Previous Next

332. Reconstruct Itinerary

Feb. 19, 2020 | 22.3K views

Given a list of airline tickets represented by pairs of departure and arrival airports `[from, to]`, reconstruct the itinerary in order. All of the tickets belong to a man who departs from `JFK`. Thus, the itinerary must begin with `JFK`.

Note:

1. If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string. For example, the itinerary `["JFK", "LGA"]` has a smaller lexical order than `["JFK", "LGB"]`.
2. All airports are represented by three capital letters (IATA code).
3. You may assume all tickets form at least one valid itinerary.
4. One must use all the tickets once and only once.

Example 1:

```
Input: [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]
Output: ["JFK", "MUC", "LHR", "SFO", "SJC"]
```

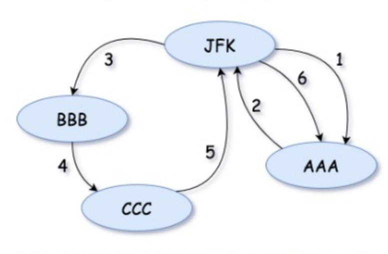
Example 2:

```
Input: [["JFK","SFO"],["JFK","ATL"],["SFO","ATL"],["ATL","JFK"],["ATL","SFO"]]
Output: ["JFK","ATL","JFK","SFO","ATL","SFO"]
Explanation: Another possible reconstruction is ["JFK","SFO","ATL","JFK","ATL","SF
O"].
             But it is larger in lexical order.
```

Solution

Overview

Overall, we could consider this problem as a **graph traversal** problem, where an airport can be viewed as a *vertex* in graph and flight between airports as an *edge* in graph.



We would like to make a few clarification on the input of the problem, since it is not clear in the description of the problem.

As one might notice in the above example, the input graph is NOT what we call a **DAG** (*Directed Acyclic Graph*), since we could find at least a cycle in the graph.

In addition, the graph could even have some duplicate edges (*i.e.* we might have multiple flights with the same origin and destination).

Approach 1: Backtracking + Greedy

Intuition

As common strategies for problems of graph traversal, we often apply the methodologies of **backtracking** or **greedy**. As it turns out, we can apply both of them for this problem.

> Typically, backtracking is used to enumerate all possible solutions for a problem, in a trial-fail-and-fallback strategy.

At each airport, one might have several possible destinations to fly to. With backtracking, we enumerate each possible destination. We mark the choice at each iteration (*i.e.* trial) before we move on to the chosen destination. If the destination does not lead to a solution (*i.e.* fail), we would then *fallback* to the previous state and start another iteration of trial-fail-and-fallback cycle.

> A greedy algorithm is any algorithm that follows the problem-solving *heuristic* of making locally optimal choice at each step, with the intent of reaching the global optimum at the end.

As suggested by its definition, a greedy algorithm does not necessarily lead to a globally optimal solution, but rather a reasonable approximation in exchange of less computing time.

Nonetheless, sometimes it is the way to produce a global optimum for certain problems. This is the case for this problem as well.

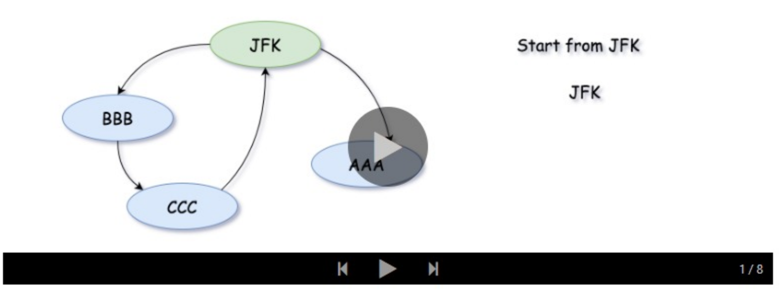
At each airport, given a list of possible destinations, while backtracking, at each step we would pick the destination ***greedily*** in lexical order, *i.e.* the one with the smallest lexical order would have its trial first.

With this ***greedy*** strategy, we would ensure that the final solution that we find would have the *smallest lexical order*, because all other solutions that have smaller lexical order have been trialed and failed during the process of backtracking.

Algorithm

Here we explain how we implement a solution for this problem, by combining the strategies of backtracking and greedy.

