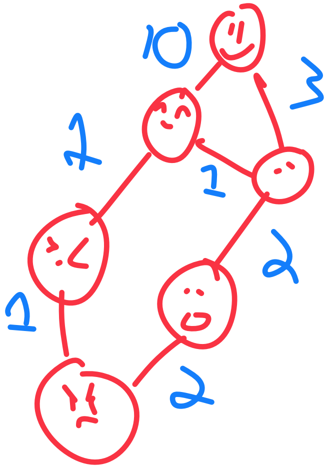
Data Structures

Single Source Shortest Path

CS 225
Brad Solomon

November 4, 2024



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Exam 4 (11/13 — 11/15)

Autograded MC and one coding question

Manually graded short answer prompt

Practice exam will be on PL

Topics covered can be found on website

Registration started October 31

https://courses.engr.illinois.edu/cs225/fa2024/exams/

Learning Objectives

Compare Kruskal and Prim MST Algorithms

Introduce Single-Source Shortest Path Problem

Discuss Dijkstra's Algorithm

Extend to All-Paths Shortest Path (if time)

Kruskal’s Algorithm

|V|= n, |E|= m

Priority Queue:	Total Running Time
Heap	O(n) + O(m) + O(m log n)
Sorted Array	O(n) + O(m log n) + O(m)

```
KruskalMST(G):
  DisjointSets forest
  foreach (Vertex v : G.vertices()):
    forest.makeSet(v)

  PriorityQueue Q    // min edge weight
  Q.buildFromGraph(G.edges())

  Graph T = (V, {})

  while |T.edges()| < n-1:
    Vertex (u, v) = Q.removeMin()
    if forest.find(u) != forest.find(v):
       T.addEdge(u, v)
       forest.union( forest.find(u),
                     forest.find(v) )

  return T

```

Prim's Algorithm

Sparse Graph: m ~ n

Adj List Heap best

Dense Graph: m ~ n^2

Unsorted Array best

```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

unsorted array good here

O(1)

	Adj. Matrix Dense	Adj. List Sparse
Heap	$O(n^2 + \underset{n}{m} \lg(n))$ $n^2 \log n$	$n \log n$ $O(n \lg(n) + m \lg(n))$ Dense $n^2 \log n$
Unsorted Array	$O(n^2)$	$O(n^2)$

MST Algorithm Runtime:

Kruskal's Algorithm:

$\underline{\mathbf{O(n + \underset{n}{\cancel{m}}\ log\ (n)\ )}}$

Prim's Algorithm:

$\underline{\mathbf{O(n\ log(n) + \underset{n}{\cancel{m}}\ log\ (n)\ )}}$

Sparse Graph: m ~ n

↳ Both $n \log n$

Dense Graph: m ~ n^2

↳ Both become $n^2 \log n$

MST Algorithm Runtime:

Kruskal's Algorithm:

O(n + m log (n))

Sparse Graph: m ~ n

Dense Graph: m ~ n^2

Prim's Algorithm:

$$O(\underbrace{n\log(n)}_{\uparrow \text{ find min}} + \underbrace{m\log(n)}_{\uparrow \text{ update costs}})$$

(log n) per update

Fib heap

O(1)* per update

$$O(n\log n + m)$$

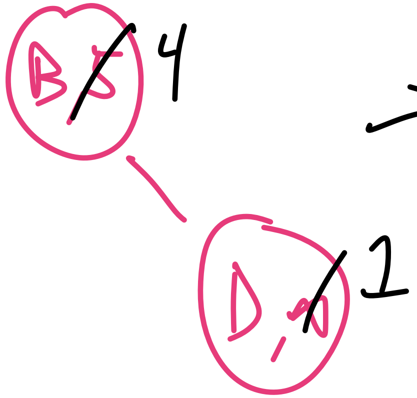
Suppose I have a new heap:

	1950s Binary Heap	1980s Fibonacci Heap
Remove Min	O(lg(n))	O(lg(n))
Decrease Key	O(lg(n))	O(1)*

What's the updated running time?

Prim = O(n logn + m)

Look back @ Friday



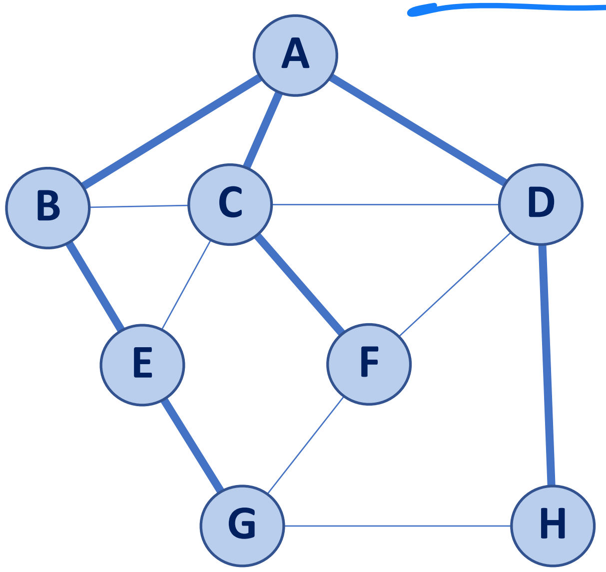
→ New heap needs to change

