Optimizing Circuits Paper

Pranavi Boyalakuntla
Thomas Jagielski

Discrete Math

Abstract

Logic graphs are essential in the construction of silicone chips as they allow engineers to minimize the area used and ensure that the source and drain diffusions are used most efficiently. This paper will walk you through how to construct logic gates from CMOS transistors and how to turn that into a logic graph in the way best suited to help optimize chip real estate. There is supplementary code that is included.

Logic Gates

When constructing logic circuits, there are several basic logic gates that are used: "and" gates and "or" gates. At the higher level, these are most commonly represented by the images below.

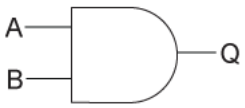

Figure 1: And Gate

Figure 2: Or Gate

These two logic gates take two inputs and have one output. In the case of the "and" gate, the output, Q, is only high when both inputs, A and B, are high. With the "or" gate, the output, Q, is high when any of the inputs are high. The behavior of these gates is shown in the truth tables below.

Input		Output
A	B	F = A.B
0	0	0
0	1	0
1	0	0
1	1	1

Figure 3: And Gate Truth Table

Input		Output
A	B	F = A+B
0	0	0
0	1	1
1	0	1
1	1	1

Figure 4: Or Gate Truth Table

MOS Transistors

These gates are actually higher level abstractions for what actually constructs them. "And" and "or" gates are constructed by two types of transistors: nMOS and pMOS transistors.

NMOS transistors conduct when a high voltage is applied to the gate and will not conduct when a low voltage is applied to the gate.

PMOS transistors conduct when a low voltage is applied to the gate and will not conduct when a high voltage is applied to the gate. In this sense, NMOS and PMOS act opposite to each other.

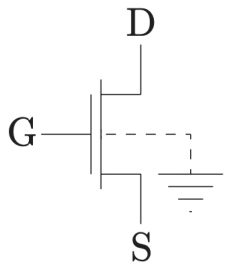

Figure 5: nMOS transistor

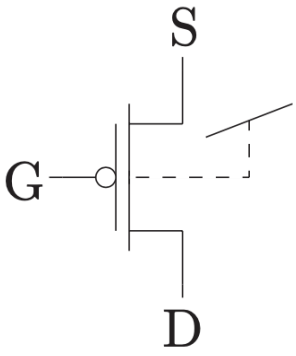

Figure 6: pMOS transistor

As seen above, the circuits for the two transistors look relatively similar. One of the major differences when depicting the two in circuits is the circle unfilled circle in the pMOS transistor.

Representing Logic with CMOS Transistors

Referencing figure 7 below, when constructing "and" gates, two MOS transistors are placed in series such that X will only connect electrically to Y if both inputs, A and B, are high. If A is high, but B is not high, then the first nMOS transistor will conduct, but the second nMOS transistor will not. Therefore, X will not be electrically connected to Y. This applied if A is high and B is not as well.

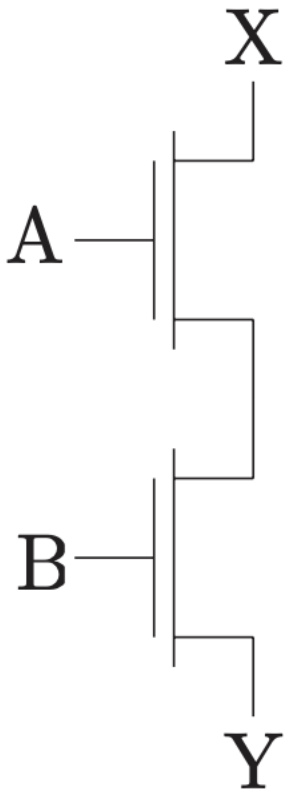

Figure 7: "And" gate constructed from two nMOS transistors in series

Referencing figure 8 below, when constructing "or" gates, two nMOS transistors are placed in parallel such that X will connect to Y if either A or B is high. If A and B are both high, X will still be electrically connected to Y.

