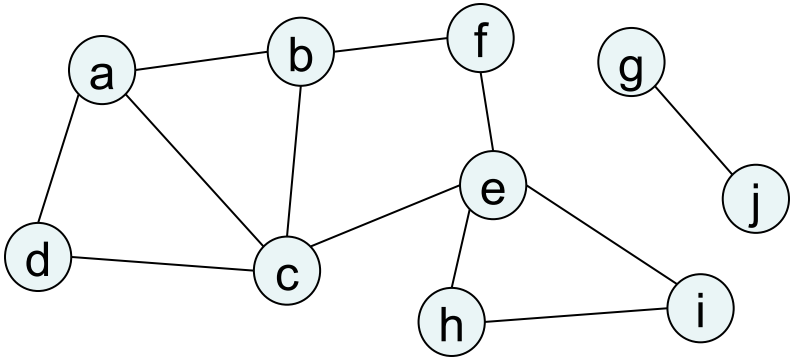
COL702:Advanced Data Structures and Algorithms

Thanks to Miles Jones, Russell Impagliazzo, and Sanjoy Dasgupta at UCSD for these slides.

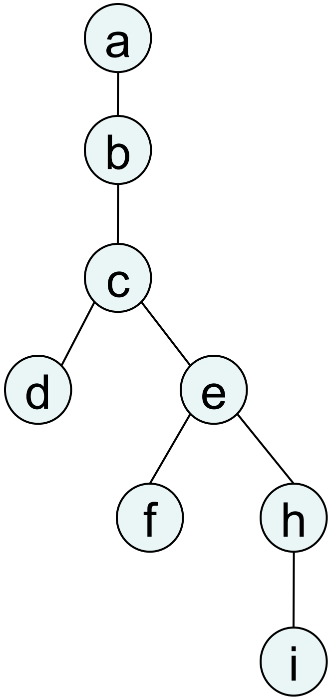
Paths in graphs

The classic 15-puzzle

Graph G = (V,E)
V = {configurations of puzzle}
E: edges between neighboring configurations



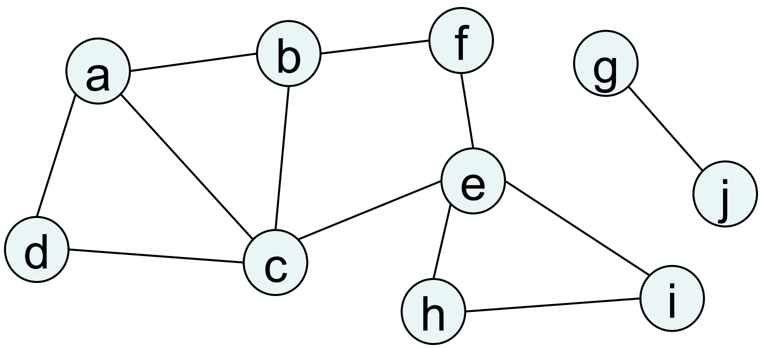
explore(G,a):



Finds a path from a to i.
But this isn't the shortest possible path!

Distances in graphs

Distance between two nodes
= length of shortest path between them

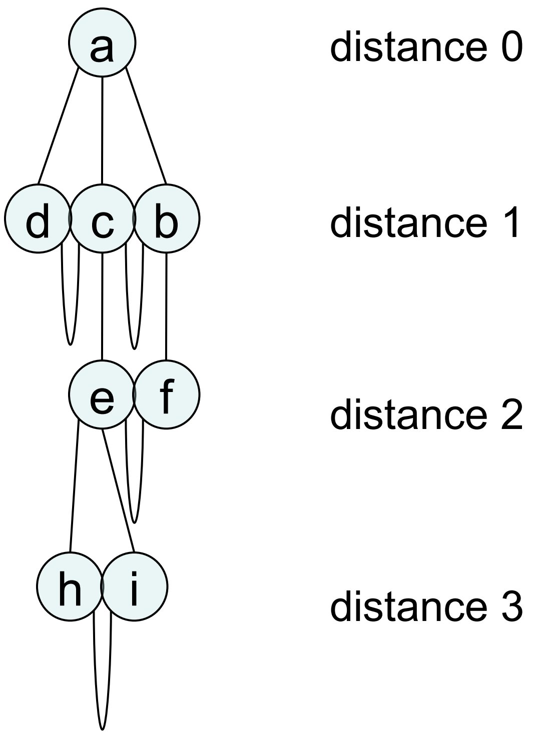


dist(a,e) = ?
dist(d,g) = ?