```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

updating now O(1)*

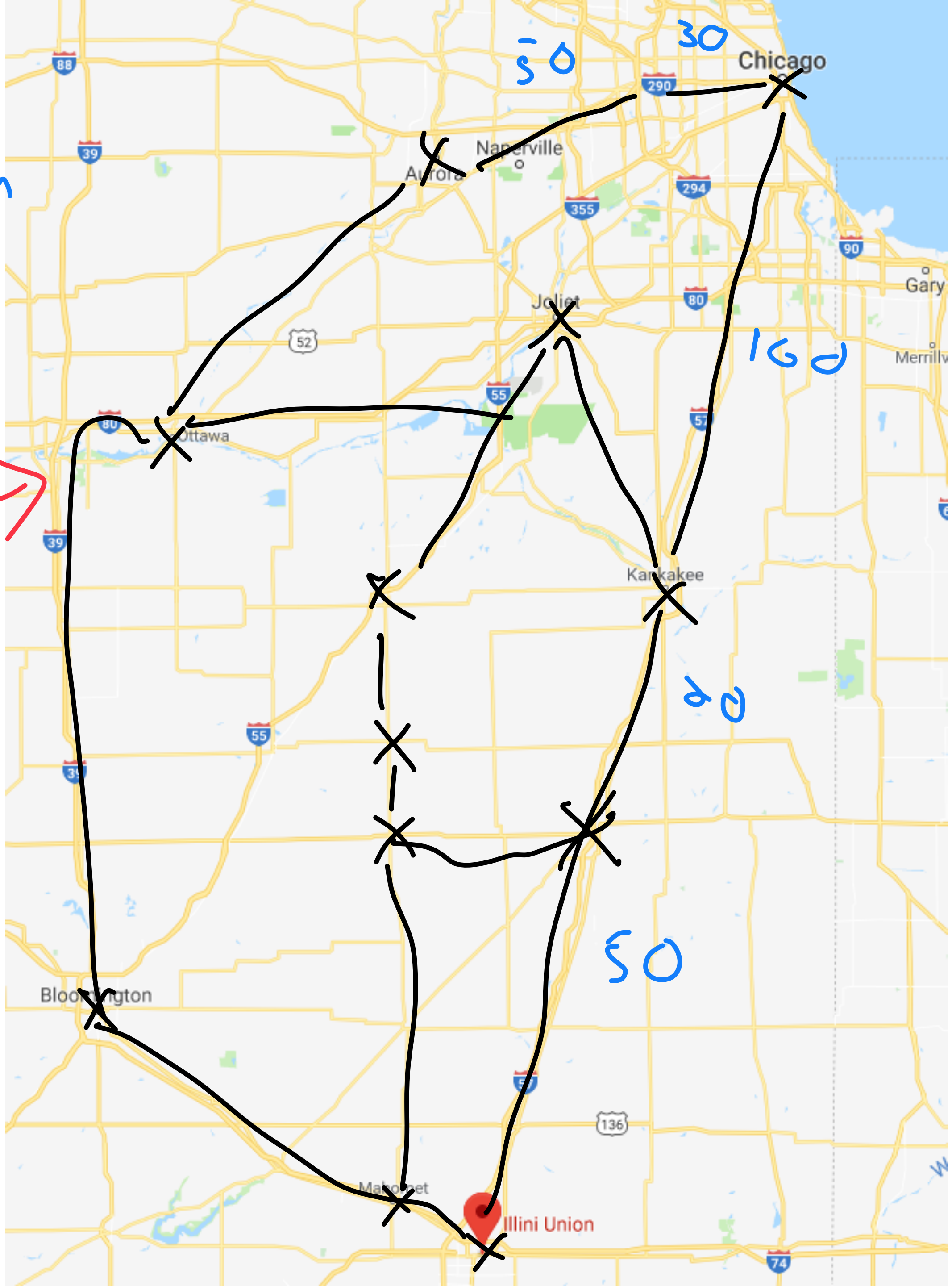
Shortest Path

BFS solved for unweighted graph



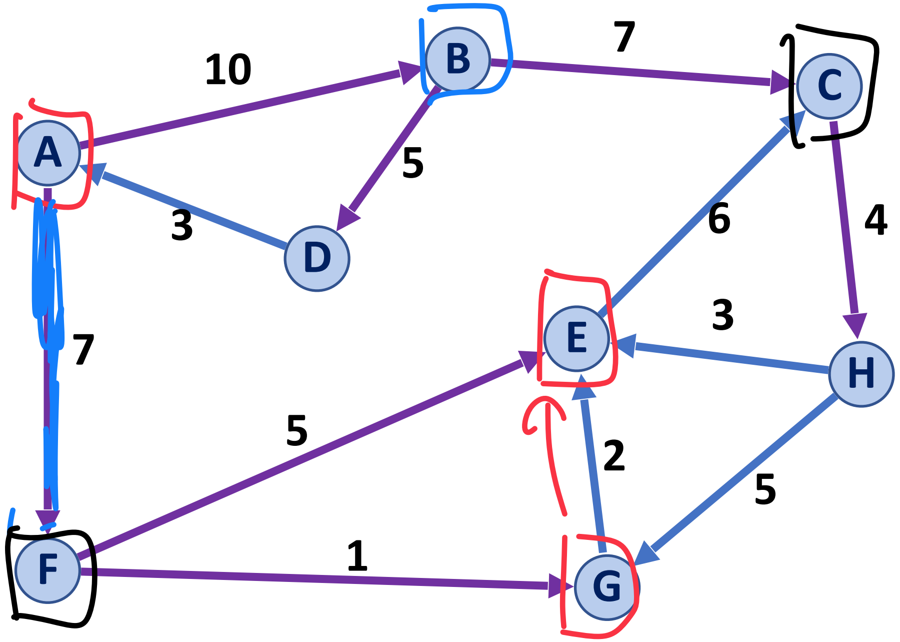
Could be dist!

estimated time!



