Most Efficient Algorithm for finding an Euler Trail in a graph: Hierholzer vs. Fleury

Project for TRU Math 4430, Fall 2023

N.Nguimbous, R.Lundy, K.Yamanaka

December 3, 2023

1 Introduction

In this paper, we will compare and contrast Fleury's and Hierholzer's algorithm to find an Euler trail. And, we will implement Hierholzer's in code. Then, we will compare and conclude their efficiency based on time complexity.

2 Definitions and Properties

We now define the key terms we need for this paper. In our paper, we assume our graph is simple and undirected.

Definition 2.1. A trail that traverses every edge of G is called an Euler trail of G.

Definition 2.2. An Euler tour or Euler circuit is a closed walk that traverses each edge in exactly once.

Below are some properties to help us identify whether a graph has an Euler tour or trail, or neither of them don't exist. These will be used in our algorithm.

Theorem 2.3. *Let G be a non empty connected graph. Then G has Euler tour if and only if it has no vertex of odd degree.*

Corollary 2.4. *A connected graph G has an Euler trail if and only if it has at most 2 vertices of odd degree.*

3 Algorithms

We will introduce BFS/DFS which is used in Hierholzer's and Fleurry's algorithms.

3.1 BFS and DFS

Breadth-First Search (BFS) and Depth-First Search (DFS) are two fundamental graph traversal algorithms, each with distinct approaches and applications. BFS explores the graph by systematically examining all nodes at the present level before moving to deeper levels, utilizing a queue data structure. This method guarantees the shortest path between nodes but may consume more memory due to the queue. On the other hand, DFS probes the graph by diving as deeply as possible along each branch before backtracking, using a stack or recursion. It requires less memory compared to BFS but might not guarantee the shortest path. Both algorithms possess a time complexity of O(V + E), where V represents the number of vertices and E signifies the number of edges. This complexity arises from visiting each vertex and edge once during the traversal process, making them efficient for exploring and analyzing graphs and trees. We now introduce Hierholzer's algorithm. The main idea of Hierholzer's is to construct pairwise edge disjoint circuits in a graph, then we combine these circuits together using a common vertex to form an Euler circuit. We explain further in details.

4 Hierholzer's Algorithm

Algorithm 1: Hierholzer's Algorithm

Step 1 Select any vertex $v \in V$. Traverse outgoing edges from v until a circuit C_0 is encountered. Let $i = 0$.

Step 2 If the edges of circuit C_i represent the edges of G, then stop the algorithm since C_i is the Euler circuit of G. Otherwise, select a vertex v_i from the current circuit C_i such that v_i is incident to an edge not present in C_i, that is any edge from $G - E(C_i)$. Then, build another circuit C_i^* beginning with v_i on graph $G - E(C_i)$. Note, C_i^* also contains v_i.

Step 3 Build a new circuit C_{i+1} by merging the edges of C_i and C_i^*. This is done by beginning at v_i, traversing C_i, returning to v_i, then traversing C_i^* and returning to v_i. Now, set $i = i + 1$ and return to Step 2.

Let's take an example in figure 1. Notice that figure 1 satisfies Theorem 2.3, thus the graph contains Euler circuit. Start with $v1$, then our circuit is $C_0 = v1v0v5v1$. Note that $E(C_0) \neq E(G)$, thus start with $v1$ and build another $C_0^* = v1v2v4v1$. Now, merge C_0 and C_0^* to get a new circuit $C_1 = v1v2v4v1v0v5v1$. Again, our condition $E(C_1) \neq E(G)$, thus a new circuit is constructed $C_1^* = v2v5v4v3v2$. Finally, we merge C_1 and C_1^* to get our Euler circuit.

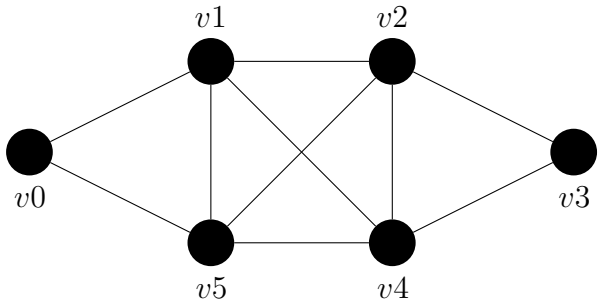


Figure 1

4.1 Time complexity

Time complexity for Hierholzer's is $O(E)$. In each step of the algorithm, when creating a new circuit, the algorithm traverses through a constant number of edges to form the circuit. Since each edge is traverse only once, total number of edges in G is $|E|$, the time complexity is linearly proportional to the number of edges.

One way to implement this is to prepare an array representing a trail and its elements are traversed edges that was used to form the subset circuits. And, DFS can be used to find unused edges to construct new subset edges.

