Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Finding a Compatible Euler cycle: a Fast Algorithm

Tesis de Licenciatura en Ciencias de la Computación

Alejandro Candioti
amcandio@gmail.com

Directora: Verónica Becher

Buenos Aires, 1 de diciembre de 2019

UN ALGORITMO RÁPIDO PARA ENCONTRAR UN CAMINO EULERIANO COMPATIBLE EN UN GRAFO DIRIGIDO

Abstract

Un ciclo Euleriano en un grafo G es un camino cerrado que usa todos los arcos de G exactamente una vez. Dos ciclos Eulerianos son compatibles si no comparten ningún camino de longitud 2. Fleischner y Jackson demuestran en 1990 que para todo camino Euleriano en un grafo dirigido de grado mínimo 3, existe otro ciclo Euleriano compatible. El resultado principal de esta tesis es un algoritmo para calcular un ciclo Euleriano compatible a uno dado en un grafo dirigido de grado mínimo 3, con complejidad de peor caso $O(|E| * log(|V|))$ donde $|V|$ y $|E|$ la cantidad de vértices y arcos del grafo. Nuestro algoritmo se basa en las ideas de Lin, Ward, Jain y Skiena de 2011. Un segundo resultado de esta tesis responde una pregunta de Becher y Heiber en 2011 y es un algoritmo para extender una secuencia de Bruijn de orden n a otra de orden $n + 1$ para alfabetos de 3 o más símbolos. Nuestra solución de este problema se basa en el algoritmo previamente descripto que genera un ciclo Euleriano compatible a otro dado. Esta solución también puede usarse para extender otras secuencias que son variantes de las secuencias de Bruijn, como los llamados collares perfectos.

Palabras claves: ciclos eulerianos, secuencias de Bruijn, collares perfectos

FINDING A COMPATIBLE EULER CYCLE: A FAST ALGORITHM

Abstract

An Euler cycle of a graph G is a closed path that contains all the edges in G. Two Euler cycles are compatible if they do not share a path of length 2. Fleischner and Jackson proved in 1990 that for every Euler cycle in a directed graph of minimum degree 3 there exists another Euler cycle compatible to it. The main result of this Thesis is an algorithm to calculate an Euler cycle compatible to a given one in a directed graph of minimum degree 3 with worst case time complexity of $O(|E| * log(|V|))$, where $|V|$ and $|E|$ are the amount of vertices and edges of the graph. Our algorithm is based on the ideas of Lin, Ward, Jain y Skiena in 2011. A second result of this work answers a question proposed by Becher and Heiber in 2011 and is an algorithm to extend a de Bruijn sequence of order n to another de Bruijn sequence of order $n+1$ in alphabets with 3 or more symbols. Our solution for this problem is based on the described algorithm that generates an Euler cycle compatible to a given one. This solution can also be used to extend another sequences that are variants of de Bruijn sequences, like the perfect necklaces.

Key words: Euler cycle, de Bruijn sequences, perfect necklaces

CONTENTS

1. INTRODUCTION AND STATEMENT OF RESULTS

We start with the classical definitions. A thorough presentation of this material can be read from the classical books [9, 12].

A Hamiltonian cycle of a (di)graph G is a closed (directed) path (a walk containing no repeated edges) that contains all of the vertices in G.

A (di)graph G is eulerian if it contains a closed (directed) path that contains all of the edges in G. This closed path is known as an Euler cycle. The length of a path is the number of its edges. Two Euler cycles are compatible if they do not share a path of length 2.

Example 1.1. In the (di)graph from figure 1.1 if we consider the Euler cycles $T_1 = [00, 01, 11, 12, 22, 20, 02, 21, 10]$ and $T_2 = [01, 12, 20, 00, 02, 22, 21, 11, 10]$. These cycles are compatible because there is no pair of consecutive edges present in both cycles.

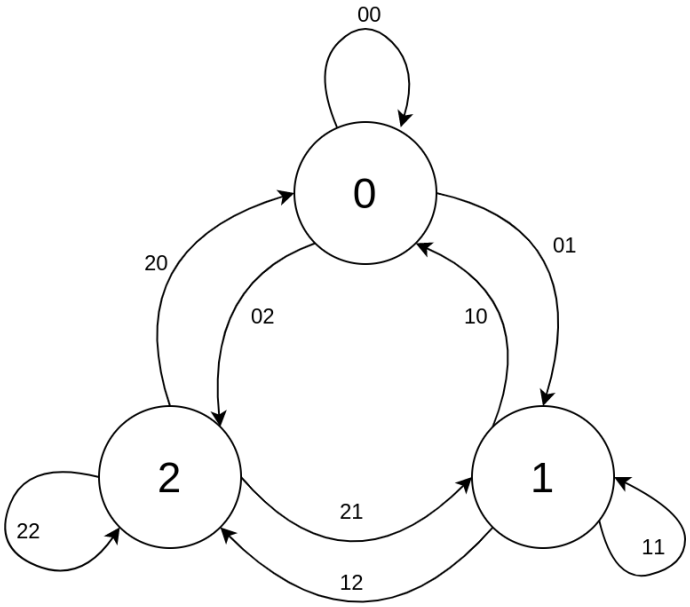

Fig. 1.1: An eulerian digraph. The edges are named based in the vertices they connect

