Calculating Parameters of De Bruijn Graphs

Written by

JOEL RICHARD MOORE

CS598-05, Project

In Partial Fulfillment of The Masters in Computer Science

The State University of New York Polytechnic Institute

DEC, 2014

Contents

1 Acknowledgments

I owe a great deal of recognition for the research enclosed in this report to my technical mentor Victoria Horan, who when approached and asked if I may assist her in researching de Bruijn graphs, graciously and without hesitation welcomed my involvment. I extend my sincere gratitude to her for her guidance and patience, the fruit of which this project manifests.

I also wish to thank my project advisor, Jorge Novillo, for not only guiding me through this project, but also through two of the most rigorous courses in the department (Algorithms and Automata).

Also a special thank you to Todd Humiston, who is not only my branch manager but also my professional mentor and the one person—above all others—whom I look up to for sound professional direction. Thank you for giving me the opportunity of a lifetime.

Additionally, I wish to thank my benefactor, the United States Air Force; and the Air Force Research Laboratory, Information Directorate at Rome, New York for giving me this opportunity to pursue a master's degree and the best career that I could ever imagine!

2 Abstract

Background: De Bruijn networks are a special topology of graphs that are interesting as a network model because their physical properties endow high fault-tolerance and offer a more robust communication than many other network models. Sometimes when an arbitrary node within the network is excited, it is desirable to know which other nodes are in the vicinity and through proximity may also be affected. In other words, given a node, x, find all nodes within a distance d from x. An obscure subject, there is little literature covering de Bruijn graphs outside of general characteristics and theory. In our research, we have found no published mention of an algorithm to generate all nodes within a given distance d from a node x.

Results: In this paper, we present our research to design and implement a library compiled in MATLAB. This library calculates many useful de Bruijn graph parameters to include calculating directed and undirected distance between nodes, producing adjacency and minimum distance matrices, and some functions to generate lists of nodes that are within a prescribed distance from an arbitrary node.

Conclusion: The resulting library works quite well. It produces the desired parameters within our targeted tolerance of efficiency.

3 Introduction

Imagine you wish to set up a network to transmit radio messages between two points, A and H. The two transmitters are far enough apart that they cannot communicate directly and need at least two repeaters to relay messages under ideal conditions. Imagine further that the area between A and H is fraught with interference and therefore reliable communication improves with each addition of a communications path. Since the goal is to set up a safe network configuration that would offer a reliable channel of communication, what would be a good network model? In the interest of minimizing cost while ensuring an acceptable tolerance of reliability, what number of relay towers and in what configuration would be best? Keeping with the scenario, what if we were able to extrapolate by some means that a particular tower was experiencing interference. Is there a way to identify which communication paths transmit through the afflicted tower and avoid using them? In this paper, we will show why a de Bruijn Network would be a good candidate model for the above situation. In Section 4, we will discuss key properties and parameters of de Bruijn Graphs. In Section 5 we will introduce a MATLAB library, *DeBruijn*, a collection of functions written for the purpose of calculating and manipulating de Bruijn graph parameters. In Section 6 we discuss functions of the *DeBruijn* library that may be used to calculate distance between nodes. In Section 7 we further explore the *DeBruijn* library by introducing many useful functions for calculating useful parameters of de Bruijn graphs. Lastly, in Section 8 we introduce *DeBruijn* library functions for generating lists of nodes both at a prescribed distance from a given node as well as functions that generate lists of nodes within a prescribed distance of a specified node.

4 De Bruijn Graph Fundamentals

Before delving into the algorithms that calculate their parameters, generate and manipulate their graphs, and exploit their physical properties to parse their structures more efficiently, we will present a cursory introduction of de Bruijn graphs. The de Bruijn graph is named after the Dutch mathematician Nicolaas Govert de Bruijn (1918-2012) who discovered that in a d–ary sequence of order n there exists a cyclic string of length d^n such that every d–ary, n–tuple appears exactly once. In graph theory, a graph is typically described by the notation $G = (V, E)$ where V is the set of vertices (or nodes for a network) and E is the set of edges (or arcs). A de Bruijn Graph is a directed (unidirectional or bidirectional) graph of the form $DG(d, n)$ where d denotes the number of characters in the string "alphabet", and n denotes a tuple size. Taken in conjunction the d-character alphabet is systematically arranged into tuples of size n to uniquely identify each node (this is where de Bruijn comes in). The number of nodes in a de Bruijn graph is calculated with these values, d^n, as well as the number of edges joining them, d^{n+1}. It should be pointed out that the d also has significance in that it represents the in-degree and out-degree of each node. Figure 1 shows the de Bruijn graph $DG(3, 1)$.

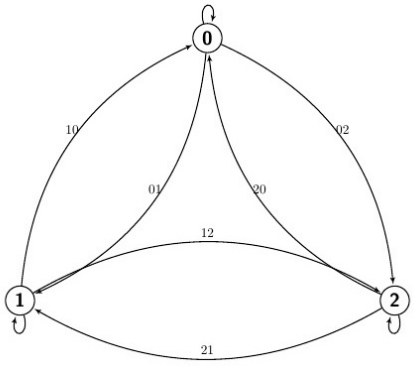


Figure 1: DG(3, 1)

Note that in this case $d = 3$, and this represents the in–degree and out–degree of the nodes. Looking at the graph, one can see that each node has three edges that leave it and enter it. Also note that $n = 1$, the string tuple or alphabet size. The string is used to label and uniquely identify the nodes. Also a quick calculation (or glance) reveals that the number of nodes is $|V| = 3^1$, and the number of edges is $|E| = 3^{1+1}$. Also notice that the values used to label the edges are not weighted values, they are rather for identification purposes. For example there are two edges joining nodes 0 and 1. Edge 01 is the edge leaving node 0 and going to node 1. Similarly, the node leaving node 1 and traveling to node 0 is labeled 10. One last point of interest from the figure is that it is a directed graph. One can tell this because the edges have arrows indicating direction at the distal end from the node of origin.

In this paper, we are using the naming convention $DG(d, n)$ for a directed graph and $UG(d, n)$ for an undirected graph.

As mentioned in the introduction, de Bruijn graphs have many properties that make them appealing as a network model. At its essence a network is a structure of objects joined by communications paths. The success of a network is how reliable these communications paths work and how efficient they work. If a network with fewer nodes and edges performs just as reliably as its more costly rival then it is a better network model. De Bruijn graphs have relatively few edges; they are considered sparse networks. Also their branching factor, or the number of children that each node points, is low yet predictable. These are considered attractive features. Interestingly de Bruijn graphs also boast numerous path alternatives. A Hamiltonian path is a traversal or route through a graph that visits every node exactly once. If the path is a cycle, meaning that it starts and ends at the same node, then it is a Hamiltonian cycle. By contrast, a Eulerian path, is a traversal that visits once every edge in a graph. De Bruijn graphs boast many Hamiltonian paths. In fact, it has been shown [1] that the number of Hamiltonian paths in a binary de Bruijn graph is $2^{2^{n-1}-1}$. The graph of $UG(2, 2)$ is illustrated in Figure 2. The yellow highlighted edges show a Hamiltonian cycle starting and ending at node 00. The red edges show a Hamiltonian path starting at node 00 and ending at 11. As predicted by the formula above there are exactly $2^{2^{2-1}-1} = 2$ Hamiltonian paths, ($\{00, 01, 10, 11\}$ and $\{00, 10, 01, 11\}$). The advantage of having many path alternatives is greater fault tolerance.

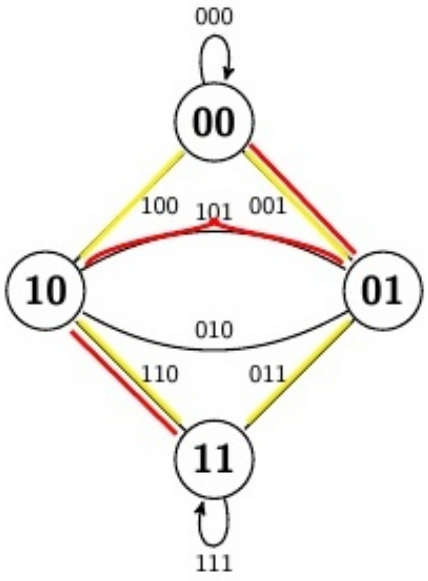
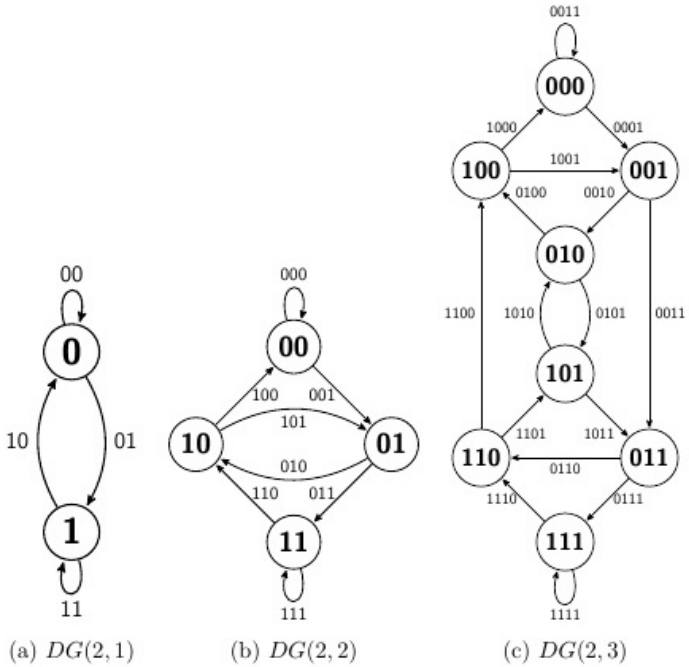


Figure 2: $UG(2, 2)$

De Bruijn graphs tend to be very symmetrical in appearance. The larger the graph, the more nodes that it contains in its structure, the more intricate and aesthetically appealing they become. The secret for this elegant appearance is found at the heart of the algorithm that makes a de Bruijn graph. De Bruijn graphs grow through a process called doubling. To double the graph $DG(d, n)$ simply add one to the string tuple size $DG(d, n+1)$. In this new, doubled, graph there are now d^{n+1} nodes and d^{n+2} edges. Figures 3 shows the first three doublings of $DG(2, n)$ and Figure 4 shows the first three doublings of $DG(3, n)$. In sub graph Fig 3 (a) $DG(2, 1)$, the graph has only two nodes, in Fig 3 (b) *DG(2, 2)* has four nodes, and in Fig 3 (c) there are eight nodes. The reader should note that it is only when doubling binary de Bruijn graphs (those graphs where $d = 2$) that the number of nodes actually "doubles" after each expansion. By contrast, when doubling de Bruijn graphs where $d = 3$ (as in Fig 4) the number of nodes triples, and when doubling those graphs where $d = 4$ the nodes quadruples. In short, after doubling a de Bruijn graph the number of nodes will be $N = d^{n+1}$ which coorelates to the number of edges in the previous graph. This fact gives us a hint about how to construct a doubled de Bruijn graph. When doubling a de Bruijn graph, remove the original nodes and replace every edge in the original graph with a new node. The new node takes on the label of the edge that it replaced. For the edges use the rule that every node has d in- and out-degree and use the left-bit shift and then toggle, that is systematically replace from $0 : d$, the right-most bit to find the current node's neighbors. For example, in Fig 3 (c) node 111 goes to two nodes because $d = 2$. By bit shifting one left and then toggling we see that the node's neighbors are 110 and itself 111. As an example note that in Fig 3 (a) graph $DG(2, 1)$ the edge joining nodes 0 and 1 is labeled 01. After doubling in Fig 3 (b) where edge 01 was in $DG(2, 1)$, it is now node 01.



