Anthony Ventresque
anthony.ventresque@ucd.ie

Graphs

School of Computer Science, UCD Scoil na Ríomheolaíochta, UCD

Outline

- Graph?
- What is a graph
- Array-based Representations
- DFS and BFS
- Single-source shortest path: Dijkstra and MapReduce

Graphs

- Ubiquitous in modern society
  - Hyperlink structure of the Web
  - Social networks
    - Email flow
    - Friend patterns
  - Transportation networks

- Nodes and links can be annotated with metadata
  - Social network nodes: age, gender, interests
  - Social network edges: relationship type and importance

Real-world Problems to Solve

- Graph search
  - Friend recommendation in social networks, Expert finding in social networks
- Path planning
  - Route of network packets, Route of delivery trucks
- Graph clustering
  - Sub-communities in large graphs
- Minimum spanning tree
  - building roads and laying optical fiber
- Bipartite graph matching
  - job seekers looking for employment, singles looking for dates

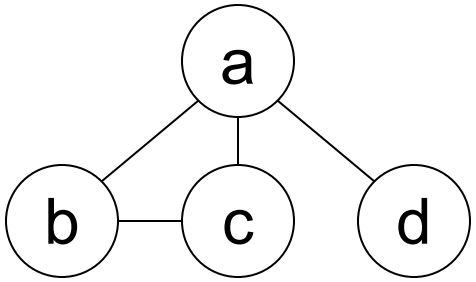
Real-world Problems to Solve

- A common feature: millions or billions of nodes and millions or billions of edges
- Real-world graphs are often sparse, the number of actual edges is far smaller than the number of possible edges

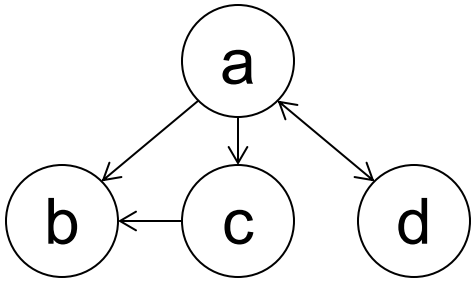
GRAPH "THEORY"

Graph

- The Graph ADT represents the mathematical concepts of
  - directed graph
  - undirected graph



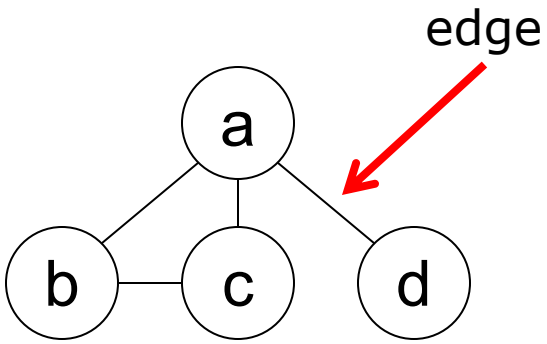
undirected graph



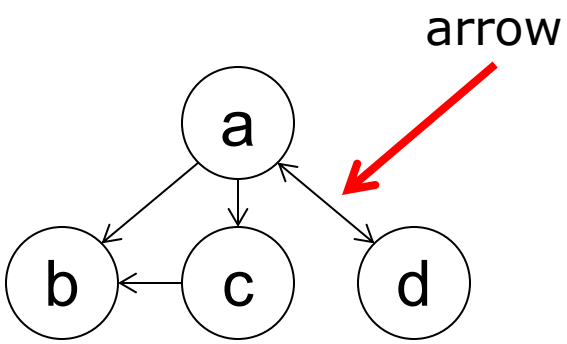
directed graph

Graph

- Connection between vertices:
  - undirected graph: edges, arcs
  - directed graph: arrows, directed edges, directed arcs



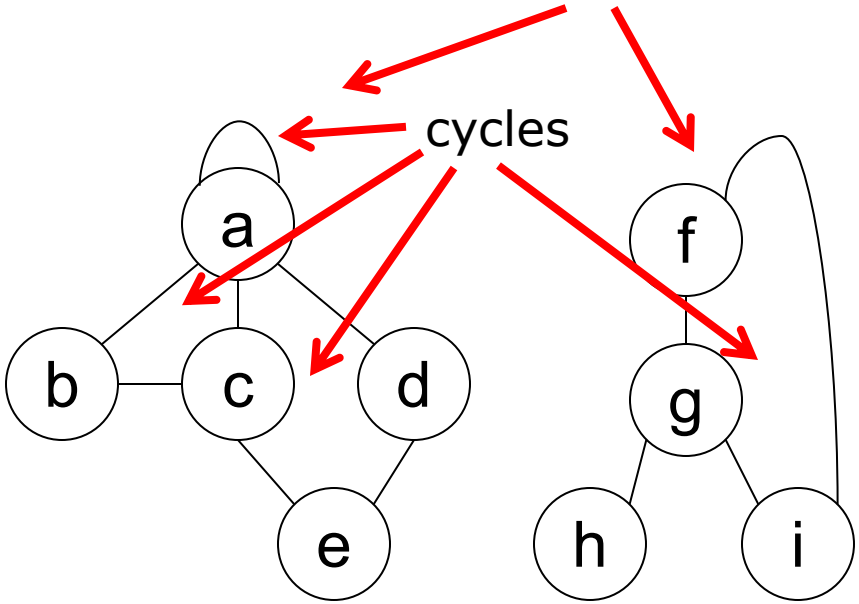
undirected graph



directed graph

Graph

- Contrary to Trees, Graphs
  - can have cycles
  - can be composed of different components

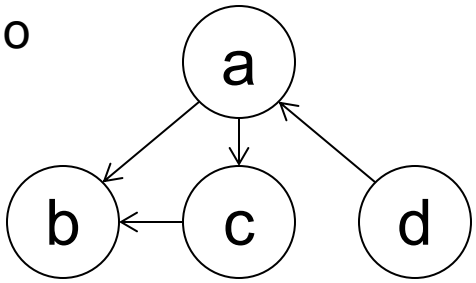


(undirected graph)

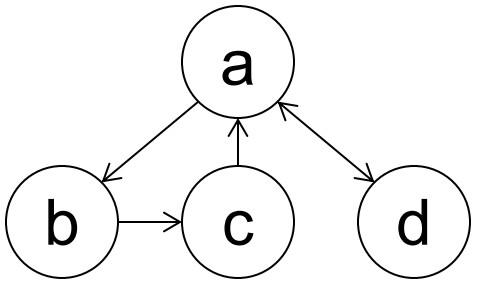
Connected Components

- Strongly connected components: directed graph with a path from each node to every other node

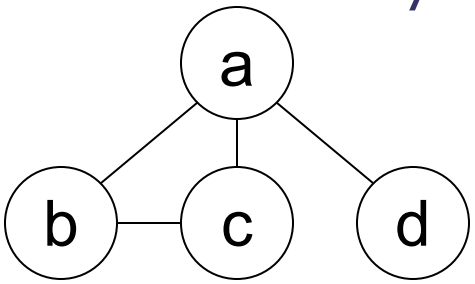
No (b cannot go anywhere; nobody can reach d)



yes



- Weakly connected component: directed graph with a path in the underlying undirected graph from each node to every other node



Weighted Graph

- Sometimes graphs have a weights on edges.
  - e.g., road netwok with the distance between any two points

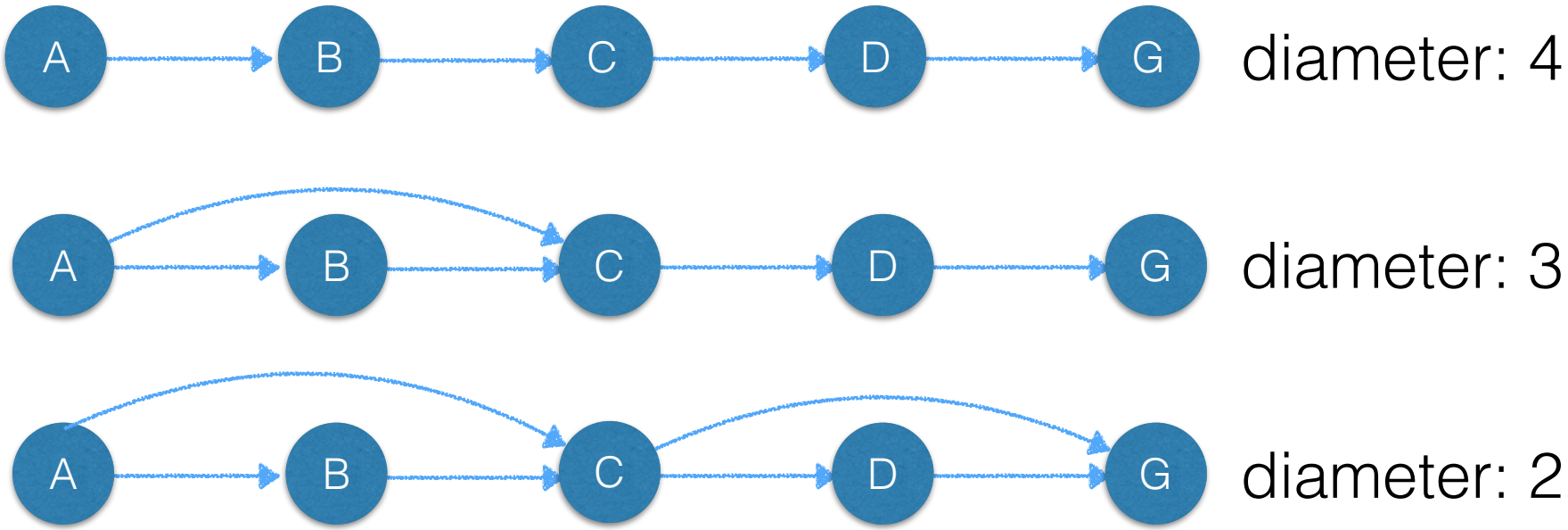
Example of Distance Graph

Example of Distance Matrix

Distances Chart (in KM) 1 KM = 0.6 Miles	Brantford	Collingwood	Elora	Fort Erie	Goderich	Grand Bend	Hamilton	Kitchener-W	London	Niagara Fal	Niagara-on-t	Oakville	Orangeville	Owen Soun	Point Pelee	Port Dover	Port Stanley	Sarnia	Port Elgin	St. Catharin	St. Thomas	Tobermory	Toronto	Windsor
Brantford	0	234	70	149	155	144	39	39	91	112	130	70	116	186	262	53	119	161	200	108	112	302	95	277
Collingwood	234	0	110	282	175	220	200	140	310	256	262	169	75	58	477	273	336	409	107	240	325	170	145	486
Elora	70	110	0	208	128	149	76	30	136	177	153	94	42	121	303	114	164	239	125	131	151	224	114	317
Fort Erie	149	282	208	0	282	316	95	166	237	32	55	118	193	305	402	117	264	336	300	43	252	408	153	413
Goderich	155	175	128	282	0	49	172	115	91	235	258	201	140	123	279	207	130	116	89	239	126	239	214	301
Grand Bend	144	220	149	316	49	0	186	105	60	371	296	236	182	196	248	169	99	67	136	274	96	267	203	218
Hamilton	39	200	76	95	172	186	0	64	126	63	75	36	98	179	298	60	165	196	200	53	146	294	63	319
Kitchener-Waterloo	39	140	30	166	115	105	64	0	91	123	145	88	77	144	263	84	130	161	153	122	125	263	98	284
London	91	310	136	237	91	60	126	91	0	193	216	157	168	214	182	109	39	91	188	145	29	330	175	193
Niagara Falls	112	256	177	32	235	371	63	123	193	0	28	92	161	242	377	109	224	259	274	17	224	357	133	382
Niagara-on-the-Lake	130	262	153	55	258	296	75	145	216	28	0	97	172	284	382	128	243	316	280	23	231	387	132	393
Oakville	70	169	94	118	201	236	36	88	157	92	97	0	79	191	323	109	163	255	208	76	171	294	40	333
Orangeville	116	75	42	193	140	182	98	77	168	161	172	79	0	101	341	161	207	256	151	150	189	295	74	361
Owen Sound	186	58	121	305	123	196	179	144	214	242	284	191	101	0	361	240	274	239	53	263	272	116	203	434
Point Pelee	262	477	303	402	279	248	298	263	182	377	382	323	341	361	0	301	172	187	342	360	177	535	361	74
Port Dover	53	273	114	117	207	169	60	84	109	109	128	109	161	240	301	0	98	234	242	106	93	341	153	311
Port Stanley	119	336	164	264	130	99	165	130	39	224	243	183	207	274	172	98	0	127	244	222	15	387	220	182
Sarnia	161	409	239	336	116	67	196	161	91	259	316	255	256	239	187	234	127	0	214	294	115	336	293	108
Port Elgin	200	107	125	300	89	136	200	153	186	274	280	208	151	53	342	242	244	214	0	258	232	122	239	318
St. Catharines	108	240	131	43	239	274	53	122	145	17	23	76	150	263	360	106	222	294	258	0	210	366	111	371
St. Thomas	112	325	151	252	126	96	146	125	29	224	231	171	189	272	177	93	15	115	232	210	0	376	208	187
Tobermory	302	170	224	408	239	267	294	263	330	357	387	294	295	116	535	341	387	336	122	366	376	0	296	439

