DFS, BFS, and Dijkstra's Algorithm Demonstrations

Written by Melanie Baybay
University of San Francisco

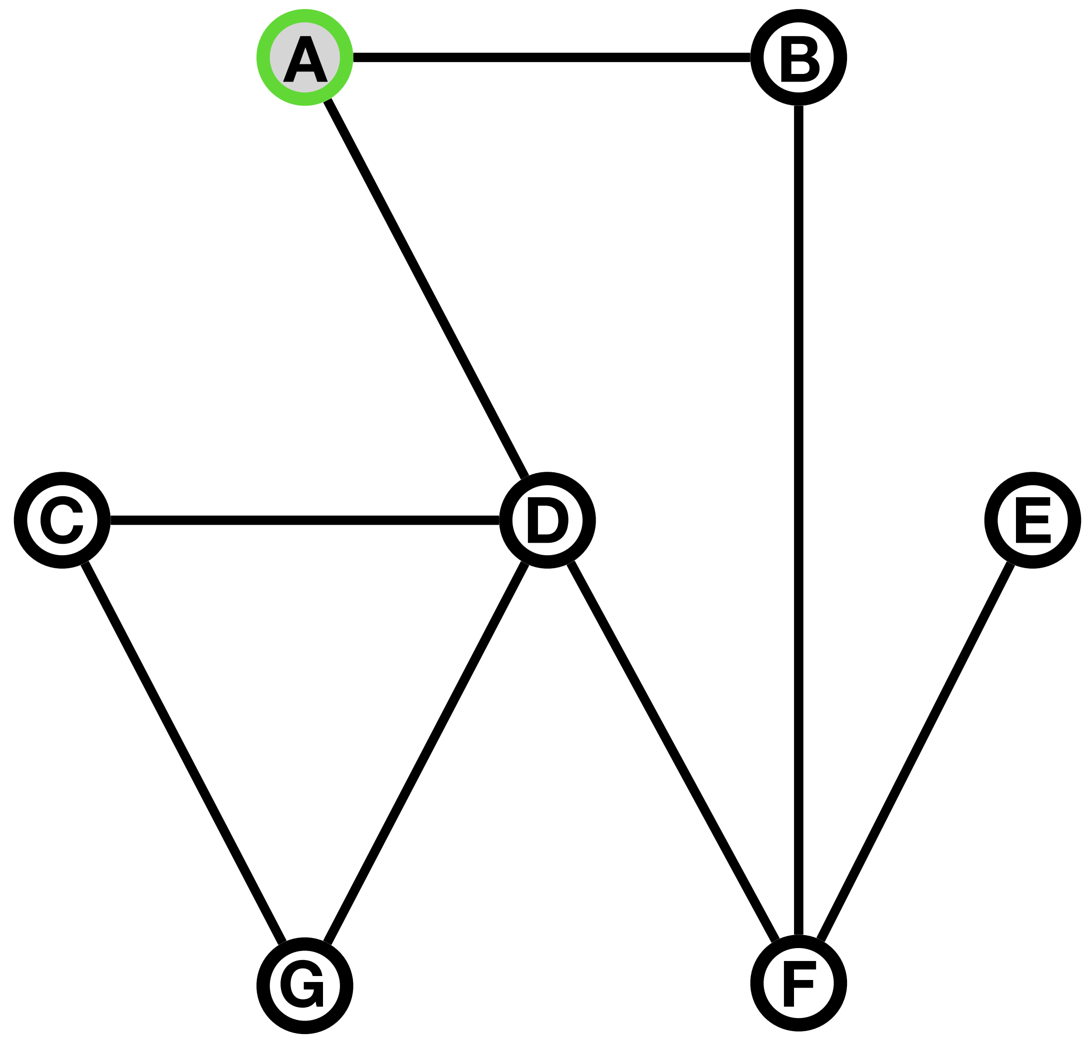
Depth First Search

explore nodes as *deep* as possible before looking *wide* (i.e. examine all descendants of a node before moving on to siblings)

- track visited nodes
- recursively visit neighbors until all nodes have been visited

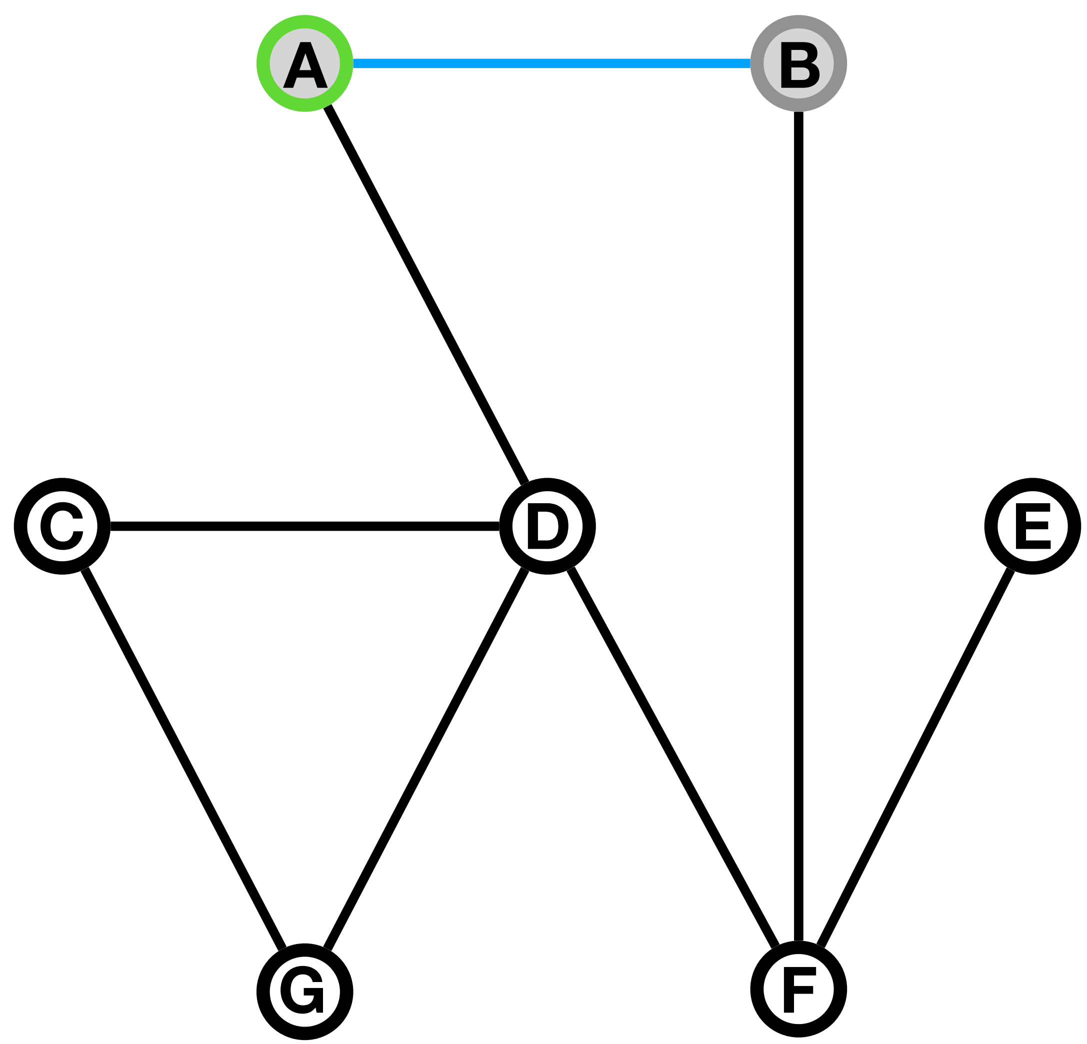
Depth First Search

explore nodes as *deep* as possible before looking *wide* (i.e. examine all descendants of a node before moving on to siblings)



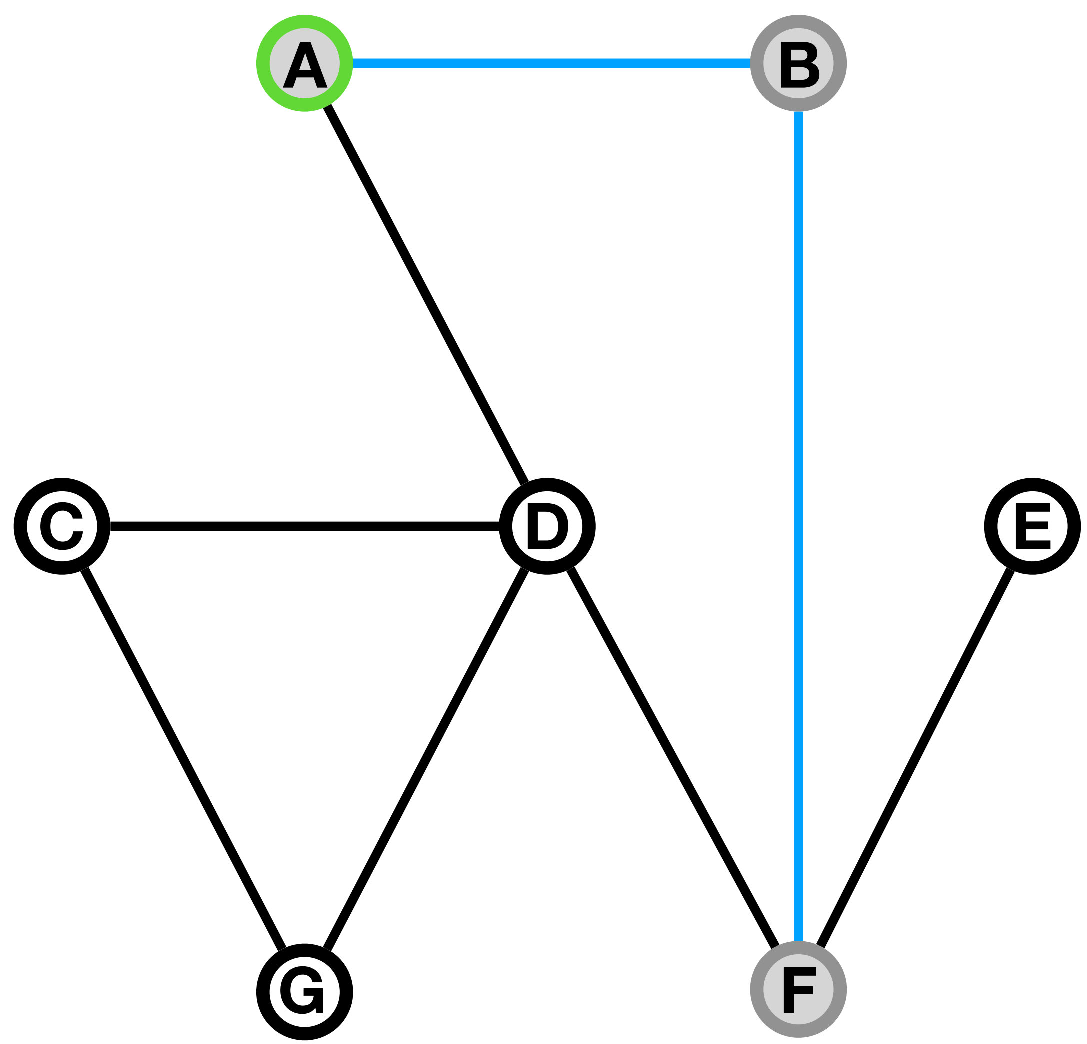
Depth First Search

explore nodes as *deep* as possible before looking *wide* (i.e. examine all descendants of a node before moving on to siblings)



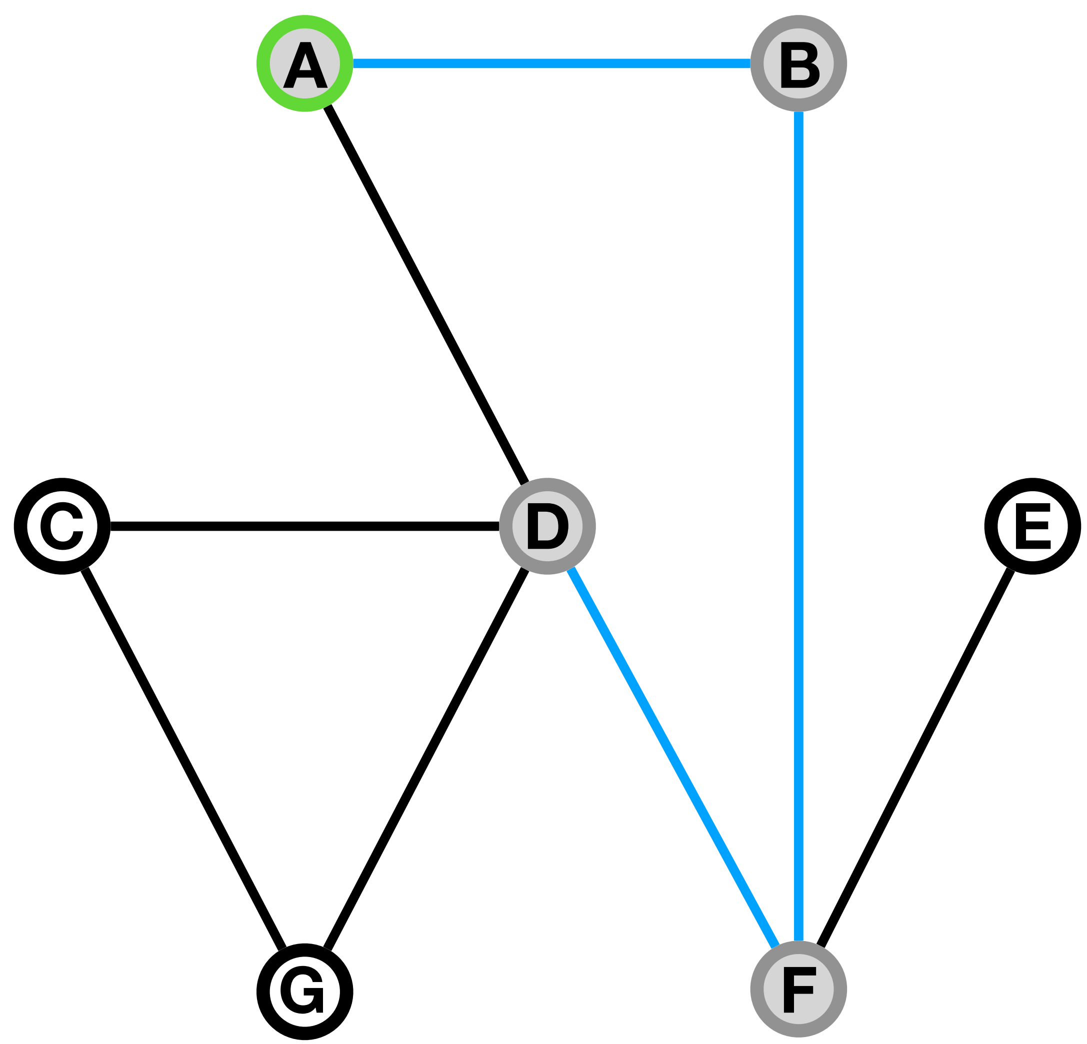
Depth First Search

explore nodes as *deep* as possible before looking *wide* (i.e. examine all descendants of a node before moving on to siblings)



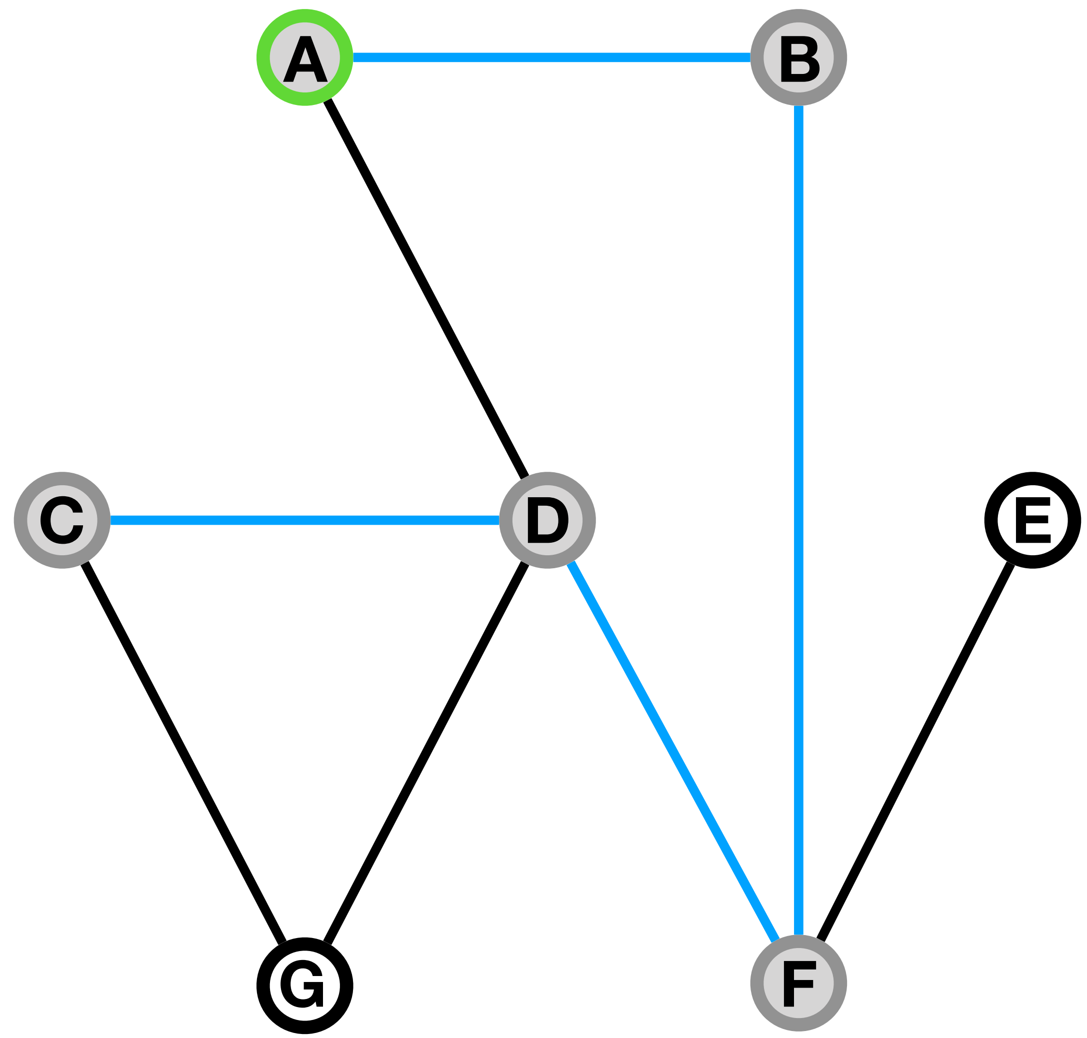
Depth First Search

explore nodes as *deep* as possible before looking *wide* (i.e. examine all descendants of a node before moving on to siblings)



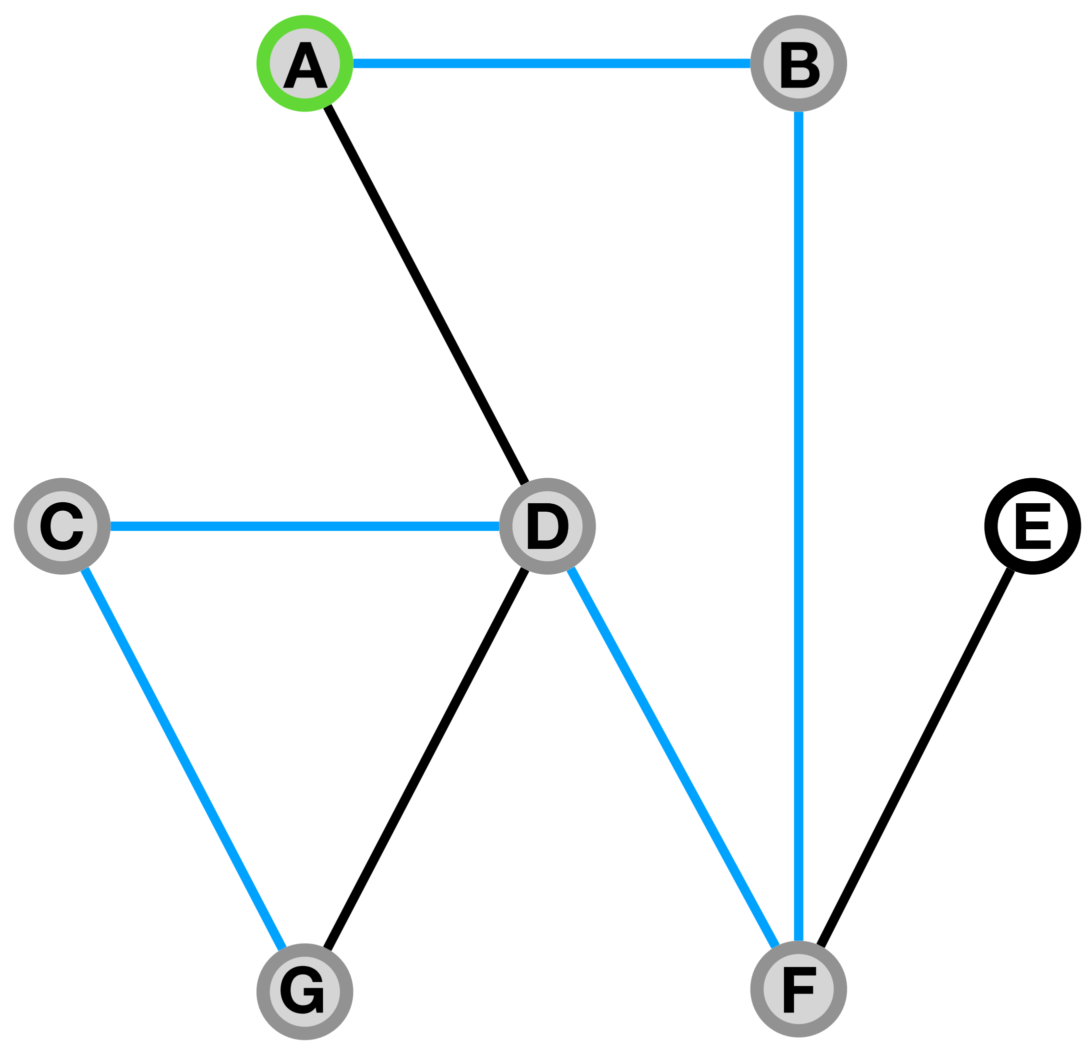
Depth First Search

explore nodes as *deep* as possible before looking *wide* (i.e. examine all descendants of a node before moving on to siblings)



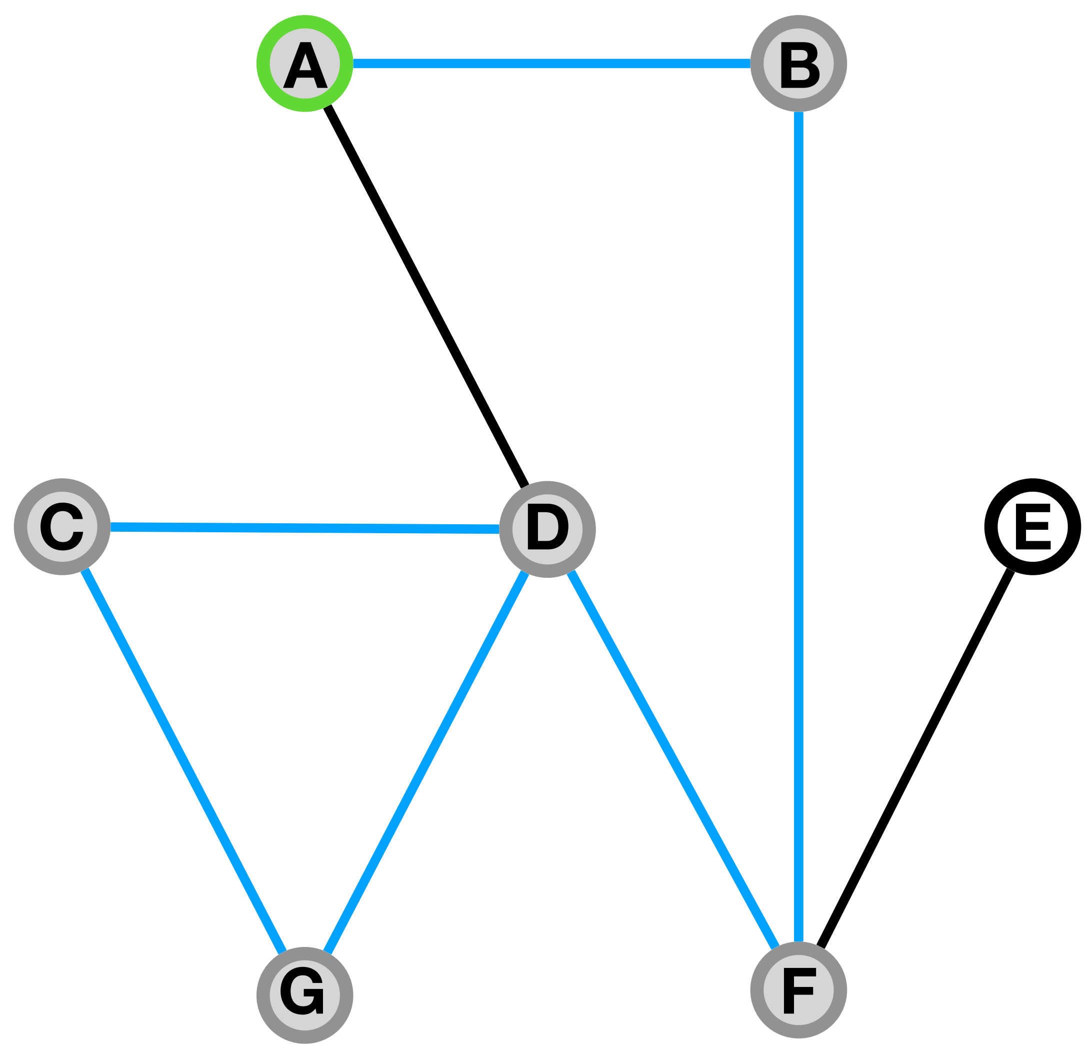
Depth First Search

explore nodes as *deep* as possible before looking *wide* (i.e. examine all descendants of a node before moving on to siblings)



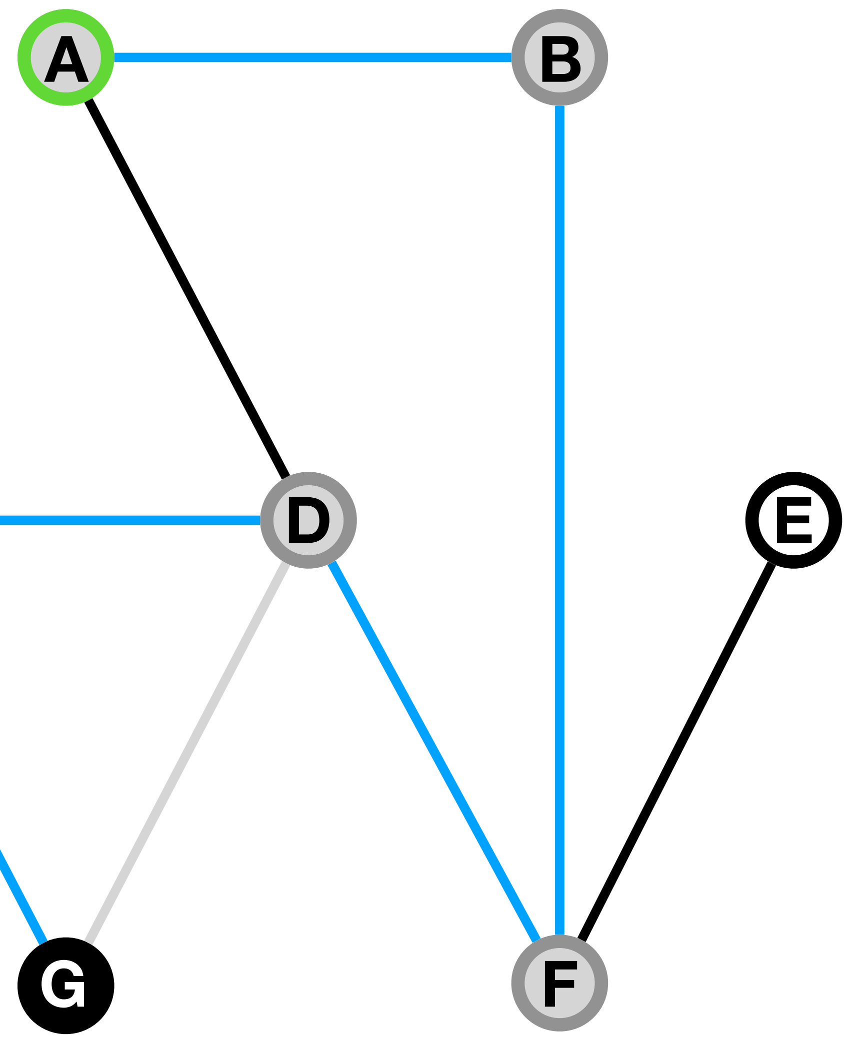
Depth First Search

explore nodes as *deep* as possible before looking *wide* (i.e. examine all descendants of a node before moving on to siblings)



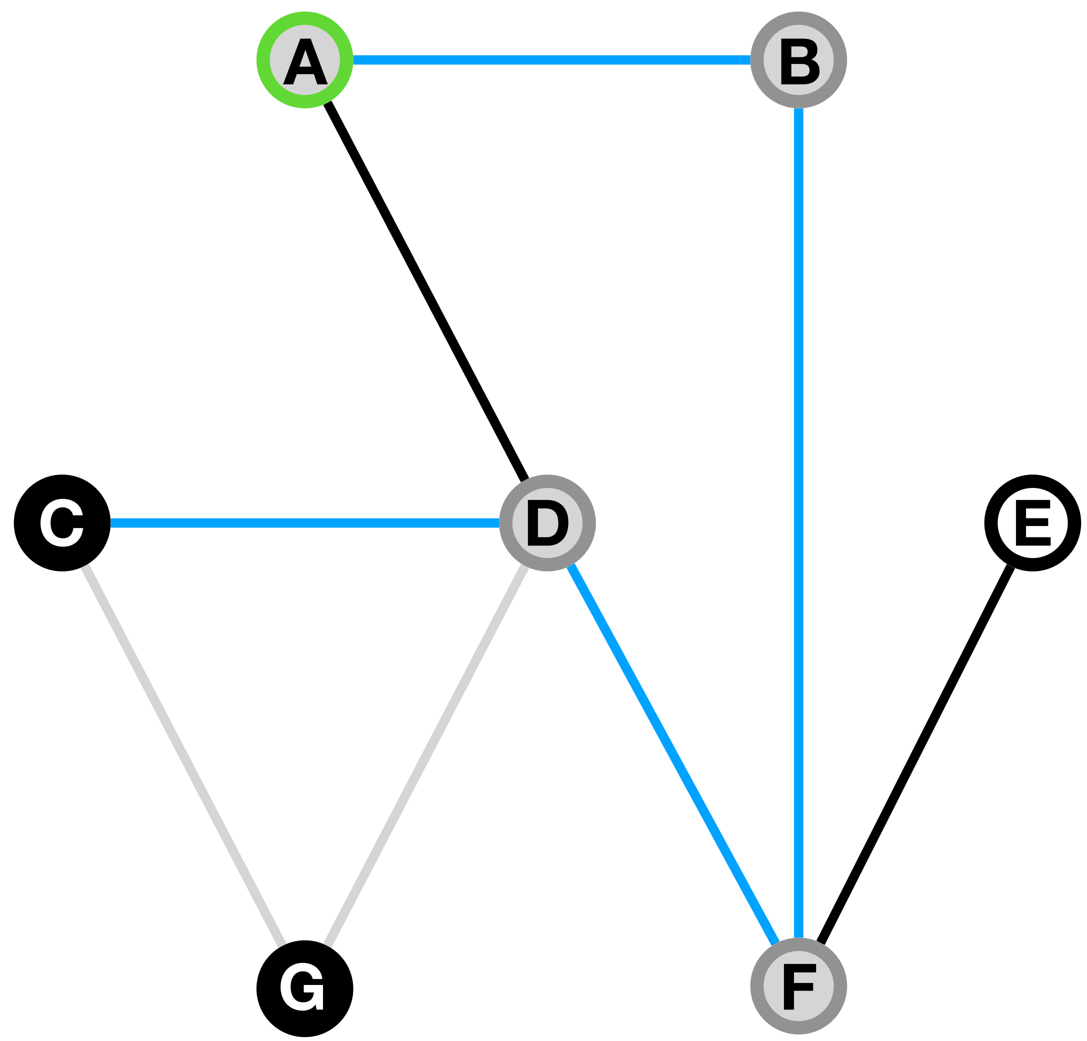
Depth First Search

explore nodes as *deep* as possible before looking *wide* (i.e. examine all descendants of a node before moving on to siblings)



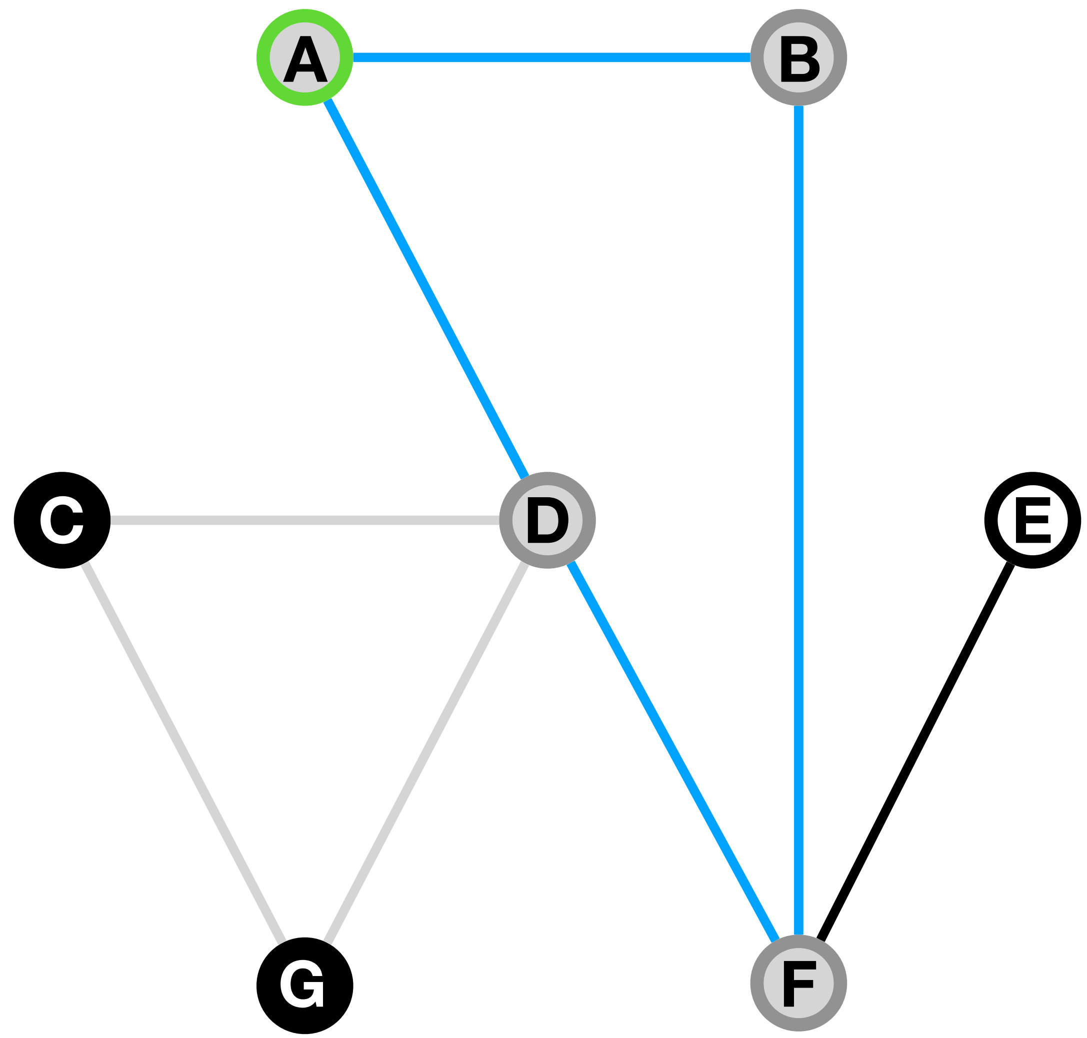
Depth First Search

explore nodes as *deep* as possible before looking *wide* (i.e. examine all descendants of a node before moving on to siblings)



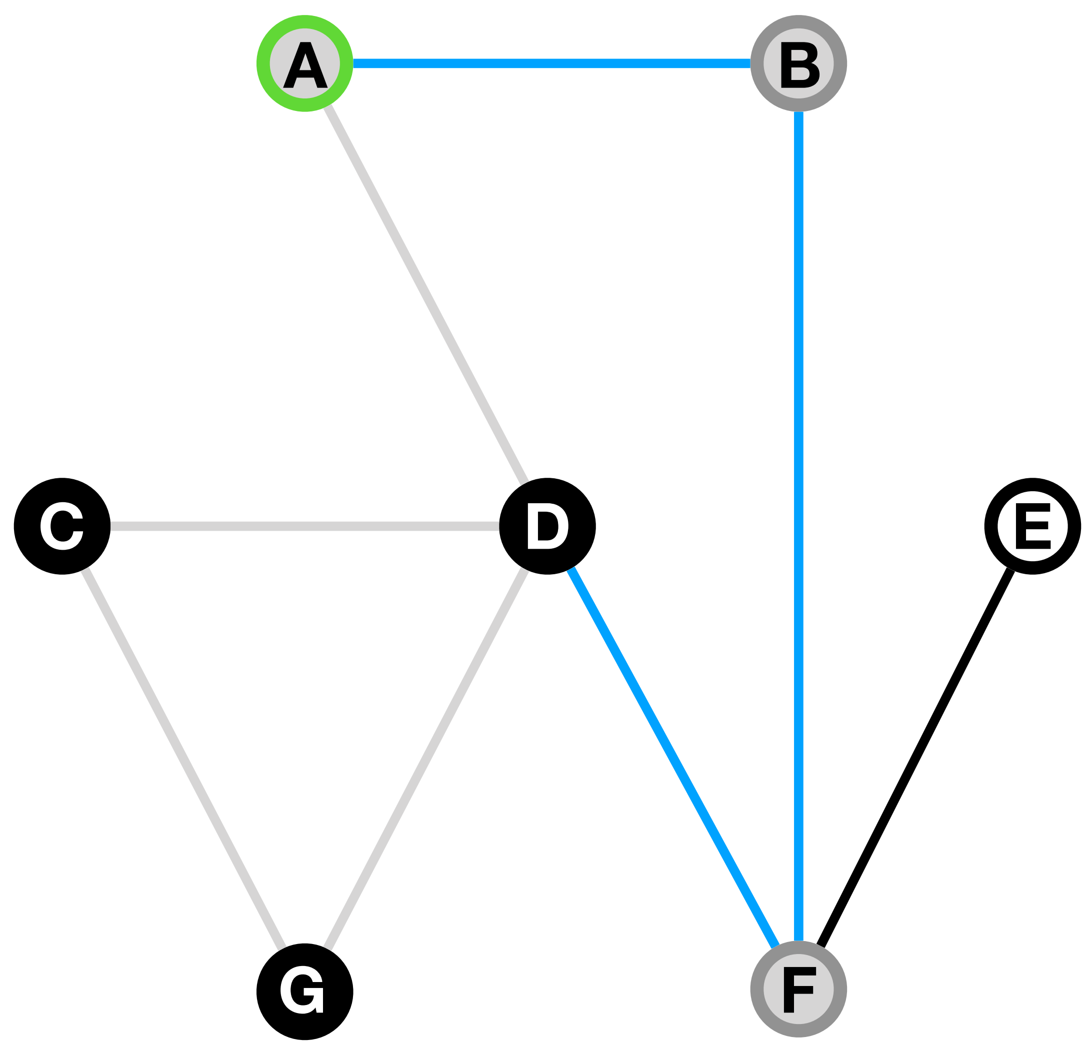
Depth First Search

explore nodes as *deep* as possible before looking *wide* (i.e. examine all descendants of a node before moving on to siblings)