(a) $DG(2, 1)$ (b) $DG(2, 2)$ (c) $DG(2, 3)$

Figure 3: Doubling $DG(2, n)$

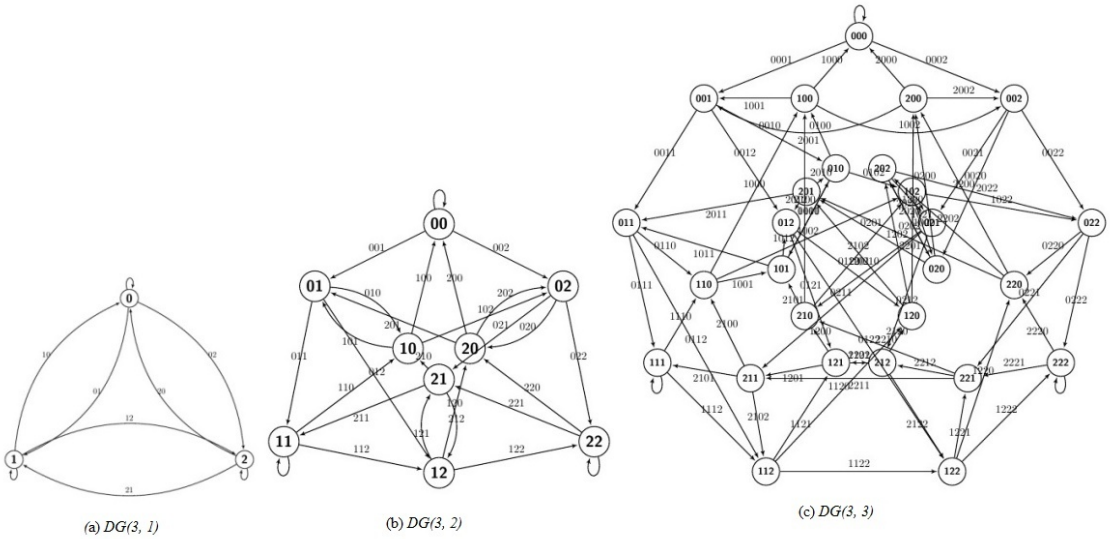


Figure 4: Doubling $DG(3, n)$

If we examine the sequence of node counts in $DG(2, n)$ we find that as the graph grows the number of nodes are $2, 4, 8, 16, ...$ likewise for the graph $DG(3, n)$ the number of nodes is $3, 9, 27, 81, ...$ It becomes evident that to calculate the number of nodes in a de Bruijn graph after doubling it, one need only use the formula $N_{k+1} = N_k d$. De Bruijn graphs are symmetric because their growth rate is a geometric series.

Distance between two nodes in a de Bruijn graph (denoted $DD(X, Y)$ where X and Y are the nodes under evaluation) is calculated by counting the number of edges that are in the shortest path between the two nodes. This is because the edge weights all have a value of one. We have already discussed that the value n represents the string tuple size in a de Bruijn graph. The value n also represents the **diameter** or largest distance between any two nodes in a de Bruijn graph. Referring back to Fig 3 (c) $DG(2, 3)$ for an example, observe that the distance between nodes 000 and 111 is 3 and that the diameter of the graph is also 3. When there are few nodes, as in $DG(2, 3)$, it is easy to just look at the graph and count the edges between nodes to figure out the distance between them, but for a graph with many nodes it becomes tedious to count. The graph of $DG(3, 3)$ above has twenty-seven nodes, and it is already a challenge to figure out the distance between 000 and 122 by counting edges to figure out which path is the shortest. Imagine trying to count the distance between nodes in $DG(4, 3)$ (Figure 5). The person charged with this task might begin to contemplate an easier way to calculate distance. Fortunately there is an easier way. Dr. Zhen Liu writes two equations [3] for calculating the distance between nodes in de Bruijn graphs. The distance between two nodes in a directed de Bruijn graph can be calculated using the following equation

$$DD(d, n) = n - l,$$

$$\text{where } l = \max\{1 \leq s \leq n, x_{n-s+1}x_{k-s+2} \ldots x_n = y_1 y_2 \ldots y_s\} \tag{1}$$

When the maximum is an empty set, then the distance is n. What this algorithm does is compare the two bit-strings for equality. First by checking the strings at full length, n,

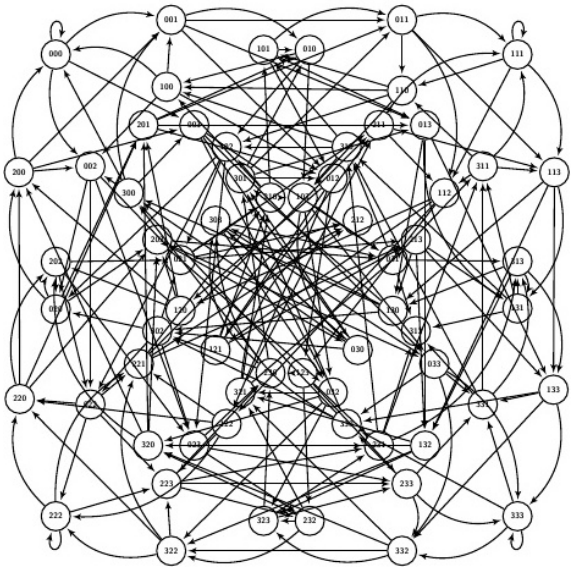


Figure 5: Doubling $DG(3, n)$

failing a match the procedure decrements the strings' size $n-1, ..., 0$ by repeatedly shifting the second string right until a substring match is found or no match is found (in which case, l is 0).

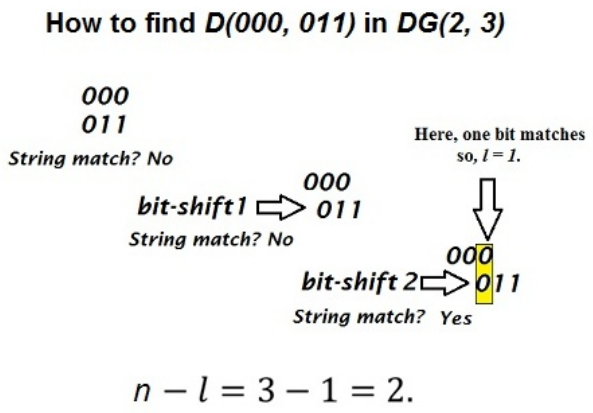


Figure 6: Finding $DD(000, 011)$ in $DG(2, 3)$

The second equation Liu gives is the distance for an undirected de Bruijn graph. This equation is a bit more involved because there are twice the number of paths to check.

$$DD(d, n) = 2n - 1 + \min\{\min_{1 \le i,j \le n}(i - j - l_{i,j}(X, Y)), \min_{1 \le i,j \le n}(-i + j - r_{i,j}(X, Y))\} \quad (2)$$

where,

$$l_{i,j}(X, Y) = \max\{s | s \le j, s \le n - i + 1, x_i x_{i+1} \dots x_{i+s-1} = y_{j-s+1} y_{j-s+2} \dots Y_j\} \quad (3)$$

$$r_{i,j}(X, Y) = \max\{s | s \le i, s \le n - j + 1, x_{i-s+1} x_{i-s+2} \dots x_i = y_j y_{j+1} \dots Y_{j+s-1}\} \quad (4)$$

That is twice the diameter minus one plus the minimum number of maximum matches in the two strings. To evaluate s, we are systematically comparing i and j, the subscripts of node strings X and Y. This brute-force comparison checks the range of all combinations that the radix of base d alphabet can assume in a string tuple of size n. As an example let us calculate $DD(010, 110)$ in $UG(2, 3)$. The example that follows is a survey of the process taken to solve the problem. Refer to Appendix 1 for a thorough treatment of this problem.

Step 1. Calculate the number of pairs of substrings that the subscripts i and j can form.

$$2n - 1 + \min\{\min_{1\le i,j\le n}(i - j - (\max\{s\})), \min_{1\le i,j\le n}(-i + j - (\max\{s\}))\}$$

Since our tuples X and Y have tuple size 3 and there are 2 of them, we have 3^2 combinations to check.

$$\{(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)\}$$

Step 2. Calculate the maximums for every i, j combination.
There will be a LHS and a RHS maximum.

$$2n - 1 + \min\{\min_{1\le i,j\le n}(i - j - (\max\{s\})), \min_{1\le i,j\le n}(-i + j - (\max\{s\}))\}$$

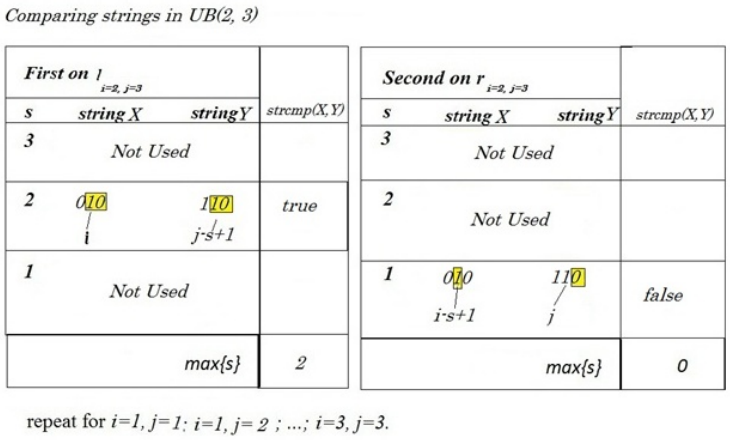


Figure 7: Comparing substrings $i = 2, j = 3$

For brevity we show an example of the comparison of substrings $i = 2, j = 3$ in Fig 6 and a list of the other eight substring maximums in Fig 7. The complete procedure is included in Appendix 1.