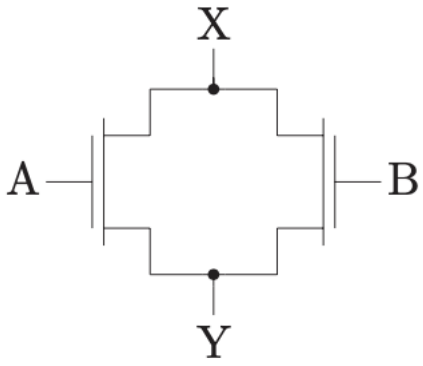

Figure 8: "Or" gate constructed from two nMOS transistors in parallel

Both examples detailed above use nMOS transistors. If you would like the transistor to conduct if the input is low, you would use pMOS transistors. Examples are shown in figures 9 and 10.'

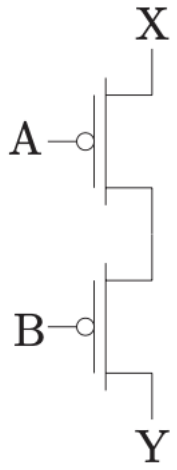

Figure 9: "Not A and Not B" gate constructed from two pMOS transistors in series

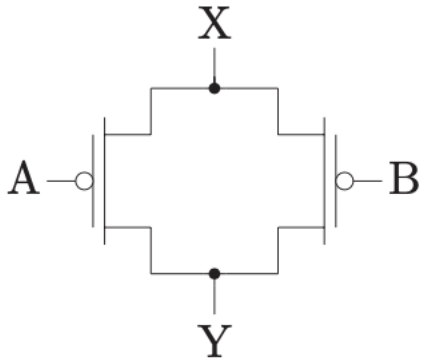

Figure 10: "Not A or Not B" gate constructed from two pMOS transistors in series

CMOS Gate Structure

There are two sub-networks in a CMOS gate: the pull up network and the pull down network. The pull up network is made of pMOS transistors and connected to VDD while the pull down network is made of nMOS transistors and connected to ground.

The pull up and pull down networks are complements of each other. In other words, if you represented the pull up circuit using boolean algebra rules, the pull down circuit would be the direct complement of that. Another way to arrive at this is by replacing the nMOS transistors with pMOS transistors and all series connections with parallel connections.

Referencing figure 11 below, the pull up network with VDD as the input is constructed of pMOS transistors. It can be represented by Z is equal to not A and B (Z = ~(A*B)). The pull down network is the direct complement of the pull up network and can be represented by Z is equal to A or B (Z = A+B). This pull down network can also be reached by replacing the pMOS transistors with nMOS transistors and turning serial connections into parallel connections.

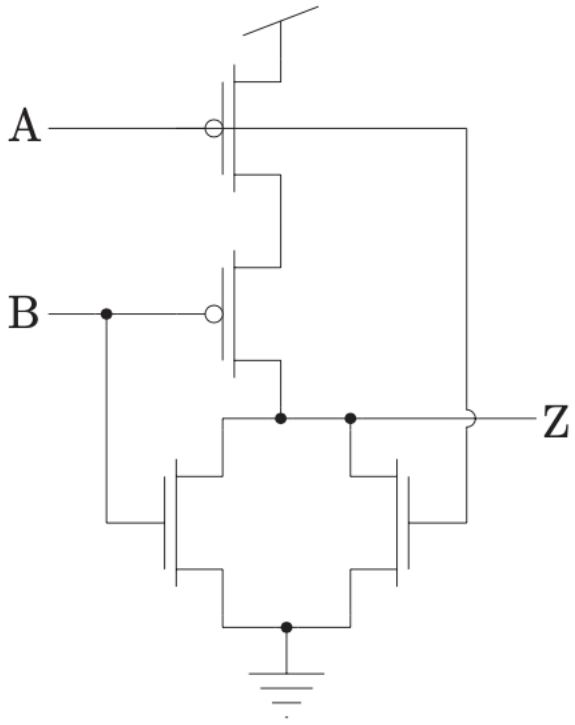

Figure 11: CMOS Logic Gate for the equation Z = ~(A*B)

It can be helpful to note for future reference that the open circle represents an inversion acts as an inversion.