Suppose we want to compute distances from some starting node s to all other nodes in G.

Strategy: layer-by-layer
- first, nodes at distance 0
- then, nodes at distance 1
- then, nodes at distance 2, etc.

Physical model:
Vertex – ping-pong ball
Edge – piece of string



Breadth-first search

Suppose we have seen all nodes at distance $\leq$ d.

How to get the next layer?

Solution:

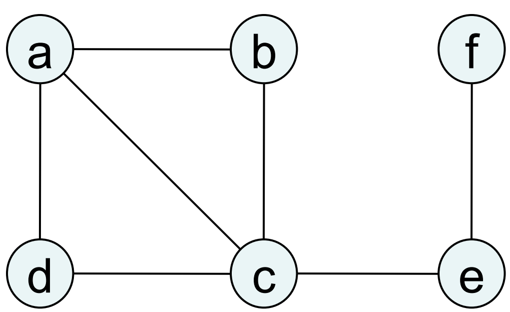
A node is at distance d+1 if:

- it is adjacent to some node at distance d
- it hasn't been seen yet

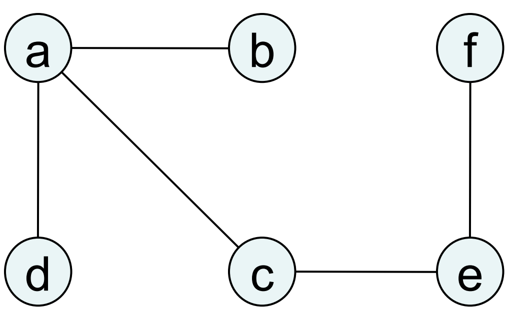
```
procedure bfs(G,s)

input: graph G = (V,E); node s in V
output: for each node u, dist[u] is
   set to its distance from s

for u in V:
   dist[u] = ∞
dist[s] = 0
Q = [s]  // queue containing just s

while Q is not empty:
   u = eject(Q)
   for each edge (u,v) in E:
        if dist[v] = ∞:
              inject(Q,v)
              dist[v] = dist[u]+1
```

BFS example

```
procedure bfs(G,s)

for u in V:
   dist[u] = ∞
   prev[u] = nil
dist[s] = 0
Q = [s]  // queue containing just s
while Q is not empty:
   u = eject(Q)
   for each edge (u,v) in E:
        if dist[v] = ∞:
                inject(Q,v)
                dist[v] = dist[u]+1
                prev[v] = u
```



Queue	Distances					
	a	b	c	d	e	f
[a]	0	∞	∞	∞	∞	∞
[bcd]	0	1	1	1	∞	∞
[cd]	0	1	1	1	∞	∞
[de]	0	1	1	1	2	∞
[e]	0	1	1	1	2	∞
[f]	0	1	1	1	2	3
[]	0	1	1	1	2	3



Shortest path tree

Why does BFS work?

```
procedure bfs(G,s)

for u in V:
    dist[u] = ∞
dist[s] = 0
Q = [s]
while Q is not empty:
    u = eject(Q)
    for each edge (u,v) in E:
        if dist[v] = ∞:
            inject(Q,v)
            dist[v] = dist[u]+1
```

Claim For any distance d = 0,1,2,..., there is a point in time at which:

