Dijkstra

Dijkstra

- Definition **+ Use case**
    - Shortest Path Distance

- **Dijkstra Template**
- **Talk is cheap. Show me the code.**
    1. **Cheapest Flights Within K Stops**
    2. **Network Delay Time**
    3. **The Maze II**

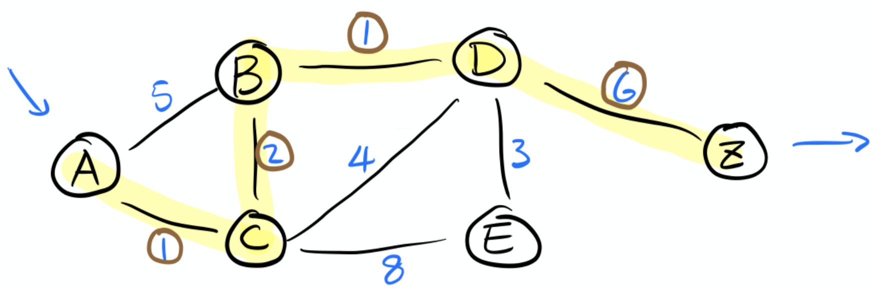
Dijkstra | Definition

> Dijkstra's algorithm (or Dijkstra's Shortest Path First algorithm, SPF algorithm)[2] is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks.

关键词

> `Weighted Graph`
> BFS vs Dijkstra?

Dijkstra



Shortest Path:

A → C → B → D → Z

算法思维:

Initialization

- **Create Adjacency List**
- **Initialize a Visited Set，If vertex has been visited, ignore the current vertex**
- **Initialize a Heap，Enqueue source vertex**

BFS

- **Regular BFS**
- **Enqueue Condition: Not in Visited Set**
- During **Enqueue，update weight**

▼ Dijkstra (Start to End)

```
# graph is type {int:list}, start and end are integers. return type : int
from heapq import heappush, heappop
def dijkstra(graph, start, end):
    heap = [(0, s)] # dist | start point
    visited = set()



    while heap:
        distance, node = heappop(heap)
        visited.add(node)
        if node == end:
           return distance
        if node not in visited:
            for neighbor, d in graph[node]:
                heappush(heap, (distance + d, neighbor))
    return -1
```

▼ **Dijkstra (Shortest Paths)**

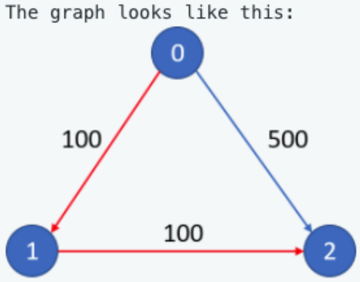
```
# graph is type {int:list}, start is type int. return type : int
from heapq import heappush, heappop
def dijkstra(graph, start):
    heap = [(0, s)]
    visited = {}
    while heap:
        distance, node = heappop(heap)
        if node not in visited:
            visited[node] = distance
            for neighbor, d in graph[node]:
                heappush(heap, (distance + d, neighbor))
    return visited
```

Dijkstra | Questions

▼ **Cheapest Flights Within K Stops**

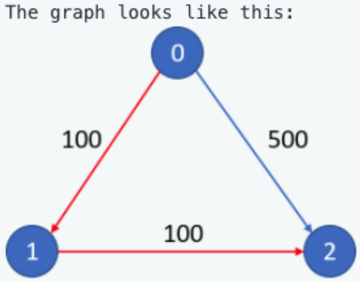
> There are n cities connected by m flights. Each fight starts from city u and arrives at v with a price w.
>
> Now given all the cities and flights, together with starting city src and the destination dst, your task is to find the cheapest price from src to dst with up to k stops. If there is no such 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.
```

▼ Code **(Final)**

💡 Time Complexity: `O(nlogn)` | Space Complexity: `O(n)`

```
from collections import defaultdict
from heapq import heappush, heappop
class Solution(object):
    def findCheapestPrice(self, n, flights, src, dst, total_stops):
        dic = defaultdict(list)
        for start, end, price in flights:
            dic[start].append((end, price))

        stop = 0
```

```
usa -> chn x  and # of stops -1 + 1 = 0
curstop = stop: stop there

heap = [(0, -1, src)]  # Price | # of stops used | City

while heap:
    cur_price, cur_stop, cur_city = heappop(heap)
    if cur_city == dst:
        return cur_price
  # if cur_stop == total_stops and we haven't reach dst, path is invalid.
    if cur_stop < total_stops:
        for neighbor, neighbor_price in dic[cur_city]:
            heappush(heap, (cur_price + neighbor_price, cur_stop + 1, neighbor))
return -1
```

▼ More Reading

bwv988 ★ 238 Last Edit: January 16, 2019 1:25 AM

Some explanation.
The key difference with the classic Dijkstra algo is, we don't maintain the global optimal distance to each node, i.e. ignore below optimization:
`alt ← dist[u] + length(u, v)`
`if alt < dist[v]:`
Because there could be routes which their length is shorter but pass more stops, and those routes don't necessarily constitute the best route in the end. To deal with this, rather than maintain the optimal routes with 0..K stops for each node, the solution simply put all possible routes into the priority queue, so that all of them has a chance to be processed. IMO, this is the most brilliant part.
And the solution simply returns the first qualified route, it's easy to prove this must be the best route.

57 Show 6 replies Reply Share Report

overmars34 ★ 39 June 6, 2019 1:02 PM

For dummies like me who spend a lot of time understanding this brilliant code :)

1. put all flights into a `prices` map -> `Map<Integer, Map<Integer, Integer>>`
   // source city : Map<destination city, price>
2. init a min pq -> each object in pq should be an int array with
   `top[0]` = current total price
   `top[1]` = current source city
   `top[2]` = max distance to destination allowed
   pq compares each object by total price so far
3. add original source city to pq with price = 0 & distance allowed = k + 1
4. while exists cities to explore
   --> get min object then remove it from pq
   --> get current total price, current source city & distance to destination allowed from min object
   --> if current source == destination (obviously distance from original source to current source [which is destination] is less than k) -> return current total price
   else find (from prices map) all connected flights that fly from current source + calculate new price, new current source & new distance + add them to pq
5. If no city left to explore and no flight that fits criteria found till now, return -1

21 Reply

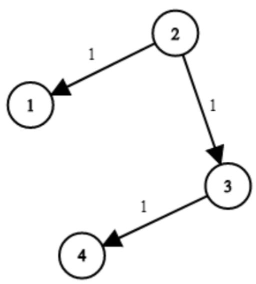
▼ **Network Delay Time**

There are N network nodes, labelled 1 to N.

Given times, a list of travel times as directed edges times[i] = (u, v, w), where u is the source node, v is the target node, and w is the time it takes for a signal to travel from source to target.

Now, we send a signal from a certain node K. How long will it take for all nodes to receive the signal? If it is impossible, return -1.

Example 1:




```
Input: times = [[2,1,1],[2,3,1],[3,4,1]], N = 4, K = 2
Output: 2
```

▼ Code **(Final)**

Time Complexity: `O(V + ElogE)` | Space Complexity: `O(V + E)`

```
from collections import defaultdict
from heapq import heappush, heappop
class Solution:
    def networkDelayTime(self, times, N, src):


        time -> (start, end, dist)


        graph = defaultdict(list)
        for start, end, time in times:
            graph[start].append([end,time])
        heap = [(0 , src)] # distance from src to current vertext | vertex
        visited = {}  # key | val -> vertext | distance from src to current vertext

        while heap:
            #1. pop things out
            dist, node = heappop(heap)


            if node in visited: continue

            visited[node] = dist
            for neighbor, neighbor_dist in graph[node]:
                heappush(heap , (dist + neighbor_dist, neighbor))


        return max(visited.values()) if len(visited) == N else -1
```

▼ **The Maze II**

There is a ball in a maze with empty spaces and walls. The ball can go through empty spaces by rolling up, down, left or right, but `it won't stop rolling until hitting a wall.` When the ball stops, it could choose the next direction.

Given the ball's start position, the destination and the maze, find the shortest distance for the ball to stop at the destination. The distance is defined by the number of empty spaces traveled by the ball from the start position (excluded) to the destination (included). If the ball cannot stop at the destination, return -1.

The maze is represented by a binary 2D array. 1 means the wall and 0 means the empty space. You may assume that the borders of the maze are all walls. The start and destination coordinates are represented by row and column indexes.

Example 1:

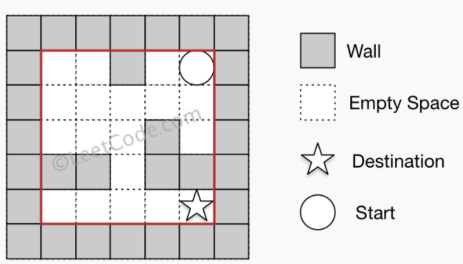
```
Input 1: a maze represented by a 2D array

0 0 1 0 0
0 0 0 0 0
0 0 0 1 0
1 1 0 1 1
0 0 0 0 0

Input 2: start coordinate (rowStart, colStart) = (0, 4)
Input 3: destination coordinate (rowDest, colDest) = (4, 4)

Output: 12

Explanation: One shortest way is : left -> down -> left -> down -> right -> down -> right.
             The total distance is 1 + 1 + 3 + 1 + 2 + 2 + 2 = 12.
```



Example 2:

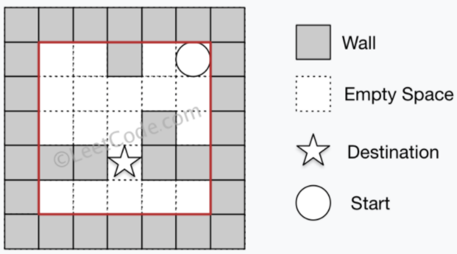
```
Input 1: a maze represented by a 2D array

0 0 1 0 0
0 0 0 0 0
0 0 0 1 0
1 1 0 1 1
0 0 0 0 0

Input 2: start coordinate (rowStart, colStart) = (0, 4)
Input 3: destination coordinate (rowDest, colDest) = (3, 2)

Output: -1

Explanation: There is no way for the ball to stop at the destination.
```



▼ Code (Final)

💡 Time Complexity: `O(mn + log()mn)` | Space Complexity: `O(mn)`

```
from heapq import heappush, heappop
class Solution:
    def shortestDistance(self, maze, start, end):
        visited = {} # key : val -> (x , y) : distance from src
        src_x, src_y = start[0], start[1]
        heap = [(0 , src_x, src_y)] # distance | x , y

        while heap:
            dist, x, y = heappop(heap)
            # Terminating Contiditon
            if [x, y] == end: return dist
            if (x , y) in visited: continue

            visited[(x , y)] = dist



            for dx, dy in [(1 , 0), (-1 , 0), (0 , 1), (0 , -1)]:
                new_x = x
                new_y = y
                count = 0
                while 0 <= new_x + dx < len(maze) and 0 <= new_y + dy < len(maze[0]) and maze[new_x + dx][new_y + dy] != 1:
                    new_x += dx
                    new_y += dy
                    count += 1
                heappush(heap, [dist + count, new_x, new_y])
        return -1
```

▼ More Reading