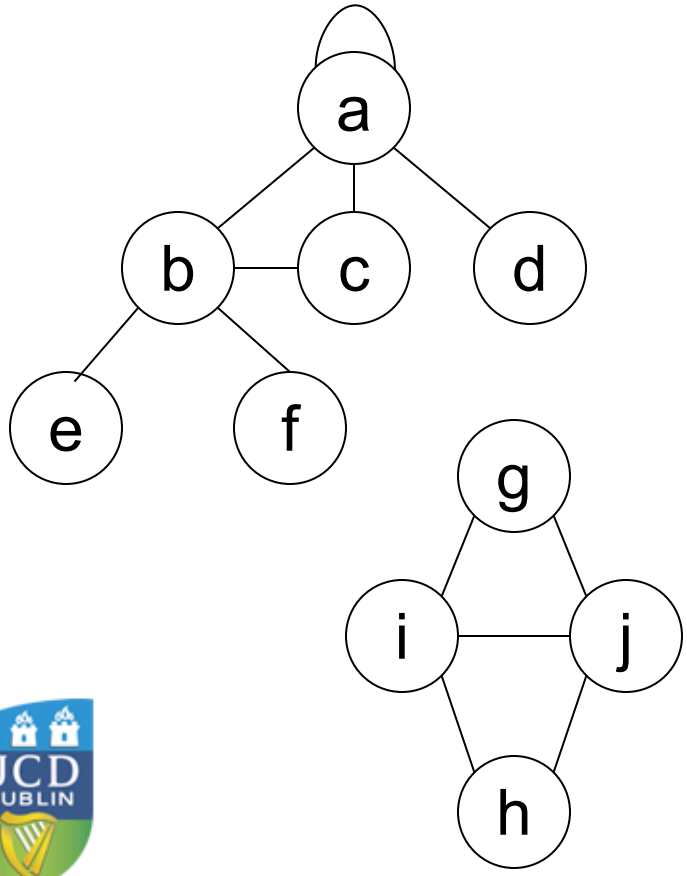
Diameter

- Longest shortest path in the graph



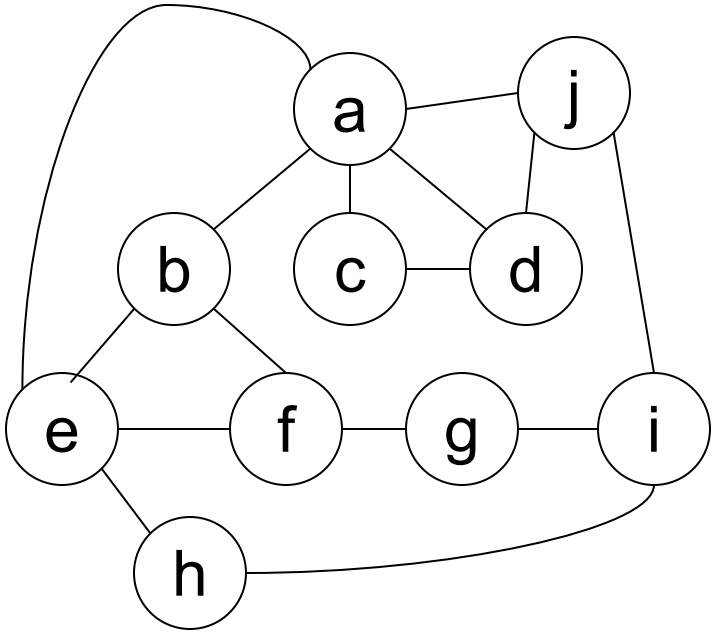
GRAPH REPRESENTATIONS

Adjacency List



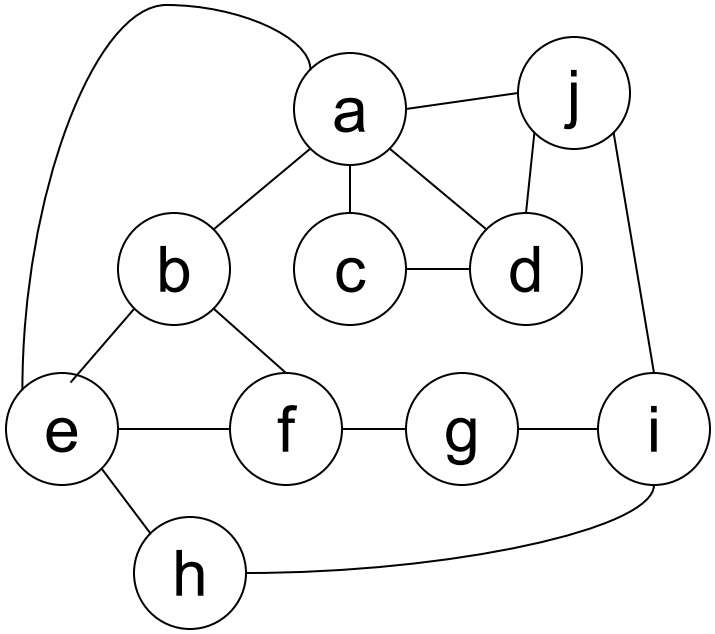
a	a, b, c, d
b	a, c, e, f
c	a, b
d	a
e	b
f	b
g	i, j
h	i, j
i	g, h, j
j	g, h, i

Adjacency List - Exercise



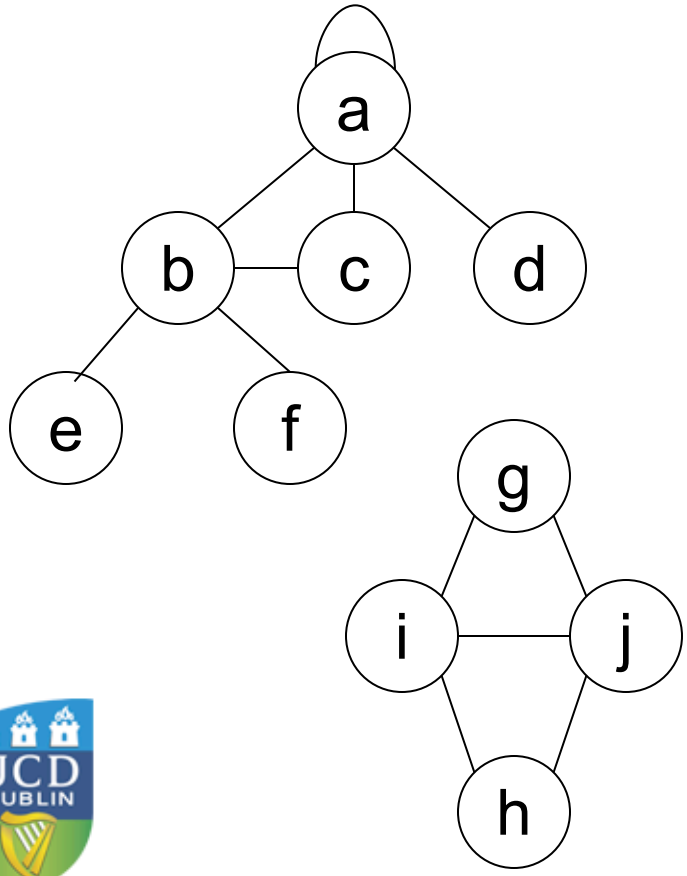
a	
b	
c	
d	
e	
f	
g	
h	
i	
j	

Adjacency List - Exercise



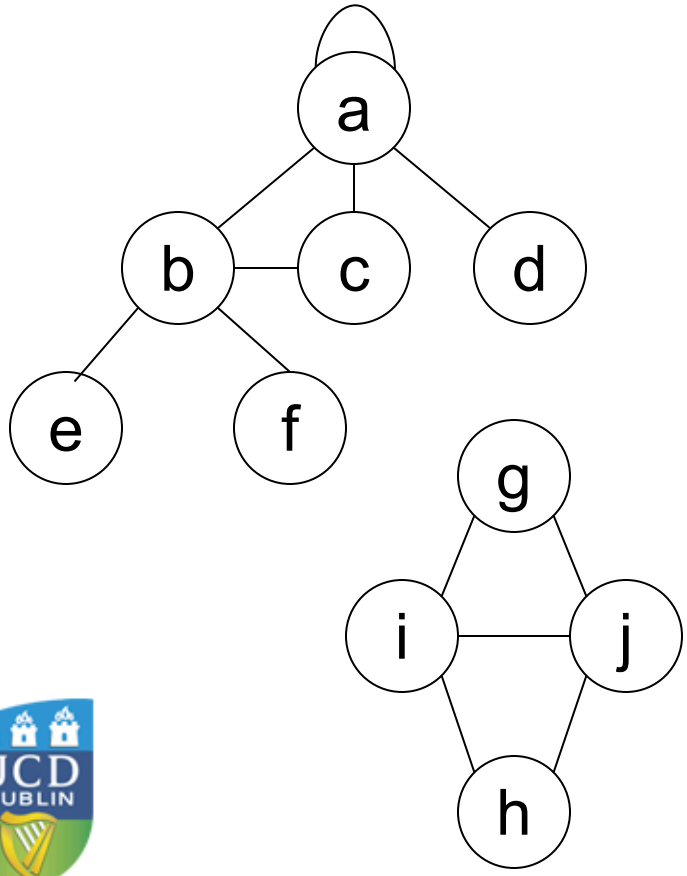
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Adjacency Matrix



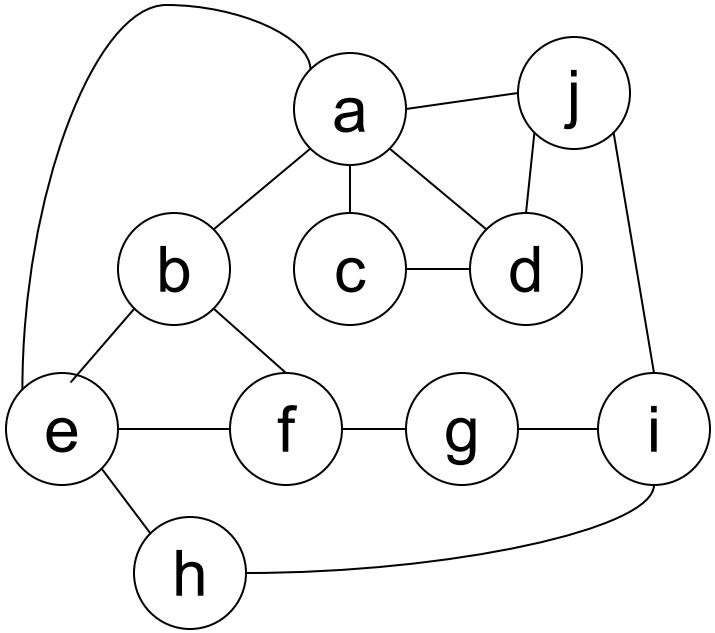
	a	b	c	d	e	f	g	h	i	j
a	1	1	1	1						
b	1		1		1	1				
c	1	1								
d	1									
e		1								
f		1								
g									1	1
h									1	1
i							1	1		1
j							1	1	1	

Adjacency Matrix



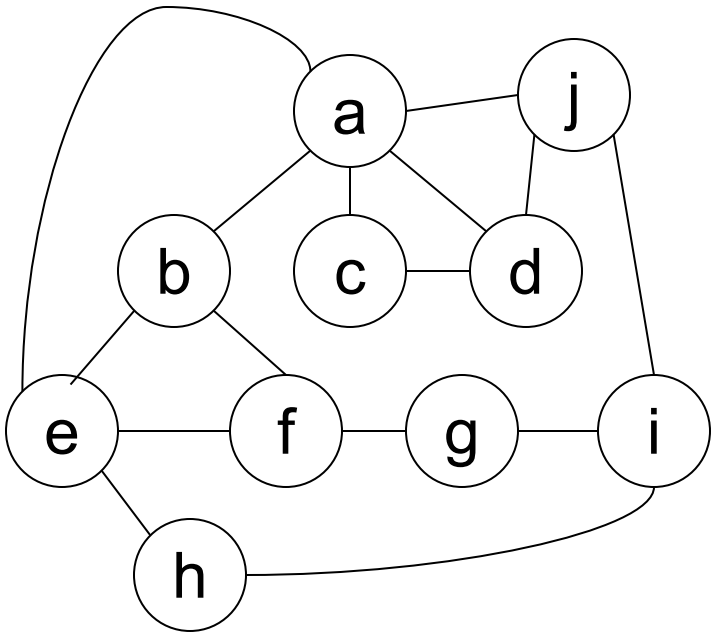
	a	b	c	d	e	f	g	h	i	j
a	1	1	1	1						
b			1		1	1				
c										
d										
e										
f										
g									1	1
h									1	1
i										1
j										