- As the first step, we build a graph data structure from the given input. This graph should allow us to quickly identify a list of potential destinations, given an origin. Here we adopted the hashmap (or dictionary) data structure, with each entry as `<origin, [destinations]>`.
- Then due to our greedy strategy, we then should order the destination list for each entry in lexical order. As an alternative solution, one could use `PriorityQueue` data structure in the first step to keep the list of destinations, which would maintain the order at the moment of constructing the list.
- As the final step, we kick off the backtracking traversal on the above graph, to obtain the final result.
  - At the beginning of the backtracking function, as the bottom case, we check if we have already obtained a valid itinerary.
  - Otherwise, we enumerate the next destinations in order.
  - We mark the status of visit, before and after each backtracking loop.



Note that there is certain code pattern that one can follow in order to implement an algorithm of backtracking. We provide an example in the Explore card of Recursion II.

Java Python Copy

```python
class Solution(object):
    """ """
    def findItinerary(self, tickets):
        """
        :type tickets: List[List[str]]
        :rtype: List[str]
        """
        from collections import defaultdict
        self.flightMap = defaultdict(list)

        for ticket in tickets:
            origin, dest = ticket[0], ticket[1]
            self.flightMap[origin].append(dest)

        self.visitBitmap = {}

        # sort the itinerary based on the lexical order
        for origin, itinerary in self.flightMap.items():
        # Note that we could have multiple identical flights, i.e. same origin and destination.
            itinerary.sort()
            self.visitBitmap[origin] = [False]*len(itinerary)

        self.flights = len(tickets)
        self.result = []
        route = ['JFK']
        self.backtracking('JFK', route)
```

Complexity

- Time Complexity: $\mathcal{O}(|E|^d)$ where $|E|$ is the number of total flights and d is the maximum number of flights from an airport.
  - It is tricky to estimate the time complexity of the backtracking algorithm, since the algorithm often has an early stopping depending on the input.
  - To calculate a loose upper bound for the time complexity, let us consider it as a combination problem where the goal is to construct a sequence of a specific order, *i.e.* $|V_1 V_2 ... V_n|$. For each position V_i, we could have d choices, *i.e.* at each airport one could have at most d possible destinations. Since the length of the sequence is $|E|$, the total number of combination would be $|E|^d$.
  - In the worst case, our backtracking algorithm would have to enumerate all possible combinations.
- Space Complexity: $\mathcal{O}(|V| + |E|)$ where $|V|$ is the number of airports and $|E|$ is the number of flights.
  - In the algorithm, we use the graph as well as the visit bitmap, which would require the space of $|V| + |E|$.
  - Since we applied recursion in the algorithm, which would incur additional memory consumption in the function call stack. The maximum depth of the recursion would be exactly the number of flights in the input, *i.e.* $|E|$.
  - As a result, the total space complexity of the algorithm would be $\mathcal{O}(|V| + 2 \cdot |E|) = \mathcal{O}(|V| + |E|)$.

Approach 2: Hierholzer's Algorithm

Eulerian Cycle

> In graph theory, an Eulerian trail (or **Eulerian path**) is a trail in a finite graph that visits every edge exactly once (allowing for revisiting vertices).

In our problem, we are asked to construct an itinerary that uses all the flights (edges), starting from the airport of "JFK". As one can see, the problem is actually a variant of Eulerian path, with a fixed starting point.

> Similarly, an Eulerian circuit or **Eulerian cycle** is an Eulerian trail that starts and ends on the same vertex.

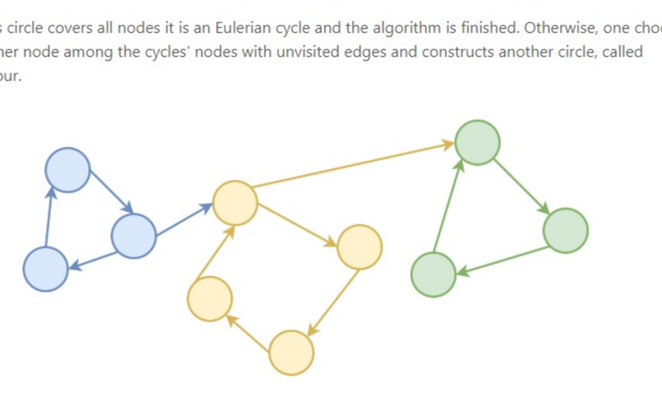
The Eulerian cycle problem has been discussed by Leonhard Euler back in 1736. Ever since, there have been several algorithms proposed to solve the problem.

In 1873, Hierholzer proposed an efficient algorithm to find the Eulerian cycle in linear time ($\mathcal{O}(|E|)$). One could find more details about the Hierholzer's algorithm in this course.

> The basic idea of Hierholzer's algorithm is the stepwise construction of the Eulerian cycle by connecting disjunctive circles.