Circuits to Logic Graphs

When turning a circuit into a logic graph, it can be helpful to break it down into the PUN and the PDN. It is important to keep in mind that the pull up network utilizes pMOS transistors while the pull down network utilizes nMOS transistors. This is what we do in figures 12 and 13 below. Figure 12 shows the PUN and figure 13 shows the PDN. This circuit can be represented by the boolean equation below.

$$Z = \bar{A}\bar{B}$$

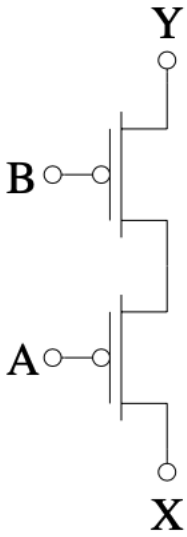

Figure 12: PUN for the circuit

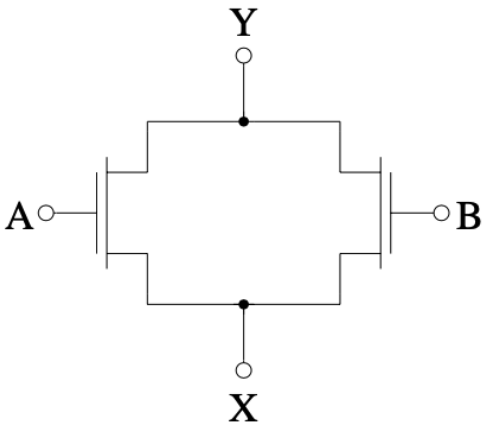

Figure 13: PDN for the circuit

These can then be individually turned into logic graphs. An open circle represents an inversion while a closed circle represents an nMOS transistor. These logic graphs are shown below and are relatively straightforward.

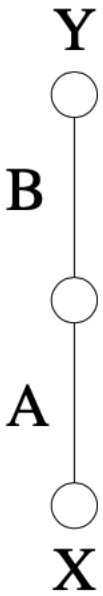

Figure 14: PUN logic graph for this circuit

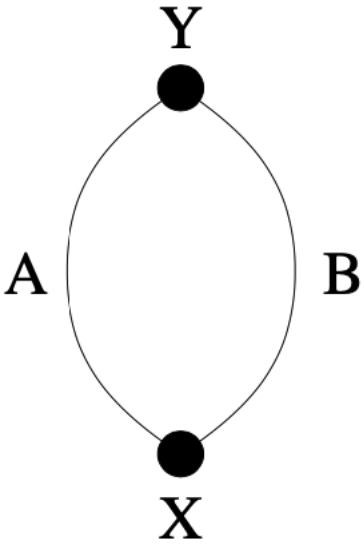

Figure 15: PDN logic graph for this circuit

As you can see above in figures 14 and 15, the transistors aren't represented by a node, but rather the edges en between.

Euler Paths

An Euler path is a path through all of the nodes in a logic graph so that an edge is only travelled through once. If we find a consistent Euler path through the PUN and PDN, we can design a more space efficient chip and share the source and drain diffusion better.

Consider the boolean equation below. This can be represented by the circuit in figure 16.