Adjacency Matrix - Exercise



	a	b	c	d	e	f	g	h	i	j
a										
b										
c										
d										
e										
f										
g										
h										
i										
j										

Adjacency Matrix - Exercise



	a	b	c	d	e	f	g	h	i	j
a		1	1	1	1					1
b	1				1	1				
c	1			1						
d	1		1							1
e	1	1				1		1		
f		1			1		1			
g						1			1	
h					1				1	
i							1	1		1
j	1			1					1	

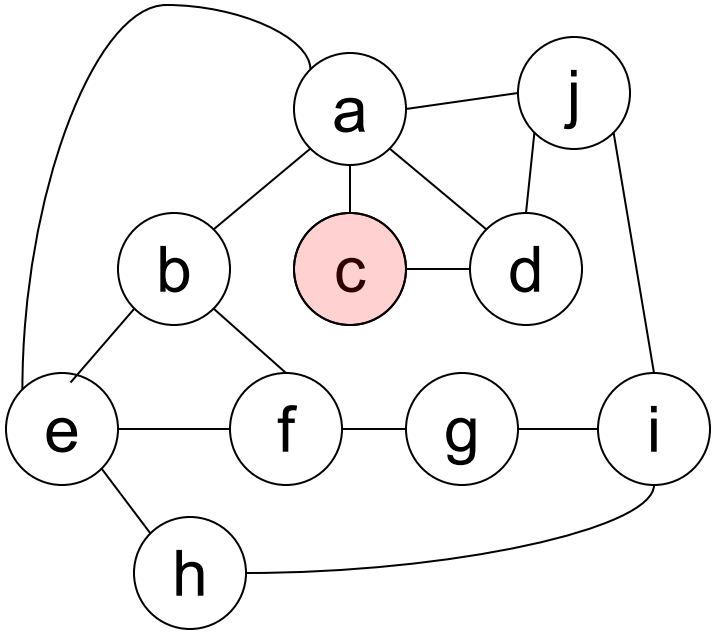
Comparison

- Adjacency Matrix: mathematically easy representation ***but*** waste of space
- Adjacency List: A much more compressed representation (for sparse graphs) ***but*** some graph operations are more difficult compared to the adj. matrix

- Counting inlinks:
  - Matrix: scan the column and count
  - List: difficult, worst case all data needs to be scanned
- Counting outlinks:
  - Matrix: scan the rows and count
  - List: outlinks are natural

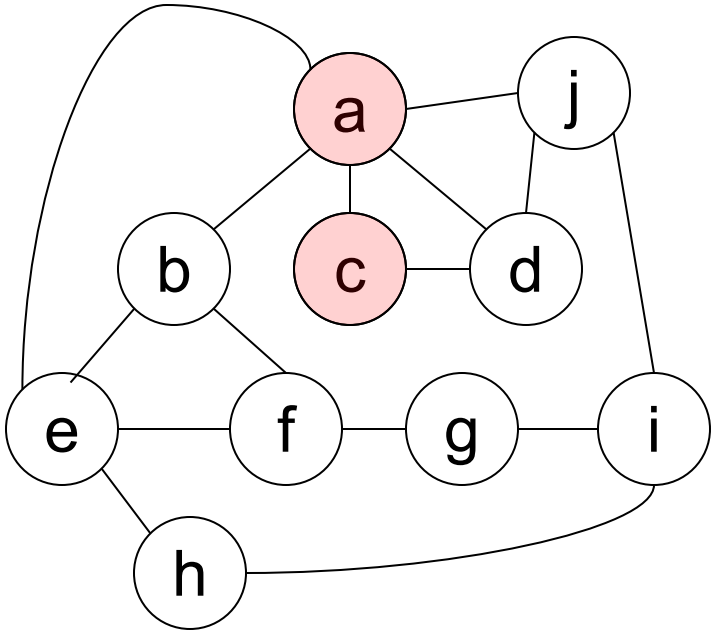
(EXAMPLE OF) GRAPH ALGORITHMS

Depth First Search



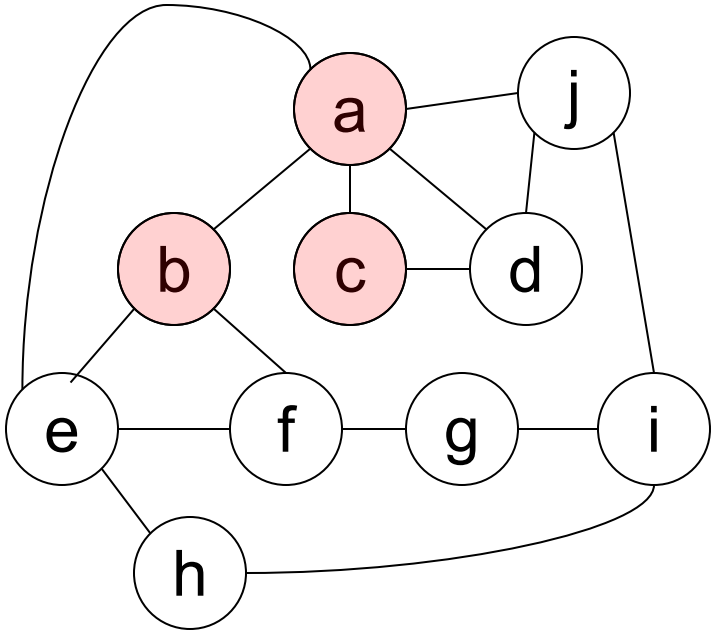
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search



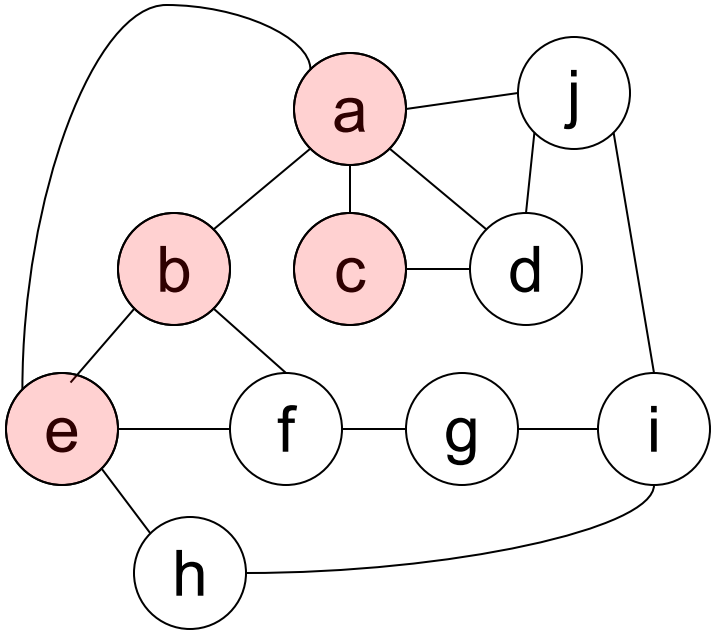
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search



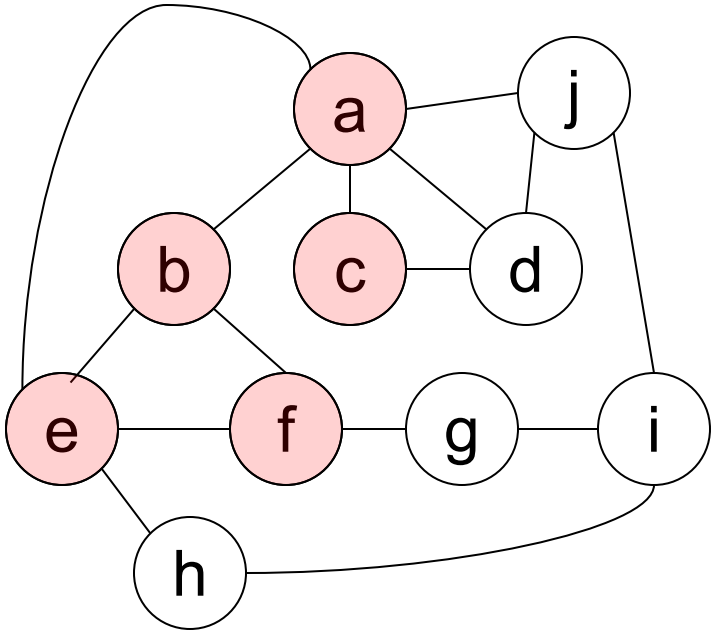
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search



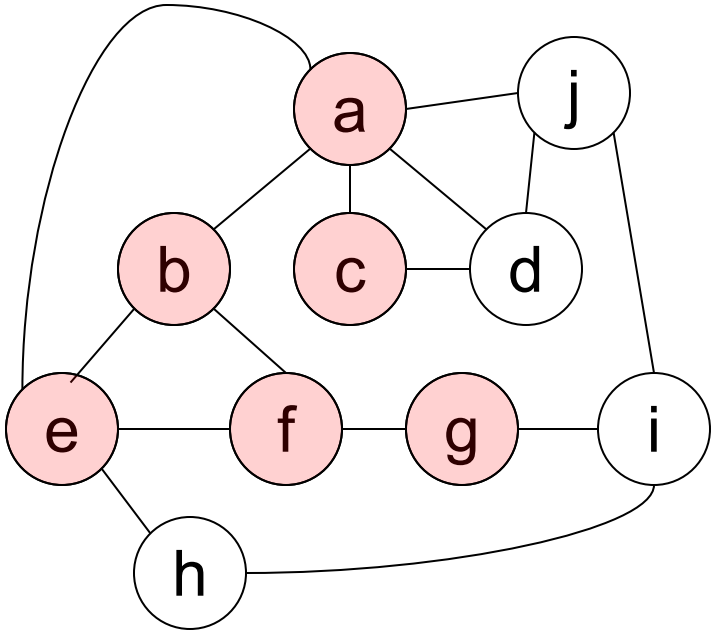
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search



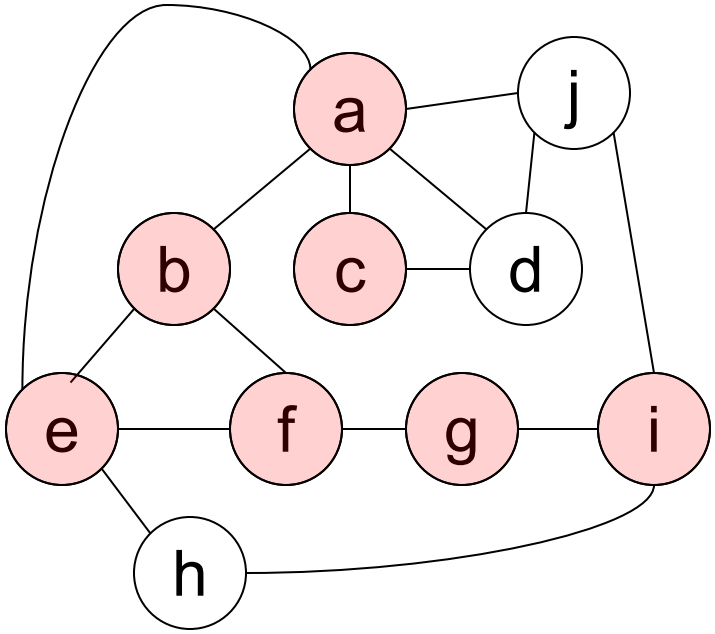
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search



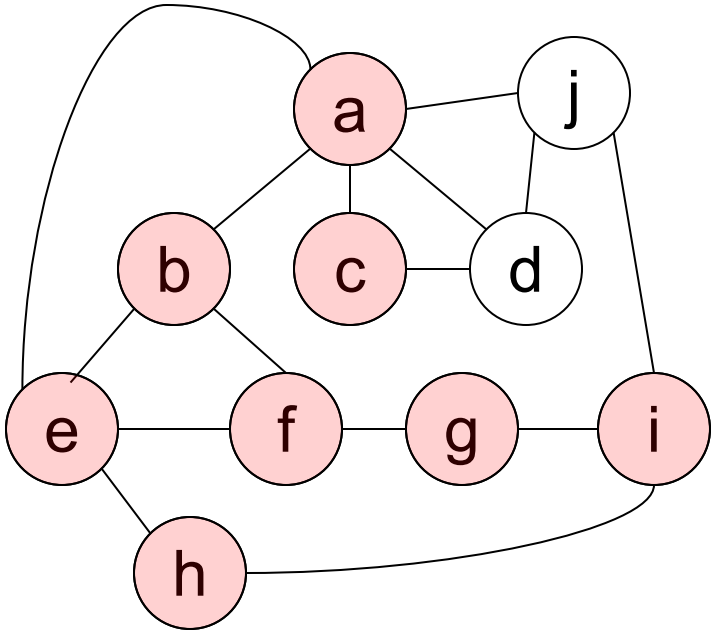
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search



