Dijkstra's algorithm and the shortest path problem

1. **Background**
2. **Data structures**
3. **Procedures**
4. **Priority queue issues**

1. Background

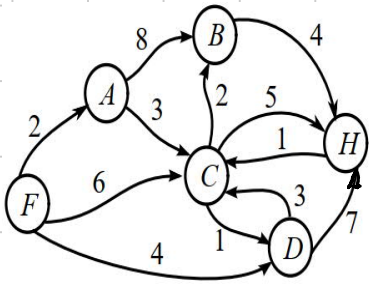
At start: a weighted digraph $G = (V,E)$ and a starting node (source node) s from the digraph

Goal: find shortest path from s to all nodes that are reachable from s

- Weight of edge (x,y) is $w((x,y))$.
- Weight $w((x,y))$ is interpreted as a distance.
- All weights are assumed to be non-negative.
- This problem is sometimes called the single-source shortest path problem.

Example

Paths from F to H and their total lengths

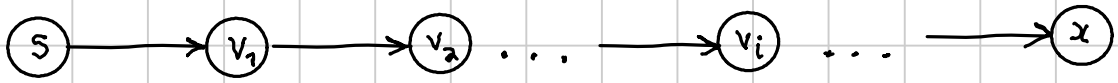


path	<F, D, H>	<F, A, B, H>	<F, A, C, D, H>
total length	11	14	13

□

Dynamic programming principle used in solving shortest path problem:

> Let the shortest length path from s to x be given by
>
> $$\langle s, v_1, v_2, \ \ldots \ x \rangle$$
>
> When v_i is in this path, then the subpath $\langle s, v_1, v_2, \ \ldots \ v_i \rangle$ is the shortest path from s to v_i.



Dynamic programming is not 'programming' in some computer language: C++, Java, etc.

Dynamic programming is an algorithm design technique.

Dijkstra's algorithm:

- solves solves single source shortest path problem using the dynamic programming
- follows BFS with addition of priority queue to take into account weights of edges
- produces shortest path tree

2. Data structures

To perform Dijkstra's algorithm, for each node we use the following attributes:

- $x.d$ length of shortest path x from source s to node x (thus far)
- $x.colour$ = color of node
- $x.\pi$ = parent of node x in shortest path tree
- $x.Adj$ set containing nodes that are adjacent to x

Significance of colors:

- *x.colour* = white: node *x* not yet discovered
- *x.colour* = gray: node *x* discovered, but shortest path not yet found
- *x.colour* = black: node *x* discovered and shortest path from *s* has been found

For each edge Dijkstra needs weight *w*((*x*,*y*)).

Dijkstra's algorithm maintains a priority queue *Q* containing pairs (x, *x.d*) for gray nodes.

In Dijkstra: node *x* priority = *x.d*.

Two basic operations:

- INSERT(Q, x, $x.d$): inserts pair $(x.d, x)$ into Q.
- EXTRACT-MIN(Q): extracts pair $(x.d, x)$ having smallest priority

Example

Start with empty priority queue: Q:

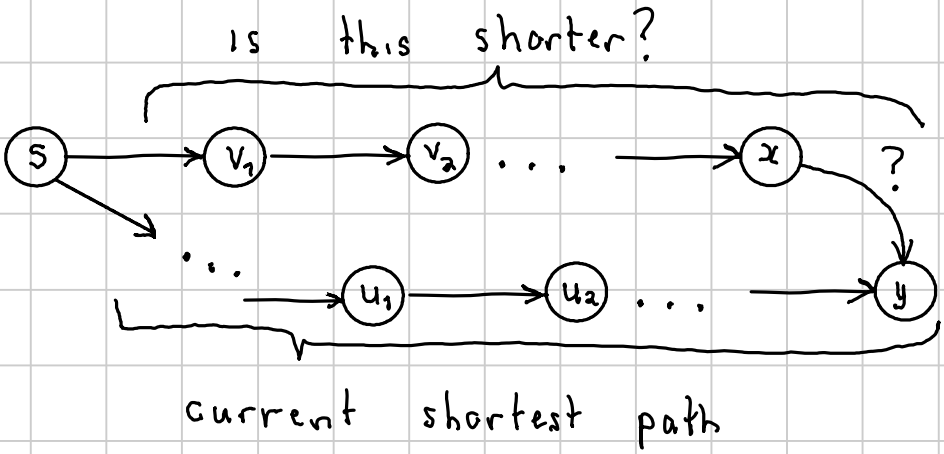
operation	Q
INSERT(Q, a, 6)	(6,a)
INSERT(Q, b, 8)	(6,a), (8,b)
INSERT(Q, b, 5)	(5,b), (6,a), (8,b)
EXTRACT-MIN(Q)	(6,a), (8,b)

