Visualization of Dijkstra’s Algorithm

> CS201 Discrete Mathematics - Final Project

Name: Liu Leqi (刘乐奇)
SID: 12011327

Introduction

`Dijkstra's algorithm` is an algorithm to find the shortest paths from source vertex **s** to any other vertices in a graph. This graph can be either directed or undirected; either weighted or unweighted. However, the weights must be non-negative.

`Dijkstra's algorithm` is a kind of greedy algorithm. Operating on the graph G = (V, E), it starts from source vertex and an empty vertex set S. Every iteration it extracts a vertex from V to S that has a shortest path to the source.

One of the core operations in `Dijkstra's algorithm` is called **edge relaxation**. Given an edge (v, u), we relax it as

```
if distance[u] > distance[v] + weight(v, u):
    distance[u] = distance[v] + weight(v, u)
```

Implementation

Pseudo-code

```
DIJKSTRA(G,w,s):
    INITIALIZE-SINGLE-SOURCE(G,s)
    S <- Ø
    Q <- V[G]
    while Q != Ø:
        u <- EXTRACT-MIN(Q)
        S <- S∪{u}
        for each vertex v in Adj[u]:
            RELAX(u,v,w)
```

Python code

```python
def dijkstra(s):
    distance[s] = 0
    q = PriorityQueue(V)
    q.put_nowait(s)
    while q.qsize()!=0:
        v = q.get_nowait()
        visited[v] = True
        for u in range(V):
            if graph[v][u] == math.inf or visited[u]:
                continue
            else:
                q.put_nowait(u)
            if distance[u] > distance[v] + graph[v][u]:
                distance[u] = distance[v] + graph[v][u]
```

Java code

```java
public static void dijkstra(s){
    distance[s] = 0;
    PriorityQueue<Integer> q =
        new PriorityQueue<>((o1, o2) -> o1 - o2);
    q.offer(s);
    while (!q.isEmpty()) {
        int v = q.poll();
        visited[v] = true;
        for (int u = 1; u <= N; u++) {
            if (graph[v][u] != null && !visited[u]) {
                q.offer(u));
                long cost = distance[v] + graph[v][u];
                if (distance[u] > cost) {
                    distance[u] = cost;
                }
            }
        }
    }
}
```

Running time

Implementing the priority queue with binary heap, the running time of Dijkstra’s algorithm is:

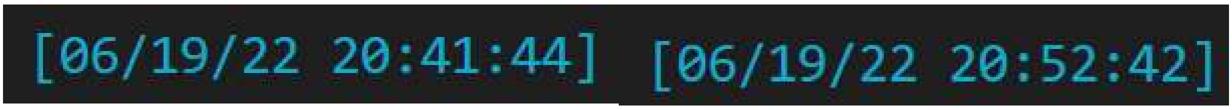
$$O((|V| + |E|) * \log|V|)$$

How to perform better

- Use adjacent list to present the graph instead of adjacent matrix, for example, `list` in Python or `ArrayList` in Java, or a technique called 链式前向星 (I cannot find its English name). Then it will perform better with less running time, especially in sparse graph.
- Use Fibonacci heap to implement the priority queue. It can reduce the running time to be $O(|V|\log|V| + |E|)$.

Visualization of Dijkstra’s Algorithm

It was packed within the zip file.



I used a library of Python called `manim` to generate the visualization video. In this video, every step is shown in detail. The library performed poorly in rendering. It took about 10 minutes to render out the video.

Reference

[1] Introduction to Algorithms (Second Edition)