Dijkstra's algorithm for finding the shortest path from a given vertex in an undirected graph

[1]Turdiev S.S.,   [2]Qutbaev A.B.
[1]Tashkent State Transport University, [2]Novosibirsk State University

e-mail: s.turdiev@alumni.nsu.ru, aydosqutbaev@gmail.com

Abstract: Finding an optimal path in a graph. This task is quite common in everyday life and in the world of technology. Dijkstra's Algorithm works on oriented (with some additions and on undirected) graphs, and is designed to search for the shortest paths between a given vertex and all other vertices in the graph. on an undirected graph, where we will search for the shortest path using Dijkstra's Algorithm.
Keywords: graph, Caley graph, Star graph, Dijkstra Algorithm, Algorithm, Python, shortest paths, vertices, edges

Introduction

As a rule, a graph is denoted as a set of vertices and edges $G=(V,E)$ where the number of edges can be given by m and the number of vertices by n.

The shortest path problem is the problem of finding the shortest path (chain) between two points (vertices) in a graph, in which the sum of weights of the edges composing the path is minimised.

There are different formulations of the shortest path problem:

- Shortest path problem to a given destination. It is required to find the shortest path to a given destination vertex t that starts at each vertex of the graph (except t). By reversing the direction of each edge belonging to the graph, this problem can be reduced to a single source vertex problem (in which the shortest path from a given vertex to all other vertices is found).

- The problem about the shortest path between a given pair of vertices. It is required to find the shortest path from a given vertex u to a given vertex v.

- The problem about the shortest path between all pairs of vertices. It is required to find the shortest path from each vertex u to each vertex v. This problem can also be solved using an algorithm designed to solve the problem about one initial vertex, but it is usually solved faster.

The most popular algorithms for solving the problem of finding the shortest path in a graph:

Dijkstra's algorithm finds the shortest path from one vertex of a graph to all other vertices. The algorithm works only for graphs without edges of negative weight.

Bellman-Ford algorithm finds the shortest paths from one vertex of the graph to all other vertices in a weighted graph. The weight of edges can be negative.

The *A** search algorithm finds the least cost route from one vertex (initial) to another (target, final) using the first best match algorithm in the graph.

The Floyd-Warshell algorithm finds the shortest paths between all vertices of a weighted directed graph.

Johnson's algorithm finds the shortest paths between all pairs of vertices of a weighted directed graph (there must be no cycles with negative weight).

Lee's algorithm (wave algorithm) is based on the breadth-first search method. It finds the path between vertices s and t of the graph (s is not the same as t), containing the minimum number of intermediate vertices (edges). The main application is tracing of electrical connections on microchip crystals and printed circuit boards. It is also used to find the shortest distance on the map in strategy games.
Finds the shortest distance from the starting vertex to the other vertices in an unweighted graph (all edges have length 1). (If the graph is undirected, such an adjacency matrix is symmetric with respect to the main diagonal).

```
A = [[0, 1, 1, 1, 0], [1, 0, 0, 1, 1], [1, 0, 0, 1, 1],
[1, 1, 1, 0, 1], [0, 1, 0, 1, 0]]
for x in A:
    print(x)
W = [[]] * 6
W[1] = [2, 3, 4]
W[2] = [1, 4, 5]
W[3] = [1, 4]
W[4] = [1, 2, 3, 5]
W[5] = [2, 4]
print(W)
```

For an undirected graph it is similar. In this method it is convenient to search edges leaving vertex i (it is just a list W[i]), but it is difficult to check if there is an edge between vertices i and j - to do this you need to check if the number j is contained in the list W[i]. But in Python, you can make this part more efficient by replacing lists with sets - then checking if an edge exists between two vertices will also be done for .
With the help of adjacency matrices and adjacency lists we can also represent undirected graphs. In the case of adjacency matrix A[i][j] will be equal to 1 if there is an edge starting at vertex i and ending at vertex j. In the case of adjacency lists, the presence of an edge from vertex i to vertex j means that there is a number j in the list W[i].

