Ternary Directed Acyclic Word Graph Dictionary for Spelling Correction and String Auto-Completion

Terrence Gausi **Christian Lopez-Farnes** **Hangbo Gu**

{terrence.gausi, chris.farnes, hangbo.gu}@sjsu.edu

A. Introduction

In the article titled "Ternary Search Trees," Robert Sedgewick and Jon Bentley describe a data structure that implants the idea of a Binary Search Tree (BST) at every level of the data structure--each node of a Ternary Search Tree (TST) has three links: the middle link points to the node for the next character in a word, the left and right links point to the nodes that have characters smaller or larger than the current node [1]. Years later, Miyamoto et al optimized the TST for pattern matching, engendering the the Ternary Directed Acyclic Word Graph (TDAWG)--each node of the TDAWG maintains the three links model of the TST but with the proviso that when identical subtrees are found in the TST, all links to these equal subtrees must be replaced with a link that leads to one of the subtrees [2].

For our project, we implemented a spelling corrector that stores an English dictionary as a TDAWG. The spell corrector is fashioned on the Peter Norvig spell checker [3], which is implemented in the TST-Edits algorithm.

I. What is the problem you are solving?

Put simply, the TDAWG solves the problem of affix stripping when dictionaries are implemented as hash tables. Given an English dictionary, the TDAWG enables a user to verify the correctness of a word against a dictionary of their choosing. In our implementation, the team reads in each instance of a word from a downloaded online dictionary and stores every word as a set of nodes, where every node contains a character; in order to form a complete word, the middle pointer is linked with another node until a *isAWord* boolean flag is raised. It is evident from this pattern that words that have a similar prefix (i.e. words that begin with the same characters) will share nodes until a suffix character sets them apart. This behavior can be observed in *Figure 1*.

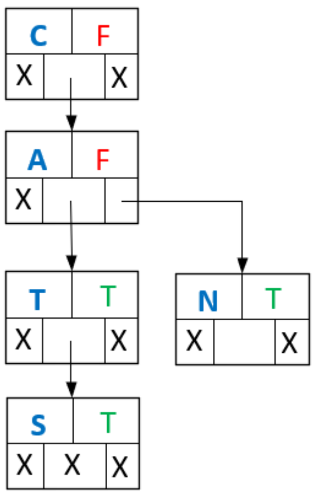


Figure 1: *Cat and Cats may share nodes because they have similar a similar Prefix. Note that the third and fourth nodes both have isAWord boolean value set to true. The word Can does not share a common suffix so a new node is created.*

It is evident that the shared prefix problem has been solved. However, there exist words that share a common suffix, (i.e. words that end with the same characters). We define these repetitions as cycles that consume unnecessary memory. Nodes that represent a common suffix can be reused as shown in *Figure 2*. In order to re-use nodes the team implements a hash table that will hold the hashed value of each node; if the value of a node has been previously hashed, it follows that we can recycle the last node with the same hash value and release the current one from memory while a temporary pointer keeps track of the next node to be hashed.

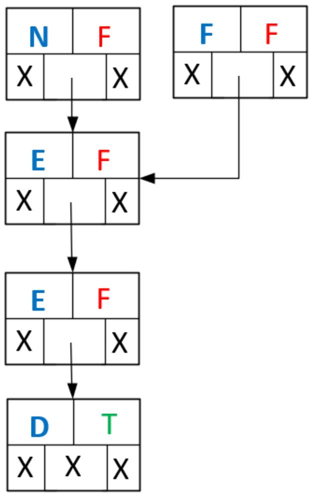


Figure 2: *The words Need and Feed may share nodes because they end with the same Suffix i.e. "eed".*

A different dictionary implementation can be achieved by constructing a hash table where only the roots of the words are stored (i.e. devoid of prefix and suffix) this data structure provides a faster search time than the TDAWG; however, the hash table implementation is highly specialized in a single language only, that is to say that if a user wished to use the hash table implementation in a different language, it would be necessary to re-define the roots of every word. On the other hand the TDAWG is able to function with several different types of strings with little regard of what language or the content of the strings that are used as an input. In addition, one of the main concerns taken into account when constructing the TDAWG is the space used to store the final dictionary. TDAWG's counterpart maintains

† https://github.com/hangbogu/CMPE130

a hash table capable of accommodating large (greater than 100,00 words) dictionaries with minimal collisions can be rather large and space inefficient. It can be stated then that the TDAWG is flexible in terms of what input it is able to handle and it is also a highly nimble data structure because one of the main reasons for it's construction is data compression.

II. Why is it interesting?

The TDAWG is an interesting implementation of a word dictionary because it is an ambiguous data structure which exploits the most convenient features of several data structures. The first data structure used is a Ternary Search Tree (TST) [1]. This structure is chosen because of its fast search time and flexibility at time of insertion. When our dictionary is created there is a need to balance the TST because the input dictionary is alphabetized, rendering our tree right side heavy. To balance, the TST is treated as a BST: meaning that the tree is balanced from the lowest level from left to right, grouping three nodes at a time, a process that can be appreciated in *Figure 3*. After the tree is balanced we must free unnecessary memory by realizing that words that end with the same suffixes can be categorized as cycles. A temporary hash table is created to delete repetitions and arrive at the finished product, a TDAWG. This process is shown in *Figure4*.

