CS 4701 Final Report

Team Mixed Bag
Daniel Vicuña (dav74), Zachary Hurwitz (zrh7), Sasha Badov (sb965)

14 May 2019

Keywords: Natural language processing, Wikipedia Game

1 Project Description

1.1 Background

Our goal for this project is to create an AI for the Wikipedia Game [1][2]. The Wikipedia Game is played as follows: players start on the same randomly selected source Wikipedia article and must navigate to another randomly selected target Wikipedia article using only the hyperlinks within the article. The goal of the game is to use the fewest number of clicks (or the least time, depending on the version of the game).

1.2 Envisioned System

Our expectation for the baseline system is that it would be able to simulate an iteration of the Wikipedia game. That is, given a start and end article, it would be able to find and return a path provided such a path exists. The bulk of our project would consist of creating a model for an informed search algorithm using linguistic data from the articles themselves. With the aid of this additional data, we hoped to avoid traversal along paths that do not lead to the target article. We planned to experiment with different machine learning/natural language processing/information retrieval techniques to produce a weight for each node representing how semantically similar it is to the target article. We would then traverse the graph from the start point according to these weights. The motivation for this approach follows.

It is likely that the closer in space two articles are, the more semantically similar they are. Thus, the closer in space an article is to the target article, the more semantically similar it will be to the target article. So, if we can obtain some metric of semantic "closeness" between two articles, we will be able to use a best-first search to traverse the paths along the articles that are most likely to lead to the target article and avoid much needless searching. We predict (and hope) that this approach will greatly speed up the baseline search process (which will be lengthy due to the large number of neighboring nodes) by removing the need to search very broadly. The crux of this problem will be to determine how to measure semantic similarity between two articles.

2 Final System

2.1 Data Representation

We created a graph representation (implemented as a dictionary) where each node represents a Wikipedia article, and a directed edge exists from node A to node B if article A has a hyperlink to article B.

2.2 Search

To test the feasibility of our idea, we initially tested basic search algorithms as a proof of concept. We first created a breadth-first search, which was theoretically functional, but had a major flaw. Each Wikipedia article has dozens, and sometimes even hundreds of outgoing links, so it was extremely slow to run an algorithm that checked every possible article. We then implemented an iterative deepening search. This was somewhat more efficient than the breadth-first search was, but was still slow due to needing to check every outgoing link. It was at this point that we determined that the sheer size of the Wikipedia network of articles was much too large to search every outgoing link, so we needed to switch to an algorithm that only checked a fraction of the potential nodes.

We determined we would use use a priority beam search to traverse the graph. Priority beam search with beam size k expands only the k most promising children of a node in a breadth-first fashion. It thus offers a compromise between best-first search, a very greedy approach that can perform very poorly if the metric used to rank the nodes is noisy, and breadth-first search, which will provably find the node we are searching for but requires exponential space.

2.3 Similarity Heuristics

We then chose to train two models offline on the Wikipedia corpus to produce a vector that would represent each article. The metric used to rank our nodes is the cosine similarity between the vector that corresponds to the target article, and the vector that corresponds to the article we are ranking.

2.3.1 Tf-idf on Articles

Our first such model was tf-idf cosine similarity. tf-idf, or term-frequency by inter-document-frequency is a metric often used in the field of information retrieval and natural language processing as a way to represent a document. This metric scores highly words that are unique to a document and gives a low score to words that are either very common across most documents or are rare in the corpus and in the document. Thus, we can represent a document as a vector where each entry in the vector corresponds to a term in the vocabulary and contains that term's tf-idf score for the given document. We can then use these vector representations to measure similarity between documents with cosine similarity. Document vectors that are similar will receive a high cosine similarity score. Intuitively, this metric is awarding a high similarity score to documents that contain the same unique words. As we get "closer" to the target article, the expectation is that the tf-idf cosine similarity will increase.

