Summary:

The main goal of this project was to build an AI that can get from one Wikipedia page to another, given random starting and ending pages. This task is informally known as the "Wikipedia Game", where it is generally played with two people racing to get from a randomly selected initial Wiki page to a target page, by only using hyperlinks contained within the website. Ever since I saw a visualization of word embedding from word2vec, I've found semantic analysis to be one of the more interesting topics within NLP, and figured that a Wikipedia game AI would be a great application of this, along with my general interest in AI. This project challenged me to incorporate both pre-trained GloVe word embeddings through Gensim's API, along with the Wikipedia Python API. It also tested my understanding of AI search algorithms, as I coded everything from scratch. While there is no GUI or website for running the AI, any machine with Jupiter Notebooks installed can run the project to explore a link between any two Wiki pages. The code for the project, along with instructions contained in the README can be found on the repository's page on GitHub. The slides for my final presentation are also contained in the repo, which offer a concise exploration of the project along with some visualizations.

Methods:

I solved this problem by using a search algorithm, where the search space is defined by all possible Wikipedia pages. I used the Wikipedia Python API. In the search algorithm, for a given initial Wiki page, every link to another page represents the "frontier" of states that the algorithm can move to. To decide which of these links to take, I applied a heuristic function that uses the cosine similarity of the word embeddings of words in each of the potential next pages with the destination page. These word embeddings were downloaded from a pre-trained GloVe model that was trained on an English Wikipedia dump and English Gigaword 5th Edition dataset, which is an archive of newswire text data that has been collected by the Linguistic Data Consortium (LDC). It includes sources such as the Associated Press and the New York Times.

My algorithm is classified as a greedy search algorithm because it always selects the page with the highest cosine similarity to the target. Like all greedy algorithms, this can sometimes cause problems when a local optimum is not the global optimum. It was also a perfect candidate for recursion because after each page is selected, the new state space can be defined just like the

beginning of an entirely new problem with the current page as the new initial page. All that was necessary was to make sure the recursive calls returned a list of previously visited pages to make sure that it did not loop in circles between pages. When running the program, you can see the heuristic score beside the next page that it is choosing. When working correctly, this score will approach 1 until the program finds the target page.

One modification I tried to combat the problem of the greedy algorithm finding local optimums was to implement backtracking such that the algorithm would remember the best page that it has visited so far. If the algorithm dives down a specific branch of the search tree and is not finding any better pages, it will backtrack to the best recent page it has seen. Although this did not boost the performance of the algorithm, which I measured in the average number of degrees it took to find a solution, this backtracking modification did help prevent the algorithm from getting lost in very specific corners of Wikipedia that had little to do with the target page.

Another modification I tried was adding a score to the overall heuristic score that indicated how useful a potential next page was. By looking ahead at a potential page and counting the number of outgoing links from that page, we have a rough estimate of how general that page is, and how useful it is for extending to new concepts. I only used this extra score when the normal cosine similarity heuristic score was belong a threshold of .5. This models the way that many humans play the Wiki game: starting off more general before finding a slightly similar abstract concept and then getting more specific until the goal is reached. This modification actually did not benefit the overall performance of the model in any significant way. It is also important to note testing this model was very difficult. In the last cell of the main code, you'll see that I initially attempted to create two lists of pages, one for the initial pages and one for the targets, and then loop through those pages while calling the functions on them. When errors would occur, however, all subsequent function calls would create errors. I spent a few hours debugging this and trying lots of try-and except-statements to no avail, so I individually called the functions in single cells.

