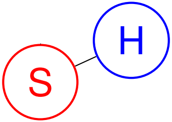
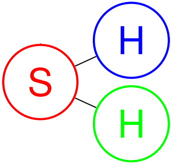
CS 170: Algorithms

CS 170: Algorithms

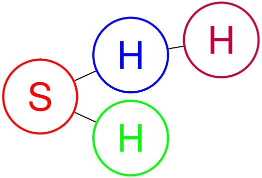
CS 170: Algorithms



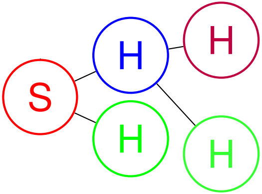
CS 170: Algorithms



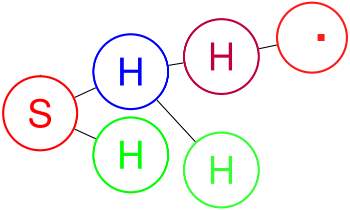
CS 170: Algorithms



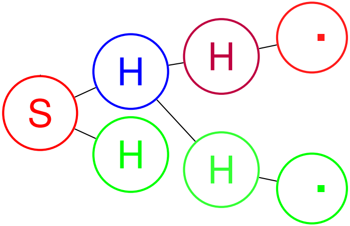
CS 170: Algorithms



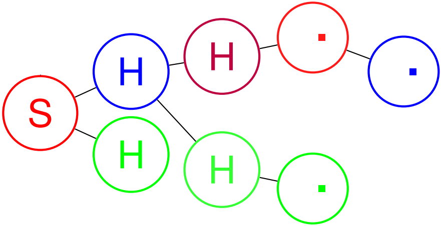
CS 170: Algorithms



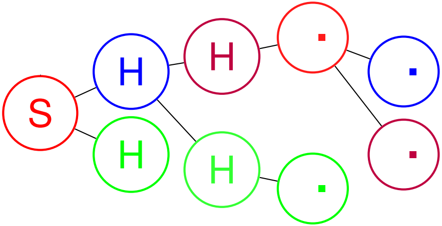
CS 170: Algorithms



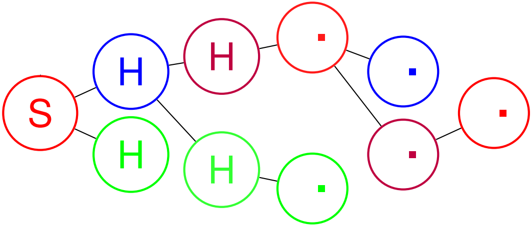
CS 170: Algorithms



CS 170: Algorithms



CS 170: Algorithms



Breadth First Search/Dijkstra.

Lecture in a minute.

Lecture in a minute.

Breadth First Search of graph:
Search with queue instead of stack.
Get “distances” from source.
Proof idea: queue has distance 0, then level 1, . . .

Lecture in a minute.

Breadth First Search of graph:
  Search with queue instead of stack.
  Get “distances” from source.
    Proof idea: queue has distance 0, then level 1, ...

Djikstra: shortest paths in weighted graph:
  Replace weights by paths + BFS
  Implement using priority queue.
    Idea: ignores “new” nodes.
  Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Lecture in a minute.

Breadth First Search of graph:
  Search with queue instead of stack.
  Get "distances" from source.
    Proof idea: queue has distance 0, then level 1, ...

Djikstra: shortest paths in weighted graph:
  Replace weights by paths + BFS
  Implement using priority queue.
    Idea: ignores "new" nodes.
  Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
  Implementation: degree d tree.

Lecture in a minute.

Breadth First Search of graph:
  Search with queue instead of stack.
  Get “distances” from source.
    Proof idea: queue has distance 0, then level 1, ...

Djikstra: shortest paths in weighted graph:
  Replace weights by paths + BFS
  Implement using priority queue.
    Idea: ignores “new” nodes.
  Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
  Implementation: degree d tree.
    Heap Property: children larger than parent.

Lecture in a minute.

Breadth First Search of graph:
  Search with queue instead of stack.
  Get "distances" from source.
    Proof idea: queue has distance 0, then level 1, ...

Djikstra: shortest paths in weighted graph:
  Replace weights by paths + BFS
  Implement using priority queue.
    Idea: ignores "new" nodes.
  Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
  Implementation: degree d tree.
    Heap Property: children larger than parent.
      Minimum at top.

Lecture in a minute.

Breadth First Search of graph:
  Search with queue instead of stack.
  Get “distances” from source.
    Proof idea: queue has distance 0, then level 1, ...

Djikstra: shortest paths in weighted graph:
  Replace weights by paths + BFS
  Implement using priority queue.
    Idea: ignores “new” nodes.
  Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
  Implementation: degree d tree.
    Heap Property: children larger than parent.
      Minimum at top.
    Remove min: $O(d \log_d n)$ time: Replace min/percolate down.

Lecture in a minute.

Breadth First Search of graph:
  Search with queue instead of stack.
  Get "distances" from source.
    Proof idea: queue has distance 0, then level 1, ...

Djikstra: shortest paths in weighted graph:
  Replace weights by paths + BFS
  Implement using priority queue.
    Idea: ignores "new" nodes.
  Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
  Implementation: degree d tree.
    Heap Property: children larger than parent.
      Minimum at top.
    Remove min: $O(d \log_d n)$ time: Replace min/percolate down.
    Reduce Key: $O(\log_d n)$: percolate up.

Lecture in a minute.

Breadth First Search of graph:
  Search with queue instead of stack.
  Get "distances" from source.
    Proof idea: queue has distance 0, then level 1, ...

Djikstra: shortest paths in weighted graph:
  Replace weights by paths + BFS
  Implement using priority queue.
    Idea: ignores "new" nodes.
  Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
  Implementation: degree d tree.
    Heap Property: children larger than parent.
      Minimum at top.
    Remove min: $O(d \log_d n)$ time: Replace min/percolate down.
    Reduce Key: $O(\log_d n)$: percolate up.

Paths in graphs.

$G = (V, E)$.

Paths in graphs.

$G = (V, E)$.
Distance - length of shortest path between u and v.

Paths in graphs.

$G = (V, E)$.
Distance - length of shortest path between u and v.

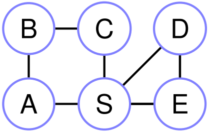
Source s.

Paths in graphs.

$G = (V, E)$.
Distance - length of shortest path between u and v.

Source s.

Definition:
Distance$(s) = 0$, Distance$(v) = \min_{N(v)} d(v) + 1$

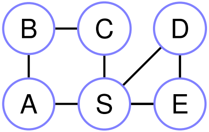


Paths in graphs.

$G = (V, E)$.
Distance - length of shortest path between u and v.

Source s.

Definition:
Distance(s) $= 0$, Distance(v) $= \min_{N(v)} d(v) + 1$



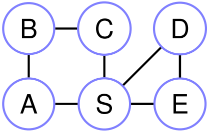
Try depth first search.

Paths in graphs.

$G = (V, E)$.
Distance - length of shortest path between u and v.

Source s.

Definition:
Distance(s) $= 0$, Distance(v) $= \min_{N(v)} d(v) + 1$



Try depth first search.

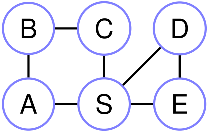
Do you think this will work?

Paths in graphs.

$G = (V, E)$.
Distance - length of shortest path between u and v.

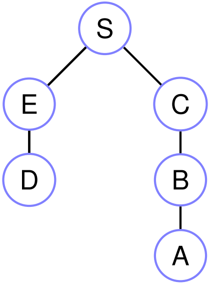
Source s.

Definition:
Distance$(s) = 0$, Distance$(v) = \min_{N(v)} d(v) + 1$



Try depth first search.

Do you think this will work?

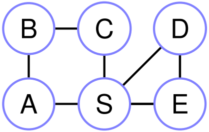


Paths in graphs.

$G = (V, E)$.
Distance - length of shortest path between u and v.

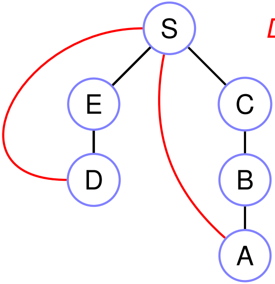
Source s.

Definition:
Distance$(s) = 0$, Distance$(v) = \min_{N(v)} d(v) + 1$



Try depth first search.

Do you think this will work?



D and A at distance 1.

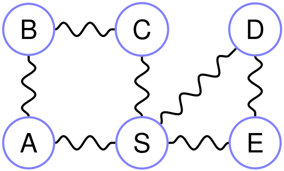
Algorithm 2: Pick it up!

$G = (V, E)$.

Algorithm 2: Pick it up!

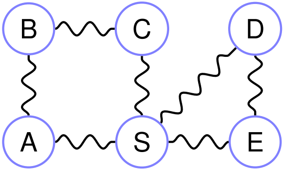
$G = (V, E)$.

Distance - length of shortest path between u and v.



Algorithm 2: Pick it up!

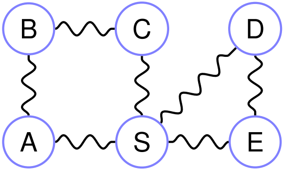
$G = (V, E)$.
Distance - length of shortest path between u and v.



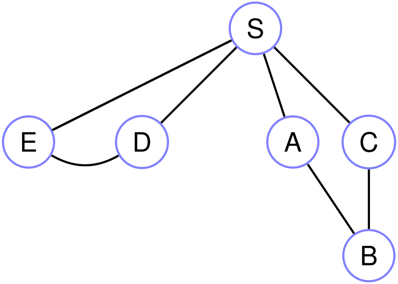
Pick it up by s!

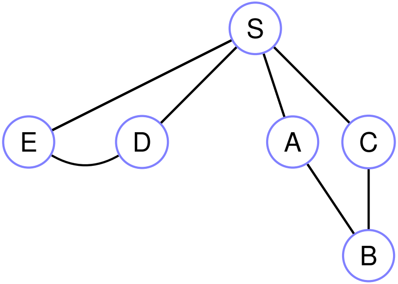
Algorithm 2: Pick it up!

$G = (V, E)$.
Distance - length of shortest path between u and v.



Pick it up by s!

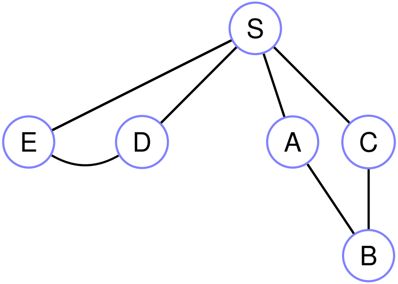




Distance 0.

Distance 1.

Distance 2.

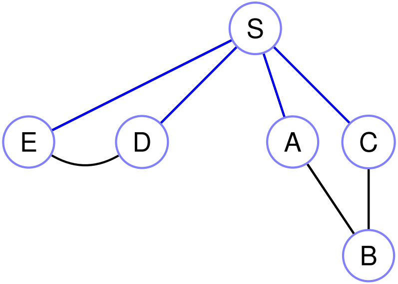


Distance 0.

Distance 1.

Distance 2.

So..proceed layer by layer.
Distance 0.

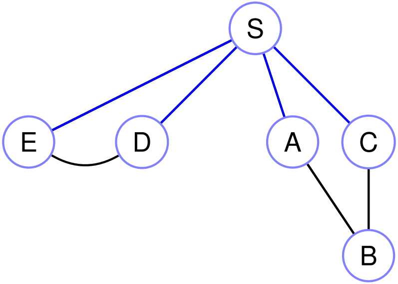


Distance 0.

Distance 1.

Distance 2.

So..proceed layer by layer.
Distance 0.
Neighbors of “0” node are distance 1 nodes.

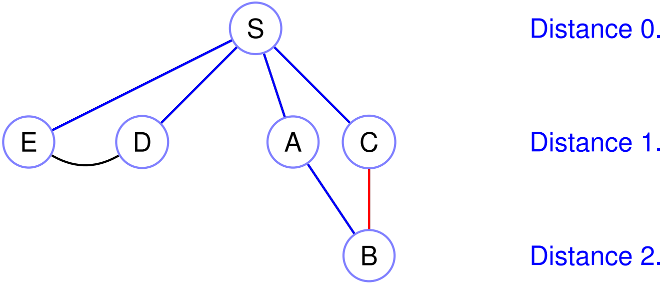


Distance 0.