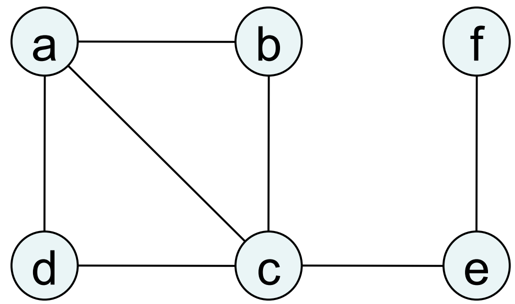
(i) all nodes at distance $\leq$ d have their dist[] values correctly set

(ii) all other nodes have dist[] = ∞

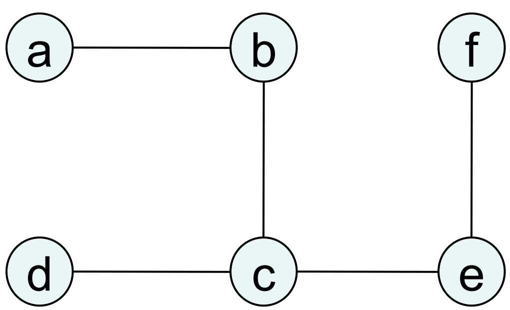
(iii) the queue Q contains exactly the nodes at distance d

Running time: O(V + E), like DFS

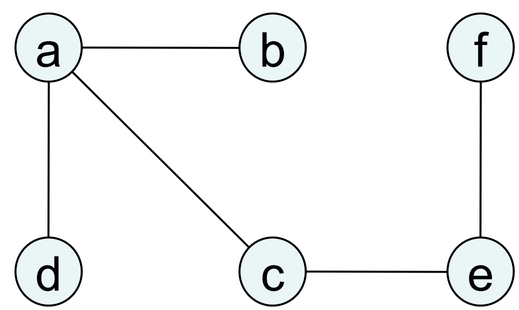
Two search strategies



Depth-first



Breadth-first



Edge lengths

BFS treats all edges as having the same length.
This is rarely true in applications.

Denote the length of edge e = (u,v) by l(e) or l_e or l(u,v)

COL351: Slides for Lecture Component 10

Thanks to Miles Jones, Russell Impagliazzo, and Sanjoy Dasgupta at UCSD for these slides.

Edge lengths

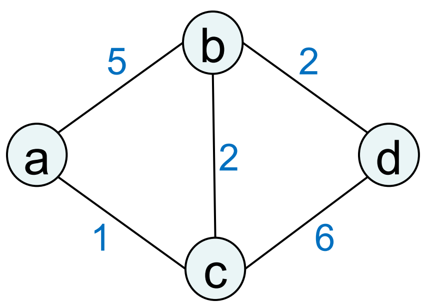
BFS treats all edges as having the same length.
This is rarely true in applications.

Denote the length of edge e = (u,v) by l(e) or l_e or l(u,v)

Extending BFS

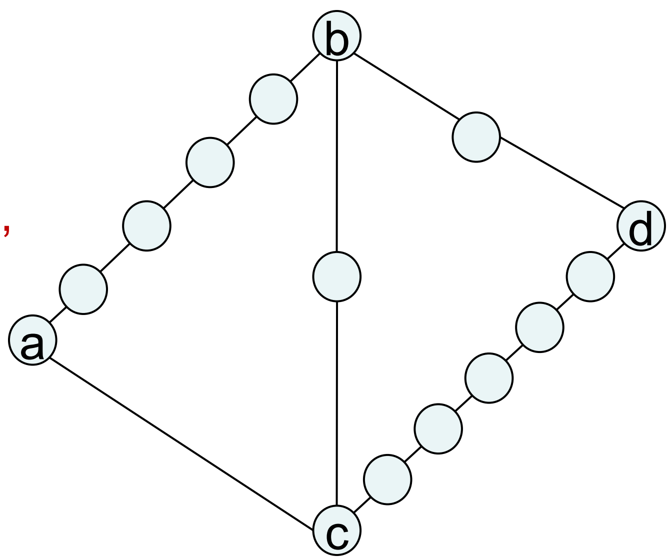
Suppose G has positive integral edge lengths

