787. Cheapest Flights Within K Stops (/problems/cheapest-flights-within-k-stops/)

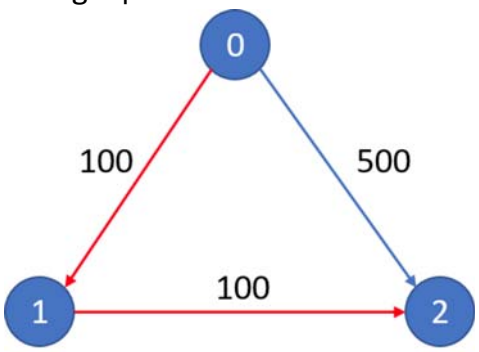
May 31, 2020 | 13.3K views

Average Rating: 3.49 (35 votes)

There are `n` cities connected by `m` flights. Each flight starts from city `u` and arrives at `v` with a price `w`.

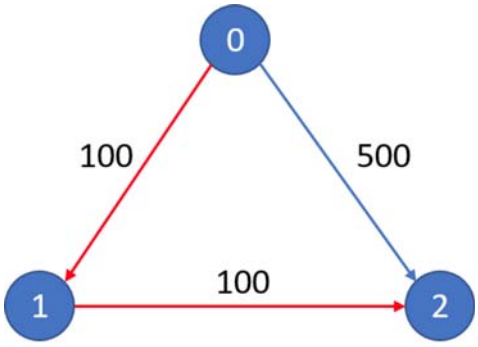
Now given all the cities and flights, together with starting city `src` and the destination `dst`, your task is to find the cheapest price from `src` to `dst` with up to `k` stops. If there is no such route, output `-1`.

Example 1:
Input:
`n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]`
`src = 0, dst = 2, k = 1`
Output: `200`
Explanation:
`The graph looks like this:`



`The cheapest price from city 0 to city 2 with at most 1 stop costs 200, as marked red in the picture.`

Example 2:
Input:
`n = 3, edges = [[0,1,100],[1,2,100],[0,2,500]]`
`src = 0, dst = 2, k = 0`
Output: `500`
Explanation:
`The graph looks like this:`



`The cheapest price from city 0 to city 2 with at most 0 stop costs 500, as marked blue in the picture.`

Constraints:

- The number of nodes `n` will be in range `[1, 100]`, with nodes labeled from `0` to `n - 1`.
- The size of `flights` will be in range `[0, n * (n - 1) / 2]`.
- The format of each flight will be `(src, dst, price)`.
- The price of each flight will be in the range `[1, 10000]`.
- `k` is in the range of `[0, n - 1]`.
- There will not be any duplicated flights or self cycles.

Solution

Approach 1: Dijkstra's Algorithm

Intuition

If we forget about the part where the number of stops is limited, then the problem simply becomes

the shortest path problem on a weighted graph, right? We can treat this as a graph problem where: *the cities can be treated as nodes in a graph* the connections between each of the cities can be treated as the edges and finally * the cost of going from one city to another would be the weight of the edges in the graph.

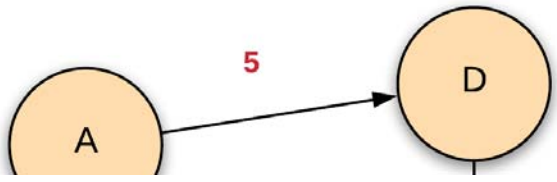
It's important to model the problem in a way that standard algorithms or their slight variations can be used for the solutions. Whenever we have a problem where we're given a bunch of entities and they have some sort of connections between them, more often than not it can be modeled as a graph problem. Once you've figured out that the question can be modeled as a graph problem, you then need to think about the various aspects of a graph i.e.

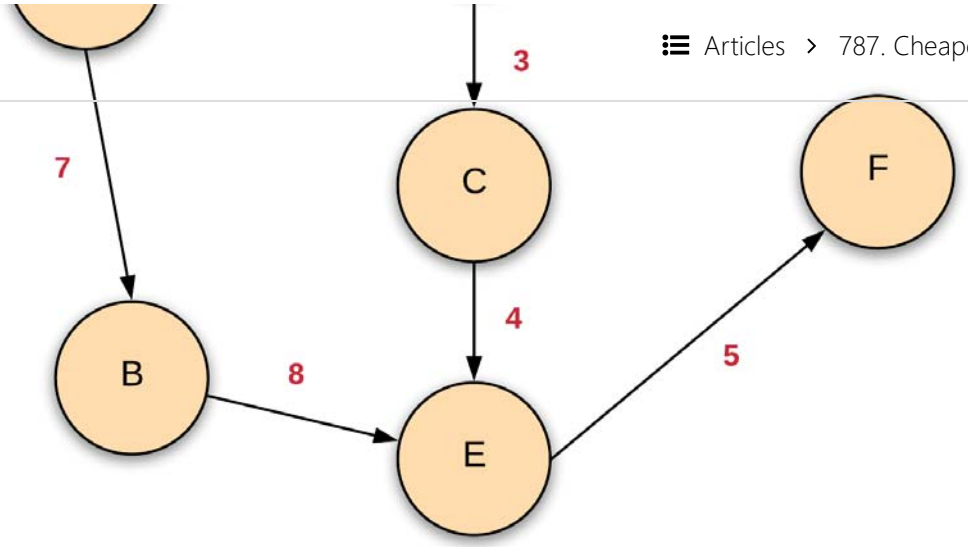
- directed vs undirected
- weighted vs unweighted
- cyclic vs acyclic

These aspects will help define the algorithm that you can consider for solving the problem at hand. For example a standard rule of thumb that is followed for solving shortest path problems is that we mostly use Breadth-first search (https://en.wikipedia.org/wiki/Breadth-first_search) for unweighted graphs and use Dijkstra's algorithm (https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm) for weighted graphs. An implied condition to apply the Dijkstra's algorithm is that the weights of the graph must be positive. If the graph has negative weights and can have negative weighted cycles, we would have to employ another algorithm called the Bellman Ford's (https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm). The point here is that the properties of the graph and the goal define the kind of algorithms we might be able to use.

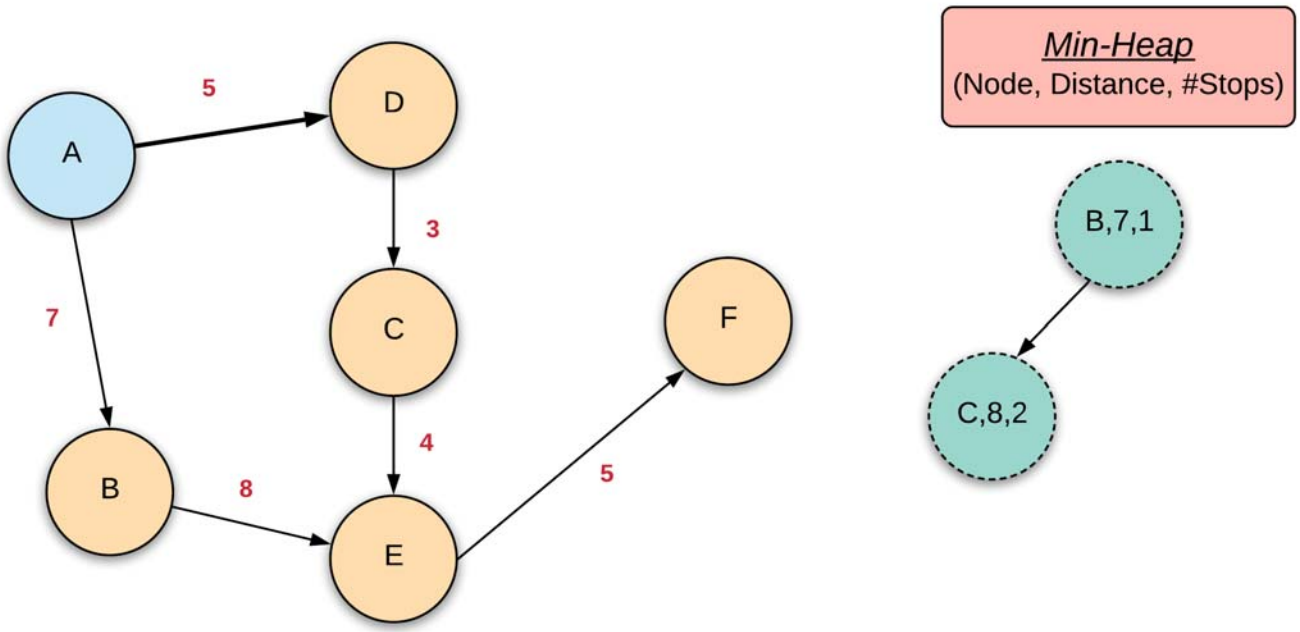
Coming back to the original statement at the beginning of the article. If we don't consider the part where the number of stops is limited, this problem becomes a standard shortest paths problem in a weighted graph with positive weights and hence, it becomes a prime candidate for Dijkstra's. As we all know, Dijkstra's uses a min-heap (priority queue) as the main data structure for always picking out the node which can be reached in the shortest amount of time/cost/weight from the current point starting all the way from the source. That approach as it is won't work out for this problem.