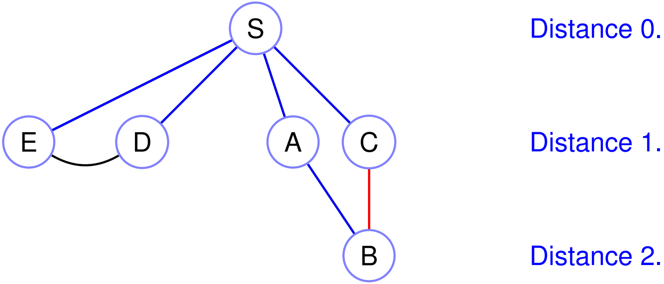
Distance 1.

Distance 2.

So..proceed layer by layer.
Distance 0.
Neighbors of “0” node are distance 1 nodes.
⋮



So..proceed layer by layer.
Distance 0.
Neighbors of “0” node are distance 1 nodes.
⋮
Untouched Neighbors of d nodes are $d + 1$ nodes.



So..proceed layer by layer.
Distance 0.
Neighbors of “0” node are distance 1 nodes.

⋮

Untouched Neighbors of d nodes are $d+1$ nodes.
What data structure should we use to organize this?

Queue: Breadth First Search

Queue: Breadth First Search

d(s) = 0;

Queue: Breadth First Search

d(s) = 0;
visited[s] = **true**

Queue: Breadth First Search

d(s) = 0;
visited[s] = **true**
put *S* in *Q*

Queue: Breadth First Search

d(s) = 0;
visited[s] = **true**
put S in Q
While $u = Q.pop()$:

Queue: Breadth First Search

d(s) = 0;
visited[s] = **true**
put S in Q
While $u = Q.pop()$:
  foreach (u,v):

Queue: Breadth First Search

d(s) = 0;
visited[s] = **true**
put S in Q
While $u = Q.pop()$:
  foreach (u,v):
    if (visited[v] == **false**):

Queue: Breadth First Search

```
d(s) = 0;
visited[s] = true
put S in Q
While u = Q.pop():
   foreach (u,v):
      if (visited[v] == false):
         visited[v] = true
```

Queue: Breadth First Search

```
d(s) = 0;
visited[s] = true
put S in Q
While u = Q.pop():
   foreach (u,v):
      if (visited[v] == false):
         visited[v] = true
         d[v] = d[u]+1
```

Queue: Breadth First Search

d(s) = 0;
visited[s] = **true**
put *S* in *Q*
While $u = Q.pop()$:
    foreach (u,v):
        if (visited[v] == **false**):
            visited[v] = **true**
            d[v] = d[u]+1
            put *v* in *Q*

Queue: Breadth First Search

d(s) = 0;
visited[s] = **true**
put S in Q
While $u = Q.pop()$:
  foreach (u,v):
    if (visited[v] == **false**):
      visited[v] = **true**
      d[v] = d[u]+1
      put v in Q
Nodes “explored” in order of distance from s.

Correctness.

BFS:

Correctness.

BFS: while node u in queue;

Correctness.

BFS: while node u in queue;
add unvisited neighbors of u to queue.

Correctness.

BFS: while node u in queue;
add unvisited neighbors of u to queue.

Inductive statement:.

Correctness.

BFS: while node u in queue;
add unvisited neighbors of u to queue.

Inductive statement:.
Queue only has distance d and $d+1$ nodes.

Correctness.

BFS: while node u in queue;
add unvisited neighbors of u to queue.

Inductive statement:.
Queue only has distance d and $d+1$ nodes.
When all distance d nodes explored,

Correctness.

BFS: while node u in queue;
add unvisited neighbors of u to queue.

Inductive statement:.
Queue only has distance d and $d+1$ nodes.
When all distance d nodes explored,
queue has all (and only) distance $d+1$ nodes.

Correctness.

BFS: while node u in queue;
add unvisited neighbors of u to queue.

Inductive statement:.
Queue only has distance d and $d+1$ nodes.
When all distance d nodes explored,
queue has all (and only) distance $d+1$ nodes.

Prove queue property.

Correctness.

BFS: while node u in queue;
add unvisited neighbors of u to queue.

Inductive statement:.
Queue only has distance d and $d+1$ nodes.
When all distance d nodes explored,
queue has all (and only) distance $d+1$ nodes.

Prove queue property.
Base: s is explored first.

Correctness.

BFS: while node u in queue;
add unvisited neighbors of u to queue.

Inductive statement:.
Queue only has distance d and $d+1$ nodes.
When all distance d nodes explored,
queue has all (and only) distance $d+1$ nodes.

Prove queue property.
Base: s is explored first.
$\implies$ all its neighbors in queue

Correctness.

BFS: while node u in queue;
add unvisited neighbors of u to queue.

Inductive statement:.
Queue only has distance d and $d+1$ nodes.
When all distance d nodes explored,
queue has all (and only) distance $d+1$ nodes.

Prove queue property.
Base: s is explored first.
$\implies$ all its neighbors in queue
$\implies$ queue has all distance 1 nodes.

Correctness.

BFS: while node u in queue;
add unvisited neighbors of u to queue.

Inductive statement:.
Queue only has distance d and $d+1$ nodes.
When all distance d nodes explored,
queue has all (and only) distance $d+1$ nodes.

Prove queue property.
Base: s is explored first.
$\implies$ all its neighbors in queue
$\implies$ queue has all distance 1 nodes.

$d+1$ node has a distance d neighbor def of distance

Correctness.

BFS: while node u in queue;
add unvisited neighbors of u to queue.

Inductive statement:.
Queue only has distance d and $d+1$ nodes.
When all distance d nodes explored,
queue has all (and only) distance $d+1$ nodes.

Prove queue property.
Base: s is explored first.
$\implies$ all its neighbors in queue
$\implies$ queue has all distance 1 nodes.

$d+1$ node has a distance d neighbor def of distance
last distance d node explored (queue!)

Correctness.

BFS: while node u in queue;
add unvisited neighbors of u to queue.

Inductive statement:.
Queue only has distance d and $d+1$ nodes.
When all distance d nodes explored,
queue has all (and only) distance $d+1$ nodes.

Prove queue property.
Base: s is explored first.
$\implies$ all its neighbors in queue
$\implies$ queue has all distance 1 nodes.

$d+1$ node has a distance d neighbor def of distance
last distance d node explored (queue!)
$\implies$ every node at distance $d+1$ is visited

Correctness.

BFS: while node u in queue;
add unvisited neighbors of u to queue.

Inductive statement:.
Queue only has distance d and $d+1$ nodes.
When all distance d nodes explored,
queue has all (and only) distance $d+1$ nodes.

Prove queue property.
Base: s is explored first.
$\implies$ all its neighbors in queue
$\implies$ queue has all distance 1 nodes.

$d+1$ node has a distance d neighbor def of distance
last distance d node explored (queue!)
$\implies$ every node at distance $d+1$ is visited
$\implies$ queue has all dist. $d+1$ nodes.

Correctness.

BFS: while node u in queue;
add unvisited neighbors of u to queue.

Inductive statement:.
Queue only has distance d and $d+1$ nodes.
When all distance d nodes explored,
queue has all (and only) distance $d+1$ nodes.

Prove queue property.
Base: s is explored first.
$\implies$ all its neighbors in queue
$\implies$ queue has all distance 1 nodes.

$d+1$ node has a distance d neighbor def of distance
last distance d node explored (queue!)
$\implies$ every node at distance $d+1$ is visited
$\implies$ queue has all dist. $d+1$ nodes.

Has **no** $l > d+1$ **nodes** since only $\leq d$ level nodes explored.

Correctness.

BFS: while node u in queue;
add unvisited neighbors of u to queue.

Inductive statement:.
Queue only has distance d and $d+1$ nodes.
When all distance d nodes explored,
queue has all (and only) distance $d+1$ nodes.

Prove queue property.
Base: s is explored first.
$\implies$ all its neighbors in queue
$\implies$ queue has all distance 1 nodes.

$d+1$ node has a distance d neighbor def of distance
last distance d node explored (queue!)
$\implies$ every node at distance $d+1$ is visited
$\implies$ queue has all dist. $d+1$ nodes.

Has **no** $l > d+1$ **nodes** since only $\leq d$ level nodes explored.
Has **no** $l < d$ **nodes**, since removed by induction.

Correctness.

BFS: while node u in queue;
add unvisited neighbors of u to queue.

Inductive statement:.
Queue only has distance d and $d+1$ nodes.
When all distance d nodes explored,
queue has all (and only) distance $d+1$ nodes.

Prove queue property.
Base: s is explored first.
$\implies$ all its neighbors in queue
$\implies$ queue has all distance 1 nodes.

$d+1$ node has a distance d neighbor def of distance
last distance d node explored (queue!)
$\implies$ every node at distance $d+1$ is visited
$\implies$ queue has all dist. $d+1$ nodes.

Has **no** $l > d+1$ **nodes** since only $\leq d$ level nodes explored.
Has **no** $l < d$ **nodes**, since removed by induction.
$\implies$ only distance $d+1$ nodes.

In action!

Breadth First Search:

In action!

Breadth First Search:

$d(s) = 0$

In action!

Breadth First Search:

$d(s) = 0$
visited[s] = **true**

In action!

Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q

In action!

Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q
While $u = Q$:

In action!

Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q
While $u = Q$:
  foreach (u,v):

In action!

Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q
While $u = Q$:
  foreach (u,v):
    if (visited[v] == **false**):

In action!

Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q
While $u = Q$:
    foreach (u,v):
        if (visited[v] == **false**):
            visited[v] = **true**

In action!

Breadth First Search:

```
d(s) = 0
visited[s] = true
put S in Q
While u = Q:
   foreach (u,v):
      if (visited[v] == false):
         visited[v] = true
         d[v] = d[u]+1
```

In action!

Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q
While $u = Q$:
  foreach (u,v):
    if (visited[v] == **false**):
      visited[v] = **true**
      d[v] = d[u]+1
      put v in Q

In action!

Breadth First Search:

```
d(s) = 0
visited[s] = true
put S in Q
While u = Q:
   foreach (u,v):
      if (visited[v] == false):
         visited[v] = true
         d[v] = d[u]+1
         put v in Q
```

In action!

Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q
While $u = Q$:
  foreach (u,v):
    if (visited[v] == **false**):
      visited[v] = **true**
      d[v] = d[u]+1
      put v in Q

Queue: S
0

In action!

Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q
While $u = Q$:
  foreach (u,v):
    if (visited[v] == **false**):
      visited[v] = **true**
      d[v] = d[u]+1
      put v in Q

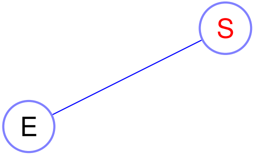
Queue:

In action!

Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q
While $u = Q$:
  foreach (u,v):
    if (visited[v] == **false**):
      visited[v] = **true**
      d[v] = d[u]+1
      put v in Q

Queue: E
1

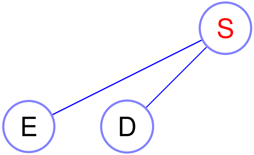


In action!

Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q
While $u = Q$:
  foreach (u,v):
    if (visited[v] == **false**):
      visited[v] = **true**
      d[v] = d[u]+1
      put v in Q

Queue: E, D
$1, 1$

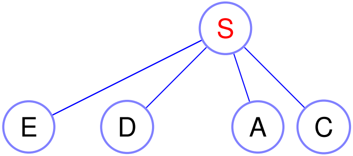


In action!

Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q
While $u = Q$:
  foreach (u,v):
    if (visited[v] == **false**):
      visited[v] = **true**
      d[v] = d[u]+1
      put v in Q

Queue: E, D, A, C
$1, 1, 1, 1$

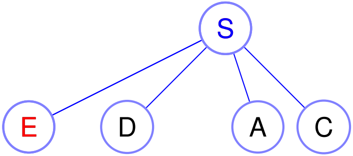


In action!

Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q
While $u = Q$:
  foreach (u,v):
    if (visited[v] == **false**):
      visited[v] = **true**
      d[v] = d[u]+1
      put v in Q

Queue: D, A, C
$1, 1, 1$

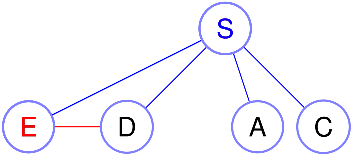


In action!

Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q
While $u = Q$:
  foreach (u,v):
    if (visited[v] == **false**):
      visited[v] = **true**
      d[v] = d[u]+1
      put v in Q

Queue: D, A, C
$1, 1, 1$

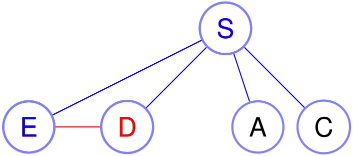


In action!

Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q
While $u = Q$:
  foreach (u,v):
    if (visited[v] == **false**):
      visited[v] = **true**
      d[v] = d[u]+1
      put v in Q

Queue: A, C
$1, 1$

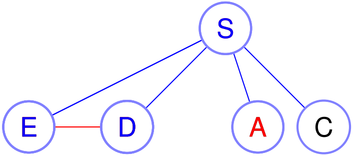


In action!

Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q
While $u = Q$:
  foreach (u,v):
    if (visited[v] == **false**):
      visited[v] = **true**
      d[v] = d[u]+1
      put v in Q

Queue: C
1

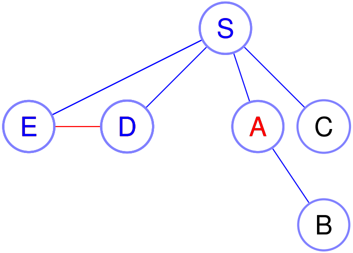


In action!

Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q
While $u = Q$:
  foreach (u,v):
    if (visited[v] == **false**):
      visited[v] = **true**
      d[v] = d[u]+1
      put v in Q

Queue: C, B
$1, 2$

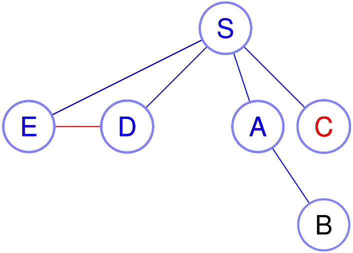


In action!

Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q
While $u = Q$:
  foreach (u,v):
    if (visited[v] == **false**):
      visited[v] = **true**
      d[v] = d[u]+1
      put v in Q

Queue: B
2



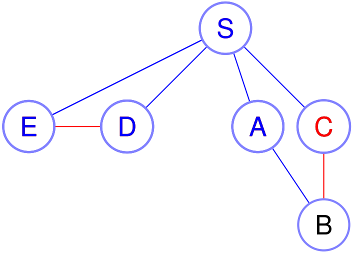
In action!

Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q
While $u = Q$:
  foreach (u,v):
    if (visited[v] == **false**):
      visited[v] = **true**
      d[v] = d[u]+1
      put v in Q

Queue: B

2

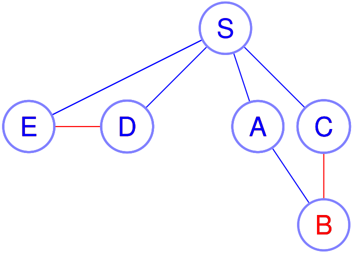


In action!

Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q
While $u = Q$:
  foreach (u,v):
    if (visited[v] == **false**):
      visited[v] = **true**
      d[v] = d[u]+1
      put v in Q

Queue:

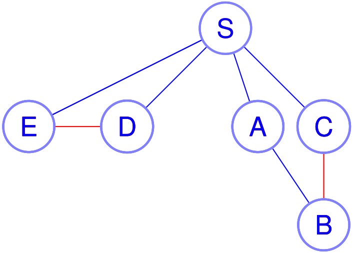


In action!

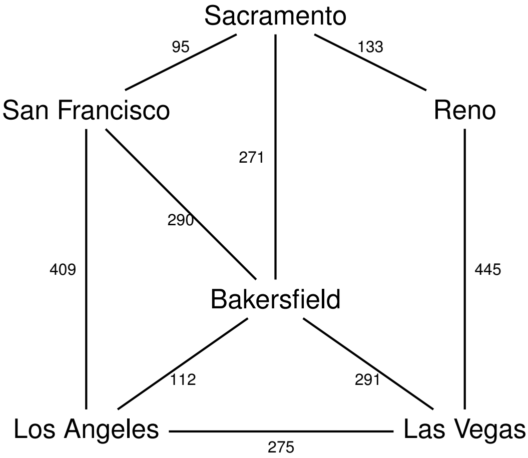
Breadth First Search:

d(s) = 0
visited[s] = **true**
put S in Q
While $u = Q$:
  foreach (u,v):
    if (visited[v] == **false**):
      visited[v] = **true**
      d[v] = d[u]+1
      put v in Q

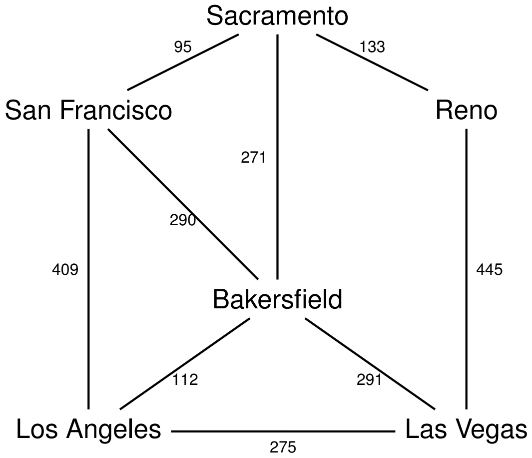
Queue:



Edge lengths: BFS?

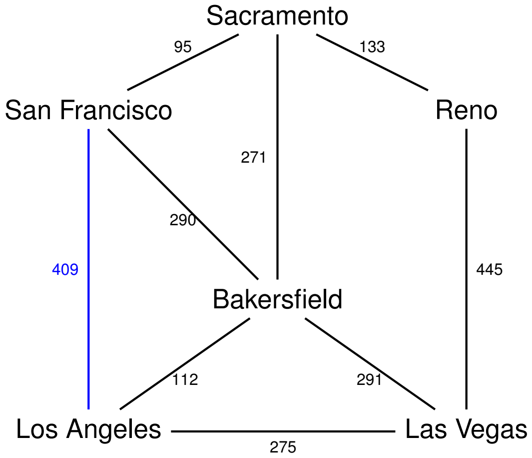


Edge lengths: BFS?



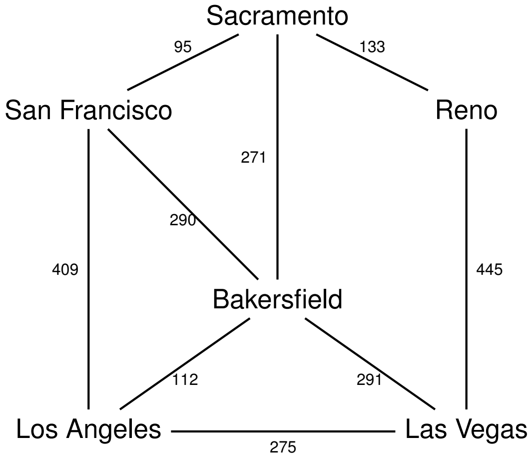
S.F. to L. A.:

Edge lengths: BFS?



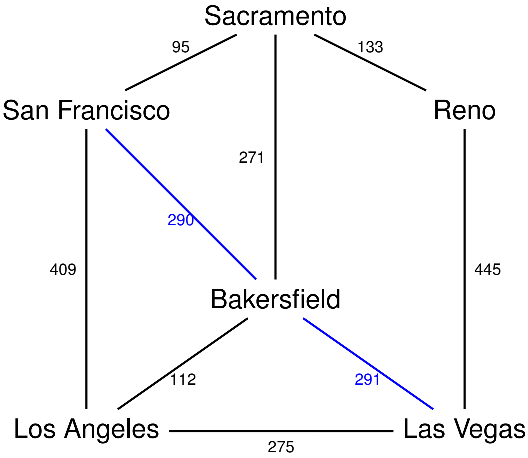
S.F. to L. A.: One hop.

Edge lengths: BFS?



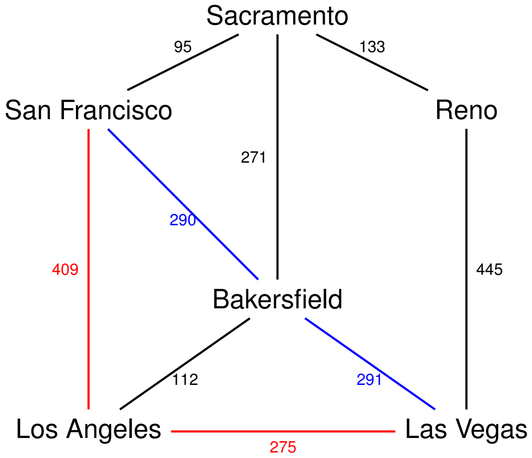
S.F. to L. A.: One hop.
S.F. to Vegas:

Edge lengths: BFS?



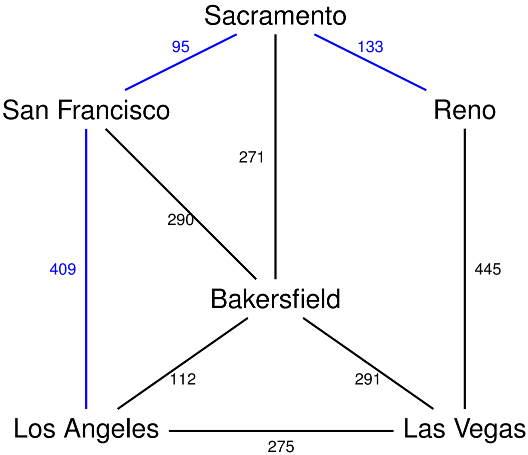
S.F. to L. A.: One hop.
S.F. to Vegas: Two hops

Edge lengths: BFS?



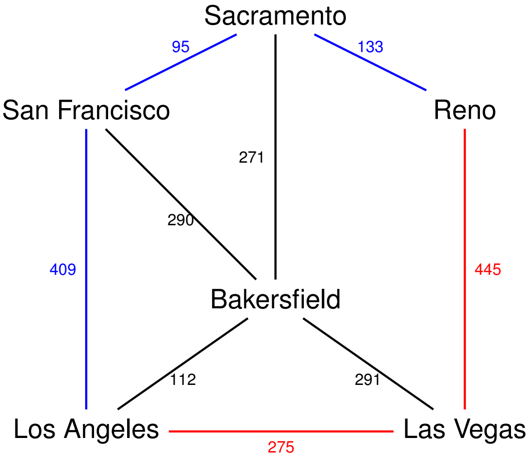
S.F. to L. A.: One hop.
S.F. to Vegas: Two hops (which two).

Edge lengths: BFS?



S.F. to L. A.: One hop.
S.F. to Vegas: Two hops (which two).
Reno to L.A.: Three hops.

Edge lengths: BFS?



S.F. to L. A.: One hop.
S.F. to Vegas: Two hops (which two).
Reno to L.A.: Three hops. not two!

BFS with edge lengths?

Graph: $G = (V, E)$.

BFS with edge lengths?

Graph: $G = (V, E)$.
Length of e: l_e.

BFS with edge lengths?

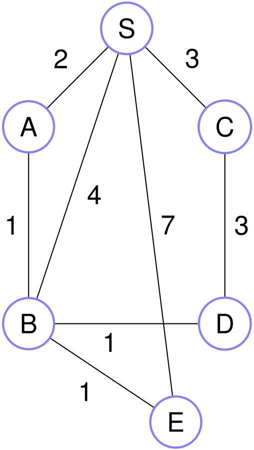
Graph: $G = (V, E)$.
Length of e: l_e.