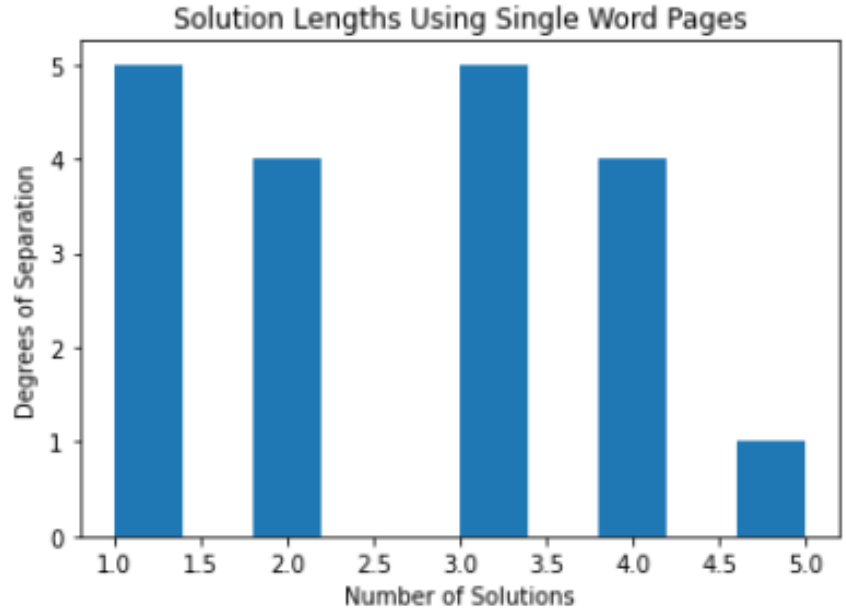
Experiments:

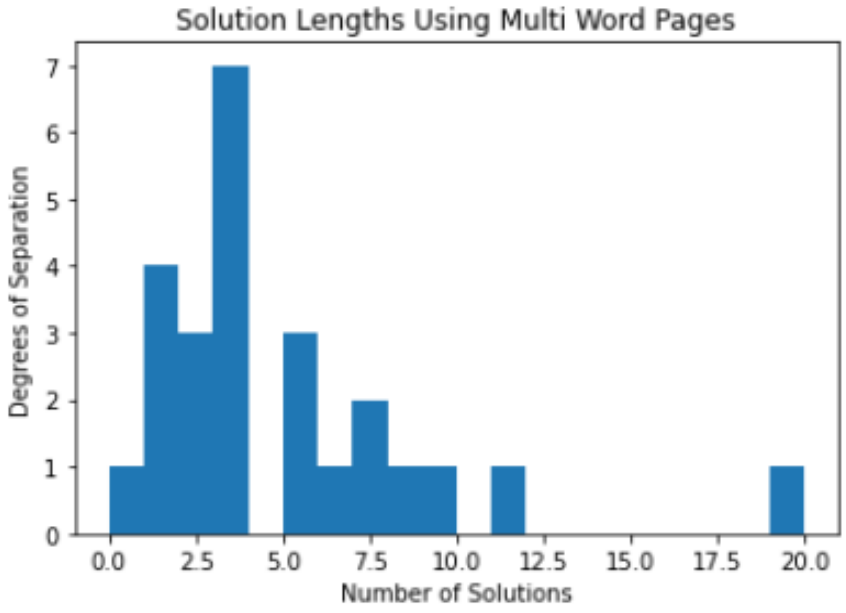
The final and most important modification I made was allowing the algorithm to use Wiki pages with multi-word titles. Not only did this allow me to have initial and target pages with multiple words, but more importantly, it expanded the state space to include Wiki pages with multiple words. This increases the number of potential paths the algorithm can take. I did