$$Z = \bar{A}\bar{B} + \bar{C}$$

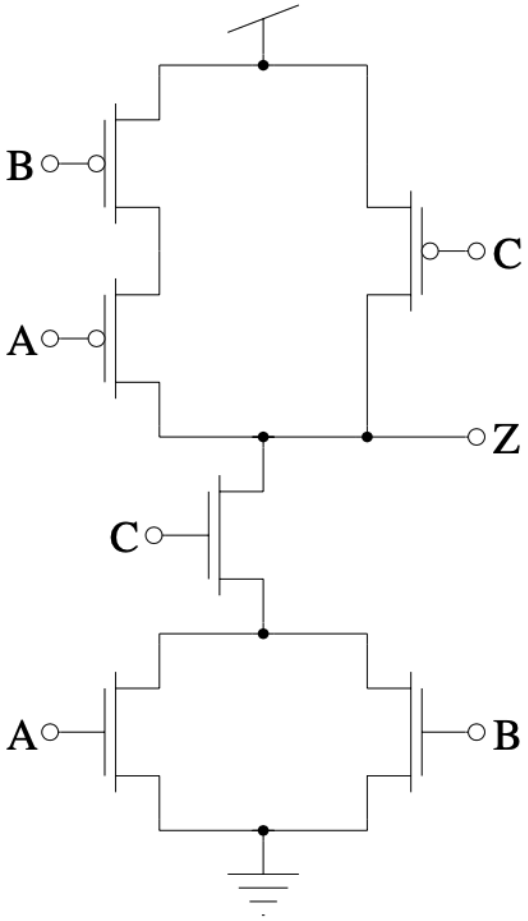

Figure 16: Combined PUN and PDN for this circuit

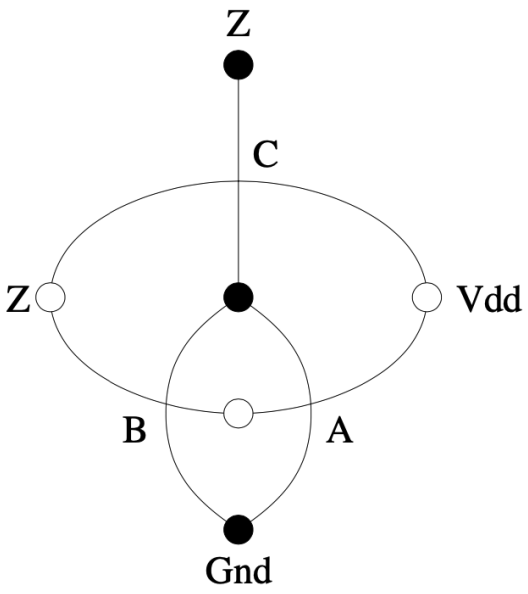

Figure 17: Corresponding logic graph for the circuit shown in figure 16.

It can be seen here that the PDN is located from Z to ground while the PUN is located from VDD to Z. They are complements of each other. This circuit can be represented by the logic graph in figure 17 above.

The path with the open circles in figure 17 represent the PUN while the path with the closed circles represents the PDN. They can be overlapped as such because they both use the same inputs: A, B, and C. This becomes more clear after the Euler path is shown in figure 18.

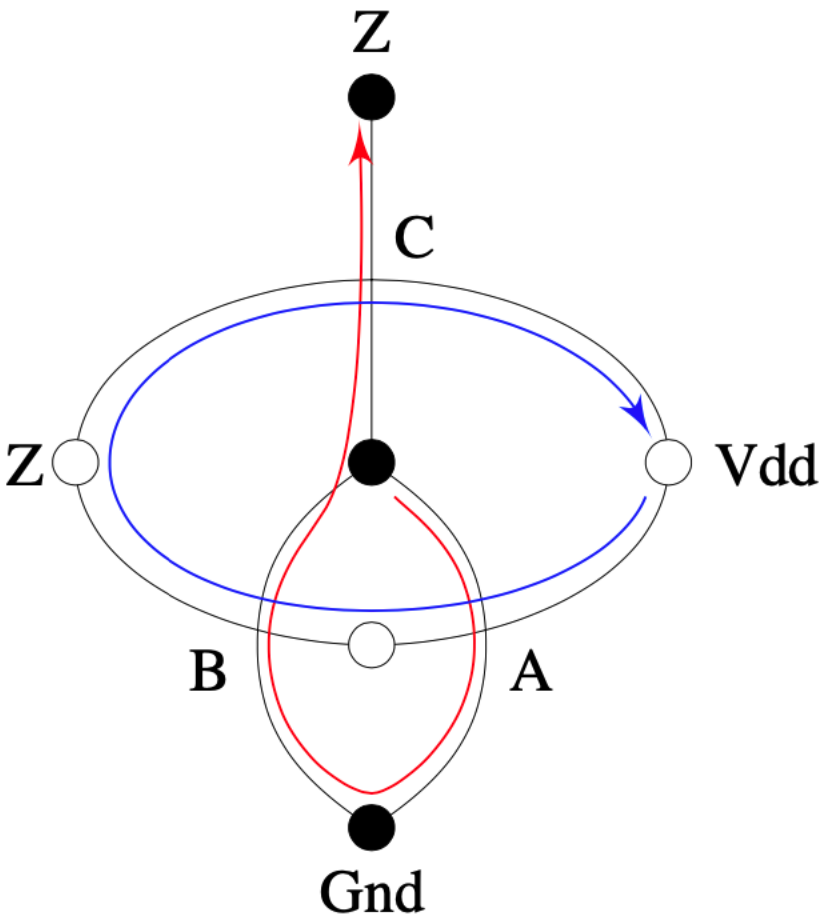

Figure 18: Euler path for the circuit

Notice how the Euler path for the PUN passes once through A, B, and then C and the Euler path for the PDN passes once through the inputs in the same order. This means that a consistent Euler path was found.