Definition. *An eulerian (di)graph G is 2-eulerian if every Euler cycle of G admits a compatible Euler cycle.*

We say a digraph has minimum degree n if all its vertices have in-degree and out-degree at least n. Fleischner and Jackson [5] proved that an eulerian digraph G with minimum degree 3 is 2-eulerian. Lin, Ward, Jain, Skiena [10] give a constructive proof of the same result.

In this thesis we focus on the problem of quickly find a compatible Euler cycle to a given one in eulerian digraphs of minimum degree 3. Our algorithm uses the ideas of the

mentioned proof by Lin, Ward, Jain, Skiena [10]. The following is the main result of this work:

Theorem 1. *Given an Euler cycle a digraph $G = (V, E)$ with minimum degree at least 3, there is an algorithm that finds a compatible Euler cycle in time $\mathcal{O}(|E| * \log(|V|))$.*

The algorithm we present in the proof of Theorem 1 provides a solution where the efficiency comes from choosing a data structure that is convenient for the graph manipulation we do. We believe that this solution can be further improved.

In this thesis we also focus on the problem of how to extend a Hamiltonian cycle to an Euler one in some class of graphs.

Given a directed graph G, its line graph $L(G)$ is a digraph such that each vertex uv of $L(G)$ represents an edge (u, v) of G; and there is an edge from x to y in $L(G)$ if and only if the end vertex of the corresponding edge of x is the start vertex of the corresponding edge of y, in other words, if x has the form uv and y has the form vw.

A (directed) cycle C' extends a (directed) cycle C in a (di)graph G if, under some rotation, the sequence of edges of C is contiguously contained in the sequence of edges of C'. Notice that since the first and last vertex of C are the same, that vertex must appear at least twice in C'.

We prove the following result about extending a Hamiltonian cycle:

Theorem 2. *Let G be a 2-eulerian digraph with all vertices having the same out-degree and let H be a Hamiltonian cycle of $L(G)$. There exists an Euler cycle of $L(G)$ that extends H.*

We give an algorithm to generate an Euler cycle E that extends a Hamiltonian cycle from $L(G)$ where G is a digraph. We adapt the method derived from the proof of the BEST theorem [13] described by Frederiksen [7] to generate an Euler cycle based on a spanning in-tree.

With this algorithm we answer an open question of Becher and Heiber [2] on giving an algorithm to extend a de Bruijn sequence of order n to a de Bruijn sequence of order $n+1$. We also obtain an alternative of the proof given in [2] of the existence of one extension.

The results we obtained for de Bruijn sequences can be adapted to extend a variant of de Bruijn sequences, called perfect necklaces defined by Alvarez, Becher, Ferrari and Yuhjtman in [1].

2. FINDING A COMPATIBLE EULER CYCLE

The characterization of eulerian directed graphs by I.J. Good [8] states that a directed graph is eulerian if and only if it is strongly connected and the in-degree and out-degree of each vertex coincide.

As done by Lin, Ward, Jain and Skiena [10], an Euler cycle in a graph G corresponds to a pairing of each in-edge to its out-edge for each vertex $v \in G$. Such an edge-pairing defines a perfect matching between input edges to output edges of v. We call such an edge-pairing an (edge) wiring of v. Two wirings of a vertex are disjoint if the corresponding matchings are edge-disjoint. Notice that an Euler cycle defines a specific wiring for each vertex in G; however, a set of arbitrary wirings for vertices of G usually ends up with several disconnected (edge) cycles.