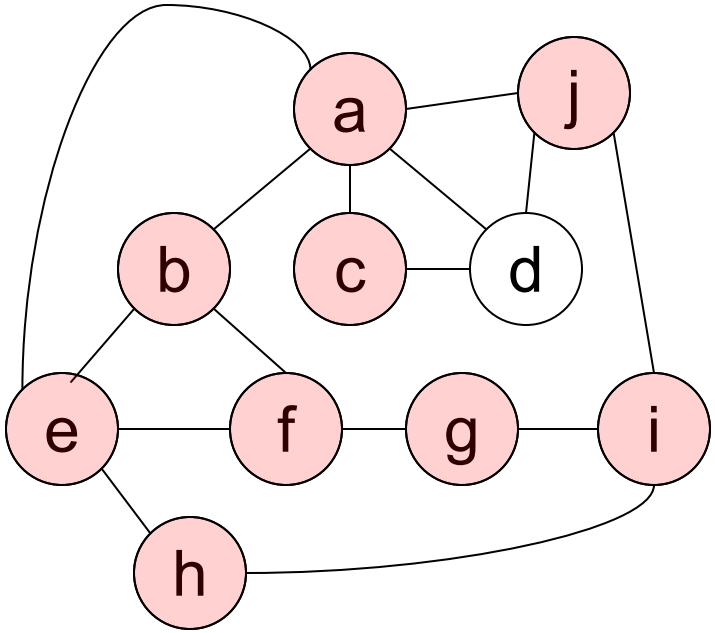
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search



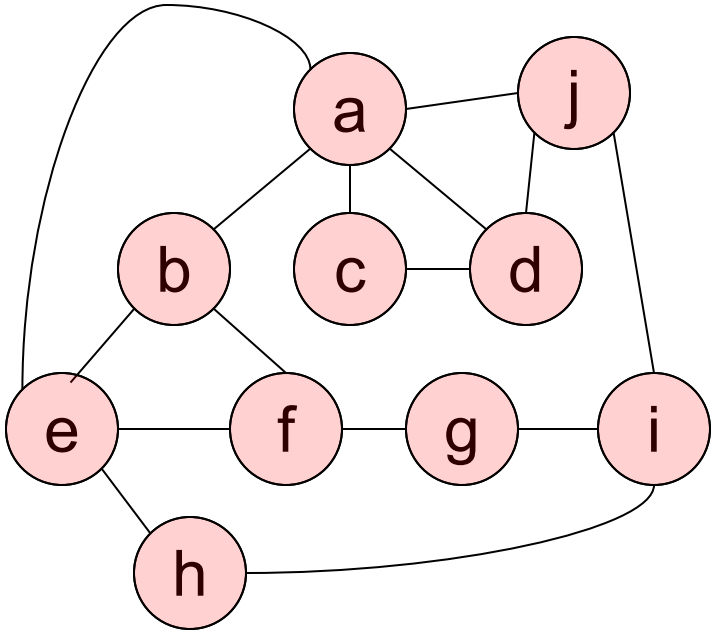
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search



a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Depth First Search



a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

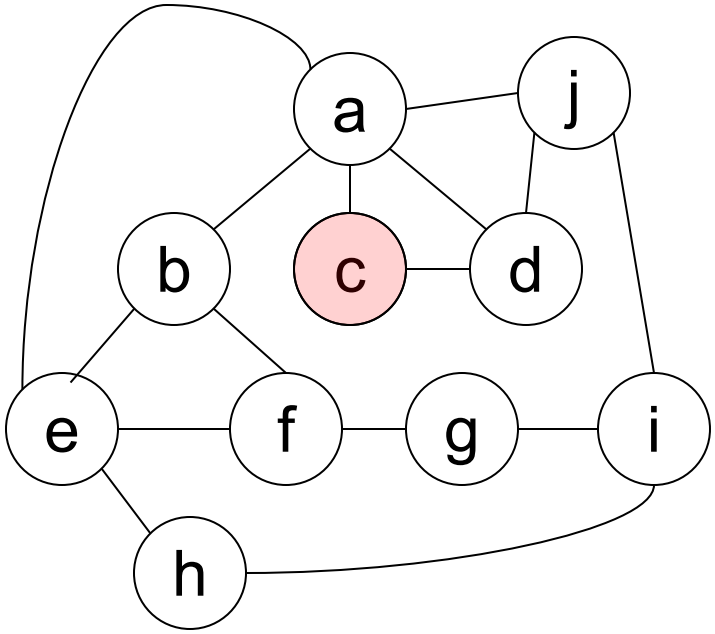
Depth First Search (non-recursive)

```
procedure dfs
Input: a Graph g and a node n
Output: the procedure explores every node starting from
n
to_visit ← empty stack; add n to to_visit
visited ← empty sequence (?)
while to_visit is not empty do
        current ← pop to_visit # get the first element
        add current to visited
        push all neighbours of current (not in visited)
        to to_visit
        do something on current
endfor
```

Depth First Search (recursive)

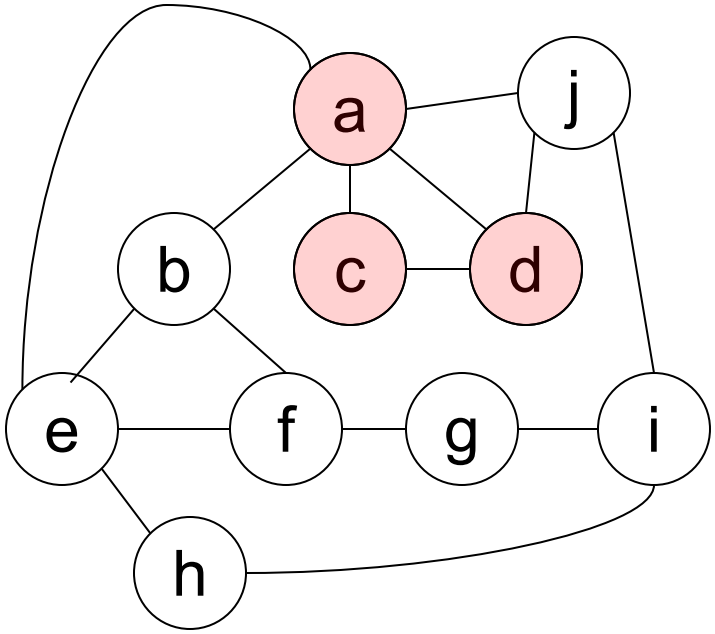
```
function dfs
Input: a Graph g and a node n
Output: the function explores every node from n
flag n as visited
do something
for each neighbour n_c of n which is not visited do
        dfs(n_c)
endfor
```

Breadth First Search



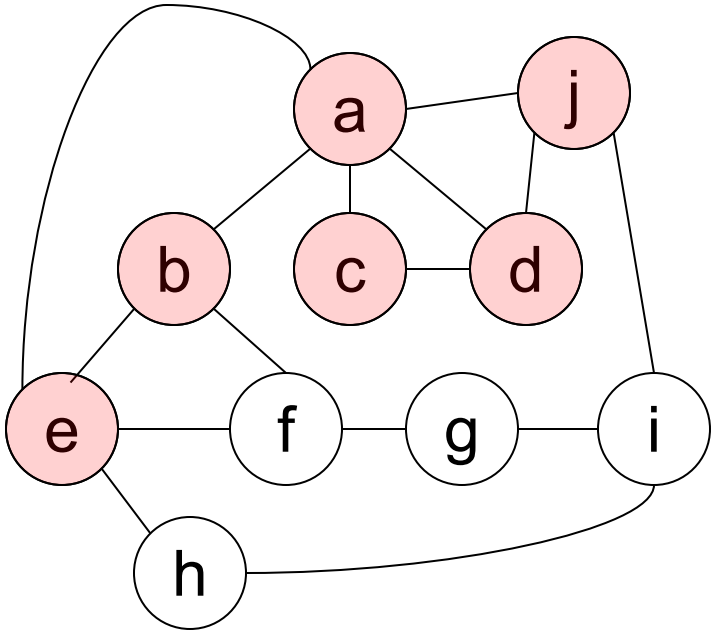
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Breadth First Search



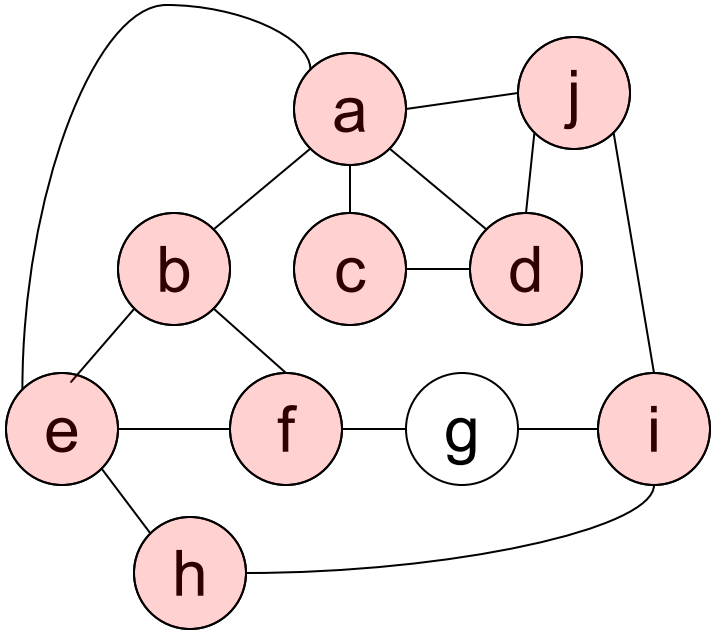
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Breadth First Search



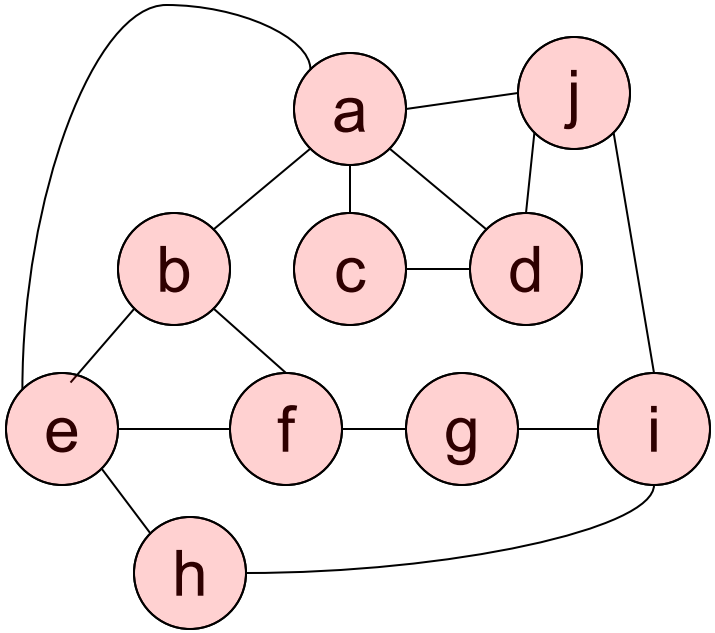
a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Breadth First Search



a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

Breadth First Search



a	b, c, d, e, j
b	a, e, f
c	a, d
d	a, c, j
e	a, b, f, h
f	b, e, g
g	f, i
h	e, i
i	g, h, j
j	a, d, i

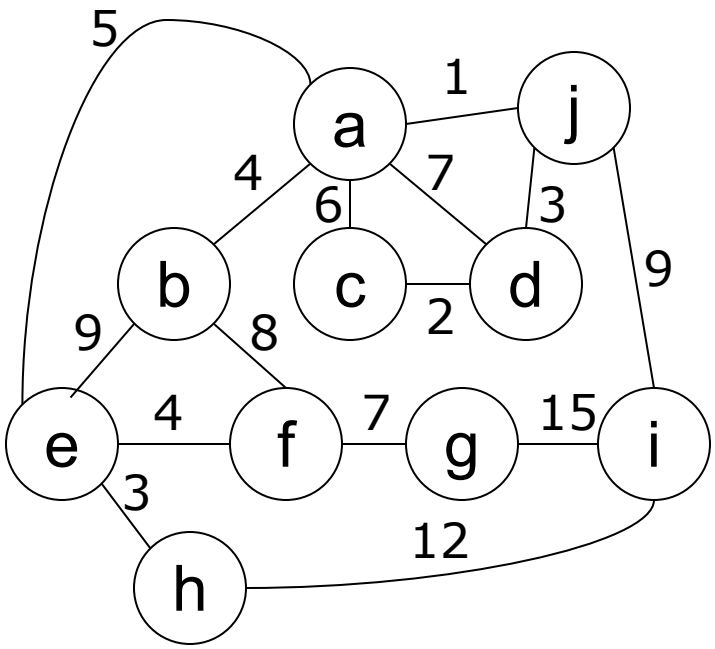
Breadth First Search (non-recursive)

```
procedure bfs
Input: a Graph g and a node n
Output: the procedure explores every node of g from n
to_visit is a queue; enqueue n
visited is a sequence (?)
while to_visit is not empty do
        n_current ← dequeue to_visit
        add n_current to visited
        for each neighbour n_c of n_current that is not
        visited do
                enqueue n_c to to_visit
        endfor
        do something on n_current
endwhile
```