First of all, we need to keep track of the number of stops taken to reach a node (city), in addition to the shortest path from the source node. This is important because if at any point we find that we have exhausted K stops, we can't progress any further from that node because the number of stops are bounded by the problem. Let's consider a simple example and run through it with the basic Dijkstra's algorithm and see why we might run into a problem with the off-the-shelf code i.e. Dijkstra's without any modifications.



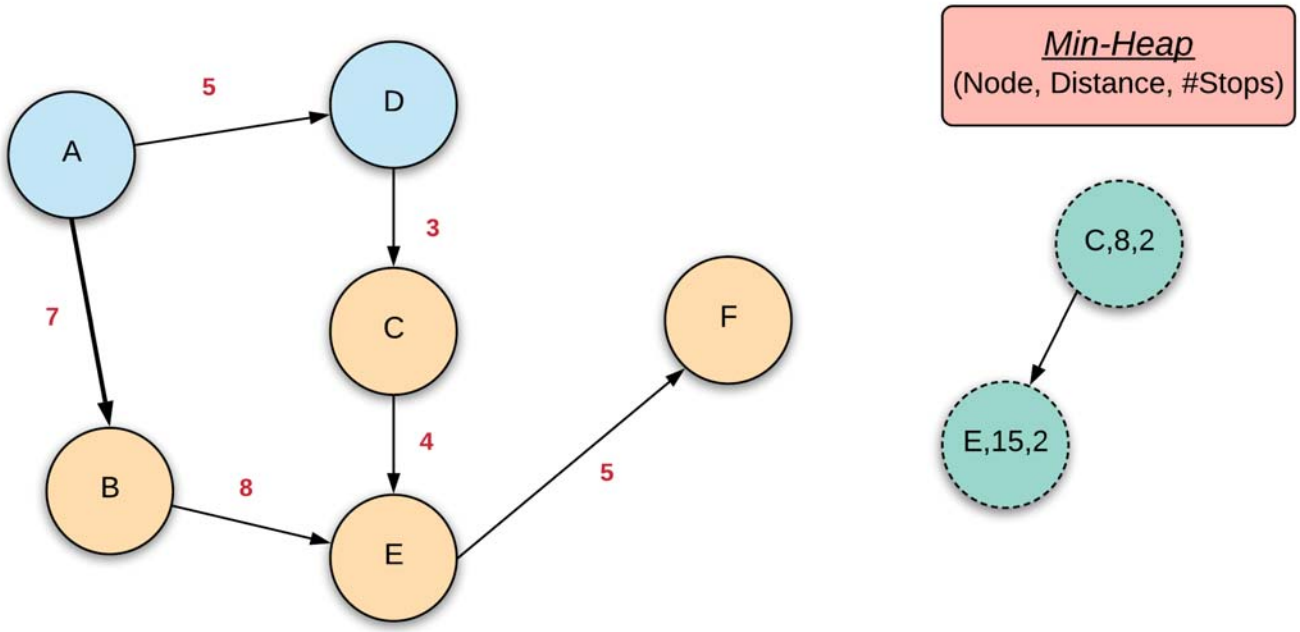


Now suppose that we want to go from the source node `A` in the graph above to the destination node `E` via the cheapest possible route with at most `2` stops. Let's ignore the number of stops for now and see how the usual Dijkstra would unfold and pick the nodes. So first of all, we will consider the neighbors of the source node and add them to our min-heap. Next, we will pick the element with the current shortest distance which would be `D` with a value of `5` as opposed to `B` with `7`.

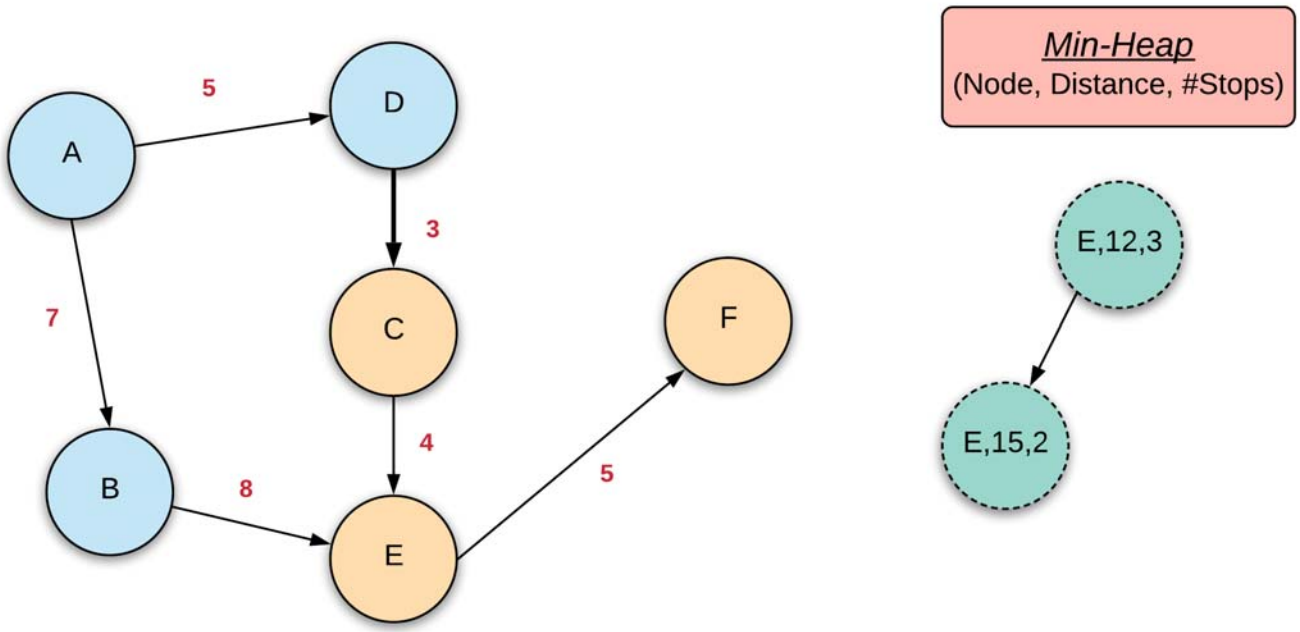


Moving on, the next node that will be picked is `B` since it has the current shortest distance from the source. Let's see what the heap looks like once we pick `B` and process its neighbors. Note that according to the algorithm, once a node has been processed i.e., once a node is popped from the min-heap, we never consider that node again in some other node's neighbors i.e., we never add it again to the heap down the line. This is because of the greedy nature of the algorithm. When a node is removed from the heap, it is guaranteed that the distance from the source at that point is the

shortest distance. The processed nodes are marked in blue in the figures here.



Moving on, the algorithm will pick `C` and its neighbor `E` will be added into the heap. You'll notice that there are two nodes containing the city `E` which is fine since `E` hasn't been processed yet and this just means there are multiple paths of reaching `E`.



Next, we will remove the node `E` with a distance of `12` from the source and `3` stops from the source. At this point, we cannot go any further i.e. we cannot consider its neighbor because We have already exhausted the number of stops in this example. So, we don't add the neighbor which also happens to be the destination node to the heap. The only node left in the heap is `E` with a distance

of `15` from source and `2` stops from the source.

> Here's the problem now. We will not consider this node because we have already processed the node `E` in the previous step. Clearly, the distance `15` is greater than `12`. So Dijkstra's will discard this heap node and the algorithm will finish, without ever reaching the destination!