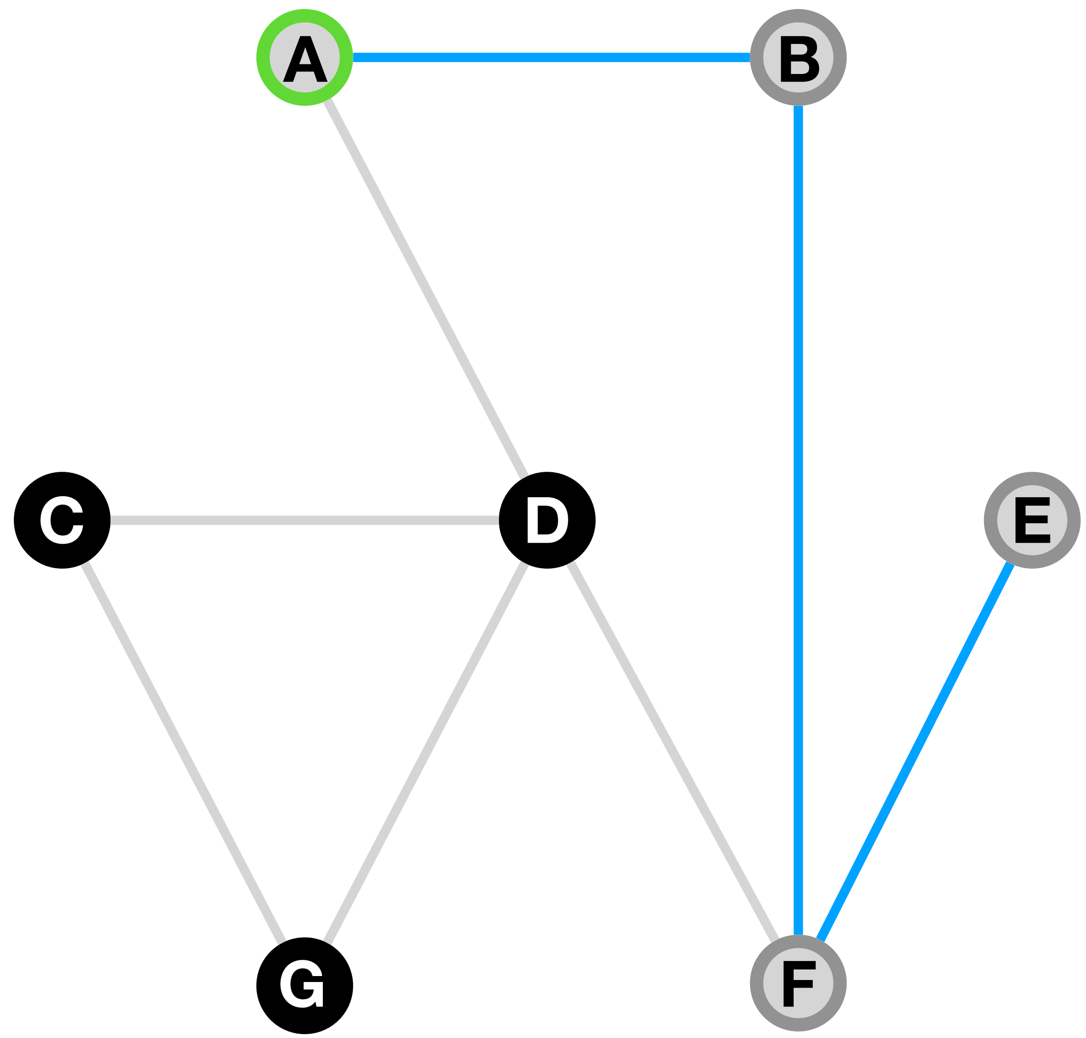
Depth First Search

explore nodes as *deep* as possible before looking *wide* (i.e. examine all descendants of a node before moving on to siblings)



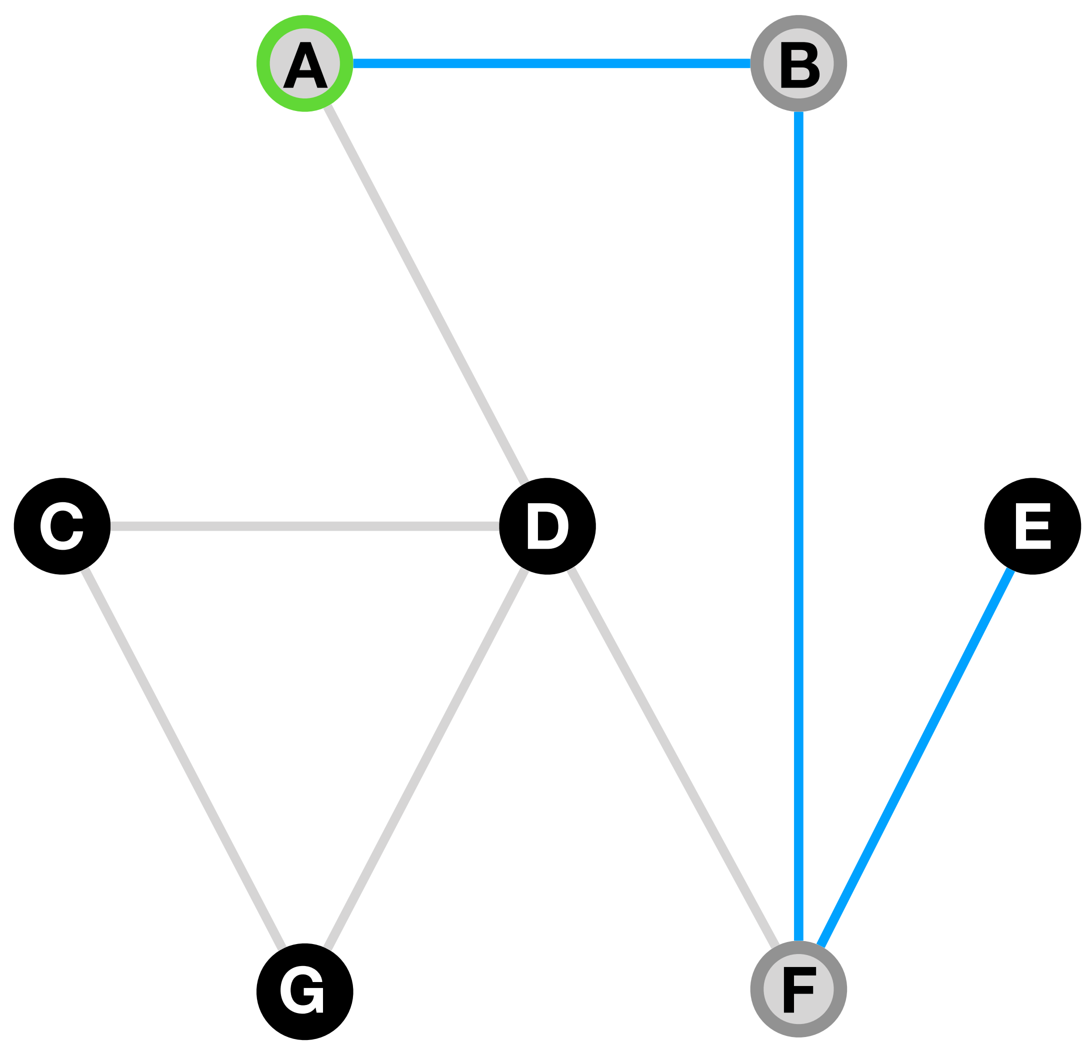
Depth First Search

explore nodes as *deep* as possible before looking *wide* (i.e. examine all descendants of a node before moving on to siblings)



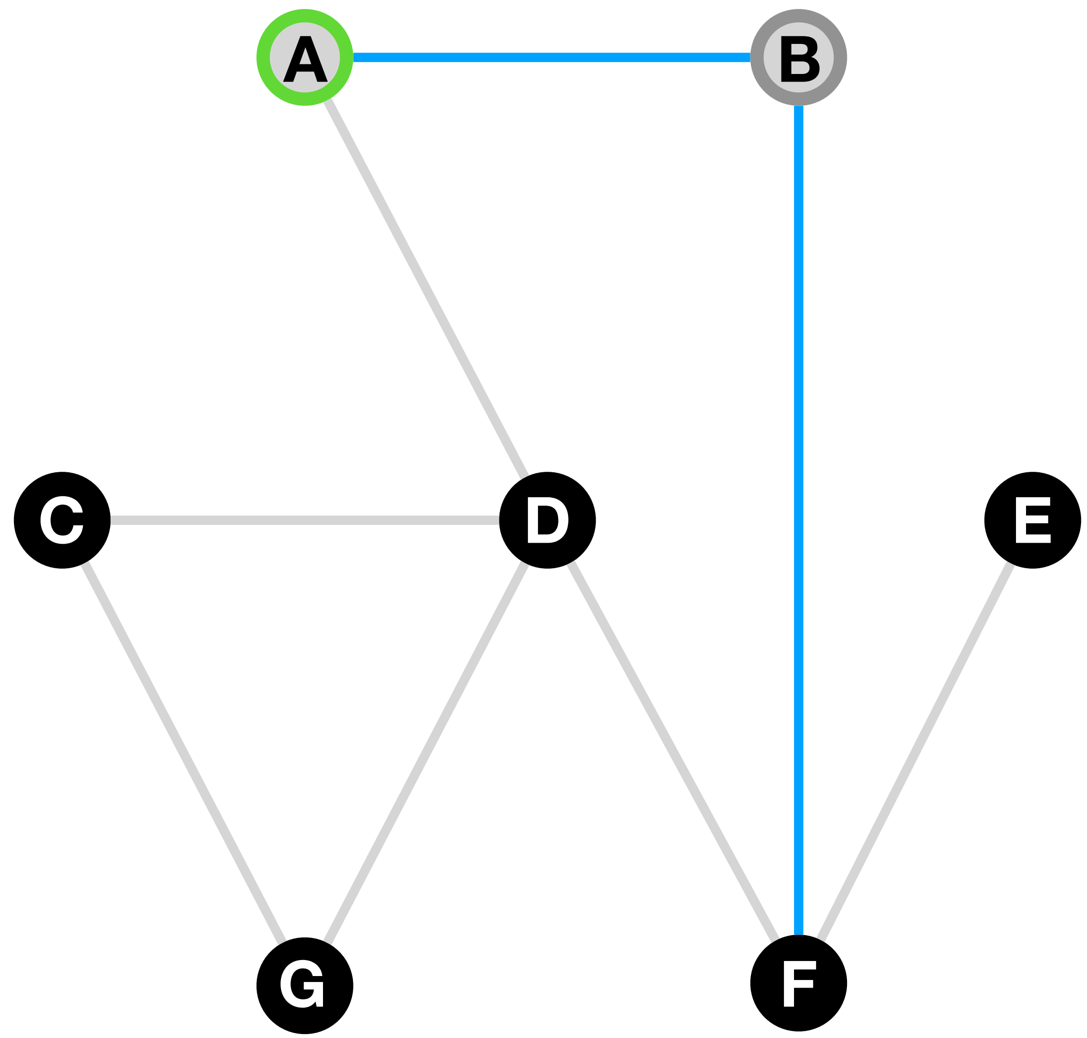
Depth First Search

explore nodes as *deep* as possible before looking *wide* (i.e. examine all descendants of a node before moving on to siblings)



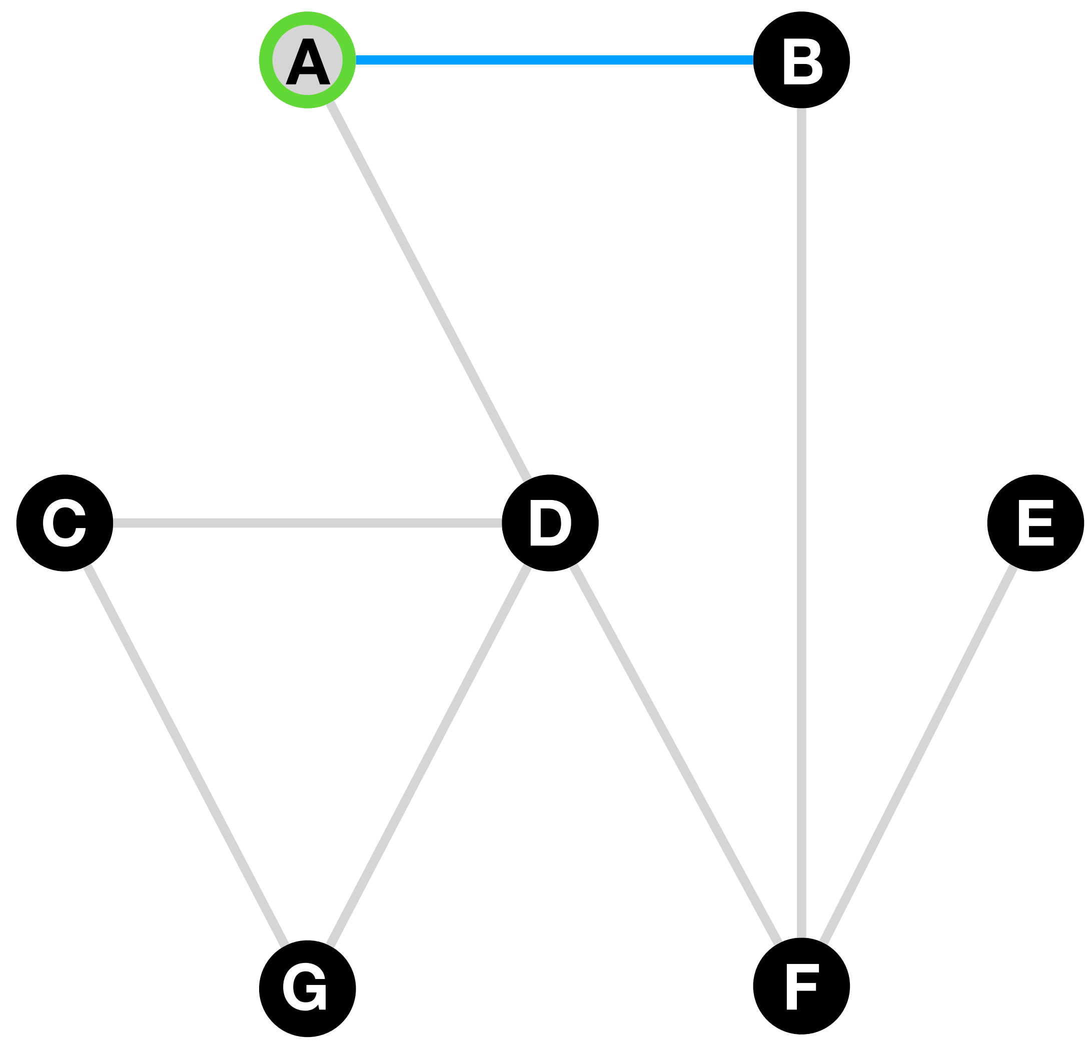
Depth First Search

explore nodes as *deep* as possible before looking *wide* (i.e. examine all descendants of a node before moving on to siblings)



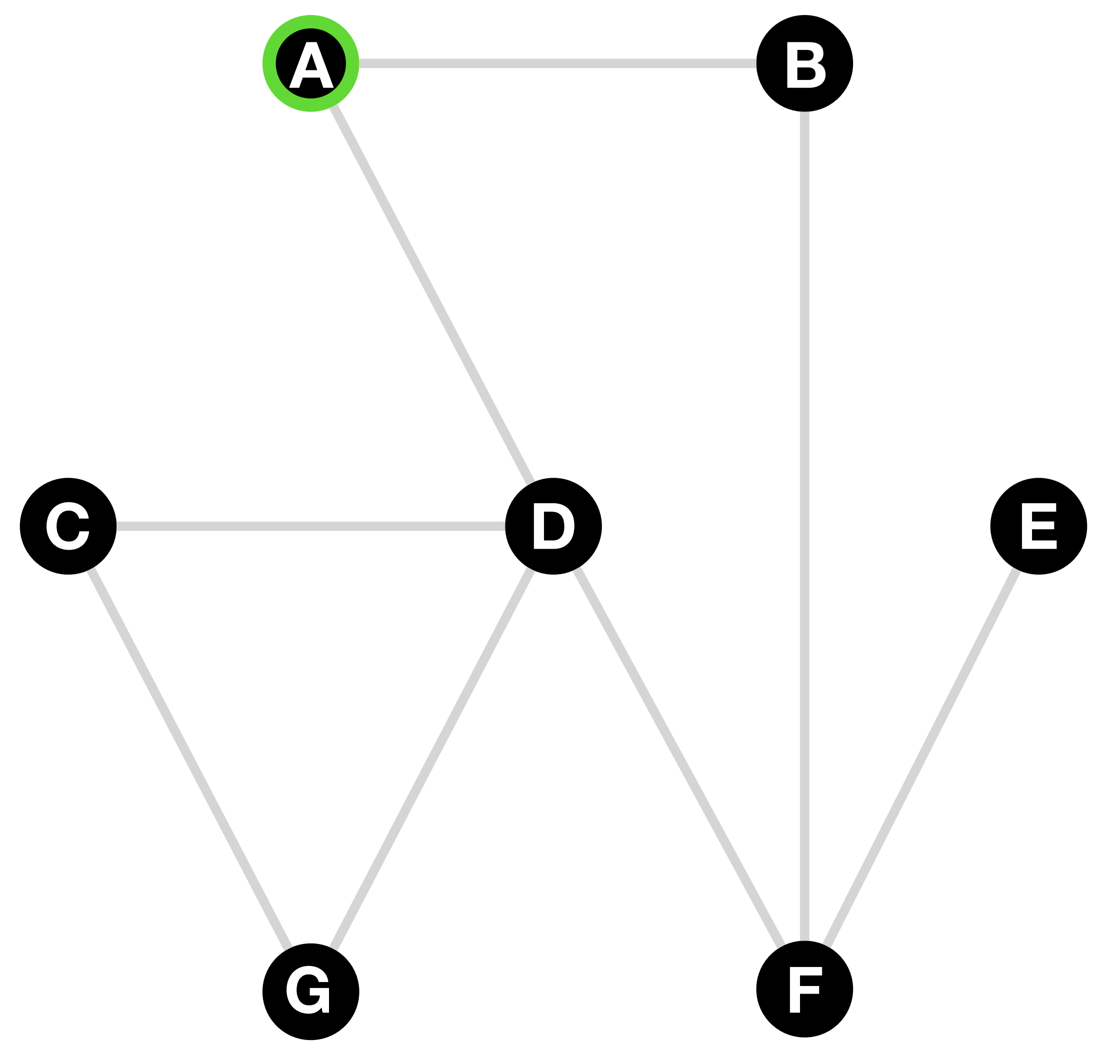
Depth First Search

explore nodes as *deep* as possible before looking *wide* (i.e. examine all descendants of a node before moving on to siblings)



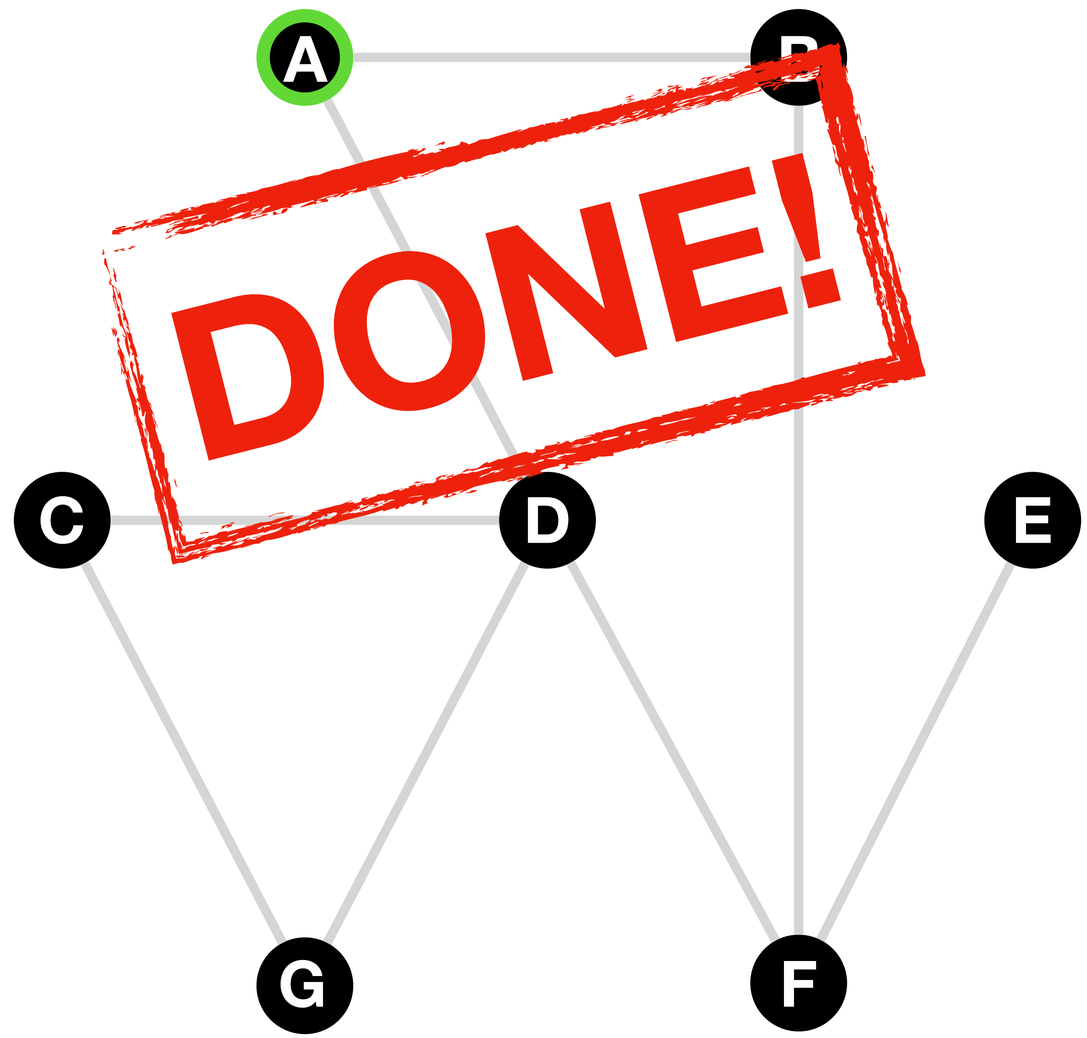
Depth First Search

explore nodes as *deep* as possible before looking *wide* (i.e. examine all descendants of a node before moving on to siblings)



Depth First Search

explore nodes as *deep* as possible before looking *wide* (i.e. examine all descendants of a node before moving on to siblings)



Depth First Search

```
DepthFirstSearch(G, vertex, visited[ ]) {
  visited[vertex] = True         # mark vertex as visited
  for each in G[vertex]:         # for each neighbor of vertex
    if ( !visited[neighbor] ) {  # if neighbor has not been visited
      DepthFirstSearch(G, neighbor, visited[ ]) # visit neighbor
    }
}
```

Complexity:

O(V + E). Each vertex and edge is visited at most once

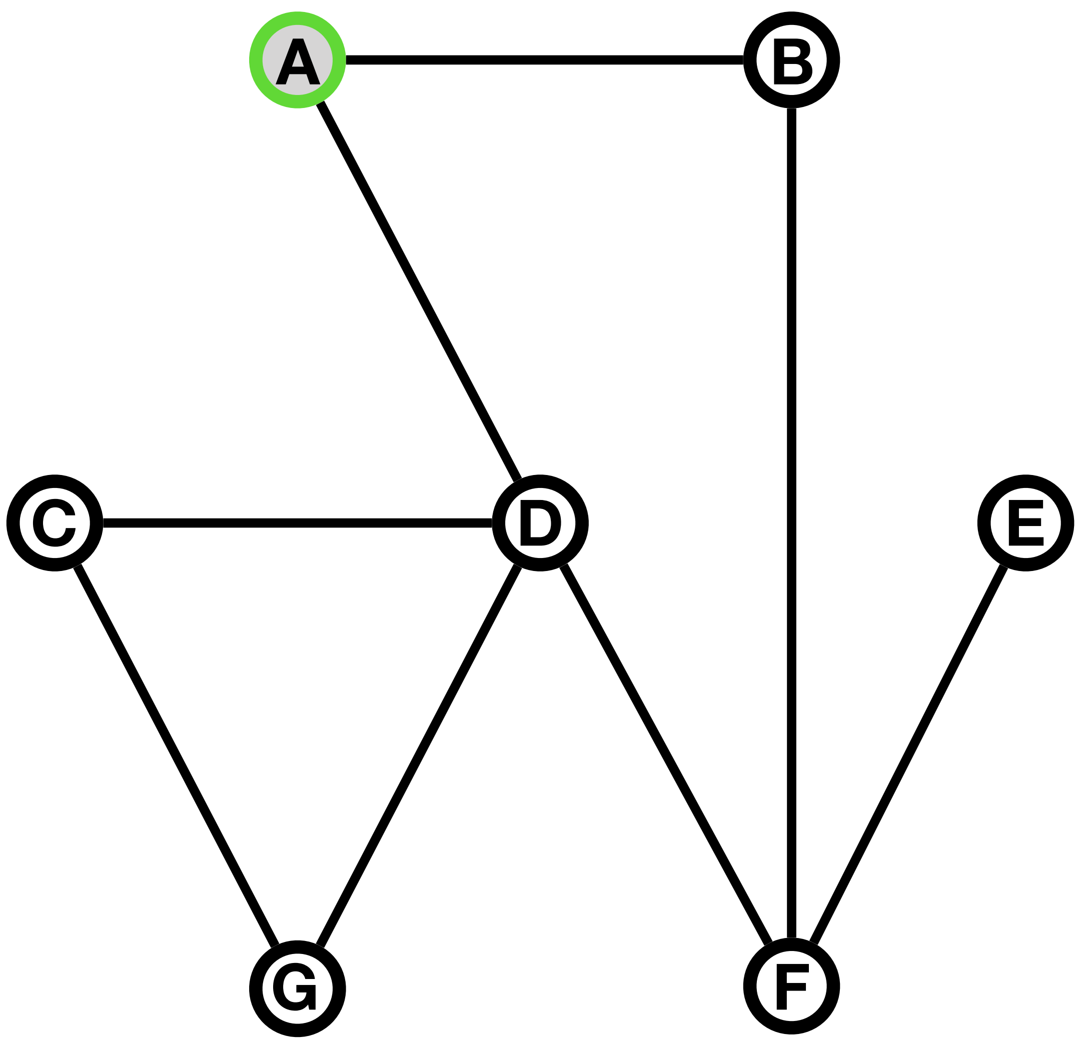
Breadth First Search

explore nodes as *wide* as possible before looking *deep*
(i.e. examine all siblings before descendants)

- keep a queue to track visited nodes
- when visiting a node, add all its *unvisited* neighbors to the queue
- until queue is empty: remove node from top of queue and visit it.

Breadth First Search

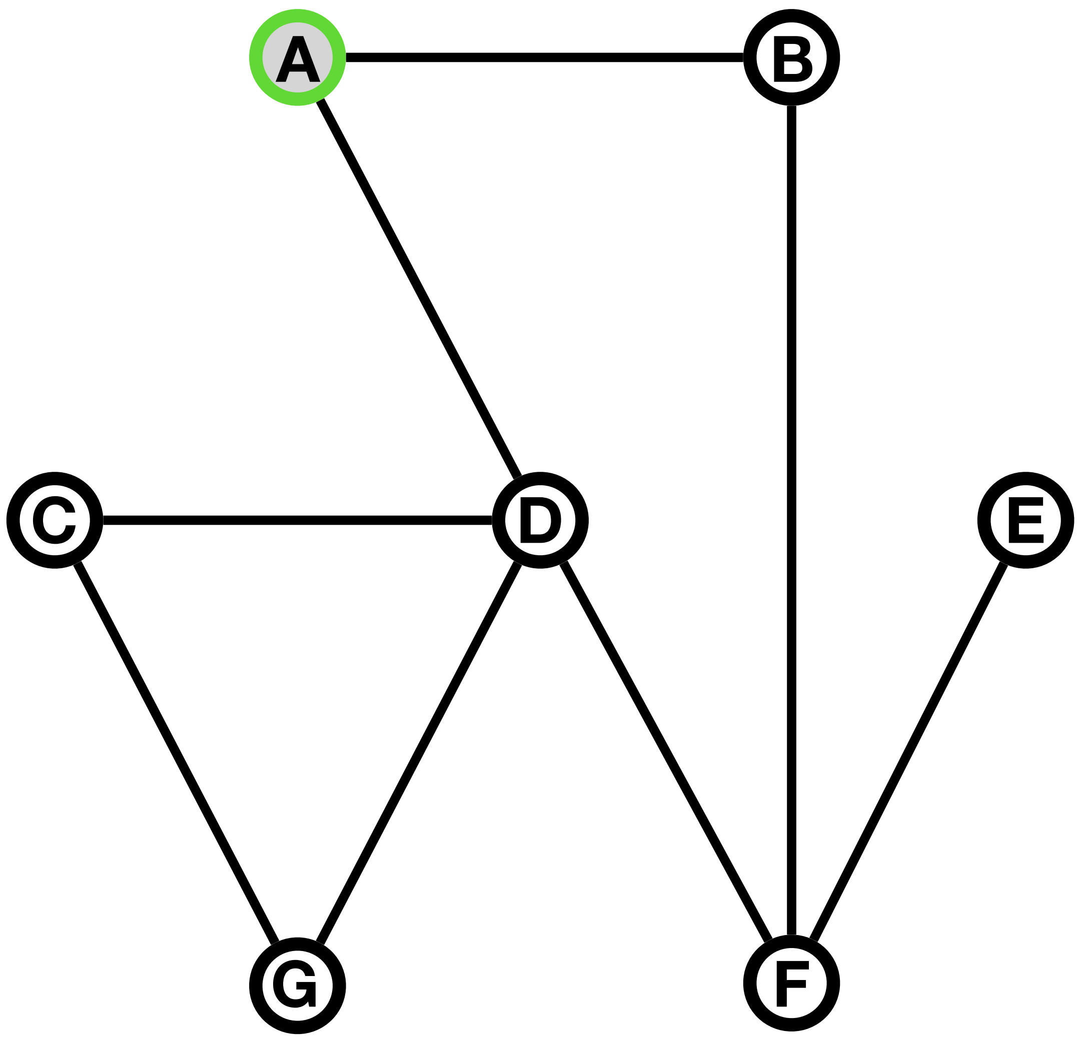
explore nodes as *wide* as possible before looking *deep*
(i.e. examine all siblings before descendants)



QUEUE:
A

Breadth First Search

explore nodes as *wide* as possible before looking *deep*
(i.e. examine all siblings before descendants)

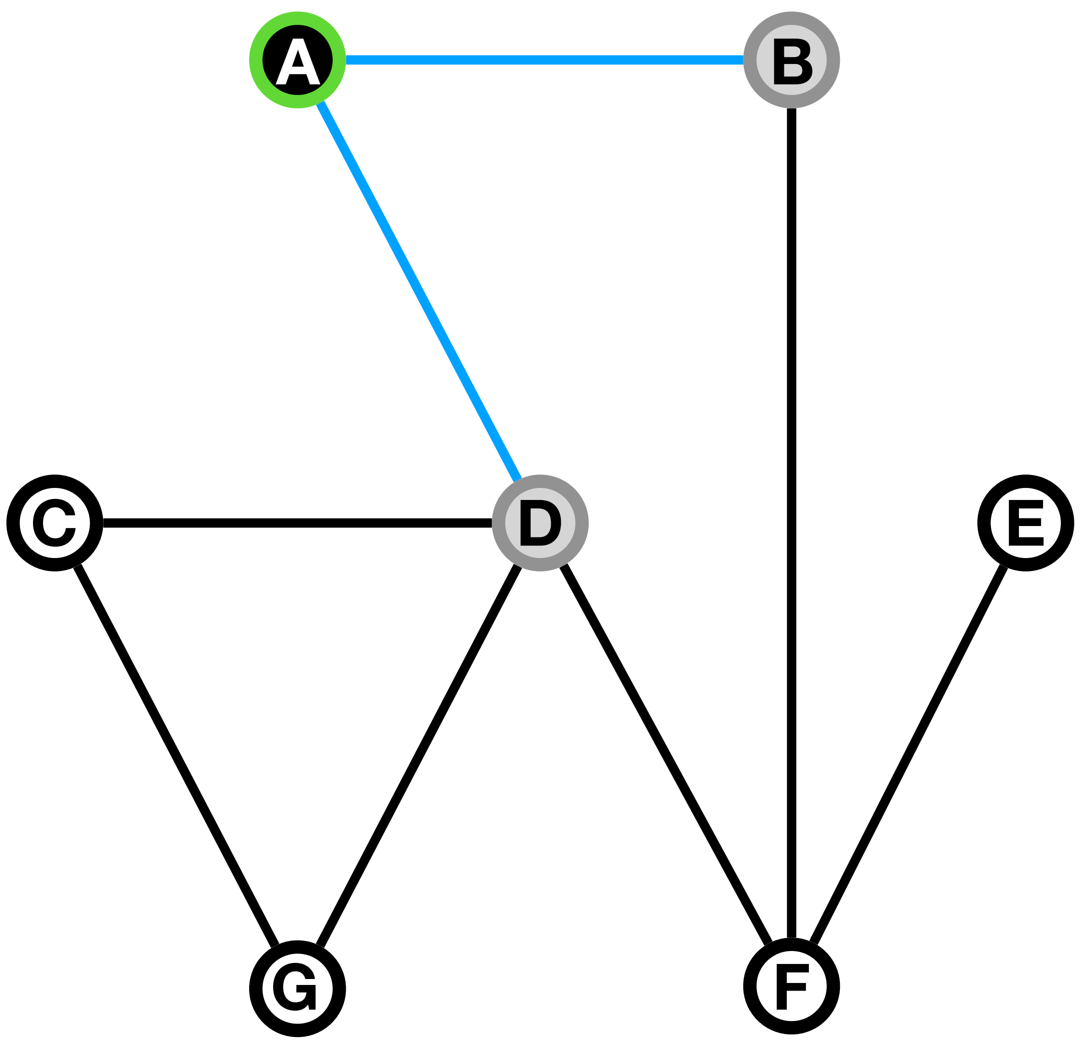


QUEUE:

A →

Breadth First Search

explore nodes as *wide* as possible before looking *deep*
(i.e. examine all siblings before descendants)

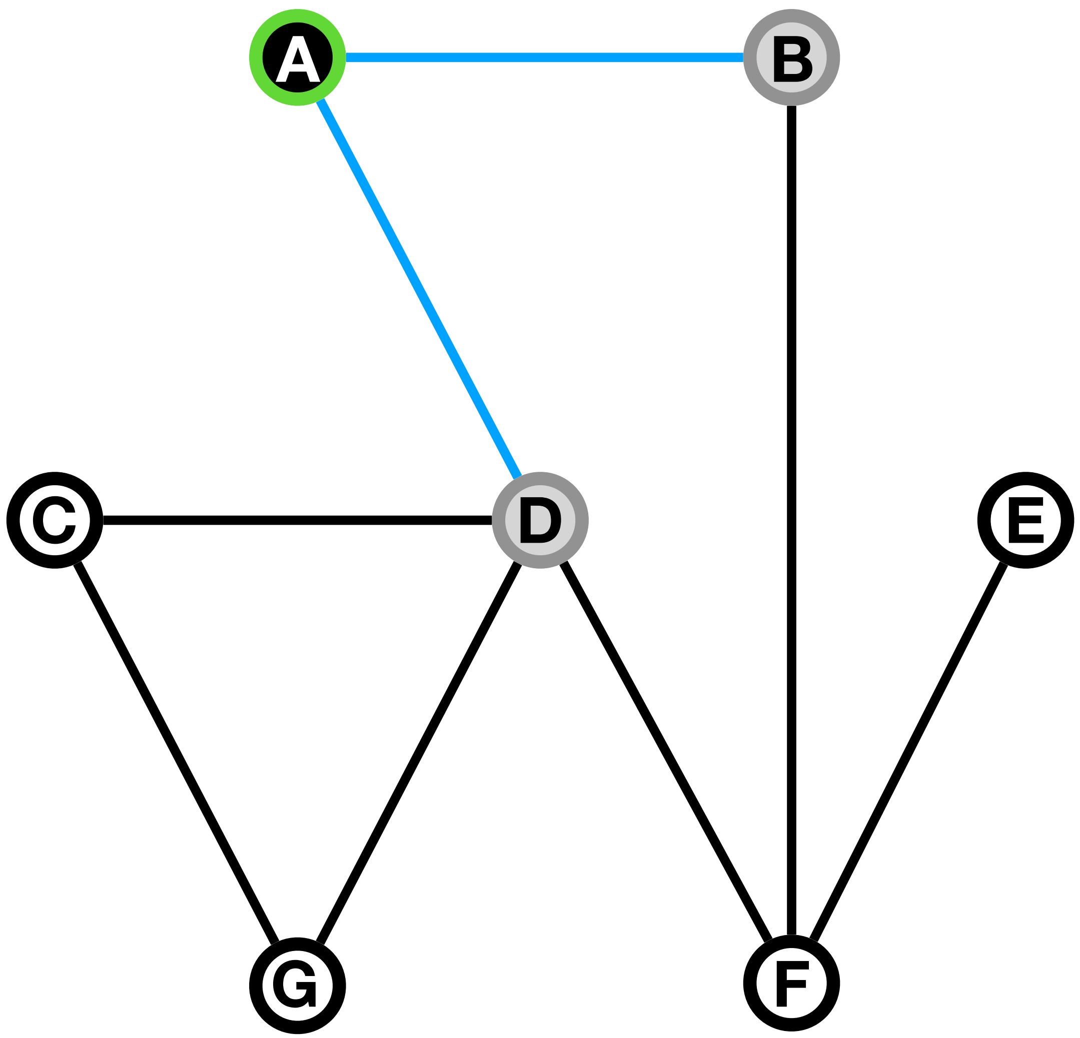


QUEUE:

B
D

Breadth First Search

explore nodes as *wide* as possible before looking *deep*
(i.e. examine all siblings before descendants)

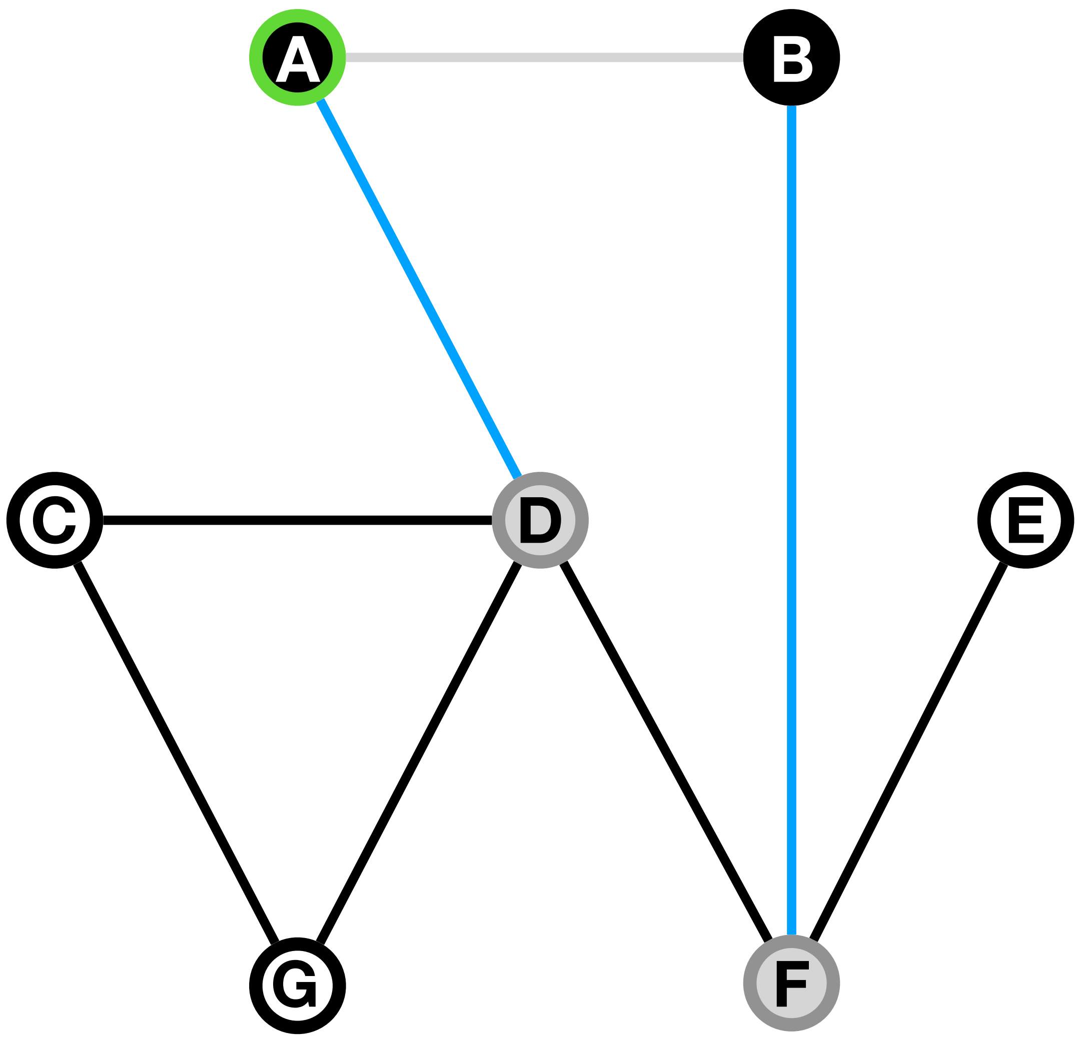


QUEUE:

B
D

Breadth First Search

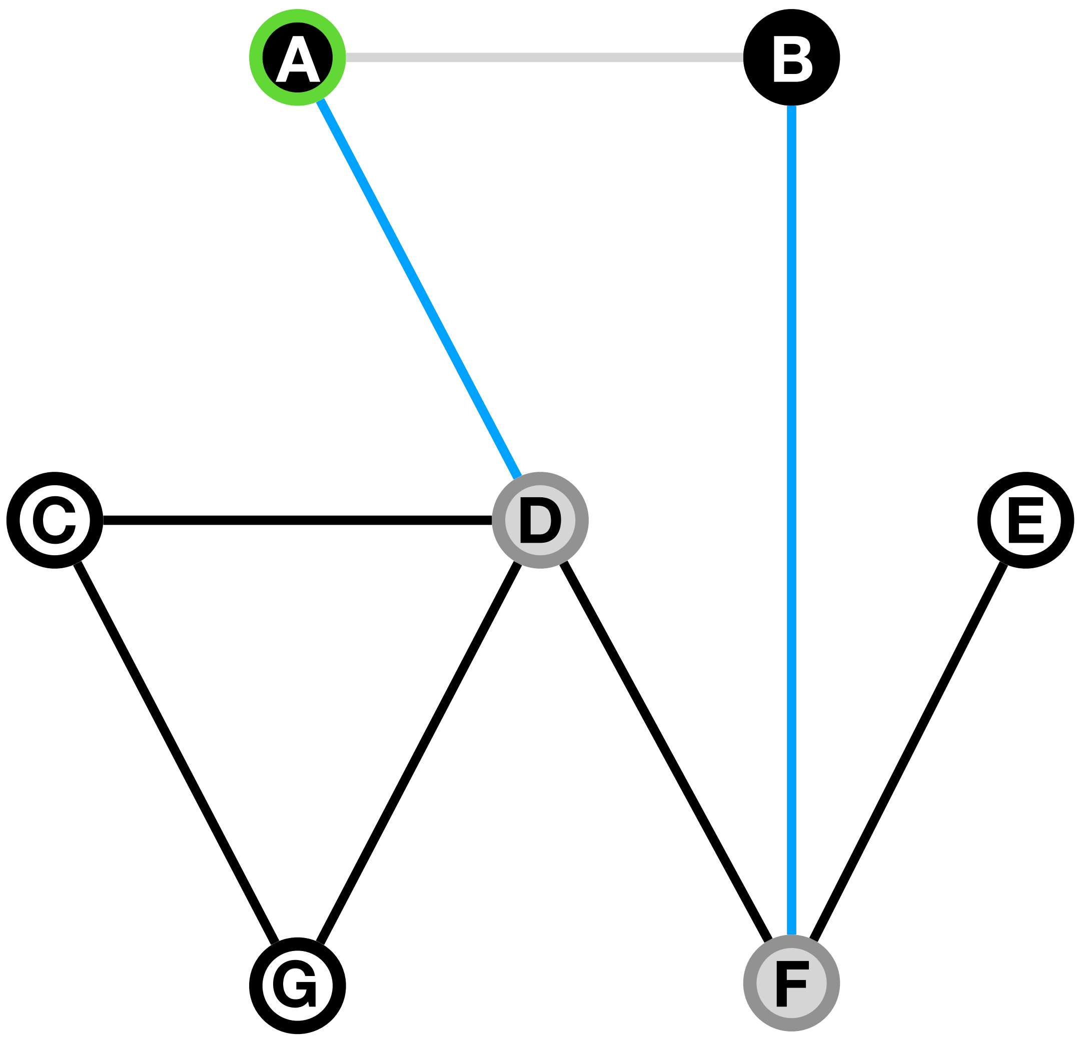
explore nodes as *wide* as possible before looking *deep*
(i.e. examine all siblings before descendants)