Once a consistent Euler path is found, we can order the inputs the same way when laying it out on silicone.

This process of finding a consistent Euler path can be found by using the code attached to this project. In order to use this code however, a matrix detailing the graph needs to be made for the PUN (adjacency matrix). Each vertex has both a row and a column and the contents of that row correspond to the number of connections between the two vertices. For example, the contents of row 0 and column 0 would be 0 as there are no edges between the vertex and itself.

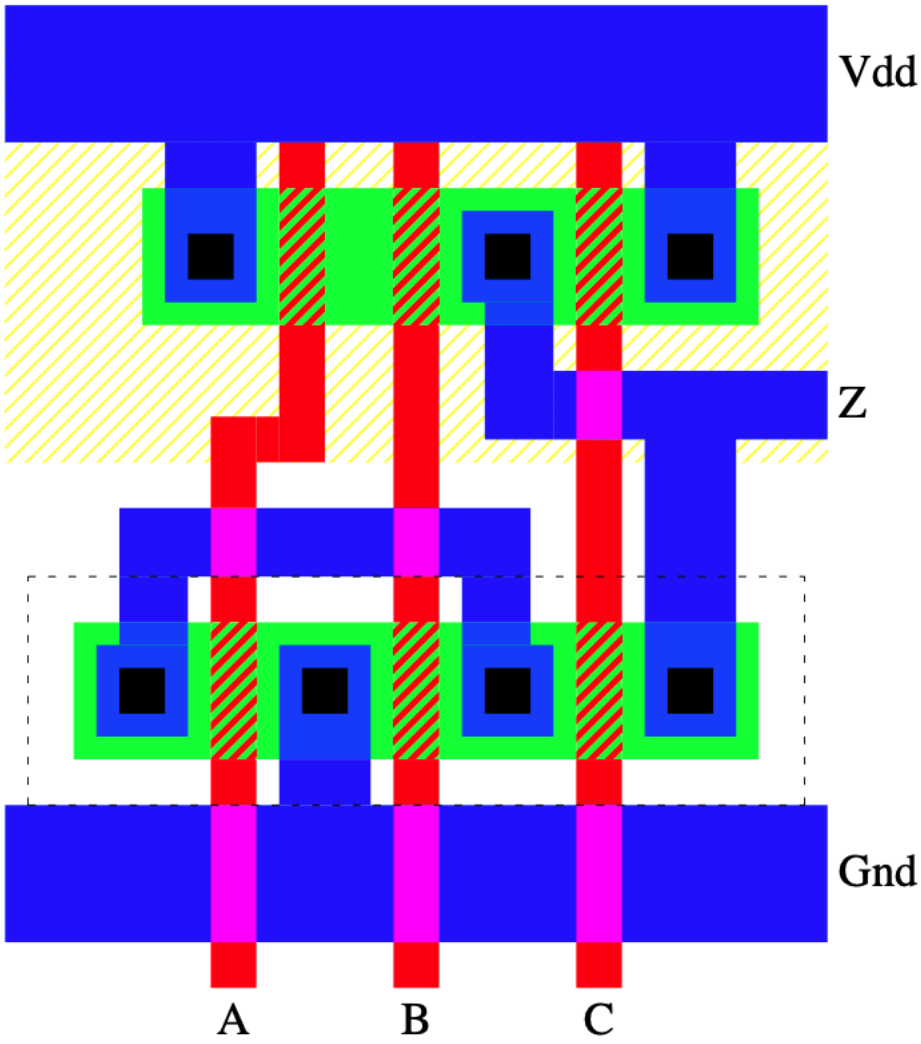

Figure 19: Layout for this circuit

This layout uses the consistent Euler path found by placing in the inputs in the same order from left to right: A, B, and then C. The green poly strips connect horizontally. The top half of this layout is the PUN and the bottom is the PDN. As shown in the PUN Euler path, VDD connects to both A and C directly, while both B and C connect to Z. This is represented in the top half of the circuit. As shown in the PDN Euler path, GND connects to A and B, C is connected to Z, and all three are connected as well. This is also shown in the layout above.