The thing we need to modify here is that we need to re-consider a node if the distance from the source is shorter than what we have recorded. So we won't change the min-heap's priority which is to pick nodes with the shortest distance from the source. However, if we ever encounter a node that has already been processed before but the number of stops from the source is lesser than what was recorded before, we will add it to the heap so that it gets considered again! That's the only change we need to make to make Dijkstra's compliant with the limitation on the number of stops.

Algorithm

1. Initialize a min-heap or a priority queue. Let's call it `H` for our algorithm.
2. We will need a couple of arrays here. One would be for maintaining the shortest distances of each node from the source and another one would be for maintaining the shortest number of stops from the source.
3. Next, we need to convert the input into an adjacency matrix format. So, we will process the given input and build an adjacency matrix out of it.
4. Add $(\text{source}, 0, 0)$ into the heap. The middle value represents the current shortest distance from the source and the last value represents the current minimum number of stops from the source to reach this node.
5. We assume that these values for all the other nodes in the graph are `inf`.
6. We continue processing the nodes until either of the following conditions are met:
    1. We reach the destination node or
    2. We exhaust the heap which would mean we were not able to reach the destination at all.
7. At each step, we remove a node from the heap i.e. `ExtractMin` operation on the min-heap. This would represent the node with the shortest distance from the source amongst the ones in the heap. Let's call this node `C`.
8. We iterate over all of `C's` neighbors which we can obtain from our adjacency matrix. For each neighbor, we check if the value $\mathrm{dC} + \mathrm{W}_{\mathrm{C,\,V}}$ is less than dV where V represents the neighbor node, dC and dV represent the shortest distances (from the dictionary) of these nodes from the source and finally, $\mathrm{W}_{\mathrm{C,\,V}}$ represents the weight (cost of the flight) from node (city) C to V.
9. If this is not the case then we check if number of stops for node C `+ 1` is lower than the number of stops for the node V (from the other dictionary). If that is the case, then it means there is a path from the source to the node V which is slightly expensive than what we have

right now, but it has lesser stops and hence, it should be considered.

10. If either of the two conditions above are satisfied, we add the node V to the heap with updated distance and number of stops. In any case, we will update the corresponding dictionary as well.

Java Python3

Articles › 787. Cheapest Flights Within K St

Copy

```python
import heapq

class Solution:

    def findCheapestPrice(self, n: int, flights: List[List[int]], src: int, dst: int, K: int) -> int:

        # Build the adjacency matrix
        adj_matrix = [[0 for _ in range(n)] for _ in range(n)]
        for s, d, w in flights:
            adj_matrix[s][d] = w

        # Shortest distances array
        distances = [float("inf") for _ in range(n)]
        current_stops = [float("inf") for _ in range(n)]
        distances[src], current_stops[src] = 0, 0

        # Data is (cost, stops, node)
        minHeap = [(0, 0, src)]

        while minHeap:

            cost, stops, node = heapq.heappop(minHeap)

            # If destination is reached, return the cost to get here
            if node == dst:
                return cost

            # If there are no more steps left, continue
            if stops == K + 1:
                continue

            # Examine and relax all neighboring edges if possible
            for nei in range(n):
                if adj_matrix[node][nei] > 0:
                    dU, dV, wUV = cost, distances[nei], adj_matrix[node][nei]

                    # Better cost?
                    if dU + wUV < dV:
                        distances[nei] = dU + wUV
                        heapq.heappush(minHeap, (dU + wUV, stops + 1, nei))
                    elif stops < current_stops[nei]:

                        #  Better steps?
                        current_stops[nei] = stops
                        heapq.heappush(minHeap, (dU + wUV, stops + 1, nei))

        return -1 if distances[dst] == float("inf") else distances[dst]
```

Complexity Analysis

- Time Complexity: Let E represent the number of flights and V represent the number of cities. The time complexity is mainly governed by the number of times we pop and push into the heap. We will process each node (city) atleast once and for each city popped from the queue, we iterate over its adjacency matrix and can potentially add all its neighbors to the heap. Thus, the time taken for extract min and then addition to the heap (or simply, heap replace) would be $\mathrm{O}(\mathrm{V}^2 \cdot \log \mathrm{V})$.
    - Let's talk a bit more about the implementation of Dijkstra's here. The traditional algorithm is not exactly written the way we've explained above.
    - The traditional algorithm adds *all the nodes* into the heap with the source having a distance value of `0` and all others having a value `inf`.
    - When we process the neighbors of a node and find that a particular neighbor can be reached in a shorter distance (or lesser number of stops), we *update its value in the heap*. In our implementation, we add a new node with updated values rather than updating the value of the existing node. To do that, we will need another dictionary that will probably keep the index location for a node in the heap or something like that. This would be necessary because a heap is not a binary search tree and it doesn't have any search properties for quick search and updates.
    - If we keep the number of nodes in the heap fixed to V, then the complexity would be $\mathrm{O}((\mathrm{V} + \mathrm{E}) \cdot \log \mathrm{V})$. Granted, in our case, the heap might contain more than V nodes at some point due to the same city being added multiple times. Therefore, the complexity would be slightly more. That is not being accounted for here since that is an implementation detail and not necessary for the algorithm we discussed here.
    - Yet another point to keep in mind here is that we are using an adjacency matrix rather than adjacency list here. The typical Dijkstra's algorithm would use an adjacency list and that brings down the complexity slightly because you don't "check" if a connection exists or not unlike in adjacency matrix. However, since the number of nodes are very less for this problem, we preferred to take the route of adjacency matrix as that gives us sequential access to elements and leads to speed-ups due to cache localization.
- Space Complexity: $\mathrm{O}(\mathrm{V}^2)$ is the overall space complexity. $\mathrm{O}(\mathrm{V})$ is occupied by the two dictionaries and also by the heap and V^2 by the adjacency matrix structure. As mentioned above, there might be duplicate cities in the heap with different distances and number of stops due to our implementation. But we are not taking that into consideration here. This is the space complexity of the traditional Dijkstra's and it doesn't change with the algorithm modifications (not the implementation modifications) we've done here.

Approach 2: Depth-First-Search with Memoization

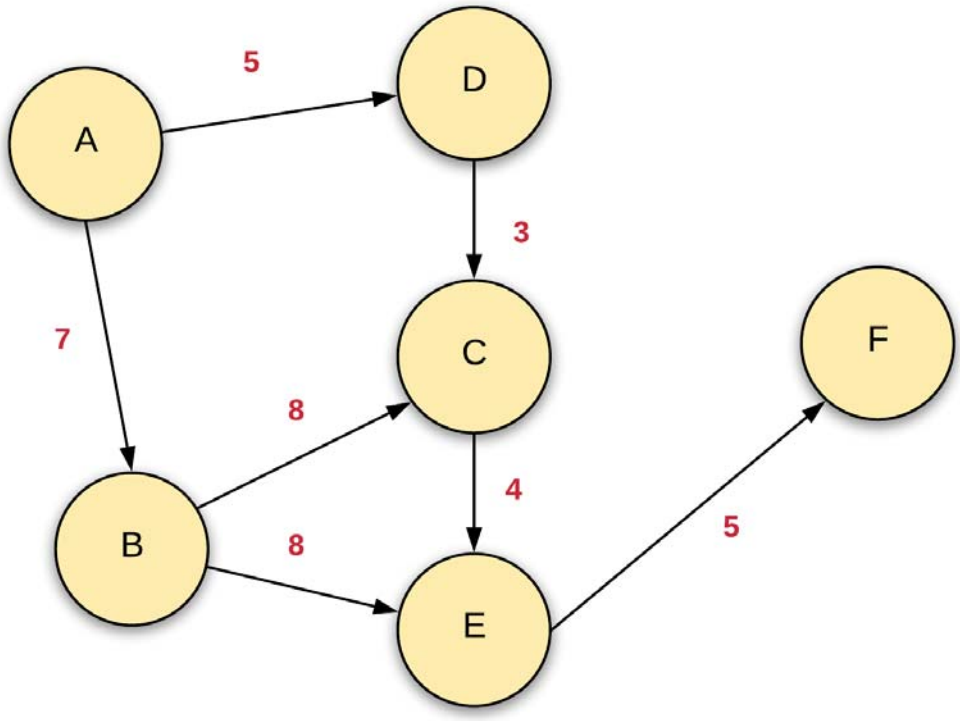
Intuition

This problem can easily be modeled as a dynamic programming problem on graphs. What does a dynamic programming problem entail? *It has a recursive structure.* A bunch of choices to explore at each step. *Use the optimal solutions for sub-problems to solve top-level problems.* A base case.