Dijkstra's Algorithm (SSSP)

→ Very similar to Prim



start

```
DijkstraSSSP(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(u,v) + dist[u] < d[v]:
        d[v] = cost(u,v) + dist[u]
        p[v] = u
```

Init

determines next vertex

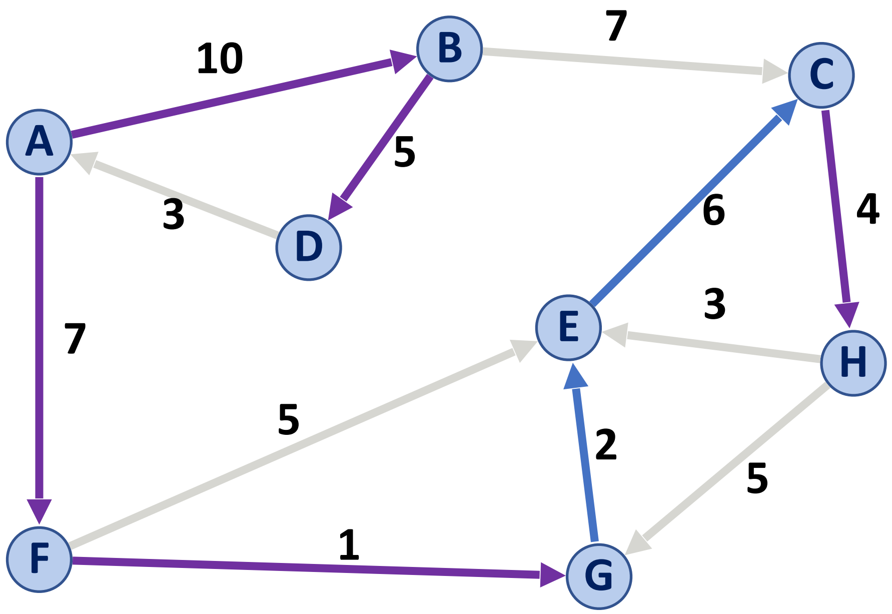
is next vertex

update dist!

Trail distance to source

	A	B	C	D	E	F	G	H
Pred	--	A	~~B~~ E	B	~~F~~ G	A	F	C
Dist	0	~~∞~~ 10	~~∞~~ ~~17~~ 16	~~∞~~ 15	~~∞~~ ~~12~~ 10	~~∞~~ 7	~~∞~~ 8	~~∞~~ 20

Dijkstra’s Algorithm (SSSP)



```
DijkstraSSSP(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(u, v) + d[u] < d[v]:
        d[v] = cost(u, v) + d[u]
        p[v] = u
```

A	B	C	D	E	F	G	H
--	A	E	B	G	A	F	C
0	10	16	15	10	7	8	20

Dijkstra's Algorithm (SSSP)

Assume heap / Fib

What is the running time of Dijkstra's Algorithm?

↳ This is Prim!