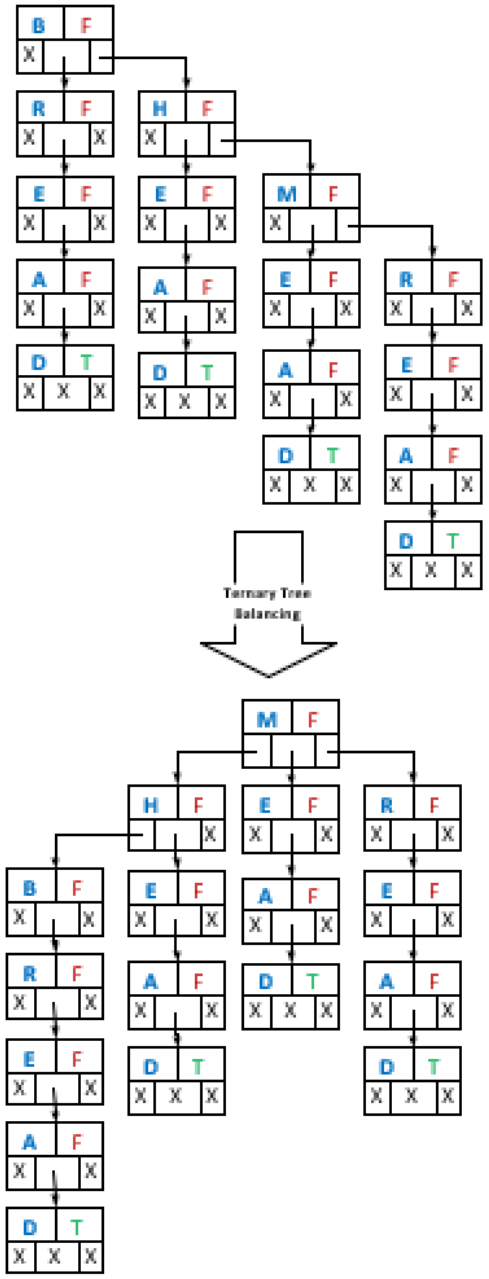


Figure 3: *Depiction of word graph right-side heaviness problem and solution found by shifting and rotating.*

† https://github.com/hangboqu/CMPE130

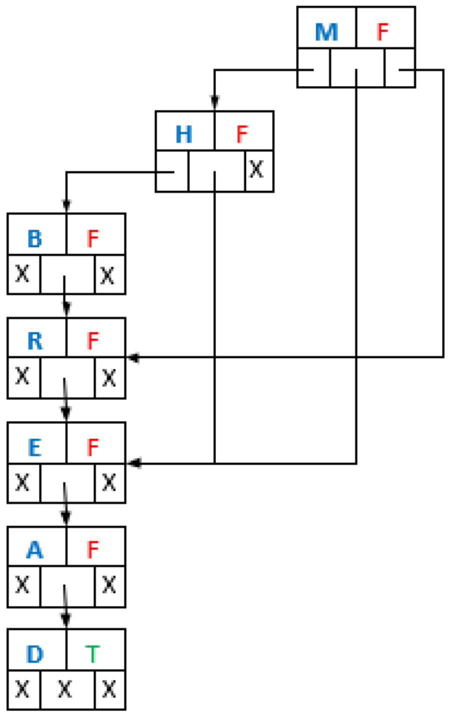


Figure 4: *Visualization of node compression*

III. Who would use it when done? And who would use it when done?

The TDAWG can be applied to nearly any application that requires a correct string input, it can be implemented in text editors, online search queries or as a string suggestion tool for key search in large data sets. As presented, the TDAWG is used as a word suggestion tool; however, it can be adapted to suggest different elements such as IP addresses, street addresses, or to suggest any data type that can be easily reversed to character values.

IV. Why is it challenging?

The project proved to be challenging because the TDAWG amalgamates several data structures to create an ambiguous structure highly optimized for search of strings. The biggest challenge stems from the right-side heaviness of the original TST.

This right-side heaviness proved difficult to solve because the tree must be balanced to take advantage of the logarithmic search time of a TST, which is $O(log_3\ n)$. The team overcame this adversity by including a *count* field to each *node* object. The count field facilitates the shifting and rotation because much like a BST, the TST can be balanced by keeping track of each node's descendants and ancestors; at each rotation or shifting function call we focus on three nodes at a time. The rotation function is recursive and nodes cannot be specified, but the algorithm can take into account the count field of each node and use it for balancing.

IV. What algorithms did you use? And why?

The principal algorithms utilized are Divide-And-Conquer and Dynamic Programming. The former was utilized to build the TST out of the downloaded word list, balance the TST, compress the TST, and then search the compressed TST or TDAWG.

† https://github.com/hangbogu/CMPE130

The latter was utilized to implement the spell corrector within an edit distance of 2.

Consider, for instance, the task of building a TST of depth *d*: following the instantiation of the root node, the left, right, and middle subtrees of the root must also be instantiated. To this end, we deduced that the task can be divided into three subtasks, in which each subtree can be instantiated recursively by applying the whole subtrees instantiation process to either the left, right, and middle subtrees of the root.