Theorem (Lin, Ward, Jain and Skiena [10, Theorem 7])**.** *An eulerian digraph G with minimum degree at least* 3 *is* 2*-eulerian.*

Proof. Let C be an arbitrary Euler cycle of G. Then, C defines a specific wiring for each vertex in G. We can rewire each vertex v in G such that the new wiring is disjoint and still forms an Euler cycle. Note that the initial wiring of v partitions edges of G into δ disjoint nonempty paths, namely $\{P_1, P_2, ..., P_\delta\}$, with $C = (P_1 P_2 ... P_\delta)$ in circular order. Let a_i and b_i denote the first and last edge of P_i respectively. Note that the vertex v wires b_i to $a_{1+(i \mod \delta)}$. It is easily verified that the newly constructed wiring of v by inverting the order of the paths in the Euler cycle produces a disjoint Euler cycle $(P_\delta P_{\delta-1} ... P_2 P_1)$ with respect the paths centered on v. This is because in the new cycle, the vertex v wires b_i to $a_{(i-2 \mod \delta)+1}$. Note that the argument fails for $\delta = 2$ where $(P_1 P_2) = (P_2 P_1)$.

This rewire operation generates a new Euler cycle by modifying just the paths of length 2 centered on vertex v, the paths of length 2 center in vertices distinct to v remain unchanged. Therefore, applying the same operation for every vertex generates a new Euler cycle compatible with the original one. □

Figure 2.1 shows this diagrammatically for an example for $\delta = 3$.

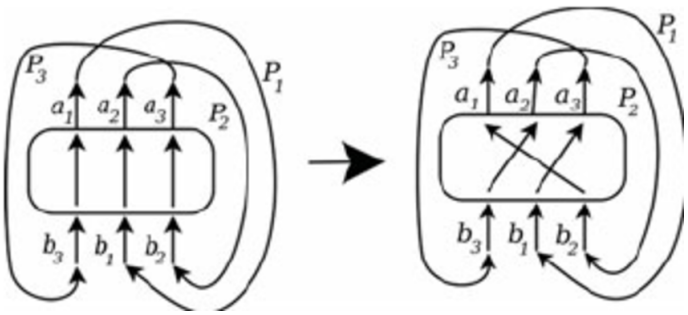

Fig. 2.1: An example of rewiring a vertex with $\delta = 3$ [10]

This proof clearly defines an algorithm to find a compatible Euler cycle to a given one (in a graph with minimum degree at least 3). At the i-th iteration a new Euler cycle E_i is generated by rewiring vertex v_i. All the paths of length 2 that have v_{i+1} as middle vertex that are present in E_{i+1} are not present in E_i. The other paths of length 2 are shared by E_i and E_{i+1}. Therefore, the last Euler cycle is compatible to the first one because the new paths of length 2 that are created in a given iteration, remain unchanged until the end.

Notice that to rewire a vertex v we must know the entering and exiting order of edges of v. After a rewire operation of a vertex, the entering and exiting order of other vertices might be altered, so that order must be recalculated after every operation to rewire that vertices. For that reason, the $\mathcal{O}(|E|)$ approach of calculating the entering and exiting order once and then rewiring every vertex does not work.

We can also think of the rewire operation of a vertex as three sub-operations on paths: split, reverse and join. At the rewire operation of a vertex v, we split the Euler cycle in multiple paths P_i all starting with v, and the we reverse and join them. This gives us an algorithm which depends on the implementation of the split, reverse and join operations on the cycle:

Algorithm 1

```
1: function FindCompatibleEulerCycle(vertices: [Vertex], cycle: Cycle)
2:     for v ∈ vertices do
3:         paths = split(cycle, v)
4:         reversedPaths = reverse(paths)
5:         cycle = join(reversedPaths)
6:     end for
7:     return cycle
8: end function
```

From now on, we refer as the set of edges and vertices of the input graph G as E and V respectively. Naturally, $|E|$ and $|V|$ are the amount of edges and vertices of G. We refer as d_v as the out-degree of vertex v, since the input graph G is eulerian, d_v it is also the in-degree of vertex v. We measure the complexity of the algorithm by considering the worst case of the amount of performed mathematical operations. As usual, we use the asymptotic big $\mathcal{O}$ notation and we say that the algorithm has time complexity $\mathcal{O}(g(x))$ to express that there is a positive constant C such that for every x the number of performed operations is at most $C|g(x)|$.

2.1 Array and List implementations

We can think of the Euler cycle as a sequence of vertices (one vertex can appear multiple times) and implement it using a circular array or a circular linked list. Since we have one vertex occurrence for every edge of G, we store $|E|$ elements on this structure.

If we use a circular array, the complexity of finding and splitting the array is $\mathcal{O}(|E|)$, as is the complexity of reversing and joining the split sub-arrays. Since we need to perform that operations to rewire each vertex, the time complexity of this implementation is $\mathcal{O}(|E| * |V|)$.

Notice that after the split of the rewire of vertex v we can have an extra sub-array (the first one) without starting with the vertex v. Since the original array represents a loop, this "orphan" sub-array must be joined with the last sub-array before reordering them.