QUEUE:
D
F

Breadth First Search

explore nodes as *wide* as possible before looking *deep*
(i.e. examine all siblings before descendants)

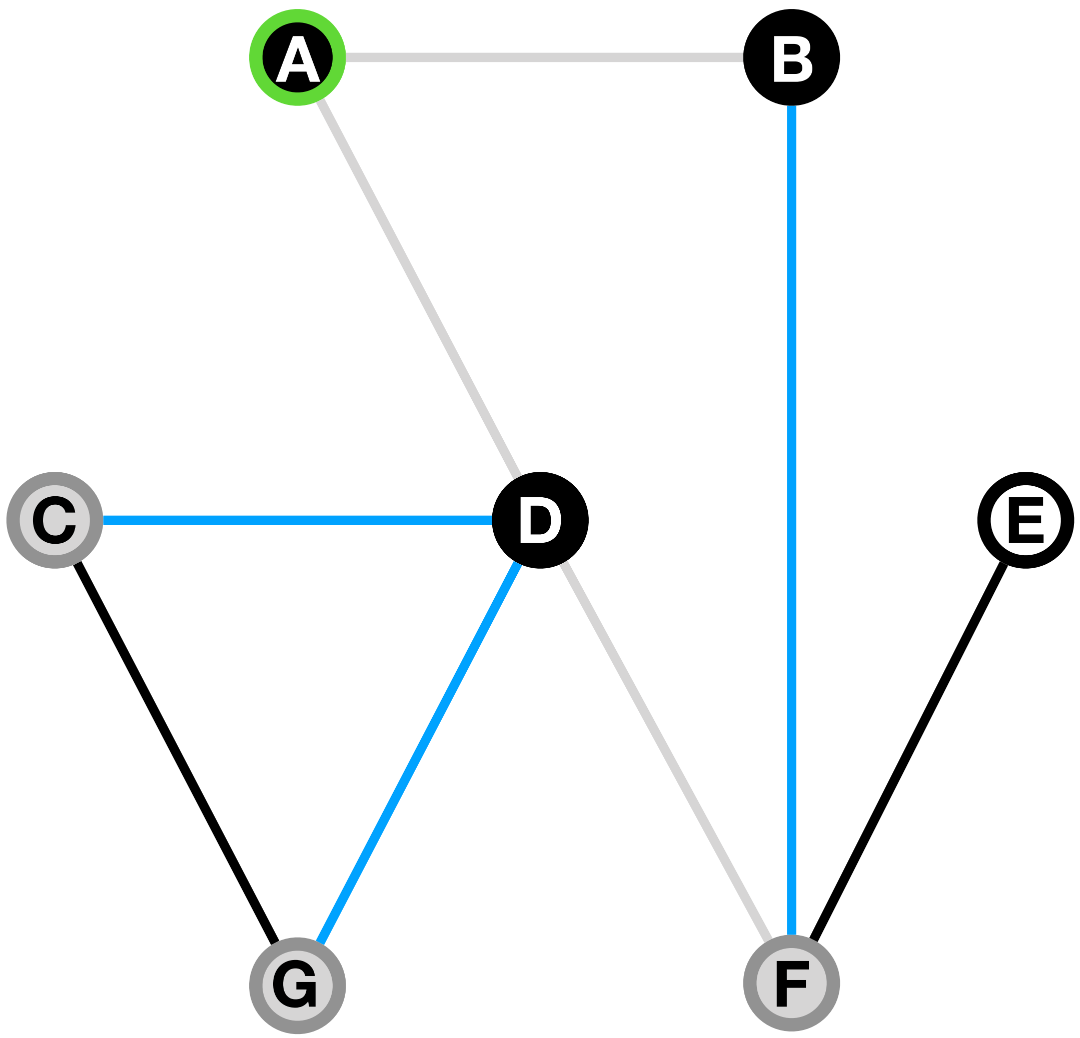


QUEUE:

D →
F

Breadth First Search

explore nodes as *wide* as possible before looking *deep*
(i.e. examine all siblings before descendants)

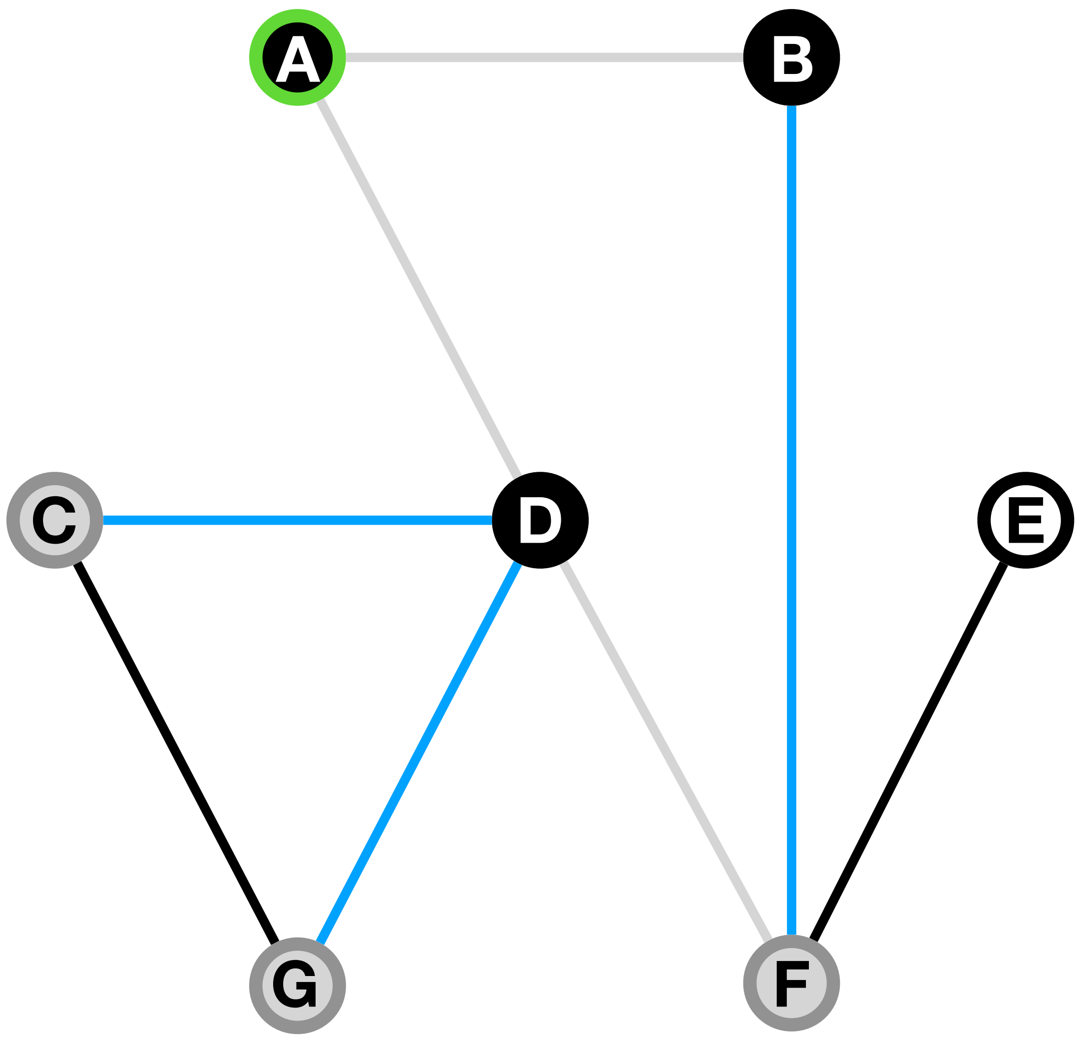


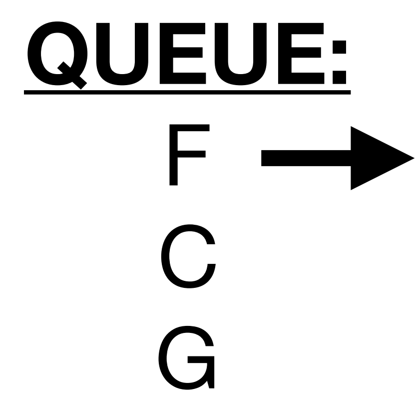
QUEUE:

F
C
G

Breadth First Search

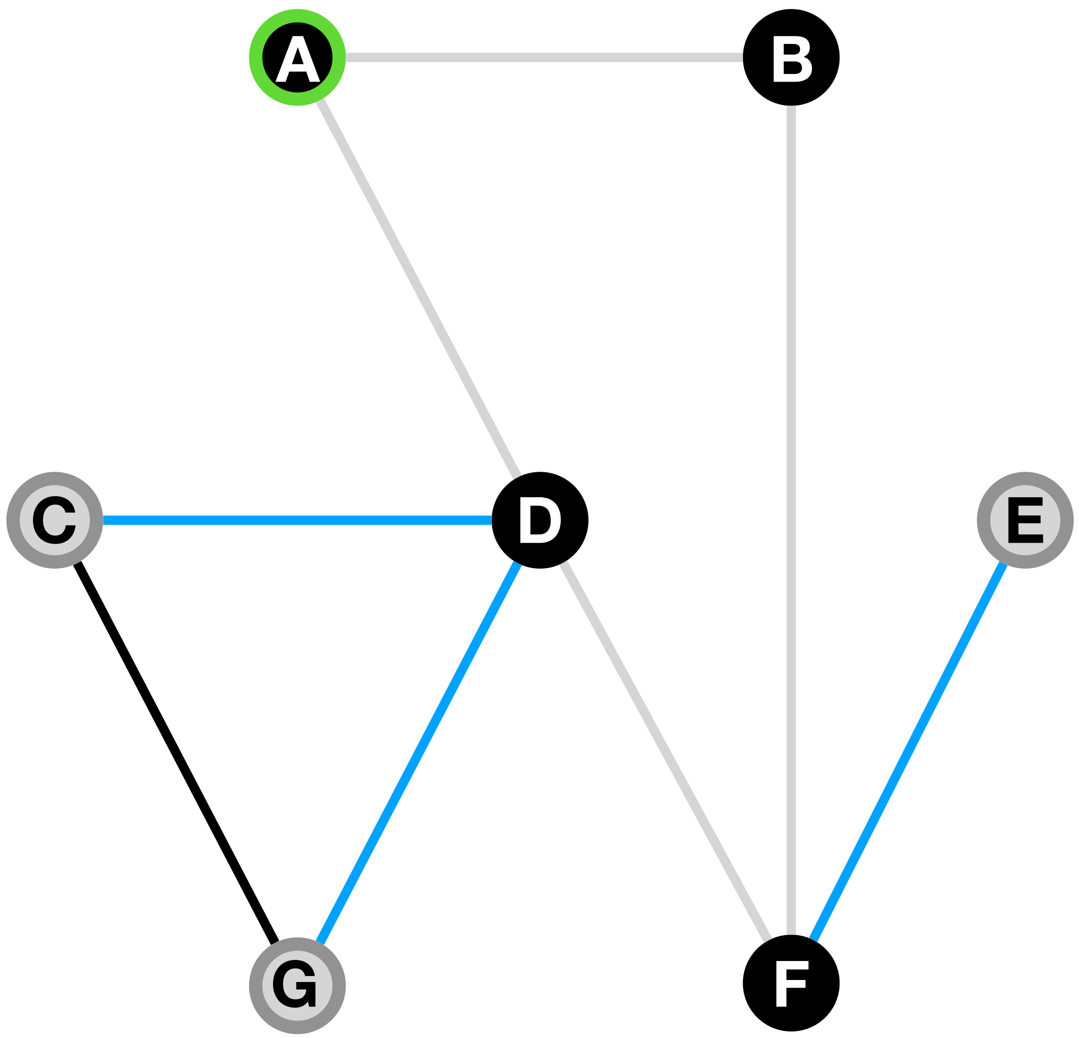
explore nodes as *wide* as possible before looking *deep*
(i.e. examine all siblings before descendants)





Breadth First Search

explore nodes as *wide* as possible before looking *deep*
(i.e. examine all siblings before descendants)

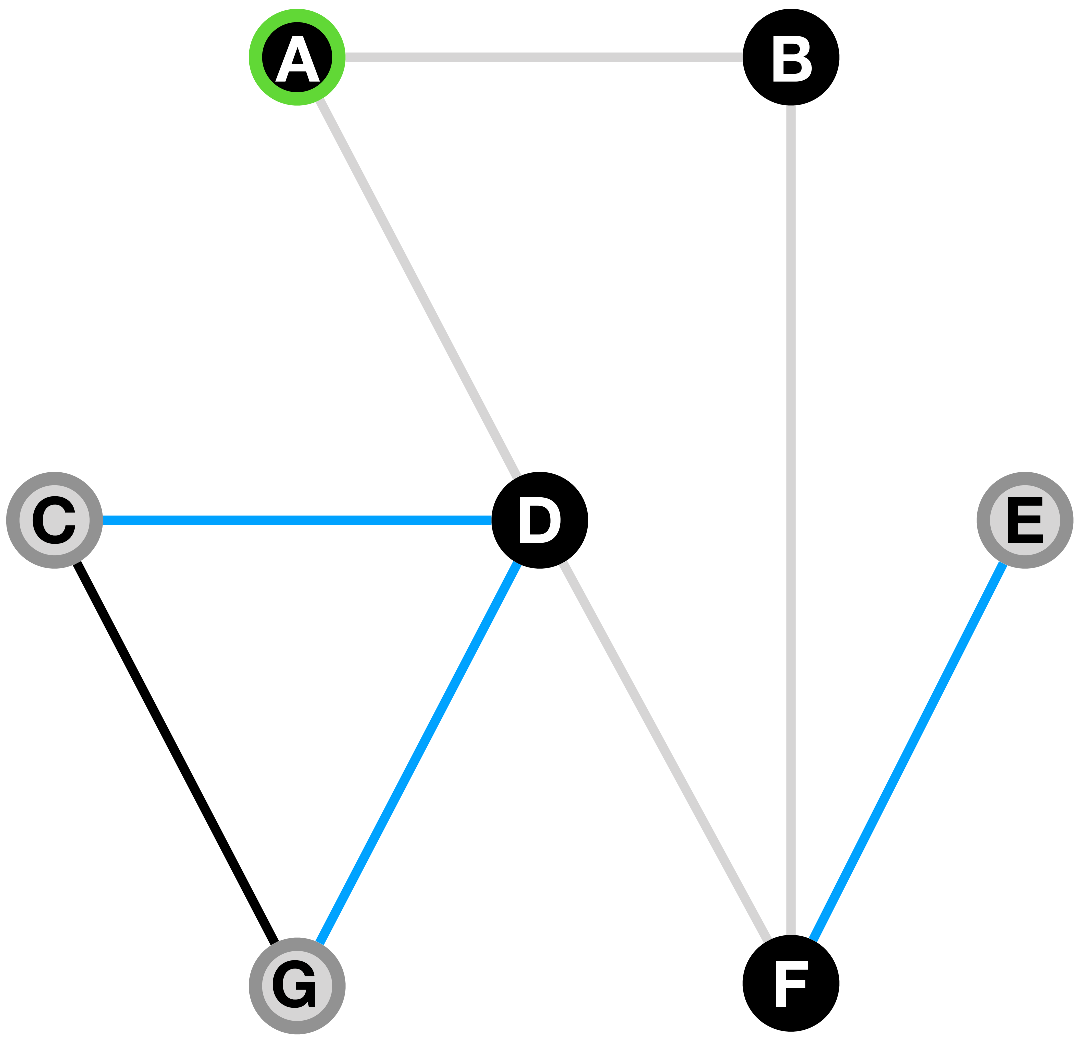


QUEUE:

C
G
E

Breadth First Search

explore nodes as *wide* as possible before looking *deep*
(i.e. examine all siblings before descendants)

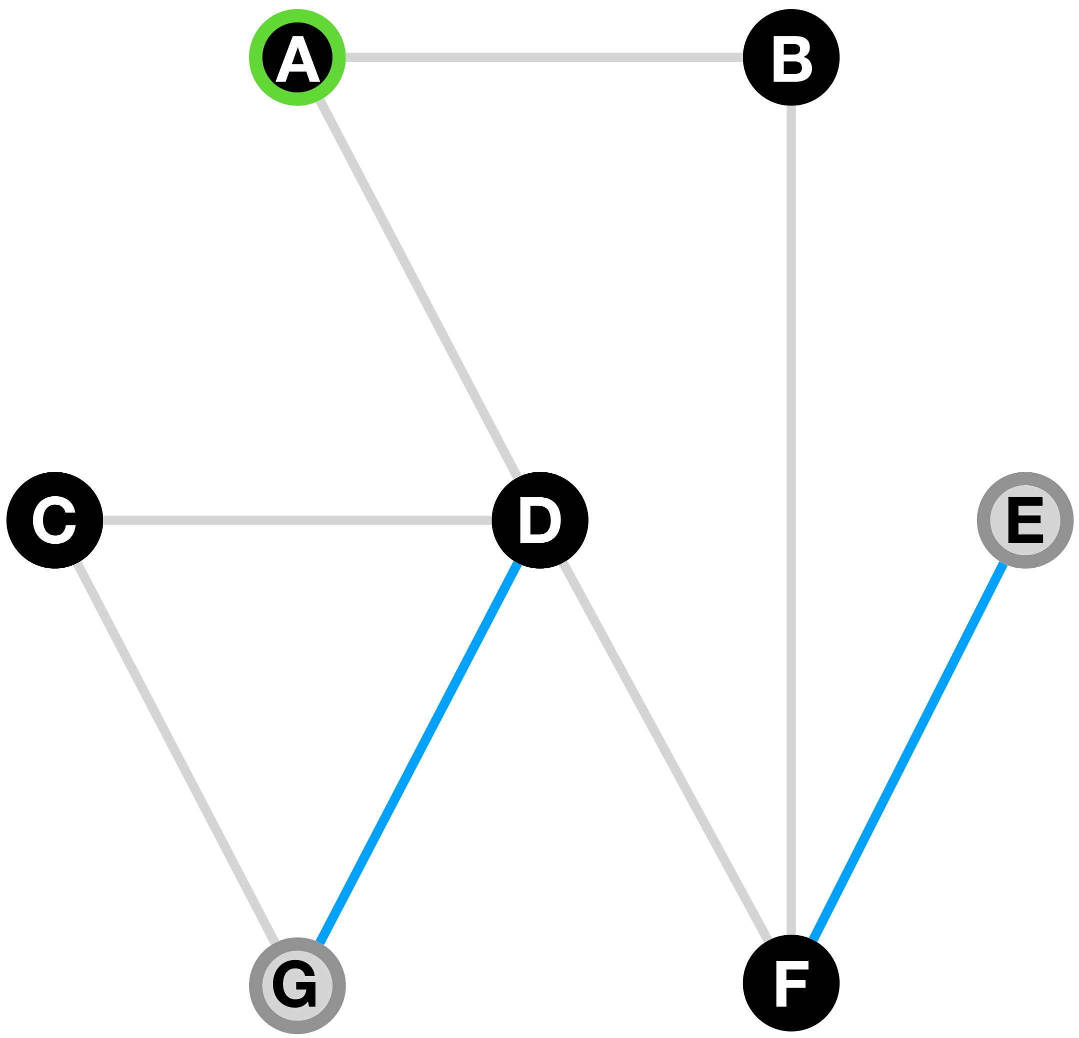


QUEUE:

C
G
E

Breadth First Search

explore nodes as *wide* as possible before looking *deep*
(i.e. examine all siblings before descendants)

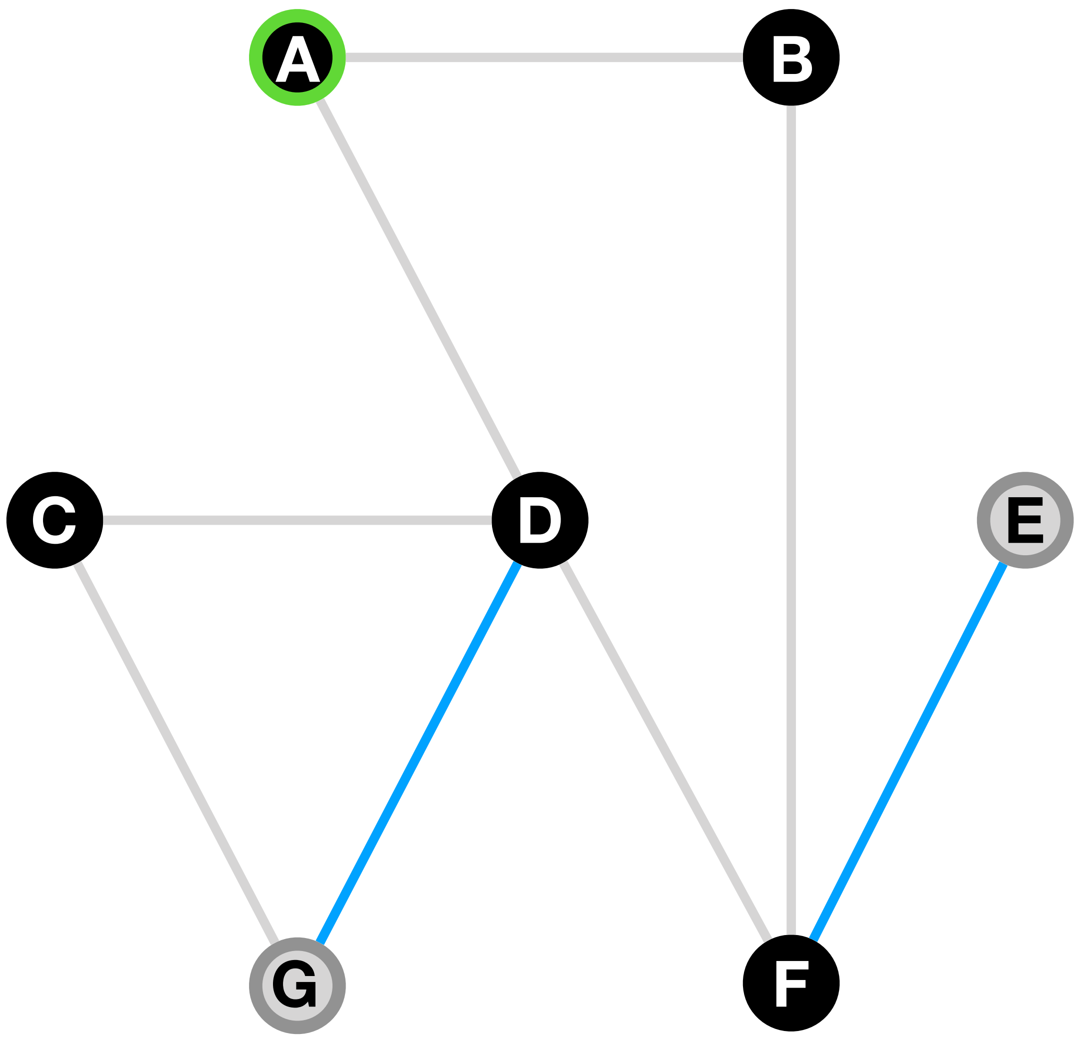


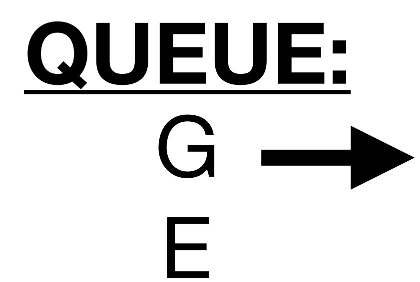
QUEUE:

G
E

Breadth First Search

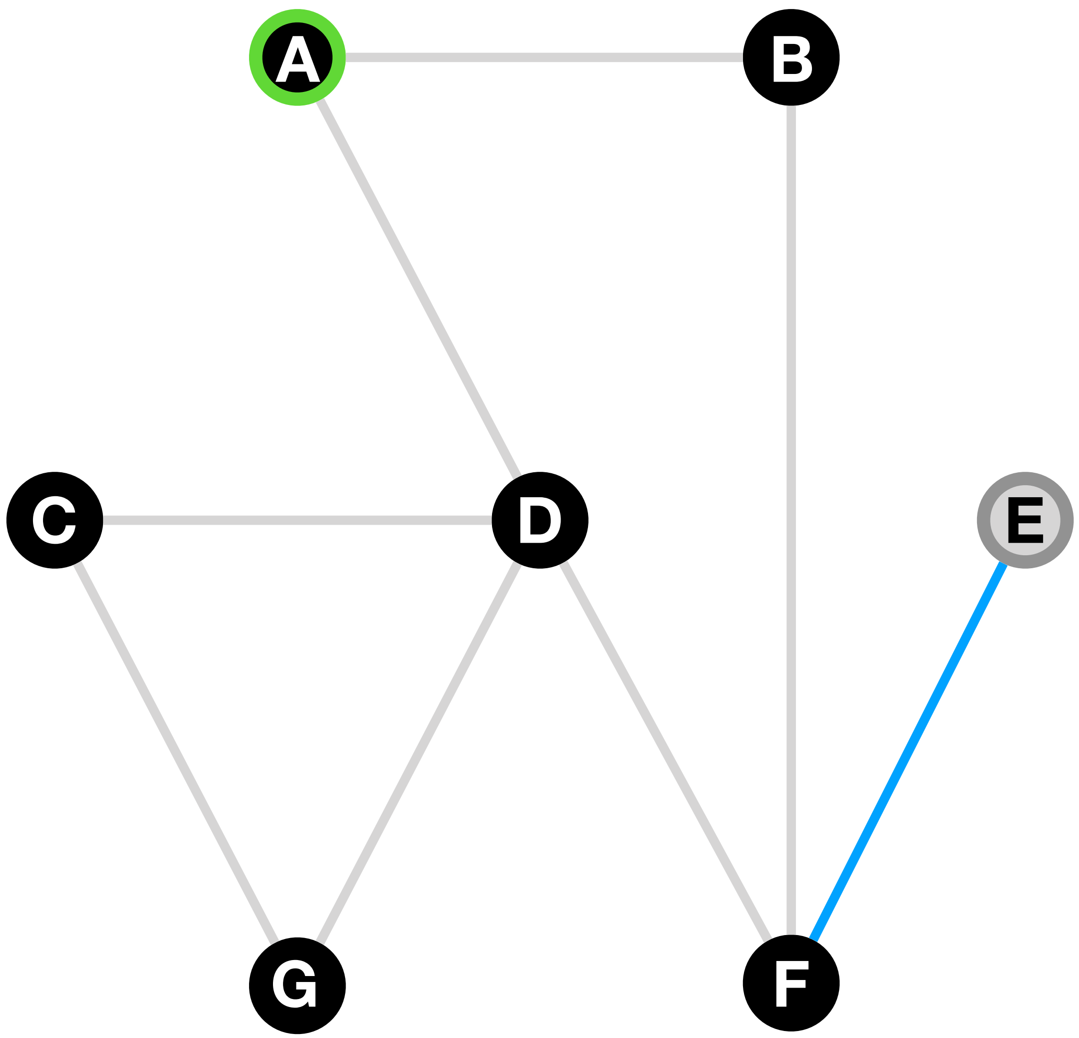
explore nodes as *wide* as possible before looking *deep*
(i.e. examine all siblings before descendants)





Breadth First Search

explore nodes as *wide* as possible before looking *deep*
(i.e. examine all siblings before descendants)

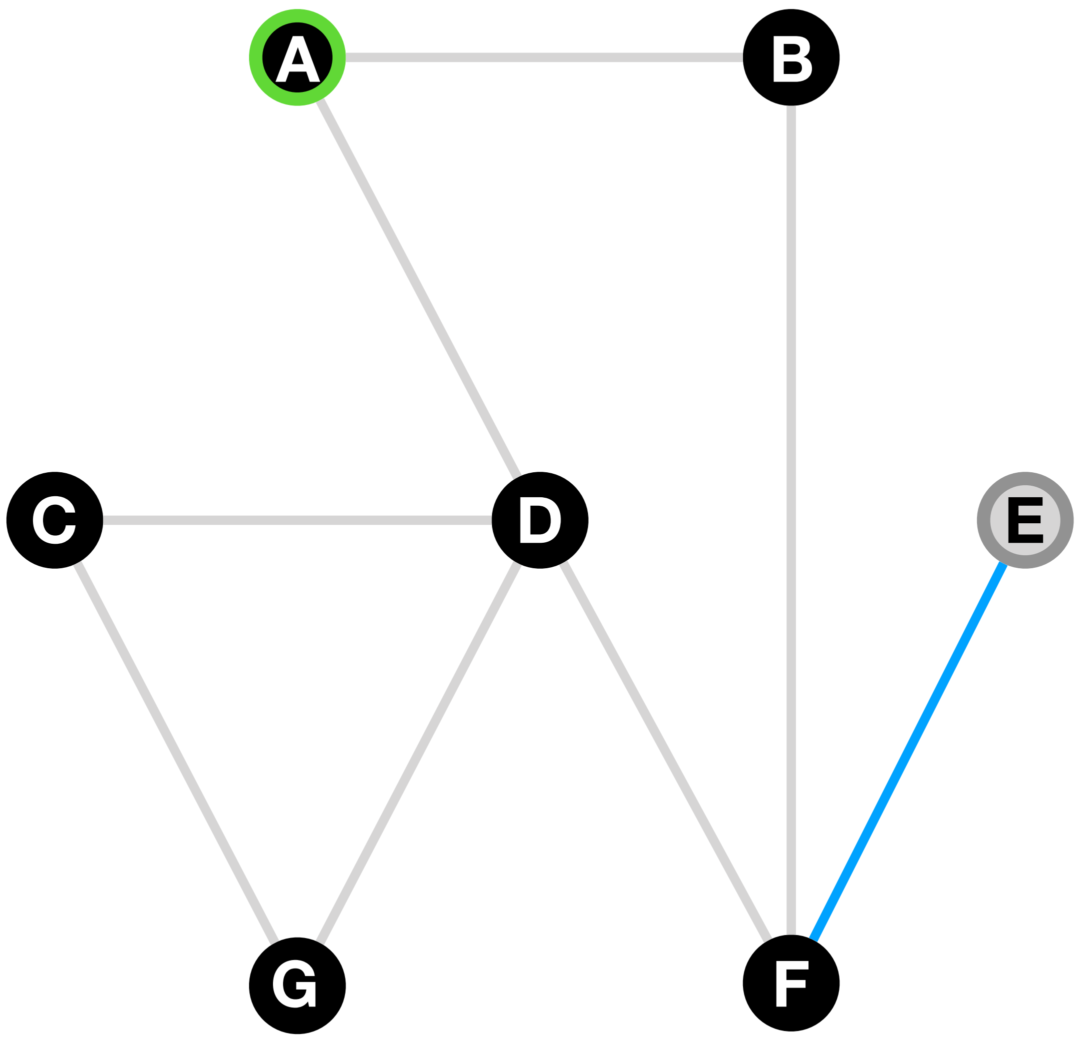


QUEUE:

E

Breadth First Search

explore nodes as *wide* as possible before looking *deep*
(i.e. examine all siblings before descendants)

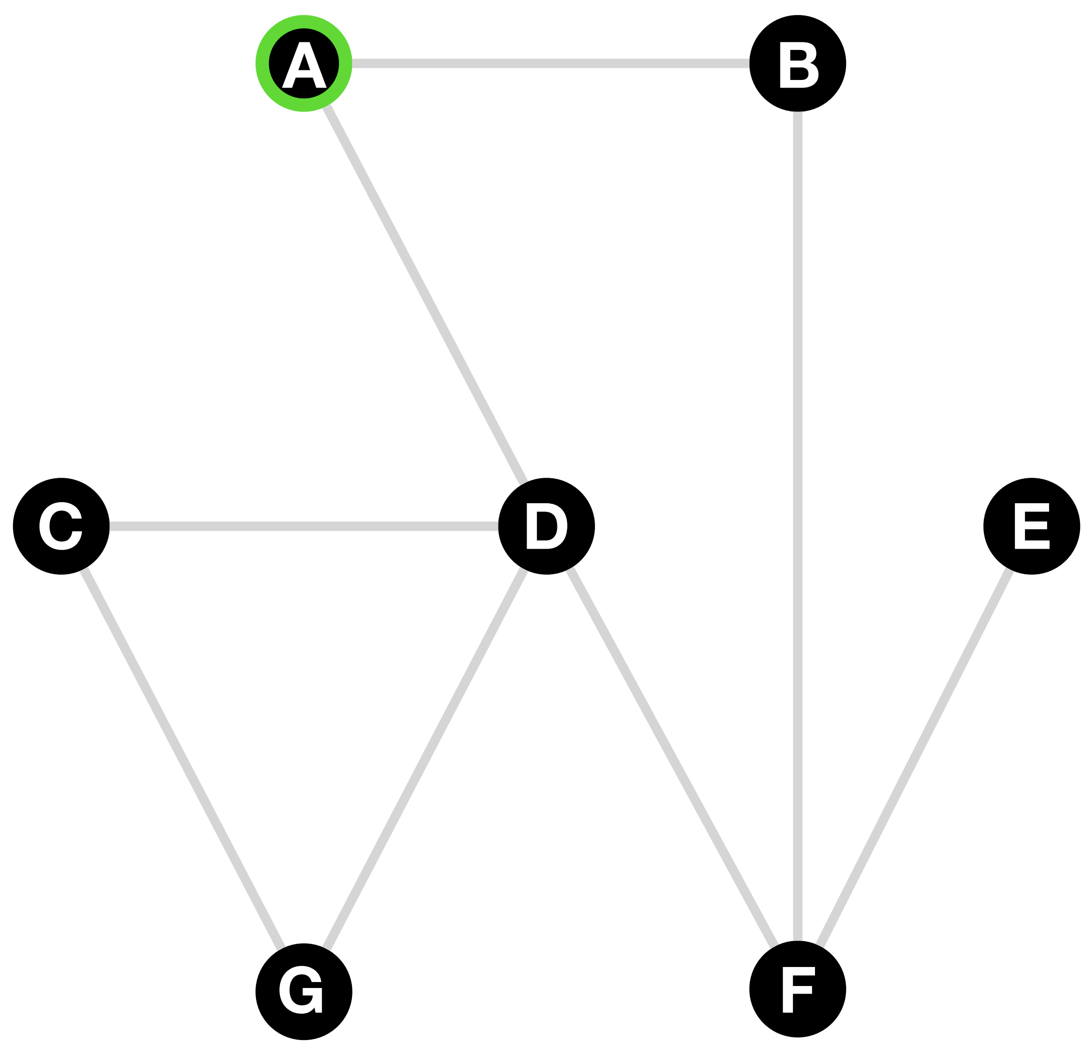


QUEUE:

E

Breadth First Search

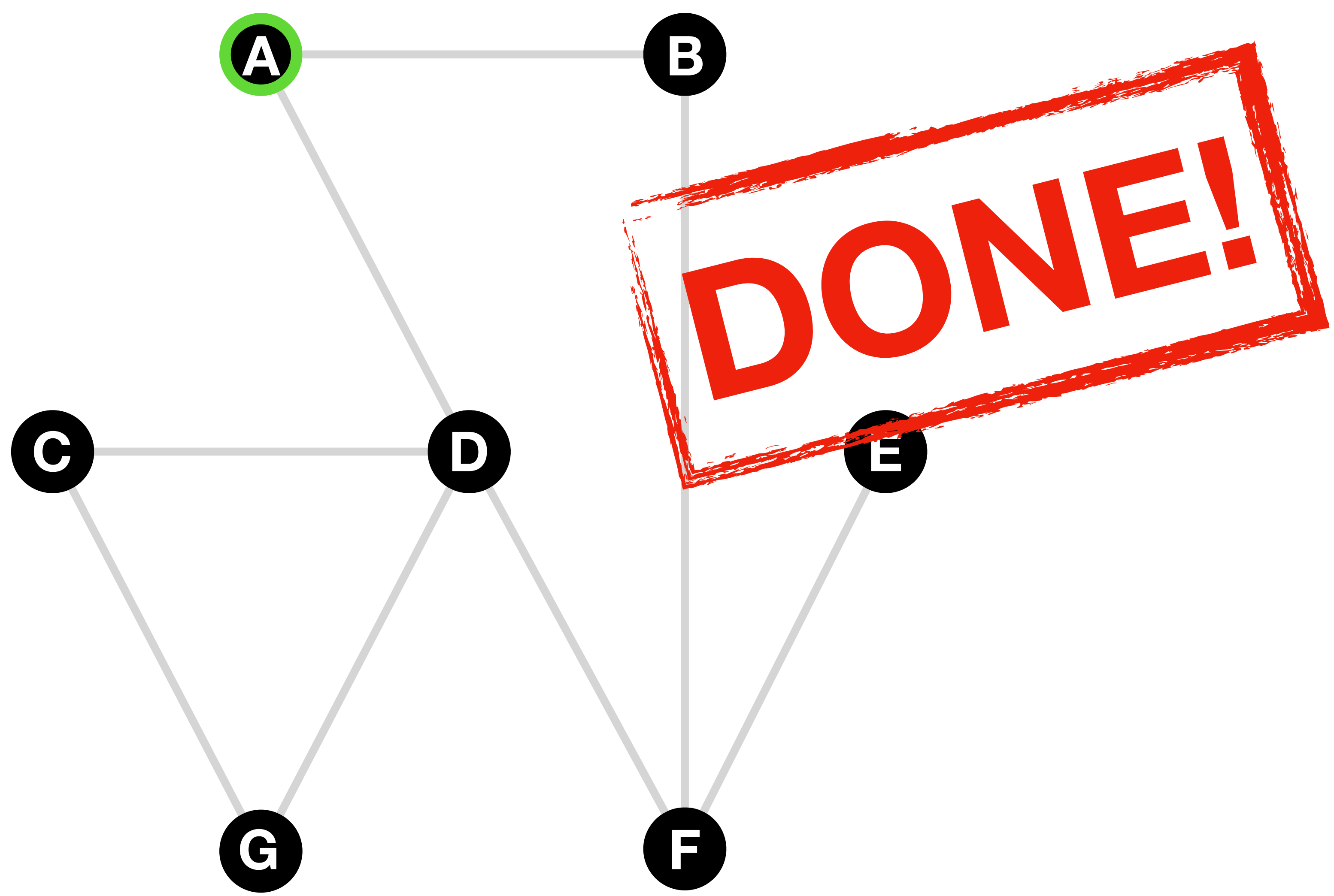
explore nodes as *wide* as possible before looking *deep*
(i.e. examine all siblings before descendants)



QUEUE:

Breadth First Search

explore nodes as *wide* as possible before looking *deep*
(i.e. examine all siblings before descendants)



QUEUE:

Breadth First Search

```
BreadthFirstSearch(G, vertex, visited[ ]) {
  initialize q                        # queue
  initialize next                     # next vertex

  q.enqueue(vertex)                   # add vertex to queue
  while (q is not empty) {
    next = q.dequeue()                # remove from top of queue
    if ( !visited[next] ) {           # next has not been visited
      visited[next] = True            # mark next as visited
      for each neighbor in G[next]:   # for all of next's neighbors
        q.enqueue(neighbor)           # add neighbor to queue
    }
  }
}
```

Complexity:

O(V + E). each vertex enqueued/dequeued once and each edge traversed once.