5 Fleury's algorithm

Fleury's algorithm is named after the Swiss mathematician Carl Friedrich Wilhem Fleury, who introduced the algorithm in the 19^{th} century, precisely in 1883. As its peers, the

algorithm was developed to solve the seven bridges of Konigsberg problem posed by the mathematician Leonard Euler in 1736.

5.1 Algorithm

Algorithm 2: Fleury's Algorithm

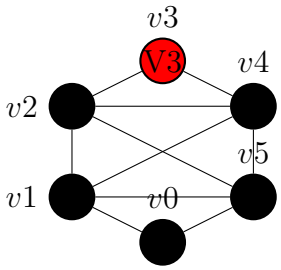
Data: Graph G
Result: Eulerian path or circuit in G
Function `Fleury(`*Graph* G`):`
- **Step 1:** Verify that G is connected and has either zero or two odd-degree vertices;
- **Step 2:** Choose the starting vertex;
- **Step 2.1: if** *G possesses 2-odd degree vertices* **then**
  - choose randomly one among them;
- **end**
- **else**
  - Choose randomly one vertex among all the vertices of G;
- **end**
- **Step 3: while** *There are remaining edges in G* **do**
  - **Step 3.1: if** *Current vertex has only one edge E* **then**
    - Traverse edge E to the next vertex V;
  - **end**
  - **Step 3.2: else if** *Current vertex has multiple edges* **then**
    - Choose an edge E from the current vertex that is not a cut-edge;
    - Traverse edge E to the next vertex V;
  - **end**
  - **Step 3.3:** Remove edge E from the graph G;
  - **Step 3.4:** Set the current vertex to V;
- **end**
- **Step 4:** Stop when all edges have been traversed;

5.2 Application

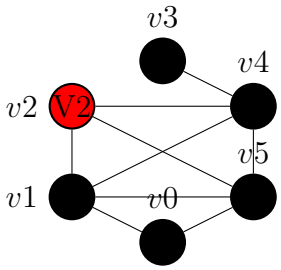
5.2.1 Eulerian path construction

- $\{v_3v_2\}$.
- $\{v_3v_2, v_2v_1\}$.

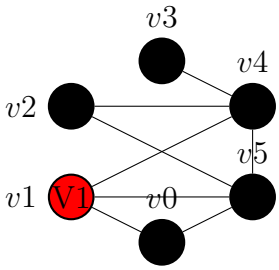
- $\{v_3v_2, v_2v_1, v_1v_0\}$.
- $\{v_3v_2, v_2v_1, v_1v_0, v_0v_5\}$.
- $\{v_3v_2, v_2v_1, v_1v_0, v_0v_5, v_5v_1\}$.
- $\{v_3v_2, v_2v_1, v_1v_0, v_0v_5, v_5v_1, v_1v_4\}$.
- $\{v_3v_2, v_2v_1, v_1v_0, v_0v_5, v_5v_1, v_1v_4, v_4v_2\}$.
- $\{v_3v_2, v_2v_1, v_1v_0, v_0v_5, v_5v_1, v_1v_4, v_4v_2, v_2v_5\}$.
- $\{v_3v_2, v_2v_1, v_1v_0, v_0v_5, v_5v_1, v_1v_4, v_4v_2, v_2v_5, v_5v_4\}$.
- $\{v_3v_2, v_2v_1, v_1v_0, v_0v_5, v_5v_1, v_1v_4, v_4v_2, v_2v_5, v_5v_4, v_4v_3\}$.