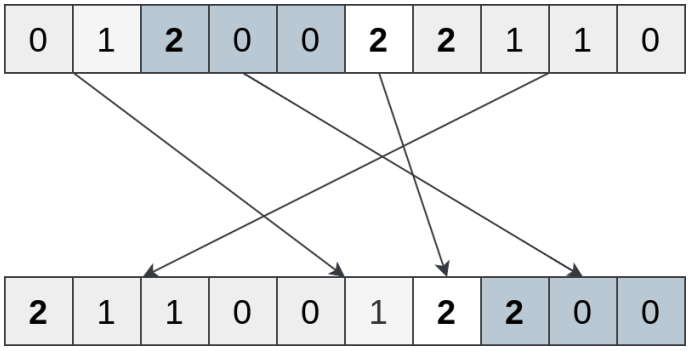

Fig. 2.2: A rewire operation on vertex 2 in the Euler cycle T_1 from Example 1.1

If we use a circular linked list to represent the cycle, the split operation of a vertex v can be made faster by keeping an array of pointers to each occurrence of v in the cycle. By doing this, the split operation is $\mathcal{O}(d_v)$. Unfortunately, the current order of the occurrences of v is uncertain, because it could had been affected by previous rewire operations. To find this order of occurrences we need to traverse the cycle and therefore $\mathcal{O}(|E|)$ operations are needed to perform a rewire operation on a vertex. This yields a $\mathcal{O}(|E| * |V|)$ time complexity for the algorithm.

We can provide a faster implementation by using ordered sets implemented as binary search trees. These ordered sets can help us to address the problem of calculating the order of occurrences of vertices.

2.2 Binary Search Tree implementation

As in the array implementation, we represent the cycle as a sequence of vertex occurrences. We refer as a vertex occurrence position as the index of the occurrence in the sequence. When we say "relative order" we refer to the order between the vertices in the sequence.

In this implementation we can represent the sequence of vertices with a Binary Search Tree where the position in the sequence is the search key of the Binary Search Tree. Every node of the Binary Search Tree corresponds to an occurrence of a vertex in the cycle, since a vertex can appear multiple times in the sequence, it can have multiple corresponding nodes in the Binary Search Tree. To recover the Euler cycle as a sequence of vertices, we must traverse the nodes of the tree in-order.

Now, the join and split operations on the cycle correspond to join and split operations on the Binary Search Tree. The join operation of two Binary Search Trees t_1 and t_2, returns a Binary Search Tree containing all the elements in t_1 and t_2. It requires that all the elements of t_1 are smaller than all the elements in t_2. The split operation of a Binary Search Tree by a node returns two Binary Search Trees t_1 and t_2. Those elements smaller than the node value will be on t_1 and the rest will be on t_2.

In this representation, to rewire a vertex v we must find its different occurrences on the cycle (in the tree), split the tree in $d_v + 1$ sub-trees, reorder and join them. Unfortunately, when reordering and joining the sub-trees the order key is violated, because the position of vertices in the cycle changes after each rewire operation. We cannot update the search key of every node before merging the trees because it is expensive.

However, since we only care for the relative order of every vertex occurrence we can use the position in the cycle as an implicit key. Instead of explicitly having the occurrence position or index, we implicitly determine the position or index by the order of the nodes in the Binary Tree, for example, the first vertex corresponds to the left most leaf of the tree, and the last one corresponds to the right most leaf. In Figure 2.3 we show an example of a BST representation for the cycle T_1 from Example 1.1.

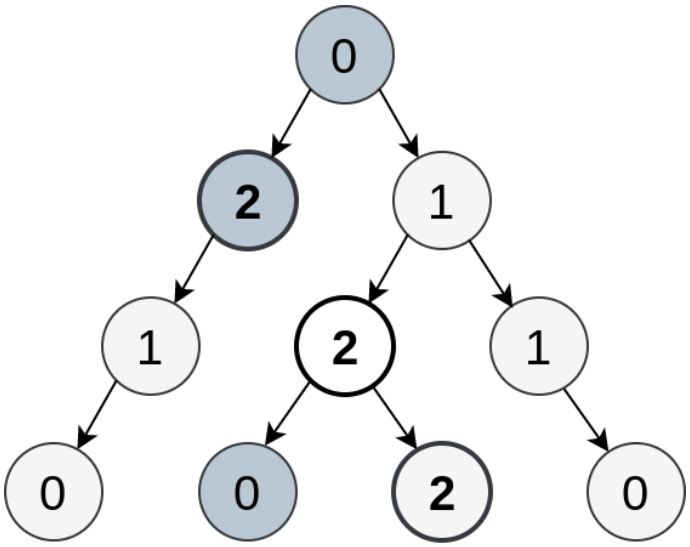

Fig. 2.3: A BST tree with implicit key representation of Euler cycle T_1 from Example 1.1

2.2.1 Finding the relative order

We need to tackle the sub-problem of finding the order of occurrences of a given vertex v in the cycle. In the tree representation of the cycle, the search key of the tree is the relative order but we need to search by vertex value (i.e. find all the occurrences of a vertex v). To avoid traversing all the structure as in the array implementation, we can use the previously mentioned idea of keeping an array of pointers to the BST nodes (occurrences) for every vertex. Now, we can calculate the relative order of the occurrences of a vertex v using two steps. First, a climbing step to calculate all the paths from the nodes to the root, and then, a DFS step to visit all the nodes in relative order (left first DFS).