Now consider the task of correcting a misspelled word: this amounts to converting a string of length *m* into another of length *n*. Four edits to the original string are required, namely, deletions, transpositions, alterations, and insertions. The results of each edit must be stored. When none of the stored strings is found in the TDAWG, a second round of the four edits on these strings is then required. When an edited string is found in the TDAWG, it becomes a candidate for correction suggestion. To this end, we deduced that the task at hand can be solved by dynamic programming.

V. What are the shortcomings of the existing work?

The existing work is optimized for word search, word correction, and word suggestion; however, the TDAWG is not able to insert into or delete elements from the dictionary. This limitation stems from the fact that when the trie is created the tree is balanced and compressed as one; this means that if a new word is inserted or deleted the tree must be rebalanced and recompressed each time. The computational time to balance, and compress the Trie is not optimized and therefore not implemented in this solution. The TDAWG is a specialized data structure that is used only for word search, correction and suggestion with the limitation that the entry dictionary must be sufficiently large for accurateness.

B. Related Work

The most ubiquitous implementation of spelling correctors utilize hashtables. That is, a dictionary is implemented as a hash table and, if a word is found in the dictionary, the word is considered correct; otherwise, the word is incorrect. When a word is deemed incorrect, it is edited by means of deleting, inserting, transposing, or altering characters in the string, and then looking up each edited word in the dictionary [3]. Some hash table dictionary implementations support affixes and capital letters, often utilizing complicated schemes such as stripping affixes and storing only root words, as well as

† *https://github.com/hangbogu/CMPE130*

utilizing pre-compiled data about words with similar sounds, a prominent example being GNU's Aspell [4] spell spelling corrector. Aspell's corrector is robust with a constant time lookup, but it requires large amounts of memory to store the stripped affixes, as well as the precompiled data.

Our implementation uses a TST to represent a dictionary of english words. The problem of affix stripping is at once eliminated: common prefixes are stored just once, and when the TST is compressed to form a TDAWG, common suffixes are also stored just once. The only memory required is to store the TDAWG dictionary itself. Our implementation is also robust but with a logarithmic lookup time.

C. Method Description

I. Provide the data structures used. Why did you choose these data structures?

The data structures used are an unbalanced TST, balanced TST, a TDAWG, an a hash table.

The TST, was chosen because of our usage of individual character nodes and affix based approach to the spellchecker. The nodes objects are comprised of an unsigned integer "Hash", an unsigned integer "Count", a boolean "IsAWord", 3 node child pointers (representing left, middle and right), and the character that the node represents. By treating the ternary trie similarly to a BST, we are able to reap the benefits of a BST, while cutting down on the total number of nodes that the dictionary portion of the spellchecker is occupying.

Unsigned Integer: Hash	**Unsigned Integer: Count**	
Char: Symbol	**Boolean: IsAWord**	
DagNodePtr: LeftChild	**DagNodePtr: MiddleChild**	**DagNodePtr: RightChild**

Figure 5: *Node object visualization*

The BST portion of this algorithm is accomplished by treating the left and right children of the TST similarly to the left and right children of a BST, representing the cases where the character from the query is lesser or greater than the parent node respectively. If we find the node that we are looking for, then the search is done for that character. However, due to the nature of the search function that we are performing, finding one character doesn't necessarily mean the completion of our search. This is where we can take advantage of the power of the TST. The middle child represents the next character in the query if we have already found the

† https://github.com/hangbogu/CMPE130

previous character. To fully complete our search, we can advance to the middle child of our previous node, and repeat the above process until one of several completion conditions.

The space reduction portion of the algorithm is inherent to the TST. When inserting words into the dictionary, words that have the same prefix can share the same nodes until the prefix expires. For example, if you were to insert “cat” and “cats” into the dictionary, both words share the prefix of “cat”. Because of this similarity, instead of inserting 7 nodes for the two words, we can insert just 4 nodes: “c”, “a” and “t” for both “cat” and “cats”, and “s” to complete “cats”. The problem of differentiating between the two words is trivial, and solved by including a flag that tells if a node is the end of a word. In our case this is the member *IsAWord*.

II. Why did you choose these data structures to solve your problem?

We then chose to balance the TST to further optimize searching. The initial alphabetical insertion into the TST creates a skewed tree that will result in an $O(n)$ search for each character due to violating the balance properties of a BST. Balancing the TST to create a pseudo BST allows us to achieve a $O(log_3(n))$ search per word, which is, naturally, faster than $O(n)$. This is accomplished by using the node member variable “Count” to keep track of the levels in the ternary trie, while recursively balancing the left and right subtrees of each word root using left and right rotations.