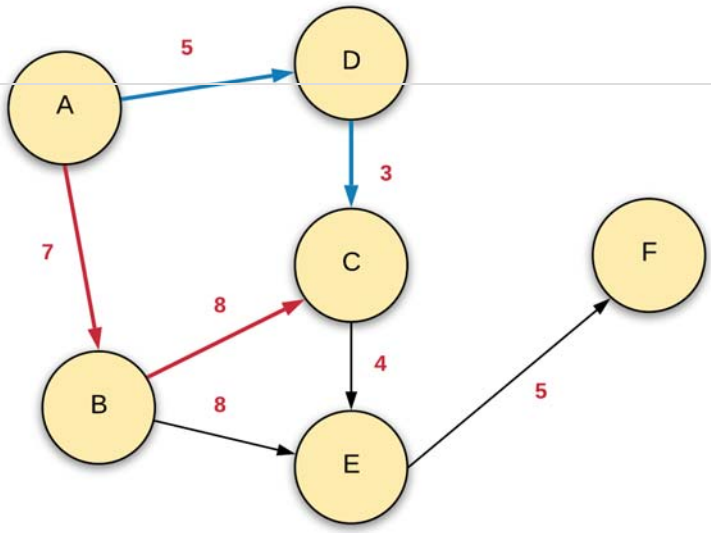
This problem fits the bill. We have a dedicated start and endpoint. We have a bunch of choices for each node in the form of its neighbors. And, we want to minimize the overall shortest distance from the source to the destination which can be represented as a recursive structure in terms of shortest distances of its neighbors to the destination. So, we can apply a dynamic programming approach to solve this problem. We'll look at a recursive implementation here with memoization first and then talk about the iterative approach as well.

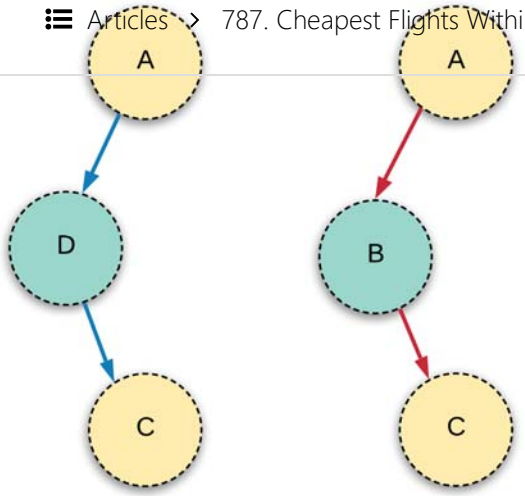
As with any recursive approach, we need to figure out the state of recursion. There are two parameters here which will control our recursion. One is obviously the node itself. The other is the number of steps. Let's call our recursion function `recurse` and define what the state of recursion looks like. $\text{recurse}(\text{node}, \text{stops})$ will basically return the shortest distance for us to reach the destination from node considering that there are `stops` left. This being said, it's easy to figure out what the top-level problem would be. It would be $\text{recurse}(0, \text{K})$.

Let's consider the following graph to understand why memoization (or caching) is required here.



Say we start the source node `A` and build our recursion tree from there. There are two possible routes of getting to the node `C` with exactly `2` stops. Let's look at what these are.





While the cost of these two paths is different, once we are at the node `C`, we have `2` steps less than what we had when we started off from the source node `A`. Our recursion representation doesn't care about the path you took to get to a node. It is about the shortest (cheapest) path from the current node with the given number of steps to get to a destination. In that sense, both these scenarios are exactly the same because both lead us to the same recursion state which is $(\text{recurse}(\text{C}, \text{K-2}))$ and hence, the result for this recursion state can be cached or memoized.

Algorithm

1. We'll define a function called `recurse` which will take two inputs: `node` and `stops`.
2. We'll also define a dictionary `memo` of tuples that will store the optimal solution for each recursion state encountered.
3. At each stage, we'll first check if we have reached the destination or not. If we have, then no more moves have to be made and we return a value of `0` since the destination is at a zero distance from itself.
4. Next, we check if we have any more stops left. If we don't then we return `inf` basically representing that we cannot reach the destination from the current recursion state.
5. Finally, we check if the current recursion state is cached in the `memo` dictionary and if it is, we return the answer right away.
6. If none of these conditions are met,we progress in our recursion. For that we will iterate over the adjacency matrix to obtain the neighbors for the current node and make a recursive call for each one of them. The `node` would be the neighboring node and the number of stops would incremeneted by `1`.
7. To each of these recursion calls, we add the weight of the corresponding edge i.e.

```
recurse(neighbor, stops + 1) + weight(node, neighbor)
```

8. We need to return the result of `recurse(src, 0)` as the answer.

Java Python3 Articles › 787. Cheapest Flights Within K St Copy

```
class Solution:

    def __init__(self):
        self.adj_matrix = None
        self.memo = {}

    def findShortest(self, node, stops, dst, n):

        # No need to go any further if the destination is reached
        if node == dst:
            return 0

        # Can't go any further if no stops left
        if stops < 0:
            return float("inf")

        # If the result of this state is already cached, return it
        if (node, stops) in self.memo:
            return self.memo[(node, stops)]

        # Recursive calls over all the neighbors
        ans = float("inf")
        for neighbor in range(n):
            if self.adj_matrix[node][neighbor] > 0:
                ans = min(ans, self.findShortest(neighbor, stops-1, dst, n) +
self.adj_matrix[node][neighbor])

        # Cache the result
        self.memo[(node, stops)] = ans
        return ans

    def findCheapestPrice(self, n: int, flights: List[List[int]], src: int, dst: int, K: int)
-> int:

        self.adj_matrix = [[0 for _ in range(n)] for _ in range(n)]
        self.memo = {}
        for s, d, w in flights:
            self.adj_matrix[s][d] = w

        result = self.findShortest(src, K, dst, n)
        return -1 if result == float("inf") else result
```

Complexity Analysis

- Time Complexity: The time complexity for a recursive solution is defined by the number of recursive calls we make and the time it takes to process one recursive call. The number of

recursive calls we can `potentially` make is $\mathrm{O}(\mathrm{V} \cdot \mathrm{K})$. In each recursive call, we iterate over a given node's neighbors. That takes time $O(\mathrm{V})$ because we are using an adjacency matrix. Thus, the overall time complexity is $\mathrm{O}(\mathrm{V}^2 \cdot \mathrm{K})$.

- Space Complexity: $\mathrm{O}(\mathrm{V} \cdot \mathrm{K} + \mathrm{V}^2)$ where $\mathrm{O}(\mathrm{V} \cdot \mathrm{K})$ is occupied by the `memo` dictionary and the rest by the adjacency matrix structure we build in the beginning.

Approach 3: Bellman-Ford

Intuition

Let's look at the official definition of the Bellman-Ford algorithm straight from Wikipedia:

Like Dijkstra's algorithm, Bellman-Ford proceeds by relaxation, in which approximations to the correct distance are replaced by better ones until they eventually reach the solution. In both algorithms, the approximate distance to each vertex is always an overestimate of the true distance and is replaced by the minimum of its old value and the length of a newly found path.

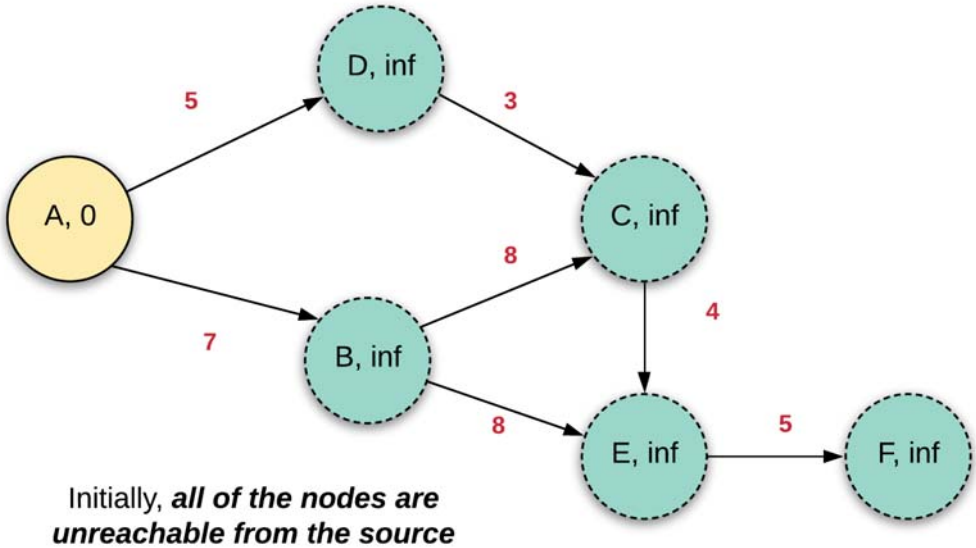
However, Dijkstra's algorithm uses a priority queue to greedily select the closest vertex that has not yet been processed, and performs this relaxation process on all of its outgoing edges; by contrast, the Bellman-Ford algorithm simply relaxes all the edges and does this $|V| - 1$ times, where $|V|$ is the number of vertices in the graph. In each of these repetitions, the number of vertices with correctly calculated distances grows, from which it follows that eventually, all vertices will have their correct distances. This method allows the Bellman-Ford algorithm to be applied to a wider class of inputs than Dijkstra.