$O(n + \underbrace{\overbrace{n \log n + m}}_{\text{Dom term}})$

Find min

update

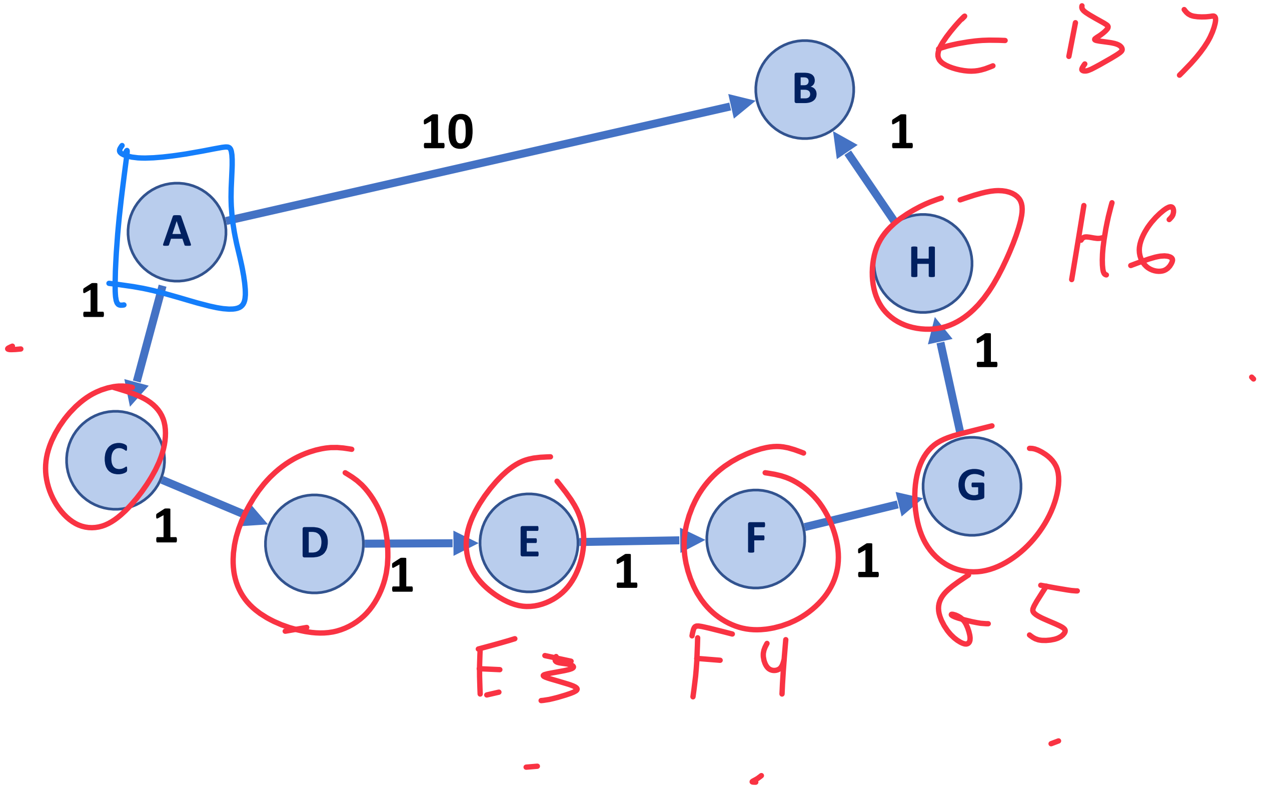
@15 + @18: $\sum_{v} \deg(v) = 2m$

Total # edge updates is m

```
   DijkstraSSSP(G, s):
6    foreach (Vertex v : G):                                O(n)
7      d[v] = +inf
8      p[v] = NULL
9    d[s] = 0
10
11   PriorityQueue Q // min distance, defined by d[v]
12   Q.buildHeap(G.vertices())
13   Graph T            // "labeled set"
14
15   repeat n times:   nx
16     Vertex u = Q.removeMin()                             O(log n)
17     T.add(u)
18     foreach (Vertex v : neighbors of u not in T):
19       if cost(u, v) + d[u] < d[v]:
20         d[v] = cost(u, v) + d[u]                         O(1*)
21         p[v] = m
22
23   return T
```

Dijkstra’s Algorithm (SSSP)

When we will visit B in the following graph?

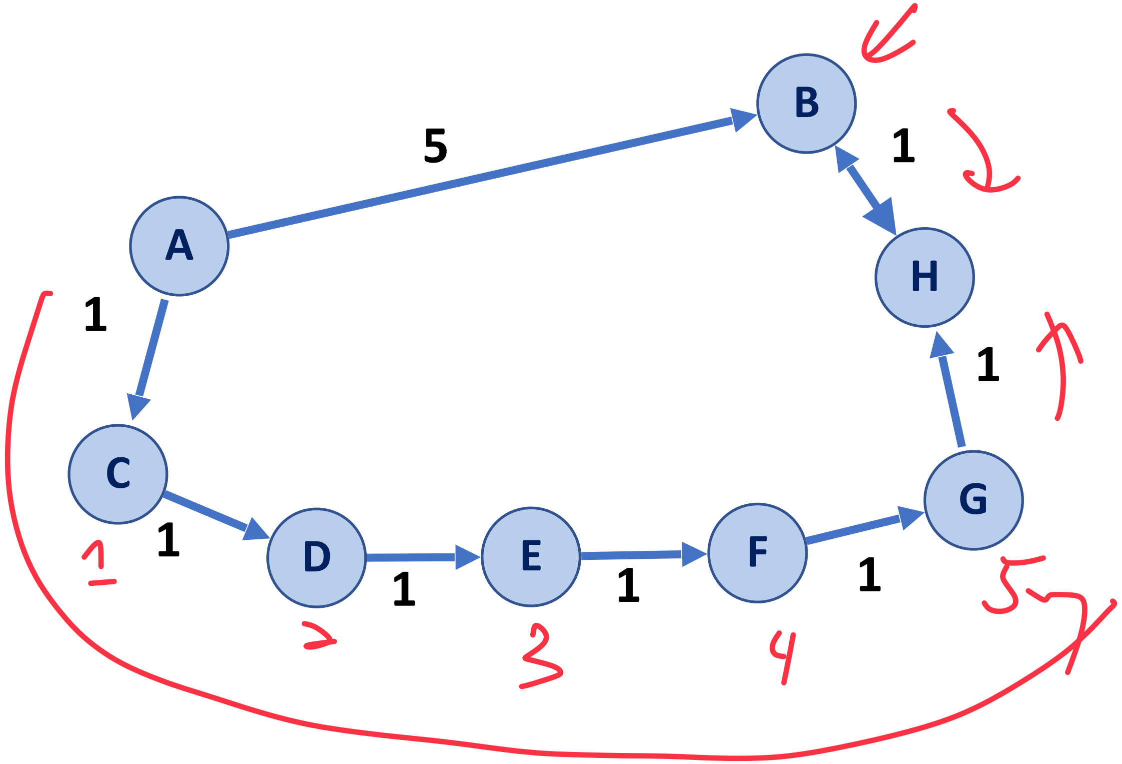


B 10
C 2
D 2

Claim: Using alg we will always visit a node through its shortest path

Dijkstra’s Algorithm (SSSP)

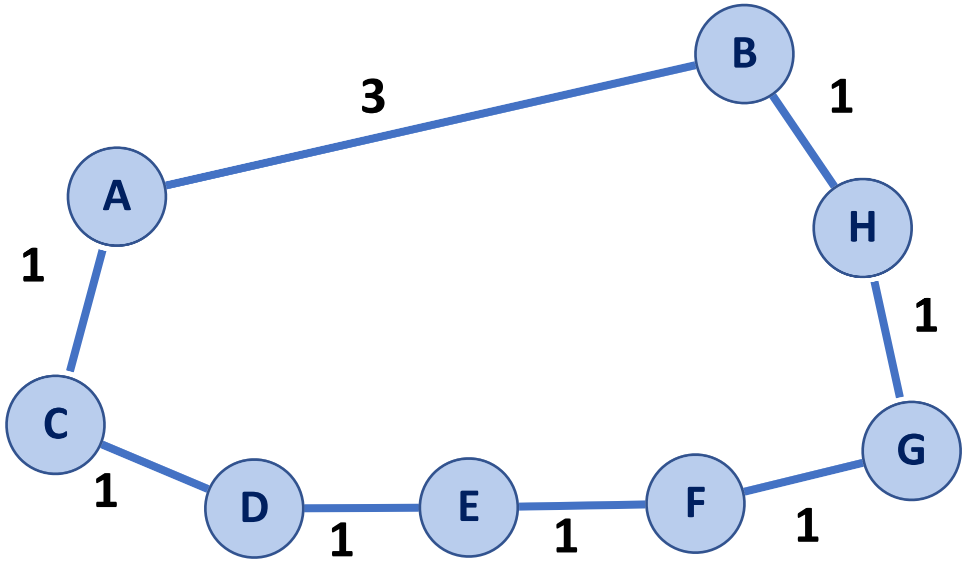
When we will visit H in the following graph?