2.3.2 Word-embeddings on Titles

For the second model, we decided to use Google's Word2Vec only on the articles' title. In particular, we compute the cosine similarity of each of the words in each title and we return a similarity that is a linear combination of the average similarity of the words, and the maximum similarity of the words. The coefficients we chose for each was .5, as we expected each of these factors to contribute equally. Nonetheless, this is a hyperparameter of the model, and could be optimized for any given use case. It is worth noting that we do not expect this model to do very well, as it completely ignores any information about the article's text.

2.3.3 Paragraph Vectors

As for the third model, we decided to create Paragraph Vectors that correspond to Wikipedia articles. In particular, we decided to implement a PV-DBOW [3] model using the Doc2Vec class in the Gensim library. This model seeks to produce a paragraph vector in a way similar to how skip-gram produces a word vector: it

samples a window from the document and tries to predict words in the window. This approach, unlike PV-DM, actually ignores the word ordering and requires less memory. We would expect that similar paragraphs should have similar predictions, and thus paragraph vectors for similar paragraphs will have to be clustered together so that they are able to make accurate predictions. This is how semantics can be extracted from the paragraph vectors.

Because the Wikipedia corpus contains more than 5 million articles, each of which is quite long, training takes disappointingly extremely long. We trained our Doc2Vec model for five epochs, ignoring any words that appeared less than 19 times in the whole corpus. Training took more than six days of CPU time.

The length of an article makes the task of training a document vector particularly hard. The theoretical motivation for a Paragraph-Vector model is to address weaknesses that bag-of-words models have in making up a vector for paragraphs/documents of variable size. Nonetheless, the length of Wikipedia articles is so long that many iterations of sampling windows and forming a classification task given the paragraph vector, and taking a gradient step. Hence, it requires a very large number of epochs to produce a very accurate model.

2.4 AI Aspect

As emphasized in class, the foundation of artificial intelligence is search. We found this to be very true in our project. We experimented with several different search methods: depth-first search, breadth-first search, iterative deepening, beam search, and priority beam search.

We also used a number of natural language processing techniques. We made use of term frequency by inter-document frequency (i.e. tf-idf), word embeddings, and sequence representations.

2.5 User Interface

For a better visual representation of our data and a more user-friendly mode to enter a source and target article, we used Flask to implement a simple HTML user interface. With this, we can run a web page on a local server allows the user to easily enter the endpoints of the path they wish to find. Once a start article and end article are entered on the home page and the submit button is pushed, these two strings are sent to the python code and used in our search algorithm to find a path. Once a path has been found, a string representation of this path is sent back to the HTML and is displayed on a new web page. This new web page is of the form

"<server address>/<start title>_to_<end title>". In fact, whenever a URL of this form is entered, the program automatically searches for a path from <start title> to <end title>. Whenever data is entered on the home page and the submit button is pressed, the server simply redirects to the URL described above, with the user inputs replacing <start title> and <end title>, which then in turn calls the search algorithm.

3 Evaluation

3.1 Proposed Questions

Can we create an automated system to play the Wikipedia game? Can we make the system more intelligent than a random agent? Can we make the system more intelligent than an informed human being?

3.2 Methods

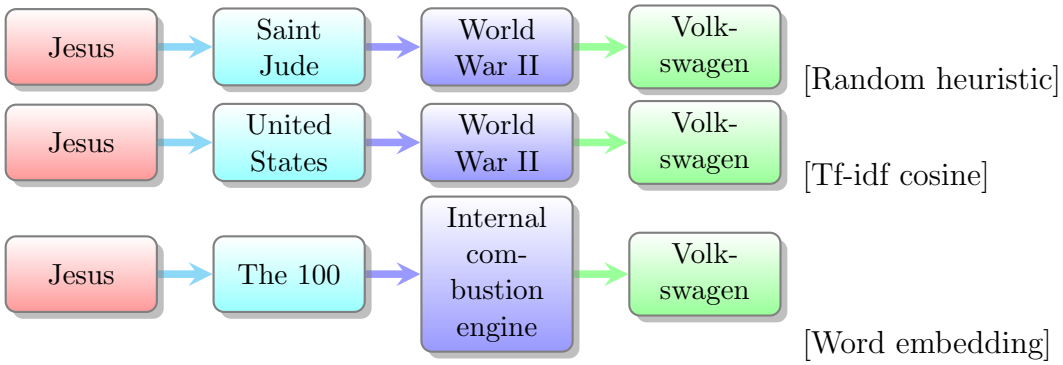
To test whether the system is able to play the Wikipedia game, it is sufficient to ensure that the agent is able to find a path from several source nodes to several end nodes in our data set, if such a path exists. To create a baseline naive model, we let the agent play out a game following random links. Then, to check whether a knowledge-based heuristic will improve the agent's success, we check how many links the system must check in order to find the target article. If our efforts are successful, the final system will visit fewer links than the naive approach.