Following the balancing of the ternary trie, we then chose to optimize the structure even further. This was accomplished by converting the balanced TST into a TDAWG. As the TST is optimized for prefix space reduction, the TDAWG serves to reduce amount of nodes required from the suffix condition. This is accomplished by removing all copies of shared suffixes but the leftmost copy. The node copies that are removed from the dictionary are determined by using a hash table, and a hashing algorithm. Nodes from the balanced TST are hashed and inserted into the hashtable. If the hash table already contains a copy of the node being inserted, the parent of the redundant node will point to the stored copy. (One stipulation is that the root node of a word cannot be removed from the balanced TST. This is because this would remove the only remaining unique portion of the stored words, the prefix.) The words that previously needed those suffixes are instead pointed to the one stored copy. For example, in a TDAWG consisting of the words “following” and “typing”, “typing”s’ copy of the “ing” suffix has been removed. Instead,

† https://github.com/hangboqu/CMPE130

the “p” node in the “typing” subtree is now pointing at the “i” node in the “following” subtree. This reduces the amount of nodes required by 3 in this case. However, a more significant difference can be seen with a completed dictionary. For example, a dictionary that required ~1.4 million nodes in the TST can be compressed to ~360 thousand nodes using the TDAWG. That’s about a 75% reduction in the number of nodes required.

The spelling correction algorithm gives suggestions for queries with up to an edit distance of 2 from words in the dictionary. It is dependent on edit distances and probability. The edit distance of a query would be the number of operations that would be need to be made to the word to changed it into another specific word. The operations are replacement, transposition, insertion and deletion. Replacement of a character involves replacing that character with any other viable character. Transposition involves swapping two characters in the query. Insertion is adding a character into any position in the query. Deletion is removing any character from the query. For example, “caata”and “cats” have an edit distance of 2 because deleting one “a” from “caata” and replacing the last “a” with an “s” results in “cats”. The probability comes in when considering what suggestions to display. Only the words with the largest edit distances will be displayed. That is, if there many words that have an edit distance of 1 from the query but just 1 word has an edit distance of 2, the spelling corrector will only suggest the latter word.

The prefix search function takes a prefix and displays a selection of words with that prefix. Due to the prefix based nature of the data structure, this is a trivial task. Using the search properties of a the TDAWG, the function could just display the entire subtree under the last character in the prefix. For the sake of not overwhelming the user, however, only a few of them are provided.

The input of the dictionary download algorithm is a list of words provided by the internet. The preferred format of these words is all lowercase, with characters ranging from ‘a’ to ‘z’. Although other characters are acceptable for most of the dictionary, the hashing function that we are using has a chance of producing segmentation faults with these values. For this reason, we are sticking to word lists with the above requirements. The output of the dictionary download algorithm is a text file containing all the words given by the online word list.
The input of the TST is the text file produced by the dictionary download algorithm. The

† https://github.com/hangbogu/CMPE130

output of the TST creation algorithm is an unbalanced TST.

The input of the balancing algorithm is an unbalanced TST. The output is a balanced TST.

The input of the TDAWG creating algorithm is a balanced TST. The output is a TDAWG.

The input for both the spelling correction and prefix search algorithms is a user-inputted string, and the TDAWG created earlier. The output for spelling correction in the case of a misspelled word will be a list of possible corrections. In the case of a word that is spelled correctly, it will output that the word was found in the dictionary. The output for the prefix search will list a limited amount of words with the input prefix. If the prefix is not found, it will output that the prefix is not found. The source code as well as all the support files used in this project can be found in the team's Github page†.

D. Analysis of Algorithm

There are several important algorithms that are used in the creation of our spelling corrector. These are, in order of utilization, downloading of the wordlist, creating the TST from the downloaded word list, balancing the TST, compressing the TST to form a TDAWG, and a spelling correction.

With a specified url, The **TST-Download** algorithm, as the name implies, is used to download an online English word list and stores the downloaded word list in a .txt file in the home directory. We chose this algorithm in order to have a web-based component to our implementation, considering that we collectively lack any experience in HTML and CSS. The TST-Download algorithm has an $O(n)$ runtime and space complexity--each line of the web page must be read and written to a .txt file.

Dictionary Download:
TST – DOWNLOAD

```
TST - Download(const char* url) :
  FILE* InputFileObj
  CURL* curlHandle
  CURLcode result
  outputFilename[FILENAME_MAX] <--url
  curlHandle <--curl_easy_init()
  if curlHandle is not NULL then :
InputFileObj <--fopen(outputFilename,
"wb")
  curl_easy_setopt(curlHandle,
CURLOPT_URL, url)
  curl_easy_setopt(curlHandle,
CURLOPT_WRITEFUNCTION, NULL)
  curl_easy_setopt(curlHandle,
CURLOPT_WRITEDATA, InputFileObj)
  result =
curl_easy_perform(curlHandle)
  if result is not equal to CURLE_OK
then :
fprintf(stderr, "curl_easy_perform()
failed: %s\n",
curl_easy_strerror(result))
return
  else curlHandle is NULL then :
fprintf(stderr, "Curl init failed!\n")
return
curl_easy_cleanup(curlHandle)
fclose(InputFileObj)
END TST - Download
```

Once the word list is downloaded, we begin to build the TST. The principal algorithms utilized are **TST-NewNode** and **TST-INSERT**: the former, with a constant runtime and space complexity, creates a node

† https://github.com/hangbogu/CMPE130

for each character in a string read from the .txt file; the latter, with an $O(\mathrm{m} \times \log_3(n))$ runtime and space complexity-- where m is the number of words in the .txt file and n is the number of nodes--inserts a word into the TST. TST-INSERT was chosen to make use of the Divide and Conquer algorithmic approach, considering that each node of the TST has three links--thus, three subtasks.