Breadth First Search (recursive)

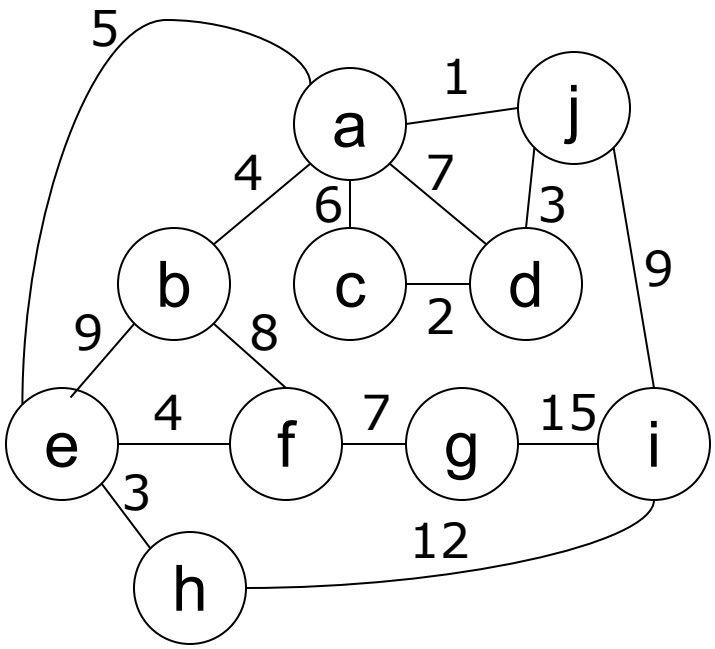
```
function bfs
Input: a Graph g a queue q (originally having a starting node)
Output: explores every node from the starting node
if q is empty then # base case
        do something (?)
else
        current ← dequeue q
        flag current as visited
        for each neighbour n_c of current not visited do
                enqueue n_c
        endfor
        do something
        bfs(q)
endif
```

Adjacency List



a	b(4), c(6), d(7), e(5), j(1)
b	a(4), e(9), f(8)
c	a(6), d(2)
d	a(7), c(2), j(3)
e	a(5), b(9), f(4), h(3)
f	b(8), e(4), g(7)
g	f(7), i(15)
h	e(3), i(12)
i	g(15), h(12), j(9)
j	a(1), d(3), i(9)

Adjacency Matrix

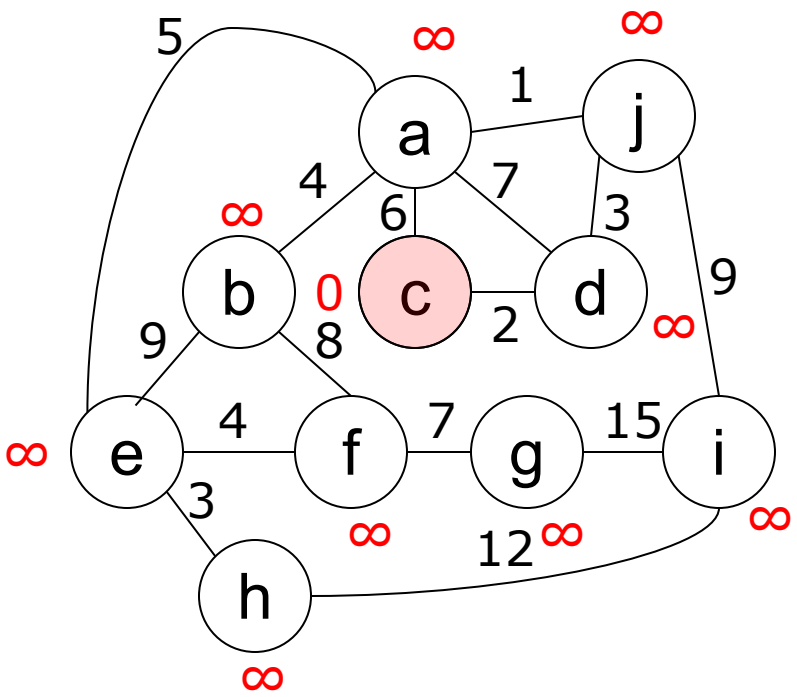


	a	b	c	d	e	f	g	h	i	j
a		4	6	7	5					1
b	4				9	8				
c	6			2						
d	7		2							3
e	5	9				4		3		
f		8			4		7			
g						7			15	
h					3				12	
i							15	12		9
j	1			3					9	

SINGLE-SOURCE SHORTEST PATH

Dijkstra's Algorithm

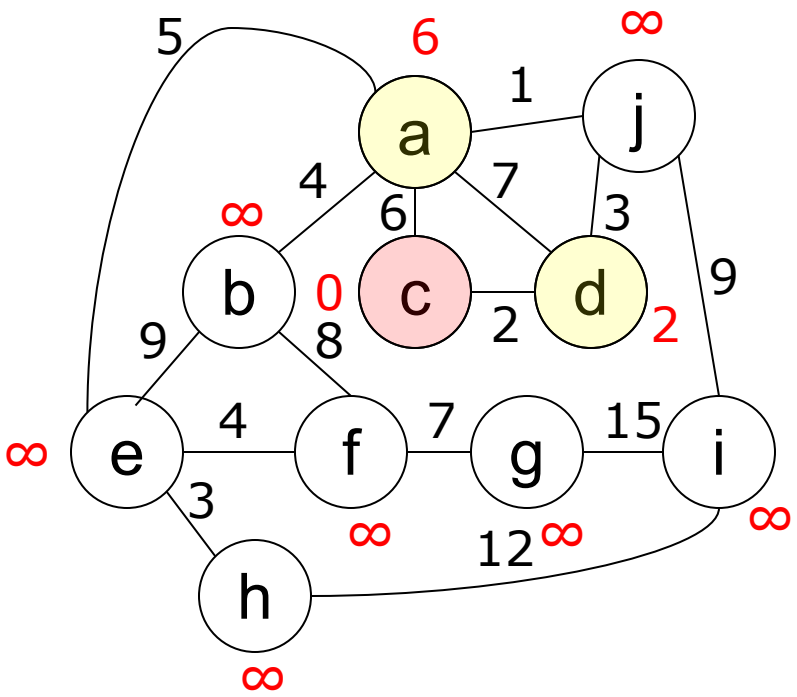
Visited = {c}



a	b(4), c(6), d(7), e(5), j(1)
b	a(4), e(9), f(8)
c	a(6), d(2)
d	a(7), c(2), j(3)
e	a(5), b(9), f(4), h(3)
f	b(8), e(4), g(7)
g	f(7), i(15)
h	e(3), i(12)
i	g(15), h(12), j(9)
j	a(1), d(3), i(9)

Dijkstra's Algorithm

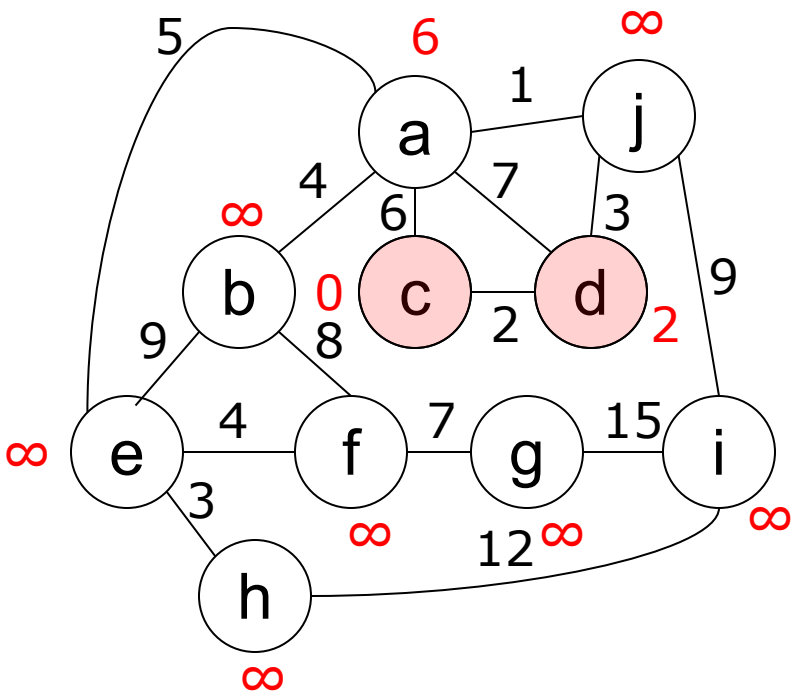
Visited = {c}



a	b(4), c(6), d(7), e(5), j(1)
b	a(4), e(9), f(8)
c	a(6), d(2)
d	a(7), c(2), j(3)
e	a(5), b(9), f(4), h(3)
f	b(8), e(4), g(7)
g	f(7), i(15)
h	e(3), i(12)
i	g(15), h(12), j(9)
j	a(1), d(3), i(9)

Dijkstra's Algorithm

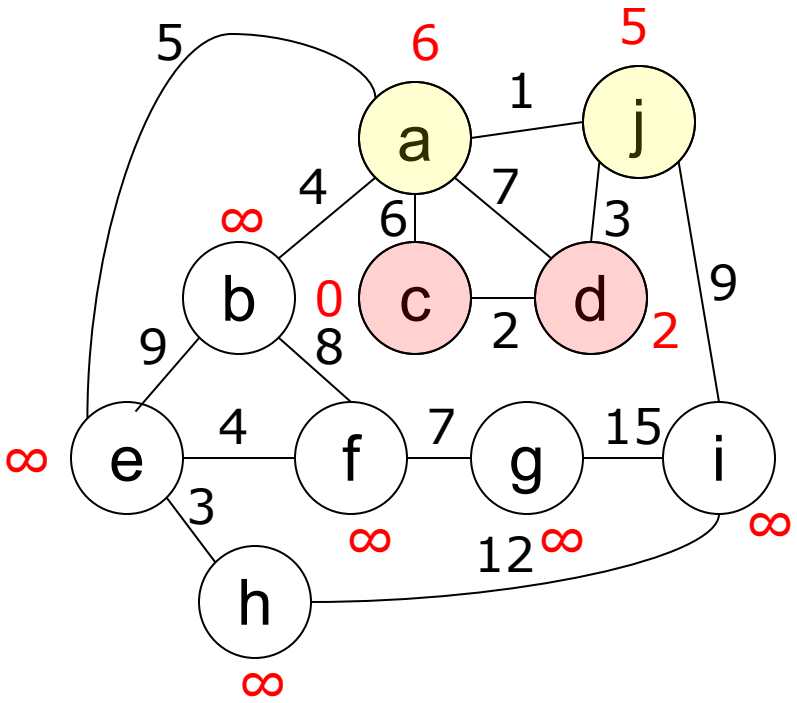
Visited = {c, d}



a	b(4), c(6), d(7), e(5), j(1)
b	a(4), e(9), f(8)
c	a(6), d(2)
d	a(7), c(2), j(3)
e	a(5), b(9), f(4), h(3)
f	b(8), e(4), g(7)
g	f(7), i(15)
h	e(3), i(12)
i	g(15), h(12), j(9)
j	a(1), d(3), i(9)