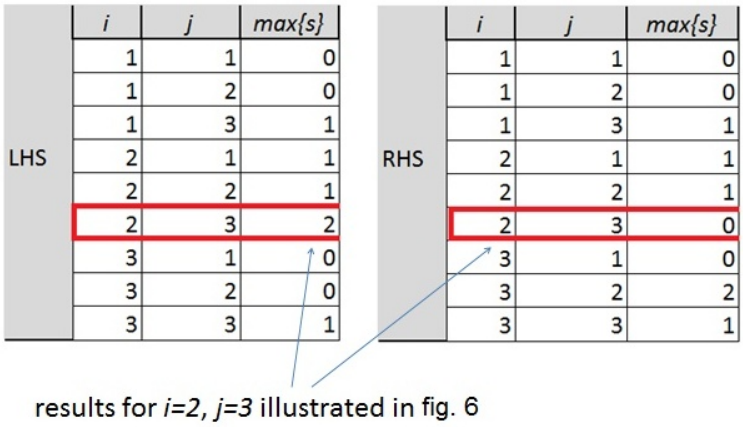


Figure 8: The computed maximums of all eight substring pairs

Step 3. Calculate the minimum i, j combination.

$$2n - 1 + \min\{\min_{1\le i,j\le n}(i - j - (\max\{s\})), \min_{1\le i,j\le n}(-i + j - (\max\{s\}))\}$$

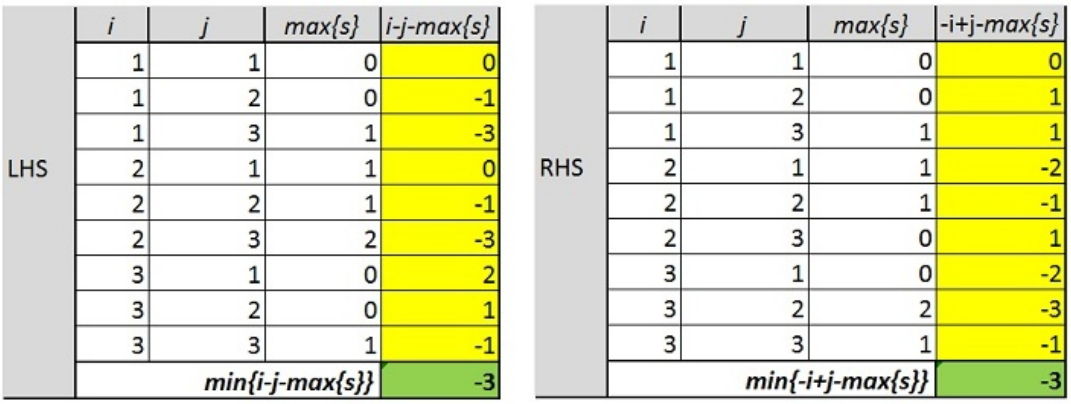


Figure 9: Finding the minimum i, j combinations

Step 4. Find the minimum of the two inner (LHS and RHS) minimums.

$$2n - l + \min\{\min_{1\le i,j\le n}(i - j - (\max\{s\})), \min_{1\le i,j\le n}(-i + j - (\max\{s\}))\}$$

And finish solving

$$(2)(3) - 1 + (-3) = 2$$

Looking at the graph we see there are two routes.

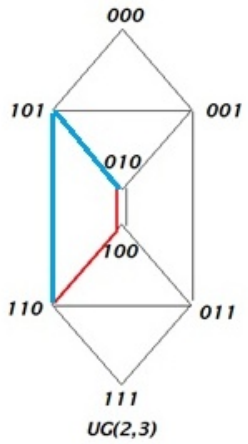


Figure 10: $UG(2,3)$ showing two routes where $UD(010,110) = 2$

These equations allow us to find the distance between nodes in the graphs without resorting to manually counting edges to find the shortest path. The equations enable us to find the distance between nodes of arbitrarily large graphs also, but there is still a hefty amount of calculation involved in the process. Would it not be nice if there were some easier way? Fortunately for us there is a much easier way: use a computer to do the work for us.

5 A Class Library to Calculate Useful Parameters of De Bruijn Graphs

Over the past fifty-five years, the electronic computer has revolutionized the way we perform calculations. Provided with an error free set of instructions, a computer can accurately and precisely solve many tedious calculations in (or very nearly so) an instant after pressing the "Enter" key—NP–Complete problems being a notable exception. It is a tool yes, but some hold that it is a prosthetic extension to our own mental capacity enabling us to achieve plateaus of computational competency dwarfing all previous ages of civilization. Consider for example the pursuit of π. The significance of the ratio of the circumference of a circle to its diameter was known by ancient Babylonians four thousand years ago, yet up until 1945 the decimal expansion of π was known to a mere 808 places. This monumental effort of human computation took mathematician D. F. Furguson nearly a year to calculate[4]. In the following year, John von Neumann used the ENIAC to calculate π to 2, 037 digits in just 70 hours. Since that time, with the aid of computers, that expansion has grown to the twelve–trillionth digit. The computer has proven to be a very handy tool indeed. So when faced with a problem like finding the distance between nodes in a graph where manually counting or calculating them grows too laborious; then it is time to automate the process.

There are many programming environments that one could choose from to implement an algorithm like one to calculate Liu's equation. One of which is MATLAB. Like a functional programming language, MATLAB was expressly designed to perform

mathematical functions. Its syntax and grammar rules facilitate function design without the overhead of imperative or spoken type languages—although MATLAB does port well to most of these languages. In fact, many MATLAB commands are directly derived from conventional mathematical symbols or keyboard adaptations (i.e., e^2 and max(A, B) are legitimate commands in MATLAB). What is more, the scripts that constitute code in MATLAB are compact programs in document type ".m" format that may be written using any text editor. Using MATLAB, we have written a library, named DeBruijn, that implements many useful functions to calculation de Bruijn graph parameters. Using tailored functions, such as those that comprise the deBruijn library, greatly expedite the process of new code development through code reuse and inheritance.

6 Functions to Calculate Distance Between Nodes

One of the first functions from the DeBruijn library is the function to implement Liu's distance formula. There are actually two functions, one for finding the distance between nodes in a directed de Bruijn graph and another for distances in an undirected graph. The code that follows implements the directed case. The code stays true to the technique described above of performing string comparisons and bit-shifting strings.

```
1  %*************************************************************************
2  % FUNCTION: DD(X,Y)
3  % LIBRARY: DeBruijn
4  % DISCRIPTION: DD finds the distance between two nodes in a directed de Bruijn Graph.
5  % INPUTS: 'X', a string representing a node in a de Bruijn Graph;
6  %          'Y', a second string also representing a node in the same de Bruijn Graph.
7  %          NOTE: Make sure inputs are in single quotes,'', to signify char
8  %          array.
9  % OUTPUTS: Returns an integer value representing the distance (number of edges between X,Y).
10 % AUTHOR/DATE: JMOORE, 7/14/15
11 %*************************************************************************
12 function distance = DD(X,Y)
13 n=length(X);
14 s=n;
15 flag=0;
16 while(flag==0)
17     if(strcmp(X,Y)==1)
18         flag=1;
19     else
20         if(s>0)
21             X(1)='';
22             Y(s)='';
23         end %if(s>0)
24         s=s-1;
25         if(s==0)
26             flag=1;
27         end %if(s==0)
28     end %if(strcmp(X,Y)==1)
29 end %while
30 distance=n-s;
31 end %function distance = DD(X,Y)
```

Input parameters X and Y (*lines 5 and 6*) correspond to the string tuples used to represent nodes in the de Bruijn graph. *Line 13* loads the string tuple length into variable n, an integer. s is another integer taking on the value of n and is later manipulated through decrementing. The *flag* value initialized to 0 is used to guard when it is time to exit the

body of the while loop. The while loop systematically compares the two strings (*line 28*) testing for equality, first at length s then, failing a match, at length $s-1$. When the strings are found not to be equal, they are each truncated, X (in *line 21*) gets cut at the tail end, and Y (in *line 22*) get the head cut off. The loop continues in this way until either a match is found or no match is found, when $s = 0$ triggering *flag* to equal 1, the exit criteria. After exiting the loop (at *line 29*), *line 30* assigns the value *distance* to be '$k - s$' which is the return value. Now let us have a little fun! Earlier when discussing techniques for manually calculating the distance between nodes, we noted how tedious it would be to attempt to visually count or manually calculate the distance between two nodes in the de Bruijn graph $DG(4, 3)$, a graph which boast 64 vertices and 256 arcs. Let us execute this challenge by running *DD(0010, 1101)* in MATLAB's command shell. The value "ans = 4" is returned before one can fully depress the "ENTER" key. Interestingly, running *UD(0100, 1101)*—our next function—returns a "3", but there is a perceivably longer wait time (perhaps enough time to let go of the "ENTER" key). What could account for this extra processing time? One important issue when executing functions that calculate values where the number of processes could become arbitrarily large is to consider how much time and how much memory the calculation will require. As mentioned, de Bruijn graph growth is modeled after a geometric series, and since the number of nodes is taken to be greater than one, the series does not converge but could become very large, countably infinite. The time effiiciency of DD(X, Y).m is linear, $O(n)$, and so it is fairly effiicient.

