Elijah Andrushenko

CPTS 453

Graph Theory

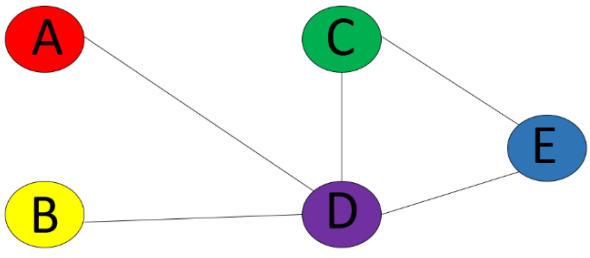
12-09-2019

Applications of Graph Theory in Genome Sequencing

Genome Sequencing is a major topic when it comes to bioinformatics. The ability to match various genome sequences between each other to notice patterns and similarities is very important when it comes to understanding our DNA and what we're made of. The one major difficulty that exists when it comes to matching genome sequences is the fact that there are so many base pairs. Humans have 23 pairs of chromosomes adding up to 46 chromosomes in total. Within these chromosomes exist base pairs which is what the DNA double helix is made of. The 4 base pairs of DNA are Adenine, Guanine, Thymine, and Cytosine. These are encoded as A, G, T, and C respectfully. The reason it is hard to sequence these base pairs has to do with the sheer number of them that exist. In the human genome there are over 3 billion base pairs. Due to this high number of base pairs it is computationally expensive in both memory storage and time complexity when trying to align two genome sequences. The gnome number is already very high in humans where as if we want to compare other species' genomes that number increases even further. Therefore to help lower computational space and time applications learned in graph theory are required. Some of these graph theory applications include the use of matrices with the use of an algorithm called the Needleman-Wunsch Algorithm also known as global alignment. Then there are also suffix trees which derive from the concept of trees in graph theory.

In graph theory a matrix is a 2-dimnesional array that consists of column vectors and row vectors. It is used to as another way to represent a graph and show the information of the graph in a different but still mathematical way. Some of the ways that a matrix can represent a graph is by using an adjacency matrix. An adjacency matrix labels every row and column as a vertex and the cells represent how many edges are between those matrices. A 0 in a cell means that the two vertices don't share an edge, and a 1 in a cell means that the two vertices do share an edge, sometimes a 2 is used to represent a vertex that has an edge to itself, which is called a loop. An example of an adjacency matrix would be

	A	***B***	***C***	***D***	***E***
A	0	0	0	1	0
B	0	0	0	1	0
C	0	0	0	1	1
D	1	1	1	0	1
E	0	0	1	1	0



Below the adjacency matrix is the graph that would be constructed from it. Taking the sums of either a column or a row of an adjacency matrix will tell you how many edges a certain vertex has. In this case Vertex D has 4 edges.

Using the basic understanding of what a matrix is, we can perform the alignments on genomes. One of the more popular alignment algorithms is called the Needleman-Wunsch algorithm, also known as, global alignment. The goal of global alignment is to be able to find the best DNA sequence whether there are gaps, matches, and mismatches. It is then scored and these scores can be compared to other alignments to see which genome sequences are most similar. To understand how it works we first need two sequences to compare. As an example we will have

$$S1 = acagag$$

$$S2 = acaagt$$

We must also have scores these will be explained later but for now we will assign

$$Match = 1$$

$$Mismatch = -1$$

Now that we have our scores and sequences figured out we can then initialize a matrix. The starting spots will have a gap and then S1 will be on the first column and S2 will be on the first row. We initialize the red 0 as a starting point and then along the first row and column we subtract down 1 until we reach the end. These are shown as green for clarity.

$$\begin{bmatrix} & - & a & c & a & a & g & t \\ - & 0 & -1 & -2 & -3 & -4 & -5 & -6 \\ a & -1 & & & & & & \\ c & -2 & & & & & & \\ a & -3 & & & & & & \\ g & -4 & & & & & & \\ a & -5 & & & & & & \\ g & -6 & & & & & & \end{bmatrix}$$

Now that the matrix has been initialized we can proceed with comparing the two sequences. To do scoring we need to know the information of three other cells in the matrix. For example to get the value of *X* we need to know the value of the cell above it, the cell to the left of it, and the cell that is diagonally left and above it. To clarify the three numbers needed to find *X* are colored in red. We must then try to maximize the value of *X*. Since the column *X* lies on has an "a" representing it as well as an "a" representing it on the row we consider this a match. So we try to find the maximum value *X* can be. This is essentially the rule to calculating every cell in the matrix.

$$X = Max\begin{cases} above = X = -1 + 1 = 0 \\ Left = X = -1 + 1 = 0 \\ Diagonal = X = 0 + 1 = 1 \end{cases}$$

$$\begin{bmatrix} & - & a & c & a & a & g & t \\ - & 0 & -1 & -2 & -3 & -4 & -5 & -6 \\ a & -1 & X & Z & & & & \\ c & -2 & Y & & & & & \\ a & -3 & & & & & & \\ g & -4 & & & & & & \\ a & -5 & & & & & & \\ g & -6 & & & & & & \end{bmatrix}$$