G



Simple trick: add *dummy nodes*

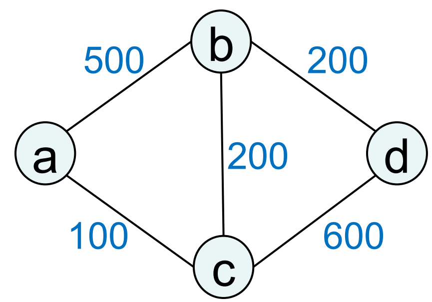
G'



(i) G' has unit-length edges
(ii) For the "real" nodes, distance in G = distance in G'

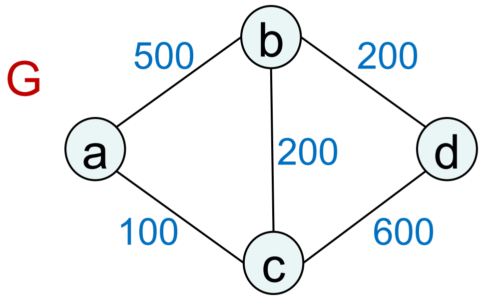
So run BFS on G' !

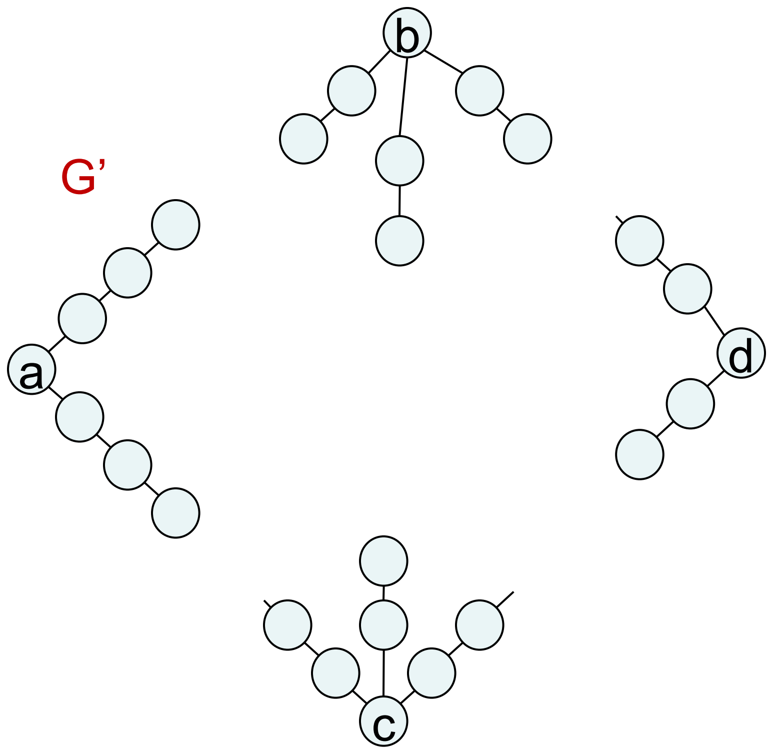
Problem: efficiency



If edge lengths in G are large:
(i) G' is enormous
(ii) BFS wastes a lot of time computing distances to dummy nodes we don't care about

Extending BFS





First 99 time steps: BFS (on G') slowly advances along a—b and a—c. Boring!

Can we snooze and have an alarm wake up us whenever BFS reaches a *real* node?

Alarm for each real node: estimated time of arrival based on edges currently being traversed.