Find shortest paths from s.

BFS with edge lengths?

Graph: $G = (V, E)$.
Length of e: l_e.

Find shortest paths from s.

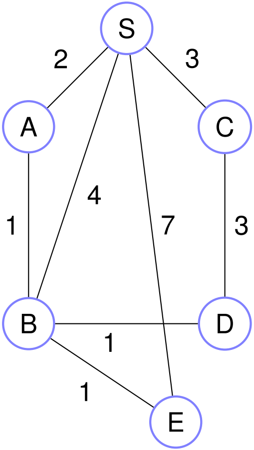


Make G' from G.

BFS with edge lengths?

Graph: $G = (V, E)$.
Length of e: l_e.

Find shortest paths from s.

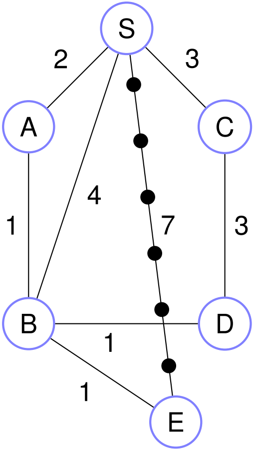


Make G' from G.
 For each edge e
  Replace w/len l_e path.

BFS with edge lengths?

Graph: $G = (V, E)$.
Length of e: l_e.

Find shortest paths from s.

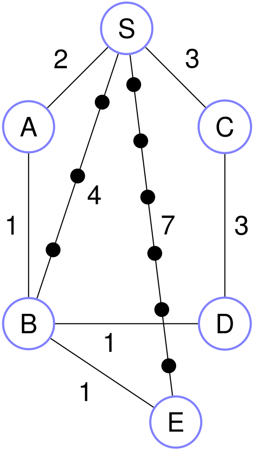


Make G' from G.
For each edge e
Replace w/len l_e path.

BFS with edge lengths?

Graph: $G = (V, E)$.
Length of e: l_e.

Find shortest paths from s.

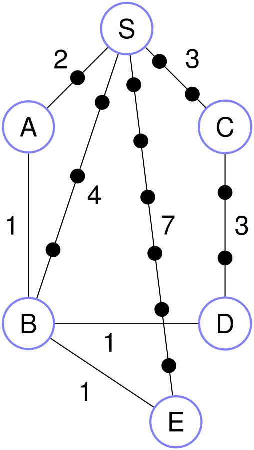


Make G' from G.
For each edge e
Replace w/len l_e path.

BFS with edge lengths?

Graph: $G = (V, E)$.
Length of e: l_e.

Find shortest paths from s.

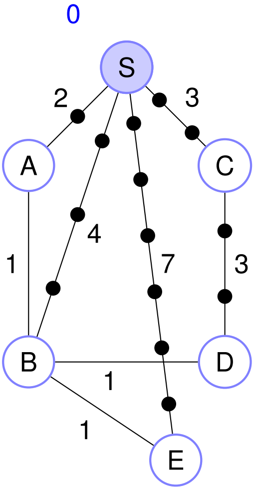


Make G' from G.
For each edge e
Replace w/len l_e path.

BFS with edge lengths?

Graph: $G = (V, E)$.
Length of e: l_e.

Find shortest paths from s.



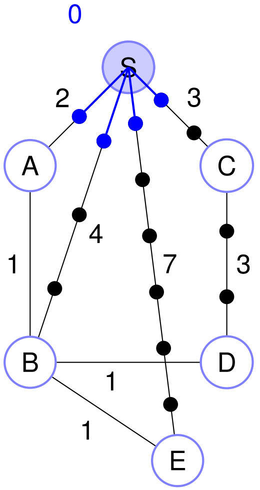
Make G' from G.
For each edge e
Replace w/len l_e path.

Run BFS on G'.

BFS with edge lengths?

Graph: $G = (V, E)$.
Length of e: l_e.

Find shortest paths from s.



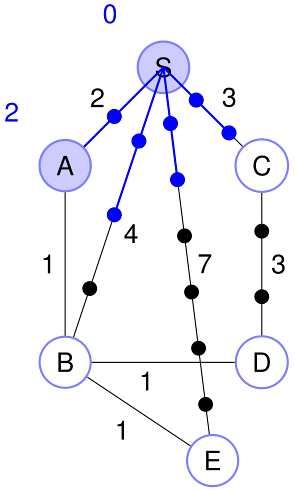
Make G' from G.
For each edge e
Replace w/len l_e path.

Run BFS on G'.

BFS with edge lengths?

Graph: $G = (V, E)$.
Length of e: l_e.

Find shortest paths from s.



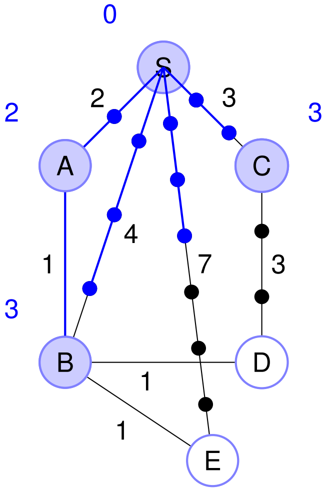
Make G' from G.
For each edge e
Replace w/len l_e path.

Run BFS on G'.

BFS with edge lengths?

Graph: $G = (V, E)$.
Length of e: l_e.

Find shortest paths from s.



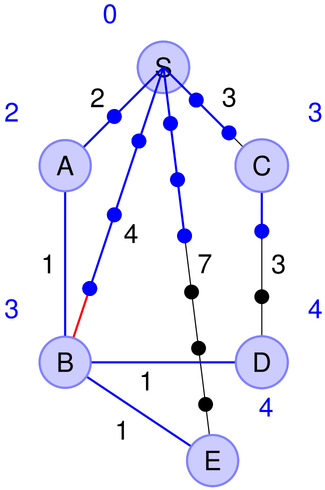
Make G' from G.
For each edge e
Replace w/len l_e path.

Run BFS on G'.

BFS with edge lengths?

Graph: $G = (V, E)$.
Length of e: l_e.

Find shortest paths from s.



Make G' from G.
For each edge e
Replace w/len l_e path.

Run BFS on G'.

BFS in exploded graph.

Correct: follows from the correctness of BFS.

BFS in exploded graph.

Correct: follows from the correctness of BFS.

Time?

BFS in exploded graph.

Correct: follows from the correctness of BFS.

Time? $O(\sum_e l_e)$.

BFS in exploded graph.

Correct: follows from the correctness of BFS.

Time? $O(\sum_e l_e)$.

Very very large.

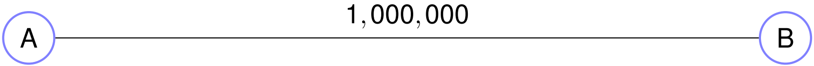
BFS in exploded graph.

Correct: follows from the correctness of BFS.

Time? $O(\sum_e l_e)$.

Very very large.

Compared to size of problem.



Size of representation: 6 digits plus one edge.

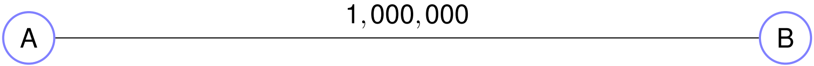
BFS in exploded graph.

Correct: follows from the correctness of BFS.

Time? $O(\sum_e l_e)$.

Very very large.

Compared to size of problem.



Size of representation: 6 digits plus one edge. 10 ish.

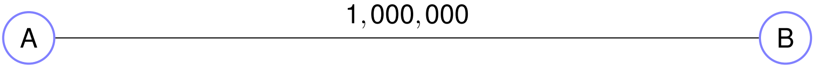
BFS in exploded graph.

Correct: follows from the correctness of BFS.

Time? $O(\sum_e l_e)$.

Very very large.

Compared to size of problem.



Size of representation: 6 digits plus one edge. 10 ish.

Time: 1,000,000.

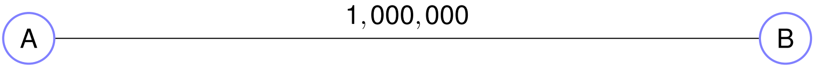
BFS in exploded graph.

Correct: follows from the correctness of BFS.

Time? $O(\sum_e l_e)$.

Very very large.

Compared to size of problem.



Size of representation: 6 digits plus one edge. 10 ish.

Time: 1,000,000. ...ish.

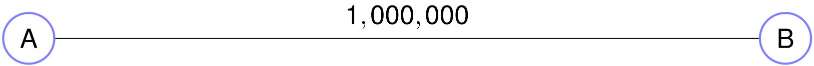
BFS in exploded graph.

Correct: follows from the correctness of BFS.

Time? $O(\sum_e l_e)$.

Very very large.

Compared to size of problem.



Size of representation: 6 digits plus one edge. 10 ish.

Time: 1,000,000. ...ish.

Hmmm...

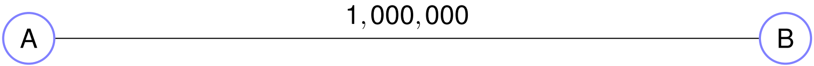
BFS in exploded graph.

Correct: follows from the correctness of BFS.

Time? $O(\sum_e l_e)$.

Very very large.

Compared to size of problem.



Size of representation: 6 digits plus one edge. 10 ish.

Time: 1,000,000. ...ish.

Hmmm...

The distance is

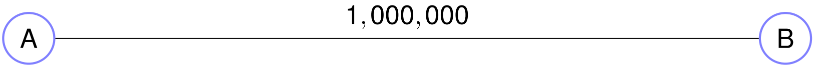
BFS in exploded graph.

Correct: follows from the correctness of BFS.

Time? $O(\sum_e l_e)$.

Very very large.

Compared to size of problem.



Size of representation: 6 digits plus one edge. 10 ish.

Time: 1,000,000. ...ish.

Hmmm...

The distance is ..obviously 1,000,000.

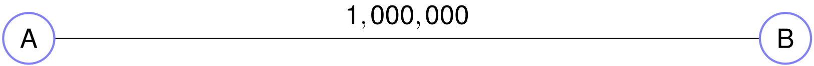
BFS in exploded graph.

Correct: follows from the correctness of BFS.

Time? $O(\sum_e l_e)$.

Very very large.

Compared to size of problem.



Size of representation: 6 digits plus one edge. 10 ish.

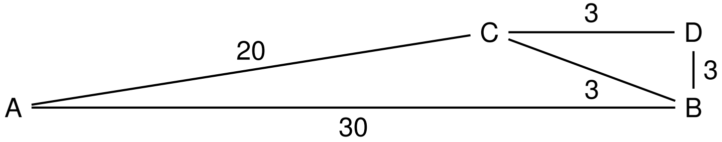
Time: 1,000,000. ...ish.

Hmmm...

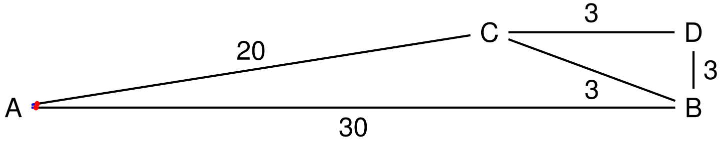
The distance is ..obviously 1,000,000.

Could it be easier?

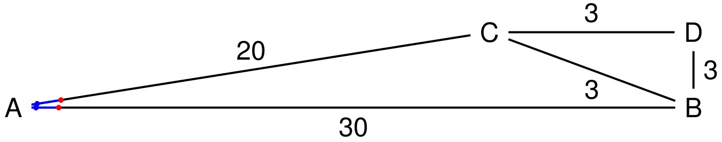
Looking again.



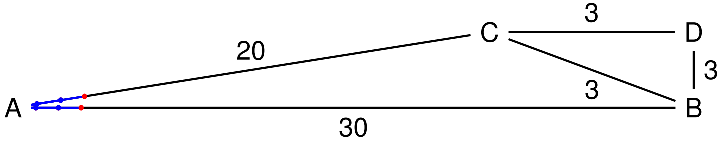
Looking again.