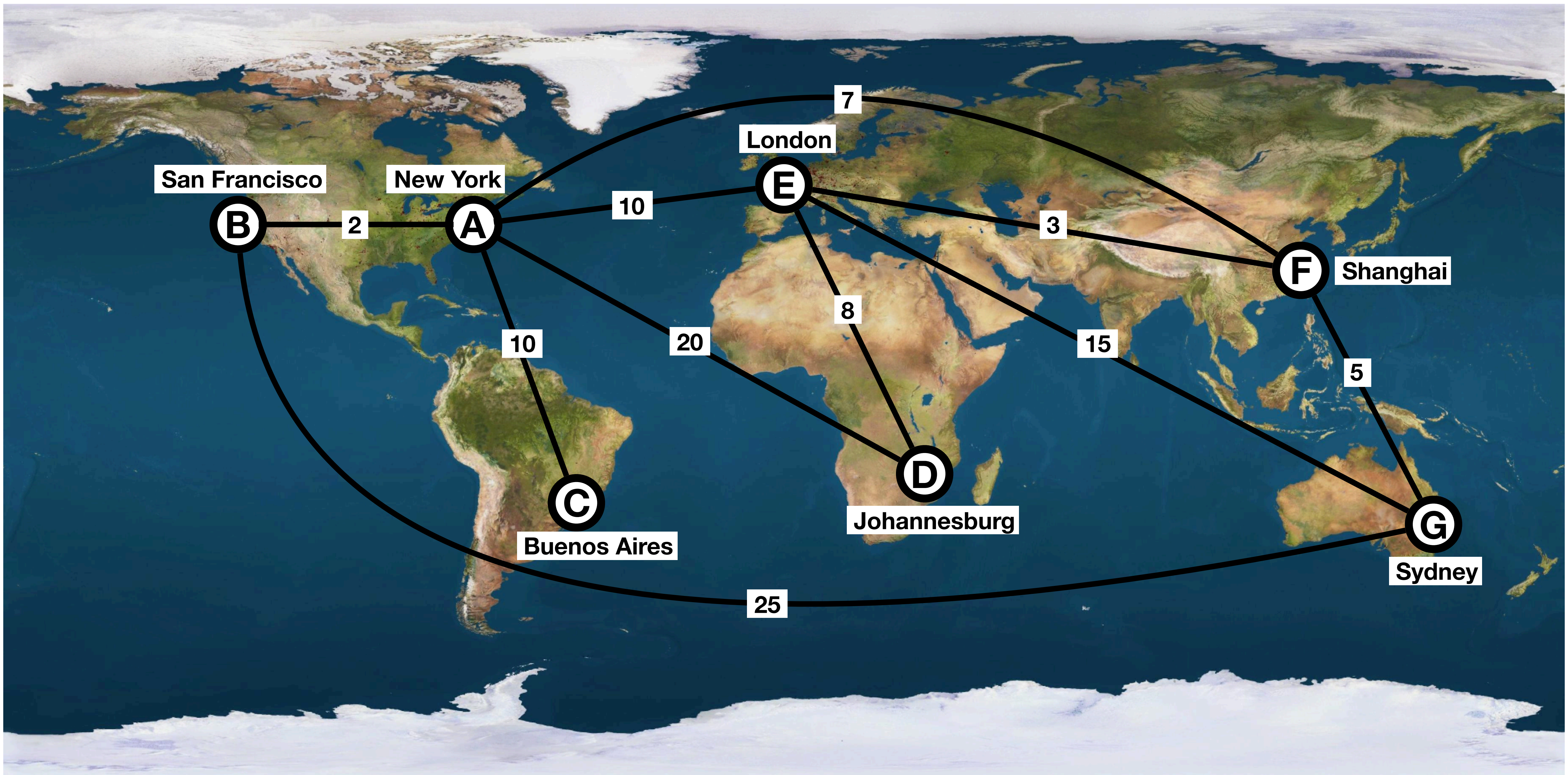
Dijkstra's Shortest Path

- keep a table to track distance from start node, previous node in path, and whether a node's place in path is known

V	DIST	PREV	KNOWN
A	0		
B	∞		
C	∞		
...	...	...	...

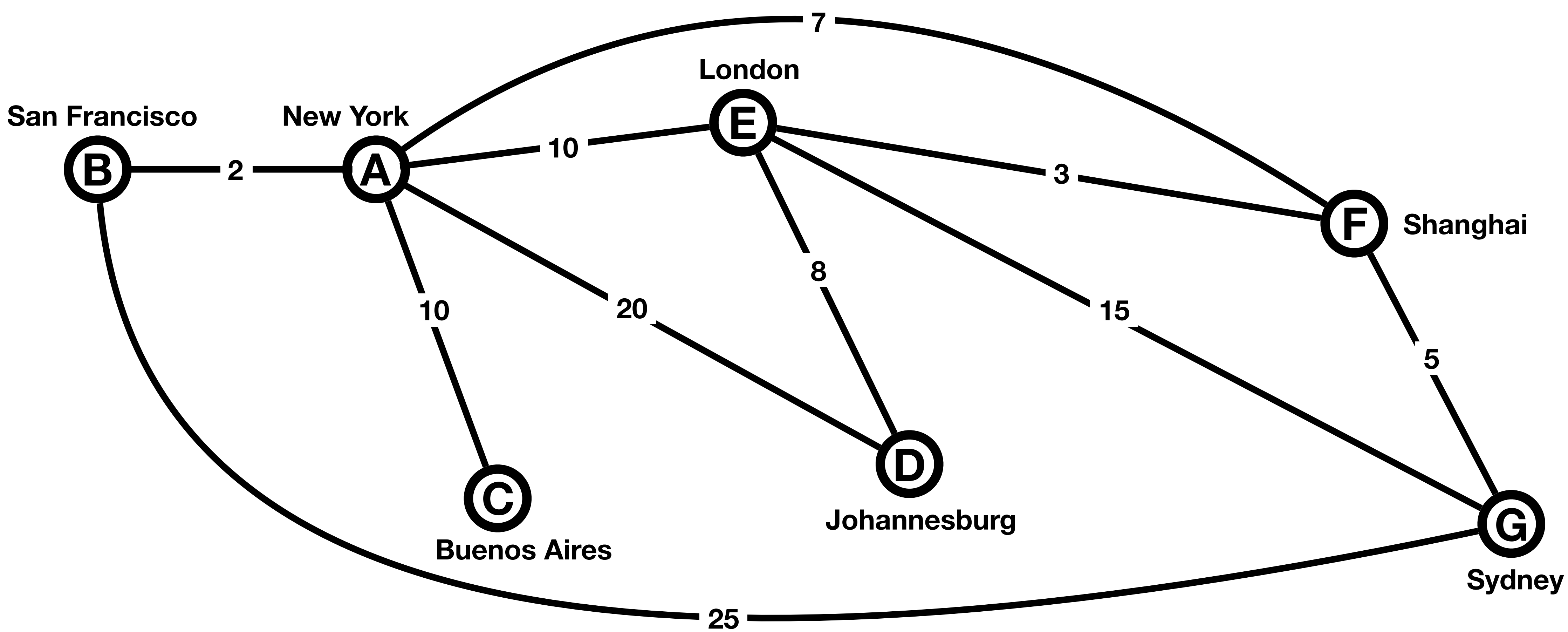
- initialize distance for start node as 0 and all other distances to infinity
- initialize current node, i = start node.
- repeat until all nodes have been visited:
  - for all j adjacent to i,
    - if $distance_i + weight_{i,j} < distance_j$, update $distance_j = distance_i + weight_{i,j}$ and $previous_j = i$
  - update i = to the unvisited node that has the shortest distance to the start node

Dijkstra’s Shortest Path

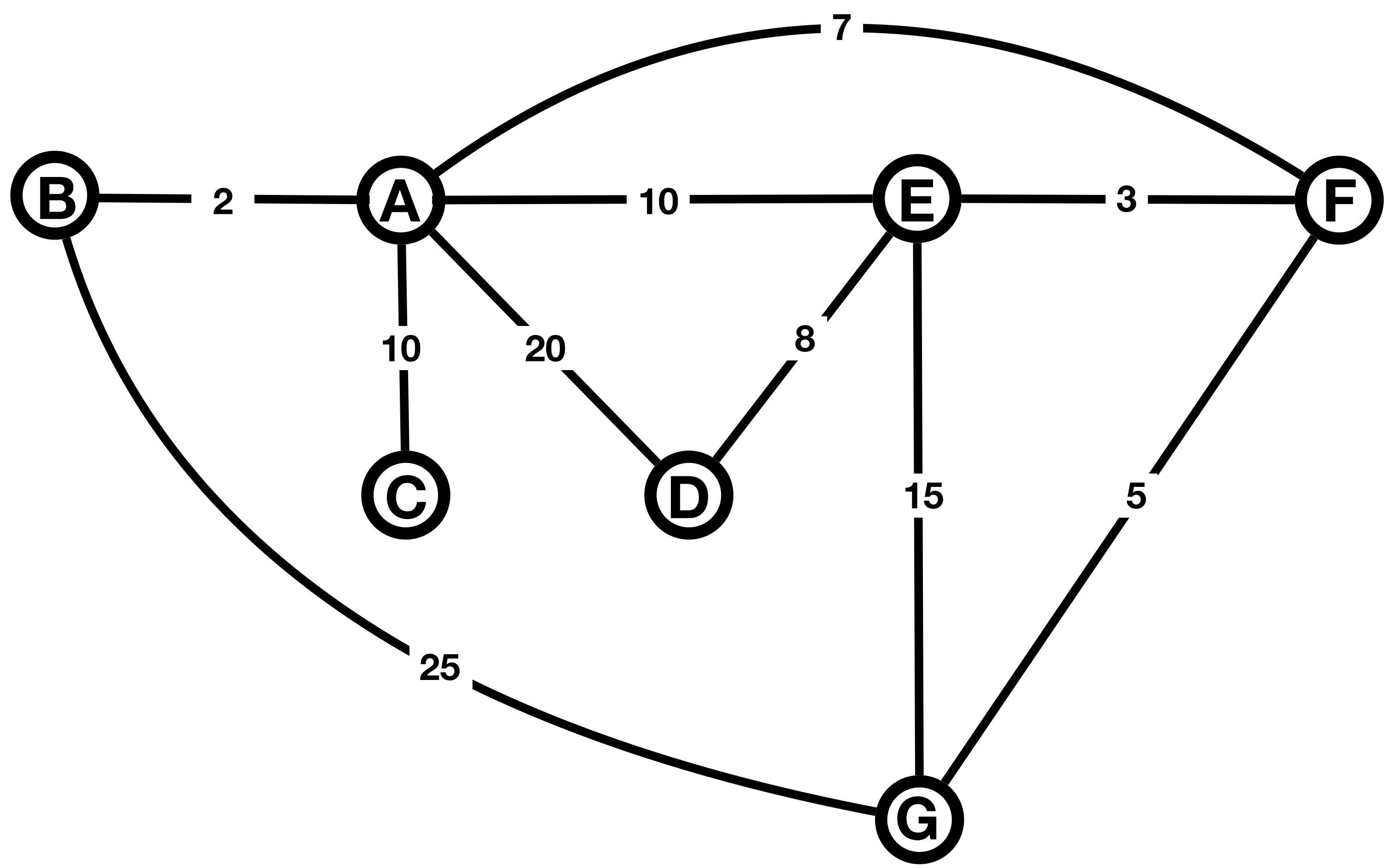


*Edge weights are in hundreds of US dollars

Dijkstra’s Shortest Path

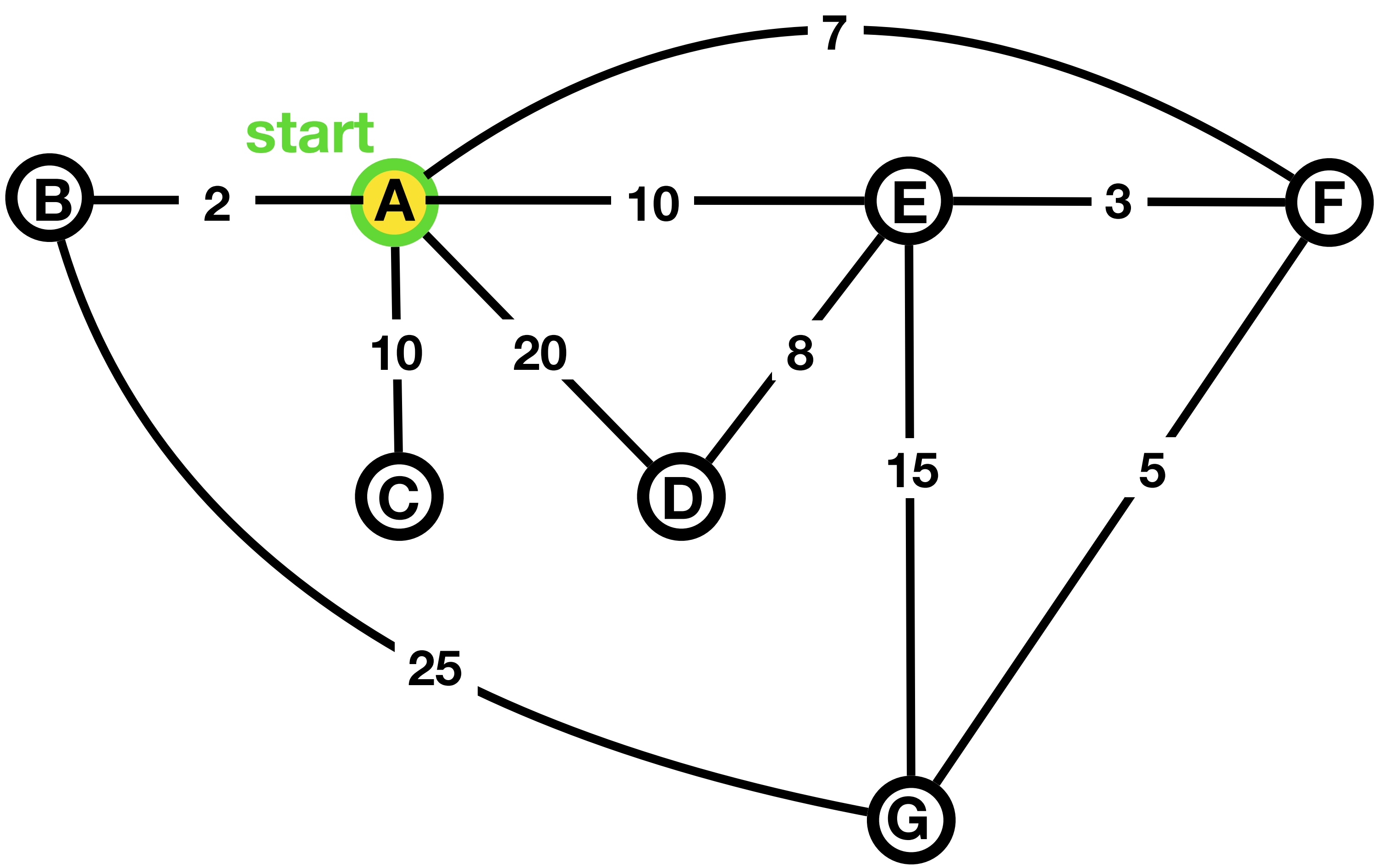


Dijkstra’s Shortest Path



Dijkstra’s Shortest Path

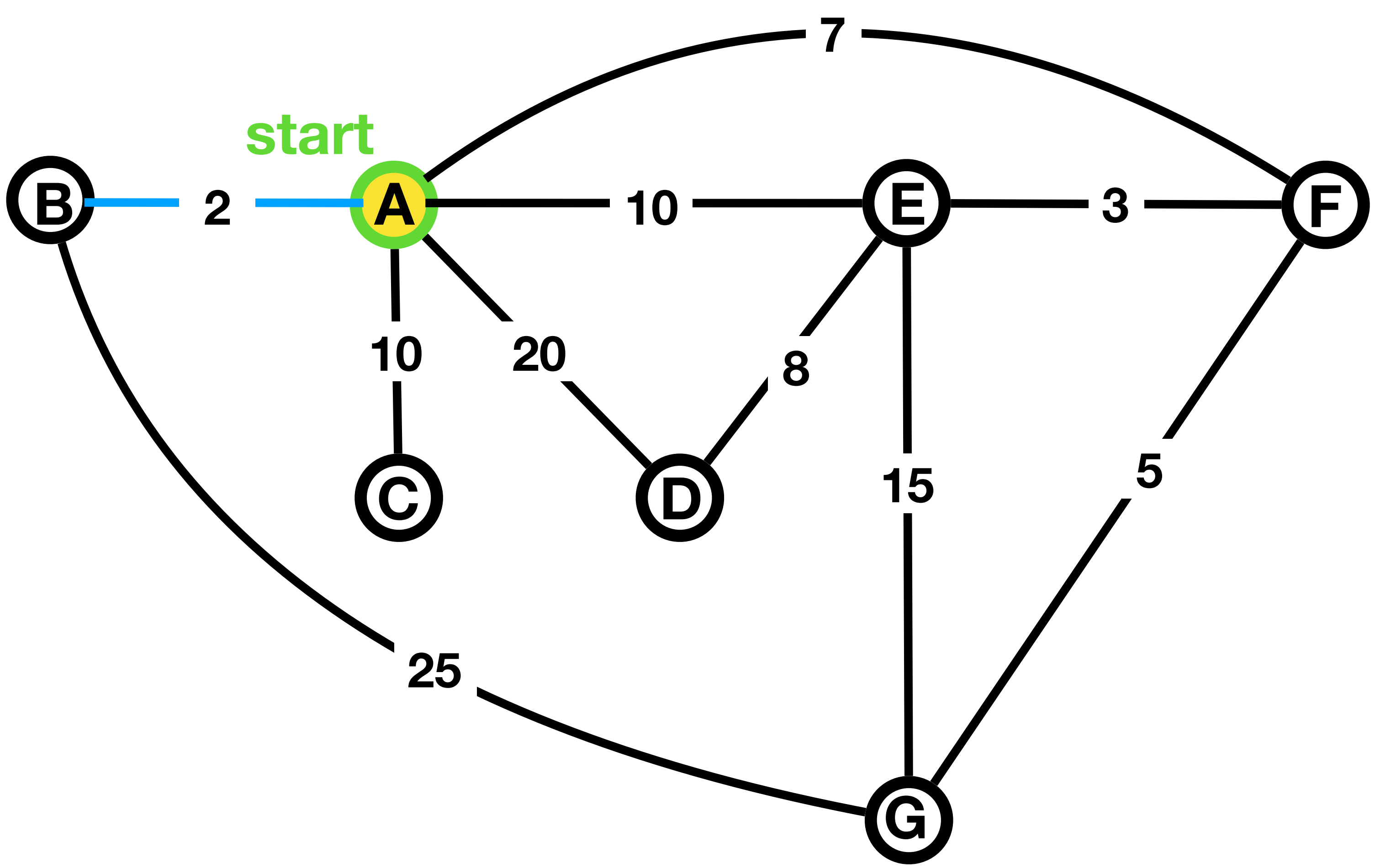
current node	A
next node	



V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	∞		F
C	∞		F
D	∞		F
E	∞		F
F	∞		F
G	∞		F

Dijkstra's Shortest Path

current, *i*	A
next, *j*	B



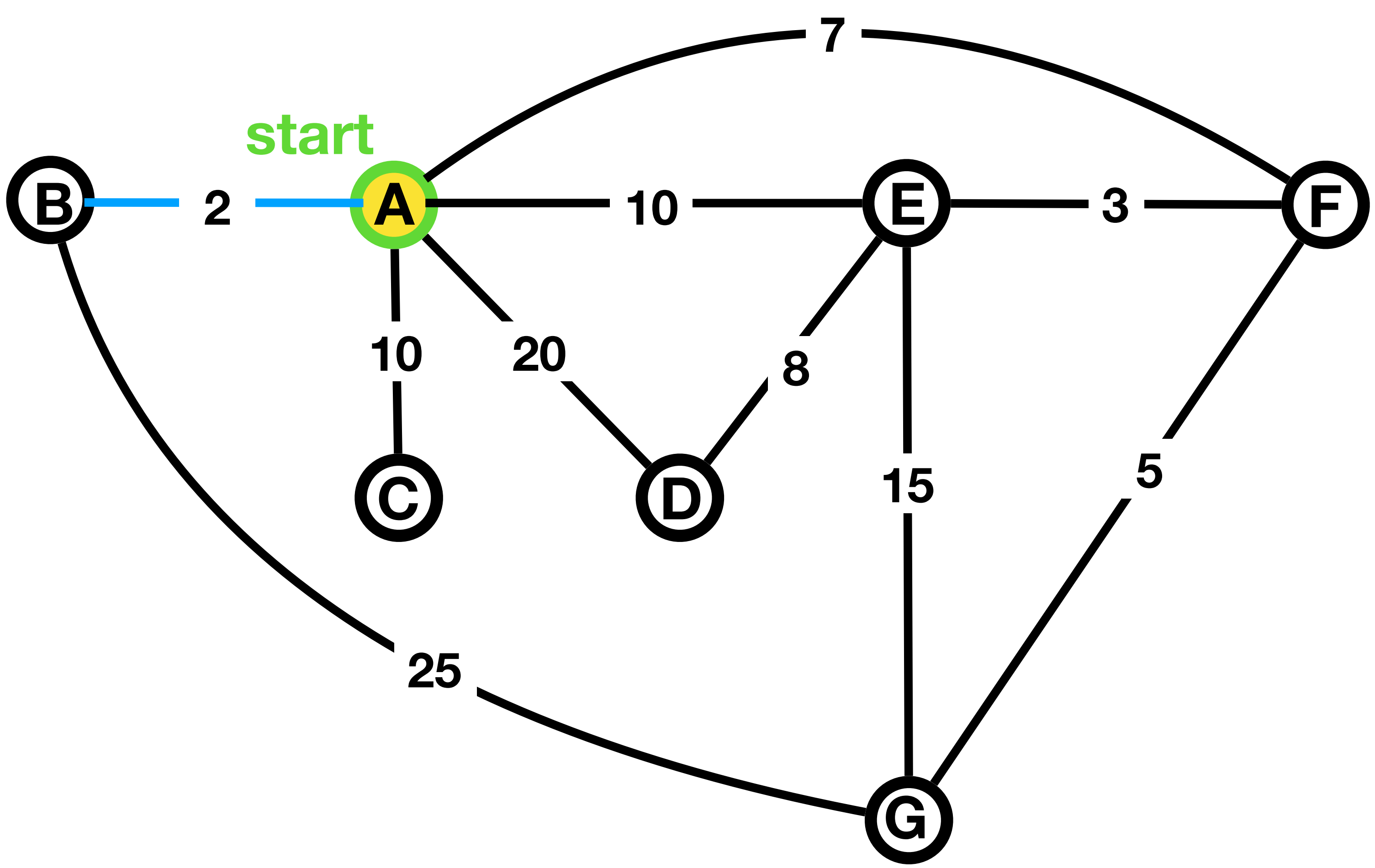
V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	∞		F
C	∞		F
D	∞		F
E	∞		F
F	∞		F
G	∞		F

Is B known ? <u>No</u>

$\text{distance}_i + \text{weight}_{i,j} < \text{distance}_j \;=\; (0) + (2) < (\infty)$?

Dijkstra's Shortest Path

current, *i*	A
next, *j*	B



V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	F
C	∞		F
D	∞		F
E	∞		F
F	∞		F
G	∞		F

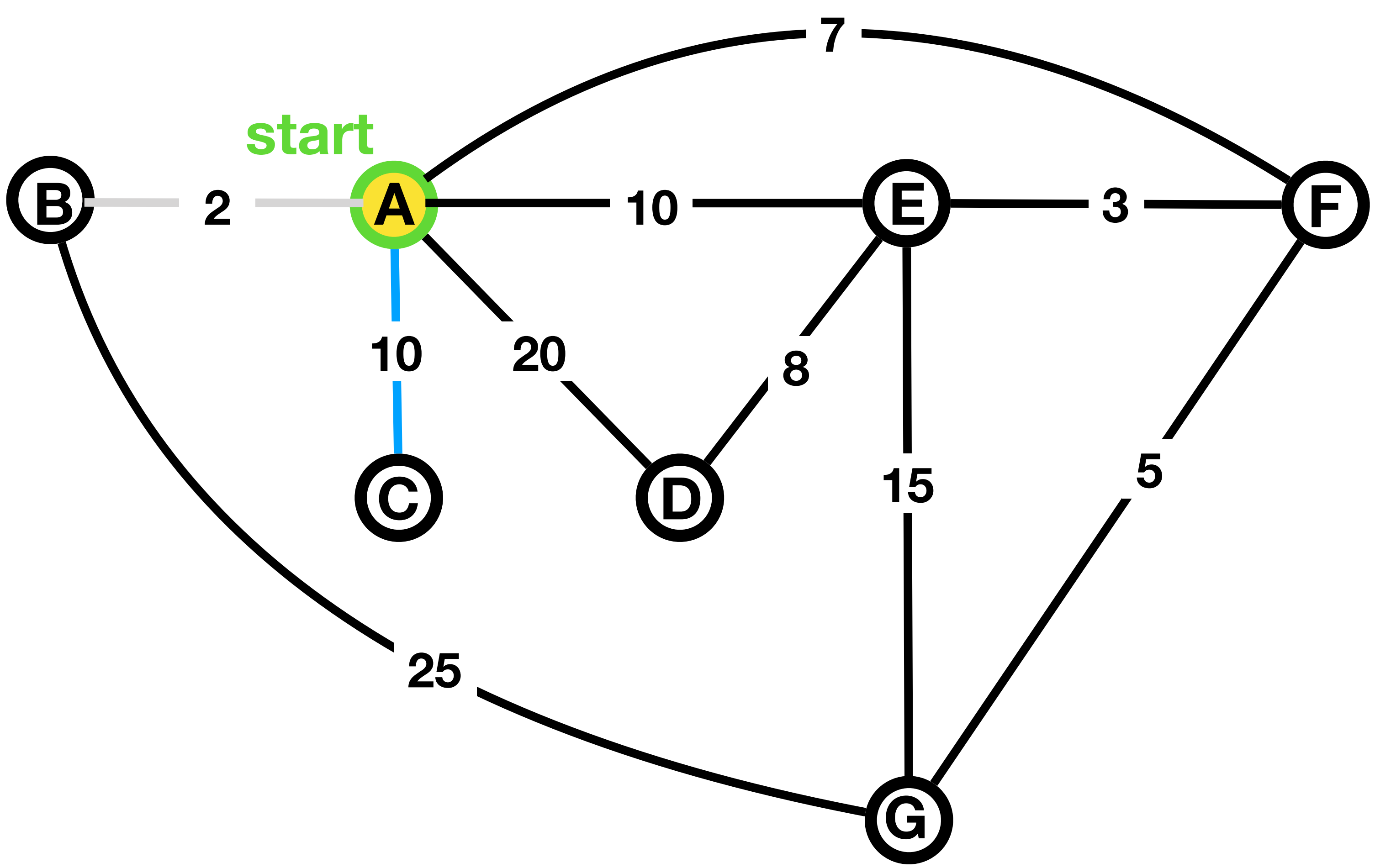
Is B known ? <u>No</u>

$\text{distance}_i + \text{weight}_{i,j} < \text{distance}_j$ = (0) + (2) < (∞) ? <u>Yes</u>

=> update distance_j = (0) + (2) and previous_j = current = A

Dijkstra's Shortest Path

current, *i*	**A**
next, *j*	**C**



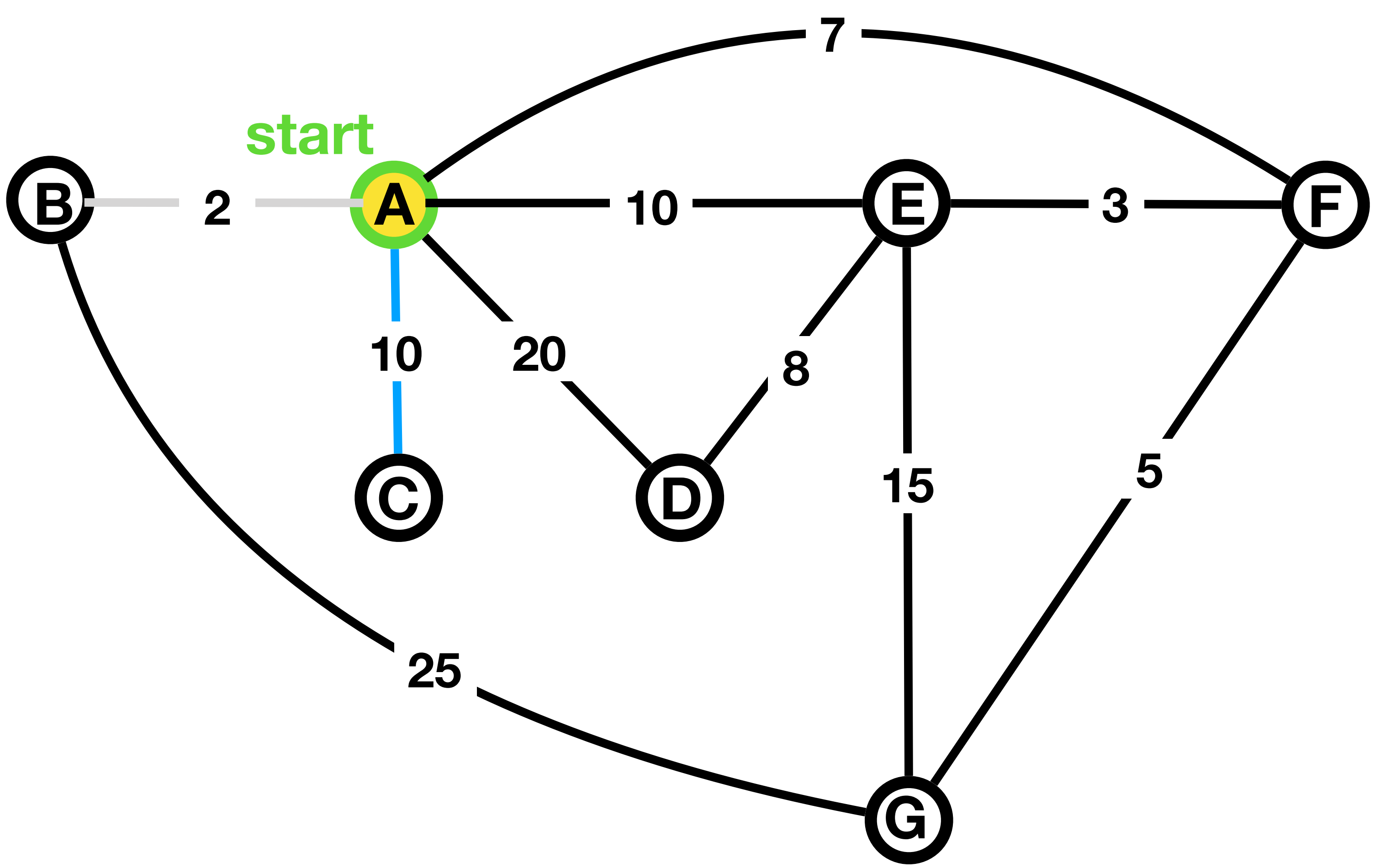
V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	F
C	∞		F
D	∞		F
E	∞		F
F	∞		F
G	∞		F

Is C known ? <u>No</u>

$\text{distance}_i + \text{weight}_{i,j} < \text{distance}_j \quad = \quad (0) + (10) < (\infty)$?

Dijkstra's Shortest Path

current, *i*	A
next, *j*	C



V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	F
C	10	A	F
D	∞		F
E	∞		F
F	∞		F
G	∞		F

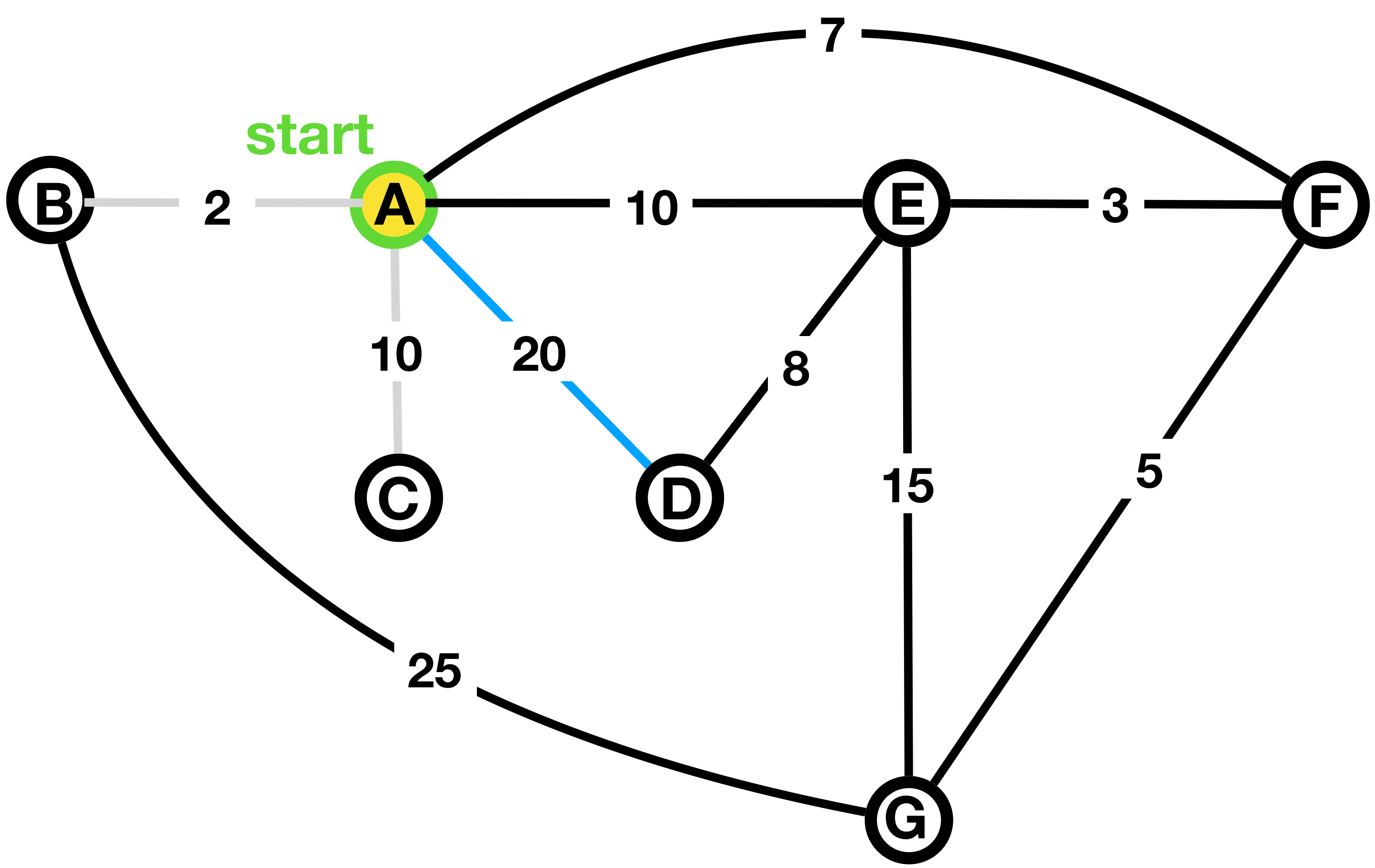
Is C known ? <u>No</u>

$distance_i + weight_{i,j} < distance_j$ = (0) + (10) < (∞) ? <u>Yes</u>

=> update $distance_j$ = (0) + (10) and $previous_j$ = current = A

Dijkstra’s Shortest Path

current, *i*	A
next, *j*	D



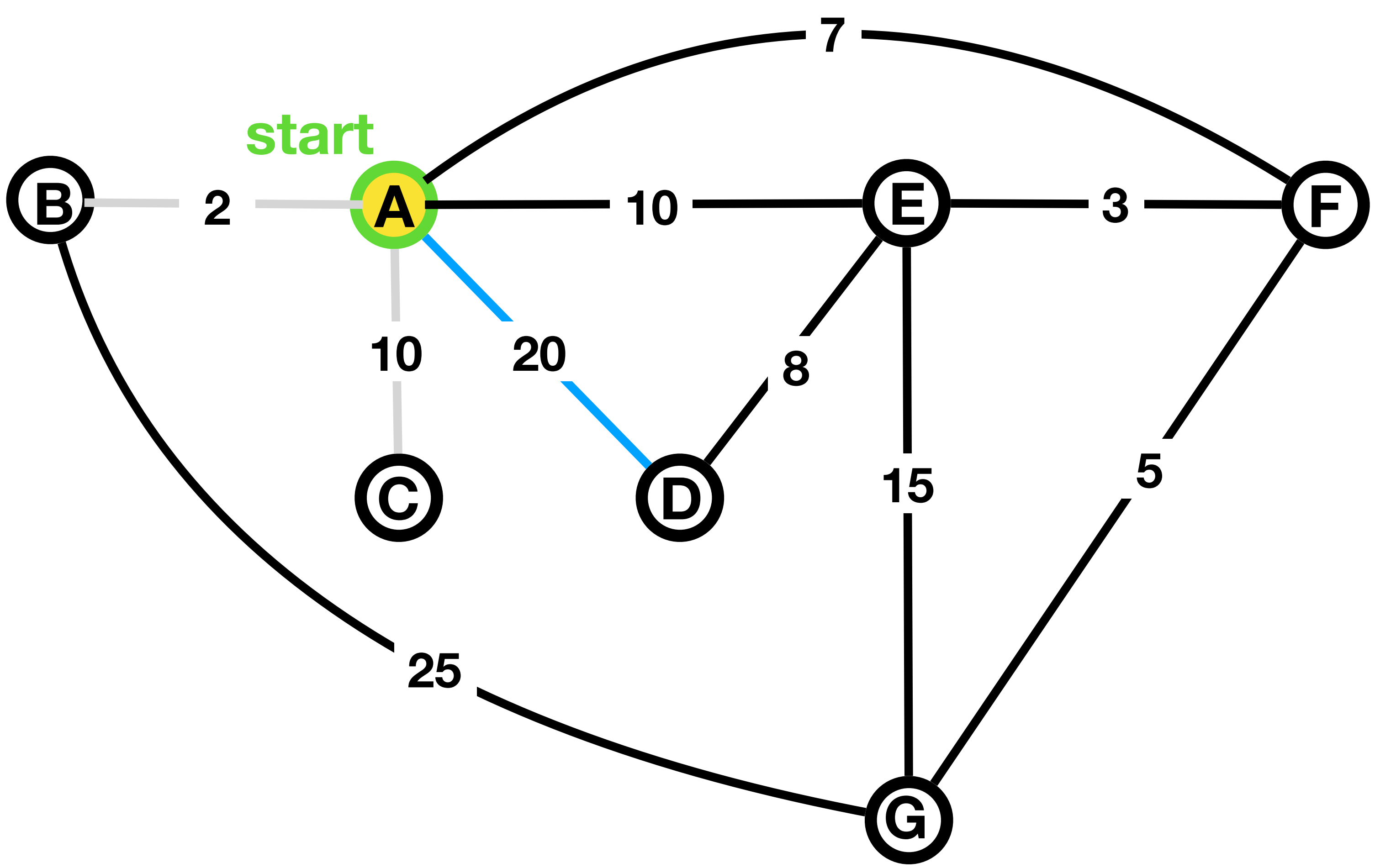
V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	F
C	10	A	F
D	∞		F
E	∞		F
F	∞		F
G	∞		F

Is D known ? <u>No</u>

$$\text{distance}_i + \text{weight}_{i,j} < \text{distance}_j \;=\; (0) + (20) < (\infty)\ ?$$

Dijkstra’s Shortest Path

current, *i*	A
next, *j*	D



V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	F
C	10	A	F
D	20	A	F
E	∞		F
F	∞		F
G	∞		F

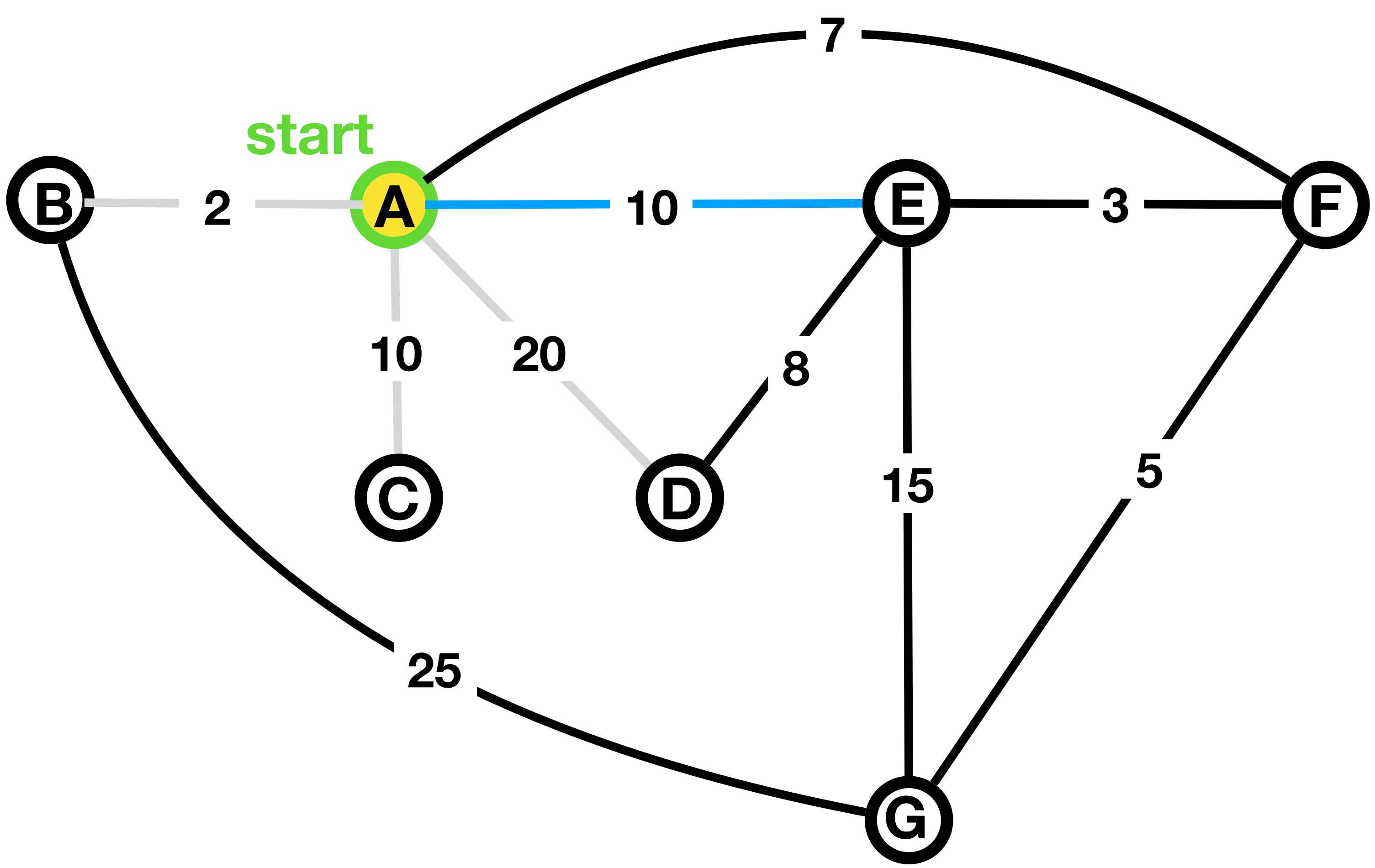
Is D known ? <u>No</u>

$\text{distance}_i + \text{weight}_{i,j} < \text{distance}_j$ = (0) + (20) < (∞) ? <u>Yes</u>

=> update distance_j = (0) + (20) and previous_j = current = A

Dijkstra's Shortest Path

current, *i*	A
next, *j*	E



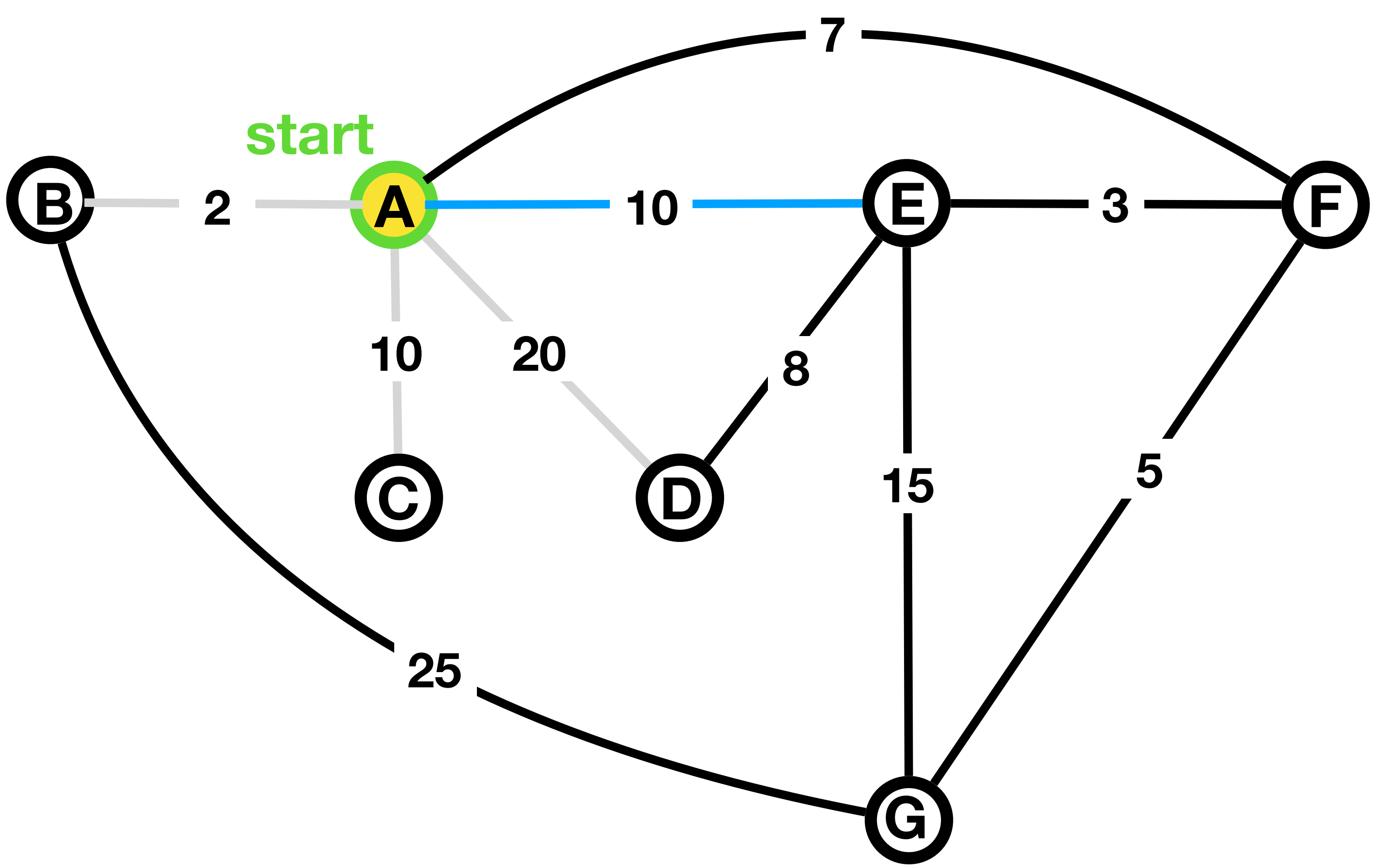
V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	F
C	10	A	F
D	20	A	F
E	∞		F
F	∞		F
G	∞		F

Is E known ? <u>No</u>

$\text{distance}_i + \text{weight}_{i,j} < \text{distance}_j \;=\; (0) + (10) < (\infty)$?