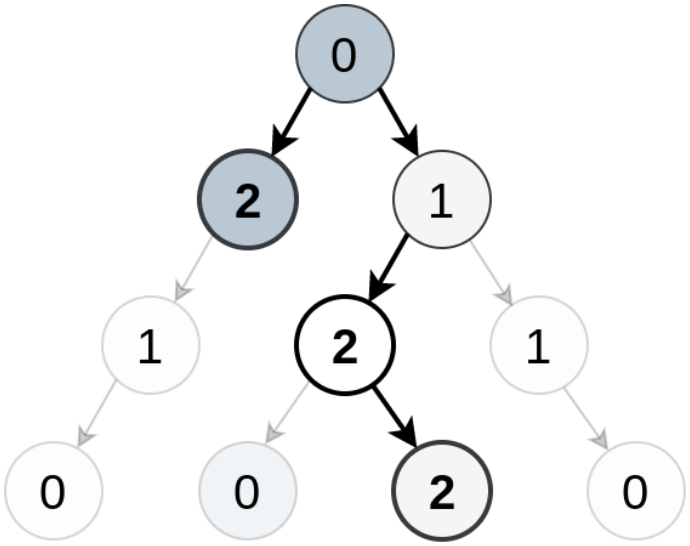

Fig. 2.4: Flagged nodes to perform the DFS step over occurrences of vertex 2 from Example 1.1

The climb step is described in Algorithm 2 and its time complexity is $\mathcal{O}(d_v * \log(|E|))$ (we use a balanced tree), because for every occurrence node of v, we climb at least $\log(E)$ times. In this step we must flag all the visited nodes to limit the search space of the DFS step, if we do not do that, the search space will be $\mathcal{O}(|E|)$ in the worst case and that will ruin the overall time complexity of the algorithm. We also have to flag the returning nodes to easily check in the next step if a node must be included in the result.

Algorithm 2

```
1: function ClimbStep(vertexNodes: [CycleNode], cycle: CycleTree)
2:     for n ∈ vertexNodes do
3:         n.return = true
4:         while n ≠ NILL and ¬n.visited do
5:             n.visited = true
6:             n = n.parent
7:         end while
8:     end for
9:     return DfsSort(cycle)
10: end function
```

The DFS traversal must clean all the flags written by the climb routine so we can perform other Climb and DFS operations later. As we want to sort the nodes in increasing order, we must visit the left child first. In a stack implementation, this translates into pushing the right child first. This implementation is described in Algorithm 3 and its running time is $\mathcal{O}(d_v * \log(|E|))$, because that is the complexity of the search space for the DFS algorithm.

Algorithm 3

```
1: function DfsSort(cycle : CycleTree)
2:     ret = emptyList
3:     stack = [cycle.root]
4:     while ¬stack.empty do
5:         current = stack.pop()
6:         if current ≠ NIL then
7:             current.visited = false
8:             if current.return then
9:                 ret.append(current)
10:                current.return = false
11:            end if
12:            stack.push(current.right)
13:            stack.push(current.left)
14:        end if
15:    end while
16:    return ret
17: end function
```

2.2.2 Splitting and joining operations

Once we have the relative order of the nodes, we can proceed to split the tree in sub-trees. Since we do not have an explicit key to split the trees, we once again need to use the pointers to each vertex nodes. Fortunately, those pointers were found by the previous section were we got the pointers sorted by their relative order. For simplicity, we split the tree by the relative order of the nodes, we start splitting the tree by the first node and so on. The time complexity of splitting a tree depends on the type of Binary Tree we choose. It is known that Red-Back trees and AVL trees support the splitting operation in logarithmic time. They also support the join operation of two trees t_1 y t_2 in $\mathcal{O}(h(t_1) - h(t_2))$ where h is the height of a tree [3]. Since we always join trees of at most size E, every merging operation done by the algorithm takes $\mathcal{O}(\log(|E|))$.

2.2.3 Proof of Theorem 1

For the ease of reading, we repeat here the statement of Theorem 1.