3.3 Results

3.3.1 Qualitative Evaluation

Here are some sample source and target articles that we used to test the model. We obtain our sample source and target articles from a subset of the paths found on Wikipedia: Six degrees of Wikipedia. Since we see that the system was able to find a path for each of these pairs, we can confirm that the system is able to play the Wikipedia game. We select three representative examples and discuss why we see the given results.

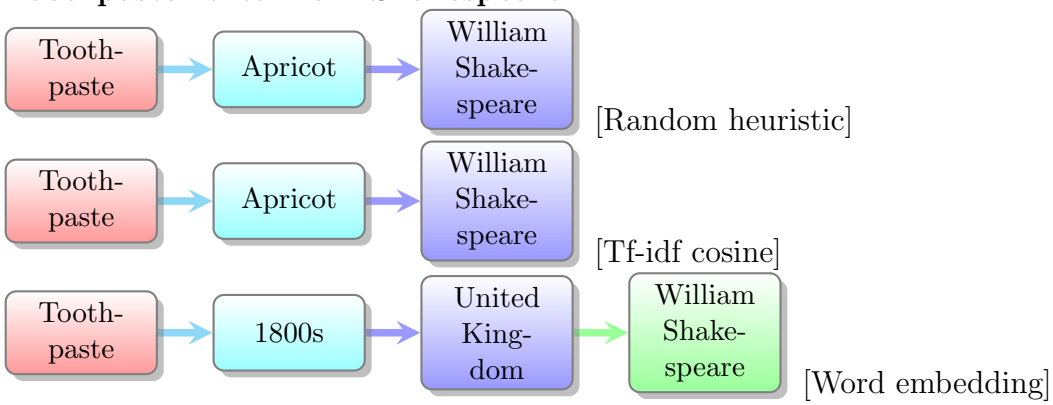
Jesus → Volkswagen



We see that all three heuristics were able to find the shortest path for this pair of source and target articles. We see that tf-idf cosine similarity identified United States as similar to Volkswagen. This follows from both articles mentioning countries frequently. The World War II is ranked highly because both that and the target article mention Germany frequently.

The word embedding heuristic resulted in a slightly different path. It is not immediately clear why The 100 (a sci-fi T.V. series) was selected initially. It is, however, very obvious that Internal combustion engine and Volkswagen would have similar word embedding, as they are both related to cars.

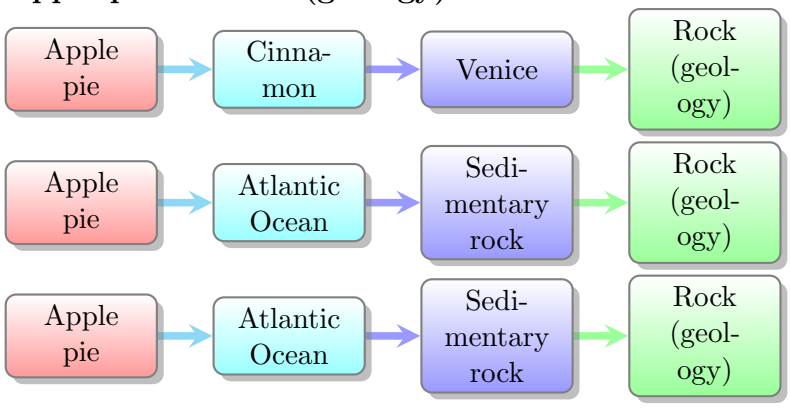
Toothpaste → William Shakespeare



In this case, the word embedding similarity metric was unable to find the shortest path. This is not entirely surprising, as we do limit the width for the beam search. The paths taken by both metrics do make a lot of sense. The discussion of the history of apricots mirrors the language used to describe Shakespeare and his works.