Dijkstra's Shortest Path

current, *i*	A
next, *j*	E



V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	F
C	10	A	F
D	20	A	F
E	10	A	F
F	∞		F
G	∞		F

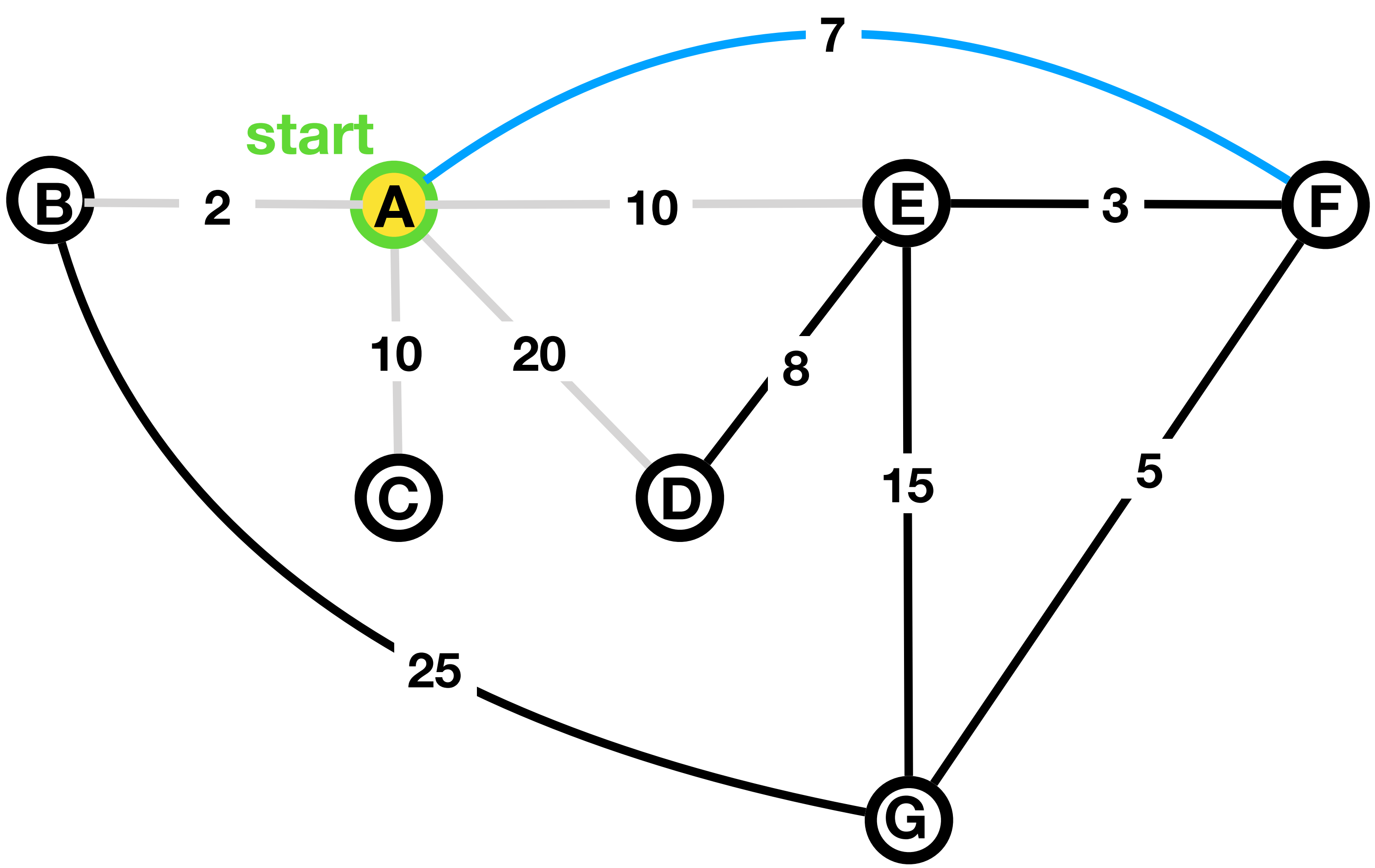
Is E known ? <u>No</u>

$distance_i + weight_{i,j} < distance_j$ = (0) + (10) < (∞) ? <u>Yes</u>

=> update $distance_j$ = (0) + (10) and $previous_j$ = current = A

Dijkstra's Shortest Path

current, *i*	A
next, *j*	F



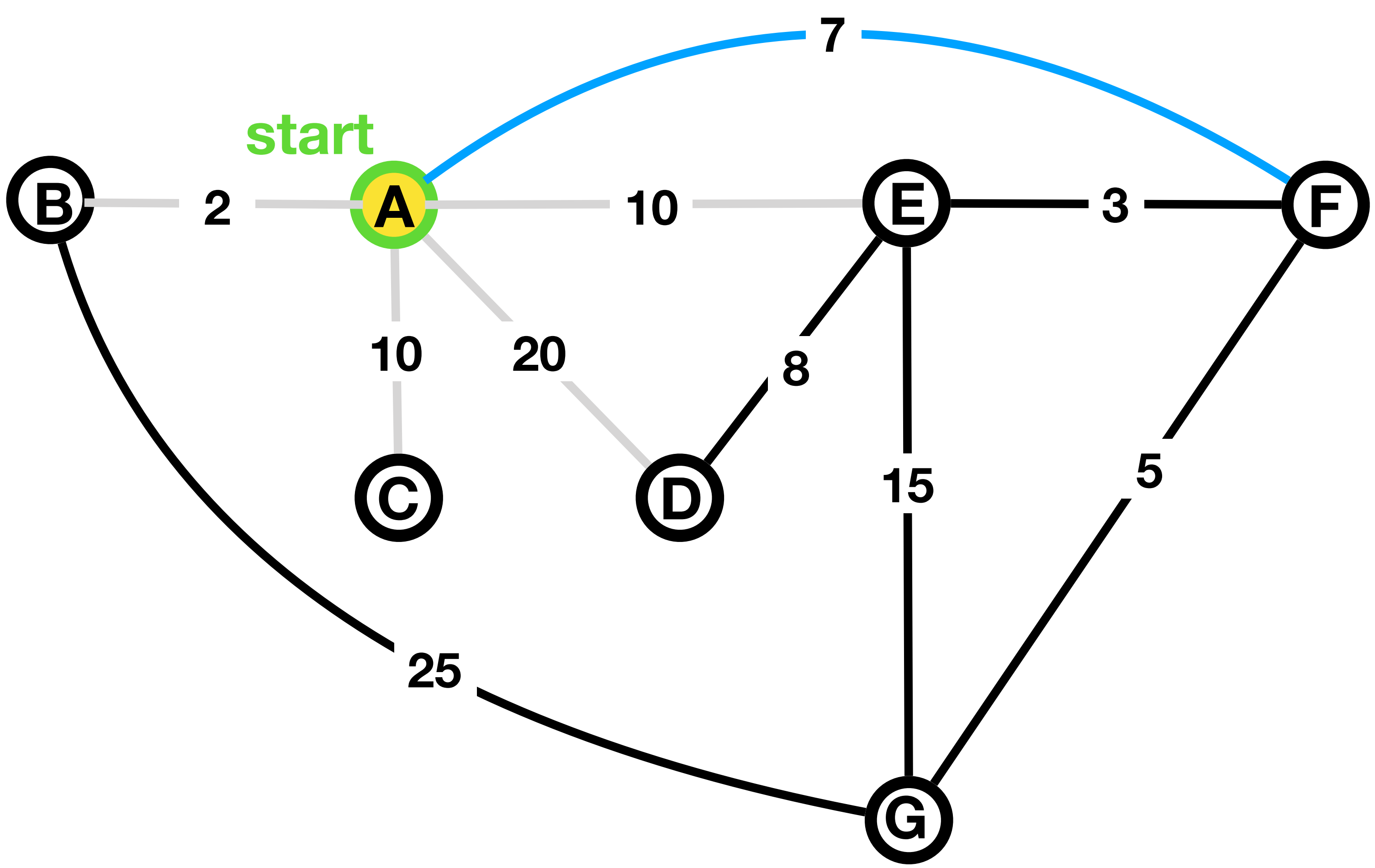
V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	F
C	10	A	F
D	20	A	F
E	10	A	F
F	∞		F
G	∞		F

Is F known ? <u>No</u>

$distance_i + weight_{i,j} < distance_j \;\; = \;\; (0) + (7) < (\infty)$?

Dijkstra's Shortest Path

current, *i*	A
next, *j*	F



V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	F
C	10	A	F
D	20	A	F
E	10	A	F
F	7	A	F
G	∞		F

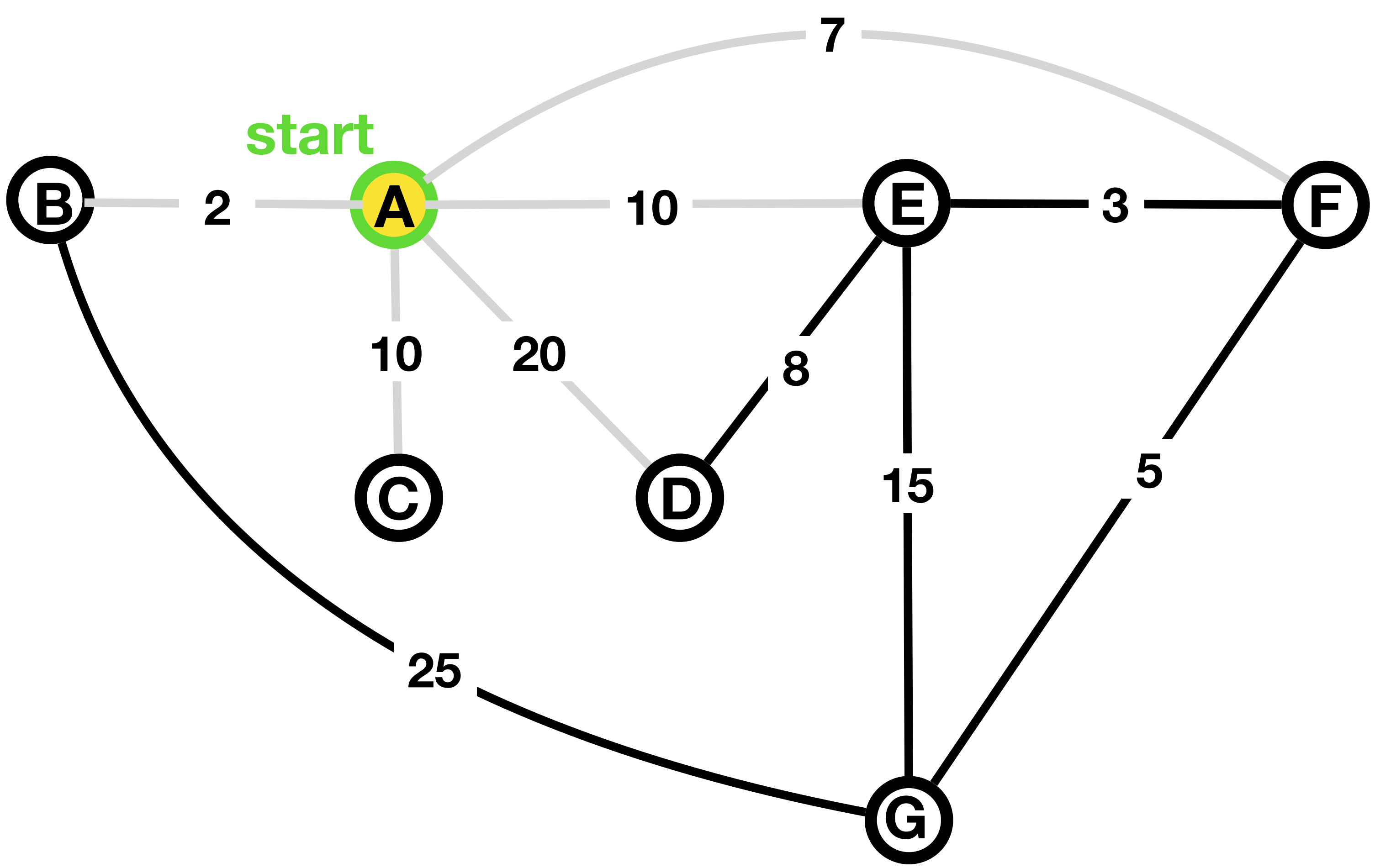
Is F known ? No

$\text{distance}_i + \text{weight}_{i,j} < \text{distance}_j$ = (0) + (7) < (∞) ? Yes

=> update distance_j = (0) + (7) and previous_j = current = A

Dijkstra's Shortest Path

current, *i*	**A**
next, *j*	

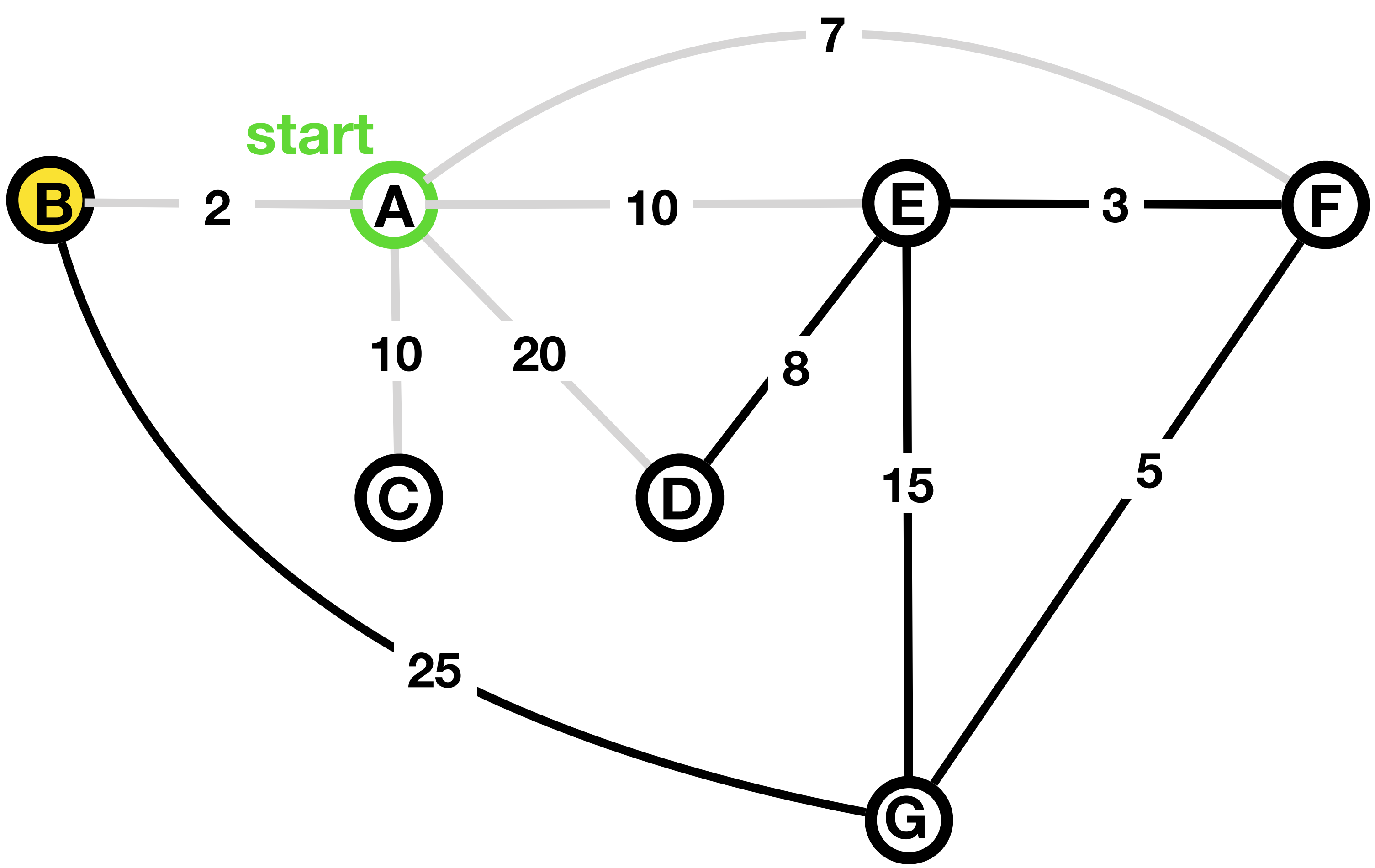


V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	F
C	10	A	F
D	20	A	F
E	10	A	F
F	7	A	F
G	∞		F

update *current* to node with smallest distance

Dijkstra’s Shortest Path

current, *i*	B
next, *j*	

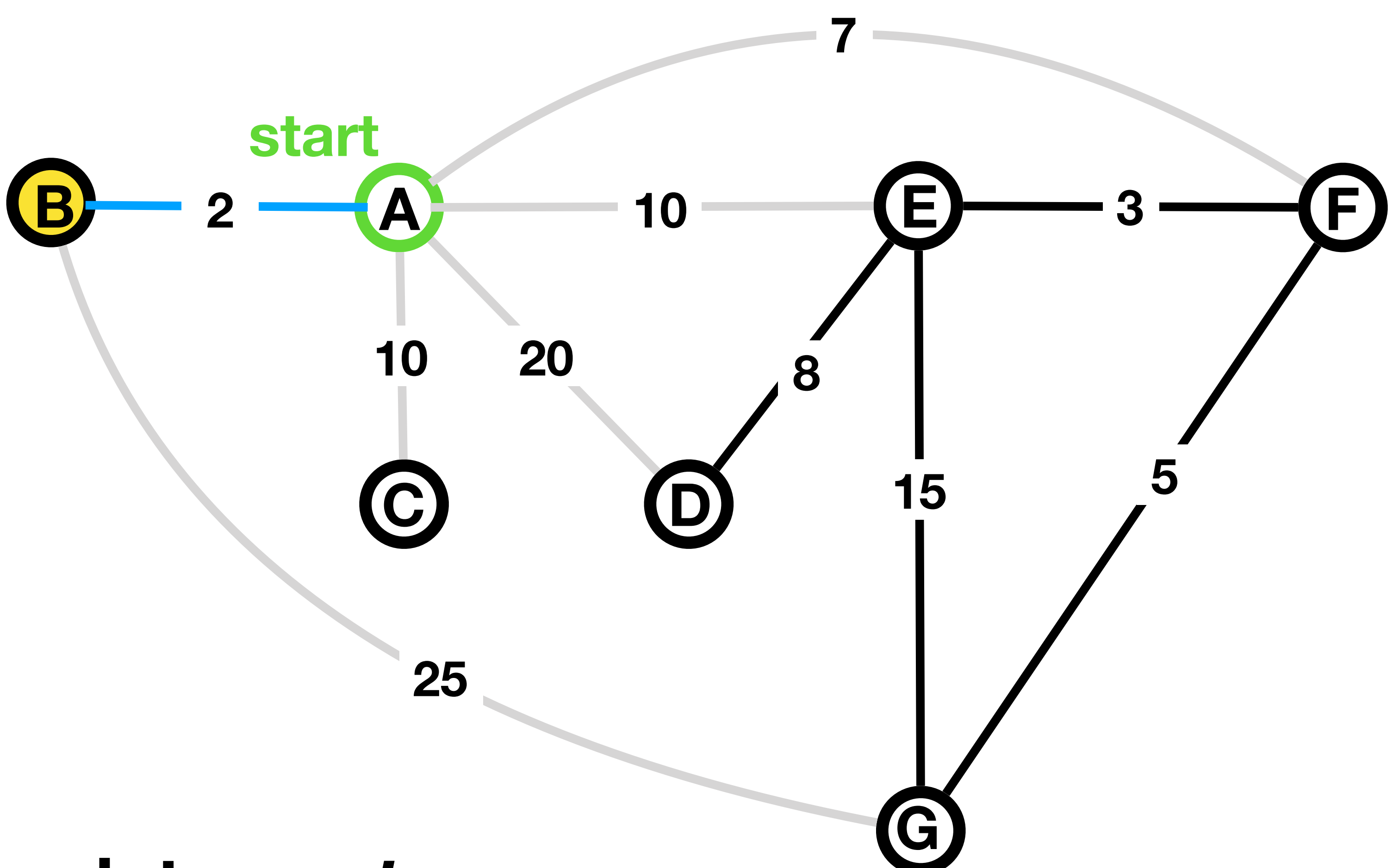


V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	F
C	10	A	F
D	20	A	F
E	10	A	F
F	7	A	F
G	∞		F

update *current* to node with smallest distance

Dijkstra's Shortest Path

current, *i*	B
next, *j*	A



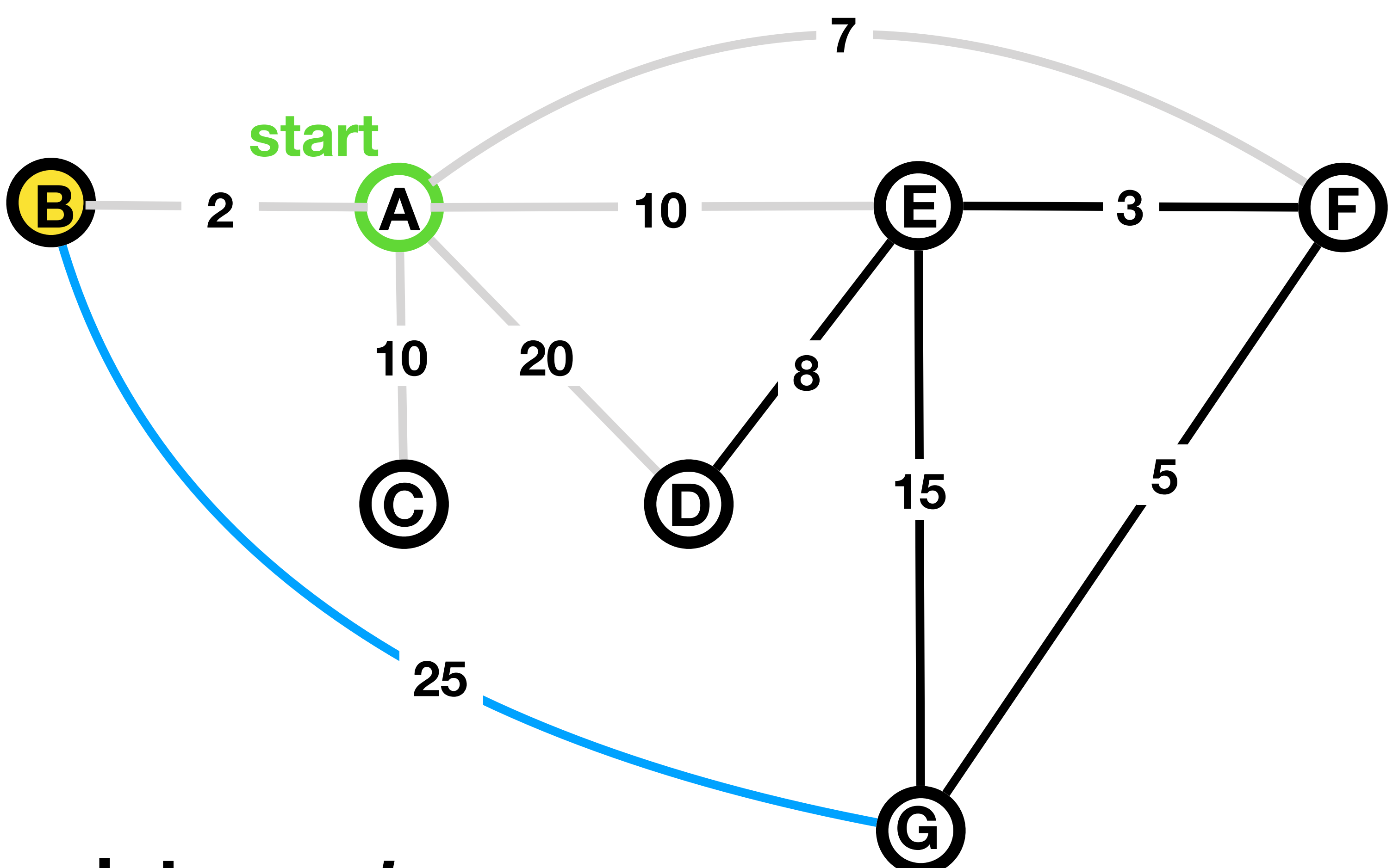
V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	F
D	20	A	F
E	10	A	F
F	7	A	F
G	∞		F

update *next*.

Is A known ? <u>Yes.</u>

Dijkstra’s Shortest Path

current, *i*	B
next, *j*	G



V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	F
D	20	A	F
E	10	A	F
F	7	A	F
G	∞		F

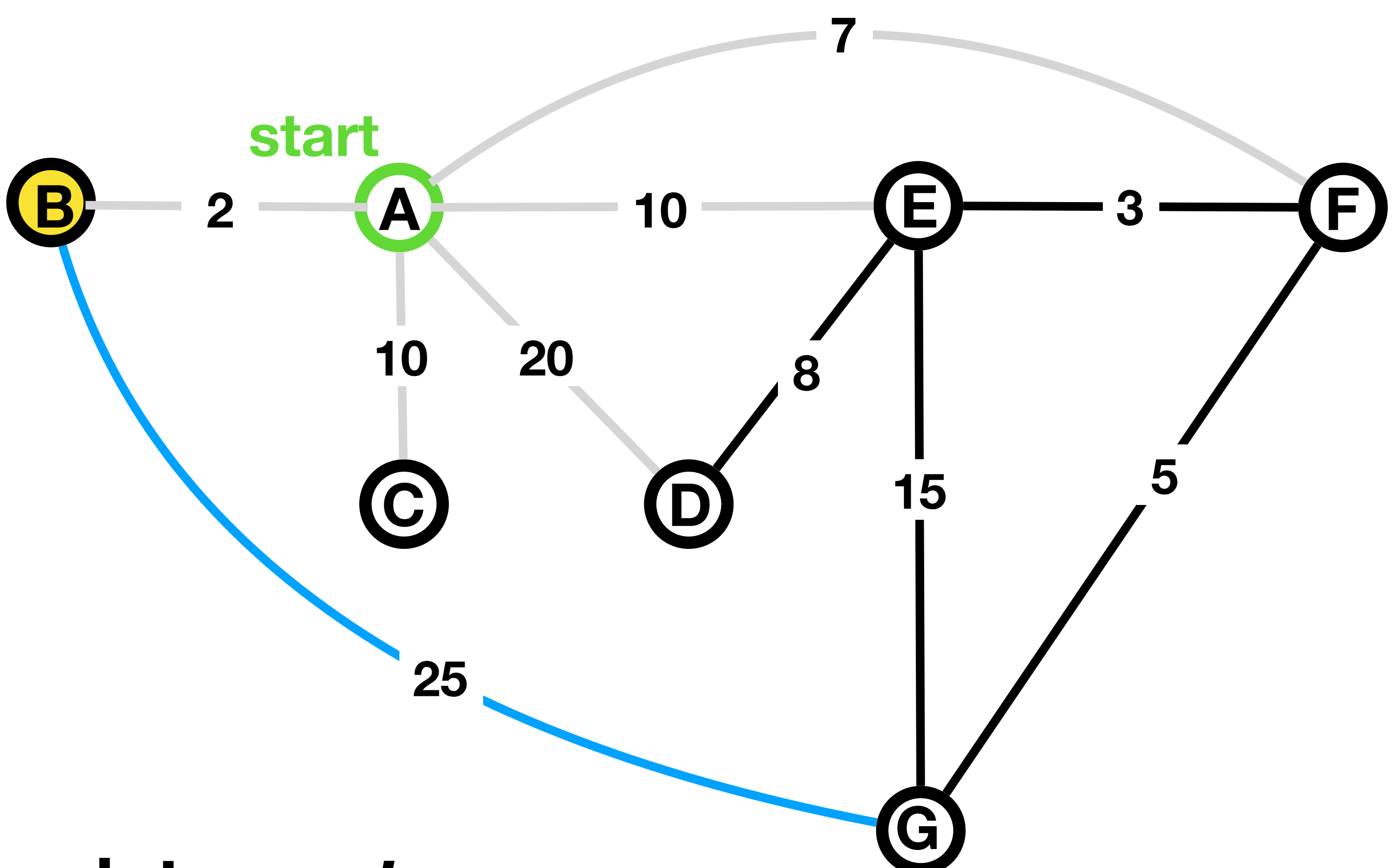
update *next*.

Is G known ? <u>No</u>

$\text{distance}_i + \text{weight}_{i,j} < \text{distance}_j \;=\; (2) + (25) < (\infty)$?

Dijkstra’s Shortest Path

current, *i*	B
next, *j*	G



V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	F
D	20	A	F
E	10	A	F
F	7	A	F
G	27	B	F

update *next*.

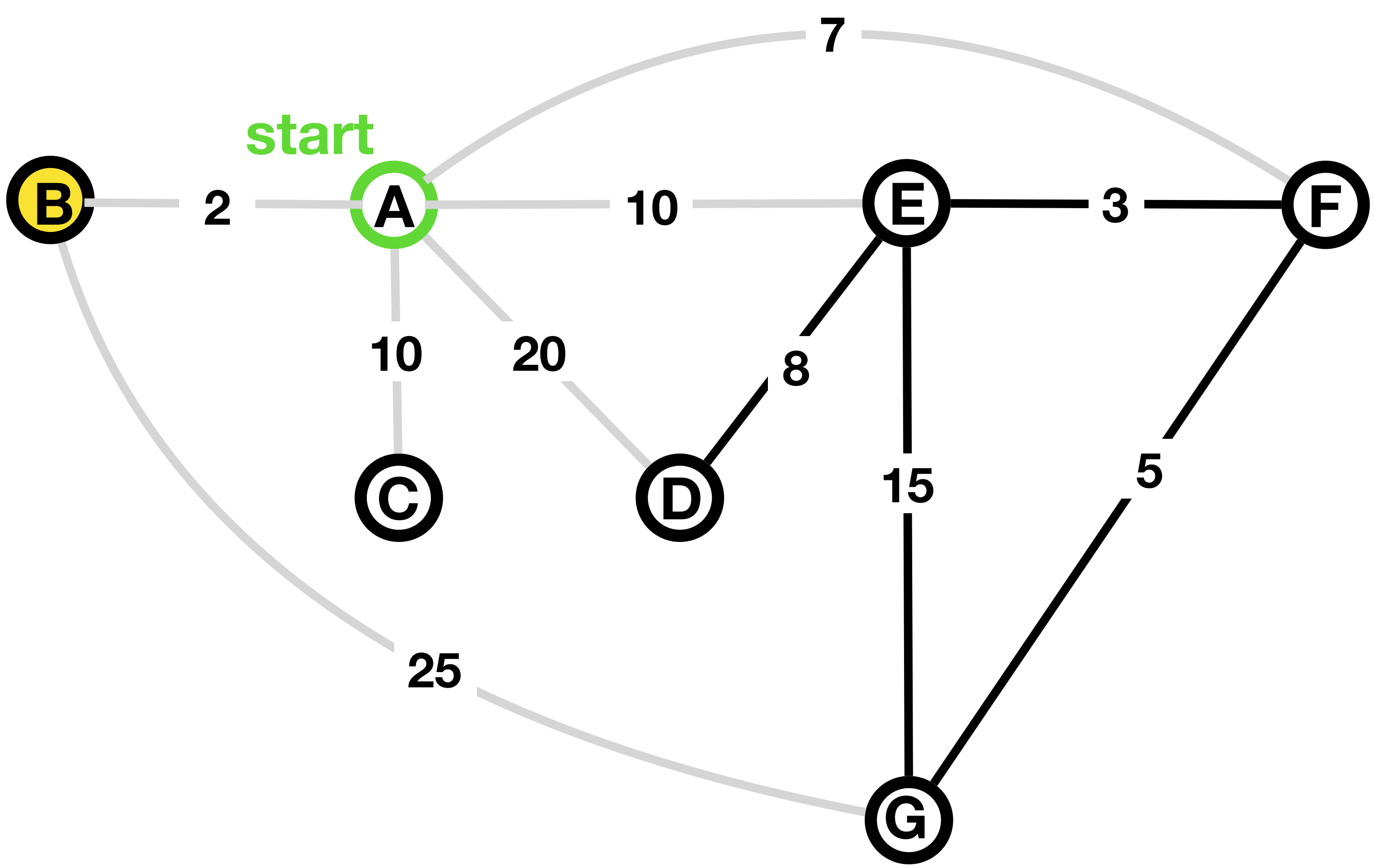
Is G known ? <u>No</u>

$\text{distance}_i + \text{weight}_{i,j} < \text{distance}_j$ = (2) + (25) < (∞) ? <u>Yes</u>