Dijkstra's Algorithm

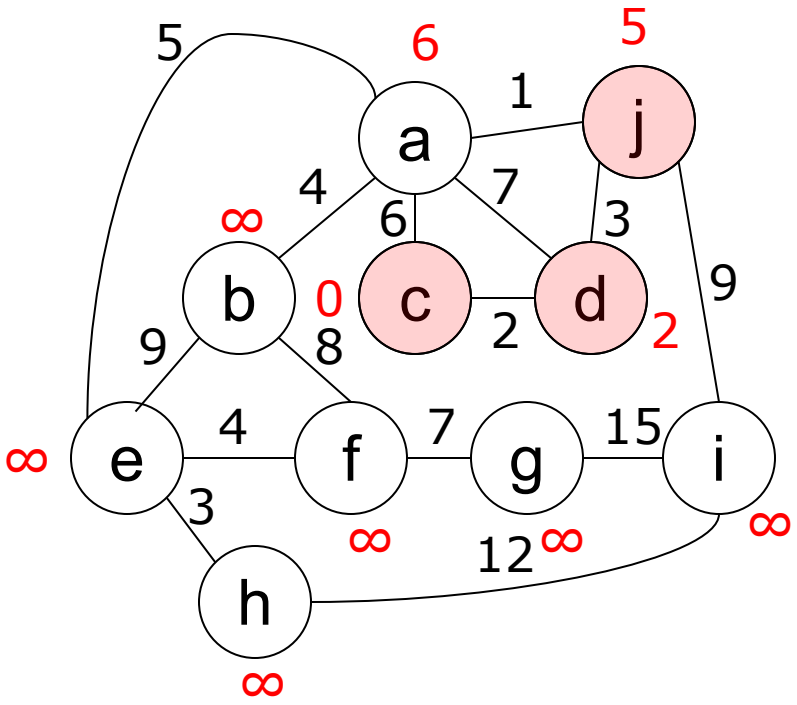
Visited = {c, d}



a	b(4), c(6), d(7), e(5), j(1)
b	a(4), e(9), f(8)
c	a(6), d(2)
d	a(7), c(2), j(3)
e	a(5), b(9), f(4), h(3)
f	b(8), e(4), g(7)
g	f(7), i(15)
h	e(3), i(12)
i	g(15), h(12), j(9)
j	a(1), d(3), i(9)

Dijkstra's Algorithm

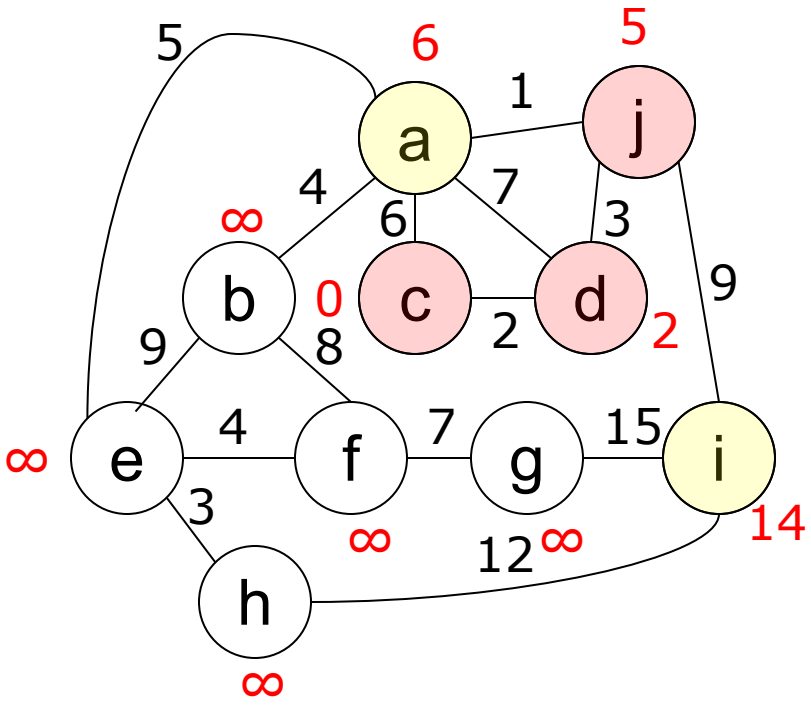
Visited = {c, d, j}



a	b(4), c(6), d(7), e(5), j(1)
b	a(4), e(9), f(8)
c	a(6), d(2)
d	a(7), c(2), j(3)
e	a(5), b(9), f(4), h(3)
f	b(8), e(4), g(7)
g	f(7), i(15)
h	e(3), i(12)
i	g(15), h(12), j(9)
j	a(1), d(3), i(9)

Dijkstra's Algorithm

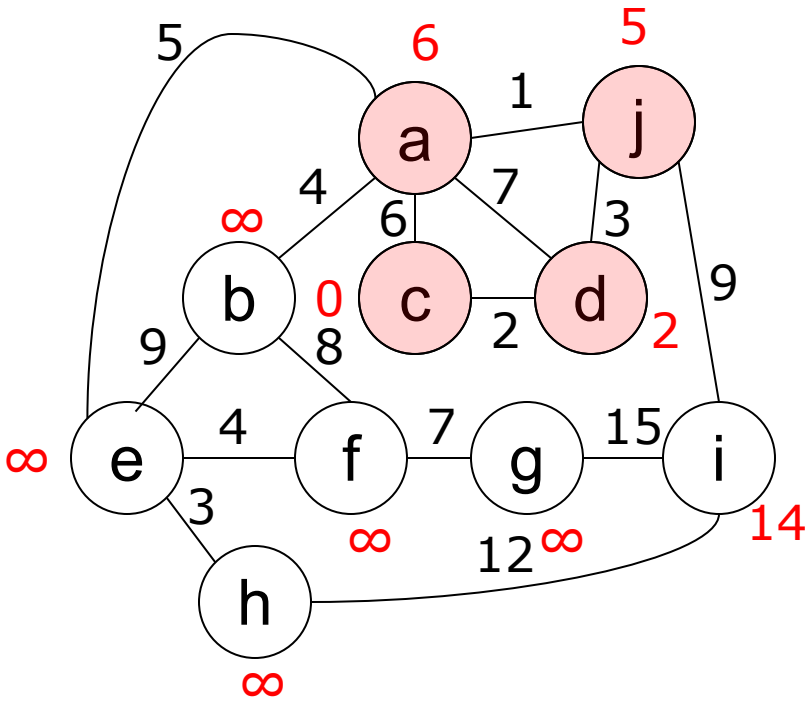
Visited = {c, d, j}



a	b(4), c(6), d(7), e(5), j(1)
b	a(4), e(9), f(8)
c	a(6), d(2)
d	a(7), c(2), j(3)
e	a(5), b(9), f(4), h(3)
f	b(8), e(4), g(7)
g	f(7), i(15)
h	e(3), i(12)
i	g(15), h(12), j(9)
j	a(1), d(3), i(9)

Dijkstra's Algorithm

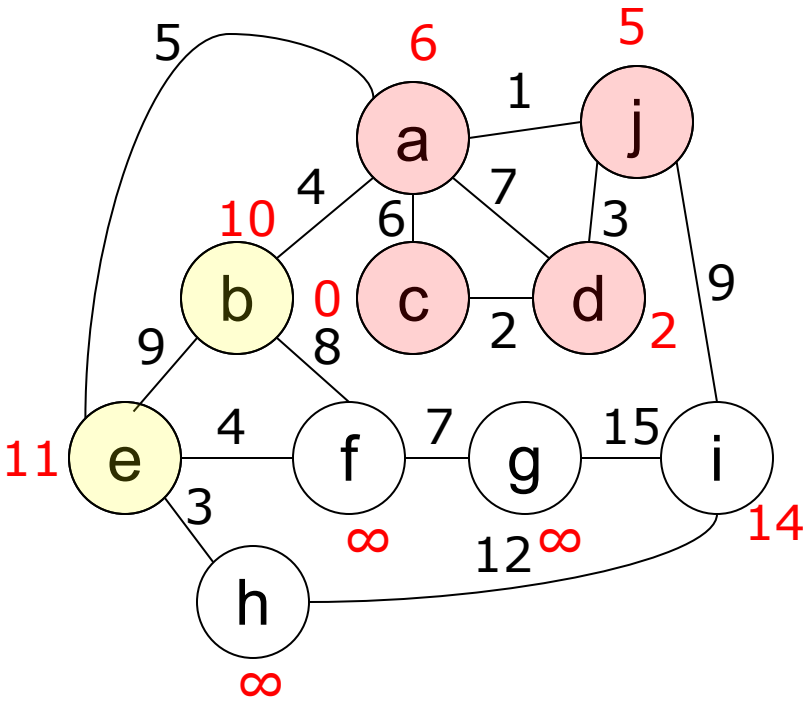
Visited = {c, d, j, a}



a	b(4), c(6), d(7), e(5), j(1)
b	a(4), e(9), f(8)
c	a(6), d(2)
d	a(7), c(2), j(3)
e	a(5), b(9), f(4), h(3)
f	b(8), e(4), g(7)
g	f(7), i(15)
h	e(3), i(12)
i	g(15), h(12), j(9)
j	a(1), d(3), i(9)

Dijkstra's Algorithm

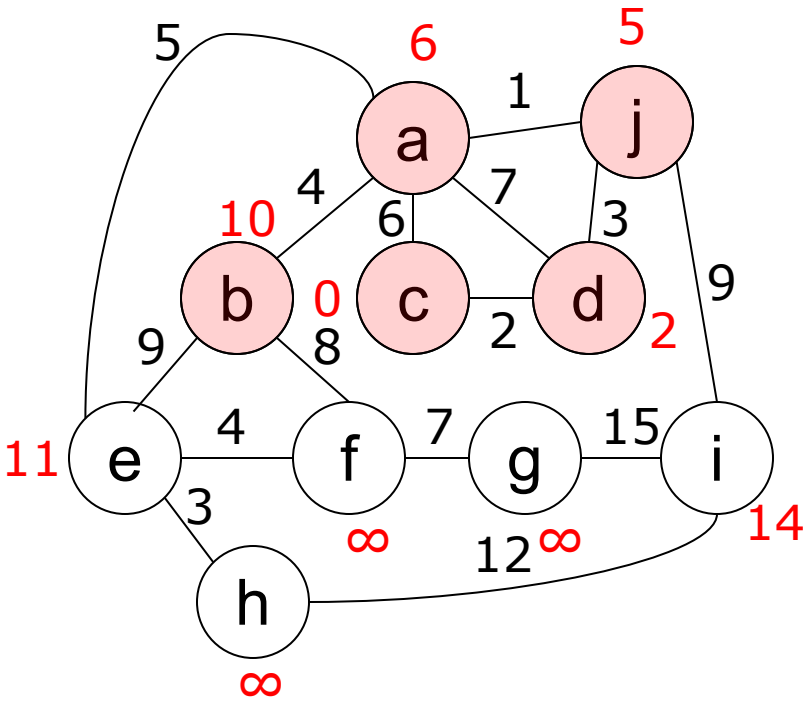
Visited = {c, d, j, a}



a	b(4), c(6), d(7), e(5), j(1)
b	a(4), e(9), f(8)
c	a(6), d(2)
d	a(7), c(2), j(3)
e	a(5), b(9), f(4), h(3)
f	b(8), e(4), g(7)
g	f(7), i(15)
h	e(3), i(12)
i	g(15), h(12), j(9)
j	a(1), d(3), i(9)

Dijkstra's Algorithm

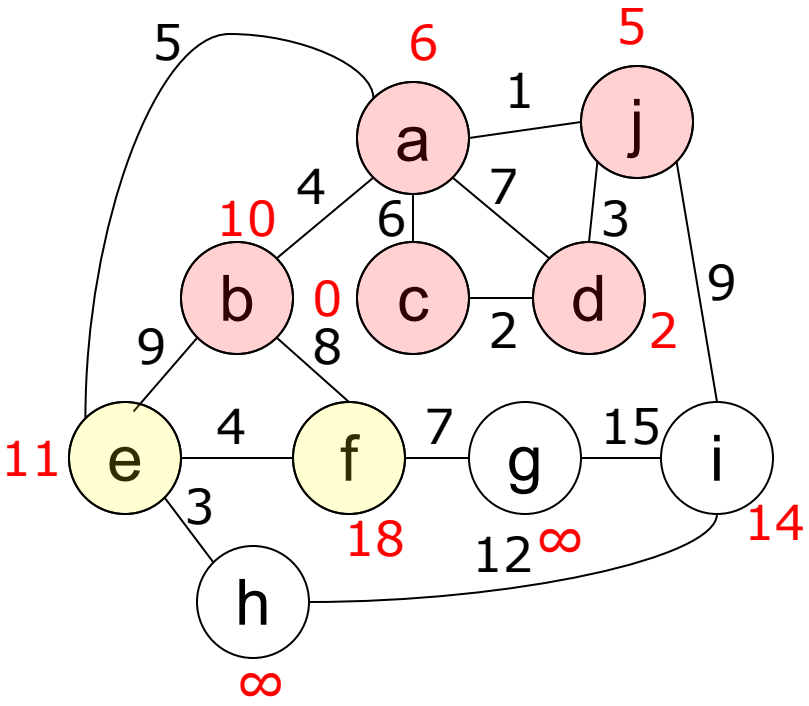
Visited = {c, d, j, a, b}



a	b(4), c(6), d(7), e(5), j(1)
b	a(4), e(9), f(8)
c	a(6), d(2)
d	a(7), c(2), j(3)
e	a(5), b(9), f(4), h(3)
f	b(8), e(4), g(7)
g	f(7), i(15)
h	e(3), i(12)
i	g(15), h(12), j(9)
j	a(1), d(3), i(9)

