CS5800 Algorithms – Section 4
Exam Fall 2018 – 2 Hours

1. Mysterious Hashing (25 points)

A smart programmer decided to improve hashing with linear open addressing using the following insertion code:

```
          Insert( T, m, x)    // Hashing table T[0..m-1] of size m, new
element x
          int dist, i, j;
        {
             j = hash(x); i = j;
             while( T[i] != EMPTY ) {
                 dist = abs(j-i) - abs(hash(T[i])-i)
                 if( i > j ) dist = abs(hash(T[i])-i) - abs(j-i)
                 if( dist < 0 ) {
                      temp = T[i]; T[i] = x; x = temp;
                 }
                 i = i-1; if( i < 0 ) i = m-1;
                 if( i == j ) Error( "Full table" );
              }
              T[i] = x;
         }
```

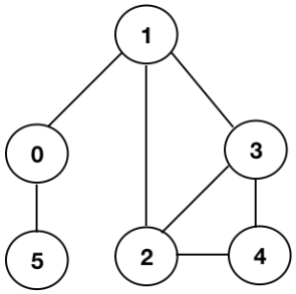
What the programmer is improving? Justify.

This insertion computes the distance to the first hashing location of the new element as well as for the old element and moves the one that has the shortest distance (15 points). Therefore this insertion improves the variance of the search time as tries to have all chains with the same length (10 points).

2. Single points of failure (25 points)

While designing a robust cellular network, Alice was worried that a single point of failure may completely break the network. A **single point of failure** is defined as a node in the network graph, which if removed (along with its edges), will cause the number of connected components in the graph to increase. Let us see this with an example:

The graph besides is a single connected component. If you remove node 2 and edges (2-1, 2-3, 2-4), the resulting graph will still be a single connected component. However, if you remove node 1 and edges (1-0, 1-2, 1-3), you will have 2 connected components: one with nodes (0,5) and other with nodes (2,3,4). Hence, node 1 is a single point of failure. Write an algorithm to find out **all nodes** that are a single point of failure in a given graph, as there can be more than one. In the example above there are 2: 0 & 1.



Brute Force Approach:

```
SINGLE_POINT_OF_FAILURE(V, E)

num_raw_components = count number of connected components in graph G(V,E)
results = []
for each vertex v in V
do
   G' = remove v from graph G along with its edges and create new graph G'
   num_new_components = count number of connected components in graph G' using DFS
   if num_new_components > num_raw_components
   do
            results.add(v)
  done
done
return results
```

Time Complexity: O(V*(V+E))

Optimal Approach:

Define DFS tree as a tree of vertices such that x -> y if vertex x discovered vertex y during DFS

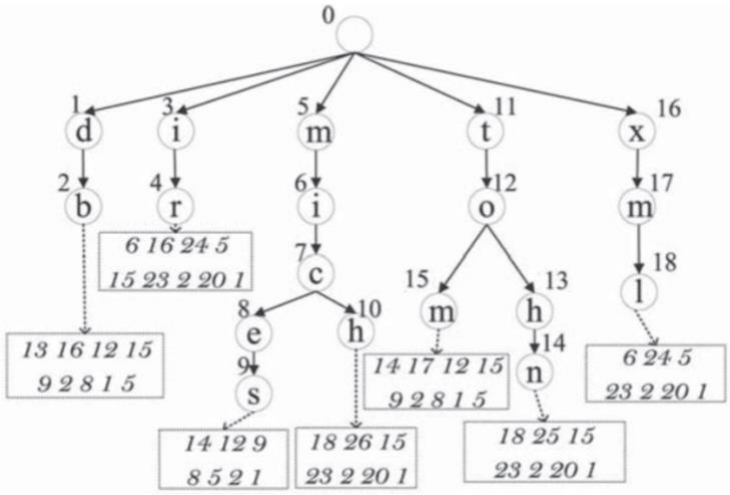
```
SINGLE_POINT_OF_FAILURE(V, E)

results = []
DFS_Tree = Create DFS tree of graph G(V,E)
If DFS_Tree.root has more than 1 children
do
    results.add(root)
done
for every non-root & non-leaf node "n" in DFS_Tree
do
   if there is no back-edge from any subtree rooted in children of node n to any parent of n
   do
       results.add(n)
   done
done
return results
```

Time Complexity: O(V+E)

3. <u>Searching Palindromes</u> (25 points)

A group of friends, Ada, Anna, Ava, Bob, Elle, Eve, Hannah, and Otto, want to know **how many occurrences of palindromic words of at least two letters** are in a set of documents as well as **how many documents contain palindromes**. They know that the documents are already indexed using an **inverted index** where the vocabulary is stored in a **trie** and each leaf of the trie points to a linked list (inverted list) that contains all documents that contain a word. Every node in the linked list contains a document id and the number of occurrences in that document, that can be retrieved using the fields *docid* and *count*. It also contains the field *next* that gives the next node in the list.

