Algorothms' project : Detection of Eulerian path a.y. 2020/2021

Matteo Bandiera, Samuele Fonio

matteo.bandiera328@edu.unito.it, samuele.fonio@edu.unito.it

Contents

1 Abstract

Imagine a real case scenario in which an American rider has to plan a road trip through America's best known routes. He has the desire to ride all the roads and, if possible, to come back home, but in doing so he wants to avoid coming trough a route he has already driven. What is the ideal path for him to follow? The answer is of course a Eulerian circuit (avoiding the requirement of coming back home then it would be a Eulerian path).

We want to help the rider, so the problem we are going to address in this paper is the one of finding the Eulerian path given a connected and undirected graph. We recall first what an Eulerian path is, some historical notes about it and what are the conditions for which we are able to find one. Later we address the task of finding one through two of the best know algorithms available, namely Fleury's algorithm and Hierholzer's algorithm. Finally we will suggest the use of one of those based on the scenario, or technique which is the most efficient given a particular graph.

2 Introduction

The problem of finding a Eulerian path is well known since 18^{th} century, when Leonard Euler solved the *"Seven bridges of Koninsberg"*. What was the focus of this problem and how do they relate? The problem consisted, in today's words, in finding conditions for establishing the existence of a path in a graph that travels every edge of it exactly once, given the number of nodes and edges of which it was made of. Euler in 1736 solved the problem, stating that such a path exists if and only if the graph is connected and has zero or two nodes of odd degree(the number of edges at which that node is connected to others is an odd number). So the problem became spotting such a path using an algorithm. This contributed in the foundation of the branch of mathematics and computational sciences named Graph Theory, of which the topic we are interested is part of. Later we will address the task of finding an Eulerian path in a graph which satisfies Euler conditions.

3 Theory

For understanding the topic we used several books and articles:[5] and [1] for a graph general overview, [3] for the specific Eulerian path problem, [2] for the algorithmic part and finally, to see some real world application in biology, we read [4].
Eulerian path: in graph theory we call a Eulerian path a path which travels through the graph using exactly one time every edge connecting the vertices (we would call it a **Eulerian circuit** if the initial vertex and final vertex of our path coincide).
Connected graph: a connected graph is a graph in which for every vertex we are able to reach any other vertex.

Degree of a vertex: number of edges in which the vertex is involved.
Theorem A Eulerian path in a connected graph exists if and only if all of its vertices have even degree except , possibly, exactly two.

4 Graph representation

4.1 Setup 1

The `setup_1` was used for all the algorithms except the `hierholzer_deq()`. It represents the graph using an adjacency list. For the operations performed we noticed that it's not the optimal idea because a set would be preferable, but trying with the set implementation the recursive function showed some problems. Since the results were good enough we decided to keep our adjacency list implementation.
Here is the list of the most important functions present in the `setup_1`. Detailed information are in the Docstring of every function.

- `edges()` : Returns the list of all the edges. Since it's an undirected graph, here are present every edge repeated with inverted order.
- `single_edges()` : Returns the list of the single edges, without repetitions or inverted order.
- `DFScount()` : A Depth First Search based function which counts the vertices reachable through the set from the given vertex.
- `degree()` : Fundamental function used to check if the graph is suitable to have a Eulerian path. It returns a dictionary with vertices and their degree.
- `is_connected()` : Another function involved in the check for the suitability of the graph. It returns if the graph is connected or not.
- `ep_check()` :Final check if the graph is suitable under the conditions of the theorem.
- `hierholzer_rec()` :The recursive version of Hierholzer's algorithm.
- `hierholzer_it()` :The iterative version of Hierholzer's algorithm.
- `fleury_rec()` :The recursive version of Fleury's algorithm.
- `fleury_it()` :The iterative version of Fleury's algorithm.
- The previous 4 functions with an ending 2 in the name of hte algorithm (e.g. `hierholzer2_rec`) are the ones which check if the graph is suitable, change the starting point if needed and run the algorithm.

4.2 Setup 2

The `setup_2` was used for the fifth implementation. We wanted to reach a good level of performance using this one, so in order to achieve it we used an adjacency set implementation.
Anyway the most exotic element of this setup is the `Deque` data type that is a queue for which it is possible to append and to pop from both the ends in constant time. It was implemented as a double-ended queue using the class `TwoWayNode`. Here is the list of the most important functions:

- `Deque.get_front()` : returns the front node of the queue.
- `Deque.is_empty()` : returns if the queue is empty.
- `Deque.enqueue_front()` : Enqueue from the front of the queue.
- `Deque.enqueue_back()` : Enqueue from the back of the queue.
- `Deque.pop()` : Pop the last element in front. There is not a function that pop the element from the left end since it was not needed.
- `Deque.rotate()` : Takes an element from the front of the queue and put it at the back end of the queue.
- `h_check()` :It's the algorithm check, it's different from the `ep_check()` of the `setup_1` since this implementation is for graphs with only even degree vertices.
- `hierholzer_deq()` : The Hierholzer's algorithm implemented with the deques.