□

3. Procedures

Dijkstra's algorithm uses procedure RELAX for detecting when a shorter path has been found.

```
1   RELAX(x, y)
2   When shorter path to y is found using edge (x,y), length y.d
3   and parent y.π are reset.
4
5   if y.d > x.d + w((x,y)) then
6   /* Shorter path than current path found via edge (x,y). */
7       y.d = x.d + w((x,y)), y.π = x
8   end
```




```
1     DIJKSTRA(s, G)
2     starting from source node s finds the shortest path to all
3     other nodes of  weighted graph G
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 priority queue Q
12    INSERT(Q,s,0)
13    while Q is not empty
14      x = EXTRACT-MIN(Q)
15
16      forEach node y in x.Adj
17          y.old = y.d, RELAX(x, y)
18          if y.color == white then
19              /* Node y is undiscovered. */
20              y.color = gray, y.π = x
21              INSERT(Q,y,y.d)
22          else
23              if y.d < y.old and y.color ≠ black then
24                /* Take into account that y.old < y.d. */
25                INSERT(Q,y,y.d)
26              end
27          end
28      end
29      x.color = black
30    end
```

Remarks

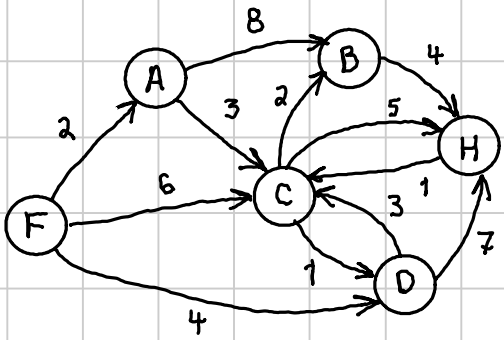
1. The pseudocode is very similar to BREADTH-FIRST-SEARCH. The main difference is that a priority queue is used in place of an ordinary queue.

2. Lines 23-26 are needed for updating *Q* when a shorter path is found.

3. Because of line 25 the same node may have several elements in *Q*.

Example

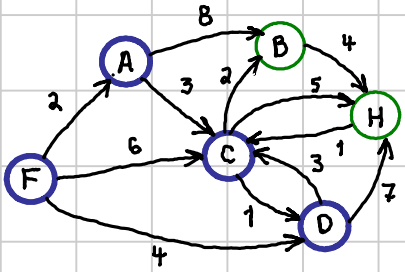
Execute DIJKSTRA with *s* = *F.*



black node

gray node

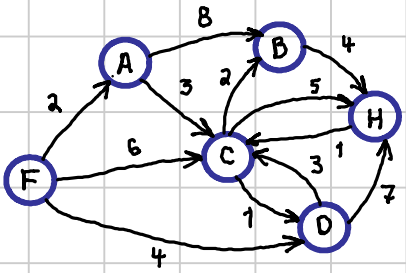
item	while-loop iteration 0	1	2	3	4
Q	(0, F)	(2, A), (4, D), (6, C)	(4, D) (5, C) (6, C) (10, B)	(5, C) (6, C) (10, B) (11, H)	(6, C) (7, B) (10, B) (10, H) (11, H)
x	—	F	A	D	C
A.d	∞	2	2	2	2
B.d	∞	∞	10	10	7
C.d	∞	6	5	5	5
D.d	∞	4	4	4	4
F.d	0	0	0	0	0
H.d	∞	∞	∞	11	10
A.π	NIL	F	F	F	F
B.π	NIL	NIL	A	A	C
C.π	NIL	F	A	A	A
D.π	NIL	F	F	F	F
F.π	NIL	NIL	NIL	NIL	NIL
H.π	NIL	NIL	NIL	D	C



item	while-loop iteration 4	6	8
Q	(6, C) (7, B) (10, B) (10, H) (11, H)	(10, B) (10, H) (11, H)	(11, H)
x	C	B	H
A.d	2	2	2
B.d	7	7	7
C.d	5	5	5
D.d	4	4	4
F.d	0	0	0
H.d	10	10	10
A.π	F	F	F
B.π	C	C	C
C.π	A	A	A
D.π	F	F	F
F.π	NIL	NIL	NIL
H.π	C	C	C

black node

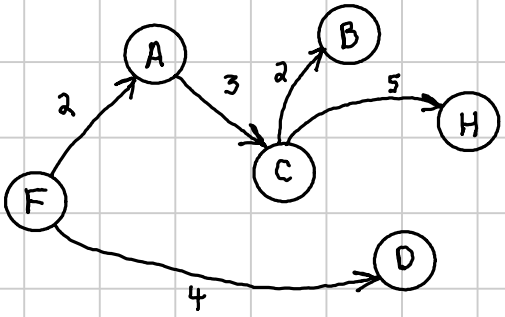
gray node



Final results:

x	A	B	C	D	F	H
$x.d$	2	7	5	4	0	10
$x.\pi$	F	C	A	F	NIL	C

Shortest path tree



4. Priority queue issues

A heap can be used as a priority queue.

- need min-heap

NOT allow

- heap does ~~allow~~ efficient changing of priorities: hence same node may appear several times

- EXTRACT-MIN will always remove element with smallest priority

- in C++ STL max-heap is std::priority_queue< >

- in C++ STL with <length, node> pair:
  std::priority_queue< std::pair<int, Node*> >

- to use max-heap as min-heap multiply priority by -1

Alternative to heap for priority queue: balanced binary search tree.

- in C++ STL balanced binary serach tree is std::set< >
- using with <length, node> pair: std::set< std::pair<int, Node*> >
- to change priority: remove old <length, node> pair and insert new <length, node> pair