The runtime efficiency of three graph algorithms

1. Background
2. BFS
3. DFS
4. Dijkstra

1. Background

At start: a digraph *G = (V,E)* possibly with weights on edges

Q: What is the size of a digraph?
A: We need to use two numbers:

$$n = \text{the number of nodes} \qquad m = \text{the number of edges}$$

We assume: at most one edge from x to y.

For a digraph: $m \leq n^2$. Typically, m is much less than n^2.

Adjacency sum result

Let the nodes of a digraph be 1, 2, 3 ... n. Let the number of adjacent nodes to node i be r_i. Then

$$r_1 + r_2 + \ldots + r_n = m$$

We will consider upper bound on efficiency: results will be $O()$.

2. BFS

```
1   BREADTH–FIRST–SEARCH(s, G)
2   executes a breadth first search on graph G starting from
3   source node s
4
5   forEach node x in G
6     x.color = white, x.d = ∞, x.π =NIL
7   end
8
9   /* Give source node appropriate values. */
10  s.color = gray, s.d = 0
11  ▷ initialize a queue in Q
12  ENQUEUE(Q,s)
13  while Q is not empty
14    x = DEQUEUE(Q)
15
16    forEach node y in x.Adj
17      if y.color == white then
18        /* Node y is undiscovered. */
19        y.color = gray, y.d = x.d + 1, y.π = x
20        ENQUEUE(Q,y)
21      end
22    end
23    x.color = black
24  end
```

THREE LOOPS

forEach (lines 5-7)

Each node processed once.

line 6: $O(n)$.

while-loop (line 13)

Each node is added to Q at most once. Why?

- Node will be added to Q only when it is white (line 17).
- When node is added to Q, it has been colored gray.
- Inside the **while**-loop, a node's color is never set to white.

Consequence: the **while**-loop has at most n-iterations.

- line 14: $O(n)$ (assuming DEQUEUE's efficicency is $O(1)$)
- line 23: $O(n)$

forEach (lines 16-22)

Assume each node is added to Q. For each node we process all its adjacent nodes once.
Consequence: owing to adjacency sum result, total number of iterations of **forEach**-loop **for all nodes** is m.

- lines 17 and 19: $O(m)$
- line 20: $O(m)$ (assuming ENQUEUE's efficicency is $O(1)$)

Total efficiency of BFS: $O(n+n+n+m+m) = O(n+m)$

3. DFS

```
1     DEPTH–FIRST–SEARCH(s, G)
2     executes a depth first search on graph G starting from
3     source node s
4
5     forEach node x in G
6       x.color = white, x.π =NIL
7     end
8
9     ▷ initialize a stack in S
10    PUSH(S,s)
11    while S is not empty
12      x = POP(S)
13
14      /* If x is white, then it has not yet been discovered.*/
15      if x.color == white then
16      /* x is discovered. Put x back into the stack and
           investigate nodes adjacent to x.*/
17        x.color = gray, PUSH(S,x)
18
19          forEach node y in x.Adj
20              if y.color == white then
21                PUSH(S,y), y.π = x
22              else if y.color == gray then
23                      ▷ a cycle that includes edge (x,y) exists
24                  end
25              end
26            end
27      else
28        x.color = black
29      end
30    end
```

THREE LOOPS

forEach (lines 5-7)

Each node processed once: $O(n)$.

forEach (lines 16-22)

This **forEach**-loop is only ever run once for each node x. Why?

- Lines 17-26 only executed when $x.color$ is white at line 12.
- At line 17, a white node is colored gray.
- Inside **while**-loop, a node's color is never set to white.

Consequence: For each node x, the **forEach**-loop has one iteration for each node adjacent to x. Owing to adjacency sum result, the **forEach**-loop has a total of (at most) m-iterations.

- line 20: $O(m)$
- line 21: $O(m)$ (assuming PUSH's efficicency is $O(1)$)
- line 23: $O(m)$ (assuming operation at line 23 are $O(1)$)

while-loop (line 13)

This loop is run at most $m+1$ times. Why?

