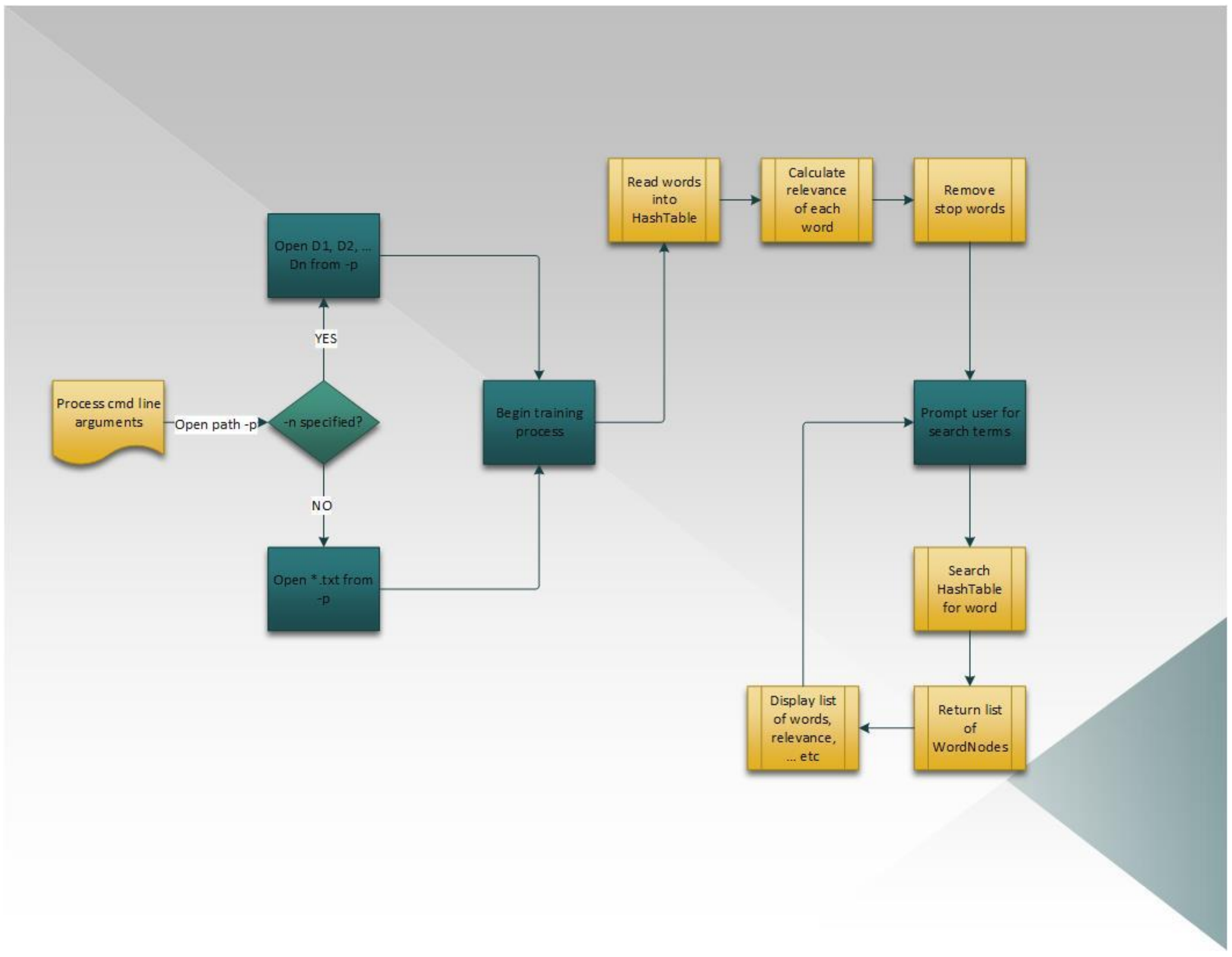
Tad Miller

CSCI 2461

Dr. Narahari

Project 5: Search Engine

Program Flow



Process cmd line arguments
Open path -p
-n specified?
YES
NO
Open D1, D2, ... Dn from -p
Open *.txt from -p
Begin training process
Read words into HashTable
Calculate relevance of each word
Remove stop words
Prompt user for search terms
Search HashTable for word
Return list of WordNodes
Display list of words, relevance, ... etc

Note: Flow chart can also be opened in separate PDF for a higher detail view

Question Responses

Q(A): If we have N documents, a document D_i having m_i words, and a query of K words. Then if every word in each document is in the singly-linked list, the retrieval runtime would be approximately O(KD), where D is the sum of D_1, D_2, …, D_N. This is because the algorithm must traverse through the entire list of size D (since we have D words), K times for each term we have.