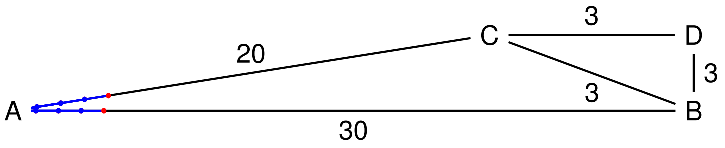
Looking again.



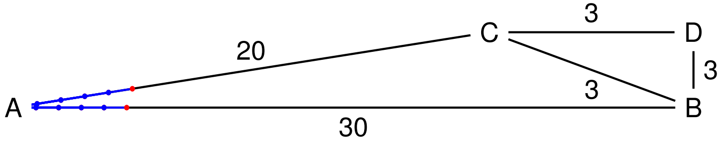
Looking again.



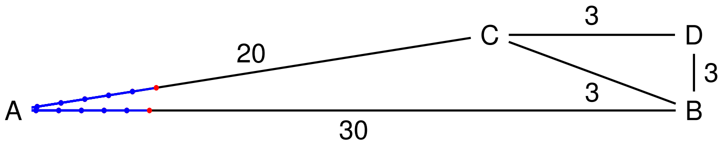
Looking again.



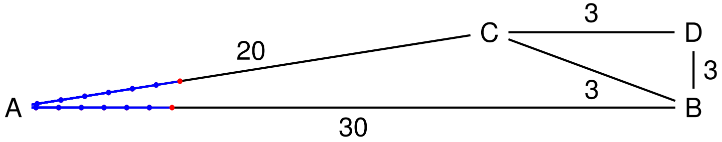
Looking again.



Looking again.

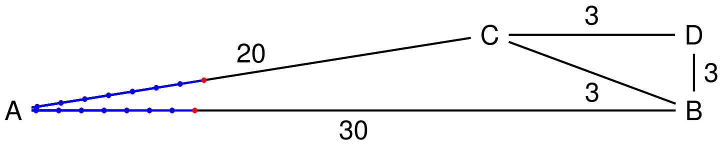


Looking again.



Queue next node on long edges again and again...

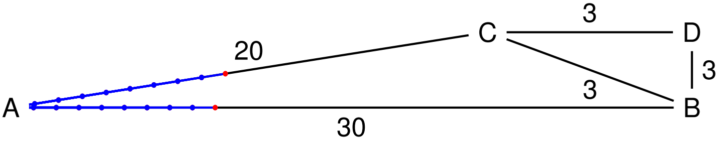
Looking again.



Queue next node on long edges again and again...

Nothing interesting until 20 steps.

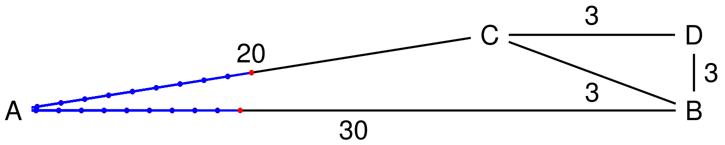
Looking again.



Queue next node on long edges again and again...

Nothing interesting until 20 steps.

Looking again.

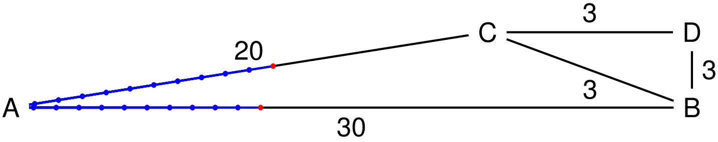


Queue next node on long edges again and again...

Nothing interesting until 20 steps.

Wake me then!

Looking again.

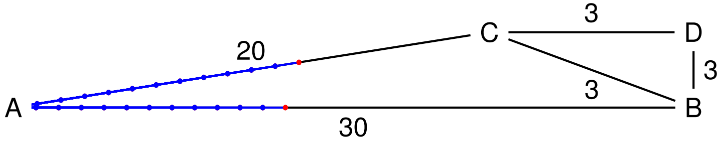


Queue next node on long edges again and again...

Nothing interesting until 20 steps.

Wake me then!

Looking again.

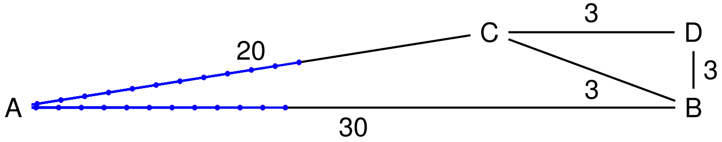


Queue next node on long edges again and again...

Nothing interesting until 20 steps.

Wake me then!

Looking again.



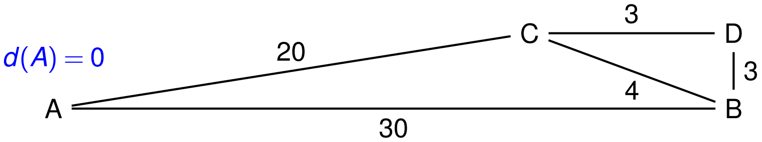
Queue next node on long edges again and again...

Nothing interesting until 20 steps.

Wake me then!

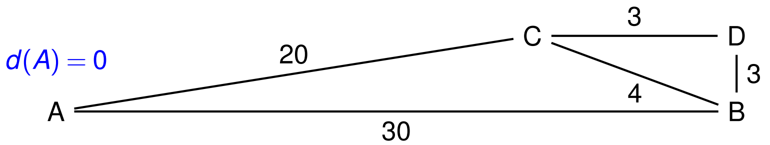
Alarms.

Alarms.



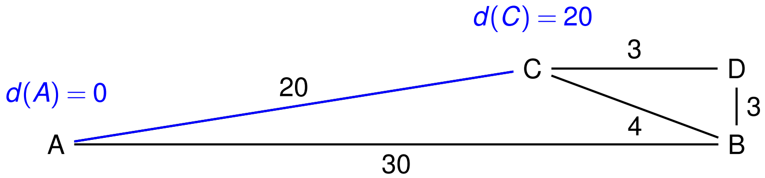
Process A: $d(A) = 0$

Alarms.



Process A: $d(A) = 0$
For Edge (A, C):

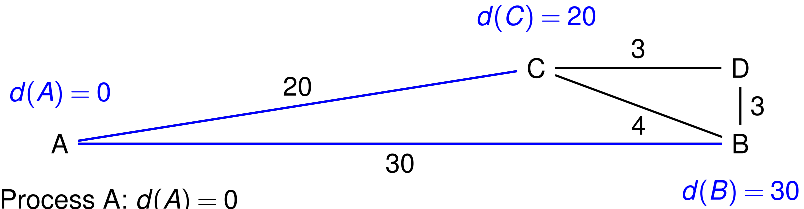
Alarms.



Process A: $d(A) = 0$

For Edge (A, C): Set $d(C) = d(A) + 20 = 20$ and alarm for 20.

Alarms.

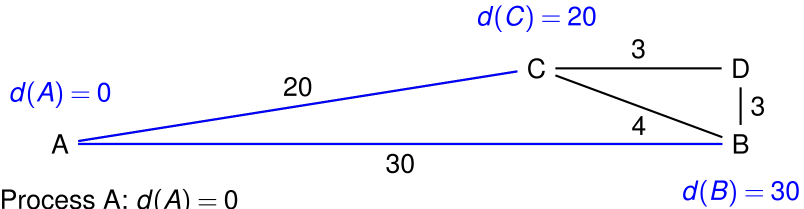


Process A: $d(A) = 0$

For Edge (A, C): Set $d(C) = d(A) + 20 = 20$ and alarm for 20.

For Edge (A, B):

Alarms.

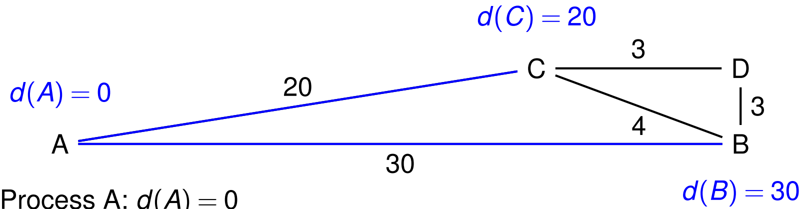


Process A: $d(A) = 0$

For Edge (A, C): Set $d(C) = d(A) + 20 = 20$ and alarm for 20.

For Edge (A, B): Set $d(B) = d(A) + 30 = 30$ and alarm for 30.

Alarms.



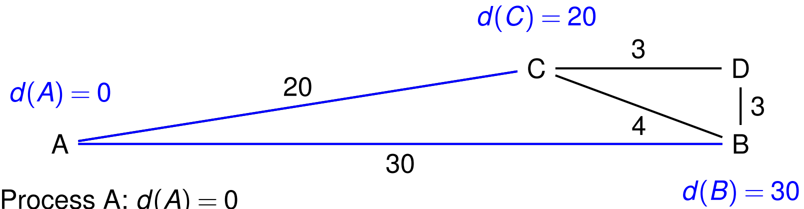
Process A: $d(A) = 0$

For Edge (A, C): Set $d(C) = d(A) + 20 = 20$ and alarm for 20.

For Edge (A, B): Set $d(B) = d(A) + 30 = 30$ and alarm for 30.

At time 20:

Alarms.



Process A: $d(A) = 0$

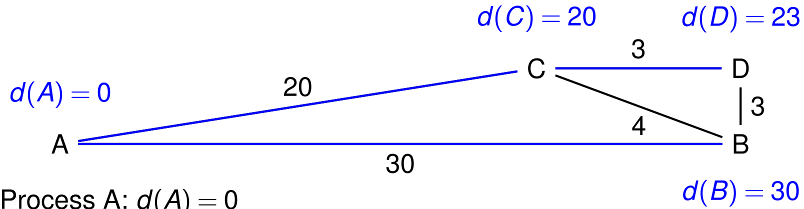
For Edge (A, C): Set $d(C) = d(A) + 20 = 20$ and alarm for 20.

For Edge (A, B): Set $d(B) = d(A) + 30 = 30$ and alarm for 30.

At time 20:

Process C.

Alarms.



Process A: $d(A) = 0$

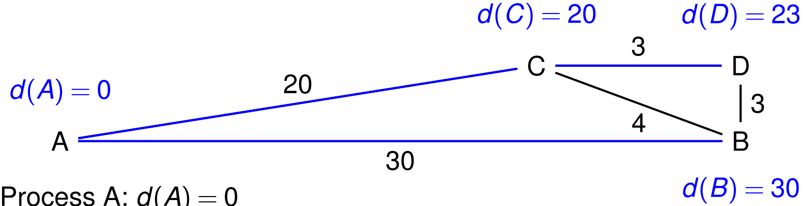
For Edge (A, C): Set $d(C) = d(A) + 20 = 20$ and alarm for 20.

For Edge (A, B): Set $d(B) = d(A) + 30 = 30$ and alarm for 30.

At time 20:

Process C.

Alarms.



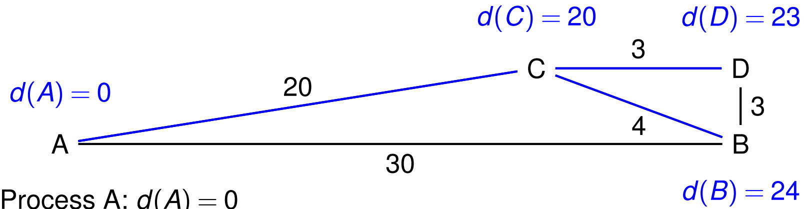
Process A: $d(A) = 0$

For Edge (A, C): Set $d(C) = d(A) + 20 = 20$ and alarm for 20.
For Edge (A, B): Set $d(B) = d(A) + 30 = 30$ and alarm for 30.

At time 20:
Process C.

For Edge (C,D): Set $d(D) = d(C) + 3 = 23$ and alarm for 23.

Alarms.