The term `relax an edge` simply means that for a given edge `U -> V` we check if $\mathrm{dU} + W_{\mathrm{U,V}} < \mathrm{dV}$ where dU and dV represent the shortest path distances of these nodes from the source right now. To relax an edge means to see if the shortest distance can be updated or not.

An important part to understanding the Bellman Ford's working is that at each step, the relaxations lead to the discovery of new shortest paths to nodes. After the first iteration over all the vertices, the algorithm finds out all the shortest paths from the source to nodes which can be reached with one hop (one edge). That makes sense because the only edges we'll be able to relax are the ones that are directly connected to the source as all the other nodes have shortest distances set to `inf` initially.

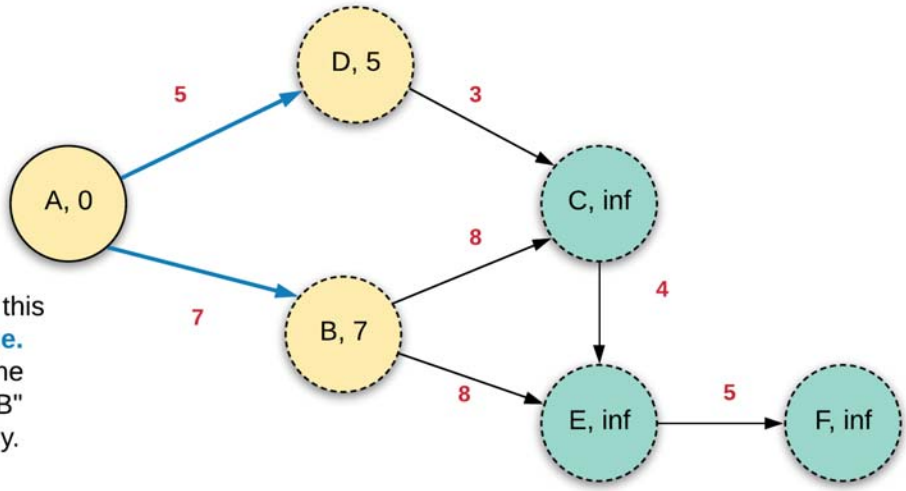
Similarly, after the $(\mathrm{K}+1)^{\mathrm{th}}$ step, Bellman-Ford will find the shortest distances for all the nodes that can be reached from the source using a maximum of `K` stops. Isn't that what the question asks us to do? If we run Bellman-Ford for $\mathrm{K}+1$ iterations, it will find out shortest paths of length K or less and it will find all such paths. We can then check if our destination node was reached or not and if it was, then the value for that node would be our shortest path!

Let's quickly look at a couple of iterations of Bellman-Ford on a sample graph to understand how relaxation works and how $K+1$ iterations can possibly give us our solution. The image below showcases the initial setup before the first iteration of Bellman-Ford is executed.



Initially, ***all of the nodes are unreachable from the source node*** "A" since we haven't relaxed any of the edges. The values next to the nodes are their current shortest distances from the source

Let's look at what the graph looks like after a single iteration.



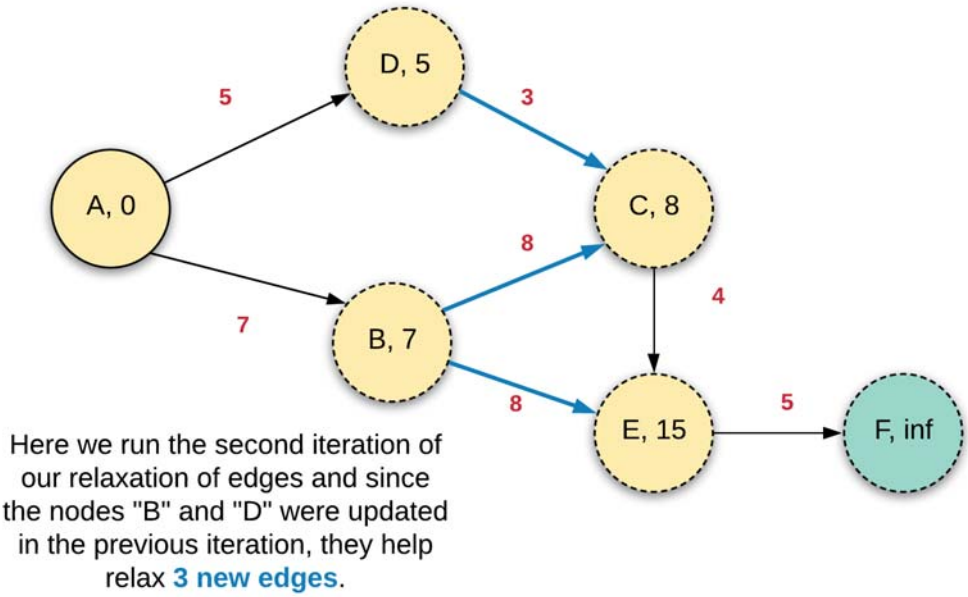
The only edges we can relax in this iteration are highlighted in **blue.** Once we relax these edges, the values of the nodes "D" and "B" change to 5 and 7 respectively.

We also highted the nodes "B" and "D" in yellow because ***there is no shorter distance of length 1 possible from source "A"***. Notice how we're not saying the overall shortest distance. We are talking about shortest distance using 1 edge only!

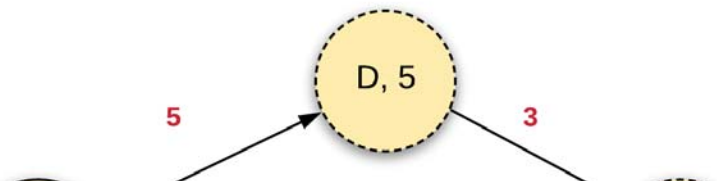
It's important to understand the meaning of what we said in the figure above. We are not saying that after the first iteration we will find the absolute shortest distance from `A` to `B` and `D`. We are just saying that the shortest distance using a single edge only will be found after first iteration. What happens in the next iteration? Well, we will find all the shortest paths that can be reached from the source by using at-most `2` edges. In this example, since the values for nodes `B` and `D` were updated in the previous iteration, they will be re-used in the next iteration to relax edges `B -> E`, `B -> C`, and `D -> C`.