The following function calculates the distance between nodes in an undirected de Bruijn graph. Note how this function calls two helper functions.

```
1  %*********************************************************************
2  % FUNCTION: UD(X,Y)
3  % LIBRARY: DeBruijn
4  % DISCRIPTION: For de Bruijn Graphs in an undirected case,
5  % this function calculates the distance between two nodes.
6  % INPUT: X,Y the two nodes strings to assess.
7  % OUTPUT: distance, a scaler value representing the distance between X,Y.
8  % AUTHOR/DATE: VHORAN, 07/01/14
9  %*********************************************************************
10 function distance = UD(X, Y)
11 n=length(X);
12 M=zeros(n,n);
13 for i=1:n
14    for j=1:n
15      M(i,j)=i-j-max(LHS(i,j,X,Y),RHS(j,i,X,Y));
16    end
17 end
18 distance=2*n-1+min(min(M),[],2);
19 end
```

As in the previous example, the variable n gets assigned the string length of X. In *line 12* the value M is an $n \times n$ array initialized with zeros. The nested *for* loops that follow re-populate M using a part of Liu's equation (2). The two helper functions *LHS()* and *RHS()* (which will be presented very soon) are used to iterate through the strings X and Y using indicies i and j. On *line 18*, after exiting the loops, the value *distance* is calculated using yet another part of Liu's equation (2) and return. The time efficiency of *UD(X, Y)* is cubic, $O(n^3)$, since it must iterate through a pair of nested *for* loops and within the inner loop must visit *LHS().m* and *RHS().m* which each have a *while* loop. However, since *LHS().m* and *RHS().m* are visited in series, their cost are linear. As an illustration of how time efficiency affects processing time, let us contrast the number of processes required to

execute $DG(2,10)$ and $UG(2,10)$. At linear time, $DG(2,10)$ performs $2^{10} = 1,024$ processes. Meanwhile, at cubic time, $UG(2,3)$ must perform $(1024)^3 = 1,073,741,824$ processes. If we had to store one byte of information per process, it would require just over a gigabyte in memory for the undirected case. Now let us examine the helper functions called by $UG(d,n)$.

```
1  %*******************************************************
2  % FUNCTION: LHS(i,j,X,Y)
3  % LIBRARY: DeBruijn
4  % DISCRIPTION: This function is used with function UD(X, Y).m
5  % INPUT: i, iterator; j, iterator; X, first string; Y, second string
6  % OUTPUT: sMax the maximum value of s.
7  % AUTHOR/DATE: VHORAN, 7/01/14
8  %*******************************************************
9  function sMax=LHS(i,j,X,Y)
10 n=length(X);
11 sMax=0;
12 s=0;
13 while (s<=j)&&(s<=n-i+1)
14    if(strcmp(X(i:i+s-1),Y(j-s+1:j)))
15       sMax=s;
16    end
17    s=s+1;
18 end
19 end
```

In *LHS().m*, the value of *sMax* is initialized to 0, but upon exiting the function it will hold the value of the largest string comparison of the strings X and Y that exist when searching the left hand side edge branching to look for the shortest path from X to Y. The value of s also initialized to 0 serves double duty. Firstly, it is a loop guard, the exit criteria to leave the loop is listed in the logical *and* comparison associated with the *while* statement. The second purpose of s is to relay to *sMax* through assignment in *line 15* the loop iteration count where a match was found in *line 14*. Since s is incrementing *(line 17)* every iteration of the loop, its value at assignment is guaranteed to be the maximum yet found, and so upon exiting the loop the maximum string comparison that exist. Leaving now the first helper function, the second helper function *RHS().m* is very similar.

```
1  %*******************************************************
2  % FUNCTION: RHS(i,j,X,Y)
3  % LIBRARY: DeBruijn
4  % DISCRIPTION: This function is used in conjunction with function
5  % INPUT: i, iterator; j, iterator; X, first string; Y, second string
6  % OUTPUT: sMax the maximum value of s for the right hand side.
7  % AUTHOR/DATE: VHORAN, 7/01/14
8  %*******************************************************
9  function sMax=RHS(i,j,X,Y)
10 n=length(X);
11 sMax=0;
12 s=0;
13 while (s<=i)&&(s<=n-j+1)
14    if(strcmp(X(i-s+1:i),Y(j:j+s-1)))
15       sMax=s;
16    end
17    s=s+1;
18 end
19 end
```

In fact, the only difference is that the subscripts follow the right hand side branching edges. For the sake of brevity, we omit the redundant code trace, leaving it for the reader's discretion for indulgence.

7 Generating Other Useful Parameters

In this section we present some DeBruijn library functions that return many useful parameters. Although the following functions are useful enough to warrant their

inclusion in this project report, they do not, taken individually, justify their own dedicated section. Therefore, we opted to present them all in this section. The first of these useful functions that we present is one called *GenerateNodes(d, n)*. This function returns the string representation in the desired base–of–radix.

```
1  %*********************************************************************
2  % FUNCTION: GenerateNodes(d,n)
3  % LIBRARY: DeBruijn
4  % DISCRIPTION: This method generates node names given input d, k.
5  % Node names (the string designation of every node) are in appropriate
6  % base of radix and of correct string length for their respective
7  % de Bruijn Graphs.
8  % INPUT: d, the alphabet size; n, the string tuple size.
9  % OUTPUT: The node names.
10 % AUTHOR/DATE: JMOORE, 07/15/14
11 %*********************************************************************
12 function nodes=GenerateNodes(d,n)
13 N=d^n; %number of nodes in the graph
14 nodes =cell(1,n);
15 for i=1:N %each row in matrix
16     nodes{i}=dec2base(i-1,d,n);
17 end %for i=1:N
18 end %function nodes=generagteNodes(d,n)
```

Notice that in *line 16* the MATLAB *string library* function *dec2base(i-1, d, n)* is called. This is where the true work gets done in this function. This method converts the value $i-1$ from a decimal to one of base n. As an example we executed *GenerateNodes(3, 2)* in MATLAB, and the output was.

$$\text{ans } = \{00, 01, 02, 10, 11, 12, 20, 21, 22\}$$

An alternative method for representing a graph, rather than in the vertex and arc graph that has been used up until now in this paper, is to use an N–by–N matrix, where N is the number of vertices in the graph. For example an *adjacency matrix* for a graph is used to represent which vertices in a graph are adjacent to which other vertices. The code to produce an adjacency matrix for $DG(d, n)$ follows.

```
1  %************************************************************
2  %   FUNCTION: AdjacencyMat
3  %   LIBRARY: DeBruijn
4  %   DISCRIPTION: AdjacencyMat produces the adjacency matrix
5  %   for a de bruijn graph without self-pointing node edge values counted.
6  %   INPUTS:  d = alphabet size, n = string length
7  %   OUTPUTS:  A = adjacency matrix for directed B(d,n)
8  %   NOTE: Output Modification: this function has been modified
9  %   to return the adjacency matrix without self-pointing edge weights
10 %   AUTHOR/DATE: JMOORE, 7/15/14
11 %************************************************************
12 function A = AdjacencyMat(d, n)
13 N=d^n; %number of nodes in the graph
14 for count=1:N
15 A = zeros(N); %adjacency matrix
16 for i=1:N %each row in matrix
17     x = dec2base(i-1,d,n); %convert row name to string
18     for j=0:(d-1)
19         y=strcat(x(2:n),num2str(j)); %add new letter to end
20         z=base2dec(y,d); %convert new string to decimal
21         A(i,z+1)=1; %1 in A means edge from i to z+1
22     end %for j
23 end %for i
24 %this modification removes edge weight from the self-pointing nodes
25 for i=1:N
26     for j=1:N
27         if(j==i)
28             A(i,j)=0;
29         end %if
30     end %j
31 end %i
32 end %count
33 end %end fun
```

There are a few more remarkable MATLAB *string library* functions used in *AdjacencyMat(d, n)*. In *line 17* we see the function *dec2base(decimal, n, base)* again. In this case, the function converts decimal strings n to a base system specified in the third parameter. Function *base2dec()* in *line 20* performs the opposite operation. The function *num2string(j)* converts a number j into a string. The function *strcat(x, (2:n))* is used to concatenate string x removing string positions 2 onward to the tail. Essentially what this function does is revealed in *line 21* where matrix A gets re–populated when adjacent nodes are found. That is, it visits every row i placing a 1 at every column positioned $z + 1$ (N, the number of nodes minus d, the degree) distance apart. *Lines 24–32* were added to the function to exclude self–pointing nodes. When this method is executed using the following command (as an example) in MATLAB "AdjacencyMat(2, 4)", the following adjacency matrix is returned.

Table 1: Adjacency Matrix for $DG(2, 4)$

XXXX	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0001	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0010	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0
0011	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0
0100	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0
0101	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0
0110	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0
0111	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
1000	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1001	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
1010	0	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0
1011	0	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0
1100	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0
1101	0	1	0	0	0	0	0	0	0	0	1	1	0	0	0	0
1110	0	1	0	0	0	0	0	0	0	0	0	0	1	1	0	0
1111	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0

To read the MATLAB output for *AdjacencyMat(2, 4)* (in Table 1 above), select a vertex string identifier (name) from the row label (highlighted blue) and then scan across the row associated with the vertex. If a 1 appears in the row then the vertex listed in the corresponding column label (also highlighted blue) is adjacent to the vertex under consideration. By contrast, a 0 in the row indicates that the vertex in the column label is not adjacent to the vertex under consideration. From the standpoint of efficiency, *AdjacencyMat(d, n)* is somewhat wasteful. The structure of the nested *for* loops alone cause the function to go cubic. In addition to this, the MATLAB functions that are called within *AdjacencyMat()*, *strcat(), num2str(), and base2dec()*, undoubtedly come at a cost as well.