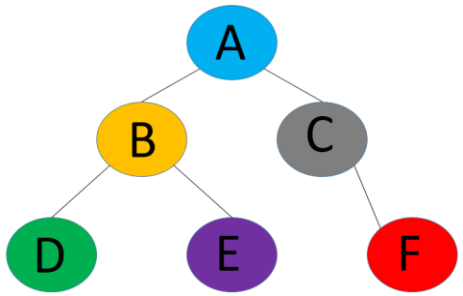
Since the maximal score we can get for X is 1 we can relabel it as a 1. Thus we have labeled our first cell. As a reminder we only added 1 because it was a match we would subtract 1 if it was a mismatch. Finding the value of X now gives us enough information to be able to calculate both Z and Y since we now have the three cells required to solve for them and as we continue to solve cells we will be able to solve other that were previously unsolvable, which is how we will complete this table.

Here is the completed table that we have been working on. This number shown in red represents the similarity score of these two sequences S1 and S2.

$$\begin{bmatrix} & - & a & c & a & a & g & t \\ - & 0 & -1 & -2 & -3 & -4 & -5 & -6 \\ a & -1 & 1 & 0 & 1 & 2 & 1 & 0 \\ c & -2 & 0 & 2 & 1 & 0 & 1 & 0 \\ a & -3 & 1 & 1 & 3 & 3 & 2 & 1 \\ g & -4 & 0 & 0 & 2 & 2 & 3 & 2 \\ a & -5 & 1 & 0 & 3 & 3 & 2 & 2 \\ g & -6 & 0 & 0 & 2 & 2 & 3 & 2 \end{bmatrix}$$

The Needleman-Wunsch algorithm is very time effective when it comes to scoring two sequence alignments. It has a worst-case time and space complexity of O(mn) where m and n are the two sequences that are being compared. This is linear time which is considered relatively fast for computer science and bioinformatics standards.

Trees in graph theory are graphs that are connected, acyclic, and undirected. A graph G is a tree if it has n number of vertices and $n - 1$ number of edges. Removing an edge from the tree would result in the tree being disconnected and would no longer be a tree which is a property of all trees. An example of a tree can be seen here.

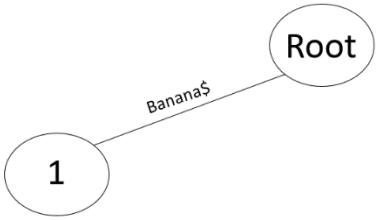


Some terminology associated with trees exist as well. For example using the tree above the vertex A is called the root, it is the parent of vertices B and C. Vertex A is also the grandparent of vertices D, E, and F. Another term to know is that vertices D and E are called siblings to each other since they have the same parent, vertex B. Lastly vertices D, E, and F are often called leaves, which means that they don't have any children.

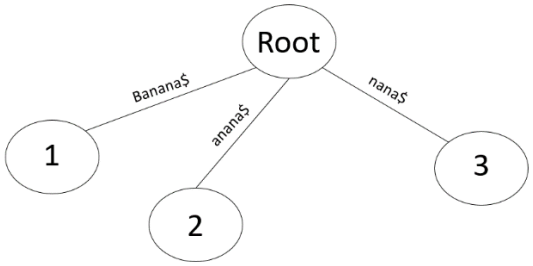
Trees are a useful way to store information, they are often used in computer science data structures for the purposes of searching for and sorting data. There exists a variant type of tree and it is called a suffix tree. Suffix trees allow for efficient storage of data and quick searching when inserting more suffixes into the tree. Since we understand what a tree is, we can then understand what a suffix tree is and how it works. First we need a string of letters, for use in our example we will build a suffix tree out of the work banana. Before we begin we need to add a wildcard character to end of the word we are using. In our case we will use the symbol $ to represent the end character, so now the string we are working with is banana$. With this word we must now generate all of the suffixes of banana$ which are,

1. $banana\$$
2. $anana\$$
3. $nana\$$
4. $ana\$$
5. $na\$$
6. $a\$$
7. $\$$

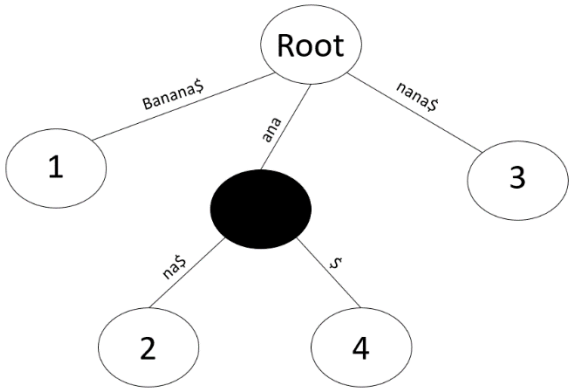
To start we will initialize a root node. And then we will insert the largest suffix first and the smaller ones after, they are ordered numerically above. Edges are labeled with the suffix and the numerical order will label the vertex that connects the suffix edge.