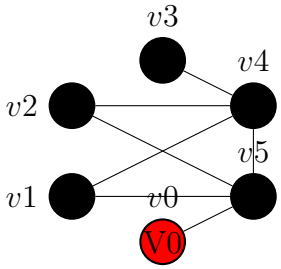
(a) Original graph



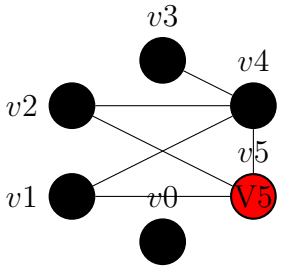
(b) step1



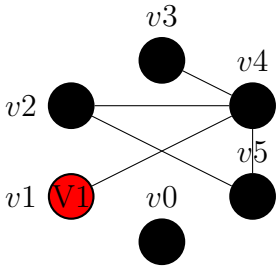
(c) step2



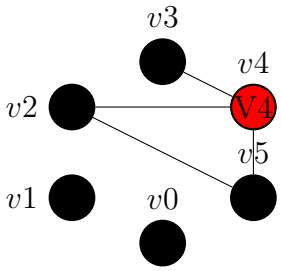
(a) step3



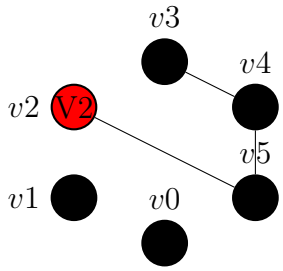
(b) step4



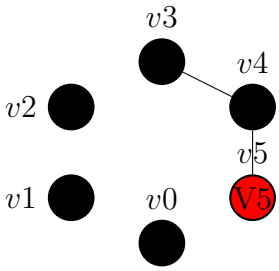
(c) step5



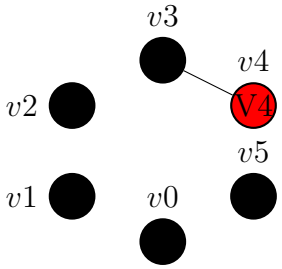
(a) step6



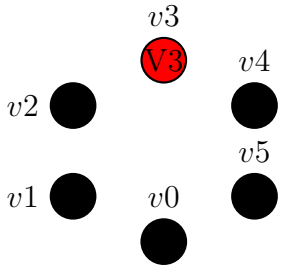
(b) step7



(c) step8



(a) step9



(b) step10

5.3 Time complexity

- **step 1:** requires to find a path to every vertex from the starting one. For that, we can use a graph traversal algorithm such as **BFS** or **DFS** which both have a complexity of $\mathcal{O}(E+V)$, with E and V the number of edges and vertices in the graph respectively.
- **step 2:** is edge and vertex independent. The complexity is $\mathcal{O}(1)$.
- **step 3:** The loop runs until all the edges are removed from the graph. The complexity is edge dependent and then $\mathcal{O}(E)$.
- **step 3.1:** is $\mathcal{O}(1)$ for the same reason as before.
- **step 3.2:** We check if an edge is a bridge or cut-edge. For that, we remove the corresponding edge from the graph and we check if the graph is still connected using **BFS** or **DFS** as before. The complexity for this step is then $\mathcal{O}(E+V)$.
- **step 3.3** and **step 3.4** are both $\mathcal{O}(1)$.
  Since step **3.1, 3.2, 3.3, 3.4** are both in the loop, the overall complexity of the loop is

$$\mathcal{O}(E \times (E+V+1+1)) \sim \mathcal{O}(E^2).$$

- **step 4** is $\mathcal{O}(1)$.

The algorithm's overall complexity is determined by the maximum complexity among its individual main steps

$$\mathcal{O}(G) = \mathcal{O}(\max(E^2, 1, E + V)).$$

5.4 Space complexity

The space complexity of the algorithm is $\mathcal{O}(E + V)$ since we have to store all the edges and vertices.

5.5 Limitations

Even though the Fleury's algorithm is elegant and straightforward, it is quite inneficient compare to other programs such as Hierholzer's algorithm.

6 Comparisons

Let's compare the above algorithms.

The main idea why Fleury's Algorithm is slower than Hierholzer's lies in the difference of edge exploration. In Fleury's, the process of visiting each edge once has a time complexity proportional to the number of edges in the graph, thus $O(E)$ is required. Then, it includes an additional step to check whether an edge is a cut edge. This condition requires $O(E)$ comparisons, which leads to $O(E^2)$ overall. In contrast, Hierholzer's does not necessitate this additional edge classification step. It only keeps track of the visited edges, then starts from a vertex that is incident to an unused edge and aims to find a new circuit. Thus, Fleury's is slower.

7 Hierholzer's implementation

Link to code: **Colab**.

7.1 Code

The algorithm itself is within the `euler_finder()` method, all other methods are helper methods. The beginning of the method (before the while loop) deals with acquiring a

starting vertex, using a random number generator to grab a random edge from the edge list (note: for graphs with odd vertices the list is restricted to those two numbers). The odd edges take O(E) (technically O(3E)) time to get, since it is just a method that iterates linearly over the vertex adjacency list (max O(2E) counting the number of vertices then saving the degree temporarily. It then iterates over that temporary degree list to count the number of odd vertices (O(E)). Once the initial vertex is added to the stack the process plays out as the algorithm's mathematical representation would indicate. It will continuously find an edge leading from the vertex at the top of the stack, adding the destination vertex to the stack, until it enters a vertex with no unused edges. It will then remove that vertex from the stack, adding it to the trail being built, and look at the new top vertex of `_stack`, continuing to utilise the same logic of exploring until exhaustion then popping onto the trail. In this way, it should eventually reach every edge in the connected graph. The program then returns what it believes is the trail. The proof of the algorithm is enough to show why (for an eligible graph) this process works, but the program does do a linear search through every edge in the edge list to see if it's "used" flag is True or False (it is looking for all True's since we want every edge used). This takes O(E) time, and there are some constant time logic that attempts to give some indication of why a failure may have occurred, ineligible number of odd edges or not all edges visited being the two currently coded in.

