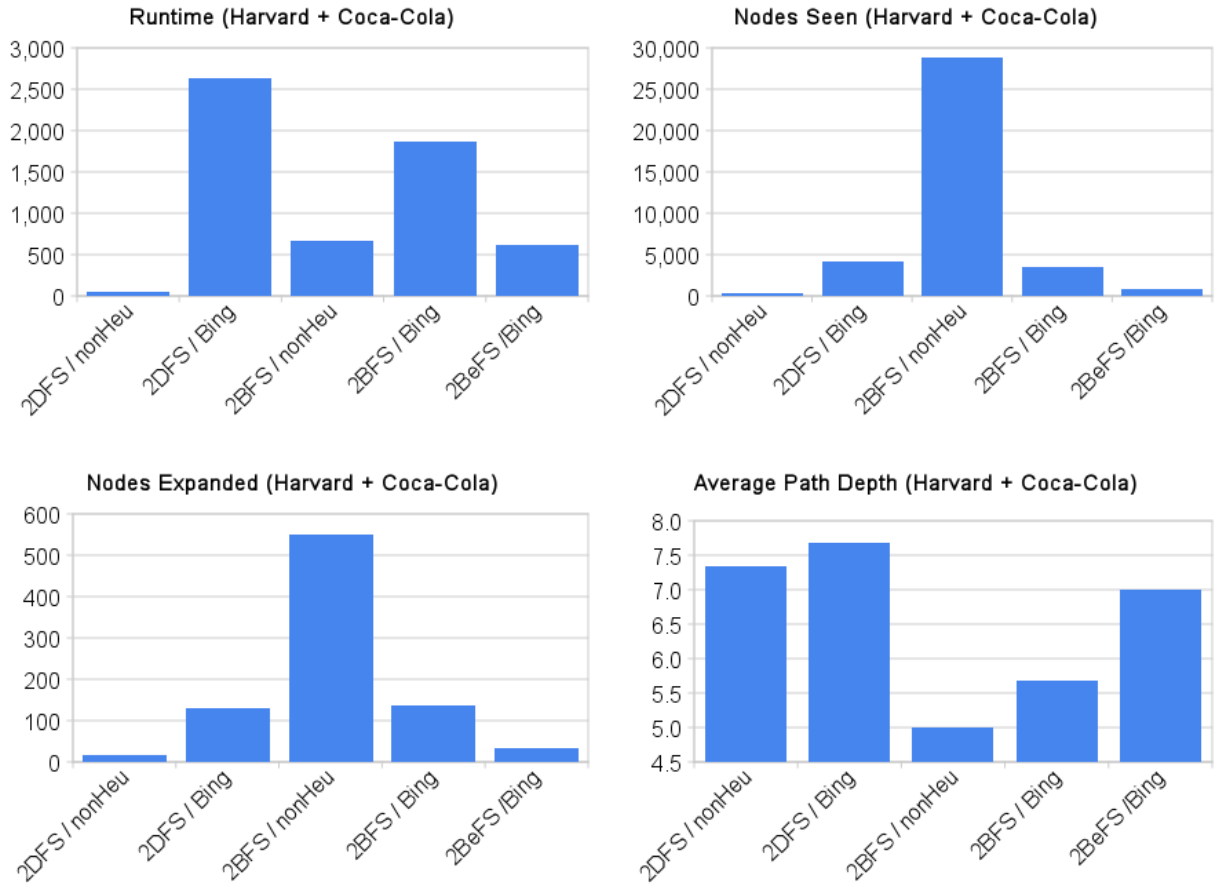
In terms of the numbers of nodes seen and nodes visited (Figures 2 and 3), all bi-directional search algorithms also fare much better than uni-directional search algorithms. In the case of uni-directional Breadth-First Search, we also notice that running it with the Bing heuristic significantly reduces the numbers of nodes seen by 29% and reduces the numbers of nodes visited by 41%.

Lastly, Figure 4 shows the average depth of the resulting paths. Expectedly, Depth-First Search yields results with longest path depths, whereas Breadth-First Search yields results with shortest path depths.

6.2 Hard Experiment

The second pair of query we ran is between *Harvard University* and *Coca-Cola,* the drink. This case is expected to be harder, as the shortest known result is 5. To make it more interesting, we set the path depth limit to 8, while keeping the number of resulting paths to 3. We run this pair with all uni-directional search algorithms, but none of them yields results within an hour. So, we discard them in this section. Figures 5, 6, 7, and 8 demonstrate the results of this query.

In terms of runtime (Figure 5), we get the same trend as in the easy experiment - the Bing heuristic slows down the algorithm significantly. Without the Bing heuristic, bi-directional Depth-First Search is 15 times faster than bi-directional Breadth-First Search. Among the algorithms that use the Bing heuristic, bi-directional Best-First Search is 3 times faster than bi-directional Breadth-First Search, which is 40% faster than bi-directional Depth-First Search.



Figures 5 (top left), 6 (top right), 7 (bottom left), and 8 (bottom right)

In terms of the numbers of nodes seen and nodes visited (Figures 6 and 7), bi-directional Depth-First Search without the Bing heuristic fares significantly better than

other algorithms, which we believe is an anomaly. Nevertheless, in the cases of bidirectional Breadth-First Search, the Bing heuristic reduces the number of nodes seen eightfold and the number of nodes visited fourfold. Excluding Depth-First Search, bi-directional Best-First Search is the most efficient.

Unexpectedly, the average path depth is shortest for Breadth-First Search and longest for Depth-First Search, as shown in Figure 8.

7. Future Works

In the future, we would like to bring our application online where people can use it freely to search whatever things they like. Before that, we need to do several things.

First of all, we need to improve the performance. We shall consider implementing Iterative Deepening Search to improve memory usage so that we can handle multiple tasks at the same time. Also we need to consider finding heuristics with lower costs, or try to combine the two ranking heuristics together to make it useful.

We also need to do more evaluation to figure out some parameters. For example, we mentioned we can caps for the number of links we take to expand. We need to figure out reasonable values for them. Also we need to find out the optimal depth limit.

Another thing we find out in practice is that all the algorithms tend to return multiple similar solutions. For example, Russian and New Zealand have attended many Olympic games together. As a result, searching from Tolkien to New Zealand returns multiple results only differ with the year of the Olympic games. We can try random restart when find a solution or use de-duplication methods.

Given the high cost of expanding node from Freebase, we may also try scraping Wikipedia instead in the future.

8. Conclusion

We implemented the application with DFS, BFS and BestFS—both uni-directional and bi-directional algorithms. We proposed and implemented several search techniques and heuristics to make the application perform better. We evaluated different algorithms briefly and find out that bi-directional is much better than uni-directional search. BeFS is in general better than DFS and BFS if the Bing Heuristic takes shorter time to perform. Our search techniques work well though.

We conclude that the best search algorithm in practice with this problem, in terms of running time, is the Bi-directional DFS without ranking heuristics, with look-forward, loop detection, de-duplication and link cut.

[i] A N Langville, C D Meyer and P FernÁndez. Google's pagerank and beyond: The science of search engine rankings. *The Mathematic Intelligencer*, 30(1), 2008.

[ii] E Friedman, P Resnick, and R Sami. Manipulation-resistant Reputation systems. *Algorithmic Game Theory*, 2007.

[iii] Harvard Hollis Catalog, http://hollis.harvard.edu/.

[iv] Wikipedia, http://www.wikipedia.com/.

[v] Freebase, http://www.freebase.com/.