Dijkstra's Algorithm

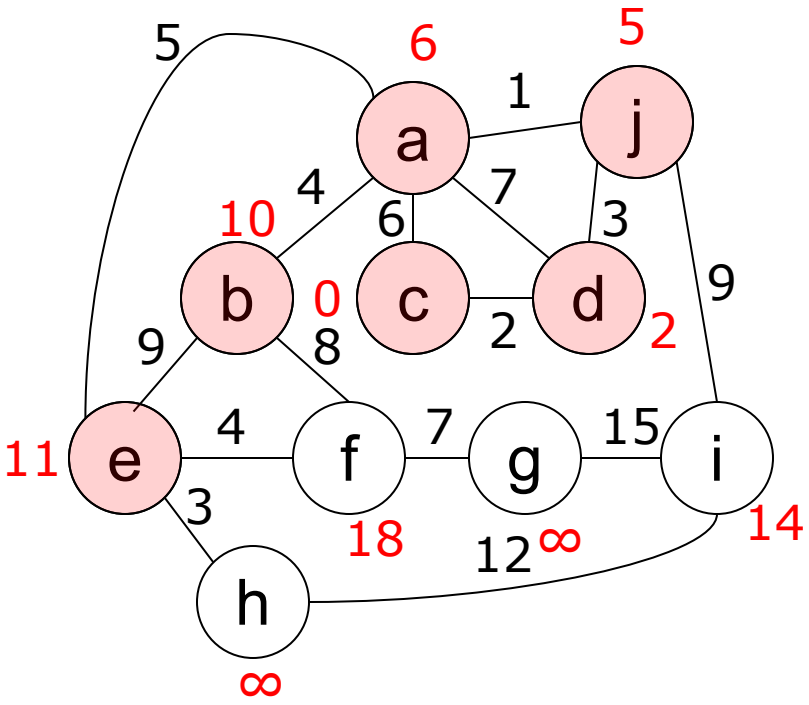
Visited = {c, d, j, a, b}



a	b(4), c(6), d(7), e(5), j(1)
b	a(4), e(9), f(8)
c	a(6), d(2)
d	a(7), c(2), j(3)
e	a(5), b(9), f(4), h(3)
f	b(8), e(4), g(7)
g	f(7), i(15)
h	e(3), i(12)
i	g(15), h(12), j(9)
j	a(1), d(3), i(9)

Dijkstra's Algorithm

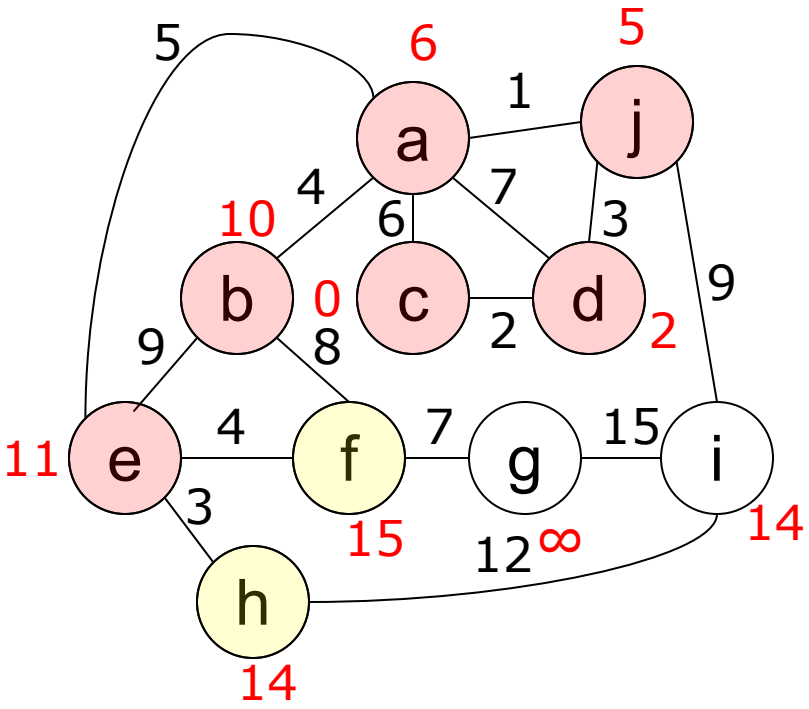
Visited = {c, d, j, a, b, e}



a	b(4), c(6), d(7), e(5), j(1)
b	a(4), e(9), f(8)
c	a(6), d(2)
d	a(7), c(2), j(3)
e	a(5), b(9), f(4), h(3)
f	b(8), e(4), g(7)
g	f(7), i(15)
h	e(3), i(12)
i	g(15), h(12), j(9)
j	a(1), d(3), i(9)

Dijkstra's Algorithm

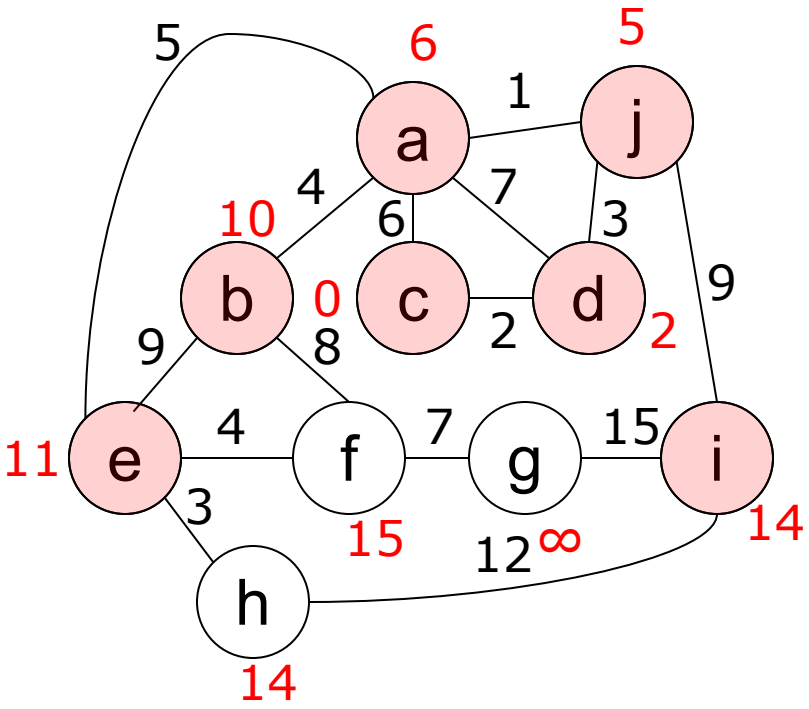
Visited = {c, d, j, a, b, e}



a	b(4), c(6), d(7), e(5), j(1)
b	a(4), e(9), f(8)
c	a(6), d(2)
d	a(7), c(2), j(3)
e	a(5), b(9), f(4), h(3)
f	b(8), e(4), g(7)
g	f(7), i(15)
h	e(3), i(12)
i	g(15), h(12), j(9)
j	a(1), d(3), i(9)

Dijkstra's Algorithm

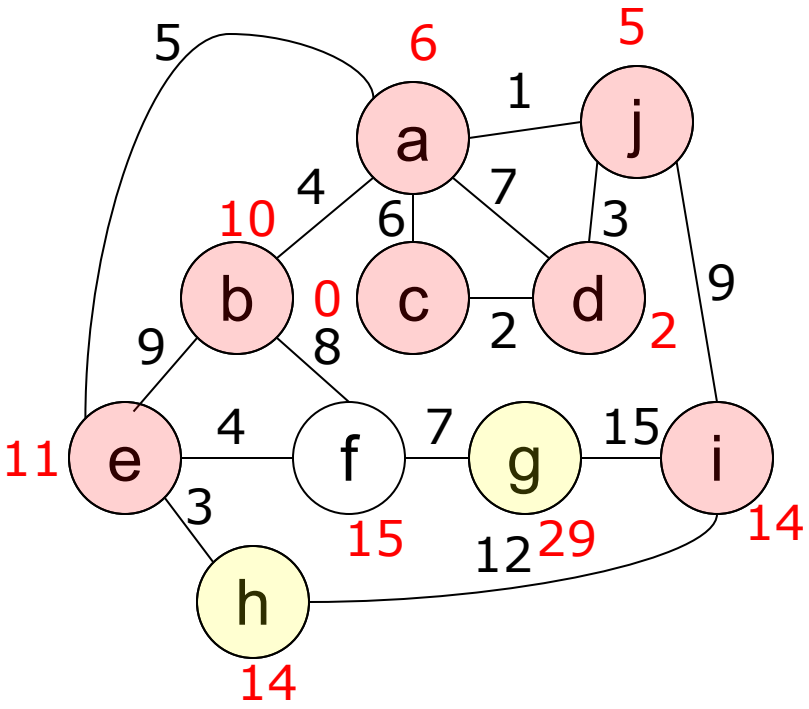
Visited = {c, d, j, a, b, e, i}



a	b(4), c(6), d(7), e(5), j(1)
b	a(4), e(9), f(8)
c	a(6), d(2)
d	a(7), c(2), j(3)
e	a(5), b(9), f(4), h(3)
f	b(8), e(4), g(7)
g	f(7), i(15)
h	e(3), i(12)
i	g(15), h(12), j(9)
j	a(1), d(3), i(9)

Dijkstra's Algorithm

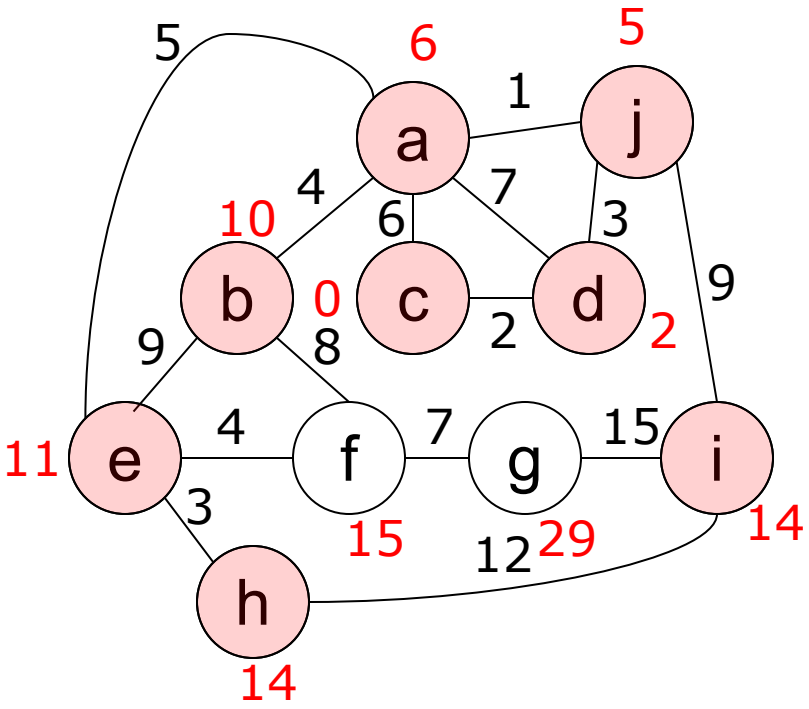
Visited = {c, d, j, a, b, e, i}



a	b(4), c(6), d(7), e(5), j(1)
b	a(4), e(9), f(8)
c	a(6), d(2)
d	a(7), c(2), j(3)
e	a(5), b(9), f(4), h(3)
f	b(8), e(4), g(7)
g	f(7), i(15)
h	e(3), i(12)
i	g(15), h(12), j(9)
j	a(1), d(3), i(9)

Dijkstra's Algorithm

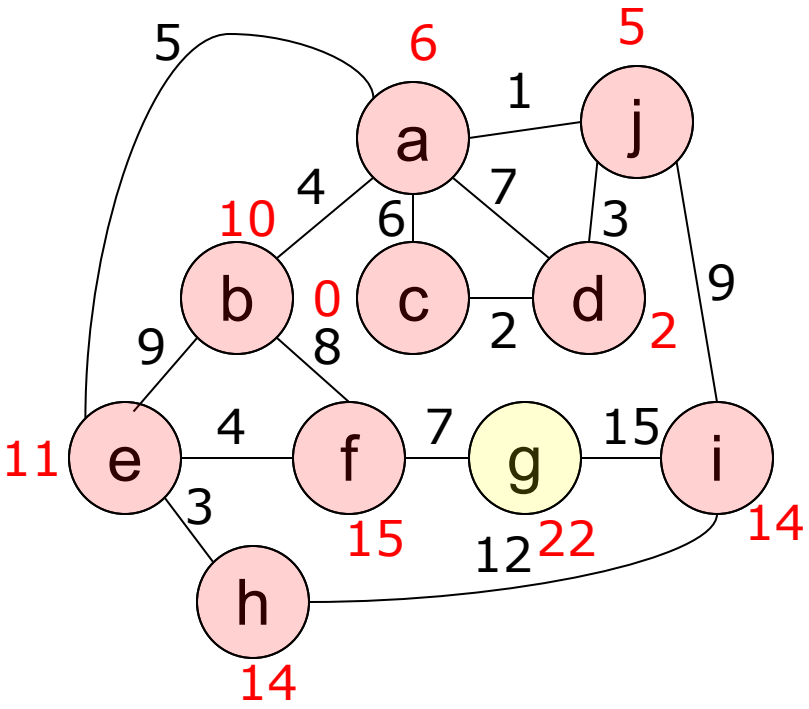
Visited = {c, d, j, a, b, e, i, h}



a	b(4), c(6), d(7), e(5), j(1)
b	a(4), e(9), f(8)
c	a(6), d(2)
d	a(7), c(2), j(3)
e	a(5), b(9), f(4), h(3)
f	b(8), e(4), g(7)
g	f(7), i(15)
h	e(3), i(12)
i	g(15), h(12), j(9)
j	a(1), d(3), i(9)