It is not puzzling that dates and people might have similar embeddings, as well as a person and the location that he is from, so deem that the word embedding method performed well on this example.

Apple pie → Rock (geology)



This is an interesting case where tf-idf cosine similarity and word embedding similarity returned the same results. These article mainly discuss things that are specifically related to the title with few extraneous unexpected topics. This the similarity of the title corresponds to the similarity of the text, since the text corresponds so closely with the title.

3.3.2 Quantitative Evaluation

Now, we evaluate our different heuristic functions and compare the length of the path from source to target article and the number of links visited in the search process.

The graph in Figure 1 plots the average number of steps taken to reach the target article from the source article for each of the different similarity metrics.

We see that all three heuristics perform approximately equally well in terms of average length of path. This makes sense, because we use a breadth-first search which should find the optimal path. The slight differences can be accounted for by the fact that we limit the width of the beam search to 10 for tf-idf and word embeddings. This large improvement in number of links visited (discussed later) is a trade-off that we knowingly made. Increasing the width of the beam size would counter-act this negative effect.

The bar graph in Figure 2 displays the average number of links visited by the priority beam search algorithm with different heuristics. The width is currently set to infinity for random so as not to potentially miss a relevant article that was randomly chosen as lower ranking. It is set to 10 for the other two heuristics. The number of links visited is inversely proportional to how quickly the algorithm finds the target article.