Without the use of the Euler path to help design this, there was a chance that it could have been designed using not continuous source/drain diffusions, or taking up more space than necessary. Figuring out the Euler path first eliminated that.

Note:

1. Euler paths do not exist for all expressions.
2. Different forms of the same expression may affect whether or not an Euler path exists.
3. Euler paths are not unique.

Source Code

```
import copy

def findAllPaths(graph,start_vertex,dictionary):
    """
    Function to find all Euler paths through a givin logic graph

    INPUTS:
    graph - adjacency matrix containing a logic graph
    start_vertex - the vertex in logic graph you begin on
    dictionary - dictionary containing transistor letter cooresponding to a
                 given edge

    OUTPUT
    euler_path - dictionary containing the edge and transistor orderings
    """
    euler_path = {} # Initialize Euler path dictionary
    graph_original = copy.deepcopy(graph) # deepcopy the original adjacency matrix
    for vertex in start_vertex:
        graph = copy.deepcopy(graph_original) # deepcopy the original adjacency matrix
        path = findPath(graph,vertex) # Find the Euler path (connecting edges)
        ordering = label_transistors(dictionary,path) # Label the paths with transistor letters
        path_ouput = {str(ordering):str(path)} # Place the connecting edge and transistor orderings
                                               # in a dictionary to store both representations
        euler_path.update(path_ouput) # Append the {key:value} pair into the output dictionary
    return(euler_path)

def findPath(graph, vertex):
    """
    Function to find a Euler path through a graph given a starting point

    INPUTS:
    graph - adjacency matrix representing graph
    vertex - starting vertex for logic graph

    OUTPUTS:
    path - ouput Euler path that begins at 'vertex'
    """
    path = [] # Initialize list to store the output path into
    counter = 1 # Initialize a counter
    while 1:
        current_point = graph[vertex] # Isolate the current point in the adjacency matrix
        check_01 = findOnesAndZeros(current_point) # Find if the current point contains only zeros and ones
        for i in range(len(current_point)):
            if not check_01: # If the point does not contain only zeros and ones
                if current_point[i] != 1 and current_point[i] != 0: # And the current edge being considered is not a zero or one
                    current_point[i]  = current_point[i] - 1 # Subtract one from the current point (to indicate you have been at thi
                    graph[vertex] = current_point # Append the subtracted vertex value to the adjacency matrix
                    opposite = graph[i] # Index to the opposite representation of the edge **NOTE: because edges are counted twice i
                    opposite[vertex] = opposite[vertex] - 1 # Subtract one from the opposite representation of the edge
                    graph[i] = opposite # Append the subtracted opposite vertex to the original graph
                    current_point = graph[i] # Set the current point as the new point
                    path.append(str(vertex) + '-' + str(i)) # Append the connection 'old-new' to the list path
                    vertex = i # Set the new vertex as variable 'vertex'
            else: # If the current point only contains zeros and ones
                if current_point[i] == 1: # Find an edge in the adjacency matrix
                    current_point[i]  = current_point[i] - 1 # subtract one from the current point to indicate you have been here
                    graph[vertex] = current_point # Set the traversed (subtracted) vertex in the adjacency matrix in the original
                    opposite = graph[i] # Index to the opposite representation of the point
                    opposite[vertex] = opposite[vertex] - 1 # Subtract one from the opposite representation of the edge
                    graph[i] = opposite # Append the subtracted opposite vertex to the original graph
                    current_point = graph[i] # Set the current point as the new point
                    path.append(str(vertex) + '-' + str(i)) # Append the connection 'old-new' to the list path
                    vertex = i # Set the new vertex as variable 'vertex'
            counter = 0 # Initialize the counter as zero