=> update distance_j = (2) + (25) and previous_j = current = B

Dijkstra's Shortest Path

current, *i*	B
next, *j*	

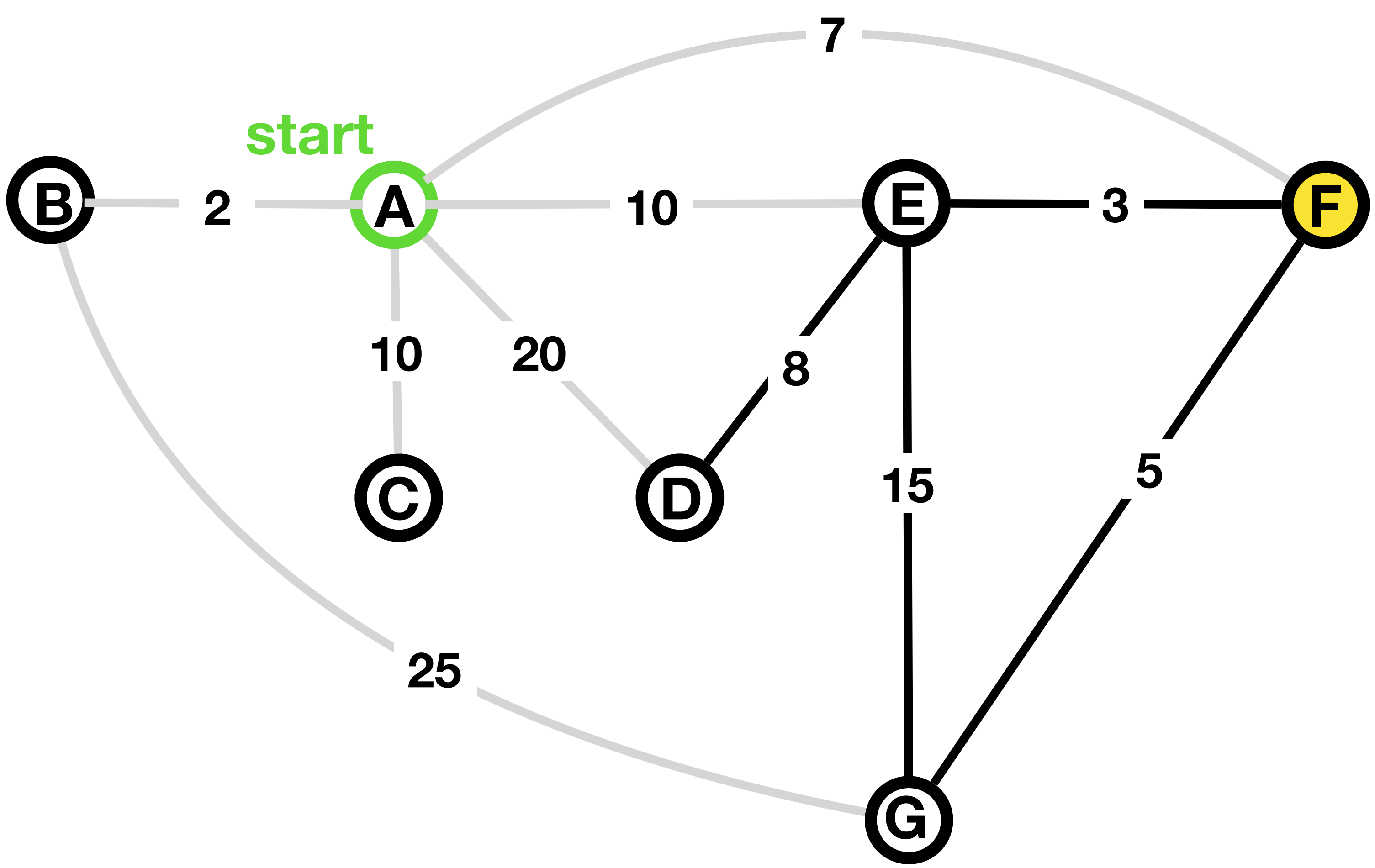


V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	F
D	20	A	F
E	10	A	F
F	7	A	F
G	27	B	F

update *current* to node with smallest distance

Dijkstra's Shortest Path

current, *i*	F
next, *j*	

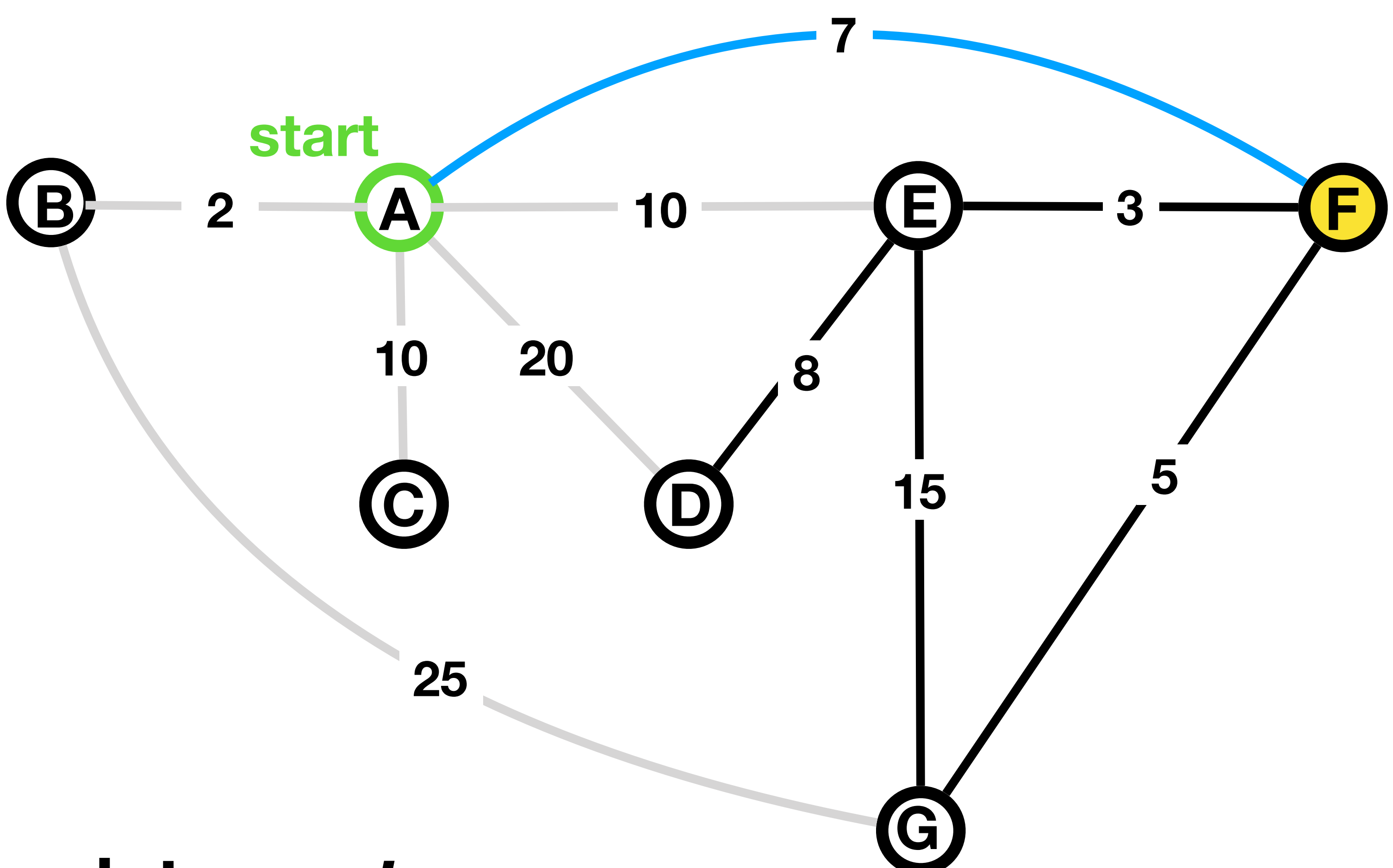


V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	F
D	20	A	F
E	10	A	F
F	7	A	F
G	27	B	F

update *current* to node with smallest distance

Dijkstra’s Shortest Path

current, *i*	F
next, *j*	A



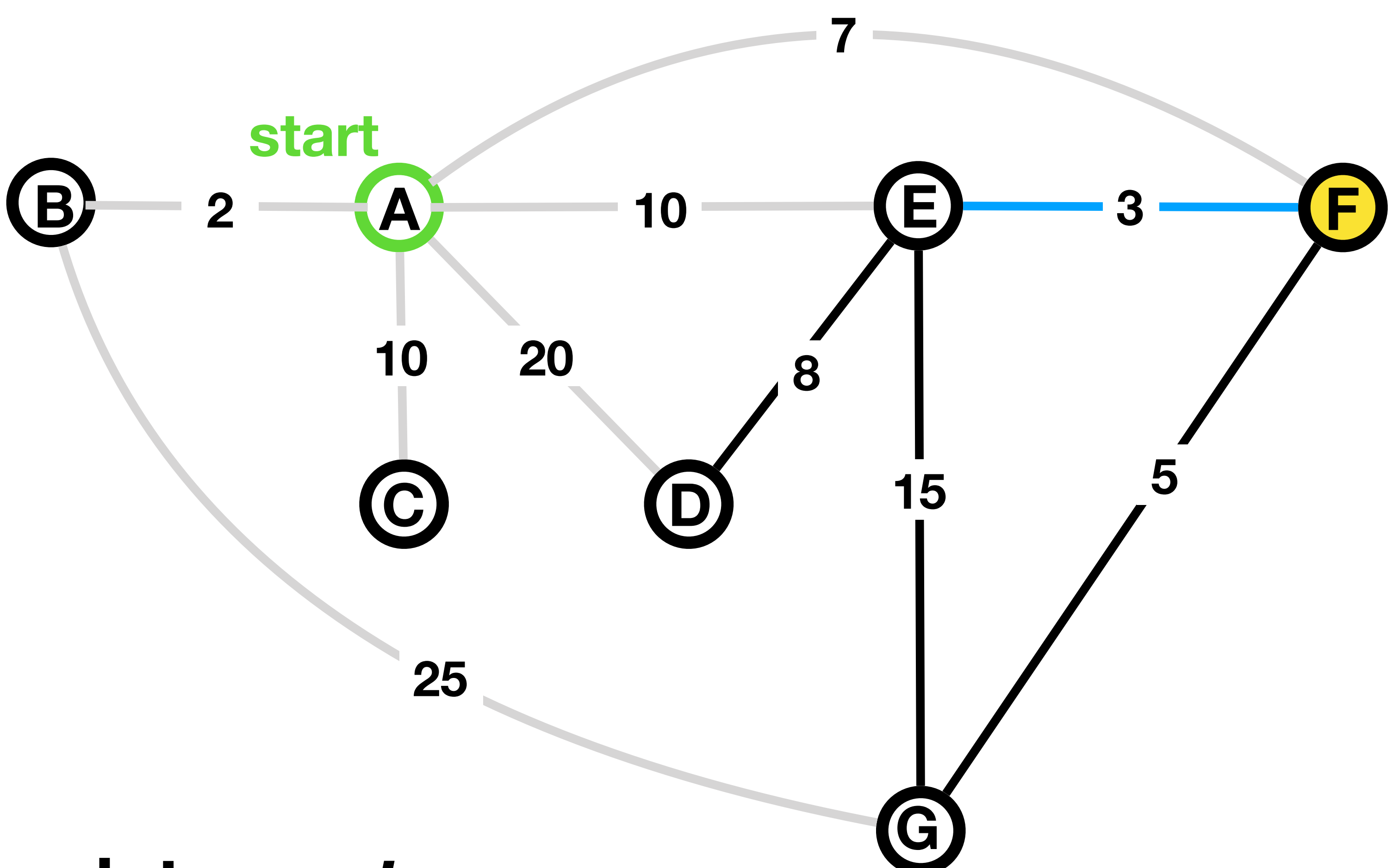
V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	F
D	20	A	F
E	10	A	F
F	7	A	T
G	27	B	F

update *next.*

Is A known ? <u>Yes.</u>

Dijkstra’s Shortest Path

current, *i*	F
next, *j*	E



V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	F
D	20	A	F
E	10	A	F
F	7	A	T
G	27	B	F

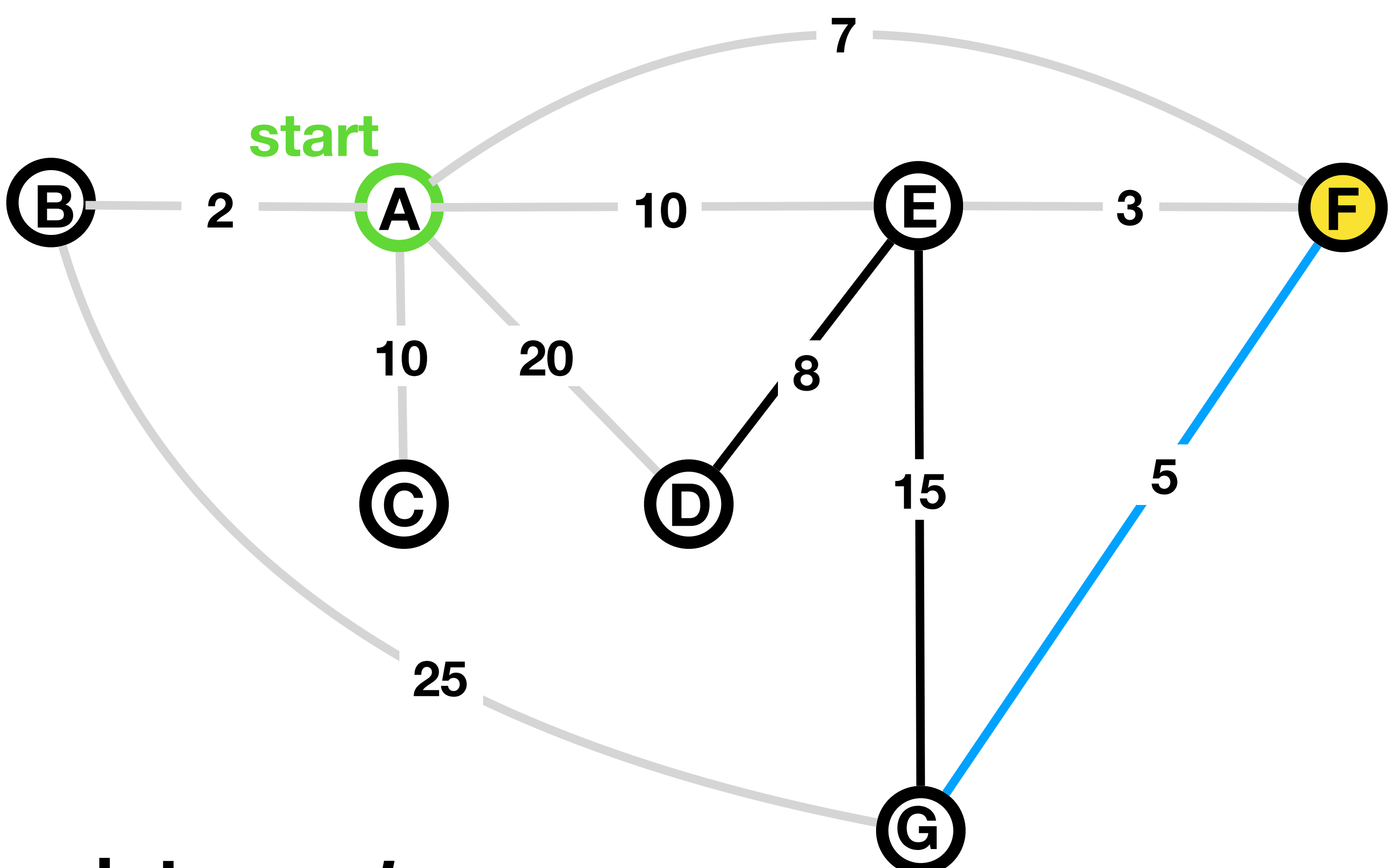
update *next*.

Is E known ? No.

$distance_i + weight_{i,j} < distance_j$ = (7) + (3) < (10) ? No

Dijkstra's Shortest Path

current, *i*	F
next, *j*	G



V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	F
D	20	A	F
E	10	A	F
F	7	A	T
G	27	B	F

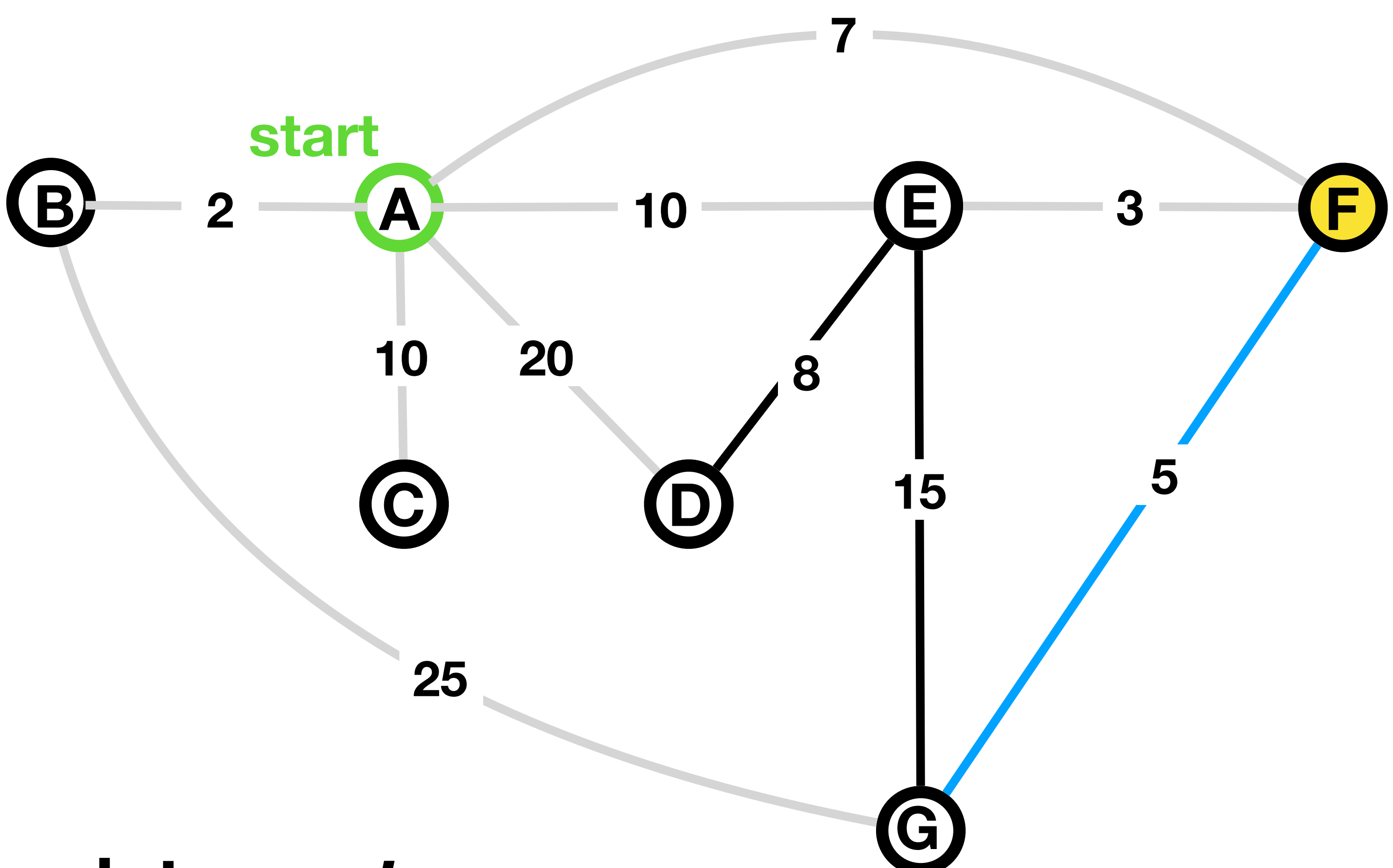
update *next.*

Is G known ? <u>No.</u>

distance$_i$ + weight$_{i,j}$ < distance$_j$ = (7) + (5) < (27) ?

Dijkstra’s Shortest Path

current, *i*	F
next, *j*	G



V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	F
D	20	A	F
E	10	A	F
F	7	A	T
G	12	F	F

update *next*.

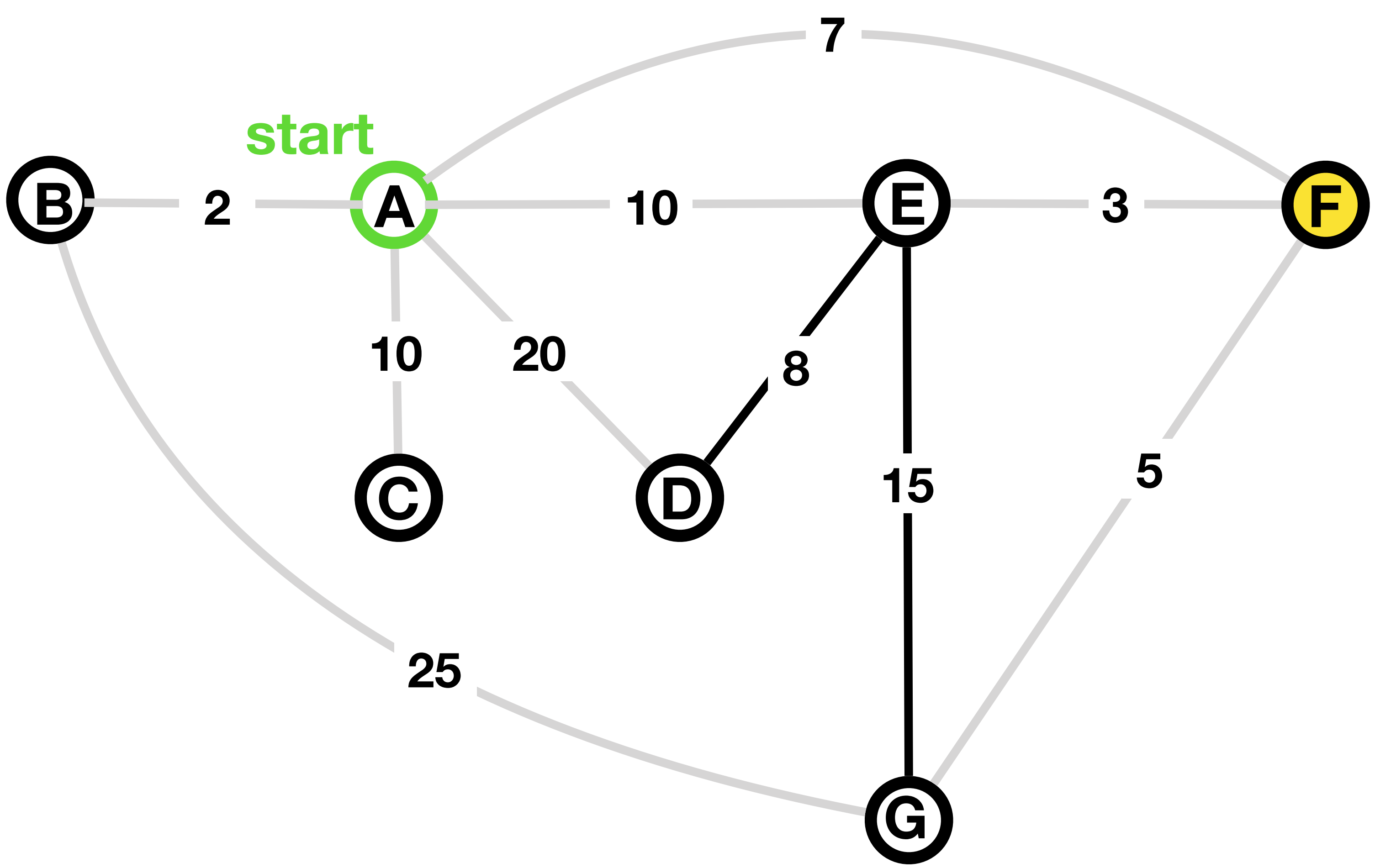
Is G known ? <u>No.</u>

$distance_i + weight_{i,j} < distance_j$ = (7) + (5) < (27) ? <u>Yes</u>

=> update $distance_j$ = (7) + (5) and $previous_j$ = current = F

Dijkstra's Shortest Path

current, *i*	F
next, *j*	

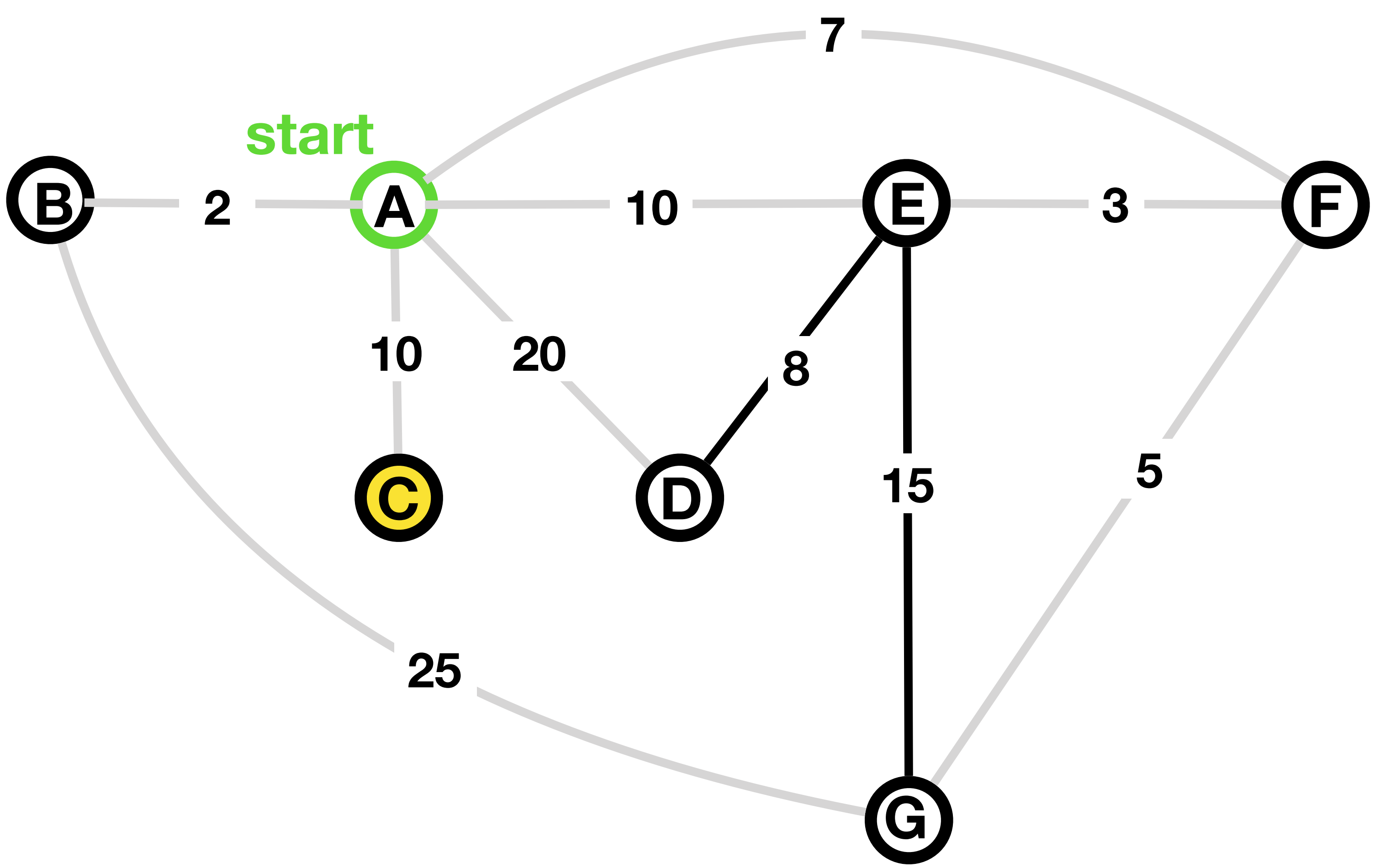


V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	F
D	20	A	F
E	10	A	F
F	7	A	T
G	12	F	F

update *current* to node with smallest distance

Dijkstra’s Shortest Path

current, *i*	C
next, *j*	

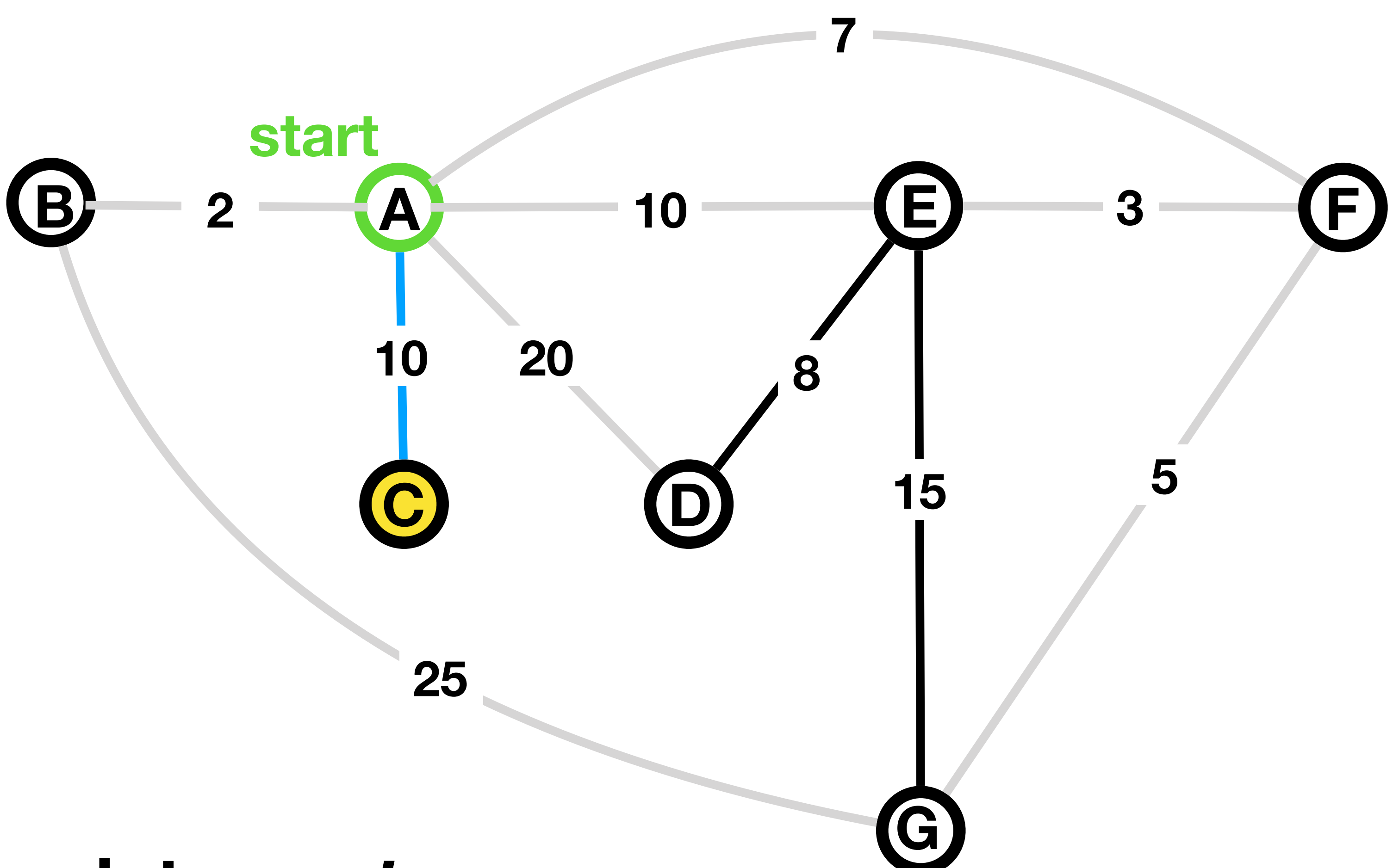


V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	F
D	20	A	F
E	10	A	F
F	7	A	T
G	12	F	F

update *current* to node with smallest distance

Dijkstra’s Shortest Path

current, *i*	C
next, *j*	



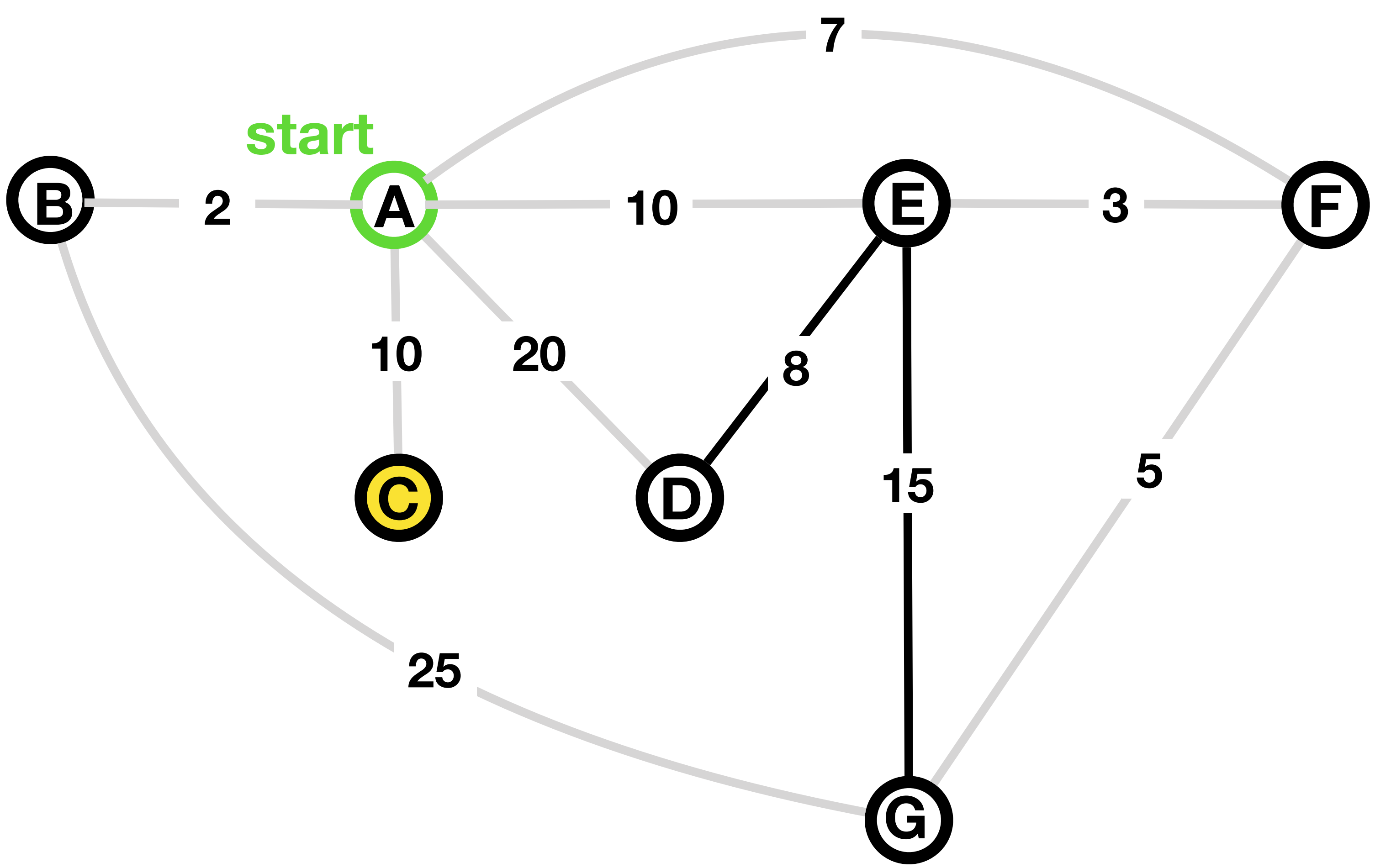
update *next.*

Is A known ? <u>Yes.</u>

V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	20	A	F
E	10	A	F
F	7	A	T
G	12	F	F

Dijkstra's Shortest Path

current, *i*	C
next, *j*	

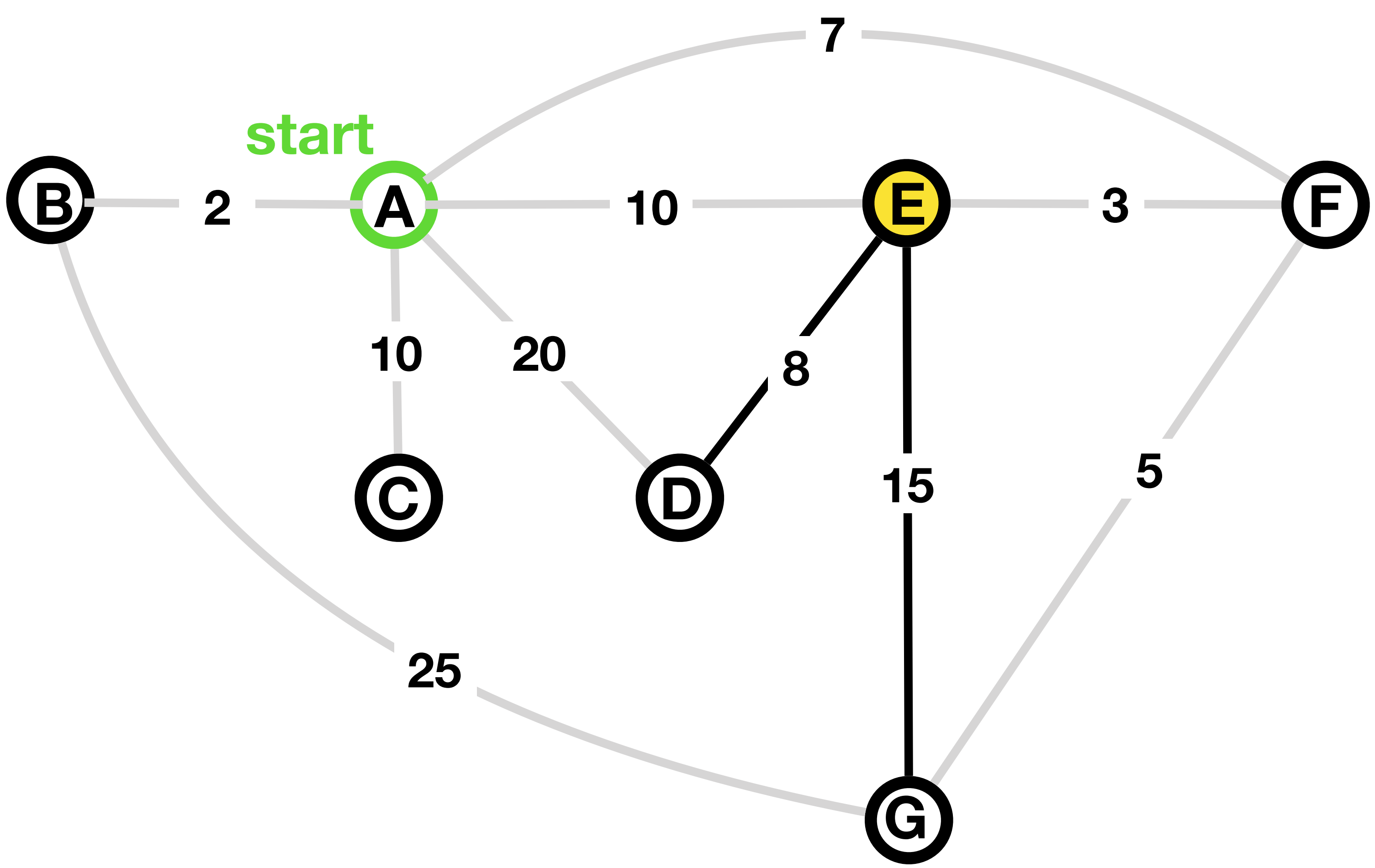


V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	20	A	F
E	10	A	F
F	7	A	T
G	12	F	F

update *current* to node with smallest distance

Dijkstra's Shortest Path

current, *i*	E
next, *j*	

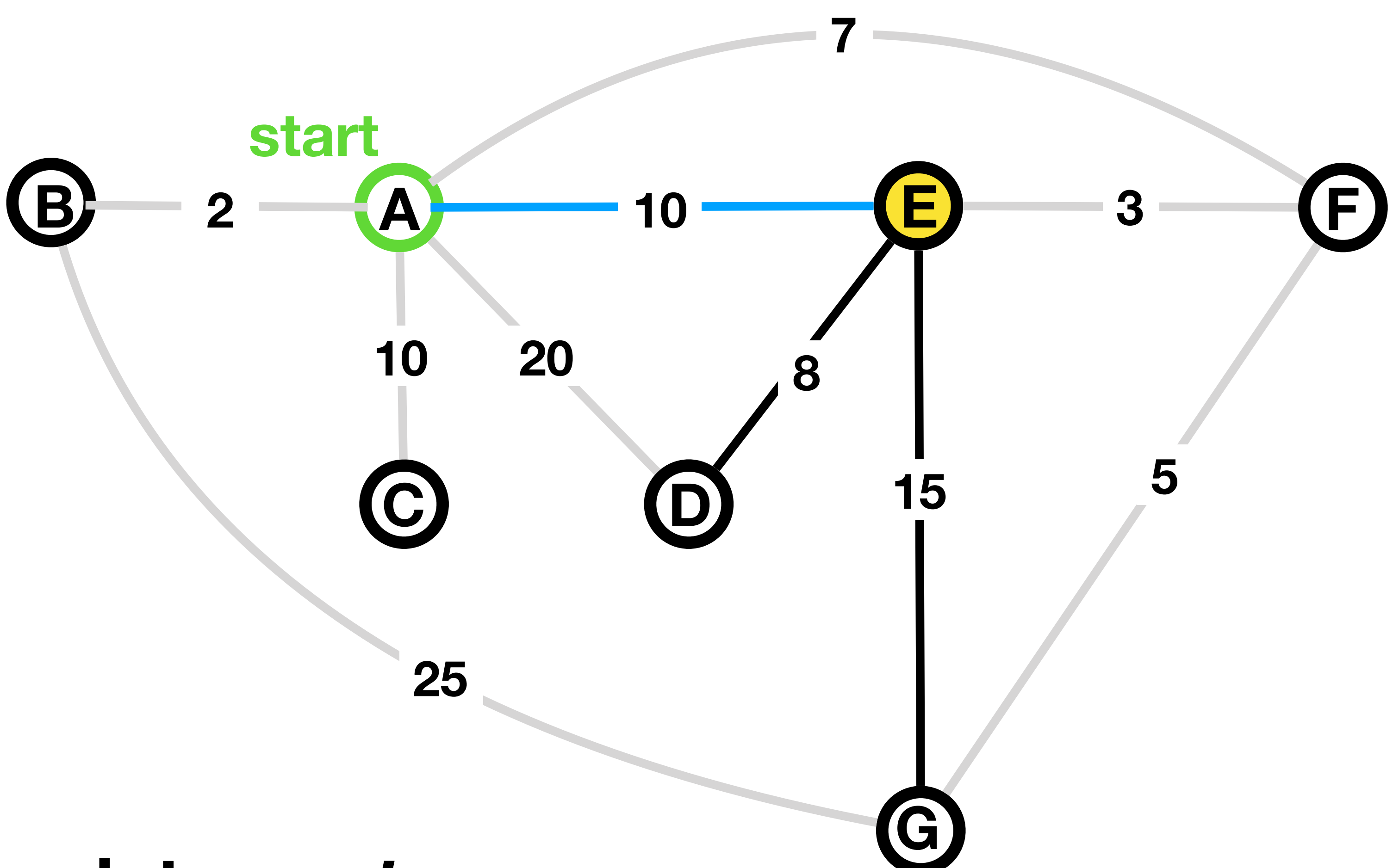


V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	20	A	F
E	10	A	F
F	7	A	T
G	12	F	F

update *current* to node with smallest distance

Dijkstra's Shortest Path

current, *i*	E
next, *j*	A



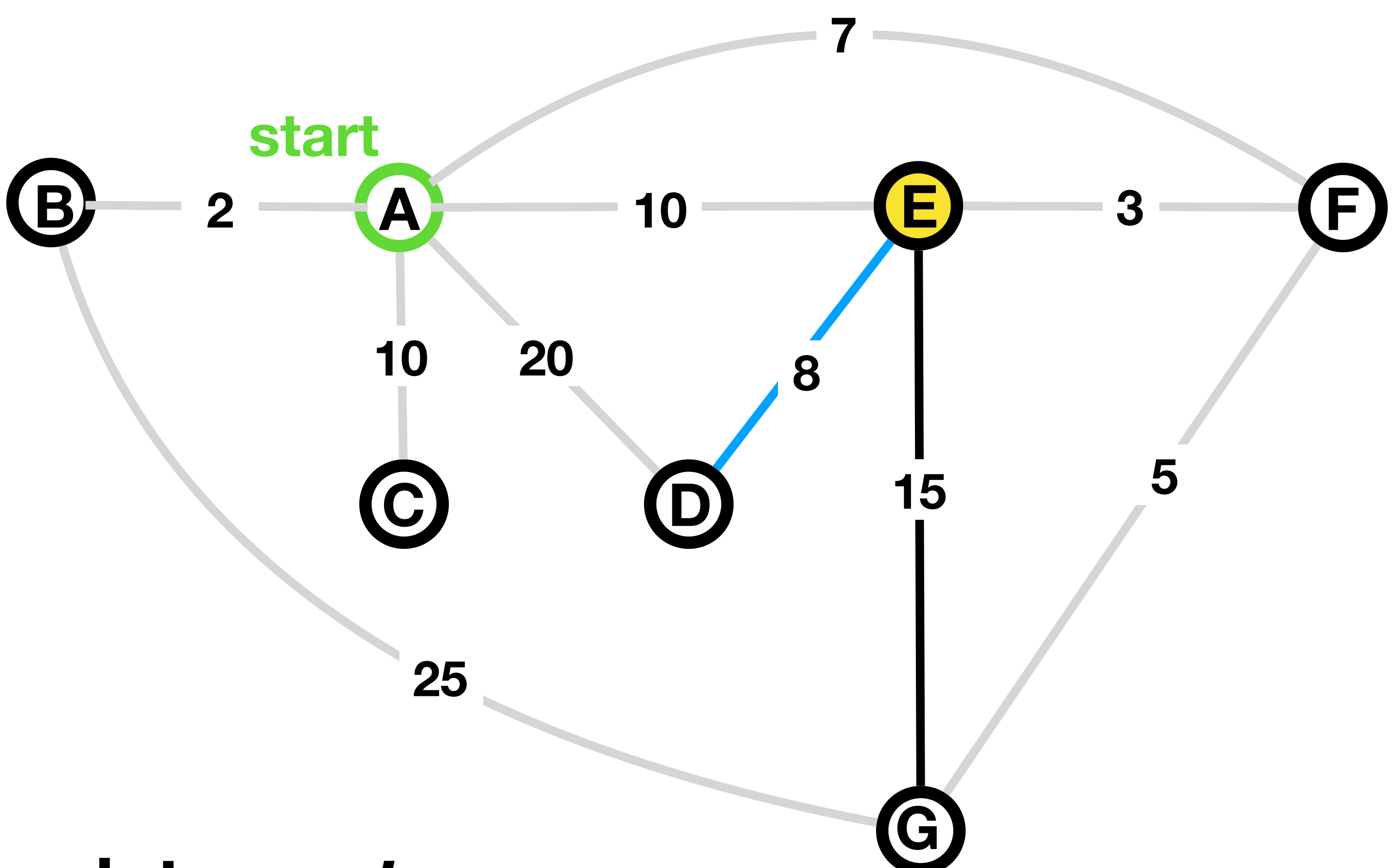
V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	20	A	F
E	10	A	T
F	7	A	T
G	12	F	F

update *next.*

Is A known ? Yes.