Theorem 1. *Given an Euler cycle of a digraph $G = (V, E)$ with minimum degree at least 3, there is an algorithm that finds a compatible Euler cycle in time $\mathcal{O}(|E| * \log(|V|))$.*

Proof. Consider the implementation of the Euler cycle as the previously described binary search tree with implicit key and an array of pointers for each vertex to its occurrences nodes. To perform a rewire operation on a vertex v, we must determine the order of the occurrences of v ($\mathcal{O}(d_v * \log(|E|))$), split the tree d_v times ($\mathcal{O}(d_v * \log(|E|))$), join the first tree with the last one ($\mathcal{O}(\log(|E|))$), then reversing the order of the trees ($\mathcal{O}(d_v)$))

and finally merging them ($\mathcal{O}(d_v * \log(|E|))$). So the cost to rewire all the n vertices is

$$\mathcal{O}(\sum_{v \in V} d_v * \log(|E|)) = \mathcal{O}(|E| * \log(|E|))$$

If we add the time of constructing the tree and the vertex pointers ($\mathcal{O}(|E|)$) and the time of traversing the tree to output the cycle ($\mathcal{O}(|E|)$), we get a total time complexity of $\mathcal{O}(|E| * \log(|E|)))$ which is equivalent to $\mathcal{O}(|E| * \log(|V|))$ because $|E|$ is at most $|V|^2$. □

2.3 Some improvements

After we perform a rewire operation on a vertex v, all the left neighbors occurrences of vertex v remain fixed through the next iterations. For that reason, we can blend each occurrence node of v to its respective left neighbor. For example, if we have already rewired vertex v and one of the occurrences of w is the left neighbor of one of the occurrences of vertex v, we can merge their respective nodes into a single node representing the two occurrences of v and w consecutively. This reduces the size of the structure where we store the current cycle, and it could reduce time complexity in the array implementation when the vertex degrees are unbalanced (if we rewire the one with highest degree first).

Also, we do not take advantage of the fact we can choose the order of rewires: it is cheaper to split a Binary Tree for the nodes nearest to the root. We can use a Treap data structure for this [11]. In this data structure every node has a search key and a priority, so the tree is a Binary Search Tree for the keys and a Heap for the priorities. In this problem, we can make same vertex occurrences have the same priority, so top priority occurrences will be on the top part of the tree. Combining this idea with previously mentioned idea of merging fixed, we can split and drop top priority vertices at each rewire operation in $\mathcal{O}(d_v)$ time. Unfortunately, join time complexity is still $\mathcal{O}(\log(|E|) * d_v)$, so the overall running time does not improve.

3. EXTENDING HAMILTONIAN CYCLES TO EULER CYCLES

We start with some properties of line graphs.

Proposition 3. *An Euler cycle of a digraph G corresponds to a Hamiltonian cycle of $L(G)$.*

Corollary 4. *Let G be a 2-eulerian (di)graph, for every Hamiltonian cycle H in $L(G)$, there exists another Hamiltonian cycle without a common edge.*

Proof. Let E be an Euler cycle of G. For every pair of consecutive edges of E we have a pair of adjacent vertices in $L(G)$. Therefore E corresponds to a Hamiltonian cycle in $L(G)$. □

Proposition 5. *If a digraph G is strongly connected, $L(G)$ is strongly connected.*

Proof. Let uv and wz be vertices from $L(G)$ corresponding to edges (u, v) and (w, z) in G. Since G is strongly connected, there is a path from v to w. Therefore there is a path that goes from a vertex of the form vx to a vertex of the form yw. Since uv has an edge to vx and yw has an edge to wz, there is a path from uv to wz. □

Proposition 6. *If G is an eulerian digraph with all vertices having same out-degree, then $L(G)$ is eulerian.*

Proof. Since G is strongly connected, so is $L(G)$. Let uv be a vertex from $L(G)$ corresponding to an edge (u, v) in G. The in-degree of uv corresponds to the in-degree of u and the out-degree of uv corresponds to the out-degree of v. Since G is eulerian and all out-degrees are the same, the in-degree of uv is the same as its out-degree. □

3.1 Proof of Theorem 2

For ease of reading we write again the statement of Theorem 2.

Theorem 2. *Let G be a 2-eulerian digraph with all vertices having the same out-degree and let H be a Hamiltonian cycle of $L(G)$. There exists an Euler cycle of $L(G)$ that extends H.*

Proof. By proposition 6, $L(G)$ is eulerian. By corollary 4, there exists a Hamiltonian cycle H' having no edge in common with H. Let $LG' = L(G) - H$ be the result of removing all the edges of H from $L(G)$. LG' contains a Hamiltonian cycle so it is strongly connected and every vertex of LG' has same in-degree to out-degree, therefore LG' is eulerian. Let E' be an Euler cycle of LG' and let $E = H + E'$ be the concatenation of the sequences of edges of H and E'. Then, E is an Euler cycle and extends H. □

Note that since H is a closed path, the first vertex of H visited in the Euler cycle extension E is the same as the last one.