Build TST:
Initializing new node

```
TST-NewNode(char ch):
  curr <-- new TSTnode

  curr->symbol = ch
  curr->isAWord = false
  curr->leftChild = NULL
  curr->middleChild= NULL
  curr->rightChild = NULL

  return curr
END TST-NewNode
```

Build TST:
TST - INSERT

```
TST-INSERT(TSTnode* root, char *word) :
if root is NULL then :
  root <--TST - NewNode(*word)

if *word is less than root->ch then :
  root->leftChild =
  TST - INSERT(root->leftChild, word)
else if *word is greater than root->ch:
  root->rightChild = TST - INSERT(root-
>leftChild, word)
else do:
  if *(word + 1) is not NULL then :
    root->middleChild=TST-INSERT(root-
>middleChild, word + 1)
else do :
  root->isAWord = true

END TST - INSERT
```

TST-INSERT terminates with a constructed TST, which is unbalanced, considering that the .txt file is alphabetized. Searching in the unbalanced TST degenerates into $O(l \times n)$, where l is the length of a word and n is the number of nodes in the TST. In order to take full advantage of the BST ideas implanted in the TST, it became apparent that we had to balance the TST to facilitate a logarithmic search time of $O(l \times log_3\, n)$, where n is the number of nodes. To this end, a number of algorithms were utilized:

TST-BALANCE: Sets the count field of each node, balances a level of the TST, as well as balance all levels of the TST in $O(m \times log_3\, n)$ time and space, where m is the number of levels to be balanced and n is the number of nodes in the unbalanced TST. It takes as input the root of the unbalanced TST and produces a balanced TST with a new root as its output;

Balance TST:
TST - BALANCING

```
TST - BALANCE(TSTnode* root) :
        root->count = TST -
SetCount(root)
        root = TST - BalanceALevel(root)
        TST - BalanceAllLevels(root)
        END TST - BALANCE
```

TST-SetCount: Sets the number of children nodes on the same level. Its input is the root of the unbalanced TST, and it's output is assigning a value to the count field of each node in the unbalanced TST;

† https://github.com/hangbogu/CMPE130

Balance TST:
TST – SET-COUNT

```
TST – SetCount(TSTnode* root) :
        if root is NULL then : return 0

                root->count = TST -
SetCount(root->leftChild) + TST -
SetCount(root->rightChild) + 1
                TST – SetCount(root-
>middleChild)

                return root->count
                END TST – SetCount
```

TST-BalanceALevel: Balances a level of the TST. Its input is the root of a subtree of the TST, and its output is the balanced subtree, as well as its left and right children subtrees;

Balance TST:
TST – BALANCE-LEVEL

```
TST – BalanceALevel(TSTnode* root) :
if root is NULL or root->count is 1
then : return root
//Make middle node the root
root <--divide(root, (root->count / 2))
//Recursively balance subtrees
root->leftChild =
  balanceLevel(root->leftChild)

root->rightChild =
  balanceLevel(root->rightChild)
return root
END TST – BalanceALevel
```

TST-Divide: Finds the pivot node--that is, the subtree whose root will be the new root of the TST. Its inputs are the root of an unbalanced subtree and a precomputed pivot position; its output is a pointer to the new pivot node;

Balance TST:
TST – DIVIDE

```
TST – Divide(TSTnode* root, unsigned
pivotPos) :
        leftChildCount <-(root-
>leftChild is NULL)?0:root->leftChild-
>count

if pivotPos is less than leftChildCount
then :
root->leftChild = TST – Divide(root-
>leftChild, pivotPos)
root <--TST – RotateRight(root)

else if pivotPos is greater than
leftCount then :
root->rightChild =
TST – Divide(root->rightChild,
      pivotPos – leftChildCount – 1)

root <- RotateLeft(root)

return root
END TST – Divide
```

TST-BalanceAllLevels: Balances all levels of the TST. Its input is the root of the unbalanced TST, and its output is a balanced TST;

Balance TST:
TST – BALANCE ALL LEVELS

```
TST-BalaneAllLevels(TSTnode* root) :
if root is NULL then : return
 root->middleChild =
 TST-BalaneALevel(root->middleChild)
 TST-BalaneAllLevels(root->middleChild)
 TST-BalaneAllLevels(root->leftChild)
 TST-BalaneAllLevels(root->rightChild)
END – TST – BALANCE ALL LEVELS
```

TST-RotateRight: Performs right rotation on the TST to preserve the BST order property. Its input is the root of a subtree of the unbalanced TST; its output is the restoration of the BST order property if the right subtree of a subtree does not conform to the BST order property;

```
Balance TST:
TST - ROTATE RIGHT

TST - RotateRight(TSTnode* root) :
nodeX <--root->leftChild

//Move the subtree mounted at root
//between nodeX and root
root->leftChild = nodeX->rightChild
//Swap nodeX and root
nodeX->rightChild = root
//Restore count field of root
root->count =
 (root->leftChild ? root->leftChild-
>count : 0) +
 (root->rightChild ? root->rightChild-
>count : 0) + 1

nodeX->count = (nodeX->leftChild ?
nodeX->leftChild->count : 0) +
nodeX->rightChild->count + 1

return nodeX
END TST - RotateRight
```