Dijkstra’s Shortest Path

current, *i*	**E**
next, *j*	**D**



V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	20	A	F
E	10	A	T
F	7	A	T
G	12	F	F

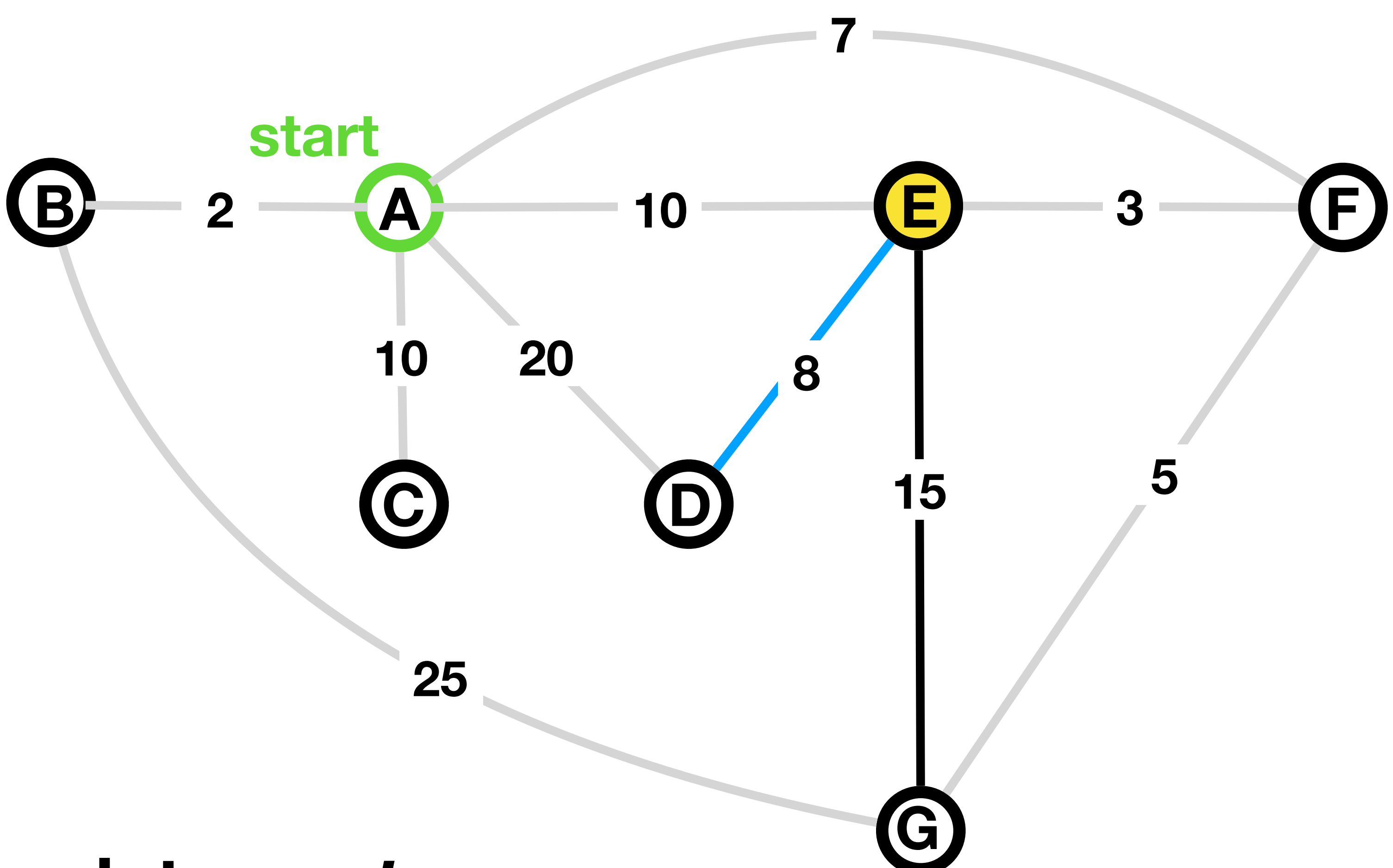
update *next*.

Is D known ? <u>No.</u>

$\text{distance}_i + \text{weight}_{i,j} < \text{distance}_j \;=\; (10) + (8) < (20)$?

Dijkstra's Shortest Path

current, *i*	E
next, *j*	D



V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	F
E	10	A	T
F	7	A	T
G	12	F	F

update *next*.

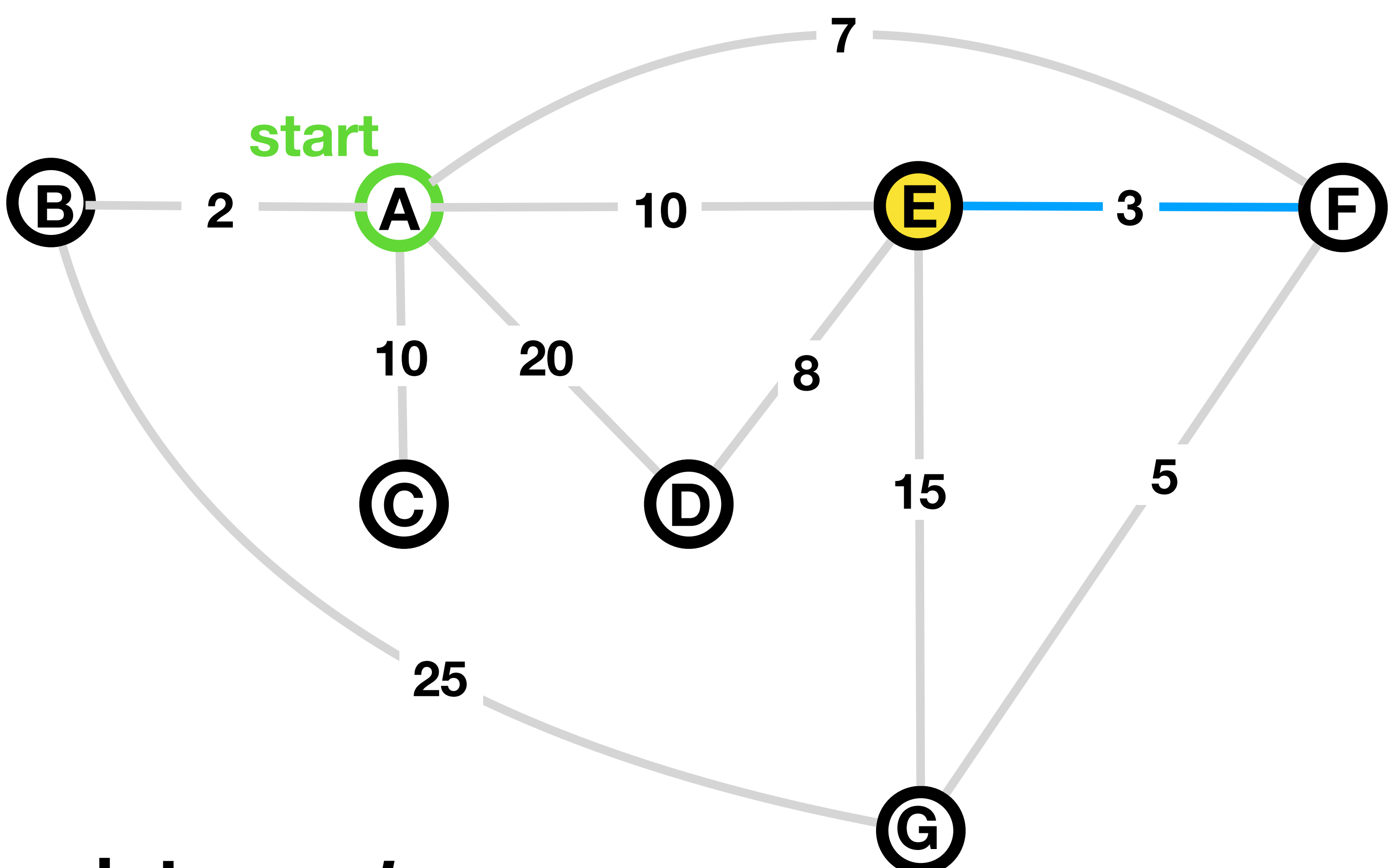
Is D known ? <u>No.</u>

distance$_i$ + weight$_{i,j}$ < distance$_j$ = (10) + (8) < (20) ? <u>Yes</u>

=> update distance$_j$ = (10) + (8) and previous$_j$ = current = E

Dijkstra's Shortest Path

current, *i*	E
next, *j*	F



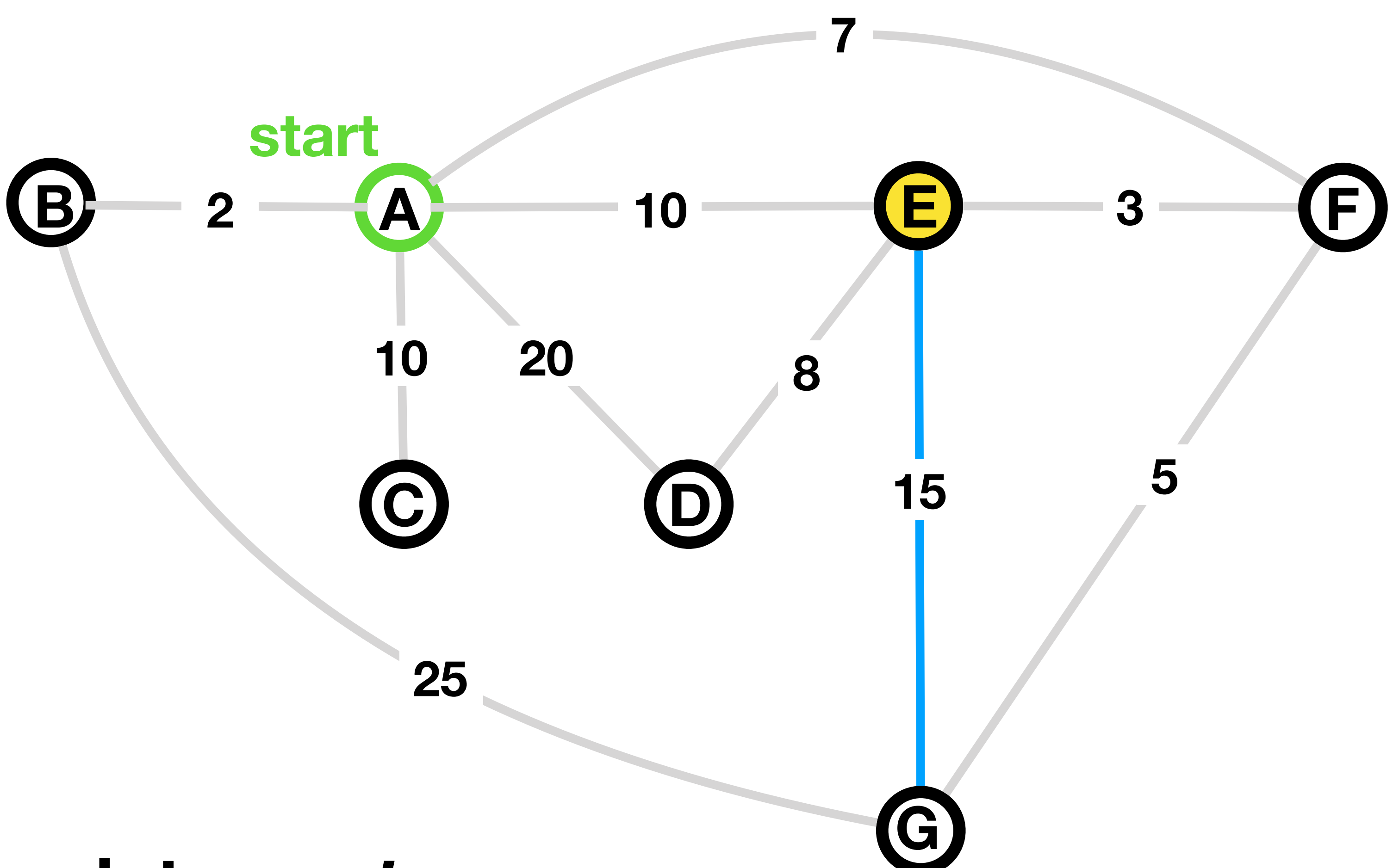
V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	F
E	10	A	T
F	7	A	T
G	12	F	F

update *next*.

Is F known ? <u>Yes.</u>

Dijkstra’s Shortest Path

current, *i*	E
next, *j*	G



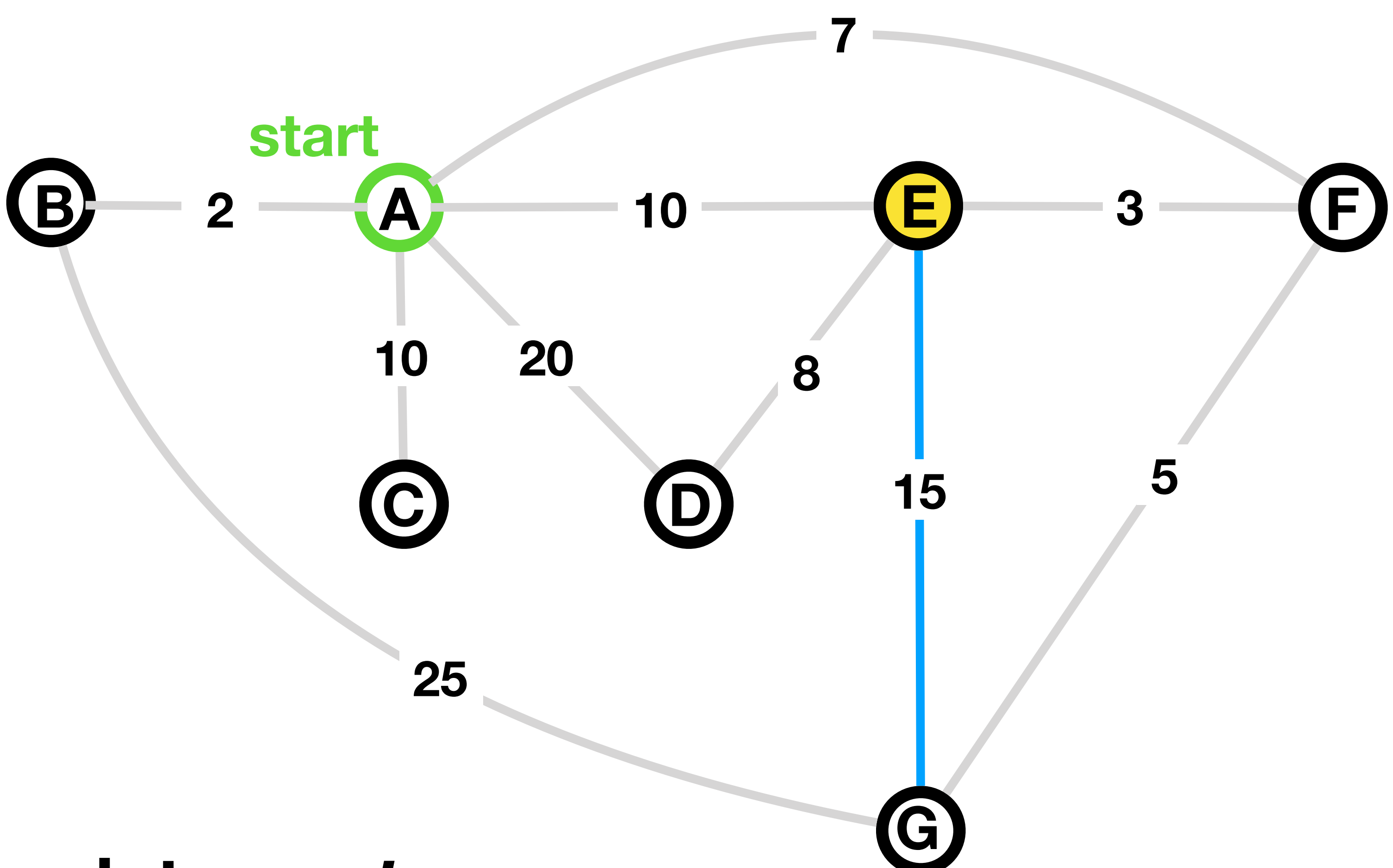
V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	F
E	10	A	T
F	7	A	T
G	12	F	F

update *next.*

Is G known ? No.

Dijkstra's Shortest Path

current, *i*	E
next, *j*	G



V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	F
E	10	A	T
F	7	A	T
G	12	F	F

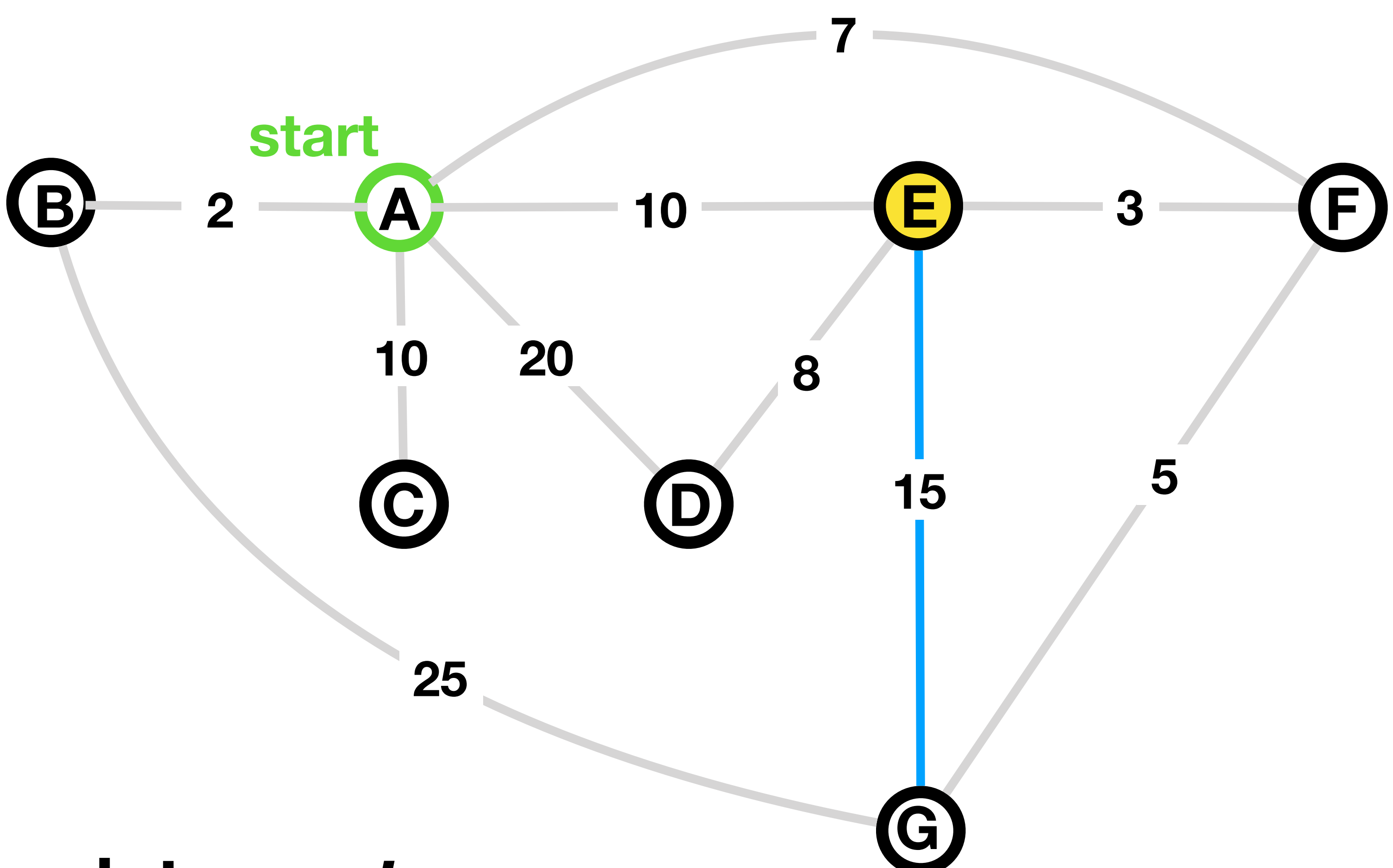
update *next.*

Is G known ? No.

$\text{distance}_i + \text{weight}_{i,j} < \text{distance}_j \quad = \quad (10) + (15) < (12)$?

Dijkstra's Shortest Path

current, *i*	E
next, *j*	G



V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	F
E	10	A	T
F	7	A	T
G	12	F	F

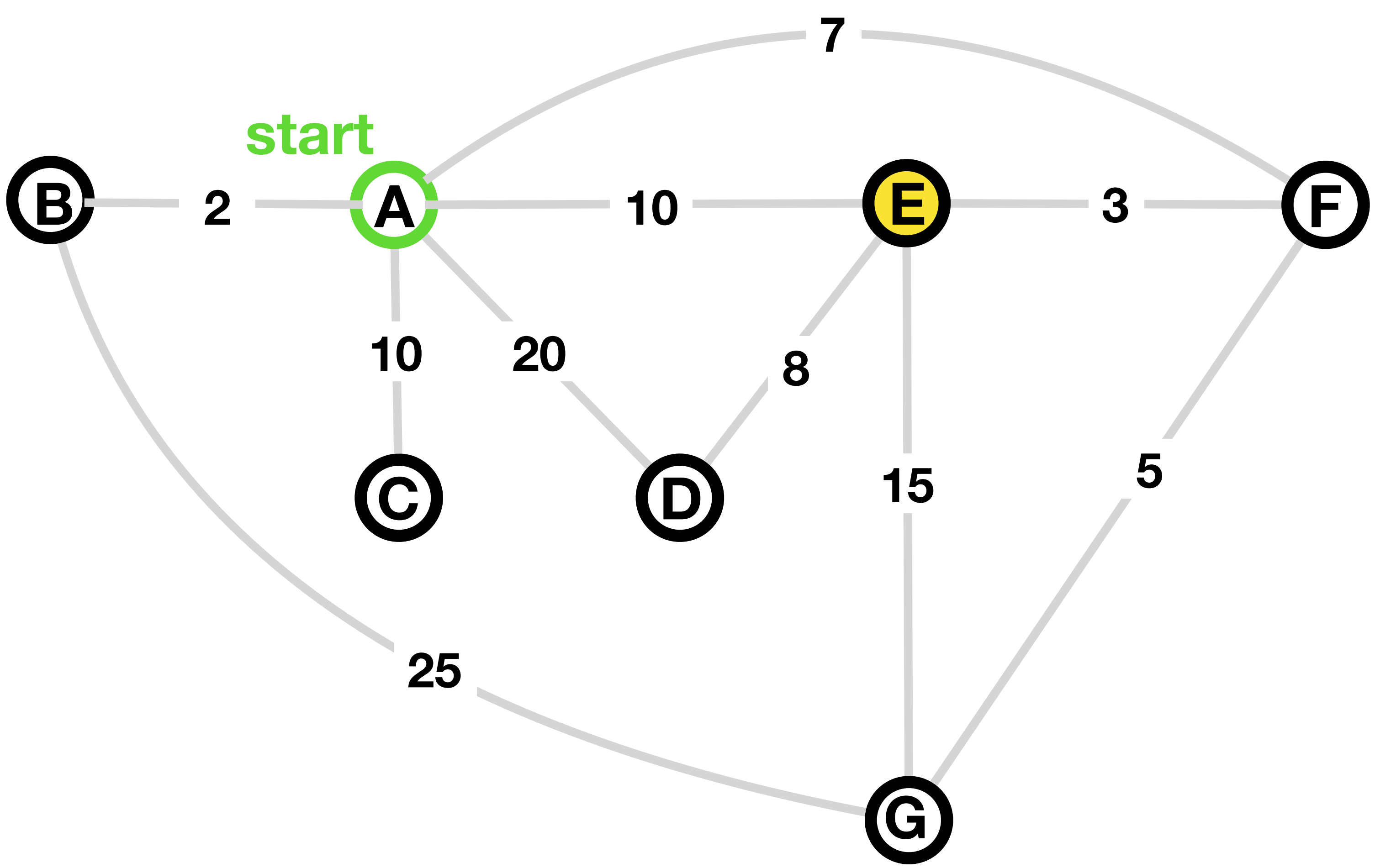
update *next.*

Is G known ? No.

$\text{distance}_i + \text{weight}_{i,j} < \text{distance}_j$ = (10) + (15) < (12) ? No

Dijkstra's Shortest Path

current, *i*	E
next, *j*	

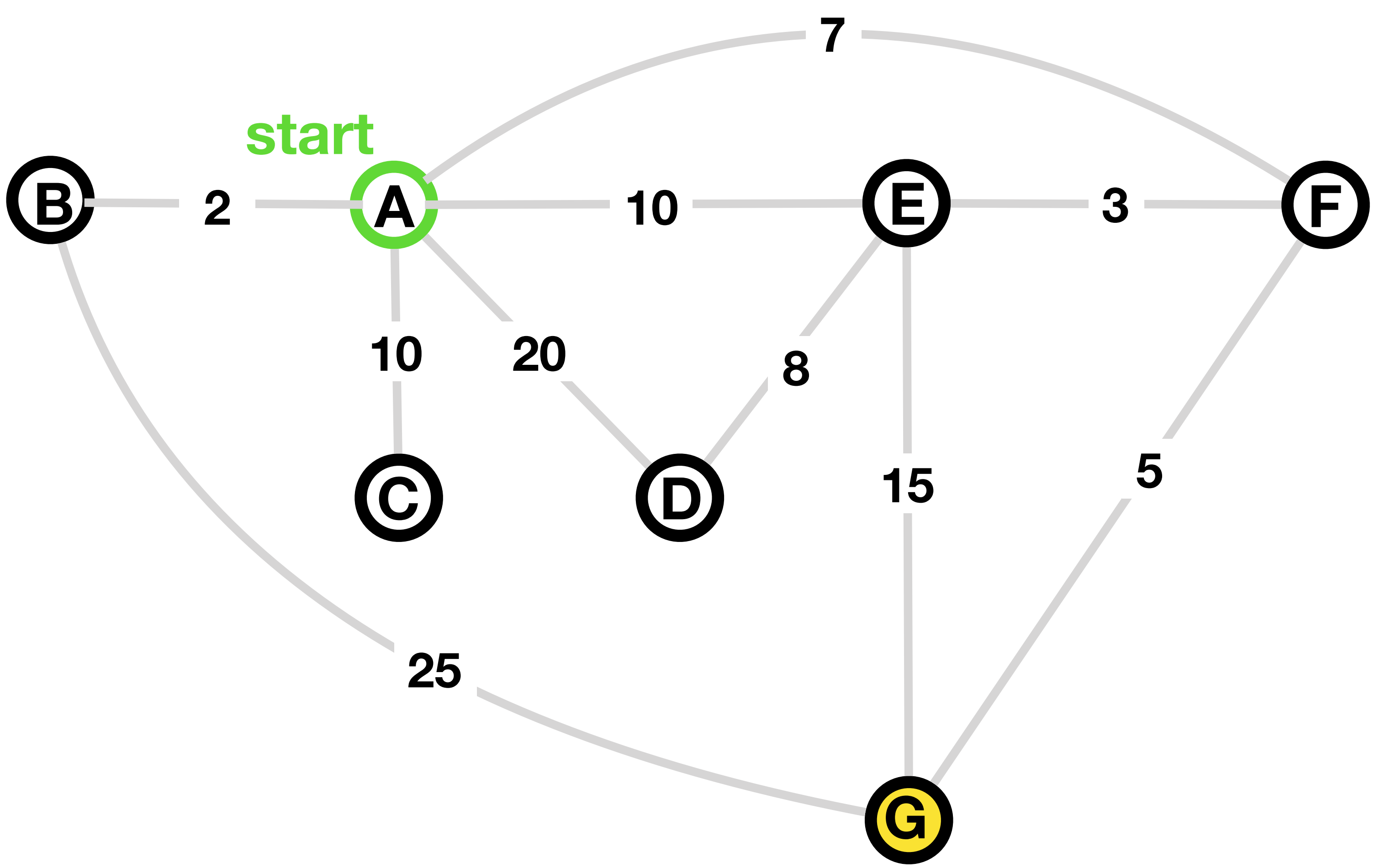


V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	F
E	10	A	T
F	7	A	T
G	12	F	F

update *current* to node with smallest distance

Dijkstra's Shortest Path

current, *i*	G
next, *j*	

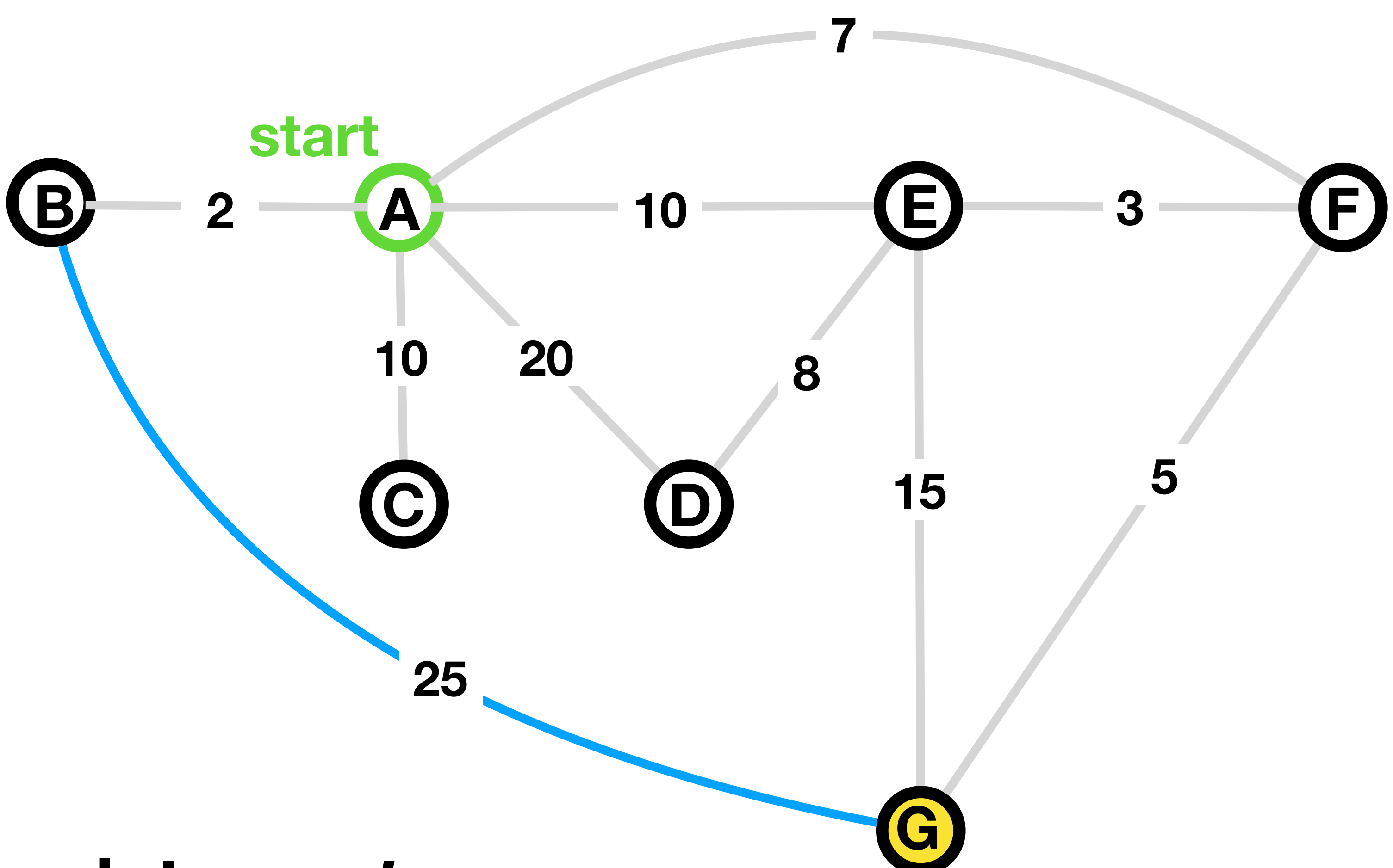


V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	F
E	10	A	T
F	7	A	T
G	12	F	F

update *current* to node with smallest distance

Dijkstra's Shortest Path

current, *i*	G
next, *j*	B



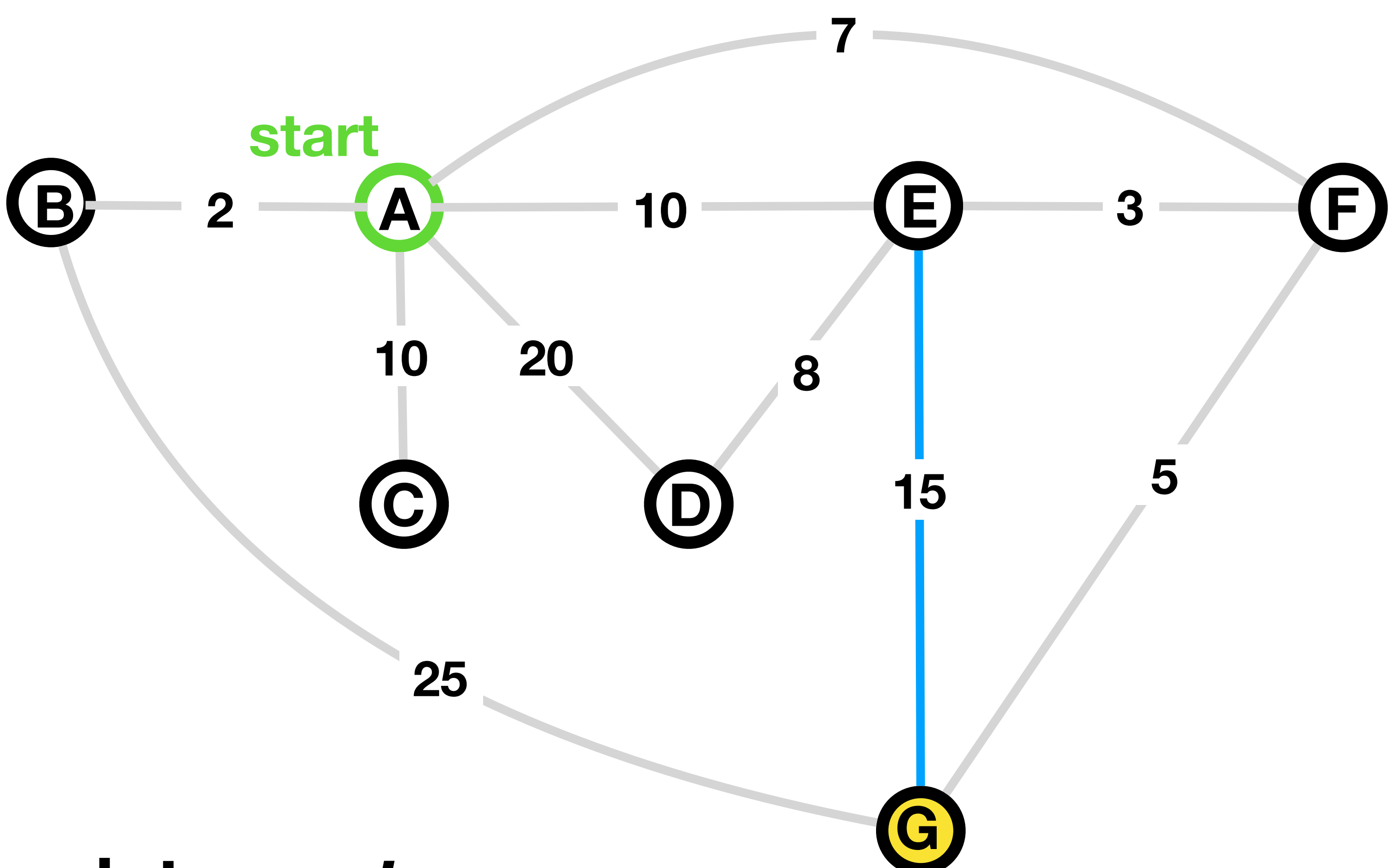
update *next.*

Is B known ? <u>Yes.</u>

V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	F
E	10	A	T
F	7	A	T
G	12	F	T

Dijkstra's Shortest Path

current, *i*	G
next, *j*	E



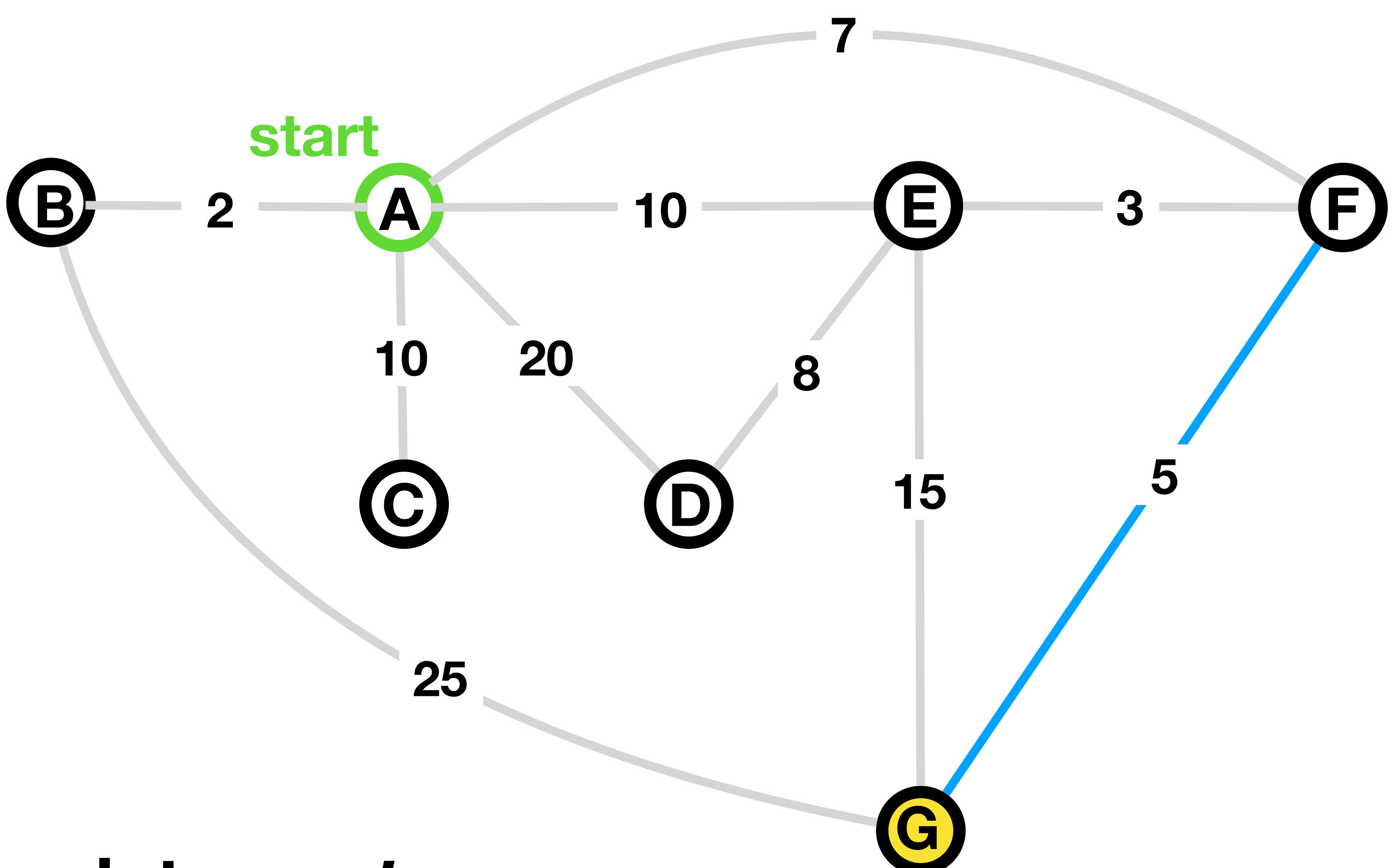
V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	F
E	10	A	T
F	7	A	T
G	12	F	T

update *next*.