Read graphs

```
n, m = [int(x) for x in input().split()]#n is the number of vertices, m is the
number of edges
# if the numbering of vertices starts from one
A = [[0] * (n + 1) for i in range(n + 1)]
for i in range(m):
u, v = [int(x) for x in input().split()]
A[u][v], A[v][u] = 1, 1 #for an undirected graph
for x in A:
print(x)
```

DFS (depth-first search)
allows to construct a traversal of an oriented or undirected graph in which all vertices accessible from the initial vertex are visited.
The difference between depth-first and breadth-first search is that (in case of undirected graph) the result of depth-first search algorithm is some route, following which it is possible to traverse sequentially all vertices of the graph accessible from the initial vertex. This makes it fundamentally different from breadth-first search, where many vertices are processed simultaneously; in depth-first search only one vertex is processed at each moment of the algorithm execution. On the other hand, depth-first search does not find the shortest paths, but it is applicable in situations where the whole graph is unknown, but is explored by some automated device.
The obvious sequence of actions of the researcher is as follows:
1. Go to some adjacent vertex.
2. Go around everything that is accessible from this vertex.
3. Return to the initial vertex.
4. Repeat the algorithm for all other vertices adjacent from the initial vertex.
More detailed algorithm:
1. Go to some adjacent vertex not previously visited.
2. Run a depth-first traversal algorithm from this vertex
3. Return to the starting vertex.
4. Repeat steps 1-3 for all adjacent vertices not visited earlier.
Since the purpose of depth-first traversal is often to build a depth-first traversal tree, we will immediately store a predecessor for each vertex.

```
isited = [False] * (n + 1)
prev = [None] * (n + 1) #prev[u] - is the vertex from which we came to u
def dfs(start, visited, prev, W):
    visited[start] = True
    for u in W[start]: #W - adjacency list for vertices
        if not visited[u]:
            prev[u] = start
            dfs(u)
```

Extraction of connectivity components
Some set of vertices of a graph such that for any two vertices from this set there exists a path from one vertex to another, and there exists no path from a vertex of this set to a vertex not from this set.

```
isited = [False] * (n + 1)
def dfs(start):
    visited[start] = True
    for v in W[start]:
```

```
                if not visited[v]:
                    dfs(v)
        ncomp = 0
        for i in range(1, n + 1):
            if not visited[i]:
                ncomp += 1
                dfs(i)
        print(ncomp)
```

Checking a graph for bipartite

A graph is called bipartite if its vertices can be split into two sets such that the ends of each edge belong to different sets. In other words, it is possible to paint the vertices of a graph in two colours such that the ends of each edge are painted in a different colour.

Let's modify the DFS algorithm so that it will check the graph for bipartite and build a graph painted in two colours (if it is bipartite). To do this, we will replace the Visited list with the Color list, where we will store the value 0 for unvisited nodes, and for visited nodes we will store the value 1 or 2 - its colour. The DFS algorithm for each edge will check the colour of the final vertex of that edge. If the vertex has not been visited, it is coloured in a colour unequal to the colour of the current vertex. If a vertex has been visited, the edge is either skipped if its ends are multicoloured, and if its ends are the same colour, it is marked that the graph is not bipartite (the IsBipartite variable is assigned the value False, by its value we can judge whether the graph is bipartite).

```
color = [0] * (n + 1)
IsBipartite = True

def dfs(start):
    for u in W[start]:
        if color[u] == 0:
            color[u] = 3 - color[start]
            dfs(u)
        elif color[u] == color[start]:
            global IsBipartite
            IsBipartite = False

for i in range(1, n + 1):
    if color[i] == 0:
        color[i] = 1
        dfs(i)
print(IsBipartite)
```

Finding a cycle in a directed graph

A cycle in a directed graph can be detected by the presence of an edge leading from the current vertex to a vertex that is currently being processed, i.e. the DFS algorithm has entered such a vertex but has not yet left it. In such a DFS algorithm we will paint vertices in three colours. Colour 0 ("white") will denote unvisited vertices. Colour 1 ("grey") will denote vertices in the process of processing, and colour 2 ("black") will denote already processed vertices. A vertex is painted in colour 1 when entering this vertex and in colour 2 when leaving it. A cycle in the graph exists if the DFS algorithm detects an edge whose end is painted in colour 1.

```
color = [0] * (n + 1)
```

```
CycleFound = False
def dfs(start):
    color[start] = 1
    for u in W[start]:
        if color[u] == 0:
            dfs(u)
        elif color[u] == 1:
            global CycleFound
            CycleFound = True
    color[start] = 2

for i in range(1, n + 1):
    if color[i] == 0:
        dfs(i)
```