Another matrix that can be used to represent a de Bruijn graph is a distance matrix. Unlike an adjacency matrix, a distance matrix shows the distance from every node to every other node in the graph. In the DeBruijn library there is a method to generate a directed distance matrix. The code follows.

```
1 % ********************************************************************
2 %FUNCTION: DirectedDistanceMat(d,n)
3 % LIBRARY: DeBruijn
4 % DISCRIPTION: This method generates a distance matrix for a directed
5 % de Bruijn Graph.
6 % INPUTS:  d = alphabet size, n = string length
7 % OUTPUTS:  D = matrix of distances from all nodes to all nodes
8 % AUTHOR/DATE: JMOORE, 7/14/15
9 %********************************************************************
10 function D=DirectedDistanceMat(d,n)
11 N=d^n; %number of nodes in the graph
12 nodes=zeros(N);
13 for i=1:N %each row in matrix
14     for j=1:N %each col in matrix
15         nodes(i,j)=DD((dec2base(i-1,d,n)),(dec2base(j-1,d,n)));
16     end %for j=1:N
17 end %for i=1:N
18 D=nodes;
19 end %function D=DirectedDistanceMat(d,n)
```

As you can see the function calls our library function *DD(d, n).m* in *line 15*. It does this using a rather unusual approach by making the parameters $d =$ *dec2base*$(i-1, d, n)$ and $n =$*dec2base*$(i-1, d, n)$. In doing it this way, we are taking the loop counting indices i, for the row, and j, for the column, and converting them to strings of size n with an alphabet of size d. This has the effect of generating the string node names from their location in the matrix. Once the node names are generated, *DD(d, n).m* computes the distance and populates the *nodes* array. After exiting the outer *for* loop, *nodes* is returned. The cost of *DirectedDistanceMat(d, n).m* is at least quadratic, $O(n^2)$, due to the nested *for* loops, and this cost does not take into account the called library functions *dec2base()*. Since *DD(d, n).m* is linear, it does not have much of an effect on efficiency. We executed *DirectedDistance(2, 4).m* using MATLAB which returned the following distance matrix in Table 2.

Table 2: Distance Matrix for $DG(2, 4)$

XXXX	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	0	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2
0001	2	0	2	2	1	1	1	1	2	2	2	2	2	2	2	2
0010	2	2	0	2	2	2	2	2	1	1	1	1	2	2	2	2
0011	2	2	2	0	2	2	2	2	2	2	2	2	1	1	1	1
0100	1	1	1	1	0	2	2	2	2	2	2	2	2	2	2	2
0101	2	2	2	2	1	0	1	1	2	2	2	2	2	2	2	2
0110	2	2	2	2	2	2	0	2	1	1	1	1	2	2	2	2
0111	2	2	2	2	2	2	2	0	2	2	2	2	1	1	1	1
1000	1	1	1	1	2	2	2	2	0	2	2	2	2	2	2	2
1001	2	2	2	2	1	1	1	1	2	0	2	2	2	2	2	2
1010	2	2	2	2	2	2	2	2	1	1	0	1	2	2	2	2
1011	2	2	2	2	2	2	2	2	2	2	2	0	1	1	1	1
1100	1	1	1	1	2	2	2	2	2	2	2	2	0	2	2	2
1101	2	2	2	2	1	1	1	1	2	2	2	2	2	0	2	2
1110	2	2	2	2	2	2	2	2	1	1	1	1	2	2	0	2
1111	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	0

The undirected counterpart for *DirectedDistanceMat(d, n)* in the *DeBruijn* library is *UndirectedDistanceMat(d, n).m*.

```
%*****************************************************************
%  FUNCTION: UndirectedDistanceMat(d,n)
%  LIBRARY: DeBruijn
%  DISCRIPTION: This method generates a distance matrix for an undirected
%  de Bruijn Graph.
%  INPUT:  d = alphabet size, n = string length
%  OUTPUT:  D = matrix of distances from all nodes to all nodes
%  AUTHOR/DATE: JMOORE, 07/15/14
%*****************************************************************
function U=UndirectedDistanceMat(d,n)
N=d^n; %number of nodes in the graph
nodes=zeros(N);
for i=1:N %each row in matrix
    for j=1:N %each col in matrix
        nodes(i,j)=UD((dec2base(i-1,d,n)),(dec2base(j-1,d,n)));
    end %for j=1:N
end %for i=1:N
U=nodes;
end %function U=UndirectedDistanceMat(d,n)
```

About the only thing worth mentioning for this function is that the time complexity and efficiency are very poor. This function calls *UD(d, n).m* which runs at $O(n^3)$ efficiency. Exacerbating an already inefficient function, *UndirectedDistanceMat()* itself runs in $O(N^2)$ efficiency. This brings our efficiency down to $(O(n^3) \times O(N^2)) = O(N^3 d^{2n})$. The distance matrix representing the output for *UndirectedDistanceMat(2, 4)* is in Table 3.

Table 3: Distance Matrix for $UG(2, 4)$

XXXX	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0000	0	1	2	2	2	3	3	3	1	3	3	4	2	4	3	4
0001	1	0	1	1	2	2	2	2	1	2	3	3	2	3	3	3
0010	2	1	0	2	1	1	3	3	2	1	2	2	2	3	3	4
0011	2	1	2	0	2	3	1	1	2	1	3	2	2	2	2	2
0100	2	2	1	2	0	2	3	3	1	1	1	3	2	2	3	4
0101	3	2	1	3	2	0	2	2	3	2	1	1	3	2	3	3
0110	3	2	3	1	3	2	0	2	2	2	2	1	1	1	2	3
0111	3	2	3	1	3	2	2	0	3	2	3	1	2	2	1	1
1000	1	1	2	2	1	3	2	3	0	2	2	3	1	3	2	3
1001	3	2	1	1	1	2	2	2	2	0	2	3	1	3	2	3
1010	3	3	2	3	1	1	2	3	2	2	0	2	3	1	2	3
1011	4	3	2	2	3	1	1	1	3	3	2	0	2	1	2	2
1100	2	2	2	2	2	3	1	2	1	1	3	2	0	2	1	2
1101	4	3	3	2	2	2	1	2	3	3	1	1	2	0	1	2
1110	3	3	3	2	3	3	2	1	2	2	2	2	1	1	0	1
1111	4	3	4	2	4	3	3	1	3	3	3	2	2	2	1	0

As mentioned above, the reason *AdjacencyMat(d, n).m* (and the distance matrices) is so inefficient is because of its nested loop structure. Looking at the returned matrix in *Table 1*, one of the first things we notice is that the array is nearly filled with zeros. This is because de Bruijn graphs are sparsely populated with relatively few nodes when compared to their edges. The adjacency matrix goes through an awful lot of work to produce a relatively small amount of useful information. This next function, *VectorNeighborGenerator(d, n).m* returns the same useful information without the superfluous generation and storage of useless data.

```
%*********************************************************************
% FUNCTION: VectorNeighborGenerator(d,n)
% LIBRARY: DeBruijn
% DISCRIPTION: Class generates a list (one-dimentional matrix) with
% size=(d^(n+1)-d), (number of nodes)*(the out-degree)-(less self-pointers).
% INPUT: d, the string alphabet size; n, the string tuple size;
% OUTPUT: A One-dimentional vector where each indexed value contains
% a string pair (separated by ':') and consisting of all edge distances
% that equal one.  The string pairs coorespond to 'out-neighbor:in-neighbor'
% node pairs with a single edge between them.  The output is comparable to
% the adjacencyMat(d,n) in that all neighbors are returned.
% AUTHOR/DATE: JMOORE, 07/15/14
%*********************************************************************
function E=VectorNeighborGenerator(d,n)
N=d^n; %|V|
size=N*d-d;
nodes=GenerateNodes(d,n); %calls external method, populates 1-d vector the de Bruijn Graph node strings
    (note: O(n))
edges=cell(1,size); %at output, will contain all neighbor nodes at distance=1.
nNeighbor=0;
count=1; %works as an index for edges
currentNode=1; %along with nNeighbor works as indicies for nodes cell array.
for i=1:size+d %size + d is minimum number of iterations found so far
    nNeighbor=nNeighbor+1;
    if(nNeighbor>N) %rolls over nNeighbor counter once it reaches N
        nNeighbor=1;
    end %if(nNeighbor>N)
    if(~strcmp(nodes{currentNode},nodes{nNeighbor})) %No self-pointing nodes
        edges{count-1}=strcat(nodes{currentNode},char(':'),nodes{nNeighbor}); %assignment neighbors to
            edges array
    else
        if(count~=1) %skip the first one
            count=count-1;
        end %if(count~=1)
    end %if(~strcmp(nodes{currentNode},nodes{nNeighbor}))
    if((mod(i,d)==0)&&(nNeighbor~=0)) %increment to next node after d iterations
        currentNode=currentNode+1;
    end %if((mod(i,d)==0)&&(nNeighbor~=0))
    count=count+1;
end %for 1:size+d
E=edges;
end %fun
```

Running this method using the same parameters as used for *AdjacencyMat(2, 4)* in the MATLAB command shell, we receive the following smaller data structure in return.

$$\text{VectorNeighborGenerator}(2,4)$$

$$\begin{aligned}\text{ans } = \{&0000:0001, 0001:0010, 0001:0011, 0010:0100, 0010:0101, 0011:0110,\\ &0011:0111, 0100:1000, 0100:1001, 0101:1010, 0101:1011, 0110:1010, 1100:1101,\\ &0111:1110, 0111:1111, 1000:0000, 1000:0001, 1001:0010, 1001:0011, 1010:0100,\\ &1010:0101, 1011:0110, 011:0111, 1100:1000, 1100:1001, 1101:1010, 1101:1011,\\ &1110:1100, 1110:1101, 1111:1110\}\end{aligned}$$

This function is an improvement over *AdjacencyMat(d, n)* because it runs in linear time complexity (omitting any costs attributed from the called function *strcmp()*). Another improvement is that it does not return any useless values, and therefore instead of returning d^{2n} values it only returns $d^{n+1} - d$ values. In the case of *DG(2, 4)* above it

returned 30 values instead of 256. This is an 88.3% improvement in efficiency. Tracing the code, note that the variable *size* is instantiated to be $N * d - d$. This is the length of the return structure, a linear *cell array* called *edges*. This brings us to our first assertion regarding the number of adjacent nodes in a directed de Bruijn graph:

In every directed de Bruijn graph, the number of nodes that are adjacent to one another will always be N, the number of nodes, multiplied by d, the number of edges leaving these nodes, less d, the number of self–pointing nodes in the graph.

Also note that in *line 27 VectorNeighborGenerator(d, n)* makes use of our library function *GenerateNodes(d, n).m*. It does this so that it may perform a string comparison rather than a nested loop search approach. The function uses a couple of variables *currentNode* to represent the current node under consideration and *nNeighbor*. The variable *nNeighbor* gets assigned by following a programmed route of nodes that are within the *currentNode's* "reach" when calculating its *d* –length *hops*. These *hops* follow a predictable path as illustrated below in figure 10.