TST-RotateRight: Performs left rotation on the TST to preserve the BST order property. Its input is the root of a subtree of the unbalanced TST; its output is the restoration of the BST order property if the left subtree of a subtree does not conform to the BST order property.

```
Balance TST:
TST - ROTATE LEFT

TST - RotateLeft(TSTnode* root) :
nodeX <--root->rightChild

//Move the subtree mounted at root
//between nodeX and root
root->rightChild = nodeX->leftChild
//Swap nodeX and root
nodeX->leftChild = root
//Restore count field of root
root->count =
 (root->leftChild ? root->leftChild-
>count : 0) +
(root->rightChild ? root->rightChild-
>count : 0) + 1;

nodeX->count =
 nodeX->leftChild->count +
(nodeX->rightChild ? nodeX->rightChild
->count : 0) + 1;

return nodeX
END TST - RotateLeft
```

After balancing the TST, the next step is to compress the TST into a TDAWG. Recall that the goal is to find equal subtrees in the balanced TST and replace all links to these subtrees with a link that leads to only one of the subtrees. To this end, a number of algorithms were utilized:

TST-Compress: Compresses the balanced TST to form a TDAWG. Its input is the root of the balanced TST, and its output is a TDAWG in which duplicate suffixes are removed in O(m × $\log_3$ n) time and space--where m is the number of the equal subtrees and n is the number of nodes of the equal subtrees;

```
Compress TST to DAWG:
TST - COMPRESS

TST - Compress() :
//Compute the hash field for all nodes
//except the root
        root->hash = TST -
ComputeHash(root)

//Remove duplicate suffixes beginning
//with the longest one
TST - RemoveDuplicateSuffixes(root)

END TST - Compress
```

TST-ComputeHash: computes the hash field for all nodes in the balanced TST except the root. This is done by computing the hashes for the children recursively and using them for the current hash in O(n) time and space. Its input is the root of the balanced TST, and its output is assigning a value to the hash field of every node in the balanced TST, save the root node;

† https://github.com/hangbogu/CMPE130

Compress TST to DAWG:
TST – COMPUTE HASH

```
TST – ComputeHash(TSTnode* root) :
if root is NULL  then : return 0
//Compute the hashes for the children
//of a node recursively and use them
//for the current hash

root->hash=(root->symbol – 'a') + 31
*TST-ComputeHash(root->middleChild)
root->hash^=TST-ComputeHash(root
->leftChild)
root->hash ^= TST – ComputeHash(root
->rightChild)

root->hash ^= (root->hash >> 16)

//Resulting hash field must be unsigned
//to facilitate the correct modulo
//computation
root->hash %= HASH_TABLE_SIZE
return root->hash
END TST – ComputeHash
```

TST-RemoveDuplicates: removes duplicate suffixes starting from the longest one. It does so by first checking if the current node is in the hash table. Its input is the root of the balanced TST, and its output is the removal of nodes of balanced TST consisting of duplicate suffixes;

Compress TST to DAWG:
TST – REMOVE DUPLICATES

```
TST – RemoveDuplicates(TSTnode* root)

if root->leftChild is not NULL then :
if TST-CheckAndRemoveDuplicates(&root
->leftChild) is not NULL then :
 TST – RemoveDuplicates(root
 ->leftChild)
if root->rightChild is not NULL then:
if TST –
CheckAndRemoveDuplicates(&root
->rightChild) is not NULL then :
   TST – RemoveDuplicates(root
->rightChild)
if root->middleChild !NULL then:
if CheckAndRemoveDuplicates(&root
->middleChild) !NULL then :
   TST – RemoveDuplicates(root
   ->middleChild)

END TST – RemoveDuplicates
```

TST-CheckAndRemoveDuplicates: determines if a node of the balanced TST has been hashed. Its input is the root of the balanced TST, and its output is true if the current node is already hashed and subsequently removed; otherwise, it is false.

Compress TST to DAWG:
TST – CHECK AND REMOVE DUPLICATES

```
SEARCH(TSTnode* root, char* word) :
//root is null: word does not exist
if root is equal to NULL then : return
false
  if *word is less than root->symbol
then :
return TST – SEARCH(root->leftChild,
word)
  else if *word is greater than root-
>symbol:
return TST – SEARCH(root->rightChild,
word)
  else if *(word + 1) is not equal to
the NULL char then :
return TST – SEARCH(root->middleChild,
word + 1)
else
return root->isAWord
END TST – SEARCH
```

Now that a TDAWG has been created, the next step is to traverse the TDAWG to lookup a query in O($l \times log_3\, n$) time and space, where l is the length of the query and n is the number of nodes in the TDAWG. To this end, the **TST-Search** algorithm takes as inputs the root of the TDAWG and a string query and returns true if the query is found, or false otherwise. Searching in the TDAWG is done in a DFS postorder traversal fashion:

† https://github.com/hangbogu/CMPE130

Spelling Corrector:
TST – SEARCH

```
SEARCH(TSTnode* root, char* word) :
//root is null: word does not exist
if root is equal to NULL then : return
false
  if *word is less than root->symbol
then :
return TST - SEARCH(root->leftChild,
word)
  else if *word is greater than root-
>symbol:
return TST - SEARCH(root->rightChild,
word)
  else if *(word + 1) is not equal to
the NULL char then :
return TST - SEARCH(root->middleChild,
word + 1)
else
return root->isAWord
END TST - SEARCH
```

When TST-Search returns false, our implementation attempts to correct the spelling to within an edit distance of 2 . The algorithm **TST-Correct**, coupled with an overloaded **TST-Correct** and **TST-Edits** are utilized for our spell corrector.