H 6 by either B or G

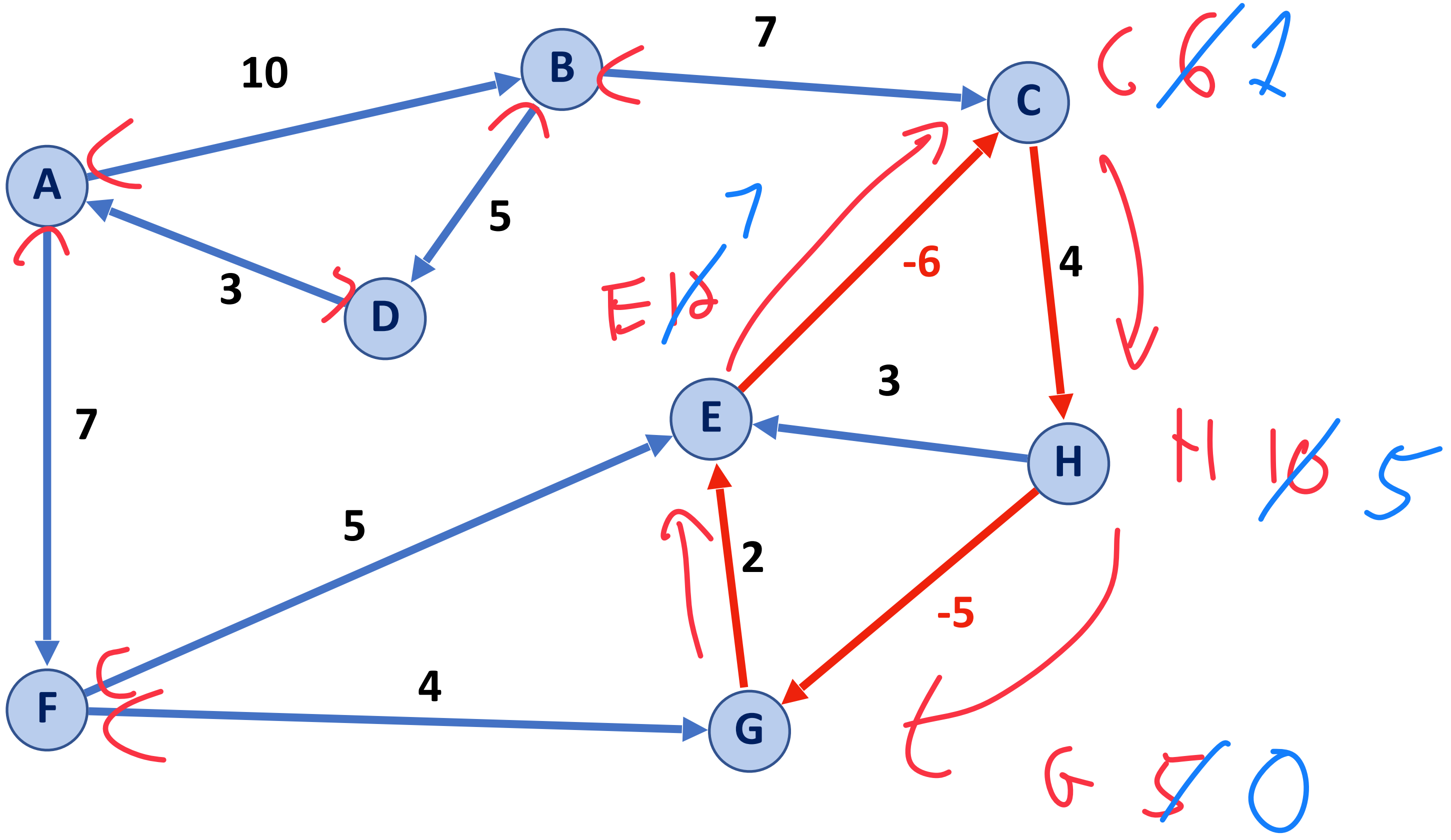
Dijkstra’s Algorithm (SSSP)

How does Dijkstra’s algorithm handle undirected graphs?



Dijkstra’s Algorithm (SSSP)