Is E known ? <u>Yes.</u>

Dijkstra's Shortest Path

current, *i*	G
next, *j*	F



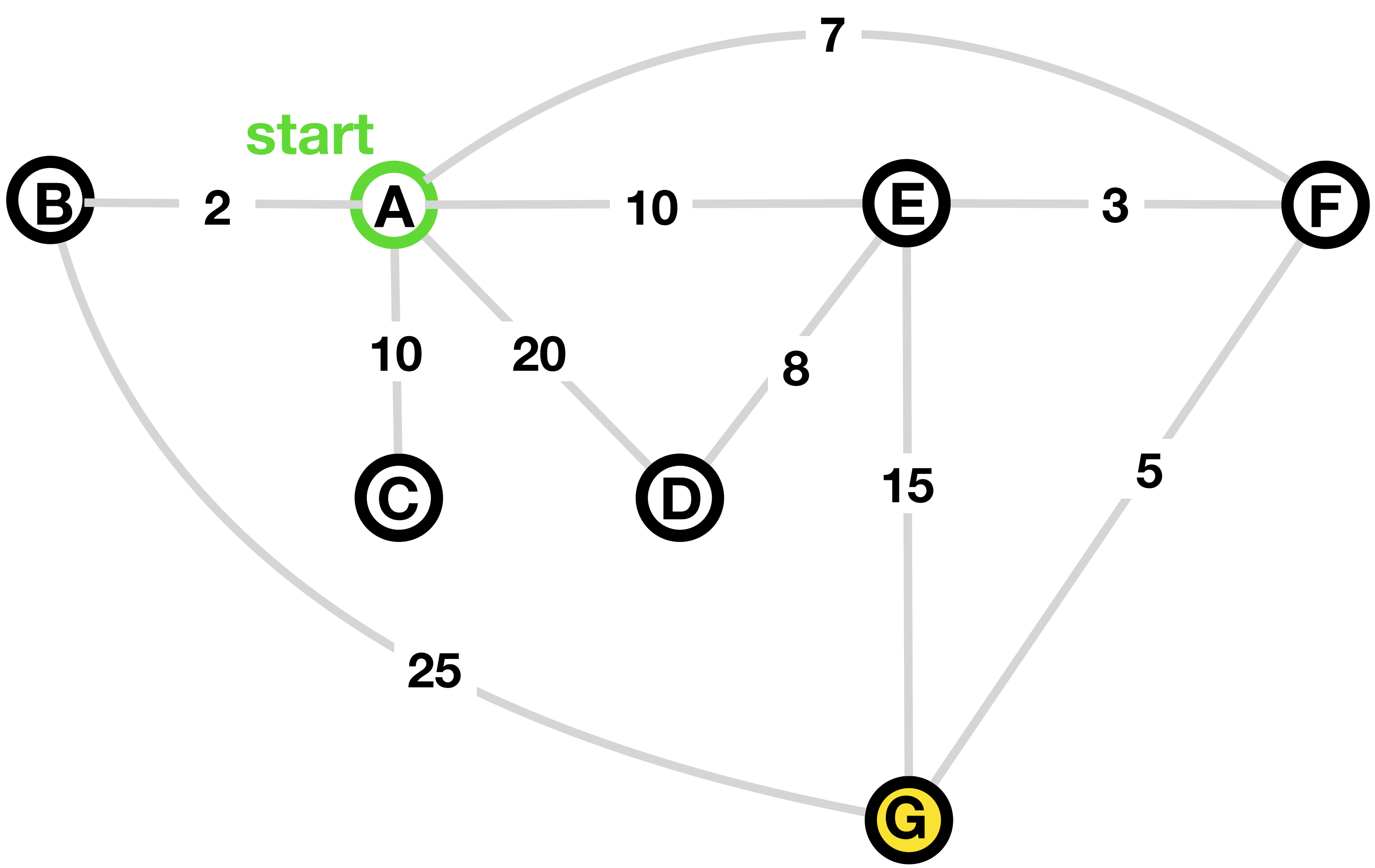
V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	F
E	10	A	T
F	7	A	T
G	12	F	T

update *next.*

Is F known ? Yes.

Dijkstra’s Shortest Path

current, *i*	G
next, *j*	

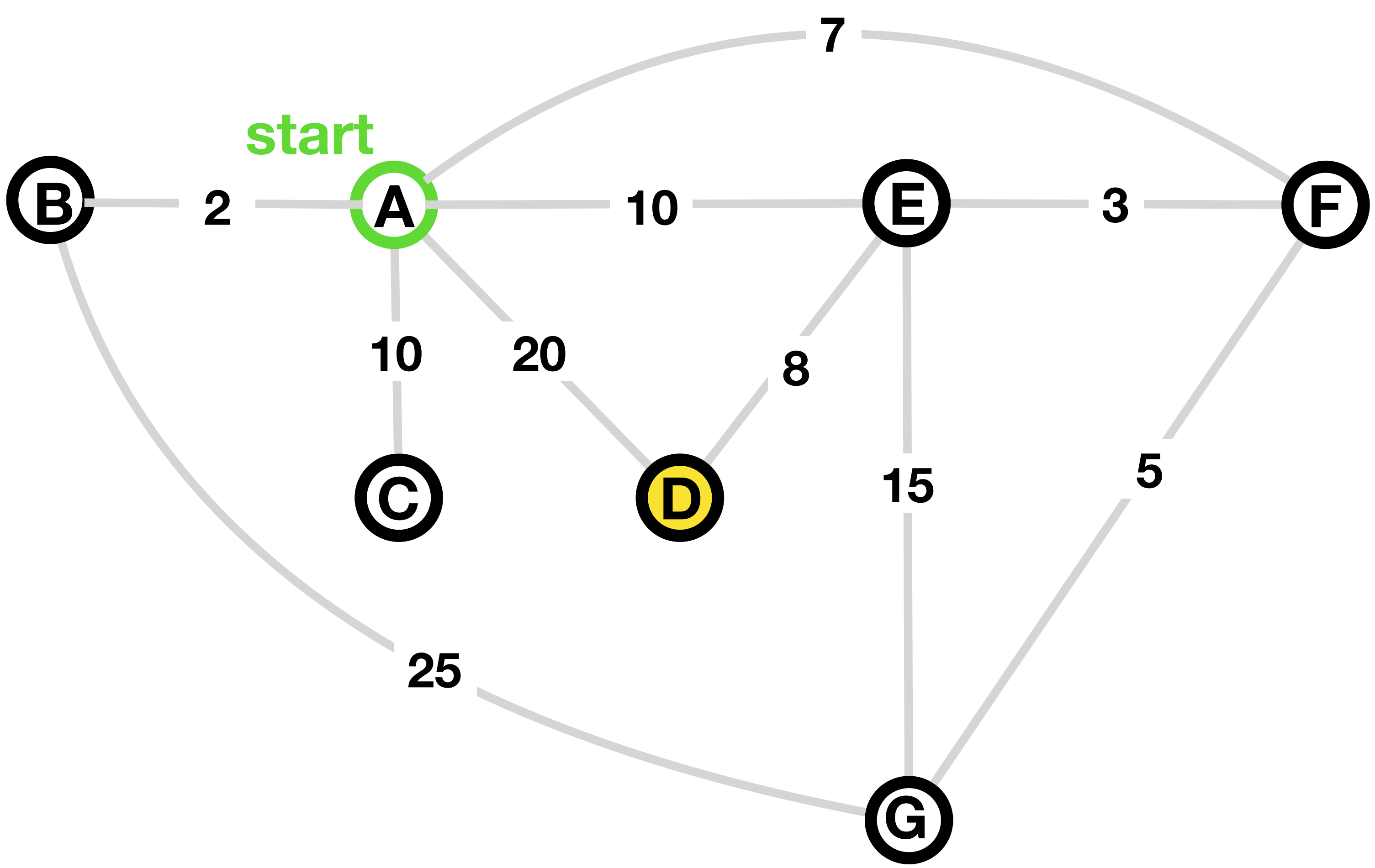


V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	F
E	10	A	T
F	7	A	T
G	12	F	T

update *current* to node with smallest distance

Dijkstra's Shortest Path

current, *i*	D
next, *j*	

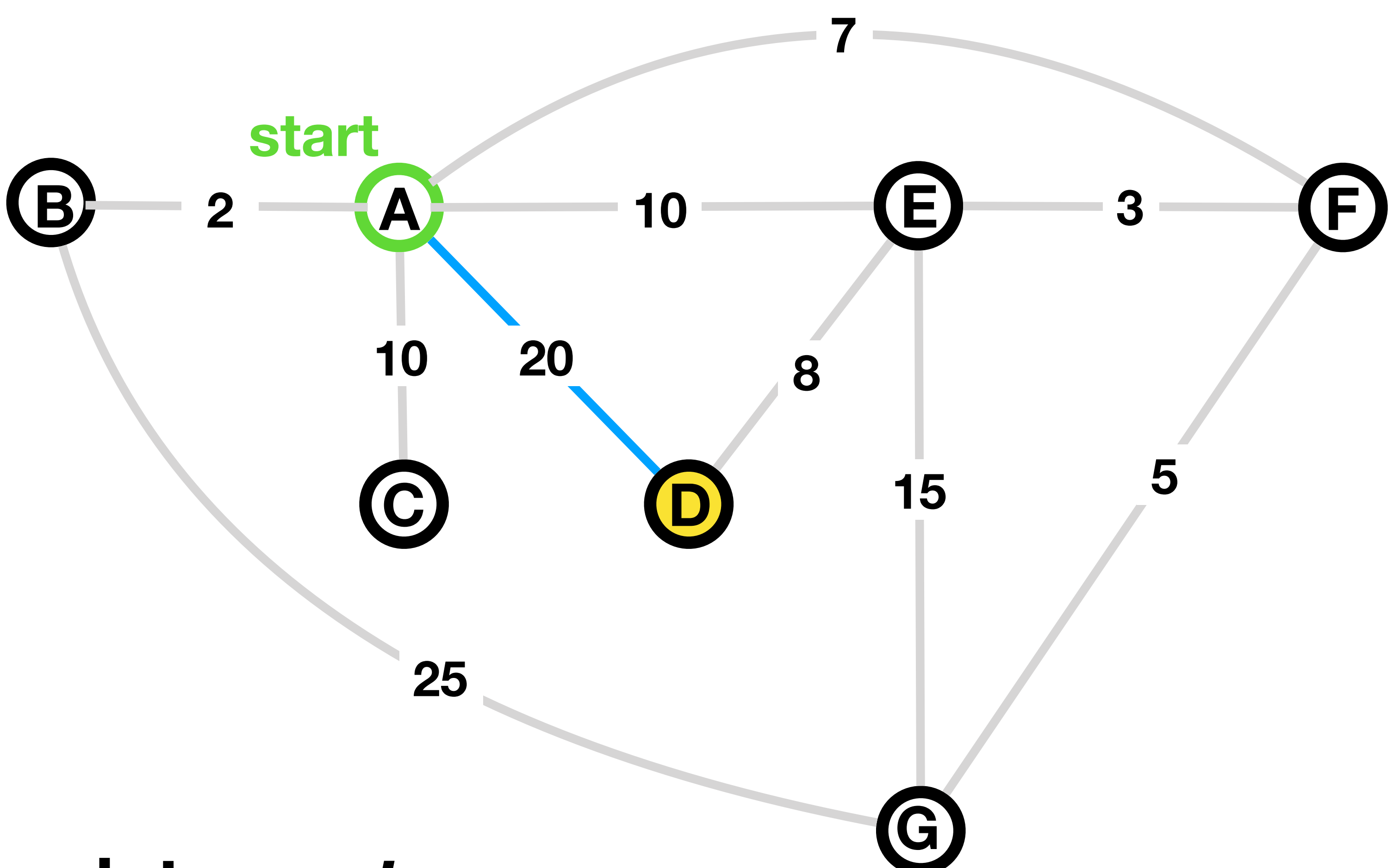


V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	F
E	10	A	T
F	7	A	T
G	12	F	T

update *current* to node with smallest distance

Dijkstra’s Shortest Path

current, *i*	D
next, *j*	A



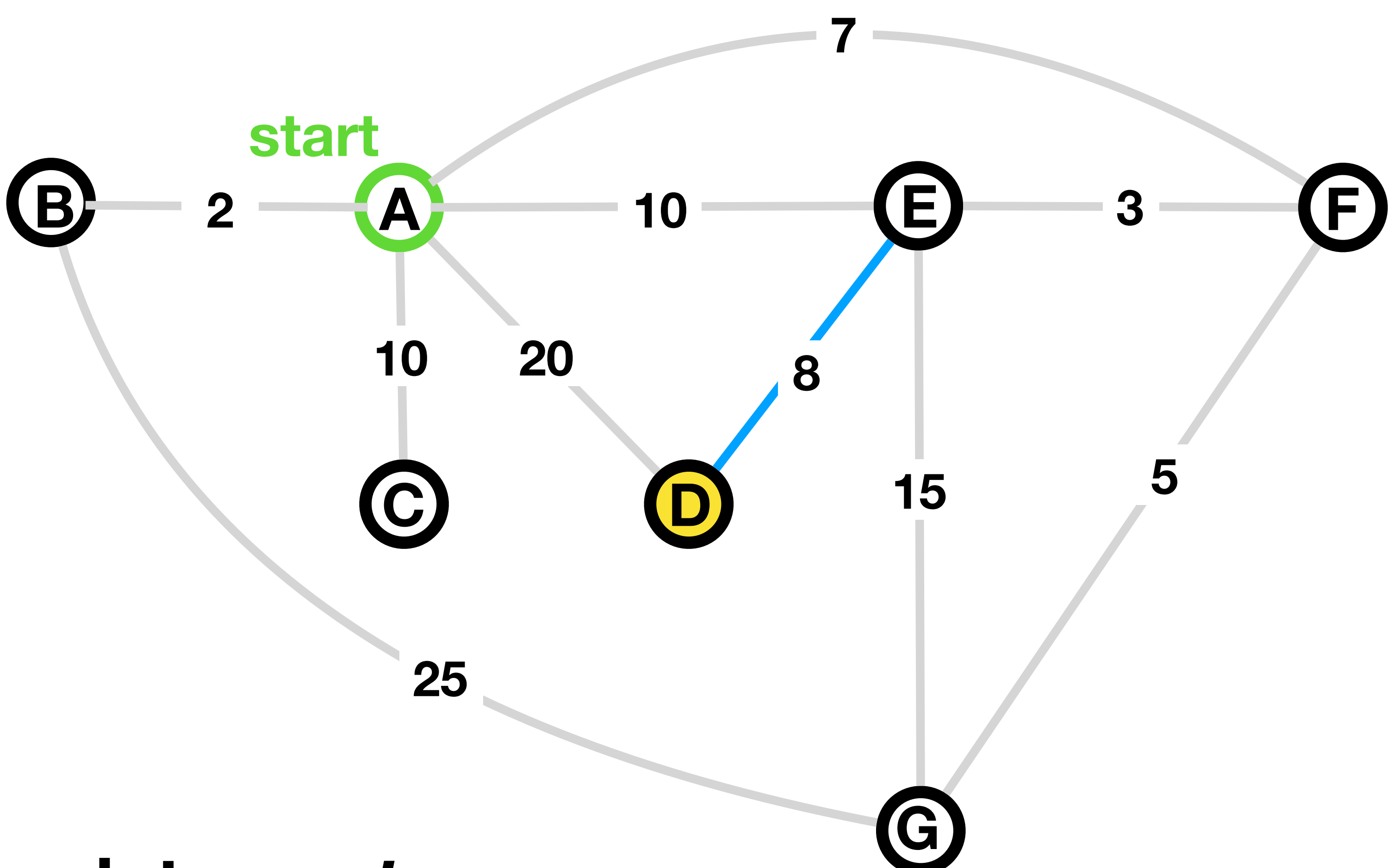
V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	T
E	10	A	T
F	7	A	T
G	12	F	T

update *next.*

Is A known ? <u>Yes.</u>

Dijkstra's Shortest Path

current, *i*	D
next, *j*	E



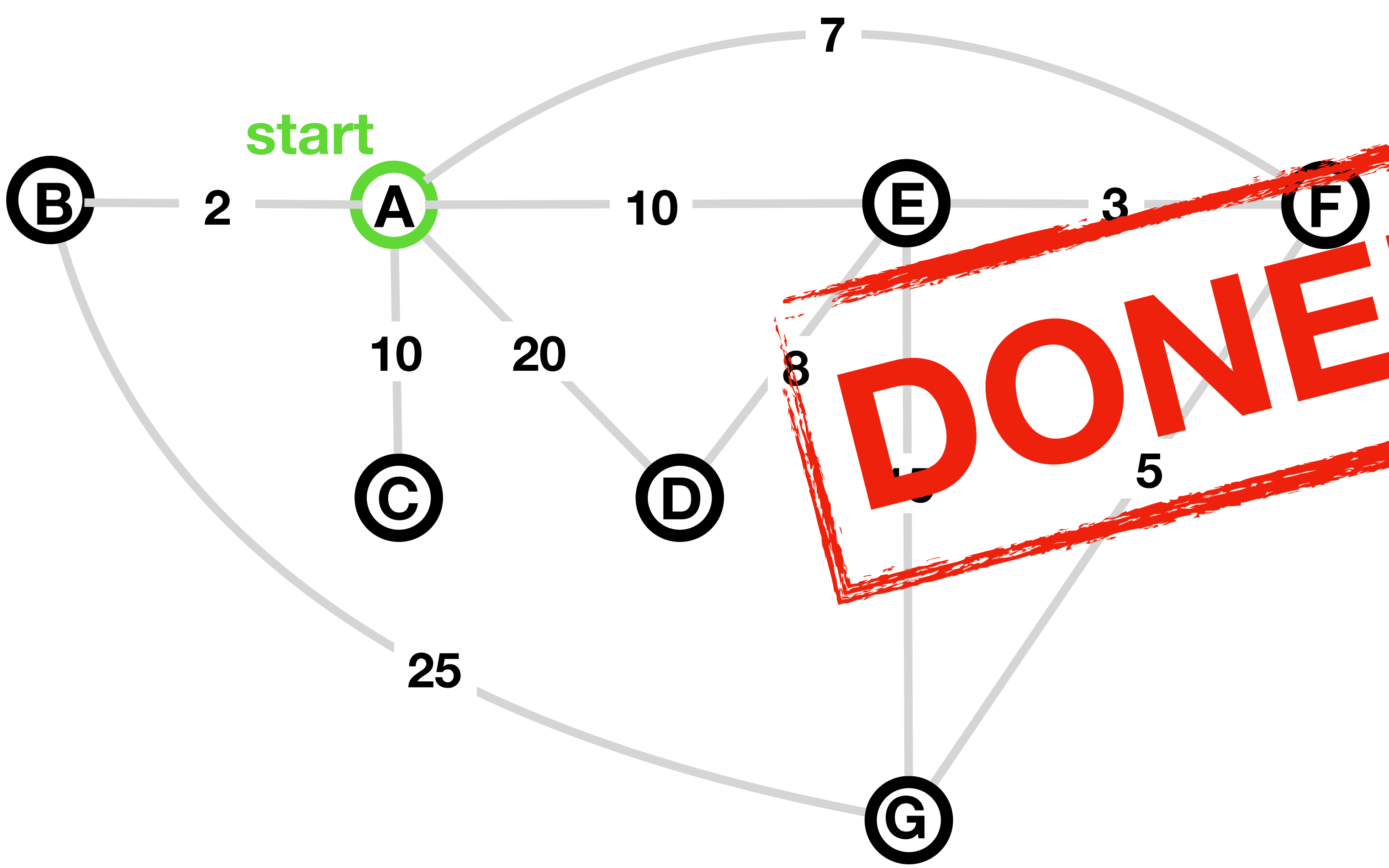
update *next*.

Is E known ? <u>Yes.</u>

V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	T
E	10	A	T
F	7	A	T
G	12	F	T

Dijkstra’s Shortest Path

current, *i*	**D**
next, *j*	



V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	T
E	10	A	T
F	7	A	T
G	12	F	T

Dijkstra’s Shortest Path

V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	T
E	10	A	T
F	7	A	T
G	12	F	T

From the table, we can extract both the shortest distance and the shortest path from the start node to all other nodes in the network

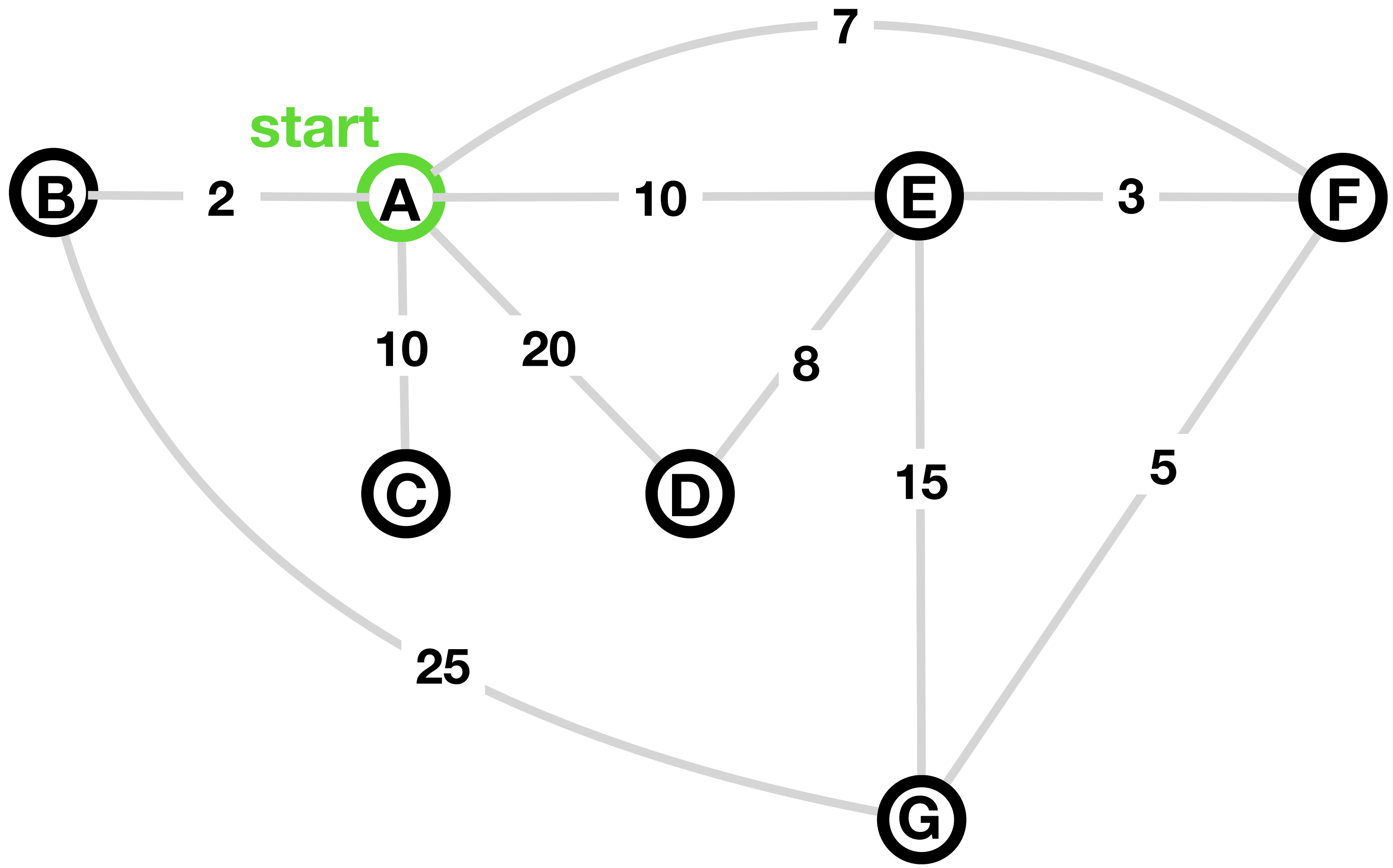
Dijkstra’s Shortest Path

V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	T
E	10	A	T
F	7	A	T
G	12	F	T

For **shortest distance** to another node, simply look at the *distance* column.

For **shortest path**, we need to backtrace the table…

Dijkstra's Shortest Path

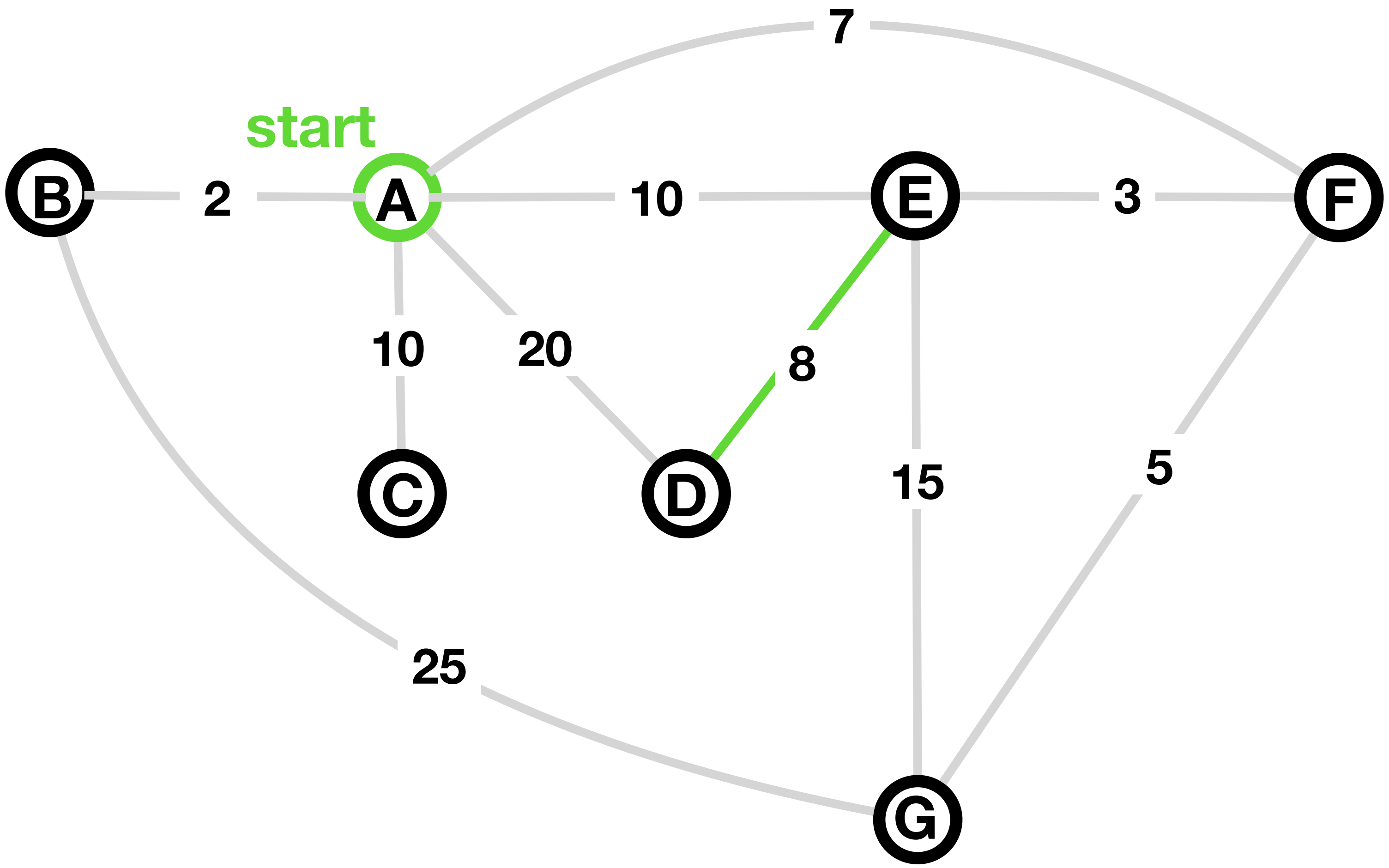


V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	T
E	10	A	T
F	7	A	T
G	12	F	T

Shortest Path from A to D:

Shortest Path from A to G:

Dijkstra's Shortest Path

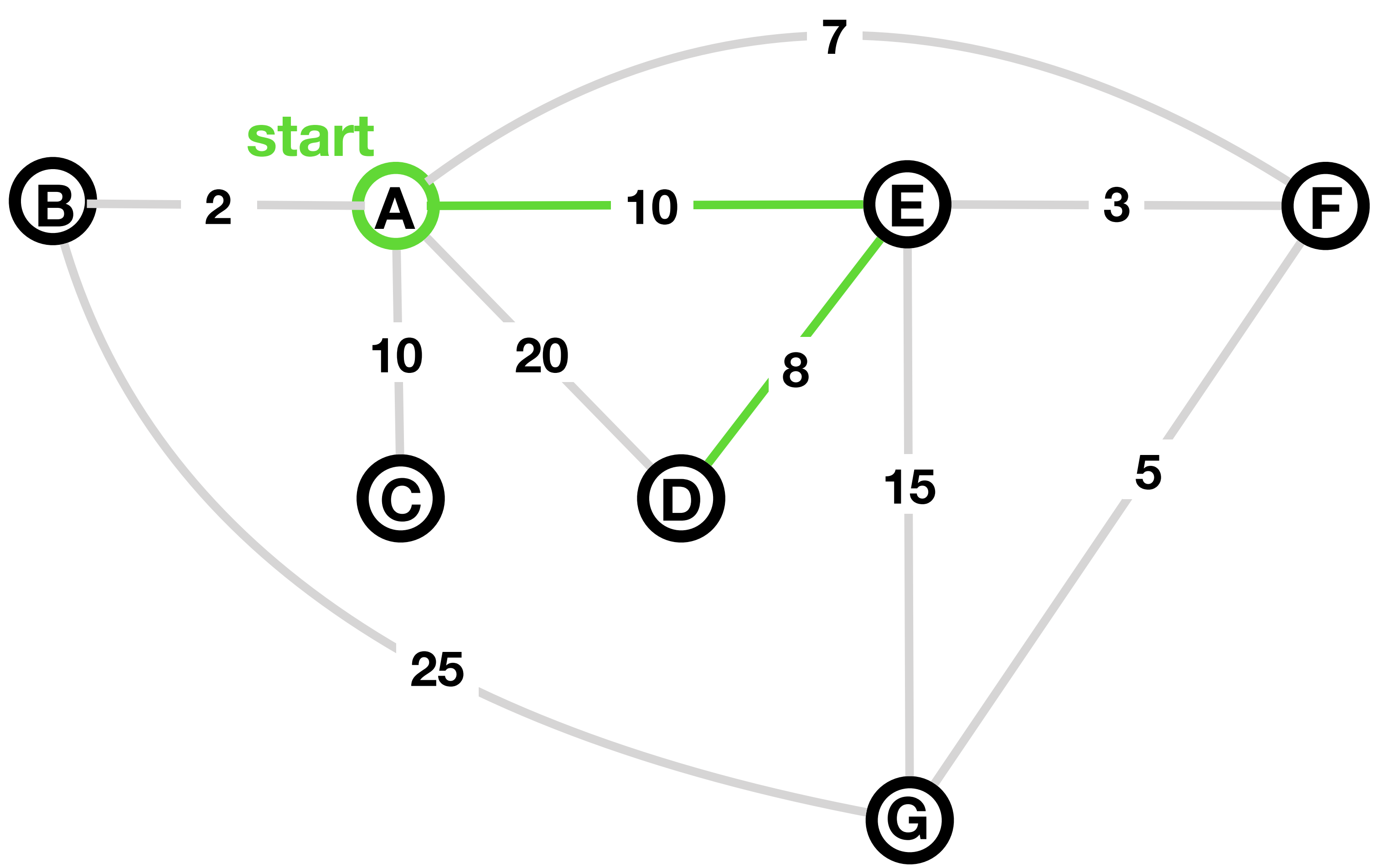


V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	T
E	10	A	T
F	7	A	T
G	12	F	T

Shortest Path from A to D: D <— E

Shortest Path from A to G:

Dijkstra’s Shortest Path

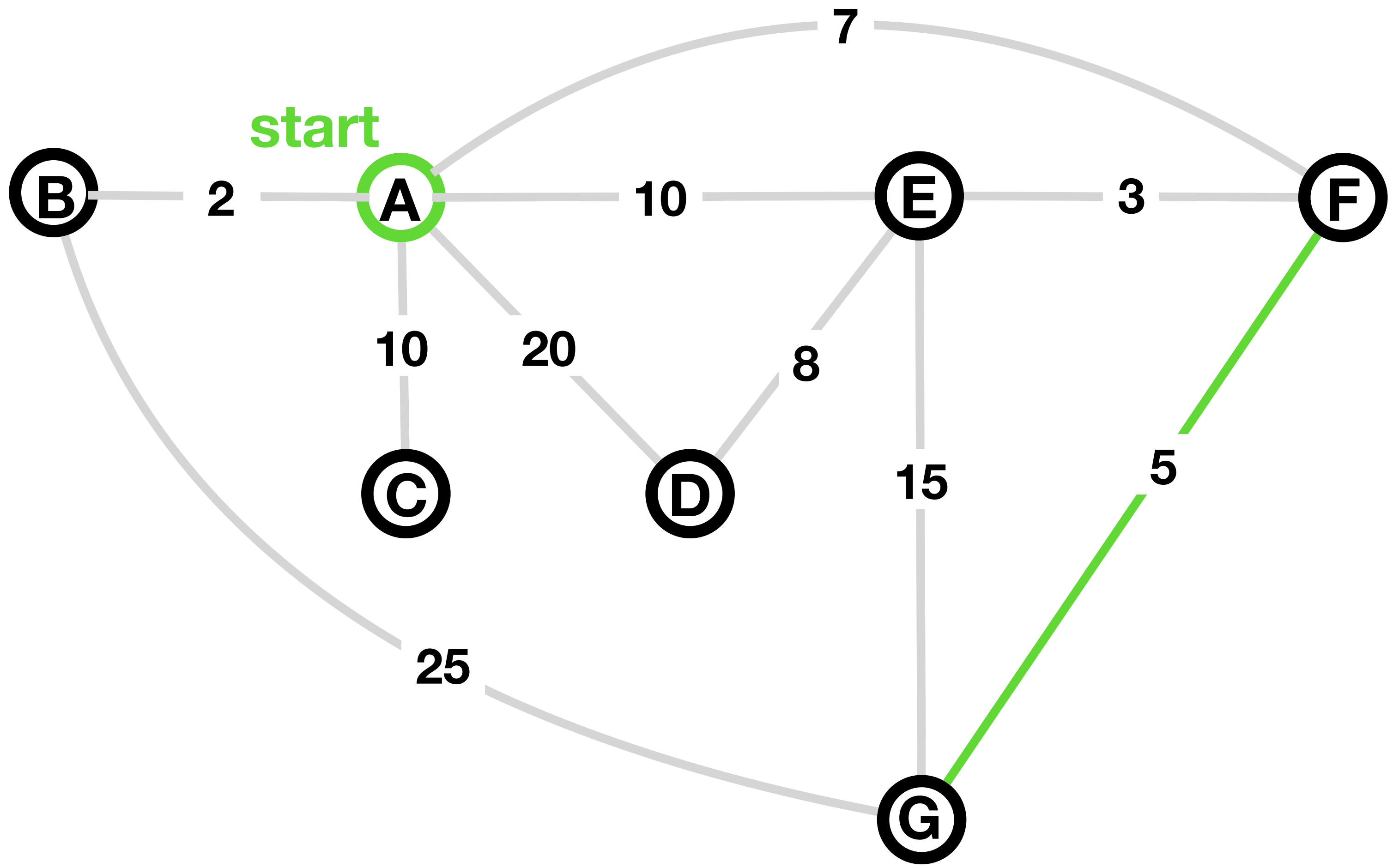


V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	T
E	10	A	T
F	7	A	T
G	12	F	T

Shortest Path from A to D: D <— E <— A

Shortest Path from A to G:

Dijkstra’s Shortest Path

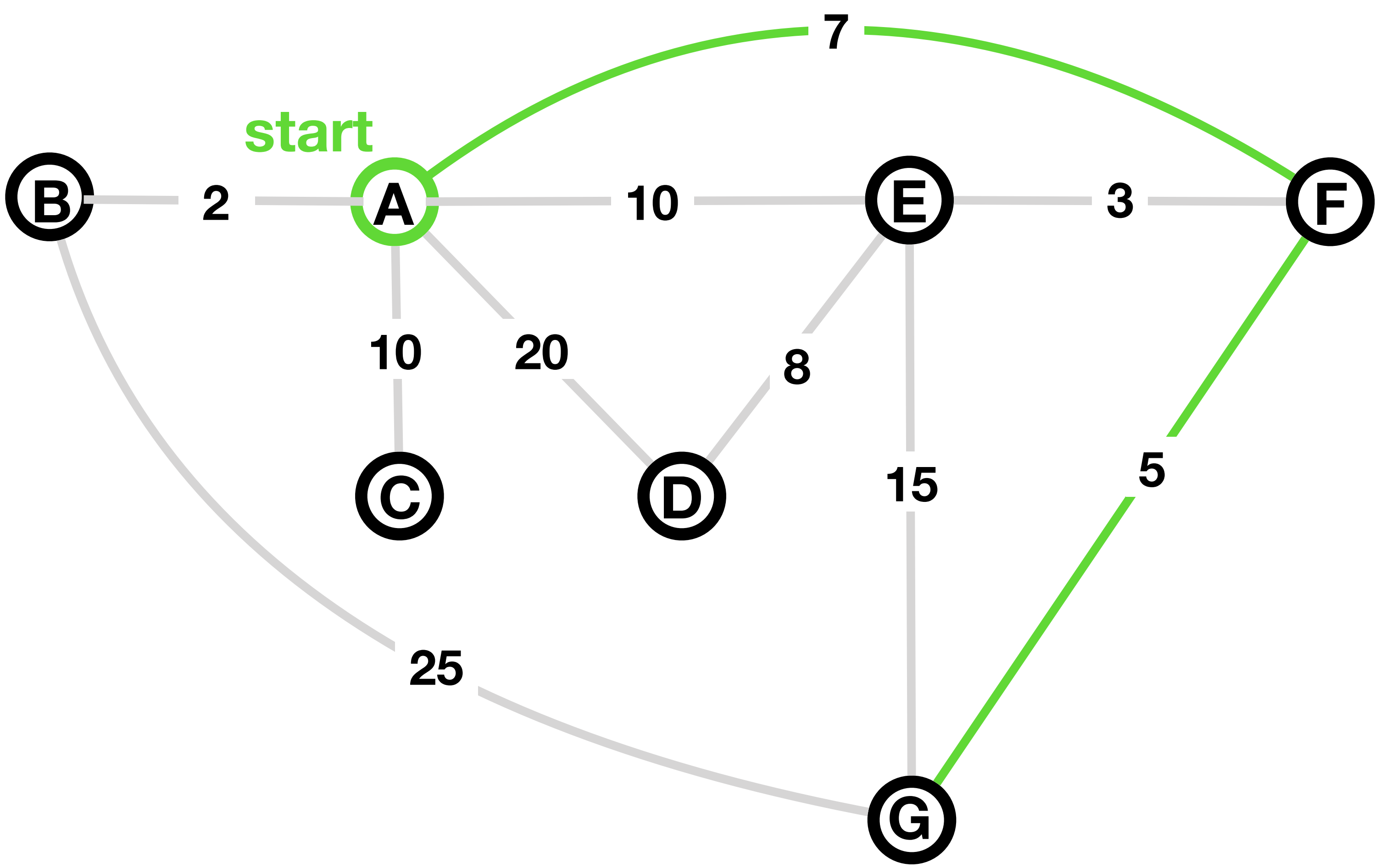


V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	T
E	10	A	T
F	7	A	T
G	12	F	T

Shortest Path from A to D: D <— E <— A

Shortest Path from A to G: G <— F

Dijkstra’s Shortest Path

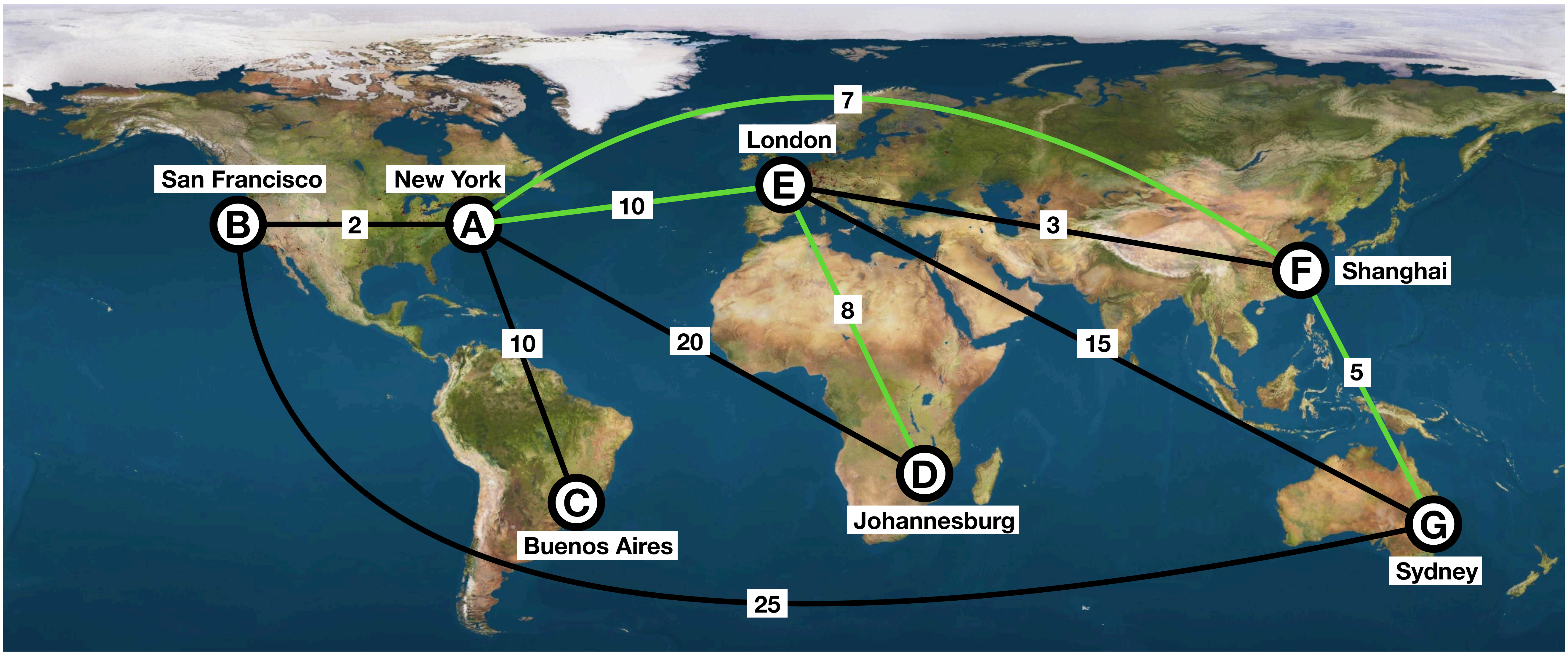


V	DISTANCE	PREVIOUS	KNOWN
A	0		T
B	2	A	T
C	10	A	T
D	18	E	T
E	10	A	T
F	7	A	T
G	12	F	T

Shortest Path from A to D: D <— E <— A

Shortest Path from A to G: G <— F <— A

Dijkstra’s Shortest Path



Shortest Path from New York (A) to Johannesburg (D):
Johannesburg (D) <— London (E) <— New York (A)

Shortest Path from New York (A) to Sydney(G):
Sydney(G) <— Shanghai(F) <— New York (A)

Dijkstra’s Shortest Path

Complexity:
$O(V^2)$ when using basic implementation
$O(E \log V)$ when using more complex data structures (adjacency list represented as a min binary heap)