Lecture 12 – Dijkstra’s Running Time

Sanjoy Dasgupta
Russell Impagliazzo
Ragesh Jaiswal
CSE101, Spring 2020, Week-03
procedure dijkstra(G,l,s)

for u in V:
    dist[u] = ∞
-dist[s] = 0
H = makequeue(V)  // key = dist[

while H is not empty:
    u = deletemin(H)
    for each edge (u,v) in E:
        if dist[v] > dist[u] + l(u,v):
            dist[v] = dist[u] + l(u,v)
            decreasekey(H,v)

Time:
O(V + E) +
V x deletemin +
V x insert +
E x decreasekey

Depends on priority queue implementation:
eg. binary heap O(E log V)
Linked list implementation

Linked list, unordered

insert:

decreasekey:

deletemin:
Binary heap

Complete binary tree: filled in row by row, left-to-right

Rule: each node’s value is smaller than that of its children

Height $\leq \log_2 n + 1$
Binary heap

insert(7)
decreasekey(19 -> 6)
delete_min
d-ary heap

Same as a binary heap, but with d children...

height:
insert
deleteMin
Running time of Dijkstra’s algorithm

<table>
<thead>
<tr>
<th></th>
<th>insert, decreasekey</th>
<th>deletemin</th>
<th>(V \times \text{deletemin} + (V+E) \times \text{insert} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>linked list</td>
<td>(O(1))</td>
<td>(O(V))</td>
<td>(O(V^2))</td>
</tr>
<tr>
<td>binary heap</td>
<td>(O(\log V))</td>
<td>(O(\log V))</td>
<td>(O((V+E) \log V))</td>
</tr>
<tr>
<td>d-ary heap</td>
<td>(O(\log_d V))</td>
<td>(O(d \log_d V))</td>
<td>(O((dV + E) \log_d V))</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>(O(1)) amortized</td>
<td>(O(\log V))</td>
<td>(O(E + V \log V))</td>
</tr>
</tbody>
</table>

Which is best depends on sparsity of graph: ratio \(E/V\) (average degree).

**Linked list vs. binary heap**

- Dense graph: \(E = \Omega(V^2)\)
  - Linked list is better: \(O(V^2)\)
- Sparse graph: \(E = O(V)\)
  - Binary heap is better: \(O(V \log V)\)

**d-ary heap**

- Best choice \(d \approx E/V\)
- Dense: \(O(V^2)\)
- Sparse: \(O(V \log V)\)
- Intermediate: \(E = V^{1+c}\)
  - \(O(E/c)\), linear!
Dijkstra and negative edges

```plaintext
procedure dijkstra(G,l,s)
    for u in V:
        dist[u] = \infty
    dist[s] = 0
    H = makequeue(V)  // key = dist[]
    while H is not empty:
        u = deletemin(H)
        for each edge (u,v) in E:
            if dist[v] > dist[u] + l(u,v):
                dist[v] = dist[u] + l(u,v)
                decreasekey(H,v)
```

Basic principle of Dijkstra’s algorithm: the shortest path to any node only goes through nodes that are closer by.

Not true if negative edges are present!

In Dijkstra’s algorithm, dist[] values:
1. are never too small
2. get changed only when updating along an edge:

```plaintext
procedure update(edge (u,v))
    if dist[v] > dist[u] + l(u,v):
        dist[v] = dist[u] + l(u,v)
```

![Graph diagram with nodes a, b, c and edges labeled with distances 6, 8, and -6]