3.2 An algorithm to extend a Hamiltonian cycle to an Euler cycle

It follows from the proof of the BEST theorem [13] that to generate an Euler cycle in a digraph we can use one of its spanning in-trees T.

Definition. *An in-tree is a directed tree with a vertex designated as root, in which other vertex than the root has out-degree exactly one.*

Definition. *A directed spanning-tree of a graph G is a directed tree that is a sub-graph of G and has all its vertices present in G.*

The algorithm traverses the digraph starting at the root of T and visits the vertices in a way that, when exiting a vertex, the edge belonging to T is not used until all other outgoing edges have been traversed. This algorithm runs in $\mathcal{O}(|E|)$ time.

Knowing that there exists an Hamiltonian cycle of H' without a common edge with H, we can drop an edge from H' to form T and use it as an spanning in-tree of $L(G)$. This gives us an extending algorithm:

In the first $|V|$ iterations we start at the root of T and enter and exit the vertices using the edges in H. In the following iterations we keep using the other edges following the rules of the previously defined method. Since none of the edges of T have been visited, we can ensure an Euler cycle is going to be generated. The complexity of this algorithm is $\mathcal{O}(|E|)$ plus the time of complexity of finding H'. Considering that H' is an Euler cycle in $G = (V, E)$ compatible with the Euler cycle H, Theorem 1 ensures that H' can be found in $\mathcal{O}(|E| * \log(|V|))$ operations. Thus, the algorithm that extends H into E takes at most $\mathcal{O}(|E| * \log(|V|))$ operations.

3.3 Applications

In this section we include some applications of Theorem 2 on extending some sequences that have a correspondence with Euler cycles in graphs.

3.3.1 Extending de Bruijn sequences

A (non-cyclic) de Bruijn sequence of order n in a k symbol alphabet is a sequence of length $k^n + n - 1$ such that every sequence of length n occurs exactly once as a consecutive substring [4, 6].

A de Bruijn graph of order n, which we denote by G_n, is a graph whose vertices are all sequences of length n, and the edges link overlapping sequences w, v such that the last $n - 1$ characters of w are equal to the first $n - 1$ characters of v. The edges of G_n can be labeled with sequences of length $n + 1$, such that the edge (w, v) is labeled with the concatenation of w and the last character of v. Then, each possible sequence of length $n + 1$ in k symbols appears in exactly one edge of G_n. Moreover, the line graph of G_n is exactly G_{n+1}. We can use the results for 2-eulerian graphs to give an alternative proof to the following theorem in [2].

Theorem (Becher and Heiber [2])**.** *Every de Bruijn sequence of order n in at least three symbols can be extended to a de Bruijn sequence of order $n + 1$.*

Proof. De Bruijn sequences of order n correspond exactly to the Hamiltonian cycles in de Bruijn graphs G_n. In turn, the Hamiltonian cycles in G_{n+1} are exactly the eulerian cycles in G_n. Every vertex of G_{n-1} has in-degree and out-degree at least three because the size of the alphabet and therefore G_{n-1} is 2-eulerian. Since G_n is the line graph of G_{n-1}, the conclusion follows from Theorem 2. □

Theorem 2 also provides a method to extend a de Bruijn sequence of order n to a de Bruijn sequence of order $n+1$. Since G_n has b^n vertices and b^{n+1} edges, this method has $\mathcal{O}(n * b^n + b^{n+1})$ time complexity. This alternative proof is interesting because its uses Theorem 2 in a way that can be applied to other de Bruijn variants.

3.3.2 Extending perfect necklaces

Perfect necklaces are a de Bruijn sequence variant defined by Alvarez, Becher, Ferrari and Yuhjtman in [1]: we call a necklace (k, n)-perfect for positive integers k and n, if each word of length k occurs exactly n times at positions which are different modulo n for any convention on the starting point.

Let A be an alphabet with cardinality b, let s be a word length and let n be a positive integer. The astute graph $G_{s,n}$ is the directed graph, with nb^s vertices where each vertex is a pair (u, v), where u is in A^s and v is a number between 0 and $n-1$. There is an edge from (u, v) to (u', v') if the last $s-1$ symbols from u coincide with the first $s-1$ symbols from u' and $(v+1) \mod n = v'$. Observe that $G_{s,n}$ is strongly regular (all vertices have in-degree and out-degree equal to b) and it is strongly connected (there is a path from every vertex to every other vertex).