5 Fleury's Algorithm

5.1 Definition and time complexity

Dated back to 1883, the Fleury's algorithm is the first answer to our question, so let us see in detail what this algorithm does:

1. Given a connected graph, with 0 or 2 odd degree vertices, for which we know a Eulerian path exists , it starts at an odd degree vertex ,or in case it has none, at an arbitrary vertex.
2. Then it chooses the next vertex to travel to, based on the fact that the deletion of the edge connecting each other would preserve the connectedness of the graph, unless we have a forced choice of having just one to connect to.

3. It deletes the edge previously chosen and iterates this process until there are no more edges to be canceled.

4. The Eulerian path is determined by the collection of the edges visited.

Now we address the task of calculating the class of complexity of which this algorithm belongs to. The algorithm (and of course its implementation) is based on the so called bridge-detection function which consists in checking if removing that particular edge would disconnect the graph or not, meaning that we would still be able to go through all the edges once deleted. This function is linear in time and is performed on all the edges, which brings the whole complexity of our algorithm to a quadratic one, i.e. $O(n^2)$ (where n=number of edges).
Concerning the space complexity, our implementations are done using arrays and dictionaries, which means that the most expensive thing in terms of space is the starting dictionary of the graph, having $O(n * v)$ complexity (where v=number of vertices).
However we have to remember the bridge-detection, that is the real problem of this algorithm: at every step it creates two dictionaries of v complexity.

5.2 Implementation

We implemented two versions of Fleury's algorithm, and we followed its definition step by step. Since the linearity of the bridge-detection function is reached only under a specific setup, which we did not recreate, and since the *setup_1* is not the best, in our implementation this function is not linear. However, it performed satisfying results.
We used a Depth First Search principle and here is the function:

```
def isValidNextEdge(self, u, v):

        """
  \\ Function that returns if an edge is valid for the path
  \\ built by Fleury's algorithm, i.e. it does not
  \\ disconnect the graph.
  \\ input: - u,v: vertices of interested edge
  \\ output: True if it's valid, i.e. it does not disconnect
  \\ the graph.
        """
      if len(self.graph[u]) == 1:
            return True
      if len(self.graph[u]) ==0:
            return False
      else:
        visited ={v:False for v in self.vertices()}
        count1 = self.DFSCount(u, visited)
        self.removeEdge(u, v)
        visited ={v:False for v in self.vertices()}
        count2 = self.DFSCount(u, visited)
        self.addEdge(u,v)
        return False if count1 > count2 else True
```

The two versions of Fleury's algorithm are recursive and iterative. In their construction they are very similar, but the quadratic complexity arise quite clearly from the iterative version. We will compare them better in the time analysis.

Here there are the two implementations:

```python
def fleuryUtil_it(self,u,l):
        """
  This iterative function returns a Eulerian Path
  when possible, starting from vertex u.
  u: starting vertex, chosen by self.fleury(l)
  l: Must be initialized by len(self.single_edges())
  output: list of edges.
        """
        final=[]
        v=list(self.graph[u])[0]
        while len(final)!=l:
            if len(self.graph[u])==0:
                print("here")
                return(final)
            i=0
            v=list(self.graph[u])[i]
            while self.isValidNextEdge(u,v)==False:
                i+=1
                v=list(self.graph[u])[i]
            final.append((u,v))
            self.removeEdge(u,v)
            u=v
        return(final)
```

```python
def fleuryUtil_rec(self,u,final=[]):

        """
  Function (recursive) to return the Eulerian path folllowing the
    Fleury's algorithm. The check will be done using another
    function.
  input: - u : starting vertex
       - final=[] : final path
  output: list of edges
        """

        for v in self.graph[u]:
            if self.isValidNextEdge(u,v):
                final.append((u,v))
                self.removeEdge(u,v)
                self.fleuryUtil_rec(v,final)
        return(final)
```

6 Hierholzer's Algorithm

6.1 Definition and time complexity

Proposed in 1873 , the Hierholzer's Algorithm faces almost the same task , but in a slightly different fashion, let us see the details :

1. It starts at an arbitrary vertex and then finds its moves through the edges and vertices making a circuit such that the starting vertex and end vertex are the same (we remark the fact that such a circuit exists for the conditions we have specified before, and it does not coincide with the path we are interested, because it could consist of just some of the vertices and some of the edges).