Topological sorting

Finally, another important application of depth-first search is topological sorting. Suppose that a directed graph contains no cycles. Then the vertices of this graph can be ordered in such a way that all edges go from vertices with smaller number to vertices with larger number. For topological sorting of the graph it is enough to run the DFS algorithm, adding a vertex to the end of the list with the answer when leaving a vertex. After the end of the algorithm, expand the list with the answer in the opposite order.

```
visited = [False] * (n + 1)
ans = []
def dfs(start):
    visited[start] = True
    for u in W[start]:
        if not visited[u]:
            dfs(u)
    ans.append(start)
for i in range(1, n + 1):
    if not visited[i]:
        dfs(i)
ans = [0] + ans[::-1]
print(ans)
```

Searching for bridges

A bridge is an edge whose removal splits the graph into two connectivity components.

The depth-first search algorithm allows us to find all bridges in a connected graph for one DFS, i.e. for complexity *O(n).*

We suspend the graph from some vertex, run a DFS from that vertex. The DFS will build a graph traversal tree, finding backward edges - edges that go from the current vertex to the vertex that is currently being processed. To each vertex u we compare the value $h(u)$ - its depth in the traversal tree.

In addition, to each vertex we compare the value of the function *f(u)*, where *f(u)* is the minimum value of *h* (v) for all vertices , which are reachable from a vertex in the traversal tree and also reachable by traversing one back edge from any descendant in the traversal tree.

Then an edge uv is a bridge if $f(v) > h(u)$.

```
void dfs(int u, int parent, int curr_h, vector <vector<int> > & g, vector
<bool> & visited, vector<int> & h, vector<int> & f)
```

```
    {
        h[u] = curr_h++;
        f[u] = h[u];
        visited[u] = true;
        for (auto v : g[u])
        {
            if (v == parent)
                continue;
            if (!visited[v])
            {
                dfs(v, u, curr_h, g, visited, h, f);
                f[u] = min(f[u], f[v]);
                if (f[v] > h[u])
                { // Найден мост u-v
                }
            }
            else
                f[u] = min(f[u], h[v]);
        }
    }
```

classical BFS algorithm in undirected graph. Finds the shortest distance from the starting vertex to the other vertices in an unweighted graph (all edges have length 1).

```
n, m = [int(x) for x in input().split()]
W = [[] for _ in range(n)]
for _ in range(m):
    i, j = [int(x) - 1 for x in input().split()]
    W[i].append(j)
    W[j].append(i)
start = 0
dist = [-1] * n
dist[start] = 0
queue = [start]
while queue:
    u = queue.pop()
    for v in W[u]:
        if dist[v] == -1:
            dist[v] = dist[u] + 1
            queue = [v] + queue
print(dist)
```

finding the number of paths from the starting vertex to all other vertices in a directed graph.

```
n, m = [int(x) for x in input().split()]
W = [[] for _ in range(n)]
for _ in range(m):
    i, j = [int(x) - 1 for x in input().split()]
    W[i].append(j)

start = 0
nums = [0] * n
```

```
nums[start] = 1
queue = [start]
while queue:
    u = queue.pop()
    for v in W[u]:
        nums[v] += 1
        queue = [v] + queue
print(nums)
```

The basic principles of Dijkstra's algorithm:

1. Start from the initial vertex and set its score equal to 0. The other vertices are assigned by infinite score.

2. Go through the vertices of the graph one by one, starting from the vertex with the lowest score.

3. For each vertex under consideration, update the estimates of all adjacent vertices if the new path through the current vertex is shorter than the previous shortest path.

4. After updating the estimates for all adjacent vertices, move to the next vertex with the smallest estimate.

5. Repeat this process until all vertices of the graph are viewed.

The main steps in the implementation of Dijkstra's algorithm:

1. Graph creation: in Python, we can represent a graph using a dictionary, where keys are vertices and values are their adjacent vertices and edge values.

2. Initialisation: setting the initial vertex with score 0 and all other vertices with infinite score.

3. Cycle: for each vertex in the graph, starting from the vertex with the lowest score, update the scores of all adjacent vertices if a new shortest path is found.

4. Shortest path retrieval: after the algorithm is finished, we can obtain the shortest path from the initial vertex to any other vertex by following the predecessors.