To be more specific, the algorithm consists of two steps:

- It starts with a random node and then follows an arbitrary unvisited edge to a neighbor. This step is repeated until one returns to the starting node. This yields a first circle in the graph.
- If this circle covers all nodes it is an Eulerian cycle and the algorithm is finished. Otherwise, one chooses another node among the cycles' nodes with unvisited edges and constructs another circle, called subtour.



By connecting all the circles in the above process, we build the Eulerian cycle at the end.

Eulerian Path

> To find the Eulerian path, inspired from the original Hierzolher's algorithm, we simply change one condition of loop, rather than stopping at the starting point, we stop at the vertex where we do not have any unvisited edges.

To summarize, the main idea to find the Eulerian path consists of two steps:

- Step 1). Starting from any vertex, we keep following the unused edges until we get **stuck** at certain vertex where we have no more unvisited outgoing edges.
- Step 2). We then backtrack to the nearest neighbor vertex in the current path that has unused edges and we **repeat** the process until all the edges have been used.

The first vertex that we got stuck at would be the **end point** of our Eulerian path. So if we follow all the stuck points backwards, we could reconstruct the Eulerian path at the end.

Algorithm

Now let us get back to our itinerary reconstruction problem. As we know now, it is a problem of Eulerian path, except that we have a fixed starting point.

More importantly, as stated in the problem, the given input is guaranteed to have a solution. So we have one less issue to consider.

As a result, our final algorithm is a bit simpler than the above Eulerian path algorithm, without the backtracking step.

> The essential step is that starting from the starting vertex (airport 'JFK'), we keep following the ordered and unused edges (flights) until we get **stuck** at certain vertex where we have no more unvisited outgoing edges.

The point that we got stuck would be the last airport that we can visit. And then we follow the visited vertex (airport) **backwards**, we would obtain the final itinerary.

Here are some sample implementations which are inspired from a thread of discussion in the forum.

Java Python Copy

```python
class Solution(object):
    def findItinerary(self, tickets):
        """
        :type tickets: List[List[str]]
        :rtype: List[str]
        """
        from collections import defaultdict
        self.flightMap = defaultdict(list)

        for ticket in tickets:
            origin, dest = ticket[0], ticket[1]
            self.flightMap[origin].append(dest)

        # sort the itinerary based on the lexical order
        for origin, itinerary in self.flightMap.items():
        # Note that we could have multiple identical flights, i.e. same origin and destination.
            itinerary.sort(reverse=True)

        self.result = []
        self.DFS('JFK')

        # reconstruct the route backwards
        return self.result[::-1]

    def DFS(self, origin):
        destList = self.flightMap[origin]
        while destList:
```

Discussion

To better understand the above algorithm, we could look at it from another perspective.

> Actually, we could consider the algorithm as the **postorder DFS** (Depth-First Search) in a directed graph, from a fixed starting point.

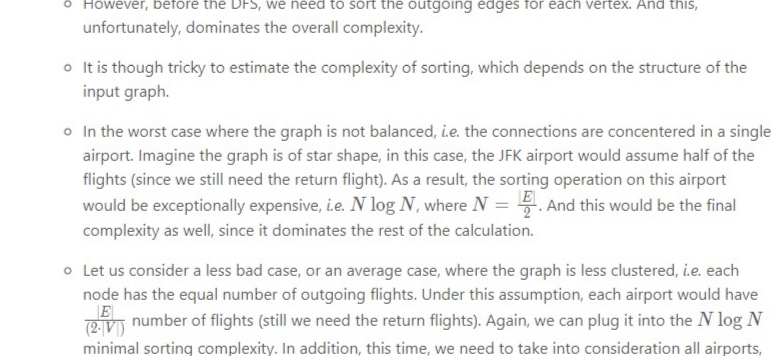
As we know that, each input is guaranteed to have a solution. Therefore, the task of the problem can be interpreted as that given a list of flights (*i.e.* edges in graph), we should find an order to use each flight *once and only once*.

In the resulted path, before we visit the last airport (denoted as `V`), we can say that we have already used all the rest flights, *i.e.* if there is any flight starting from `V`, then we must have already taken that before.

> Or to put it another way, before adding the last airport (vertex) in the final path, we have visited all its **outgoing** vertex.

Actually, the above statement applies to each airport in the final itinerary. *Before adding an airport into the final itinerary, we must first visit all its outgoing neighbor vertex.*

If we consider the outgoing vertex in a directed graph as children nodes in a tree, one could see the reason why we could consider the algorithm as a sort of **postorder DFS traversal** in a tree.