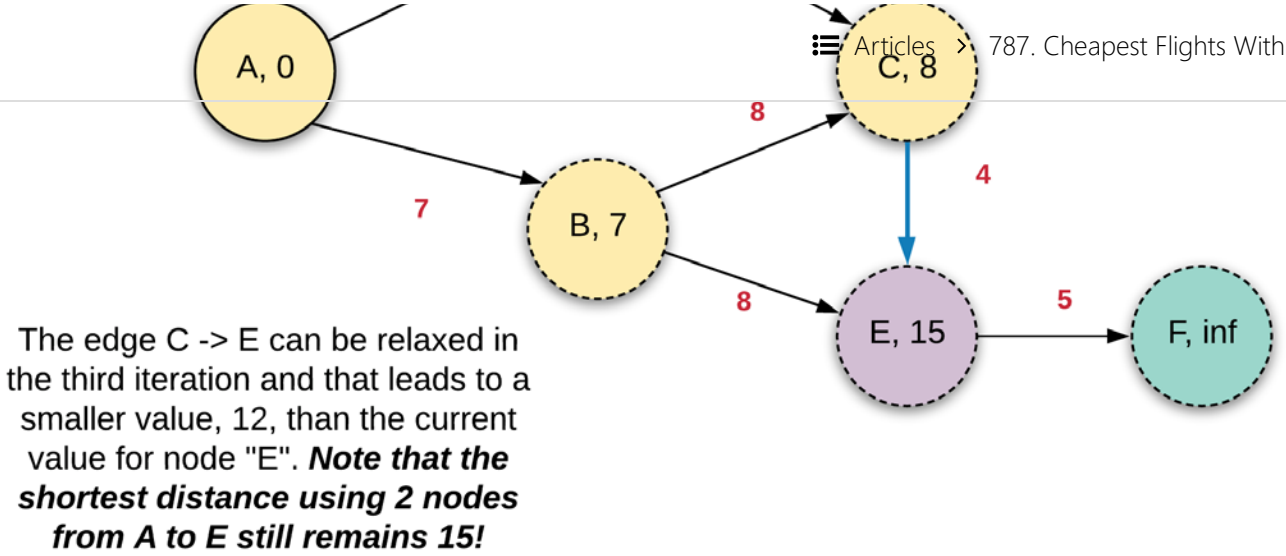
Isn't that what Dynamic Programming is all about.....

Well, yes! Using the optimal solutions to sub-problems to find optimal solutions to bigger problems. We use the optimal solutions to shortest paths using `1` edge to find shortest paths using `2` edges and so on.



We'll go one final iteration here since this is where things get interesting and this will bring some more clarity. The node `E` was discovered in the second iteration and we have the value `15` corresponding to it. However, one of the incoming edges `C -> E` wasn't relaxed in the second iteration because `C` was also discovered during that iteration. Now that we have a non-infinite value associated with `C`, we can use it to relax the edge `C -> E` and that leads to an even shorter path from `A ... E`!





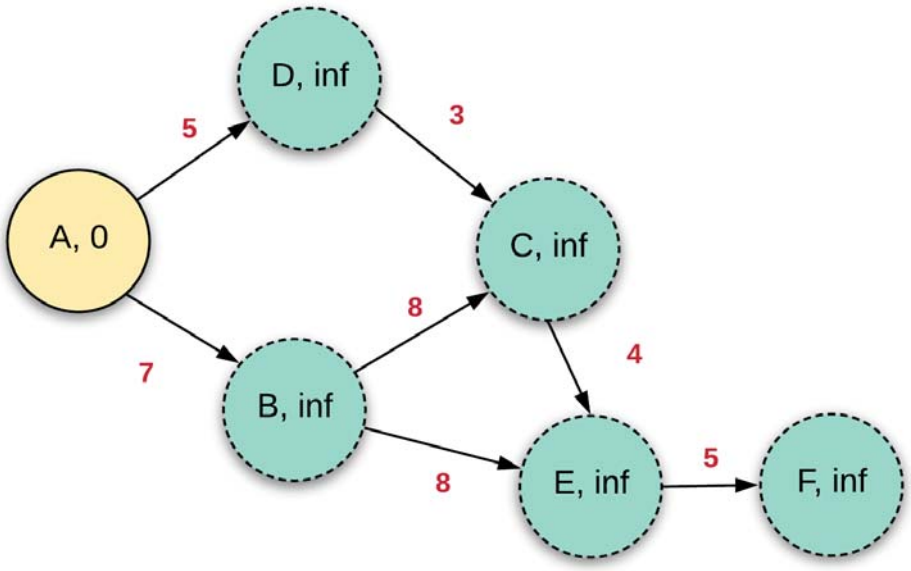
Another important thing to note about this algorithm is that we don't need to build an adjacency matrix. This algorithm simply iterates over the edges of the graph and that information is already available in the input for the program. So we save on space there as opposed to other algorithms which we've seen.

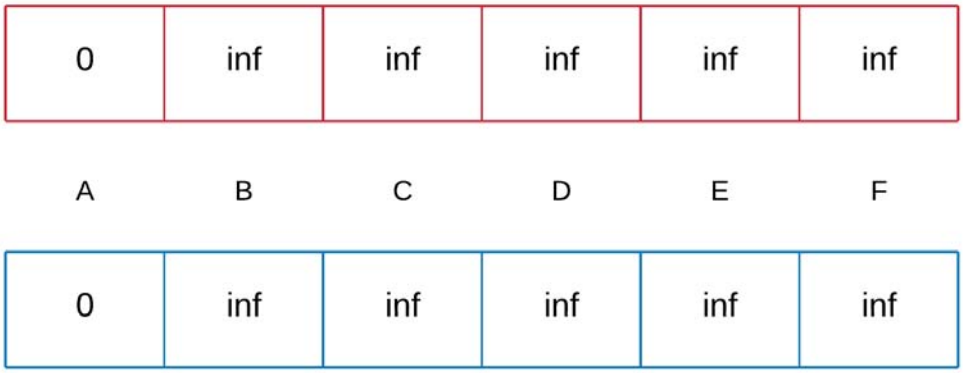
Algorithm

1. We have a loop that does `K + 1` iterations. The plus one is because we need to find the cheapest flight route with at most `K` stops in between. That translates to `K + 1` edges at most.
2. In each iteration, we loop over all the edges in the graph and try to relax each one of them. Again, note that the edges or the flights are already given to us in the input and don't need to build any kind of adjacency list or matrix structure which is otherwise standard for other graph algorithms.
3. After `K + 1` iterations, we check if the destination has been reached or not. If it's been discovered, then the distance at that point will be the shortest using at most `K + 1` edges.
4. We use an array to store the current shortest distances of each node from the source. This is possible because the number of nodes is less and we don't need to use a dictionary here. However, a single array is not sufficient here because any values updated in a particular iteration cannot be used to update other values in the same iteration. Thus, we need another distance array which will kind of server as `values in the previous iteration`. So we essentially use 2 arrays of size V and we swap between them in each iteration i.e.

```
Iteration-0 ----
   Array-1 is the main array
   Array-2 becomes the previous array
   Iteration-1 ----
   Array-2 is the main array
   Array-1 becomes the previous array
```

Let's look at how the two arrays look like at the start of the first iteration. We'll take a look at a couple of iterations so that the it's easier to understand the implementations.