Dijkstra's algorithm for finding the shortest path from a given vertex in an undirected graph. The graph is defined by a matrix of weights.

```
n = int(input())
W = [[int(x) for x in input().split()] for _ in range(n)]
start = int(input()) - 1
```

```
INF = 1e8
visited = [False] * n
dist = [INF] * n
dist[start] = 0

def gofrom():
index = 0
distmin = INF
for i in range(n):
if dist[i] < distmin and visited[i] == False:
distmin = dist[i]
index = i
return index

while False in visited:
u = gofrom()
for v in range(n):
if W[u][v] != 0 and (not visited[v]):
dist[v] = min(dist[v], dist[u] + W[u][v])
visited[u] = True
print(dist)
```

```
import math
def arg_min(T, S):
   amin = -1
   m = math.inf  # maximum value
   for i, t in enumerate(T):
       if t < m and i not in S:
           m = t
           amin = i
   return amin
D = ((0, 3, 1, 3, math.inf, math.inf),
    (3, 0, 4, math.inf, math.inf, math.inf),
    (1, 4, 0, math.inf, 7, 5),
    (3, math.inf, math.inf, 0, math.inf, 2),
    (math.inf, math.inf, 7, math.inf, 0, 4),
    (math.inf, math.inf, 5, 2, 4, 0))
N = len(D)  # number of vertices in the graph
T = [math.inf]*N   # last row of the table
v = 0       # start vertex (numbering from zero)
S = {v}     # viewed peaks

T[v] = 0    # zero weight for the starting vertex

M = [0]*N   # optimal links between vertices

while v != -1:         # loop until we look through all vertices
```

```
    for j, dw in enumerate(D[v]):   # search all connected vertices with vertex v
        if j not in S:          # if the top has not yet been viewed
            w = T[v] + dw
            if w < T[j]:
                T[j] = w
                M[j] = v       # connect vertex j with vertex v
    v = arg_min(T, S)          # select the next node with the lowest weight
    if v >= 0:                 # another peak has been chosen
        S.add(v)               # add a new vertex into consideration
#print(T, M, sep="\n")
# formation of an optimal route:
start = 0
end = 4
P = [end]
while end != start:
    end = M[P[-1]]
    P.append(end)
print(P)
```

Conclusion

In the process of using Dijkstra's algorithm in the Python programming language to find the shortest path in a graph, I was convinced of its efficiency and advantages. Dijkstra's algorithm allows to find the optimal path accurately and reliably and is the basis for many other algorithms related to working with graphs. Dijkstra's algorithm in my projects related to the analysis and processing of graphs, as it allows to solve complex problems with minimal time and resources. Combining it with advanced features, such as the use of priority queueing or the A* algorithm, will only increase its efficiency. Algorithms and graph structures that they are the foundation of many computer science and outdoor techniques. Use Dijkstra's algorithm in projects to discover the most optimal paths, solve complex problems, and achieve goals.

References

[1] Zhan, F. Benjamin; Noon, Charles E. (February 1998). "Shortest Path Algorithms: An Evaluation Using Real Road Networks". *Transportation Science*. **32** (1): 65–73.
[2] S. Lakshmivarahan S., J. S. Jwo, S. K. Dhall. Symmetry in interconnection networks based on Cayley graphs of permutation groups: a survey. // Parallel Comput. – V. 19. – 1993. – P. 361–407.
[3] S. B. Akers, B. Krishnamurthy. A group-theoretic model for symmetric interconnection networks. // IEEE Trans. Comput. – V. 38.
– 1989. P. 555–566.
[4] J. S. Jwo, S. Lakshmivarahan, S. K. Dhall. Embedding of cycles and grids in star graphs. // J. Circuits Syst. Comput. – V. 1. – 1991. –P. 43–74.

[5] Thorup, Mikkel (2000). "On RAM priority Queues". *SIAM Journal on Computing*. **30** (1): 86–109

[6] Y.-Q. Feng. Automorphism groups of Cayley graphs on symmetric groups with generating transposition sets. // Journal of Combinatorial Theory, Series B. – V. 96. – 2006. – P. 67—72.

[7] https://www.hse.ru/data/2012/02/08/1262556187/Алексеев%20В.Е.Таланов%20В.А.Графы%20Модели%20вычислений%20Структуры%20данных%20(2004).pdf

[8] https://cp-algorithms.com/graph/dijkstra.html

ISSN 2521-3261 (Online)/ ISSN 2521-3253 (Print)
https://journalofresearch.eu/