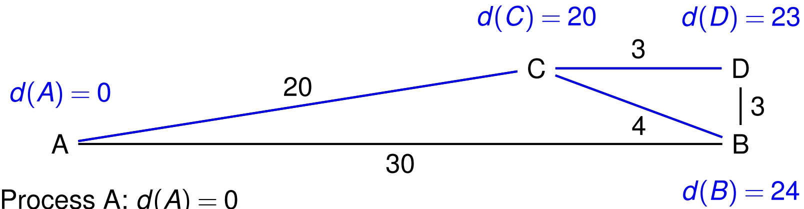
Process A: $d(A) = 0$

For Edge (A, C): Set $d(C) = d(A) + 20 = 20$ and alarm for 20.
For Edge (A, B): Set $d(B) = d(A) + 30 = 30$ and alarm for 30.

At time 20:
Process C.

For Edge (C,D): Set $d(D) = d(C) + 3 = 23$ and alarm for 23.
For Edge (C, B):

Alarms.



Process A: $d(A) = 0$

For Edge (A, C): Set $d(C) = d(A) + 20 = 20$ and alarm for 20.

For Edge (A, B): Set $d(B) = d(A) + 30 = 30$ and alarm for 30.

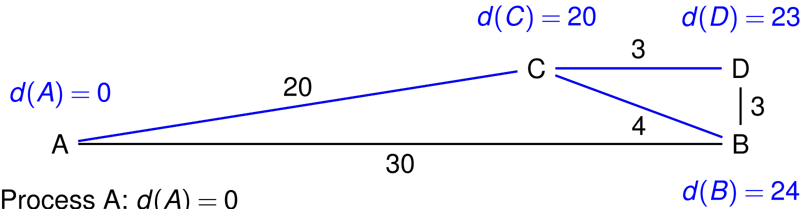
At time 20:

Process C.

For Edge (C,D): Set $d(D) = d(C) + 3 = 23$ and alarm for 23.

For Edge (C, B): Reset $d(B) = 24$ and alarm for 24.

Alarms.



Process A: $d(A) = 0$

For Edge (A, C): Set $d(C) = d(A) + 20 = 20$ and alarm for 20.

For Edge (A, B): Set $d(B) = d(A) + 30 = 30$ and alarm for 30.

At time 20:

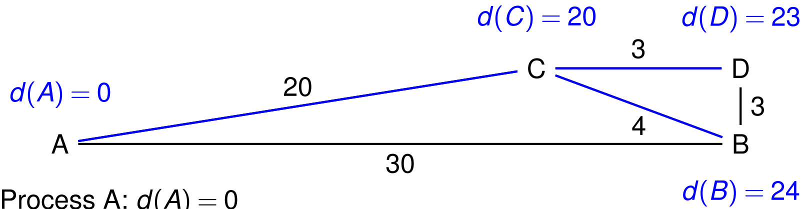
Process C.

For Edge (C,D): Set $d(D) = d(C) + 3 = 23$ and alarm for 23.

For Edge (C, B): Reset $d(B) = 24$ and alarm for 24.

At time 23:

Alarms.



Process A: $d(A) = 0$

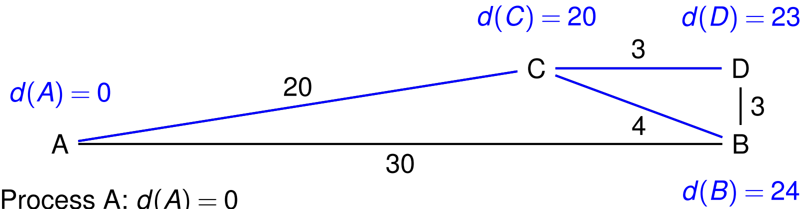
For Edge (A, C): Set $d(C) = d(A) + 20 = 20$ and alarm for 20.
For Edge (A, B): Set $d(B) = d(A) + 30 = 30$ and alarm for 30.

At time 20:
Process C.

For Edge (C,D): Set $d(D) = d(C) + 3 = 23$ and alarm for 23.
For Edge (C, B): Reset $d(B) = 24$ and alarm for 24.

At time 23:
Process D.

Alarms.



Process A: $d(A) = 0$

For Edge (A, C): Set $d(C) = d(A) + 20 = 20$ and alarm for 20.

For Edge (A, B): Set $d(B) = d(A) + 30 = 30$ and alarm for 30.

At time 20:

Process C.

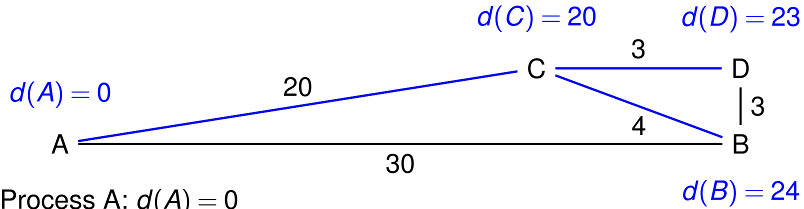
For Edge (C,D): Set $d(D) = d(C) + 3 = 23$ and alarm for 23.

For Edge (C, B): Reset $d(B) = 24$ and alarm for 24.

At time 23:

Process D. Set $d(D) = 23$.

Alarms.



Process A: $d(A) = 0$

For Edge (A, C): Set $d(C) = d(A) + 20 = 20$ and alarm for 20.

For Edge (A, B): Set $d(B) = d(A) + 30 = 30$ and alarm for 30.

At time 20:

Process C.

For Edge (C,D): Set $d(D) = d(C) + 3 = 23$ and alarm for 23.

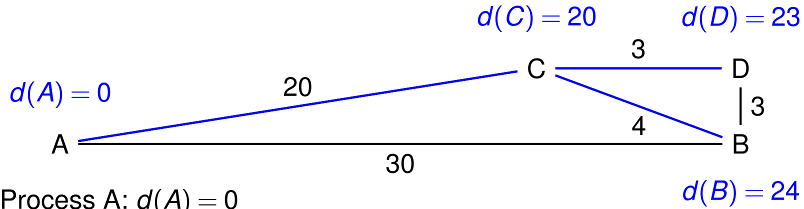
For Edge (C, B): Reset $d(B) = 24$ and alarm for 24.

At time 23:

Process D. Set $d(D) = 23$.

For Edge (D, B):

Alarms.



Process A: $d(A) = 0$

For Edge (A, C): Set $d(C) = d(A) + 20 = 20$ and alarm for 20.

For Edge (A, B): Set $d(B) = d(A) + 30 = 30$ and alarm for 30.

At time 20:

Process C.

For Edge (C,D): Set $d(D) = d(C) + 3 = 23$ and alarm for 23.

For Edge (C, B): Reset $d(B) = 24$ and alarm for 24.

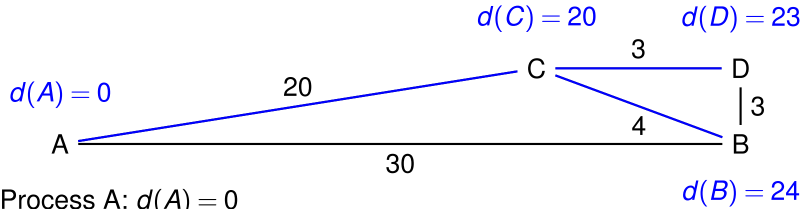
At time 23:

Process D. Set $d(D) = 23$.

For Edge (D, B):

$d(D) = 24$ which is less than $d(D) + 3 = 26$ so leave it.

Alarms.



Process A: $d(A) = 0$

For Edge (A, C): Set $d(C) = d(A) + 20 = 20$ and alarm for 20.
For Edge (A, B): Set $d(B) = d(A) + 30 = 30$ and alarm for 30.

At time 20:
Process C.

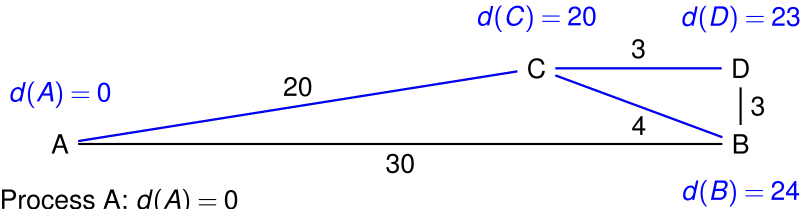
For Edge (C,D): Set $d(D) = d(C) + 3 = 23$ and alarm for 23.
For Edge (C, B): Reset $d(B) = 24$ and alarm for 24.

At time 23:
Process D. Set $d(D) = 23$.

For Edge (D, B):
$d(D) = 24$ which is less than $d(D) + 3 = 26$ so leave it.

At time 24: Process B.

Alarms.



Process A: $d(A) = 0$

For Edge (A,C): Set $d(C) = d(A) + 20 = 20$ and alarm for 20.

For Edge (A,B): Set $d(B) = d(A) + 30 = 30$ and alarm for 30.

At time 20:

Process C.

For Edge (C,D): Set $d(D) = d(C) + 3 = 23$ and alarm for 23.

For Edge (C,B): Reset $d(B) = 24$ and alarm for 24.

At time 23:

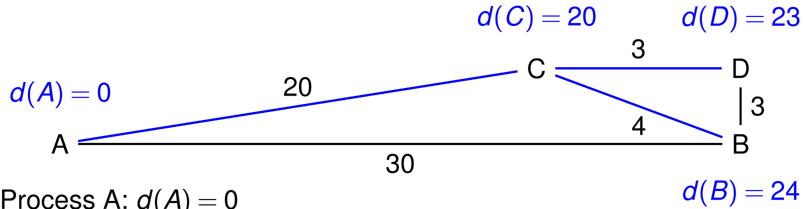
Process D. Set $d(D) = 23$.

For Edge (D,B):

$d(D) = 24$ which is less than $d(D) + 3 = 26$ so leave it.

At time 24: Process B.

Alarms.



Process A: $d(A) = 0$

For Edge (A,C): Set $d(C) = d(A) + 20 = 20$ and alarm for 20.
For Edge (A,B): Set $d(B) = d(A) + 30 = 30$ and alarm for 30.

At time 20:
Process C.

For Edge (C,D): Set $d(D) = d(C) + 3 = 23$ and alarm for 23.
For Edge (C,B): Reset $d(B) = 24$ and alarm for 24.

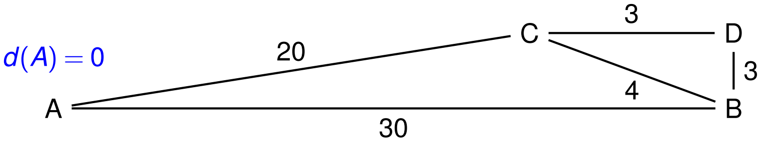
At time 23:
Process D. Set $d(D) = 23$.

For Edge (D,B):
$d(D) = 24$ which is less than $d(D) + 3 = 26$ so leave it.

At time 24: Process B. Done.

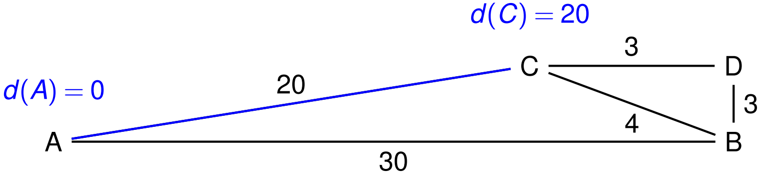
Hmmm...

..what needed to be done?



Hmmm...

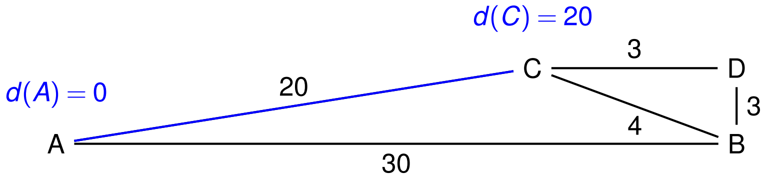
..what needed to be done?



Set a distance.

Hmmm...

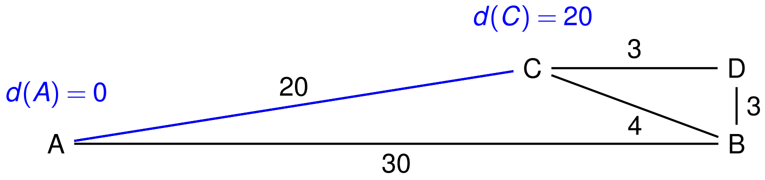
..what needed to be done?