Most of the implementation details of the code are straightforward, but I did wish to highlight a few specific methods/data structures. The program doesn't store vertices per se. What it does do is represent each vertex as first layer index in a 2D list (eg vertex 3 is at index 3). At each index is an adjacency list for that vertex that lists only the vertices it is adjacent to. In this way we can forgo storing an adjacency matrix, which would take O(E2) to traverse. The edges themselves are simply stored as a list in the order they are added (ie they are not sorted). When an edge is created it passes it's list index location to the vertex list (which is a constant time add to the respective vertices, since we know their location). One of the common issues with implementing this method in code is how to deal with undirected edges. Since both vertices need to know they can utilise an edge, and it is unknown which vertex will be encountered first, the program must "double count" the edge for both vertices. This causes time complexity issues when it comes to then removing the edge at the destination vertex (since we don't know where in the adjacency list the reference will be and can't use an adjacency matrix as mentioned above). The solution was to create a separate pointer list that will iterate upwards every time an edge is chosen from the adjacency list in the vertex array. In this way the total search space for all vertices combined is reduced to 2*E (since the point can only go to the end of any one list, and all lists combined will always equal 2*E). Hence (unlike many popular online tutorials) our implementation of an undirected Hierholzer's method will be truly O(E), and not $O(E^2)$.

7.2 Question from Presentation Question

7.2.1 What happens if the naïve nature of the algorithm leads to a dead end/no out edge situation?

The first step is to lay out the case(s) where the algorithm could end up in such a situation. It is important to note that the code does not pop off a vertex from the current stack when it utilises the last edge, or in other words, it does not pop the `current_stack` as soon as the vertex runs out of unused incident edges. The code pops the current stack only when it enters an iteration of the while loop with a vertex without any unused edges to "escape" the vertex. It is also worth noting that a vertex can appear on either the `current_stack` or the trail list multiple times (since it is just a note of a visit).

Clearly, any even degree vertex (other than the initial vertex) will not cause the loop to get "stuck", as for every entry edge to the vertex, there is an exit edge. So even if such a vertex uses all it's edges, this just means it utilised an exit edge to move to another vertex (which is then the top of the stack). Thus such a vertex can never cause the stack to get "stuck". This vertex will eventually be popped once the loop hits an "endpoint" along the path that causes the stack to have a sort of mass pop (where the stack starts to roll back and keeps hitting vertices with already exhausted incident edges).

These "endpoints" are situations where the while loop ends up going into that iteration of the loop with a vertex with no unused "out" edges (which in an undirected graph means any unused edge). Or in other words, when an "origin" vertex utilises an "entry" edge that happens to be the last edge incident to the destination vertex. This can only happen in two instances:

7.2.2 Case 1: The vertex is the initial vertex.

Case 1.1 : If this vertex is of even degree, since it travels to the vertex for "free" it only utilises one edge to exit the vertex towards the rest of the graph, hence afterwards the vertex has $\geq 2n+1$ edges available (n could be 0). Thus there is an entry into the vertex where the vertex gets "stuck". This then starts to wrap up the graph in the case of a tour*. *If the initial vertex is of even degree then there are no odd degree vertices, and thus there is a tour.

Case 1.2 : If the initial vertex is of odd degree, then the initial exit from the vertex causes the vertex to become a de-facto "even" internal vertex and then it behaves as described above, or it is a 1 degree vertex in which case it will have to wait until the entire graph wraps up, and will be the last vertex added to the trail. This is why we "reverse" the trail

after it is finished, since for a directed graph it would lay out the reverse order of how the trail would have to be traversed, in an undirected graph it would still give a reversed order actual of visits, but since we can travel either direction the trail would still be a Eulerian Trail.

7.2.3 Case 2: The entered vertex is the odd vertex that is not the initial vertex.

Then it will have 1 more "entry" edge than "exit" edges (since we need to utilise one edge to visit it in the first place). After the last "entry" into that vertex the `current_stack` will then pop that vertex (since on the next iteration it cannot leave that vertex), and then the loop will resume the previously described behaviour until exhausting all edges.

If you have any further questions about the code feel free to reach out to Robert Lundy

References

[1] J. A. Bondy and U. S. R. Murty, Graph Theory With Applications , *American Elsevier Publishing Co., Inc., New York* (1976).

[2] H.Fleischner Algorithms in Graph Theory, *TU Wien, Algorithms and Complexity Group* (2016)

[3] S. P. Srivastava Study of Different algorithm in Euler Graph,*Dr RML Avadh University* (2016)