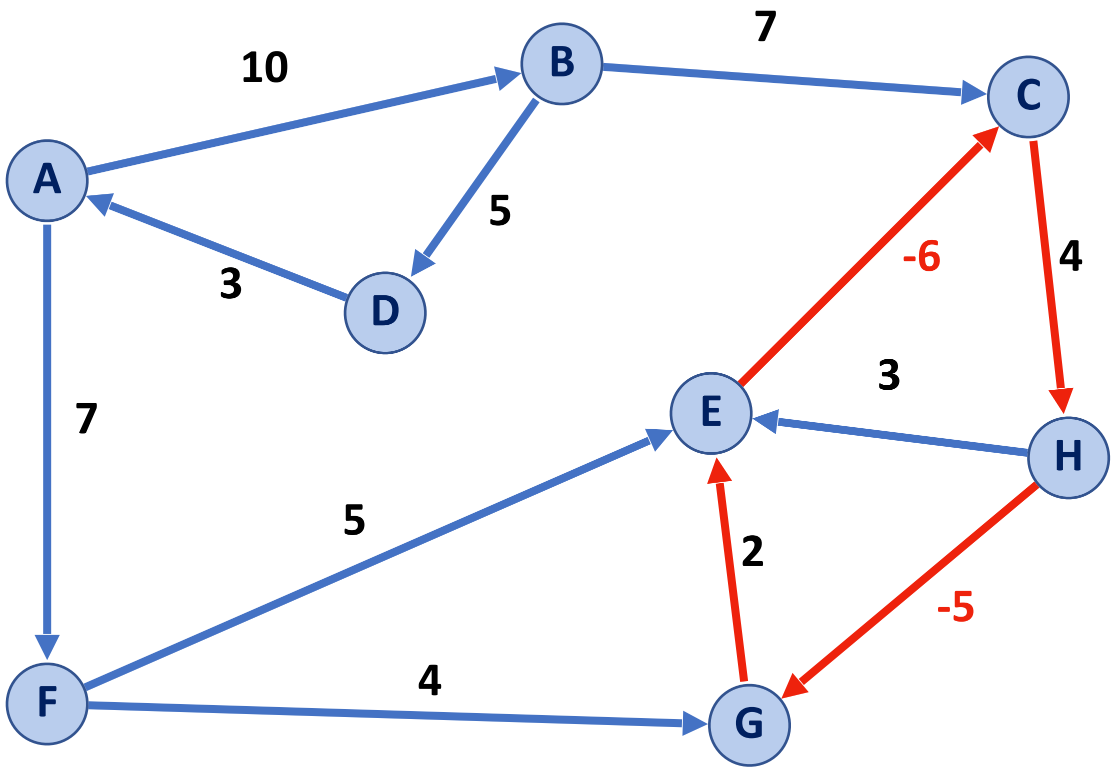
How does Dijkstras handle a negative weight cycle?



Infinite loop to -∞

Dijkstra’s Algorithm (SSSP)

How does Dijkstras handle a negative weight cycle?



Shortest Path (A → E): A → F → E → (C → H → G → E)*

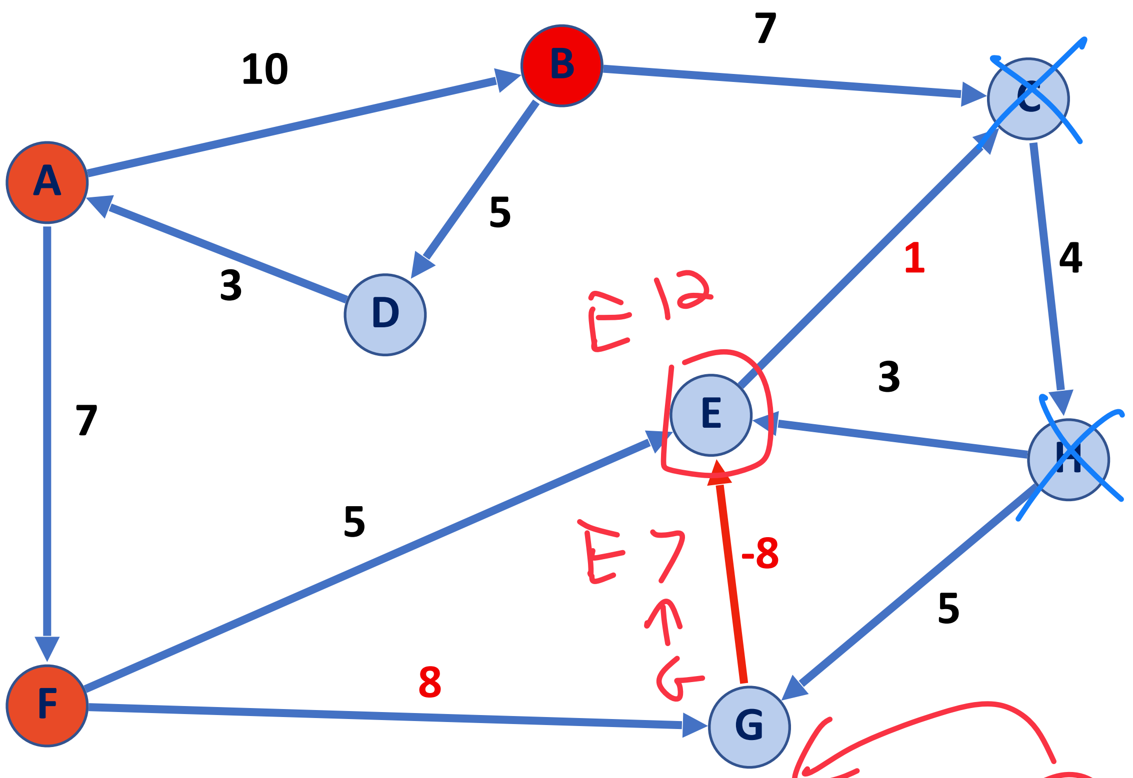
Length: 12

Length: -5 (repeatable)

Dijkstra’s Algorithm (SSSP)

How does Dijkstras handle a negative weight edge without a cycle?

↳ It doesn't!