- Line 10 is run once and line 21 is run at most m times. Hence at most $m+1$ items are pushed onto the stack.
- At each iteration one item is taken off the stack (line 12).
- line 12: $O(m)$ (assuming POP's efficicency is $O(1)$)
- lines 15 and 28: $O(m)$

Total efficiency of DFS: $O(n+m+m+m+m) = O(n+m)$

4. Dijkstra

```
1   RELAX(x, y)
2   if y.d > x.d + w((x,y)) then
3       y.d = x.d + w((x,y)), y.π = x
4   end
```

```
1   DIJKSTRA(s, G)
2   forEach node x in G
3     x.color = white, x.d = ∞, x.π =NIL
4   end
5
6   s.color = gray, s.d = 0
7   ▷ initialize a priority queue Q
8   INSERT(Q,s,0)
9   while Q is not empty
10    x = EXTRACT–MIN(Q)
11    /* Test whether shortest path to x has been found. */
12    if x.color ≠ black then
13      forEach node y in x.Adj
14          y.old = y.d, RELAX(x, y)
15          if y.color == white then
16              /* Node y is undiscovered. */
17              y.color = gray, y.π = x
18              INSERT(Q,y,y.d)
19          else
20            if y.d < y.old and y.color ≠ black then
21              /* Take into account that y.old < y.d. */
22              INSERT(Q,y,y.d)
23            end
24          end
25      end
26      x.color = black
27    end
28  end
```

RELAX has efficiency of $O(1)$.

THREE LOOPS

forEach (lines 5-7)

Each node processed once: $O(n)$.

forEach (lines 16-28)

This **forEach**-loop is only ever run once for each node x. Why?

- If x is black at line 14, then the **forEach**-loop is not executed.
- If x is not black at line 14, it is set to black after lines 16–28 are executed.
- Inside **while**-loop, once a node is black, its color does not change.

Consequence: For each node x the **forEach**-loop has one iteration for each node adjacent to x. Owing to the adjacency sum result, the **forEach**-loop has a total of (at most) m-iterations.

Consequence: size of Q is at most m

- lines 17, 18: $O(m)$
- either line 21 or 24: $O(m \log_2 m)$ (assuming INSERT is done with heap)

while-loop (line 13)

This loop is run at most m times. Why?

- An item is inserted into priority Q only at lines 21 or 24. The **forEach**-loop has a total of (at most) m-iterations.
- At each iteration one item is taken removed from the priority queue Q (line 14).
- line 14: $O(m \log_2 m)$ (assuming EXTRACT-MIN is done with heap)
- line 15: $O(m)$

Total efficiency of Dijkstra: $O(n + m + m + m \log_2 m + m \log_2 m) = O(n + m \log_2 m)$

This efficiency assumes that the priority queue in Dijkstra is implemented using a heap.

The priority queue could also be implemented using a balanced binary search tree. The DIJKSTRA2 procedure assumes this.

```
1   DIJKSTRA2(s, G)
2   forEach node x in G
3     x.color = white, x.d = ∞, x.π =NIL
4   end
5
6   s.color = gray, s.d = 0
7   ▷ initialize a priority queue Q
8   INSERT(Q,s,0)
9   while Q is not empty
10    x = EXTRACT–MIN(Q)
11    forEach node y in x.Adj
12      y.old = y.d, RELAX(x, y)
13      if y.color == white then
14          /* Node y is undiscovered. */
15          y.color = gray, y.π = x
16          INSERT(Q,y,y.d)
17      else
18        if y.d < y.old and y.color ≠ black then
19          /* Take into account that y.old < y.d. */
20          REMOVE(Q,y,y.old)
21          INSERT(Q,y,y.d)
22        end
23      end
24      x.color = black
25    end
26  end
```

Size of priority queue Q is (at most) n.

line 16 or 20 or 21: $O(m \log_2 n)$