            for i in range(len(graph)):
                counter += sum(graph[i]) # Find the sum of all values to determine if every edge has been traversed
            if counter == 0: # If there are no more edges end the for loop
                break
        if counter == 0: # Once there are no more edges end the while loop and return the edge orderings
            break
    return(path)

def findOdd(graph):
    """
    Function to find the odd verticies on a graph

    INPUT:
    graph - adjacency matrix of graph
```

```
    OUPUT:
    start_vertex - all of the odd verticies in the graph
    """
    numVertex = len(graph) # Find the length of the adjacency matrix
    start_vertex = [] # Initialize a list to store the starting verticies in
    for i in range(numVertex):
        if sum(graph[i]) % 2 != 0: # Check if the vertex is odd or even by finding sum % 2
            start_vertex.append(i) # If the vertex is odd, append it to the output list
    if start_vertex == []: # If there are no odd verticies in the graph
        for i in range(len(graph)):
            start_vertex.append(i) # Append all verticies
    return start_vertex

def findOnesAndZeros(l):
    """
    Function to find if a point on graph only contains ones and zeros

    INPUT:
    l - single line of adjacency matrix represented as a list

    OUPUT:
    True/False - True if point only contains ones and zeros
                 False if the point contains other values than only ones and zeros
    """
    for i in range(len(l)):
        if l[i] != 1 and l[i] != 0: # Check if there are any points that are non-one or non-zero
            return False
    return True

def label_transistors(dictionary,path):
    """
    Function to label the paths with transistor letterings

    INPUTS:
    dictionary - dictionary containing the cooresponding transistor and edge
    path - list containing the connecting edge orderings

    OUTPUT:
    label - list of the transistor ordering
    """
    label = [] # Initialize list for transistor letter ordering
    for i in range(len(path)):
        connection = path[i] # Find the edge with vertex-vertex (i.e. 1-0) representation
        transistor = dictionary.get(connection) # Find the cooresponding transistor letter
        if transistor == None: # If no value was found for the key
            transistor = dictionary.get(connection[::-1]) # Check the reverse ordering (i.e. "1-0" and "0-1")
        label.append(transistor) # Append the transistor letter to output list
    return label

def path_options(PDN_paths, PUN_paths):
    """
    Function to find consistent Euler paths

    INPUT:
    PDN_paths - dictionary with PDN paths and transistor ordering
    PUN_paths - dictionary with PUN paths and transistor ordering

    OUPUT:
    consistent_order - list containing the consistent Euler paths
    """
    consistent_order = [] # Initialize list to store consistent orders in
    for key in PDN_paths.keys(): # For each key in the PDN dictionary
        if key in PUN_paths: # Check if it is in the PUN dictionary
            consistent_order.append(key) # If it is contained in both, append the key to the output list
    return consistent_order


if __name__ == "__main__":
    ## CREATE THE ADJACENCY MATRICIES ##
    PDN = [[0,1,0],[1,0,2],[0,2,0]]
    PUN = [[0,1,1],[1,0,1],[1,1,0]]

    ## CREATE THE EDGE - TRANSISTOR LETTER REFERENCE ##
    PDNlabel = {"1-0":"C","1-2":"A","2-1":"B"}
    PUNlabel = {"0-1":"A","1-2":"B","2-0":"C"}

    ## RUN THE FUNCTIONS TO FIND THE EDGE/TRANSISTOR ORDERING##
    PDN_paths = findAllPaths(PDN,findOdd(PDN),PDNlabel)
    PUN_paths = findAllPaths(PUN,findOdd(PUN),PUNlabel)

    ## PRINT THE OUTPUT ##
```

```
    #print(PDN_paths)
    #print(PUN_paths)
    path_options_ouput = path_options(PDN_paths,PUN_paths)
    print(path_options_ouput)
```

Resources

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/5b08233c-cb88-47b9-95a6-b033c7113b8b/Annotated_Bibliography.pdf

Brad Minch Discrete Math Presentation

Static CMOS Logic

Optimum Gate Ordering of CMOS Logic Gates Using Euler Path Approach Some Insights and Explanations

Slides

Presentation

Code

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/2ea49314-a656-4b5b-a741-18fd18efda4b/euler_circuit.py

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/c28eedcc-8390-4afa-986e-d80c08bc4560/euler_circuit.pdf