The tree besides shows an example where the letter labels of the branches are in the trie nodes and every rectangle represents a linked list where all the document ids are shown. Notice also that, for layout purposes, the nodes 13 & 15 are in swapped positions. In this example, the words that appear in more documents (9) are "db", "ir", and "tom", while all other words appear in 7 documents. For this example, the solution to the questions above is {0, 0} as there are no palindromes at all.



Write the pseudocode of an algorithm that solves the friends' questions and indicate the temporal complexity of your solution, considering that the following operations are supported by the index:

- Search(w) returns a pointer to the inverted list of word w (*e.g.*, Search(w)->docid is the docid of the first document in the list that contains w).
- Follows(w) returns the word that follows w in the trie vocabulary (*e.g.*, Follows("db") returns "ir"). The word w may not exist and still works.

What you would change to this data structure to make your algorithm faster? Justify if your change implies more space.

The pseudocode for counting the palindromes (15 points) is the following:

```
Palindromes = 0;  Paldocs = 0;
word = Follows( "" );  // "a" is OK too, word is a pointer to a string
While( word <> NULL ) {
        If( Length_string( word ) > 1 &&      // word has more than one letter
            Equal_string( word, Reverse_string( word ))) { // is palindrome
                List = Search( word );

                While( List <> NULL ) {
                        Palindromes += List.count; Paldocs++;
                        List = List.next;
                }
```

```
            }
            word = Follows( word );
    }
    Report Palindromes, Paldocs;
```

The complexity for the searches and follows is O(sum(lengths of words in vocabulary)) as it is a trie. This will be proportional to the size of the vocabulary, V. The complexity of traversing the lists will be O(number of palindromic word occurrences in documents). This in the worst case is proportional to the sum of the number of different words in each document, which in time is proportional to the size of the documents O(N). As V is much smaller than O(N), the final complexity is linear (5 points).

A simple change would be to have counters for the total number of documents and occurrences for every word in each leaf of the tree. This implies O(V) extra space but makes the previous algorithm much faster as there is no need to traverse the lists and we get O(V) time (5 points).

4. Vertex Cover (25 points)

Vertex cover is a graph problem defined as follows. Given a graph G(V,E) and a positive integer K, you have to find if there is a subset of vertices V' of size at most K such that every edge in the graph G is connected to some vertex in V' (that is, all edges are covered).

Using NP reductions, prove that the Vertex Cover problem is NP Complete. For this problem, you can assume that Clique, 3-CNF-SAT, SAT and CIRCUIT-SAT are given as NP complete problems.

Proof: We first show that Vertex Cover is in NP. Given a graph G(V,E), an integer K and a certificate V' which is a set of vertices, it is easy to check whether each edge in G is adjacent to some vertex in V' or not. This can be done in a polynomial time algorithm. Hence vertex cover is in NP.

To prove vertex cover is in NP-Complete, we reduce the CLIQUE problem to it. This can be done by the following construction: Given an instance of a CLIQUE problem with graph G(V,E) and an integer K, we construct a new complementary graph G'(V,E'). G' is constructed in a way such that for every pair of vertices (u,v) which has an edge in G, we remove the edge in G' and vice-versa.

Now, let us assume G has a CLIQUE of size K and this set of vertices is called S. Since set S in G has a clique, the same set of vertices S in G' should have no edges among them (by construction of graph G'). Hence, every edge in G' must be incident to some vertex in V-S which by definition is a vertex cover of size (n-k) where "n" is the total number of vertices in the graph. We can also use the same logic to prove the other direction Vertex Cover -> Clique. Hence, there is a clique of size k in G if and only if there is a vertex cover of size n-k in G'.

Since we know that CLIQUE is in NP-Complete and the above reduction can be obtained in polynomial time, Vertex Cover is also in NP-Complete.

Bonus Question (10 points)

In class we have seen several data structures or techniques that are faster because one of their parameters is over a finite range of values. Name two such examples, indicating which parameter is finite and what improvement is obtained.

- radix sort, bounded integer numbers
- tries, bounded alphabet
- bit parallelism, bounded alphabet
- etc