Fact. *$G_{s,n}$ is the line graph of $G_{s-1,n}$.*

Proof. Every edge of $G_{s,n}$ from (u, v) to (u', v') is determined by the word u, the number v and the last character of u'. Therefore, it can be mapped 1-to-1 to a vertex of $G_{s+1,n}$. Two edges of $G_{s,n}$ are connected in $L(G_{s,n})$ if they have the form (x, y) and (y, w) where x, y, w are vertices from $G_{s,n}$, so every edge of $L(G_{s,n})$ is determined by the vertices (x, y) and the last character of the vertex w. Therefore every edge of $L(G_{s,n})$ corresponds to an edge of $G_{s+1,n}$ with matching corresponding start and end vertices. □

Fact. *Every (k, n)-perfect necklace of an alphabet of at least 3 letters can be extended to a $(k, n+1)$-perfect necklace.*

Proof. As we do with de Bruijn sequences, a (k, n)-perfect necklace corresponds to an Euler cycle in the astute graph $G_{k-1,n}$ which is 2-eulerian because it is strongly regular with degree at least 3. Also, $G_{k-1,n}$ is the line graph of $G_{k,n}$. The conclusion follows from Theorem 2. □

4. CONCLUSIONS AND FUTURE WORK

In this work we proposed a $\mathcal{O}(|E| * \log(|V|))$ algorithm to find compatible Euler cycles but we strongly believe there is an optimal ($\mathcal{O}(|E|)$) solution. As we mentioned before, our proposed algorithm does not leverage the fact we can choose in which order vertices are rewired. Also, our algorithm is a more a data structure based solution than a graph one, maybe with a graph theory approach we can get an optimal solution. It would be great to easily understand how the order of entering and exiting edges are affected when we rewire a vertex.

Moreover, the rewiring algorithm only works for eulerian graphs with minimum degree at least 3. It would be interesting to modify the algorithm so it works for arbitrary 2-eulerian graphs. For that, it would be helpful to have a characterization of such graphs. Regarding that family of graphs, it would also be interesting to extend the analysis to n-eulerian graphs to understand their properties. We leave these questions as open problems for future work.

BIBLIOGRAPHY

[1] Nicolás Alvarez, Verónica Becher, Pablo A. Ferrari, and Sergio A. Yuhjtman. Perfect necklaces. *Advances in Applied Mathematics*, 80(5):48 – 61, 2016.

[2] Verónica Becher and Pablo A. Heiber. On extending de Bruijn sequences. *Information Processing Letters*, 111(2):930–932, 2011.

[3] Guy E. Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '16, pages 253–264, New York, NY, USA, 2016. ACM.

[4] Nicolaas Govert de Bruijn. A combinatorial problem. *Nederl. Akad. Wetensch., Proc.*, 49(3):758–764 = Indagationes Math. 8, 461–467 (1946), 1946.

[5] Herbert Fleischner and Bill Jackson. Compatible euler tours in eulerian digraphs. In Geňa Hahn, Gert Sabidussi, and Robert E. Woodrow, editors, *Cycles and Rays*, pages 95–100. Springer Netherlands, Dordrecht, 1990.

[6] Camille Flye Sainte-Marie. Question 48. *L'interm. des math.*, 1(4):107–110, 1894.

[7] Harold Fredricksen. A survey of full length nonlinear shift register cycle algorithms. *SIAM Review*, 24:2:195–221, 1982.

[8] Irving John Good. Normal recurring decimals. *Journal of the London Mathematical Society*, s1-21(3):167–169, 1946.

[9] Frank Harary. *Graph theory.* Addison-Wesley Publishing Co., Reading, Mass.-Menlo Park, Calif.-London, 1969.

[10] Yaw-Ling Lin, Charles Ward, Bharat Jain, and Steven Skiena. Constructing orthogonal de Bruijn sequences. In Dehne F., Iacono J., and Sack JR. (eds), editors, *Algorithms and Data Structures. WADS 2011.*, volume 6844 of *Lecture Notes in Computer Science*, pages 595–606. Springer, 1 2011.

[11] Raimund Seidel and Cecilia R. Aragon. Randomized search trees. pages 540–545, 1996.

[12] William Tutte and Cedric A. B. Smith. On unicursal paths in a network of degree 4. *The American Mathematical Monthly*, 48(4):233–237, 1941.

[13] William T. Tutte. *Graph theory*, volume 21 of *Encyclopedia of Mathematics and its Applications.* Addison-Wesley Publishing Company, Advanced Book Program, Reading, MA, 1984. With a foreword by C. St. J. A. Nash-Williams.