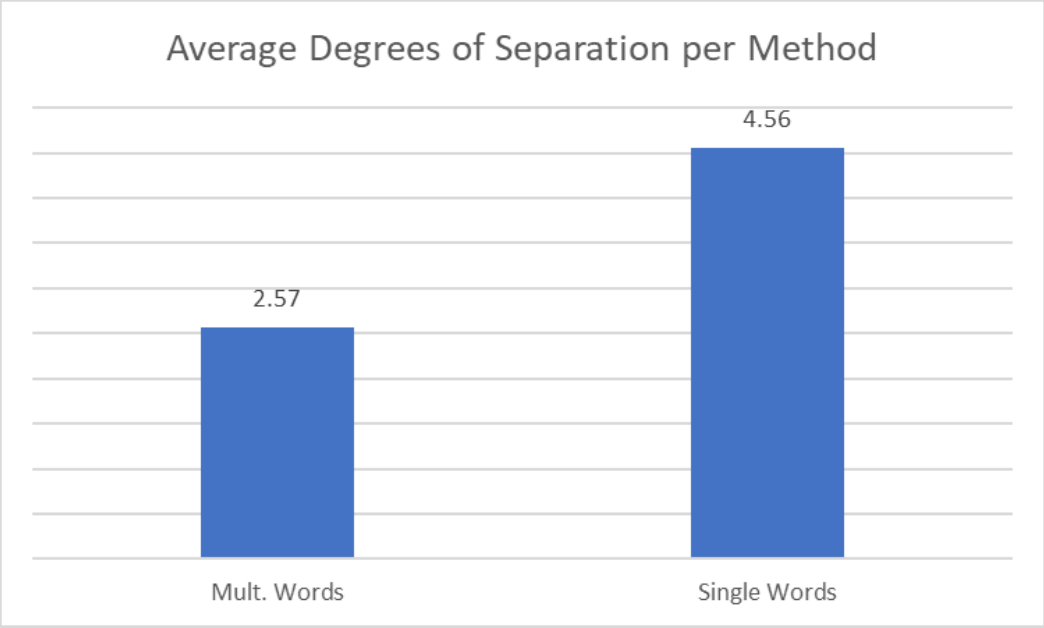
this by either averaging or summing each word vector in a page. Averaging and summing were returning very similar results, so I used summing because it is slightly more computationally efficient. Lastly, I was actually able to test how this modification quantitatively improved the performance of the algorithm. I did this by finding pairs of single-word Wiki pages so that they each pair could be tested on the single-word and multi-word versions of the algorithm. I got these pairs from the Wiki Game, which generates pairs of relatively simple words. Admittingly, this aided in bypassing the problem of proper nouns, which word embeddings are inherently challenged because of a lack of training examples. Additionally, the sum of individual words in a proper noun often does not represent the entire noun, such as the case with the location "Bay Furnace, Michigan".

Results:

When comparing the single-word and multi-word models on the same 25 pairs of pages, the single-word model had an average solution length of 4.56 compared to the multi-word model's average of 2.57. While this is a large improvement, there were also many drawbacks to the multi-word model. The largest drawback was the fact that out of the 25 tests that were run, the multi-word model resulted in an error for 6 of these. It was also significantly slower because it has many more states to search through, and also computes the extra score for generalization that is indicated by a potential page's number of outgoing links. Still, I am relatively satisfied with these results, especially because the initial baseline – which just used the single-word model – was averaging around 6 degrees of separation per pair.


Solution Lengths Using Single Word Pages
Degrees of Separation
0
1
2
3
4
5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0
Number of Solutions


Solution Lengths Using Multi Word Pages
Degrees of Separation
0
1
2
3
4
5
6
7
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Number of Solutions



Conclusions:

Overall, I was actually impressed by the efficacy of word embeddings on this task. Especially because Wikipedia is known for having a large percentage of its pages representing proper nouns, I figured that the model would do worse. I also thought that words having multiple senses, such as “mind” would give the algorithm trouble. This certainly reduced the number of potential paths between pages, but since each page is fixed to one definition of a word, each path usually has a consistent theme that does not alternate between senses of a word. I also recognized that words with multiple senses are more likely to be verbs, which are not used as frequently by this program.

I would have liked to be able to test more modifications, but I spent a really long time getting the algorithms to not give errors. There are so many strange types of Wiki pages, like lists, portals, talks, files, etc. that my program would get lost in. On top of this, the algorithm would sometimes choose an optimal link that was not actually a Wiki page, resulting in errors. These errors would often repeat over and over and then stop by just running the code again.

Future work could include improvements to the current algorithm, such as a bidirectional search that starts from both the initial and target links, or picks one to start from based on another heuristic. I also would like to compare different types of word embeddings, such as those

generated by Word2vec to see how they compare. It would be especially interesting to investigate how the window size would affect the performance. A smaller window would give more general words, which is probably more useful for the Wiki game, although a larger window could be helpful for knowing very specific relationships when getting close to the target.

Works Cited

English Gigaword Fifth Edition: https://catalog.ldc.upenn.edu/LDC2011T07

Pre-trained GloVe Model from Stanford: https://nlp.stanford.edu/projects/glove/

Resources on using the model:
https://kavita-ganesan.com/easily-access-pre-trained-word-embeddings-with-gensim/#.YnLps9rMIuR

The Wiki Game (used to collect pages for testing): https://www.thewikigame.com/group

Wikipedia API for Python: https://wikipedia-api.readthedocs.io/en/latest/wikipediaapi/api.html