2. We start again by a vertex, being part of the previously found circuit, for which there is a connection with an edge that does not belong to the circuit and we iterate the process described in point 1.

3. Iterating the process described in point 2 until there are no edges left to visit, we will exhaust the number of iterations possible and are left with the collection of circuits that will lead us to the Eulerian path we were looking for.

This algorithm is valid for Eulerian circuits, so the condition is that all the vertices must have even degree. We implemented a little variation that finds Eulerian path with the theorem conditions, so two or zero vertices with odd degree and it must start from one of them if present. From now on we will call it *general Hierholzer*. This variation requires the modification of point one, in which we do not require the same starting and ending vertex.
Concerning the class of complexity , our variation performs better than Fleury's in searching for a Eulerian path. This is due to the fact that in our implementation of Hierholzer's algorithm we basically look for loops ,so we do not have to make the previous check for the disconnection of the graph (we have already said that this task is linear in time). This brings the whole complexity of the Hierholzer's algorithm to (ideally) $O(n)$.
Concerning the space complexity this algorithm is theoretically more expensive than Fleury's one. In fact at each step we have to remember which are the unvisited edges and which vertices have unvisited edges. For the first one the complexity is $2 * n$, since the function used is the `edges()` in `setup_1` , while the second is solved using the graph representation. At each steps the implementations update the elements and do not create new ones (as Fleury does). So again, the most expensive element is the graph.

6.2 Implementation

As before we implemented two versions of the general Hierholzer, one recursive and one iterative. Since the setup was always the `setup_1` , the performance problems are latent, and since for the objects used during the algorithm (e.g.

set of unvisited edges, visited vertices, available edge,...) we did not use different data types, the complexity of our algorithm is not exactly linear, but this is due only to the update phase, in which the operations are not constant as the best form of this algorithm would require. From the code we can see clearly that the complexity is linear.
As before the two implementations are similar and we will compare later their performances. Here you can find them.

```python
def hierholzer_rec(self, start, unvisited,index=0,final=[]):
        """
Function to return the Eulerian path. Note, the check for
the existence of an Eulerian path will be done using  another
    function.
The graph traversal is based directly on the graph.
    input: -start: starting vertex
           -unvisited: list of unvisited edges (to start they
           are all unvisited, it will be initialized by
            self.edges())
           -index:used to join the missed circles in the final tour
           -final: refers to the visited edges
    output: list of edges.
        """
        if len(self.graph[start])!=0 and len(unvisited)>0:
            current=(start,self.graph[start][0])
            final.insert(index,current)
            index+=1
            current_inverted=(current[1],current[0])
            self.removeEdge(start,current[1])
            unvisited.remove(current)
            unvisited.remove(current_inverted)
            start=current[1]
            self.hierholzer_rec(start,unvisited,index,final)
        elif len(self.graph[start])==0 and len(unvisited)>0:
            i=len(final)-1
            while len(self.graph[final[i][1]])==0:
                if i==0:
                    break
                i-=1
            if i==0 and len(self.graph[final[i][1]])==0:
                index=0
                start=final[0][0]
                self.hierholzer_rec(start,unvisited,index,final)
            else:
                index=i+1
                start=final[i][1]
                self.hierholzer_rec(start,unvisited,index,final)
        return(final)
```

```python
def hierholzer_it(self,start):

        """
        Function (iterative) to return the Eulerian path. Note, the check for the existence of an Eulerian path will be done using another function.
        The graph traversal is based directly on the graph.
        input: -start: starting vertex
        output: Eulerian path that traverse every edge just once.
        """

        unvisited=self.edges()
        final=[]
        current=(start,self.graph[start][0])

        index=0


        while len(unvisited)!=0:


            final.insert(index,current)
            index+=1

            self.removeEdge(current[0],current[1])
            unvisited.remove(current)
            unvisited.remove((current[1],current[0]))

            start=current[1]


            if len(unvisited)==0:
                return(final)

            if (len(self.graph[start])==0) and len(unvisited)==0:
                final.insert(index,current)
                return(final)

            if (len(self.graph[start])==0) and len(unvisited)>0:
                i=len(final)-1

                while len(self.graph[final[i][1]])==0:

                    if i==0:
                        break
                    i-=1
                if i==0 and len(self.graph[final[i][1]])==0:

                    index=0
                    start=final[0][0]
                else:
                    index=i+1
                    start=final[i][1]
            current=(start,self.graph[start][0])
        return (final)
```

6.3 Deque implementation

We have talked about the problem of `setup_1` and the non perfect performance of our algorithms, but we did not stop at all in front of them. In fact our last implementation is a real Hierholzer's algorithm and it's based on all the data types needed to have a better performance than the previous ones. However we have to take into account different aspects. First of all, this deque implementation works only on graphs with all even degree vertices. Furthermore it does not start from the given vertex. Now we are going to explain these two aspects, but first of all let's see its very simple process.