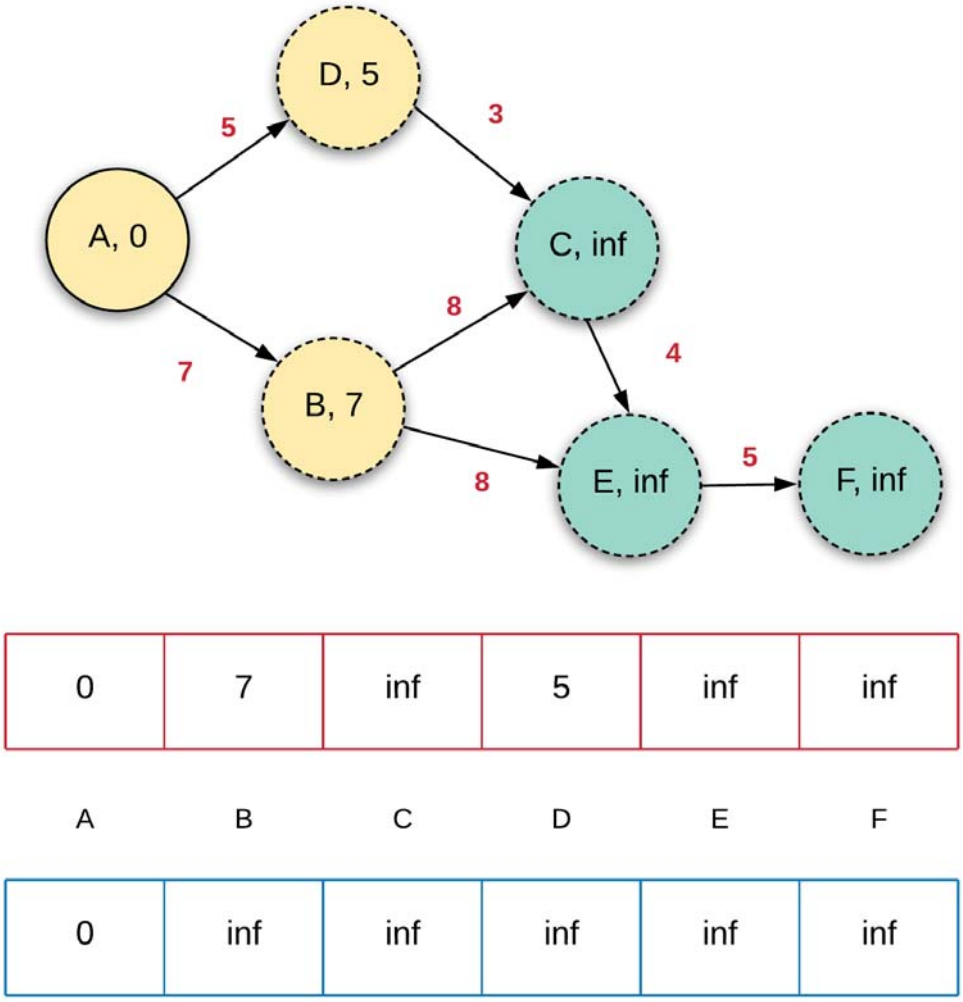




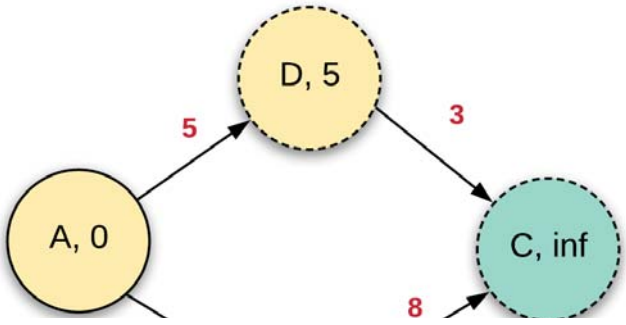
Iteration 0
Current Array = 0&1 = 0
Previous Array = 1 - (0&1) = 1

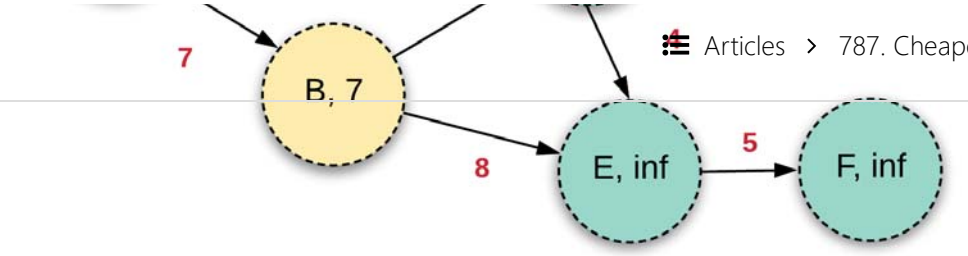
We discovered two new vertices which are directly connected from the source and their

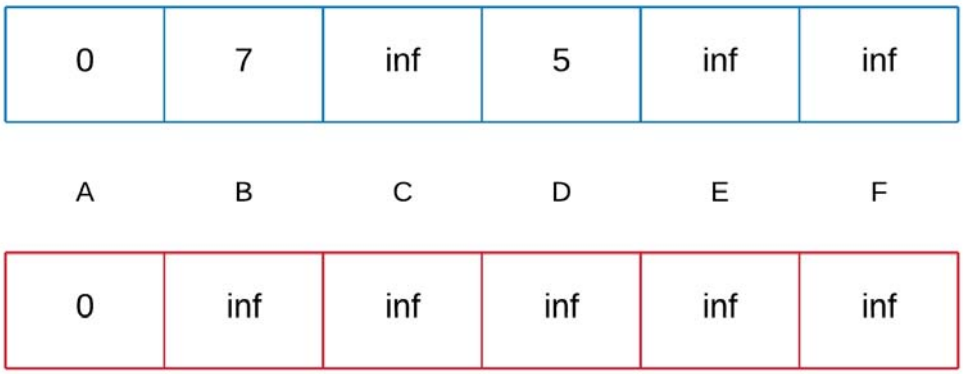
corresponding distances were updated accordingly.



Now let's look at how the two arrays would look like at the start of the second iteration. Now the roles would be reversed. The `current` array in the previous iteration now servers as the `previous` array.

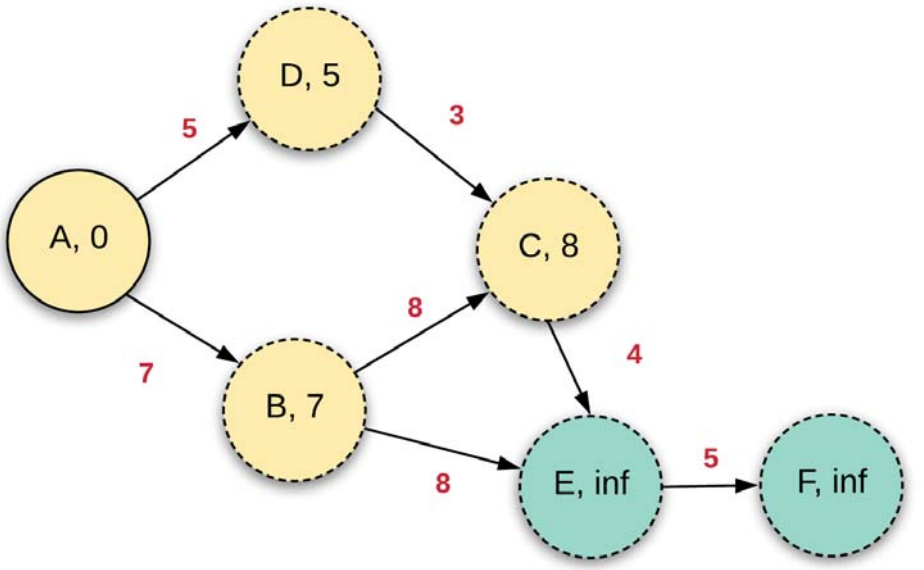






Iteration 1
Current Array = 1&1 = 1
Previous Array = 1 - (1&1) = 0

Notice how the two arrays have swapped roles. You might be thinking that even though the `red` array is the current one, it doesn't have the latest values `7` and `5`. Well, they will be used for the calculation of distance of node `C` and also, they will be copied over (re-calculated again due to the node `A` in `previous` array). Let's see how the two arrays look after the second iteration is complete.



0	7	inf	5	inf	inf

A	B	C	D	E	F
0	7	8	5	inf	inf

Iteration 1
Current Array = 1&1 = 1
Previous Array = 1 - (1&1) = 0

Java Python3 Articles > 787. Cheapest Flights Within K St Copy

```python
class Solution:

    def findCheapestPrice(self, n: int, flights: List[List[int]], src: int, dst: int, K: int) -> int:

        # We use two arrays for storing distances and keep swapping
        # between them to save on the memory
        distances = [[float('inf')] * n for _ in range(2)]
        distances[0][src] = distances[1][src] = 0

        # K + 1 iterations of Bellman Ford
        for iterations in range(K + 1):

            # Iterate over all the edges
            for s, d, wUV in flights:

                # Current distance of node "s" from src
                dU = distances[1 - iterations&1][s]

                # Current distance of node "d" from src
                # Note that this will port existing values as
                # well from the "previous" array if they didn't already exist
                dV = distances[iterations&1][d]

                # Relax the edge if possible
                if dU + wUV < dV:
                    distances[iterations&1][d] = dU + wUV

        return -1 if distances[K&1][dst] == float("inf") else distances[K&1][dst]
```

Complexity Analysis

- Time Complexity: $O(\mathrm{K} \cdot \mathrm{E})$ since we have $\mathrm{K} + 1$ iterations and in each iteration, we go over all the edges in the graph.
- Space Complexity: $\mathrm{O}(\mathrm{V})$ occupied by the two distance arrays.

Approach 4: Breadth First Search

Intuition

We say that the breadth-first search is a good algorithm to use if we want to find the shortest path in

an undirected, unweighted graph. The claim for BFS is that the first time a node is discovered during the traversal, that distance from the source would give us the shortest path. The same cannot be said for a weighted graph. For a weighted graph, there is no correlation between the number of edges composing the path and the actual length of the path which is composed of the weights of all the edges in it. Thus, we cannot employ breadth-first search for weighted graphs.

Breadth-first search has no way of knowing if a particular discovery of a node would give us the shortest path to that node. And so, the only possible way for BFS (or DFS) to find the shortest path in a weighted graph is to search the entire graph and keep recording the minimum distance from source to the destination vertex.