TST-Correct: returns the search query if the query is spelled correctly: that is, the query is found in the dictionary; otherwise, it returns an empty string within O(1) time and space:

Spelling Corrector:
TST – CORRECT

```
TST - Correct(char* word) :
//search for word in TST DWAG: if found
//return word, else return empty string
(TST - Search(root, word)) ? return
word : return ""
END TST - Correct
```

TST-Correct: the overloaded counterpart to TST-Correct, attempts to correct the misspelled query within an edit distance of 2. Its inputs are the string query and a string vector, and its output is a correct suggestion list, if any:

Spelling Corrector:
TST – CORRECT

```
TST - Correct(const char* word,
vector<char*> result) :
TST - Edits(word, result)

for i <-- 1 to candidates.size()
inclusive do :
if TST - Search(root, result[i]) then :
        candidates.push_back(result[i])
        suggestions <-- true

  if suggestions is true then :
                display strings in
candidates
  else
    do:
for <-- 1 to result.size() inclusive do
:
TST - Edits(result[i], subResult)

for i <-- 1 to subResult.size()
inclusive do :
if TST - Search(root, subResult[i])
then
  candidates.push_back(subResult[i])
  suggestions <-- true
  if suggestions is true then :
    display strings in candidates
    display word is not found in DAWG
END TST - Correct
```

TST-Edits: performs four edits on a misspelled query. These edits are insertions, deletions, transpositions, and alterations. For a given misspelled query of length n, TST-Edits performs □ deletions, □ − *1* transpositions, *36*□ alterations, and *36*(□ + *1*) insertions, for a total of *74*□ + *35*operations for edit distance 1. For edit distance 2, TST-Edits performs (*74*□ + *35*)^{2}operations, engendering O(□^{2}) time and

† https://github.com/hangbogu/CMPE130

space complexity:

Spelling Corrector:
TST – EDITS

```
TST - Edits(const char* word,
vector<char*> result) :
//let n be the length of word

//perform n deletions on word
for index <-- 0 to word.length()
exclusive do :
result.push_back(word.substr(0, index)
+
word.substr(index + 1))

//perform n-1 transpositions on word
  for index <-- 0 to word.length() - 1
exclusive do :
  result.push_back(word.substr(0,
index) + word[index + 1] +
  word[index] + word.substr(index + 2))

    for j <-- 'a' to 'z' inclusive do :
//perform 36n alterations on word
      for index <-- 0 to word.length()
exclusive do :
      result.push_back(word.substr(0,
index) + j +
          word.substr(index + 1))

//perform 36(n+1) insertions on word
        for index <-- 0 to
word.length() + 1 exclusive do :
        result.push_back(word.substr(0,
index) + j +
               word.substr(index))
END TST - Edits
```

E. Results and Findings

I. What evaluation metrics did you Use?

Our solution is evaluated utilizing the following criteria: the number of nodes and words in the TDAWG, as well as the query search speed in the TDAWG. The most critical aspect is the preservation of words in the TDAWG: it is the most important because if a word is lost in the process of balancing and compressing the TST, then the TST represents a dictionary different than that which was downloaded. Thus, the number of words must be the same in the downloaded file and the TDAWG. The second most important evaluation of the project is how well the team was able to compress the balanced TST, more specifically the number of nodes counted at the time of creating the TST, balancing the TST, and then compressing the TST. Previous sections provided a time and space complexity analyses for these and show that the TDAWG is an efficient data structure for our purposes.

I. Were other systems evaluated in the same problem? How did your system do in comparison?

The only other system evaluated on the same problem was GNU's Aspell. Using the command line, we added a medium sized dictionary of ~100,000 words to Aspell's personal dictionary and then conducted search queries of correctly spelled and misspelled words on the augmented personal dictionary. The same dictionary size and operations were used on our TDAWG.

† https://github.com/hangbogu/CMPE130

For fair timing, both Aspell and our TDAWG were ran on the same machine. Our code was optimized on the command line at compile time to match Aspell's optimization as close as possible: these include, but certainly not limited to, replacing length() with inline code, reordering tests, maintaining pointer to a pointer, and transforming recursion to iteration. The table below captures the results of the tests. The values represent the time taken in seconds:

Table 1: *TDAWG compared to its closest competitor, Aspell.*

Operations	Insert		Search / Spell Corrector	
Data Structure	Aspell (Hash)	TDAWG	Aspell (Hash)	TDAWG
CPU i3-4005U @1.70GHz	1.12s	0.93s	0.58s	0.52s

From the table above, it is clear that the two data structures have comparable results, but our TDAWG performs slightly better than that of GNU's Aspell.