Complexity

- Time Complexity: $\mathcal{O}(|E| \log \frac{|E|}{|V|})$ where $|E|$ is the number of edges (flights) in the input.
  - As one can see from the above algorithm, during the DFS process, we would traverse each edge once. Therefore, the complexity of the DFS function would be $|E|$.
  - However, before the DFS, we need to sort the outgoing edges for each vertex. And this, unfortunately, dominates the overall complexity.
  - It is though tricky to estimate the complexity of sorting, which depends on the structure of the input graph.
  - In the worst case where the graph is not balanced, *i.e.* the connections are concentered in a single airport. Imagine the graph is of star shape, in this case, the JFK airport would assume half of the flights (since we still need the return flight). As a result, the sorting operation on this airport would be exceptionally expensive, *i.e.* $N \log N$, where $N = \frac{|E|}{2}$. And this would be the final complexity as well, since it dominates the rest of the calculation.
  - Let us consider a less bad case, or an average case, where the graph is less clustered, *i.e.* each node has the equal number of outgoing flights. Under this assumption, each airport would have $\frac{|E|}{(2 \cdot |V|)}$ number of flights (still we need the return flights). Again, we can plug it into the $N \log N$ minimal sorting complexity. In addition, this time, we need to take into consideration all airports, rather than the superhub (JFK) in the above case. As a result, we have $|V| \cdot (N \log N)$, where $N = \frac{|E|}{2 \cdot |V|}$. If we expand the formula, we will obtain the complexity of the average case as $\mathcal{O}(\frac{|E|}{2} \log \frac{|E|}{2 \cdot |V|}) = \mathcal{O}(|E| \log \frac{|E|}{|V|})$
- Space Complexity: $\mathcal{O}(|V| + |E|)$ where $|V|$ is the number of airports and $|E|$ is the number of flights.
  - In the algorithm, we use the graph, which would require the space of $|V| + |E|$.
  - Since we applied recursion in the algorithm, which would incur additional memory consumption in the function call stack. The maximum depth of the recursion would be exactly the number of flights in the input, *i.e.* $|E|$.
  - As a result, the total space complexity of the algorithm would be $\mathcal{O}(|V| + 2 \cdot |E|) = \mathcal{O}(|V| + |E|)$.

Rate this article:

Previous Next

Comments: 24 Sort By

Type comment here... (Markdown is supported)
Preview Post

1jeanpaul1 ★ 28 June 10, 2020 4:18 AM
I enjoyed this problem a lot. It really makes you think. I didn't quite get it, my DFS implementation wasn't getting it done, but I enjoyed it none-the-less
9 Share Reply

dannycho7 ★ 16 March 13, 2020 11:56 PM
Isn't the runtime for solution 1 O(d^|E|)? For each of the |E| choices you need to make, there are d possible...
16 Share Reply
SHOW 2 REPLIES

qlan2 ★ 151 June 14, 2020 1:08 PM
I also think the time complexity of solution 1 is O(d^|E|), I thought the time complexity of backtracking/dps is O(branch ^ depth), in this problem, branch is d (neighbors or edges from a city), depth is |E|, which is the total flights/edges.
@qzhongx can you explain more why article's estimation is tighter? I thought O(E^d) is not necessarily
Read More
3 Share Reply

ntkw ★ 58 May 17, 2020 2:22 PM
Hi there guys, this is a great exercise, don't get discourage by the likes ratio
I'll leave a couple of notes here that resulted from the discussion with @liaison which helped me understand the exercise. Hope they might be useful for someone else.
Read More
3 Share Reply

kremebrulee ★ 52 April 17, 2020 9:54 AM
(|E| / 2 * |V|) is not correct.
Every node has E/V number of edges. Each edge is an outgoing as well as an incoming edge. For example, if we have two edges and two nodes, each node would have one edge outgoing.
2 Share Reply

vishalshah3584 ★ 46 June 11, 2020 11:22 AM
Question missing important point, solution has to cover all the cities give in list.
2 Share Reply

aysljc ★ 1 June 29, 2020 10:44 PM
This is a very good question because it forces me to think about what it means to pass by objects for functions in Python and what mutable and immutable objects are. I came up with the first method, but because I didn't know the recursive function will modify the list in place, I couldn't get the answer. Now everything is crystal clear.
1 Share Reply

ntkw ★ 58 May 12, 2020 9:29 PM
Is only me or the description is very very deceiving?
I think there are details missing man...
"If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order
Read More
1 Share Reply
SHOW 6 REPLIES

liaison STAFF ★ 5665 February 21, 2020 2:44 PM
hi @meetlore Indeed. The sorting part did slip through my fingers when I wrote up the complexity. We