1. The algorithm starts from an arbitrary vertex, and append every edge in a deque, following the oorder already explained in the Hierholzer's algorithm.
2. It can happen, as said before, that we get *stuck*. This happens when we reach the starting vertex (without any exiting non visited edges) and have other edges to explore.
3. at this point we *rotate* as explained in the `setup_2` until unstuck, i.e. we meet a vertex that is part of unvisited edges. From this it starts certainly a circuit, because of its even degree.
4. Repeating the previous points, we finish the graph.

The two aspects highlighted before are now clear: without all even degree vertices we could start from a vertex that is not part of a circuit, which means that there are cases in which it never get unstuck. And it does not start from the given vertex because of the rotations. Of course we could re-rotate the deque, but it would be inefficient.
Since for a deque append and pop are constant in time, the rotation is constant, and since we need to do at most n rotation, the algorithm is clearly linear, $O(n)$. Furthermore we are using abstract data types, which improves the space complexity. Here is the code.

```
def hierholzer_deq(self,start):
            """
            Function that uses a deque in order to get an Eulerian path (cycle) putting together many Eulerian cycles.
            The starting point is irrelevant, since during the running of the algorithm the deque rotates and it does not
            rerotate.
            The start is specified anyway, since for testing it we needed to change the starting point and see if it worked.
            Input: - Graph
                   - start : starting vertex
            output: Eulerian cycle
            """
            unvisited=self.edges()
            final=Deque()
            current=(start,list(self.graph[start])[0])
```

```
        while len(unvisited)!=0:
            final.enqueue_front(current)
            unvisited.discard(current)
            unvisited.discard((current[1],current[0]))
            self.removeEdge(current[0],current[1])
            new=current[1]
            if len(unvisited)==0:
                return (final)
            if (len(self.graph[new])==0) and len(unvisited)>0:
                t=final.get_front()
                while len(self.graph[t[1]])==0:
                    final.rotate()
                    t=final.get_front()
                new=t[1]
            current=(new,list(self.graph[new])[0])
        return
```

7 Comparison and conclusion

In this last section we are going to compare the time results we produced. For the comparison we used the `%timeit` magic function. However, since the algorithms delete the graph while running, we had to redefine it all the times. So the results shown are a sum of the three function: definition of the graph, check for suitability and algorithm. Since the operations are the same for the algorithms, the results are comparable.
What do we expect? We expect that *general Hierholzer* performs better than *Fleury* in both of the versions, and we expect the `hierholzer_deq` to be the best by far.
Here are the results:

Fleury - Recursive	164 µs ± 11 µs per loop
Hierholzer - Recursive	127 µs ± 2.33 µs per loop
Fleury - Iterative	176 µs ± 13.9 µs per loop
Hierholzer - Iterative	134 µs ± 10.1 µs per loop
Hierholzer - Deque	77.5 µs ± 1.52 µs per loop

Table 1: $n = 14$,mean ± std. dev. of 7 runs, 10000 loops each

Everything as expected.
As said before, our implementations are not the best. And this is clear if we watch the results: the difference between Fleury and Hierholzer should be more important. However we can be quite satisfied for the implementation of `hierholzer_deq`, which shows interesting results. The difference between Recursive and iterative versions are not important.
To conclude, we can say that in general we prefer the Hierholzer's algorithm even if Fleury's is more elegant. The use of a "good" Hierholzer's would perform very useful results, both in time and space complexity. Adding a not so strict

condition for the degree of the vertices (2 vertices with odd degree is not an important change from 0 vertices with odd degree), we can (and we did) get effectively performing results.

Sitography

https://en.wikipedia.org/wiki/Eulerian_path

References

[1] Norman Biggs, E Keith Lloyd, and Robin J Wilson. *Graph Theory, 1736-1936.* Oxford University Press, 1986.

[2] Thomas H. Cormen et al. *Introduction to algorithms.* MIT press, 2009.

[3] Ashish Kumar. "A study on Euler Graph and it' s applications". In: *Int. J. Math. Trends Technol* 43.1 (2017), pp. 9–14.

[4] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. "An Eulerian path approach to DNA fragment assembly". In: *Proceedings of the national academy of sciences* 98.17 (2001), pp. 9748–9753.

[5] Robin J. Wilson. "Introduction to Graph Theory". In: ().