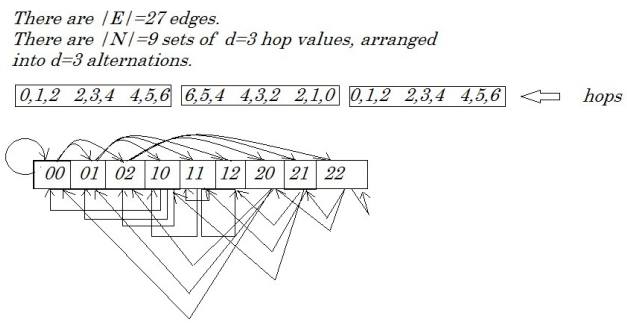


Figure 11: *Hops* in *DG(3, 2)*

This leads us to our second assertion:

If organized in numeric order, the edges in a directed de Bruijn graph can be grouped into N sets of d values. These values, called hops, represent the sum of the nodes hopped–over by an edge from the current node under consideration to the node its out–going edge points. What is more, these sets of hops are further divided into d alternations, where the hop direction changes. Taken together, this behavior makes a predictable pattern.

The significance of this observation is that since we can predict where node neighbors are located, we need not blindly search through an entire graph to find them.

Using *currentNode* and *nNeighbor* in tandem, the linear array *nodes* is traversed, *currentNode* simply iterating in sequence while *nNeighbor* follows its *hopping* route. In *line 37* a negated comparison is made to insure that they are not equal, indicating a self–pointing node, and if this comparison fails then the two nodes are assigned together as

one slot in the *edges* cell array separated by a colon (assignment done in *line 38*). After the loop counting variable reaches $i =$ size $+ d$ (in MATLAB array indices always start at 1) we exit the loop and return *edges*, the populated cell array.

8 Generating Lists of Nodes *at* and *within* Prescribed Distance

The last set of functions involve the generation of two very similar yet distinctly different lists of nodes. The first is a function to generate a list of all nodes **within** a prescribed distance from a given node. The DeBruijn library function for generating a set of all nodes contained in a *ball–of–radius D* follows.

```
1  %************************************************************
2  % FUNCTION: DirectedBallDFromX(D,X,d,n)
3  % LIBRARY: DeBruijn
4  % DISCRIPTION: This function returns a list of all nodes w/in D from X.
5  % INPUT:     D, the distance to check;
6  %            X, the node to measure distance from;
7  %            d, alphabet size;
8  %            n, string tuple size.
9  % OUTPUT: A cell array, 'nodes', the list of all nodes within D from X.
10 % AUTHOR/DATE: JMOORE/7/14/15
11 %************************************************************
12 function A = DirectedBallDFromX(D,X,d,n)
13 N=d^n; %|A|
14 j=1; %a counter for list nodes indexing
15 x=(base2dec(X,d)+1);
16 nodes=cell(1);
17 B=DirectedDistanceMat(d,n); %generate the distance array.
18 for i=1:N
19     if(B(x,i)<=D)
20           nodes{j}=dec2base(i-1,d,n);
21           j=j+1;
22     end %if(B(1,i)<=D)
23 end %for i=1:N
24 A=nodes;
25 end %function A = DirectedBallDFromX(D,X,d,n)
```

For example this function using a quintuple of parameters and produces the following output.

$$\textit{DirectedBallDFromX(1, 01, 3, 2) ans} = \{01, 10, 11, 12\}$$

This function calls *DirectedDistanceMat(d, n).m* after which it parses the matrix looking for nodes within the specified distance, $D = 1$ from a specified node *X=01* (*line 19*). When found, these nodes within D are converted to the desired base and placed into the *cell array, nodes* for return. The cost of *DirectedBallDFromX().m* is quadratic with respect to N because *DirectedDistanceMat().m* is called within it.

The undirected counterpart function to *DirectedBallDFromX(D, X, d, n).m* is called *UndirectedBallDFromX(D, X, d, n).m*. This function generates a list of nodes that are within a specified distance from a specified node also, but the list returned has twice as many nodes in it since the graph is undirected. This function also calls its appropriate distance matrix, *UndirectedDistanceMat(d, n).m*, to assess the distances between nodes.

```
%*********************************************************
%  FUNCTION: UndirectedBallDFromX(D, X, d,n)
%  LIBRARY: DeBruijn
%  DISCRIPTION: This function calculates all nodes at a ball of distance
%   for an undirected de Bruijn Graph.
%  INPUT:    D, the distance to check;
%            X, the node to measure distance from;
%            d, alphabet size;
%            n, string tuple size.
%  OUTPUT: Returns a list of all nodes w/in D from X.
%  AUTHOR/DATE: JMOORE, 07/15/14
%*********************************************************
function A = UndirectedBallDFromX(D,X,d,n)
N=d^n; %|A|
j=1; %a counter for list nodes indexing
x=(base2dec(X,d)+1);
nodes=cell(1);
B=UndirectedDistanceMat(d,n); %generate the distance array.
for i=1:N
    if(B(x,i)<=D)
          nodes{j}=dec2base(i-1,d,n);
          j=j+1;
    end %if(B(1,i)<=D)
end %for i=1:N
A=nodes;
end %function A = UndirectedBallDFromX(D,X,d,n)
```

As expected *UndirectedBallDFromX(X, D, d, n).m* requires $O(N^3N^2)$ time efficiency because it calls *UndirectedDistanceMat(d, n)* that runs in $O(N^2)$ and that function calls *UD(d, n)* that runs in $O(n^3)$ time efficiency. Although *UndirectedDistanceMat(d, n)* and *UD(d, n)* work well, they are not at all efficient.

Let us examine an application of an undirected de Bruijn graph where the function *UndirectedBallDFromX(X, D, d, n).m* is utilized. Returning to our radio network scenario that was discussed in the Introduction, imagine a radio network modeled on the undirected de Bruijn graph *UG(2, 4)* where the repeater tower, represented by node *1100* is experiencing interference. In order for our transmiter, node *0000*, to reach our receiver, node *1111*, we must find a repeater tower path that avoids the defective tower. Executing the function *UndirectedBallDFromX(1100, 1, 2, 4)* returns the list $\{0110, 1000, 1001, 1100, 1110\}$. By avoiding transmission through these towers, a fault free route can be determined. The shortest path is highlighted in green and it follows nodes $\{0001, 0011, 0111, 1111\}$ (see Figure 11).

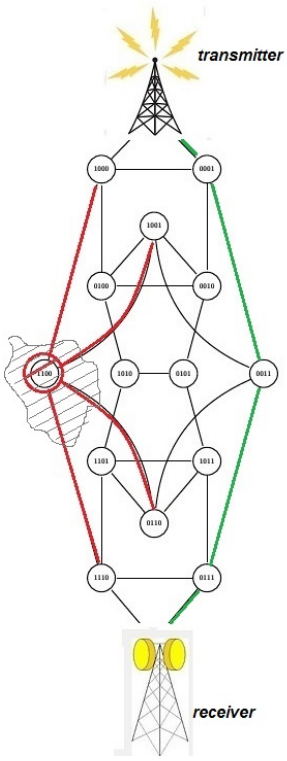


Figure 12: Radio network *DG(2, 4)* showing avoidance of fault paths

The function *DirectedAtDFromX(D, 'X', d, n).m* is similar to *DirectedDFromX()* except that instead of locating all nodes **within** a given distance, the function locates all nodes **at** the designated distance.

```
%************************************************************
% FUNCTION: DirectedAtDFromX()
% LIBRARY: DeBruijn
% DISCRIPTION: %This function returns a list of all nodes at D from X.
% INPUT:     D, the distance to check;
%            X, the node to measure distance from;
%            d, alphabet size;
%            n, string tuple size.
% OUTPUT: A cell array called 'nodes', the list of all nodes at D from X.
% AUTHOR/DATE: JMOORE, 7/14/15
%************************************************************
function A = DirectedAtDFromX(D,X,d,n)
N=d^n; %|A|
j=1; %a counter for list nodes indexing
x=(base2dec(X,d)+1);
nodes=cell(1);
B=DirectedDistanceMat(d,n); %generate the distance array.
for i=1:N
     if(B(x,i)==D)
           nodes{j}=dec2base(i-1,d,n);
           j=j+1;
     end %if(B(1,i)<=D)
end %for i=1:N
A=nodes;
end %function A = DirectedAtDFromX(D,X,d,n)
```

For example this function using a quintuple of parameters and produces the following output.

DirectedAtlDFromX(1, 01, 3, 2) $ans = \{10, 11, 12\}$

This function calls *DirectedDistanceMat(d, n).m* after which it parses the matrix looking for nodes **at** the specified distance, $D = 1$ from a specified node *X=01* (*line 19*). When found, these nodes at D are converted to the desired base and placed into the *cell array, nodes* for return. Like *DirectedBallDFromX()*, the cost of *DirectedAtDFromX().m* is, quadratic time efficiency.

The undirected counterpart function to *DirectedAtDFromX(D, X, d, n).m* is called *UndirectedAtDFromX(D, X, d, n).m*. This function generates a list of nodes that are at a specified distance from a specified node also, but the list returned has more nodes in it since the graph is undirected. This function also calls its appropriate distance matrix, *UndirectedDistanceMat(d, n).m*, to assess the distances between nodes.

Returning to our radio tower scenario, where the tower configuration is modeled after the de Bruijn network *UG(2, 4)*. This time, imagine that the transmitter system underwent an upgrade and so now it is capable of propogating signals at much higher power. Our function *UndirectedAtDFromX(D, X, d, n)* could be utilized to determine broadcast range from the transmitter tower, node 0000. Let us say we broadcast a signal at four discrete and incrementally higher power levels, 1 - 4. After each transmission, we pause and await confirmation of receipt from the other towers in our network. After our first broadcast, at power level 1, we receive notification from towers 0001, and 1000. After our broadcast at power level 2 we receive confirmation from $0010, 0011, 0100, 1100$ as well as those who previously acknowledged. On power level number 3 we receive notification from towers $0101, 0110, 0111, 1001, 1010,$ and $, 1110$ as well as all those who have previously acknowledged. Finally we broadcast at power level 4, and we receive acknowledgement from towers $1011, 1101,$ and 1111 in addition to all other towers in the network (Refer to Figure 13).