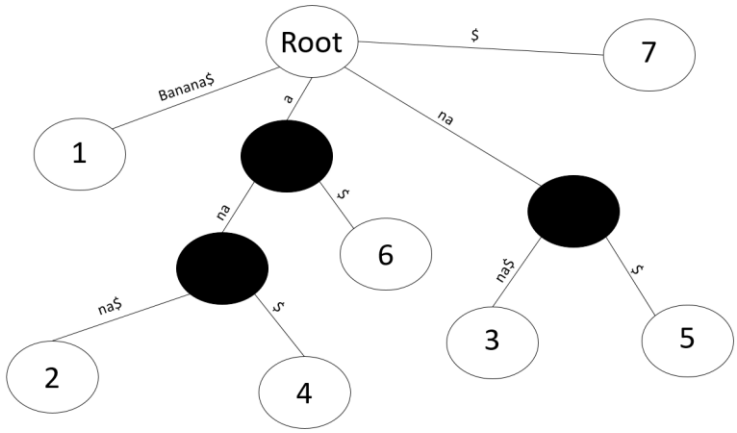
Then proceed to add in the other suffixes like anana$ and nana$.



Now when we add the 4th suffix, ana$, it is clear that is has a similar starting suffix as a suffix that is already inserted into our tree, which would be anana$. Since both suffixes ana$ and anana$ both start the same up to the point ana we will implement that into our tree like so.

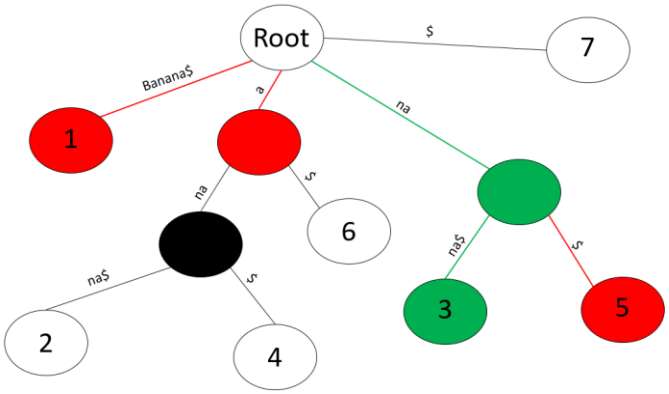


With our tree of suffixes so far we have entered in 4 of them. We do not stop at black nodes to reobtain our suffixes. To recreate suffix 4 we concatenate every string onto the end of the next edge we encounter until we reach the root. So if we start at vertex 4 and move toward the root we start with $. Then we reach a black node, and then we run into the edge ana so we take our current sum of strings which is just $ and add that to the back of ana so get ana$ which is how we get suffix number 4. The same can be done with all numbered nodes in the tree.



This is the completed tree when inserted the string banana. We can reconstruct all 7 suffixes using this tree and they are stored in an efficient way. One of the most important uses of suffix

trees are pattern matching. For example, if we want to find the substring nana$ in our tree we can search for it easily.



First we look left and compare the first character of our word we are searching for nana$ with Banana% since $n \neq B$ we check a different branch of the tree. $n \neq a$ So that entire branch is disqualified and does not need to be searched. Then we get na which matches the first two letters in nana$ so we go further down and we find nana$ which is an exact match and we are done. Although we used Banana as our example in this tree we can use much longer strings like string DNA genome sequences to find the longest common substring between two sequences. Although construction of a suffix tree is computationally slow it excels in look up which takes much longer than construction.

In conclusion there are several algorithms in graph theory that lay the foundation of problem solving in many other fields, such as computer science, mathematics, and bioinformatics. For further investigation into the application of matrices and their use in genome sequencing, it is recommended to also look at the Smith-Waterman Algorithm, also known as, local alignment. For a stronger understanding into trees and their use in genome sequencing it is recommended to look into Ukkonen's Algorithm for tree construction.

References

McCreight, Edward M. (1976), "A Space-Economical Suffix Tree Construction Algorithm", Journal of the ACM, 23 (2): 262–272, CiteSeerX 10.1.1.130.8022, doi:10.1145/321941.321946.

Hariharan, Ramesh (1994), "Optimal Parallel Suffix Tree Construction", ACM Symposium on Theory of Computing.

Farach, Martin (1997), "Optimal Suffix Tree Construction with Large Alphabets" (PDF), 38th IEEE Symposium on Foundations of Computer Science (FOCS '97), pp. 137–143.

Kim, Jin H.; Pearl, Judea (1983), "A computational model for causal and diagnostic reasoning in inference engines", in Proc. 8th International Joint Conference on Artificial Intelligence (IJCAI 1983), Karlsruhe, Germany, August 1983 (PDF), pp. 190–193.

Deo, Narsingh (1974), Graph Theory with Applications to Engineering and Computer Science (PDF), Englewood, New Jersey: Prentice-Hall, ISBN 0-13-363473-6

Biggs, Norman (1993), Algebraic Graph Theory, Cambridge Mathematical Library (2nd ed.), Cambridge University Press, Definition 2.1, p. 7.

Goodrich, Michael T.; Tamassia, Roberto (2015), Algorithm Design and Applications, Wiley, p. 363.