> That being said, Breadth-first search is actually a great algorithm of choice for this problem because the number of levels to be explored by the algorithm is bounded by K

The number of levels that the search would go to is limited by the value `K+1` in the question. So essentially, we would be trying to find the shortest path, but we won't have to explore the entire graph as such. We will just go up to the level `K+1` and we just need to return the shortest path to the destination (if reachable by level `K+1`) at the end of the algorithm.

An important consideration here is the size of the queue. We need to control it somehow otherwise, even at very small depths, the graph could grow exponentially. For this very problem however, we will be able to bound the size of a given level (and hence the queue) by V, the number of vertices in the graph. Let's think about what it means to encounter the same node multiple times during breadth first traversal.

Since we will be going only till the level `K+1`, we don't really have to worry about the number of stops getting exhausted or something. So if the number of stops are out of the way, the only way we will consider adding a node again to the queue is if we found a shorter distance from the source than what we already have stored for that node. If that is not the case then on encountering a node again during the traversal, we can safely discard it i.e not add it to the queue again.

Since this is weighted graph, we cannot assume anything about the shortest distance from source to a node when its first discovered after being popped from the queue. We will have to go to all the `K+1` levels and once we've exhausted `K+1` levels, we can be sure that the shortest distances we have are the "best" we can find with `K+1` edges or less.

Algorithm

1. This is standard BFS and we'll be using a queue here. Let's call it `Q`.
2. We'll need a dictionary to keep track of shortest distances from the source. An important thing to note in this approach is that we need to keep a dictionary with the `node, stops` as the key.

Basically, we need to keep track of the shortest distance of a node from the source provided that it takes `stops` stops to reach it.

3. Add the source node to the queue. There are multiple ways of tracking the level of a node during breadth-first traversal. We'll be using the size of the queue at the beginning of the level to loop over a particular level.
4. We iterate until we exhaust the queue or `K+1` levels whichever comes first.
5. For each iteration, we pop a node from the queue and iterate over its neighbors which we can get from the adjacency matrix.
6. For each of the neighbors, we check if the current edge improves that neighbor's shortest distance from source or not. If it does, then we update the shortest distance dictionary (array) accordingly and also add the neighbor to the queue.
7. We continue doing this for until one of our terminal conditions are met.
8. We will also maintain an `ans` variable to track the minimum distance of the destination from the source. At each step, whenever we update the shortest distance of a node from source, we check if that node is the destination and if it is, we will update the `ans` variable accordingly.
9. At the end, we simply check if we were able to reach the destination node by looking at the `ans` variable's value. If we did reach it, then the recorded distance would be the shortest in under `K` hops (or `K + 1` edges at most).

Java Python3 Copy

```
class Solution:

    def findCheapestPrice(self, n: int, flights: List[List[int]], src: int, dst: int, K: int)
-> int:

        # Build the adjacency matrix
        adj_matrix = [[0 for _ in range(n)] for _ in range(n)]
        for s, d, w in flights:
            adj_matrix[s][d] = w

        # Shortest distances dictionary
        distances = {}
        distances[(src, 0)] = 0

        # BFS Queue
        bfsQ = deque([src])

        # Number of stops remaining
        stops = 0
        ans = float("inf")

        # Iterate until we exhaust K+1 levels or the queue gets empty
        while bfsQ and stops < K + 1:

            # Iterate on current level
            length = len(bfsQ)
            for _ in range(length):
```

Complexity Analysis

- Time Complexity: $O(\mathrm{E} \cdot \mathrm{K})$ since we can process each edge multiple times depending upon the improvement in the shortest distances. However, the maximum number of times an edge would be processed is bounded by $\mathrm{K} + 1$ since that's the number of levels we are going to explore in this algorithm.
- Space Complexity: $O(\mathrm{V}^2 + \mathrm{V} \cdot \mathrm{K})$. The first part is the standard memory occupied by the adjacency matrix and in addition to that, the distances dictionary can occupy a maximum of $O(\mathrm{V} \cdot \mathrm{K})$.

Rate this article:

Previous (/articles/word-break-ii/) Next (/articles/add-digits/)

Comments: 10

Sort By

Type comment here... (Markdown is supported)

Preview Post

(/agamjolly)

agamjolly (/agamjolly) ★ 75 June 1, 2020 5:52 PM

This problem should probably be classified as hard.

68 Share Reply

HIDE 1 REPLY

(/harleyquinn97)

HarleyQuinn97 (/harleyquinn97) ★ 7 3 days ago

can't agree more

4 Share Reply

(/meowlicious99)

meowlicious99 (/meowlicious99) ★ 432 June 8, 2020 10:07 AM

Dijkstra's soln looks really convoluted for some reason. Here a simpler/straightforward implementation.

```
 def findCheapestPrice(self, n: int, flights: List[List[int]], src:
int, dst: int, K: int) -> int:
        graph = defaultdict(list)
        for source,dest,cost in flights: graph[source].append((cost,
dest))

        heap = []
        heapq.heappush(heap,(0,src,-1)) # add src to kick things off
        while heap:
            nextCost,nextDest,currSteps= heapq.heappop(heap)
            if currSteps > K: continue
            if nextDest == dst: return nextCost

            for nc,nd in graph[nextDest]:
                heapq.heappush(heap,(nc+nextCost,nd,currSteps+1))

        return -1
```

17 Share Reply

SHOW 3 REPLIES

(/firrrrred)

firrrrred (/firrrrred) ★ 7 3 days ago

for dijkstra solution, in the for loop, while you update the stops[nei], I think it should be stops[nei]=stops+1 instead of stops[nei]=stops; although I tries the two method they both pass all test cases, but I think here while stops means the minstop to reaches the current node you're visiting, while you update stops[nei] you should plus 1 more stop by reaching the neighbor(nei) via current node(node). So it should be "stops+1" instead of "stops". Does

Read More

3 Share Reply

SHOW 3 REPLIES

(/lenaredwood)

lenaredwood (/lenaredwood) ★ 5 June 6, 2020 8:34 PM

BFS solution doesn't pass existing tests.
For instance:
5, [[0,1,1],[0,2,5],[1,2,1],[2,3,1],[3,4,1]], 0, 4, 2

3 Share Reply

SHOW 2 REPLIES

(/vemikhaylov)

vemikhaylov (/vemikhaylov) ★ 2 June 7, 2020 7:44 AM

Everything written here should be rechecked thoroughly. Especially the complexity analysis for the Dijkstra and BFS looks incorrect. If you should reconsider a node each time its cheaper path is found, it cannot be O(N) for nodes processing. Also the adjacency matrix for graph storage is less efficient than the adjacency list. If you use the adjacency matrix, you cannot say that even the basic Dijkstra's algorithm works in O((N +

Read More

2 ▲ ▼ Share Reply

SHOW 2 REPLIES

(/ljnopok)

ljnopok (/ljnopok) ★ 2 June 4, 2020 11:18 PM

"we process each edge exactly once." I don't think this is true since if you find a better path to a particular node you add the node back into the queue. If you're visiting a node multiple times in the queue it means you're processing the outgoing edges of that node more than once.

2 ▲ ▼ Share Reply

SHOW 1 REPLY

(/kinssang)

kinssang (/kinssang) ★ 1 June 14, 2020 8:36 AM

I think Dijkstra solution's time complexity should be O(KV^2logV).
As you said, same node can be processed up to K times because of number of stops differences.

1 ▲ ▼ Share Reply

SHOW 3 REPLIES

(/ping_pong)

ping_pong (/ping_pong) ★ 761 a day ago

What will be the time complexity for Dijkstra and BFS if we use adjacency list.

0 ▲ ▼ Share Reply

(/shuangpan)

shuangpan (/shuangpan) ★ 12 3 days ago

Straightforward, but it has overhead. Don't know how to fix it about how to mark the visited vertex.

```
class Solution {
    public int findCheapestPrice(int n, int[][] flights, int src, int d
```

Read More

0 ▲ ▼ Share Reply

(/box_of_donuts)

box_of_donuts (/box_of_donuts) ★ 297 3 days ago

The time complexity for BFS seems to be wrong: from what I see, there is no prevention on the queue on a particular iteration having multiple copies of the same vertex. In particular, consider the case where on a particular iteration, every vertex currently in the queue has an edge to some vertex v, and we happen to process the current iteration's vertices in descending order of their distance to v. Then we would

Read More

0 Share Reply