T = 0	set alarms for b (500), c (100) snooze
T = 100	wake up, BFS is at c set alarms for b (300), d (700) snooze
T = 300	wake up, BFS is at b set alarm for d (500) snooze
T = 500	wake up, BFS is at d

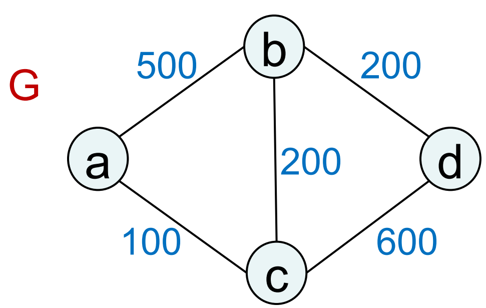
dist[c] = 100
dist[b] = 300
dist[d] = 500

Alarm clock algorithm

(Given graph G and starting node s)

set an alarm for node s at time 0
if the next alarm goes off at time T, for node u:
    distance[u] = T
    for each edge (u,v) in E:
        if no alarm for v, set one for T + l(u,v)
        if there is an alarm for v, but later than
        T + l(u,v), then reset to this earlier time

Exactly simulates BFS on G'...
we no longer need to construct G'!



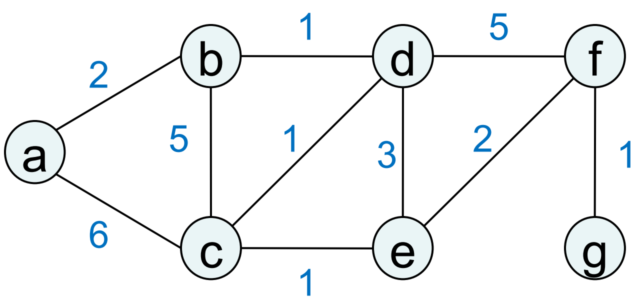
How to implement alarm?
Answer: priority queue (aka heap)

A priority queue H stores:
- a set of elements (our nodes)
-associated key values (alarm times)
and supports these operations:

insert(H,x)	insert new element into H	set a new alarm
deletemin(H)	return element with smallest key value, remove from H	which alarm is going off next?
decreasekey(H,x)	allow x's key value to be decreased	allow alarm to be reset to an earlier time
makequeue(S)	make a queue out of the elements in S (and their keys)	initialize alarms

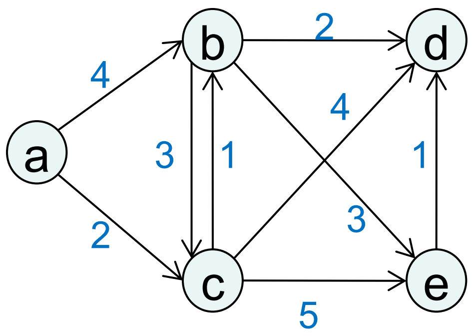
Dijkstra's algorithm

```
procedure dijkstra(G,l,s)

input: graph G = (V,E); node s;
  positive edge lengths le
output: for each node u, dist[u] is
  set to its distance from 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)
```



Another example

```
procedure dijkstra(G,l,s)

for u in V:
  dist[u] = ∞
  prev[u] = nil
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)
        prev[v] = u
        decreasekey(H,v)
```



Running time

```
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

COL351: Slides for Lecture Component 11

Thanks to Miles Jones, Russell Impagliazzo, and Sanjoy Dasgupta at UCSD for these slides.

Running time

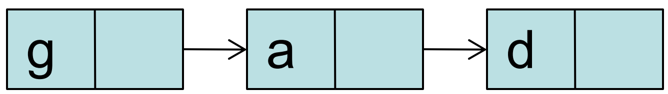
```
procedure dijkstra(G,l,s)

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:

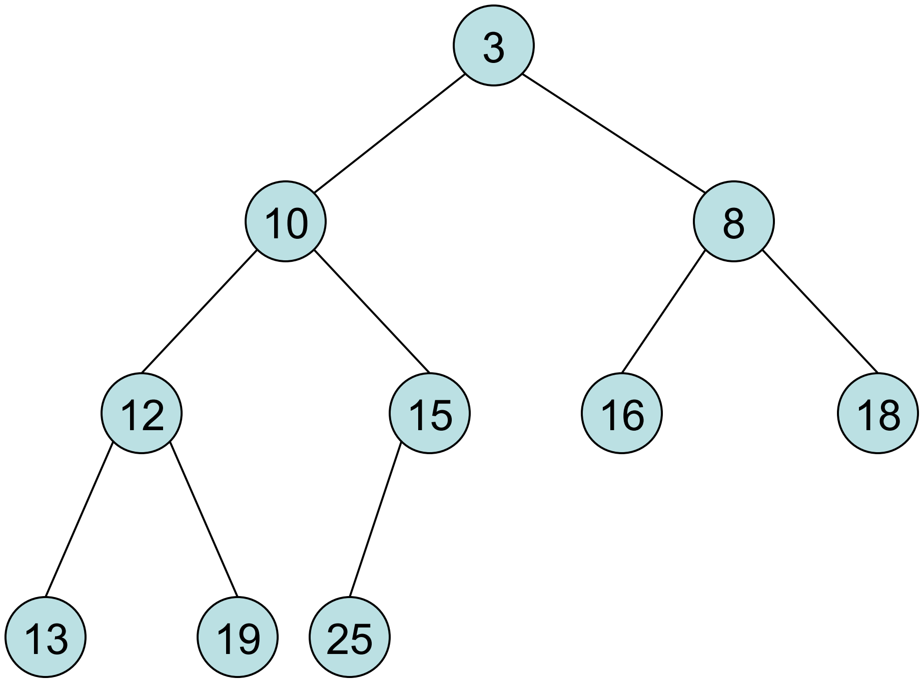
deletemin:

Binary heap

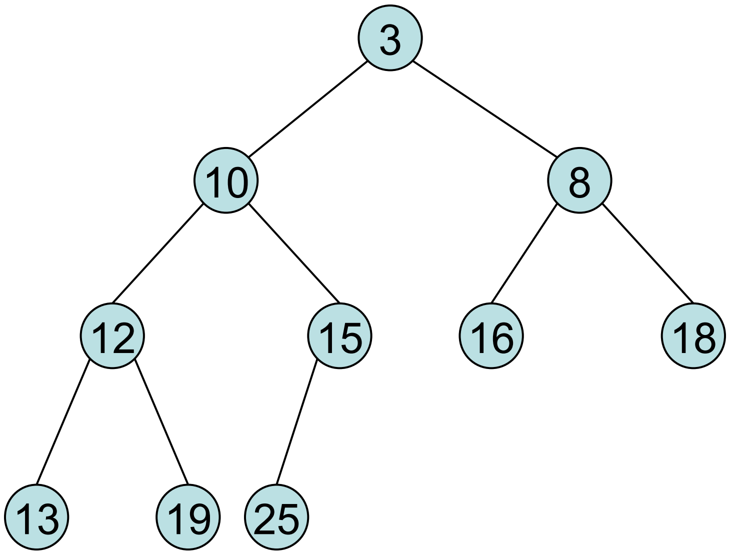
Complete binary tree: filled in row by row, left-to-right

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)

deletemin

d-ary heap

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

height:

insert

deletemin

Running time of Dijkstra's algorithm

	insert, decreasekey	**deletemin**	**V x deletemin + (V+E) x insert**
linked list	O(1)	O(V)	$O(V^2)$
binary heap	O(log V)	O(log V)	O((V+E) log V)
d-ary heap	$O(\log_d V)$	$O(d \log_d V)$	$O((dV + E) \log_d V)$
Fibonacci heap	O(1) amortized	O(log V)	O(E + V log V)

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

Linked list vs. binary heap

Dense graph: E = £(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: $E = V^{1+c}$

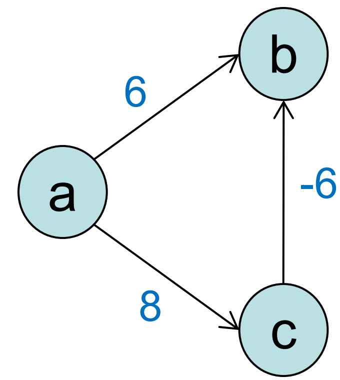
O(E/c), linear!

Dijkstra and negative edges

```
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)
```

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:
(i) are never too small
(ii) get changed only when updating along an edge:

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