Set a distance. Set an alarm.

Hmmm...

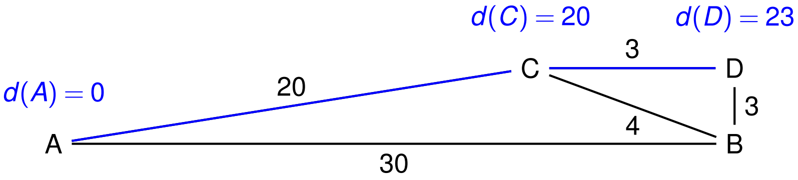
..what needed to be done?



Set a distance. Set an alarm.
Find next alarm.

Hmmm...

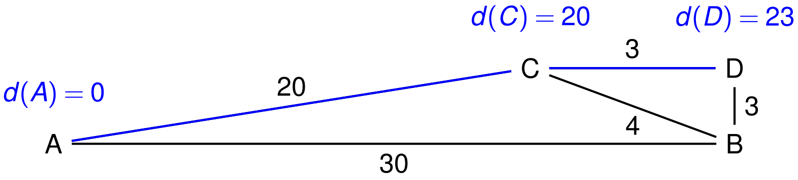
..what needed to be done?



Set a distance. Set an alarm.
Find next alarm. Set another distance.

Hmmm...

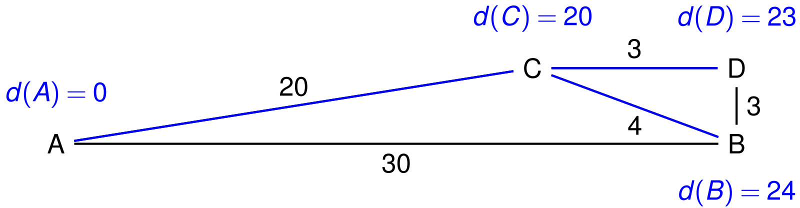
..what needed to be done?



Set a distance. Set an alarm.
Find next alarm. Set another distance. And set alarm.

Hmmm...

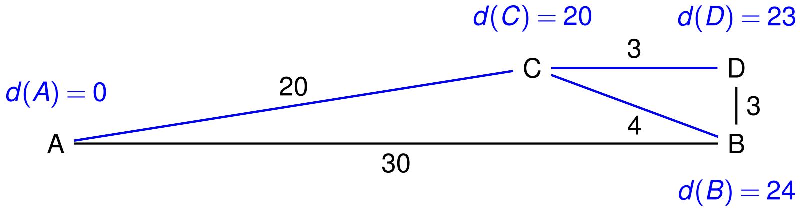
..what needed to be done?



Set a distance. Set an alarm.
Find next alarm. Set another distance. And set alarm. Reset distance.

Hmmm...

..what needed to be done?



Set a distance. Set an alarm.
Find next alarm. Set another distance. And set alarm. Reset distance. Reset Alarm.

Alarm Algorithm.

Set an alarm clock for node s at time 0.

Alarm Algorithm.

Set an alarm clock for node s at time 0.
Repeat until there are no more alarms:

Alarm Algorithm.

Set an alarm clock for node s at time 0.
Repeat until there are no more alarms:
Next alarm goes off at time T, for node u. Then:

Alarm Algorithm.

Set an alarm clock for node s at time 0.
Repeat until there are no more alarms:
Next alarm goes off at time T, for node u. Then:
– The distance from s to u is T.

Alarm Algorithm.

Set an alarm clock for node s at time 0.
Repeat until there are no more alarms:
Next alarm goes off at time T, for node u. Then:

- The distance from s to u is T.
- For each neighbor v of u in G:

Alarm Algorithm.

Set an alarm clock for node s at time 0.
Repeat until there are no more alarms:
Next alarm goes off at time T, for node u. Then:

- The distance from s to u is T.
- For each neighbor v of u in G:
  - If no alarm for v, set alarm for $T + l(u, v)$.

Alarm Algorithm.

Set an alarm clock for node s at time 0.
Repeat until there are no more alarms:
Next alarm goes off at time T, for node *u*. Then:

- The distance from *s* to *u* is T.
- For each neighbor *v* of *u* in G:
  - If no alarm for *v*, set alarm for $T + l(u, v)$.
  - If *v*'s alarm is $\geq T + l(u, v)$

Alarm Algorithm.

Set an alarm clock for node s at time 0.
Repeat until there are no more alarms:
Next alarm goes off at time T, for node u. Then:

- The distance from s to u is T.
- For each neighbor v of u in G:
  * If no alarm for v, set alarm for $T + l(u, v)$.
  * If v's alarm is $\geq T + l(u, v)$
    then reset it $T + l(u, v)$.

Alarm Algorithm.

Set an alarm clock for node s at time 0.
Repeat until there are no more alarms:
Next alarm goes off at time T, for node u. Then:

- The distance from s to u is T.
- For each neighbor v of u in G:
  * If no alarm for v, set alarm for $T + l(u, v)$.
  * If v's alarm is $\geq T + l(u, v)$
    then reset it $T + l(u, v)$.

Implementation:

Alarm Algorithm.

Set an alarm clock for node s at time 0.
Repeat until there are no more alarms:
Next alarm goes off at time T, for node u. Then:

- The distance from s to u is T.
- For each neighbor v of u in G:
  - If no alarm for v, set alarm for $T + l(u, v)$.
  - If v's alarm is $\geq T + l(u, v)$ then reset it $T + l(u, v)$.

Implementation:
Need to maintain alarm for each node.

Alarm Algorithm.

Set an alarm clock for node s at time 0.
Repeat until there are no more alarms:
Next alarm goes off at time T, for node u. Then:

- The distance from s to u is T.
- For each neighbor v of u in G:
  * If no alarm for v, set alarm for $T + l(u, v)$.
  * If v's alarm is $\geq T + l(u, v)$
    then reset it $T + l(u, v)$.

Implementation:
Need to maintain alarm for each node.
Possibly need to decrease alarm for a node.
Find next alarm time.

Alarm Algorithm.

Set an alarm clock for node s at time 0.
Repeat until there are no more alarms:
Next alarm goes off at time T, for node u. Then:

- The distance from s to u is T.
- For each neighbor v of u in G:
  - If no alarm for v, set alarm for $T + l(u, v)$.
  - If v's alarm is $\geq T + l(u, v)$
    then reset it $T + l(u, v)$.

Implementation:
Need to maintain alarm for each node.
Possibly need to decrease alarm for a node.
Find next alarm time.

Insert: (v, key)

Alarm Algorithm.

Set an alarm clock for node s at time 0.
Repeat until there are no more alarms:
Next alarm goes off at time T, for node u. Then:

- The distance from s to u is T.
- For each neighbor v of u in G:
  * If no alarm for v, set alarm for $T + l(u, v)$.
  * If v's alarm is $\geq T + l(u, v)$ then reset it $T + l(u, v)$.

Implementation:
Need to maintain alarm for each node.
Possibly need to decrease alarm for a node.
Find next alarm time.

Insert: (v, key)
DecreaseKey: $(v, newkey)$

Alarm Algorithm.

Set an alarm clock for node s at time 0.
Repeat until there are no more alarms:
Next alarm goes off at time T, for node u. Then:

- The distance from s to u is T.
- For each neighbor v of u in G:
  - If no alarm for v, set alarm for $T + l(u, v)$.
  - If v's alarm is $\geq T + l(u, v)$
    then reset it $T + l(u, v)$.

Implementation:
Need to maintain alarm for each node.
Possibly need to decrease alarm for a node.
Find next alarm time.

Insert: (v, key)
DecreaseKey: $(v, newkey)$
DeleteMin: Q **returns** v with min. key

Dijkstra's Algorithm.

foreach v: $d(v) = \infty$.

Dijkstra's Algorithm.

foreach v: $d(v) = \infty$.
$d(s) = 0$.

Dijkstra's Algorithm.

foreach v: $d(v) = \infty$.
$d(s) = 0$.
Q.Insert(s,0)

Dijkstra's Algorithm.

foreach v: $d(v) = \infty$.
$d(s) = 0$.
Q.Insert(s,0)
While u = Q.DeleteMin():
  foreach edge (u, v):

Dijkstra's Algorithm.

foreach v: $d(v) = \infty$.
$d(s) = 0$.
Q.Insert(s,0)
While u = Q.DeleteMin():
   foreach edge (u,v):
      if $d(v) > d(u) + l(u,v)$:
         $d(v) = d(u) + l(u,v)$
         Q.InsertOrDecreaseKey(v,d(v))

Dijkstra's Algorithm.

foreach v: $d(v) = \infty$.
$d(s) = 0$.
Q.Insert(s,0)
While u = Q.DeleteMin():
  foreach edge (u, v):
    if $d(v) > d(u) + l(u, v)$:
      $d(v) = d(u) + l(u, v)$
      Q.InsertOrDecreaseKey(v,d(v))

Runtime:

Dijkstra's Algorithm.

foreach v: $d(v) = \infty$.
$d(s) = 0$.
Q.Insert(s,0)
While u = Q.DeleteMin():
  foreach edge (u, v):
    if $d(v) > d(u) + l(u, v)$:
      $d(v) = d(u) + l(u, v)$
      Q.InsertOrDecreaseKey(v,d(v))

Runtime:
$|V|$ DeleteMins.

Dijkstra's Algorithm.

foreach v: $d(v) = \infty$.
$d(s) = 0$.
Q.Insert(s,0)
While u = Q.DeleteMin():
  foreach edge (u, v):
    if $d(v) > d(u) + l(u, v)$:
      $d(v) = d(u) + l(u, v)$
      Q.InsertOrDecreaseKey(v,d(v))

Runtime:
$|V|$ DeleteMins.
$|V|$ Inserts.

Dijkstra's Algorithm.

foreach v: $d(v) = \infty$.
$d(s) = 0$.
Q.Insert(s,0)
While u = Q.DeleteMin():
  foreach edge (u, v):
    if $d(v) > d(u) + l(u, v)$:
      $d(v) = d(u) + l(u, v)$
      Q.InsertOrDecreaseKey(v,d(v))

Runtime:
$|V|$ DeleteMins.
$|V|$ Inserts.
$\leq |E|$ DecreaseKeys.

Dijkstra's Algorithm.

foreach v: $d(v) = \infty$.
$d(s) = 0$.
Q.Insert(s,0)
While u = Q.DeleteMin():
  foreach edge (u, v):
    if $d(v) > d(u) + l(u, v)$:
      $d(v) = d(u) + l(u, v)$
      Q.InsertOrDecreaseKey(v,d(v))

Runtime:
$|V|$ DeleteMins.
$|V|$ Inserts.
$\leq |E|$ DecreaseKeys.

Dijkstra's Algorithm.

foreach v: $d(v) = \infty$.
$d(s) = 0$.
Q.Insert(s,0)
While u = Q.DeleteMin():
  foreach edge (u, v):
    if $d(v) > d(u) + l(u, v)$:
      $d(v) = d(u) + l(u, v)$
      Q.InsertOrDecreaseKey(v,d(v))

Runtime:
$|V|$ DeleteMins.
$|V|$ Inserts.
$\leq |E|$ DecreaseKeys.

Binary heap: $O((|V| + |E|) \log |V|)$

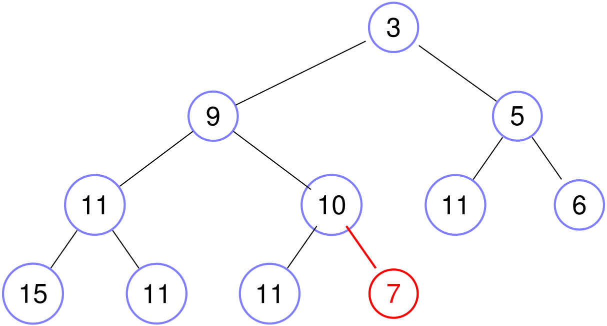
Binary Heap.

Heap[1]: bigger children.

[1] values only

Binary Heap.