Q(B): Now, assuming our HashTable has D words spread *evenly* across B buckets, finding K words would take O(K*D/B).

Q(C): Stop words are dictionary items that have little relevance in a search term due to their high frequency in a sentence. These are words like “and”, “the”, “here”, … , and they don’t add much value to a sentence other than proper structure. Removing them from search terms & the search database improves the speed at which we can send and return results, since we won’t ever have to iterate through them.

Q(D): The second approach is certainly more effective to deal with stop words. This is because when we get to the point at which we wish to remove them, it will take much less time to track them down in a 2D Linked List where we are keeping track of our words in one Linked List, and the points at which each word occurs in another Linked List, which is encapsulated by a word. In my implementation, I use a “wordnode” to store the value of the word, and an “occurencenode” to hold the documents in which each word appears. Upon removal, because we are using *malloc()*, it is infinitely more efficient to call *free()* on one Linked List for one word’s occurrences than it is to traverse through a huge Linked List and find every occurrence of such word.

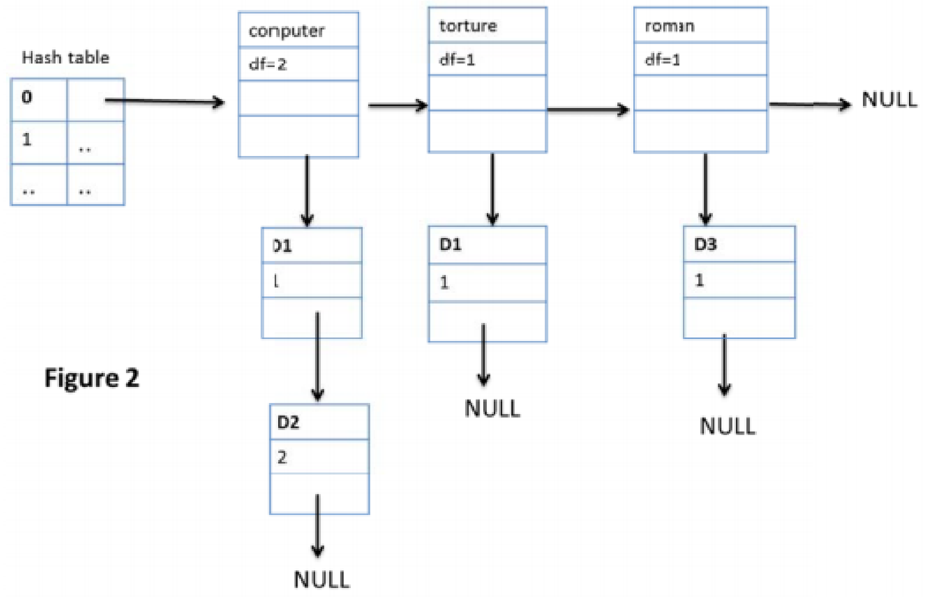


Figure 2

Extra Credit

Option 1: I didn't quite have time to implement this, however the function definitions to do so are in my program. This is quite simple to do though: For a HashTable of size n, i.e. n buckets, we could set our threshold to just be n. If any single bucket grows greater than n in size, then we would find the next prime number from n: n', and set the amount of buckets equal to this. Once we've done so, we can simply rehash the values in the bucket with the balance size. We must, however, keep track of the difference between n and n' so that we don't have to rehash any of the other buckets. This way when we insert a new value into our HashTable, since our hash code is based on our size, now n', we'll have to subtract accordingly such that we can still find our original values in our other buckets.

Option 2: I implemented this using DIR. It simply reads from a directory all the files of a type .txt and will complete the normal training process on them.