II. Show graphs/tables with results

Figure 6 depicts the number of nodes at the time that the DAWG is initially built and the number of nodes resulting from the compression. In order to provide a better measurement the team ran the algorithm in two different dictionaries; the first composed of 370,099 english words which translated to 1,397,908 nodes, and the second composed of 466,544 words and 1,420,649 nodes. After the compression algorithm is executed the number of nodes are reduced to only 273,956 nodes for the first dictionary and to 466,544 for the second dictionary.

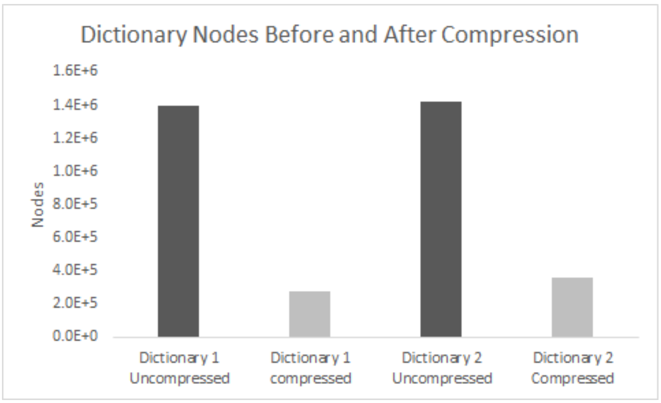
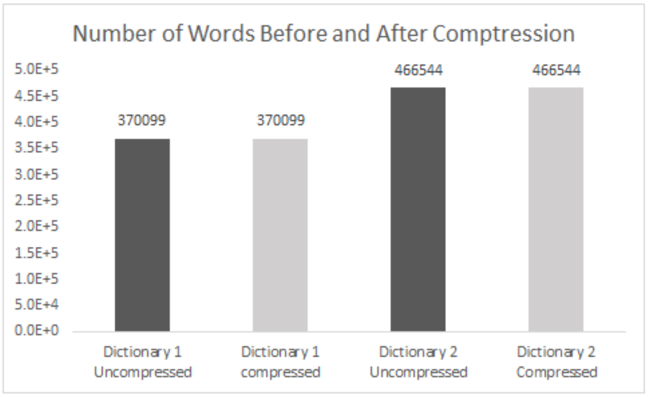


Figure 6: *Number of Nodes Before and after compression*



† https://github.com/hangboqu/CMPE130

Figure 7: *Number of words after compression. Note that the number of Nodes is significantly reduced but the number of words remains constant.*

Figure 7 shows the word count before and after the node compression. The most important aspect of our implementation is that the number of words at the beginning of the input and at the final solution must remain the same, as well as the content of the words. In order to guarantee dictionary integrity the team printed the entire dictionaries before and after compression into different files and ran the "*diff*" utility on the compressed dictionary file against its uncompressed counterpart; the utility returned no differences found in both instances.

III. Error Analysis

It must be noted that the benchmark tests did not consider the time taken to create the TDAWG. That is, the time required to balance the unbalanced TST and then compress the balanced TST to form a TDAWG. Further, the optimizations executed at the command line were fashioned to mirror Aspell's documentation as closely as possible. There may be other optimizations necessary, which may engender different results. Another factor is that the tests were done on only a single machine.

F. Conclusions and Lessons Learned

The team satisfied the collective yearning to understand how text editors and search engines generate correct misspelled words and provide suggested strings based on user input and the dictionary used. Along the way the team became familiarized with new data structures such as the Trie, Acyclic graphs as well as new algorithms such as the TST compression algorithm and the word suggestion algorithms postulated by Norvig [3].

In the final stages of the project the team attempted to enable a feature that would allow the user to either insert a new word or remove a word. However, this feature is not considered feasible because it would require large amounts of memory to be allocated to maintain the uncompressed trie. The difficulty arose because when an attempt is made to change the compressed TDAWG the inter connections of the pointers change and the entire structure is changed therefore the contents can only be manipulated when the graph has not yet been compressed.

A suggestion for future improvement is to enable the user to add or delete words from the TDAWG, as this is a highly desirable feature given that dictionaries are constantly evolving. A further suggestion for

† https://github.com/hangboqu/CMPE130

improvement is to increase the system's portability and enable it's use as a feature in a larger system given the flexibility of input of the TDAWG it can be applied as a solution to projects requiring correct string input or suggestions.

References:

[1] J. Bentley, B. Sedgewig, *Ternary Search Trees*. Boston, MA: Dr. Dobb's Journal, 1999.

[2] S. Miyamoto, S. Inenaga, M. Takeda, A. Shinohara, *Ternary Directed Acyclic Word Graphs*. Fukouka, Japan: Japan Science and Technology corporation, June 2003.

[3] Norvig, P. (2017). *How to Write a Spelling Corrector*. [online] Norvig.com. Available at: http://norvig.com/spell-correct.html [Accessed 29 Nov. 2017].

[4] GNU Aspell
In-text: (Aspell.net, 2017)
Your Bibliography: Aspell.net. (2017). *GNU Aspell*. [online] Available at: http://aspell.net/ [Accessed 30 Nov. 2017].

† *https://github.com/hangbogu/CMPE130*