Dijkstra's Algorithm

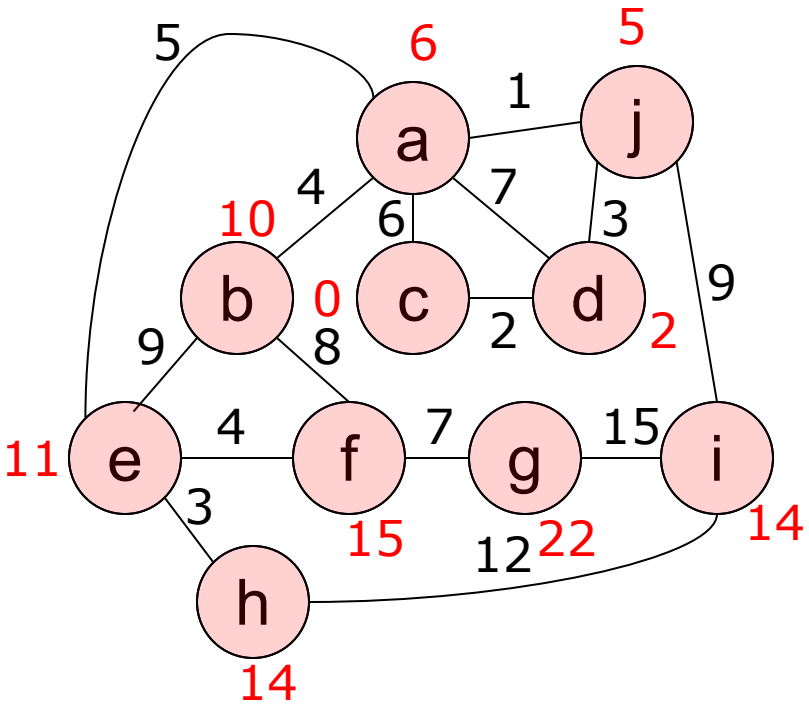
Visited = {c, d, j, a, b, e, i, h, f}



a	b(4), c(6), d(7), e(5), j(1)
b	a(4), e(9), f(8)
c	a(6), d(2)
d	a(7), c(2), j(3)
e	a(5), b(9), f(4), h(3)
f	b(8), e(4), g(7)
g	f(7), i(15)
h	e(3), i(12)
i	g(15), h(12), j(9)
j	a(1), d(3), i(9)

Dijkstra's Algorithm

Visited = {c, d, j, a, b, e, i, h, f, g}



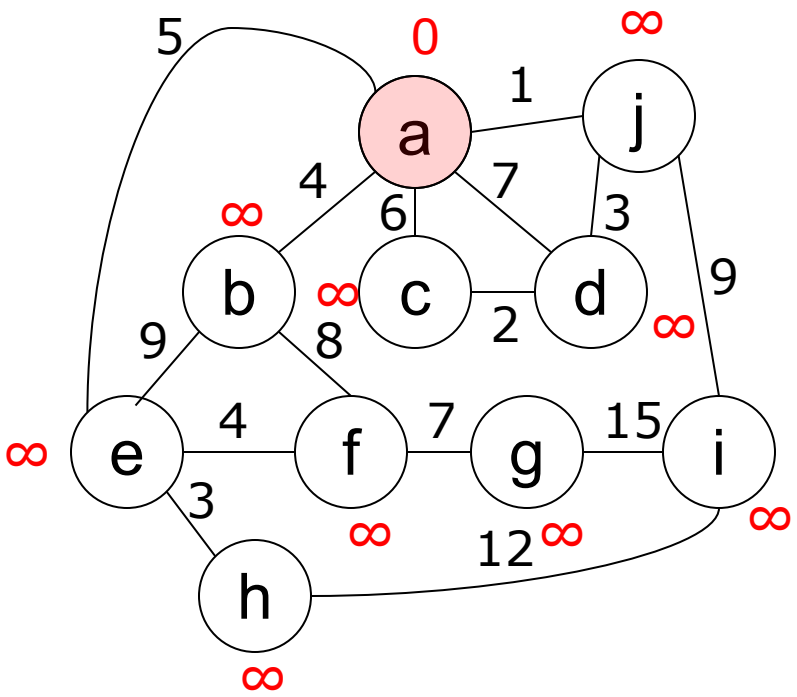
a	b(4), c(6), d(7), e(5), j(1)
b	a(4), e(9), f(8)
c	a(6), d(2)
d	a(7), c(2), j(3)
e	a(5), b(9), f(4), h(3)
f	b(8), e(4), g(7)
g	f(7), i(15)
h	e(3), i(12)
i	g(15), h(12), j(9)
j	a(1), d(3), i(9)

Dijkstra's Algorithm

```
function dijkstra
Input: a graph G, a node n
Output: each node of G gets the shortest distance from n
for each node c of G not n do
        distance[c] <- infinity
endfor
distance[n] <- 0
current <- n
visited <- {n}
while visited does not contain all nodes of G do
        for each node c neighbour of current and not visited do
                distance[c] <- min(distance[n] + weight(current, c),
distance[c])
        endfor
        current <- non visited node with smallest value
        add current to visited
endwhile
```

Dijkstra's Algorithm - Exercise

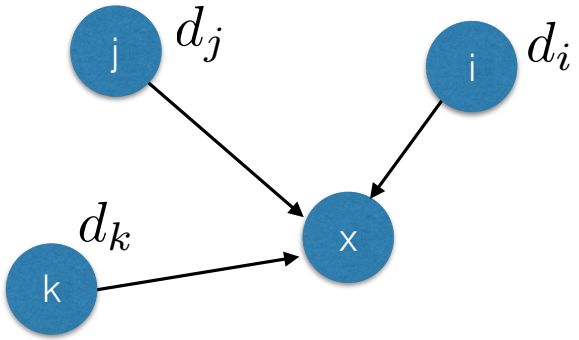
Visited = {a}



a	b(4), c(6), d(7), e(5), j(1)
b	a(4), e(9), f(8)
c	a(6), d(2)
d	a(7), c(2), j(3)
e	a(5), b(9), f(4), h(3)
f	b(8), e(4), g(7)
g	f(7), i(15)
h	e(3), i(12)
i	g(15), h(12), j(9)
j	a(1), d(3), i(9)

Shortest path in the MapReduce World

- **Task**: find the shortest path from a source node to all other nodes in the graph. Edges have unit weight.
- **Intuition**:
  - Distance of nodes *N* directly connected to the source is 1
  - Distance of nodes directly connected to nodes in N is 2
  - Multiple path to *x*: the shortest path must go through one of the nodes with an outlink to *x*; use the minimum



$$d_x = min(d_i + 1, d_j + 1, d_k + 1)$$

Shortest path in the MapReduce World

```
1: class Mapper
2:     method Map(nid n, node N)
3:         d ← N.Distance
4:         Emit(nid n, N)
5:         for all nodeid m ∈ N.AdjacencyList do
6:             Emit(nid m, d + 1)

1: class Reducer
2:     method Reduce(nid m, [d1, d2, ...])
3:         d_min ← ∞
4:         M ← ∅
5:         for all d ∈ counts [d1, d2, ...] do
6:             if IsNode(d) then
7:                 M ← d
8:             else if d < d_min then
9:                 d_min ← d
10:        M.Distance ← d_min
11:        Emit(nid m, node M)
```

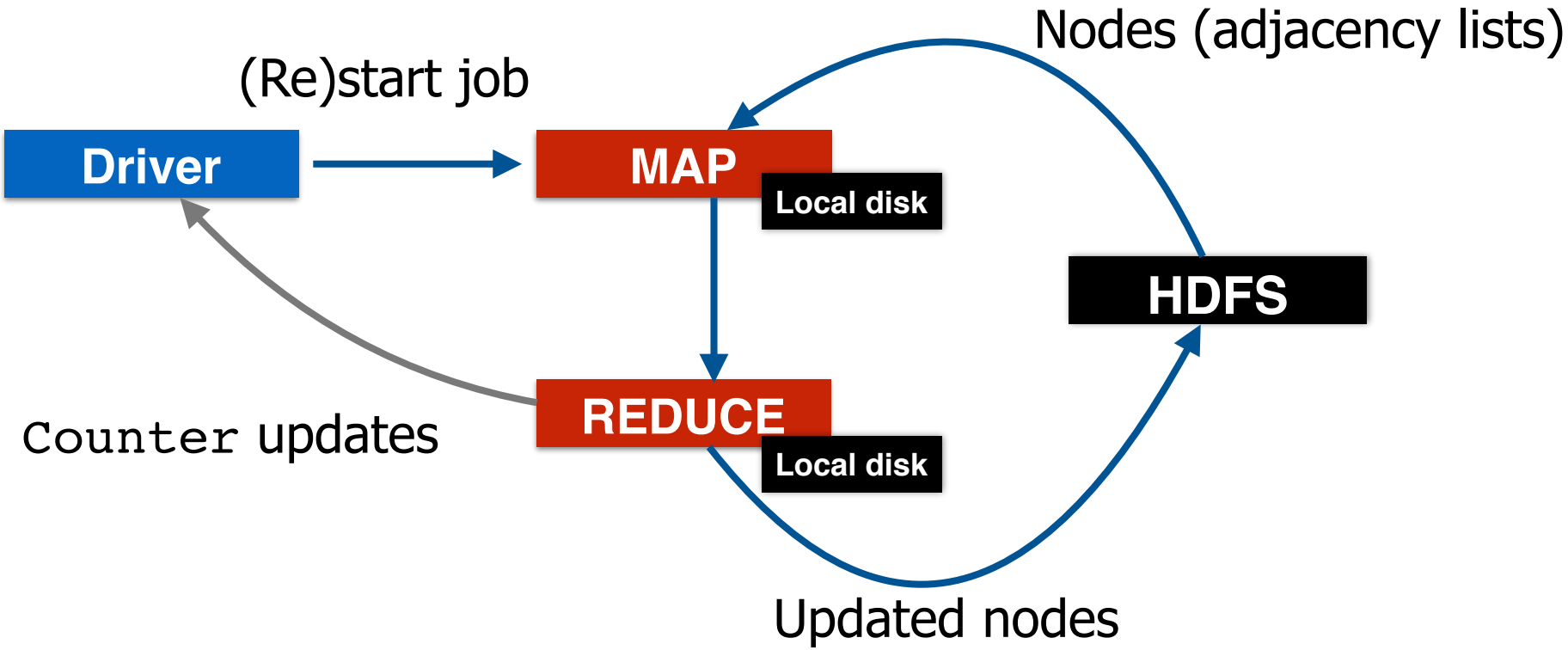
Shortest path in the MapReduce World

- Each ***iteration*** of the algorithm is ***one Hadoop job***
  - A ***map*** phase to compute the distances
  - A ***reduce*** phase to find the current minimum distance
- Iterations:
  1. All nodes connected to the source are discovered
  2. All nodes connected to those discovered in 1. are found
  3. 3. ...
- Between iterations (jobs) the ***graph structure needs to be passed along;*** reducer output is input for the next iteration

Shortest path in the MapReduce World

- How many iterations are necessary to compute the shortest path to all nodes?
  - Diameter of the graph (greatest distance between a pair of nodes)
  - Diameter is usually small ("six degrees of separation")
- In **practice**: iterate until all node distances are less than +infinity
  - Assumption: connected graph
- **Termination condition** checked "outside" of MapReduce job
  - Use Counter to count number of nodes with infinite distance

Shortest path in the MapReduce World



Shortest path in the MapReduce World

- What if the edges have weights?

- Two changes required wrt. the parallel BFS
  - **Update rule**, instead of *d+1* use *d+w*
  - **Termination criterion**: no more distance
- Num. iterations in the worst case: *#nodes-1* changes (via Counter)

Single-source Shortest Path

- Dijkstra
  - Single processor (global data structure)
  - Efficient (no recompilation of finalised states)

- Parallel BFS
  - Brute Force approach
  - A lot of unnecessary computations (distances to all nodes recomputed at each iteration)
  - no global data structure