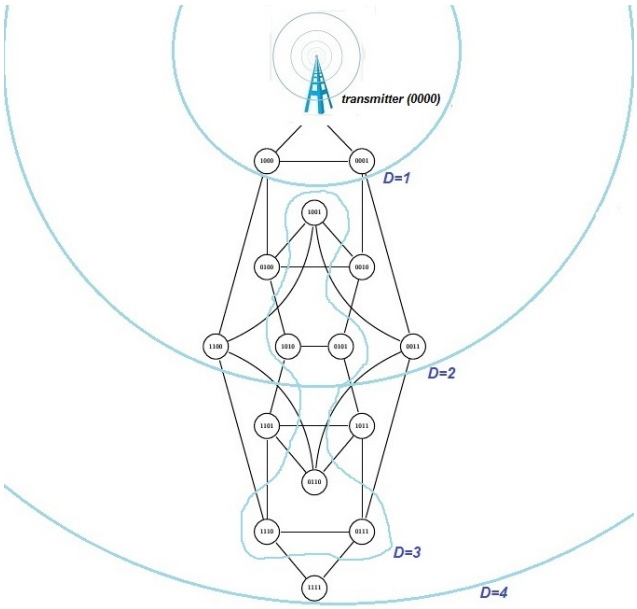


Figure 13: Radio Network *DG(2, 4)* showing Broadcast Distance

9 Conclusion

De Bruijn Graphs are a fascinating topic relevant to many fields including computer science, graph theory, and network engineering. In this project report, we discussed the DeBruijn library of MATLAB function used to manipulate these graphs and generate many useful parameters that are used in turn to describe them. Among the topics that are worth more research are improving the *UD(d, n).m* function to make it run more efficiently. We sincerely believe that it can be done in linear time complexity. In fact, making this one function more efficient would improve half of the *DeBruijn* library. Another group of functions that could be much improved are those in *Section 8* dealing with locating lists of nodes within a specified distance and at a specified distance. If improved, this would open up an opportunity to explore very large graphs of this family which could potentially improve upon many network models.

10 Bibliography

[1] Joel Baker. “De Bruijn Graphs and their Applications to Fault Tolerant Networks.” http://www.math.oregonstate.edu/~swisherh/JoelBaker.pdf.2011, [Jun 16, 2014].

[2] S. J.A. Bondy and U. S. R. Murty. "Graph Theory with Applications”. Fifth printing. MacMillan Press Ltd. New York, Amsterdam, Oxford. 1982.

[3] Zhen LIU. "Optimal Routing in the Debruijn Networks.”
INRIA Centre Sofia Antipolis 06560 Valbonne France. 1990. pp 537-543.

[4] Manish Rastog & Brajendra Gupta. "On the Number PI an Interactive E-Module"
http://www.mathresource.iitb.ac.in/linear20algebra/FinalPi_Method/PI-HTMLPAGE/Page02.htm

11 Appendix 1–Calculating Distance

11.1 Distance in Directed De Bruijn Graphs

We demonstrate here with example how to calculate distance in a directed graph. Given the distance formula for a directed graph is

$$DD(X,Y) = n - l$$

where,

$$l = \max\{s | 1 \leq s \leq n, x_{n-l+1}x_{n-l+2}\dots x_n = y_1y_2\dots y_s\}$$

find $DD(010, 110)$ in $DG(2,3)$. For starters, we know $n = 3$ because it is given in $DG(d,n)$. So we plug n into the equation first.

$$DG(2,3) = 3 - l$$

Now to solve for l.

$$l = \max\{s | 1 \leq s \leq 3\}$$

$$\text{when } s = 1, x_{n-s+1=3-1+1=3} = 0 \neq y_1 = 1, \text{ Fail}$$

$$\text{when } s = 2, x_{n-s+1=3-2+1=2} = 1 = y_1 = 1, \text{ Pass}$$

$$\text{but, } x_3 = 0 \neq y_2 = 1, \text{ so s=2 Fails}$$

$$\text{when } s = 3, x_{n-s+1=3-1+1=3} = 0 \neq y_1 = 1, \text{ Fail}$$

Since there are no string matches, $\max\{s\} = \emptyset,$ therefore $D(010, 110) = (3) - (0) = 3$.

11.2 Distance in Undirected De Bruijn Graphs

We demonstrate here using example how to calculate distance in an undirected graph. Given the formula for finding distance in an undirected graph is

$$2n - 1l + \min\{\min_{1\leq i,j\leq n}(i - j - l_{i,j}(X,Y)), \min_{1\leq i,j\leq n}(-i + j - r_{i,j}(X,Y))\}$$

where,

$$l_{i,j}(X,Y) = \max\{s | s \leq j, s \leq n - i + 1, x_ix_{i+1}\dots x_{i+s-1} = y_{j-s+1}y_{j-s+2}\dots Y_j\}$$
$$r_{i,j}(X,Y) = \max\{s | s \leq i, s \leq n - j + 1, x_{i-s+1}x_{i-s+2}\dots x_i = y_jy_{j+1}\dots Y_{j+s-1}\}$$

find $DD(010, 110)$ in $UG(2,3)$. Plugging in n we get

$$2(3) - 1 + \min\{\min_{1\leq i,j\leq 3}(i - j - l_{i,j}(X,Y)), \min_{1\leq i,j\leq 3}(-i + j - r_{i,j}(X,Y))\}$$

There are $2 * 3^2$ combinations to check. That is 9 on LHS and then 9 on the RHS.

$$\{(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)\}$$

To solve this problem, we are going to divide and conquer solving the inner two minimums first.

Starting with the minimum LHS let $i = 1, j = 1$

$$\min_{1\leq i,j\leq n}(i - j - \max\{s|s \leq j, s \leq n - i + 1\})$$

$$\min_{i=1,j=1}(1 - 1 - \max\{s|s \leq 1, s \leq 3 - 1 + 1\})$$

$$\min_{i=1,j=1}(-\max\{s \leq 1, s \leq 3\})$$

When $s = 1, x_{i=1}y_{j-s+1=1}$. Since $x_1 = 0 \neq y_1 = 1$, exclude as candidate for max.
Since there is no match, $\max(s) = 0$, $\min_{i=1,j=1}(-\max\{s \leq 1, s \leq 3\}) = 0$

Continuing on the minimum LHS let $i = 1, j = 2$

$$\min_{1\leq i,j\leq n}(i - j - \max\{s|s \leq j, s \leq n - i + 1\})$$

$$\min_{i=1,j=2}(1 - 2 - \max\{s|s \leq 2, s \leq 3 - 1 + 1\})$$

$$\min_{i=1,j=2}(-1 - \max\{s \leq 2, s \leq 3\})$$

When $s = 2, x_{i=1}y_{j-s+1=2-2+1=1}$. Since $x_1 = 0 \neq y_1 = 1$, exclude as candidate for max.
When $s = 1, x_{i=1}y_{j-s+1=2-1+1=2}$. Since $x_1 = 0 \neq y_2 = 1$, exclude as candidate for max.
Since there is no match, $\max(s) = 0$, $\min_{i=1,j=2}(-1 - \max\{s \leq 2, s \leq 3\}) = -1$

Continuing on the minimum LHS let $i = 1, j = 3$

$$\min_{1\leq i,j\leq n}(i - j - \max\{s|s \leq j, s \leq n - i + 1\})$$

$$\min_{i=1,j=3}(1 - 3 - \max\{s|s \leq 3, s \leq 3 - 1 + 1\})$$

$$\min_{i=1,j=3}(-2 - \max\{s \leq 3, s \leq 3\})$$

When $s = 3, x_{i=1}y_{j-s+1=3-3+1=1}$.Since $x_1 = 0 \neq y_1 = 1$, exclude as candidate for max.
When $s = 2, x_{i=1}y_{j-s+1=2}$. Since $x_1 = 0 \neq y_2 = 1$, exclude as candidate for max.
When $s = 1, x_{i=1}y_{j-s+1=3}$. Since $x_1 = 0 = y_3 = 0$, include as candidate for max.
Since there is a match, $\max(s) = 1$, $\min_{i=1,j=3}(-2 - \max\{s \leq 3, s \leq 3\}) = -3$

Continuing on the minimum LHS let $i = 2, j = 1$

$$\min_{1 \leq i,j \leq n} (i - j - \max\{s | s \leq j, s \leq n - i + 1\})$$

$$\min_{i=2,j=1} (2 - 1 - max\{s | s \leq 1, s \leq 2\})$$

$$\min_{i=2,j=1} (1 - \max\{s \leq 1, s \leq 2\})$$

When $s = 1, x_{i=2} y_{j-s+1=1}$. Since $x_2 = 1 = y_1 = 1$, include as candidate for max.
Since there is a match, $\max(s) = 1$, $\min_{i=2,j=1}(1 - \max\{s \leq 1, s \leq 2\}) = 0$

Continuing on the minimum LHS let $i = 2, j = 2$

$$\min_{1 \leq i,j \leq n} (i - j - \max\{s | s \leq j, s \leq n - i + 1\})$$

$$\min_{i=2,j=2} (2 - 2 - \max\{s | s \leq 2, s \leq 2\})$$

$$\min_{i=2,j=2} (- \max\{s \leq 2, s \leq 2\})$$

When $s = 2, x_{i=2} y_{j-s+1=1}$. Since $x_2 = 1 = y_1 = 1$, decrement s and continue.
$s - - = 1, x_{i++=3} y_{j++=2}$ Since $x_3 = 0 \neq y_2 = 1$, exclude as candidate for $max\{s\}$. When $s = 1, x_{i=2} y_{j-s+1=2}$. Since $x_2 = 1 = y_2 = 1$, include as candidate for max.
Since there is a match, $\max(s) = 1$, $\min_{i=2,j=2}(- \max\{s \leq 2, s \leq 2\}) = -1$

Continuing on the minimum LHS let $i = 2, j = 3$

$$\min_{1 \leq i,j \leq n} (i - j - \max\{s | s \leq j, s \leq n - i + 1\})$$

$$\min_{i=2,j=3} (2 - 3 - \max\{s | s \leq 3, s \leq 2\})$$

$$\min_{i=2,j=3} (-1 - \max\{s \leq 3, s \leq 2\})$$

When $s = 2, x_{i=2} y_{j-s+1=2}$. Since $x_2 = 1 = y_2 = 1$, decrement s and continue.
$s - - = 1, x_{i++=3} y_{j++=3}$ Since $x_3 = 0 = y_3 = 0$, include as candidate for $\max\{s\}$. When $s = 1, x_{i=2} y_{j-s+1=2}$. Since $x_2 = 1 = y_2 = 1$, include as candidate for max.
Since there is a match, $\max(s) = 2$, $\min_{i=2,j=3}(-1 - \max\{s \leq 3, s \leq 2\}) = -3$