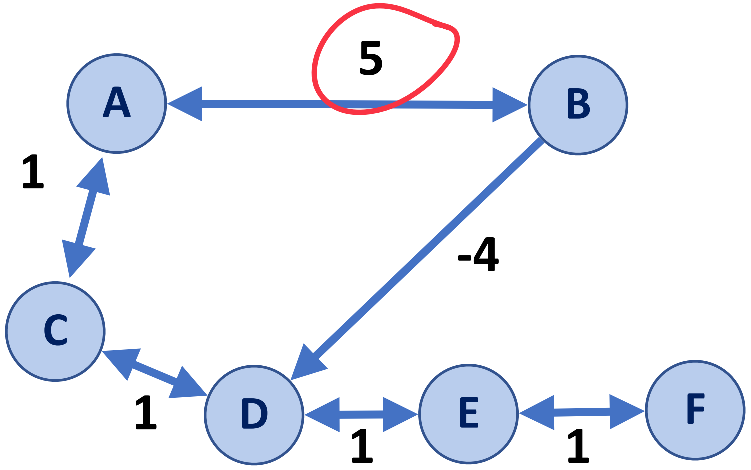
A	B	C	D	E	F	G	H
--	A	B	B	F	A	F	--
0	10	17	15	12	7	15	∞

Dijkstra's Algorithm (SSSP)

We assume that item pulled out of priority queue is **the next smallest item**

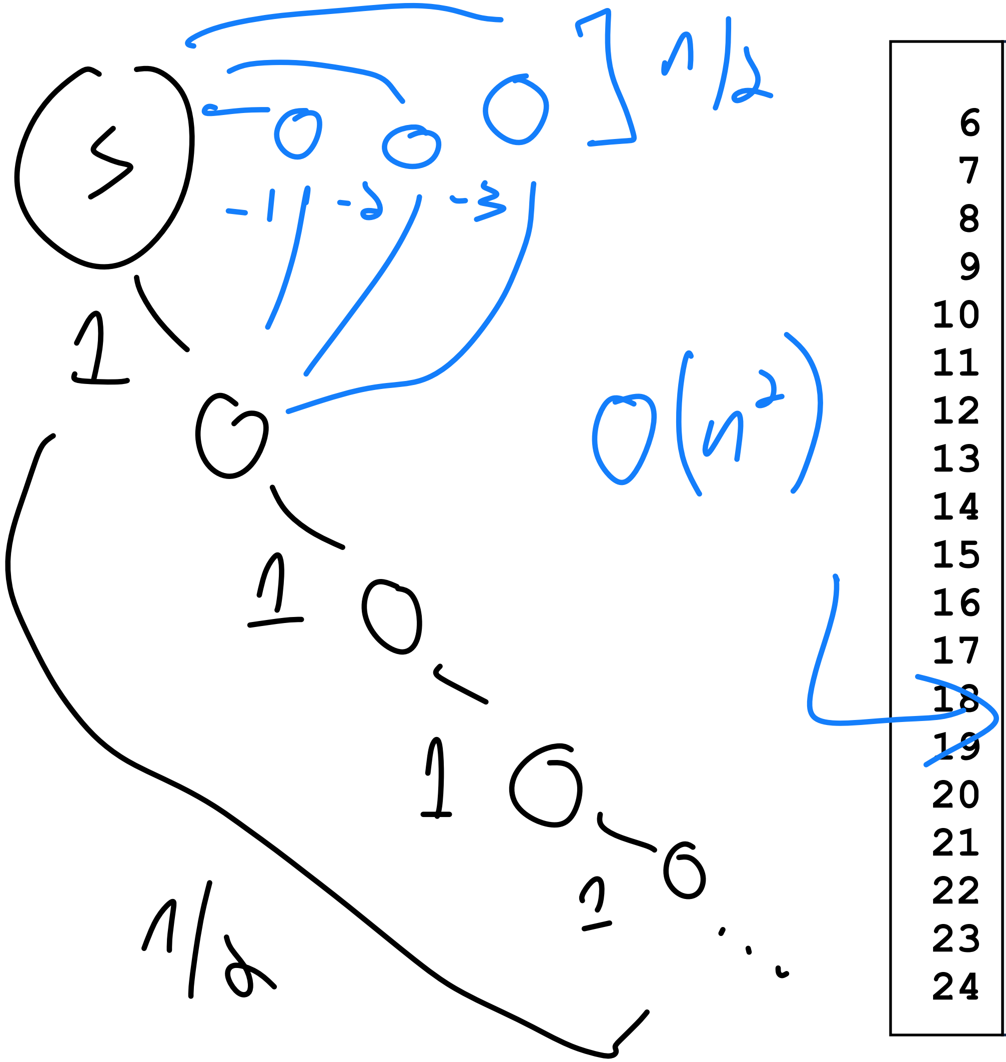
Negative weights break this assumption!

A	B	C	D	E	F
--	A	A	C	D	E
0	5	1	~~2~~ 1	~~3~~ 2	~~4~~ 3



Dijkstra’s Algorithm (SSSP)

Recalculating all distances is possible, but algorithm runtime is very bad!



```
DijkstraSSSP(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T         // "labeled set"

  repeat until Q.empty():
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(u, v) + d[u] < d[v]:
        d[v] = cost(u, v) + d[u]
        p[v] = m
        if v not in Q:
           Q.push(v)
  return T
```

Dijkstra’s Algorithm (SSSP)

Dijkstras Algorithm works only on non-negative weights

Optimal implementation:

Fibonacci Heap

If dense, unsorted list ties

Optimal runtime:

Sparse: $O(m + n \log n)$

Dense: $O(n^2)$

```
DijkstraSSSP(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(u, v) + d[u] < d[v]:
        d[v] = cost(u, v) + d[u]
        p[v] = m

  return T
```

(Basically Prim)