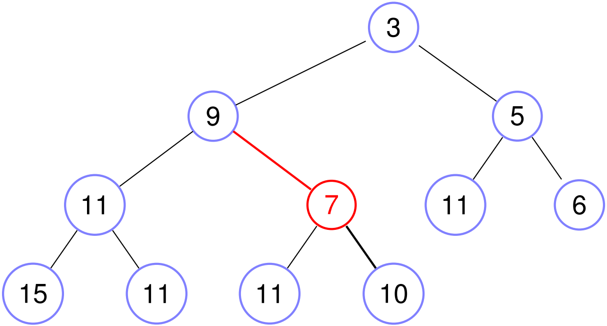
Heap[1]: bigger children.
$\Longrightarrow$ smallest at root.



[1] values only

Binary Heap.

Heap[1]: bigger children.
$\Longrightarrow$ smallest at root.

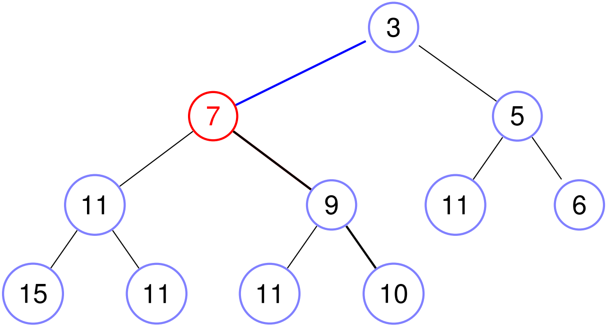


Insert(7):

[1]values only

Binary Heap.

Heap[1]: bigger children.
$\Longrightarrow$ smallest at root.

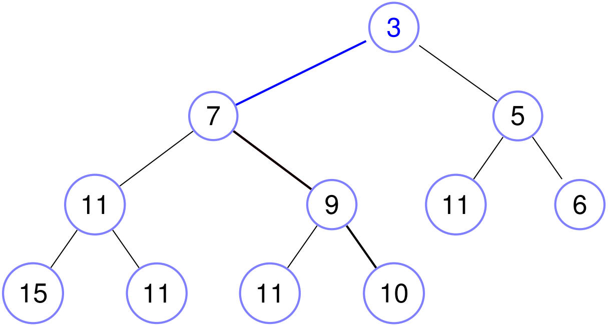


Insert(7): Bubble up: check parent.

[1] values only

Binary Heap.

Heap[1]: bigger children.
$\implies$ smallest at root.

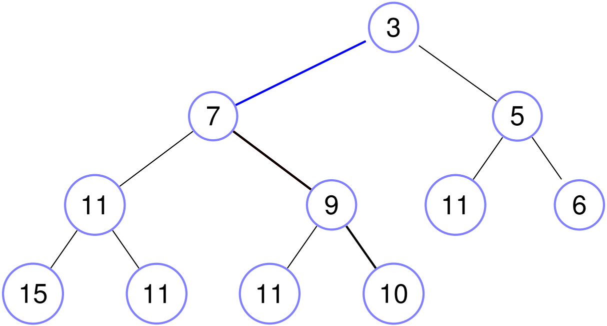


Insert(7): Bubble up: check parent. . **depth** comp.

[1] values only

Binary Heap.

Heap[1]: bigger children.
$\Longrightarrow$ smallest at root.

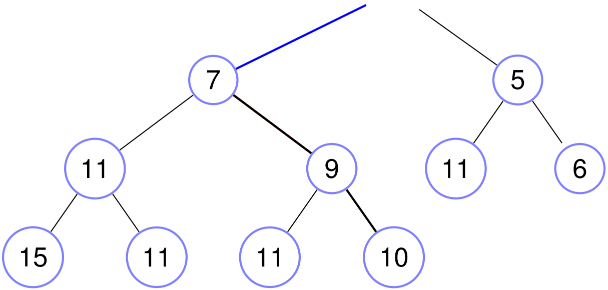


Insert(7): Bubble up: check parent. . **depth** comp.
DeleteMin:

[1] values only

Binary Heap.

Heap[1]: bigger children.
$\Longrightarrow$ smallest at root.

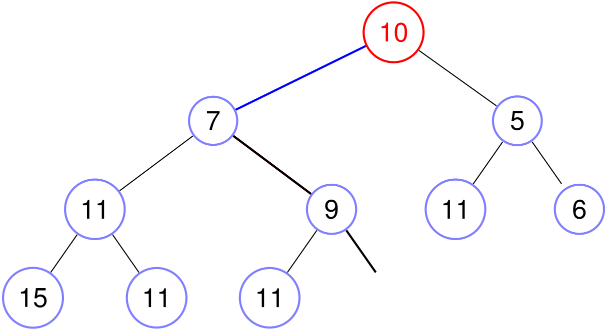


Insert(7): Bubble up: check parent. . **depth** comp.
DeleteMin:

[1]values only

Binary Heap.

Heap[1]: bigger children.
$\implies$ smallest at root.

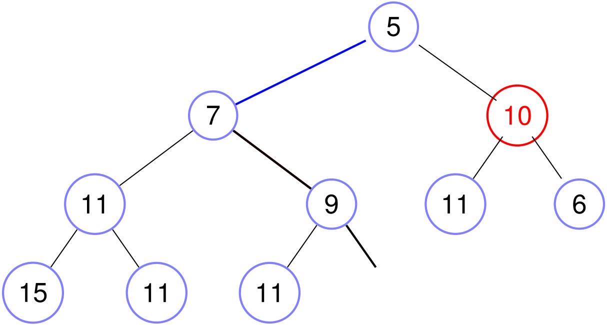


Insert(7): Bubble up: check parent. . **depth** comp.
DeleteMin: Replace.

[1] values only

Binary Heap.

Heap[1]: bigger children.
$\Longrightarrow$ smallest at root.

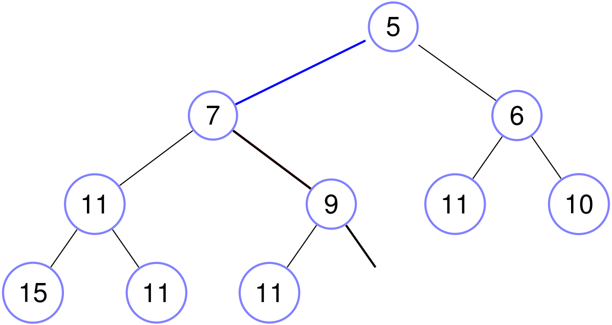


Insert(7): Bubble up: check parent. . **depth** comp.
DeleteMin: Replace. Bubble down: check **both** children..

[1] values only

Binary Heap.

Heap[1]: bigger children.
$\implies$ smallest at root.



Insert(7): Bubble up: check parent. . **depth** comp.
DeleteMin: Replace. Bubble down: check **both** children..
2$\times$ **depth** – comparisons.

[1]values only

d-ary heap

Degree – d, Depth – $\log_d n$.

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V| d \log n / \log d + |E| \log n / \log d)$.

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V| d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice:

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V| d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E|/|V|$ (average degree/2)

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V| d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E|/|V|$ (average degree/2)
$O(|E| \log n / \log d)$

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V| d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E|/|V|$ (average degree/2)
$O(|E| \log n / \log d)$

For dense graphs it approaches linear.

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V| d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E|/|V|$ (average degree/2)
$O(|E| \log n / \log d)$

For dense graphs it approaches linear.

Fibonacci Heaps:

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V| d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E|/|V|$ (average degree/2)
$O(|E| \log n / \log d)$

For dense graphs it approaches linear.

Fibonacci Heaps:
$O(\log n)$ per delete.

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V| d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E|/|V|$ (average degree/2)
$O(|E| \log n / \log d)$

For dense graphs it approaches linear.

Fibonacci Heaps:
$O(\log n)$ per delete.
$O(1)$ average decrease-key.

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V| d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E|/|V|$ (average degree/2)
$O(|E| \log n / \log d)$

For dense graphs it approaches linear.

Fibonacci Heaps:
$O(\log n)$ per delete.
$O(1)$ average decrease-key.

$O(|V| \log |V| + |E|)$.

d-ary heap

Degree – d, Depth – $\log_d n$.
Insert/DecreaseKey – $\log n / \log d$.
DeleteMin – $d \log n / \log d$. (Check all children.)

Dijkstra:
$O(|V|)$ deletemins. $O(d \log n / \log d)$ each.
$O(|E|)$ insert/decrease-keys. $O(\log n / \log d)$ each.

$O(|V| d \log n / \log d + |E| \log n / \log d)$.

Optimal Choice: Choose $d = |E| / |V|$ (average degree/2)
$O(|E| \log n / \log d)$

For dense graphs it approaches linear.

Fibonacci Heaps:
$O(\log n)$ per delete.
$O(1)$ average decrease-key.

$O(|V| \log |V| + |E|)$.
Linear for moderately dense graphs!

Lecture in a minute.

Lecture in a minute.

Breadth First Search of graph:
  Search with queue instead of stack.
  Get “distances” from source.
    Proof idea: queue has distance 0, then level 1, ...

Lecture in a minute.

Breadth First Search of graph:
  Search with queue instead of stack.
  Get "distances" from source.
    Proof idea: queue has distance 0, then level 1, ...

Djikstra: shortest paths in weighted graph:
  Replace weights by paths + BFS
  Implement using priority queue.
    Idea: ignores "new" nodes.
  Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Lecture in a minute.

Breadth First Search of graph:
  Search with queue instead of stack.
  Get “distances” from source.
    Proof idea: queue has distance 0, then level 1, ...

Djikstra: shortest paths in weighted graph:
  Replace weights by paths + BFS
  Implement using priority queue.
    Idea: ignores “new” nodes.
  Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
  Implementation: degree d tree.

Lecture in a minute.

Breadth First Search of graph:
  Search with queue instead of stack.
  Get "distances" from source.
    Proof idea: queue has distance 0, then level 1, ...

Djikstra: shortest paths in weighted graph:
  Replace weights by paths + BFS
  Implement using priority queue.
    Idea: ignores "new" nodes.
  Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
  Implementation: degree d tree.
    Heap Property: children larger than parent.

Lecture in a minute.

Breadth First Search of graph:
  Search with queue instead of stack.
  Get "distances" from source.
    Proof idea: queue has distance 0, then level 1, ...

Djikstra: shortest paths in weighted graph:
  Replace weights by paths + BFS
  Implement using priority queue.
    Idea: ignores "new" nodes.
  Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
  Implementation: degree d tree.
    Heap Property: children larger than parent.
      Minimum at top.

Lecture in a minute.

Breadth First Search of graph:
  Search with queue instead of stack.
  Get "distances" from source.
    Proof idea: queue has distance 0, then level 1, ...

Djikstra: shortest paths in weighted graph:
  Replace weights by paths + BFS
  Implement using priority queue.
    Idea: ignores "new" nodes.
  Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
  Implementation: degree d tree.
    Heap Property: children larger than parent.
      Minimum at top.
    Remove min: $O(d \log_d n)$ time: Replace min/percolate down.

Lecture in a minute.

Breadth First Search of graph:
  Search with queue instead of stack.
  Get “distances” from source.
    Proof idea: queue has distance 0, then level 1, ...

Djikstra: shortest paths in weighted graph:
  Replace weights by paths + BFS
  Implement using priority queue.
    Idea: ignores “new” nodes.
  Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
  Implementation: degree d tree.
    Heap Property: children larger than parent.
      Minimum at top.
    Remove min: $O(d \log_d n)$ time: Replace min/percolate down.
    Reduce Key: $O(\log_d n)$: percolate up.

Lecture in a minute.

Breadth First Search of graph:
  Search with queue instead of stack.
  Get "distances" from source.
    Proof idea: queue has distance 0, then level 1, ...

Djikstra: shortest paths in weighted graph:
  Replace weights by paths + BFS
  Implement using priority queue.
    Idea: ignores "new" nodes.
  Runtime: $|V|$ extracts, $|E|$ reduce-key for p-queue.

Priority Queue:
  Implementation: degree d tree.
    Heap Property: children larger than parent.
      Minimum at top.
    Remove min: $O(d \log_d n)$ time: Replace min/percolate down.
    Reduce Key: $O(\log_d n)$: percolate up.