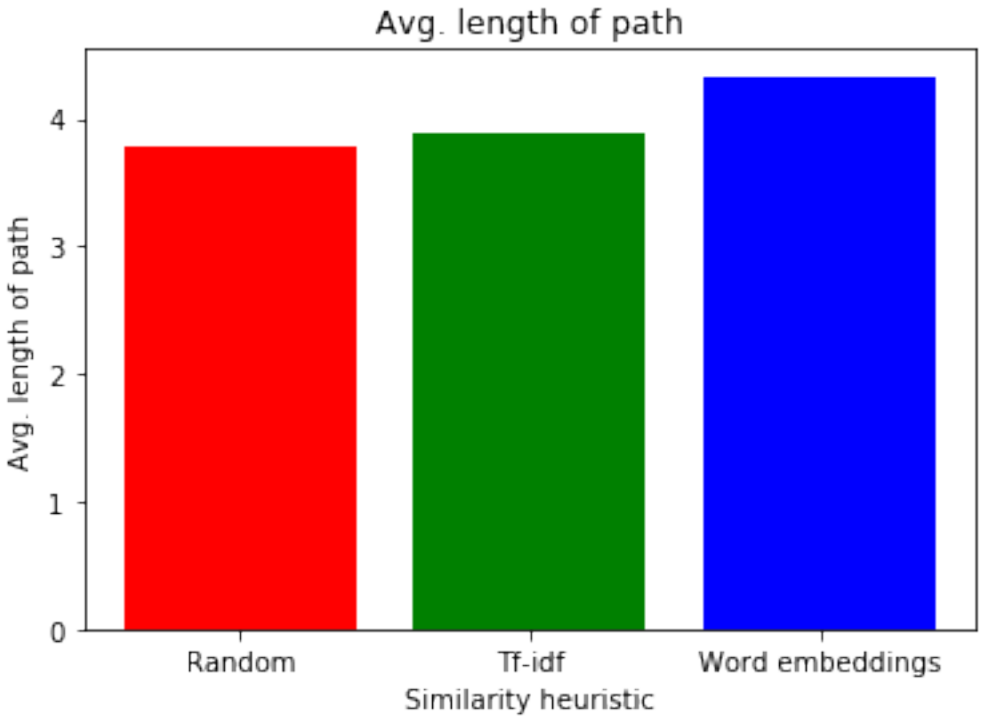


Figure 1: Average number of steps

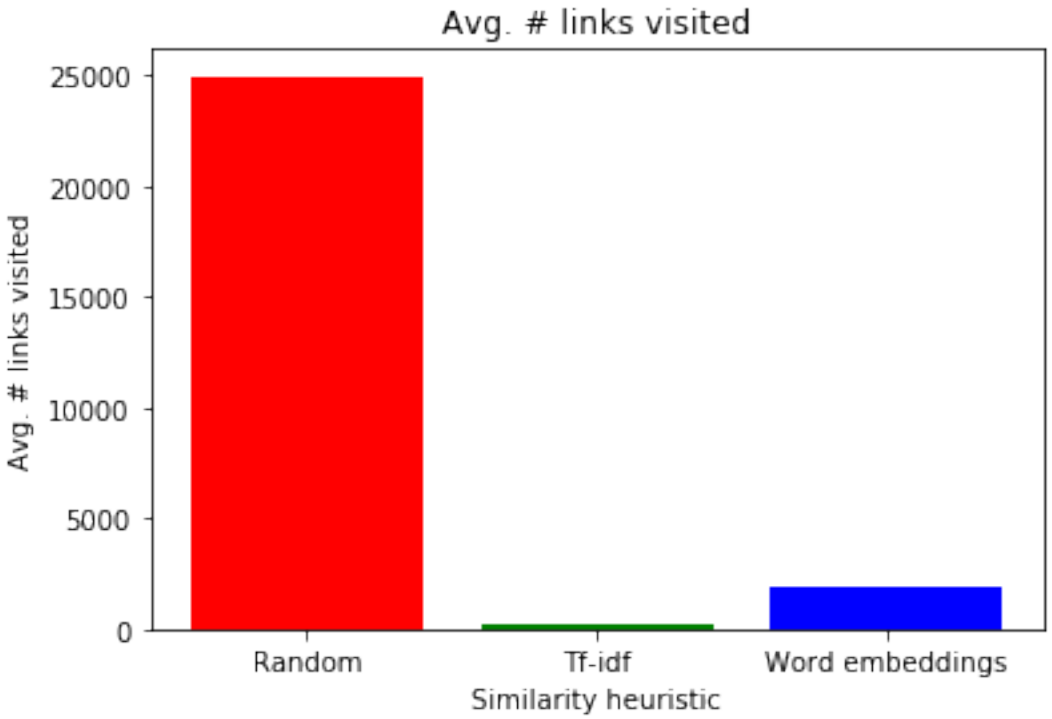


Figure 2: Average number of links visited

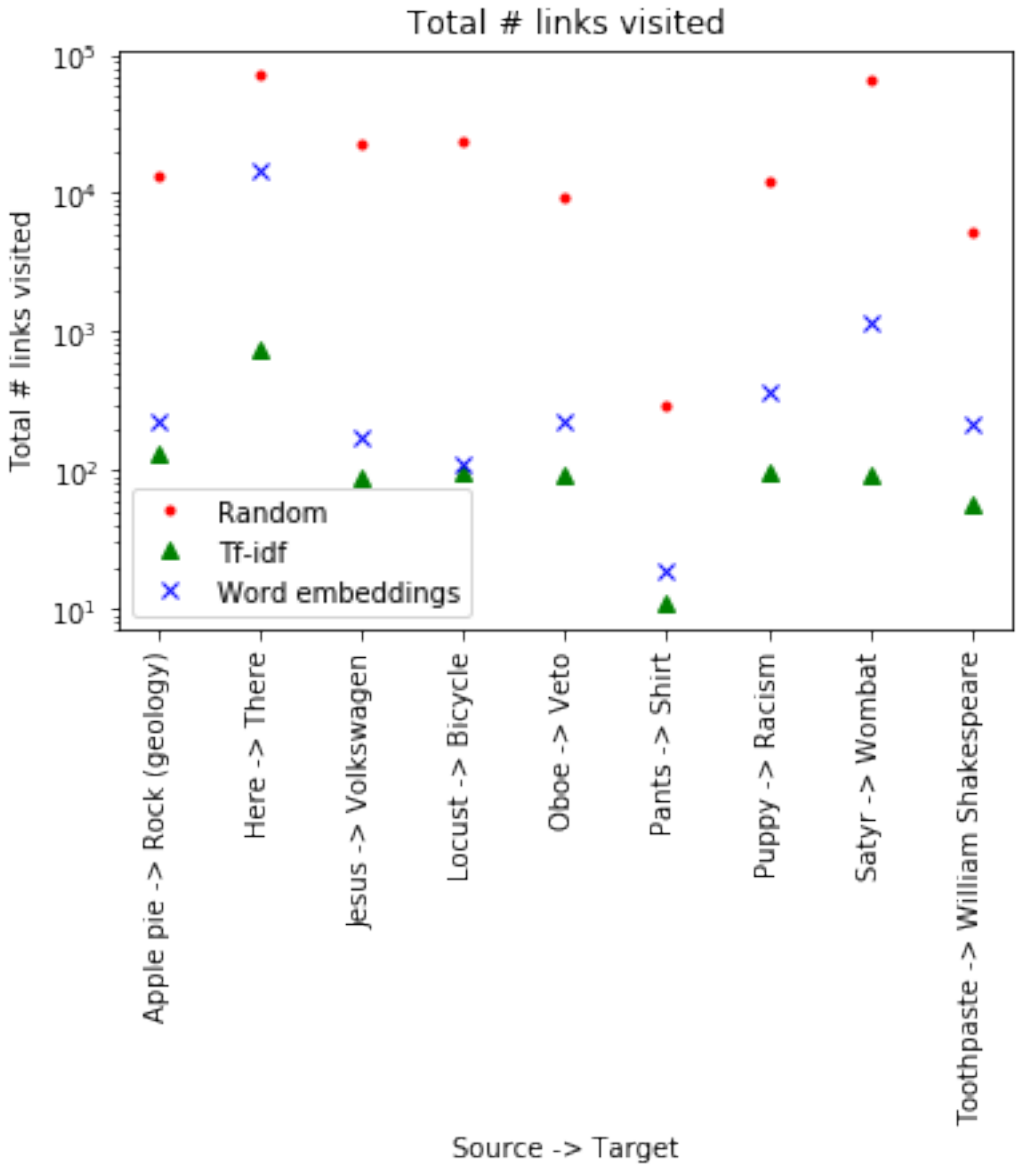


Figure 3: Total number of links visited

In Figure 3 we see a scatter plot of total number of links visited for each source/target pair in our test data set.

On its own, these numbers are not very meaningful. However, we see a clear trend showing that the similarity metrics we chose to implement improve the agent's ability to reach a goal state much more quickly. Note that we are viewing a log scale, so our metrics very significantly outperform the random baseline. We also see that for article pairs that are intuitively closely related (i.e. pants and shirt), the total number of links visited is visibly lower than that of the other pairs. We also see that though the two metrics we created do well, one does slightly better than the other. Tf-idf cosine similarity is a slightly better measure of similarity than word embeddings. That in itself may be surprising, but recall that we are using tf-idf on the entire corpus including the text of the article, while word embeddings are just used on the titles of the articles.

References

[1] "Wikirace," Feb 2019. [Online]. Available: https://en.wikipedia.org/wiki/Wikipedia:Wikirace

[2] "Wikiracing," Jan 2019. [Online]. Available: https://en.wikipedia.org/wiki/Wikiracing

[3] Q. Le and T. Mikolov, "Distributed Representations of Sentences and Documents," Google Inc., Tech. Rep., 2014. [Online]. Available: https://cs.stanford.edu/~quocle/paragraph_vector.pdf