Continuing on the minimum LHS let $i = 3, j = 1$

$$\min_{1 \leq i,j \leq n} (i - j - \max\{s | s \leq j, s \leq n - i + 1\})$$

$$\min_{i=3,j=1} (3 - 1 - \max\{s | s \leq 1, s \leq 1\})$$

$$\min_{i=3,j=1}(2 - \max\{s \leq 1, s \leq 1\})$$

When $s = 1, x_{i=3}y_{j-s+1=1}$. Since $x_3 = 0 \neq y_1 = 1$, exclude as candidate for $\max\{s\}$.
Since there is no match, $\max(s) = 0$, $\min_{i=3,j=1}(2 - \max\{s \leq 1, s \leq 1\}) = 2$

Continuing on the minimum LHS let $i = 3, j = 2$

$$\min_{1\leq i,j\leq n}(i - j - \max\{s | s \leq j, s \leq n - i + 1\})$$

$$\min_{i=3,j=1}(3 - 2 - \max\{s | s \leq 2, s \leq 1\})$$

$$\min_{i=3,j=1}(1 - \max\{s \leq 2, s \leq 1\})$$

When $s = 1, x_{i=3}y_{j-s+1=2}$. Since $x_3 = 0 \neq y_2 = 1$, exclude as candidate for $\max\{s\}$.
Since there is no match, $\max(s) = 0$, $\min_{i=3,j=2}(1 - \max\{s \leq 2, s \leq 1\}) = 1$

Continuing on the minimum LHS let $i = 3, j = 3$

$$\min_{1\leq i,j\leq n}(i - j - \max\{s | s \leq j, s \leq n - i + 1\})$$

$$\min_{i=3,j=3}(3 - 3 - \max\{s | s \leq 3, s \leq 1\})$$

$$\min_{i=3,j=3}(- \max\{s \leq 3, s \leq 1\})$$

When $s = 1, x_{i=3}y_{j-s+1=3}$. Since $x_3 = 0 = y_3 = 0$, include as candidate for $\max\{s\}$.
Since there is a match, $\max(s) = 1$, $\min_{i=3,j=3}(- \max\{s \leq 3, s \leq 1\}) = -1$

Now to calculate the minimums on the RHS let $i = 1, j = 1$

$$\min_{1\leq i,j\leq n}(-i + j - \max\{s | s \leq i, s \leq n - j + 1\})$$

$$\min_{i=1,j=1}(-1 + 1 - \max\{s | s \leq 1, s \leq 3 - 1 + 1\})$$

$$\min_{i=1,j=1}(- \max\{s \leq 1, s \leq 3\})$$

When $s = 1, x_{i-s+1=1}y_{j=1}$. Since $x_1 = 0 \neq y_1 = 1$, exclude as candidate for $\max\{s\}$.
Since there is no match, $\max(s) = 0$, $\min_{i=1,j=1}(- \max\{s \leq 1, s \leq 3\}) = 0$

Continuing to calculate the minimums on the RHS let $i = 1, j = 2$

$$\min_{1\leq i,j\leq n}(-i + j - \max\{s | s \leq i, s \leq n - j + 1\})$$

$$\min_{i=1,j=2}(-1 + 2 - \max\{s | s \leq 1, s \leq 3 - 2 + 1\})$$

$$\min_{i=1,j=2}(1 - \max\{s \leq 1, s \leq 2\})$$

When $s = 1, x_{i-s+1=1}y_{j=2}$. Since $x_1 = 0 \neq y_2 = 1$, exclude as candidate for $\max\{s\}$.
Since there is no match, $\max(s) = 0$, $\min_{i=1,j=2}(1 - \max\{s \leq 1, s \leq 2\}) = 1$

Continuing to calculate the minimums on the RHS let $i = 1, j = 3$

$$\min_{1\leq i,j\leq n}(-i + j - \max\{s | s \leq i, s \leq n - j + 1\})$$

$$\min_{i=1,j=3}(-1 + 3 - \max\{s | s \leq 1, s \leq 3 - 3 + 1\})$$

$$\min_{i=1,j=3}(2 - \max\{s \leq 1, s \leq 1\})$$

When $s = 1, x_{i-s+1=1}y_{j=3}$. Since $x_1 = 0 = y_3 = 0$, include as candidate for $\max\{s\}$.
Since there is a match, $\max(s) = 1$, $\min_{i=1,j=3}(2 - \max\{s \leq 1, s \leq 1\}) = 1$

Continuing to calculate the minimums on the RHS let $i = 2, j = 1$

$$\min_{1\leq i,j\leq n}(-i + j - \max\{s | s \leq i, s \leq n - j + 1\})$$

$$\min_{i=2,j=1}(-2 + 1 - \max\{s | s \leq 2, s \leq 3 - 1 + 1\})$$

$$\min_{i=2,j=1}(-1 - \max\{s \leq 2, s \leq 3\})$$

When $s = 2, x_{i-s+1=1}y_{j=1}$. Since $x_1 = 0 \neq y_1 = 1$, exclude as candidate for $\max\{s\}$.
When $s = 1, x_{i-s+1=2}y_{j=1}$. Since $x_2 = 1 = y_1 = 1$, include as candidate for $\max\{s\}$..
Since there is a match, $\max(s) = 1$, $\min_{i=2,j=1}(-1 - \max\{s \leq 2, s \leq 3\}) = -2$

Continuing to calculate the minimums on the RHS let $i = 2, j = 2$

$$\min_{1\leq i,j\leq n}(-i + j - \max\{s | s \leq i, s \leq n - j + 1\})$$

$$\min_{i=2,j=2}(-2 + 2 - \max\{s | s \leq 2, s \leq 3 - 2 + 1\})$$

$$\min_{i=2,j=2}(- \max\{s \leq 2, s \leq 2\})$$

When $s = 2, x_{i-s+1=1}y_{j=2}$. Since $x_1 = 0 \neq y_2 = 1$, exclude as candidate for $\max\{s\}$.
When $s = 1, x_{i-s+1=2}y_{j=2}$. Since $x_2 = 1 = y_2 = 1$, include as candidate for $\max\{s\}$..
Since there is a match, $\max(s) = 1$, $\min_{i=2,j=2}(- \max\{s \leq 2, s \leq 2\}) = -1$

Continuing to calculate the minimums on the RHS let $i = 2, j = 3$

$$\min_{1\leq i,j\leq n}(-i + j - \max\{s | s \leq i, s \leq n - j + 1\})$$

$$\min_{i=2,j=3}(-2+3-\max\{s|s\leq 2, s\leq 3-3+1\})$$

$$\min_{i=2,j=3}(1-\max\{s\leq 2, s\leq 1\})$$

When $s=1, x_{i-s+1=2}y_{j=3}$. Since $x_2=1\neq y_3=0$, exclude as candidate for $\max\{s\}$.
Since there is no match, $\max(s)=0$, $\min_{i=2,j=3}(1-\max\{s\leq 2, s\leq 1\})=1$

Continuing to calculate the minimums on the RHS let $i=3, j=1$

$$\min_{1\leq i,j\leq n}(-i+j-\max\{s|s\leq i, s\leq n-j+1\})$$

$$\min_{i=3,j=1}(-3+1-\max\{s|s\leq 3, s\leq 3-1+1\})$$

$$\min_{i=3,j=1}(-2-\max\{s\leq 3, s\leq 3\})$$

When $s=3, x_{i-s+1=1}y_{j=1}$. Since $x_1=0\neq y_1=1$, exclude as candidate for $\max\{s\}$.
When $s=2, x_{i-s+1=2}y_{j=1}$. Since $x_2=1=y_1=1$, include as candidate for $\max\{s\}$.
$s---=1$, $x_{i++=3}y_{j++=2}$. Since $x_3=0\neq y_2=1$, exclude as a candidate for $\max\{s\}$.
When $s=1, x_{i-s+1=3}y_{j=1}$. Since $x_3=0\neq y_1=1$, exclude as candidate for $\max\{s\}$.
Since there is no match, $\max(s)=0$, $\min_{i=3,j=1}(-2-\max\{s\leq 3, s\leq 3\})=-2$

Continuing to calculate the minimums on the RHS let $i=3, j=2$

$$\min_{1\leq i,j\leq n}(-i+j-\max\{s|s\leq i, s\leq n-j+1\})$$

$$\min_{i=3,j=2}(-3+2-\max\{s|s\leq 3, s\leq 3-2+1\})$$

$$\min_{i=3,j=2}(-1-\max\{s\leq 3, s\leq 2\})$$

When $s=2, x_{i-s+1=2}y_{j=2}$. Since $x_2=1=y_2=1$, include as candidate for $\max\{s\}$.
$s---=1$, $x_{i++=3}y_{j++=3}$, since $x_3=0=y_3=0$, include as a candidate for $\max\{s\}$.
Since there is a match, $\max(s)=2$, $\min_{i=3,j=2}(-1-\max\{s\leq 3, s\leq 2\})=-3$

Continuing to calculate the minimums on the RHS let $i=3, j=3$

$$\min_{1\leq i,j\leq n}(-i+j-\max\{s|s\leq i, s\leq n-j+1\})$$

$$\min_{i=3,j=3}(-3+3-\max\{s|s\leq 3, s\leq 3-3+1\})$$

$$\min_{i=3,j=3}(-\max\{s\leq 3, s\leq 1\})$$

When $s=1, x_{i-s+1=3}y_{j=3}$. Since $x_3=0=y_3=0$, include as candidate for $\max\{s\}$.
Since there is a match, $\max(s)=1$, $\min_{i=3,j=3}(-\max\{s\leq 3, s\leq 1\})=-1$

From LHS we have $\min\{0, -1, -3, 0, -1, -3, 2, 1, -1\} = -3$,
and from RHS we have $\min\{0, 1, 1, -2, -1, 1, -2, -3, -1\} = -3$
The equation simplifies to $2(3) - 1 + \min\{-3, -3\} = 2$.