← This changes

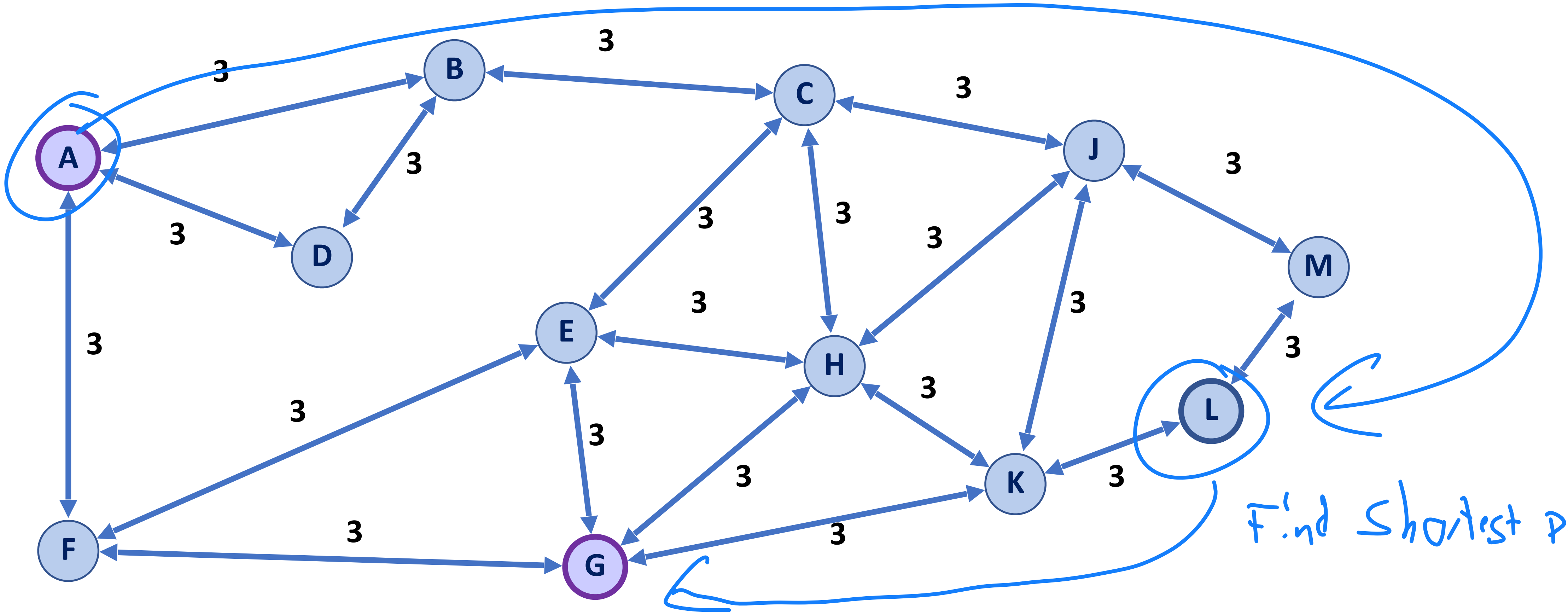
Landmark Path Problem

Source Dest

What if I wanted to get the shortest path from A to G but stopping at L along the way?

stops along way

Find short Path -> use Dijkstras



Find Shortest Path

Floyd-Warshall Algorithm

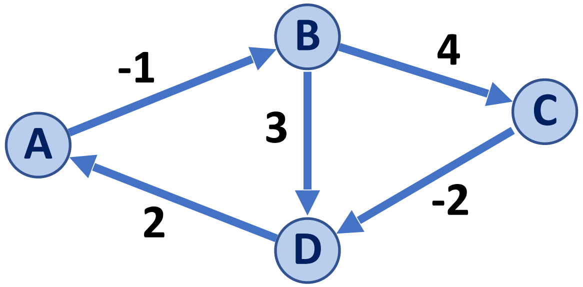
Floyd-Warshall’s Algorithm is an alternative to Dijkstra in the presence of negative-weight edges (not negative weight cycles).

```
FloydWarshall(G):
  Let d be a adj. matrix initialized to +inf
  foreach (Vertex v : G):
    d[v][v] = 0
  foreach (Edge (u, v) : G):
    d[u][v] = cost(u, v)

  foreach (Vertex u : G):
    foreach (Vertex v : G):
      foreach (Vertex w : G):
        if (d[u, v] > d[u, w] + d[w, v])
          d[u, v] = d[u, w] + d[w, v]
```

Floyd-Warshall Algorithm

```
FloydWarshall(G):
  Let d be a adj. matrix initialized to +inf
  foreach (Vertex v : G):
    d[v][v] = 0
  foreach (Edge (u, v) : G):
    d[u][v] = cost(u, v)
```

	A	B	C	D
A				
B				
C				
D				

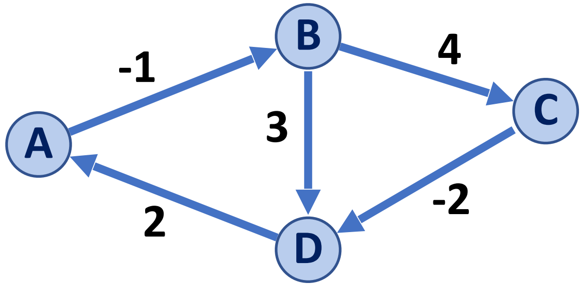


Floyd-Warshall Algorithm

```
foreach (Vertex w : G):
  foreach (Vertex u : G):
    foreach (Vertex v : G):
      if (d[u, v] > d[u, w] + d[w, v])
        d[u, v] = d[u, w] + d[w, v]
```

Let us consider comparisons where w = A:

	A	B	C	D
A	0	-1	∞	∞
B	∞	0	4	3
C	∞	∞	0	-2
D	2	∞	∞	0



Floyd-Warshall Algorithm

```
foreach (Vertex w : G):
  foreach (Vertex u : G):
    foreach (Vertex v : G):
      if (d[u, v] > d[u, w] + d[w, v])
        d[u, v] = d[u, w] + d[w, v]
```

Let **w** be midpoint

Let **u** be start point

Let **v** be end point

Is our distance shorter now?

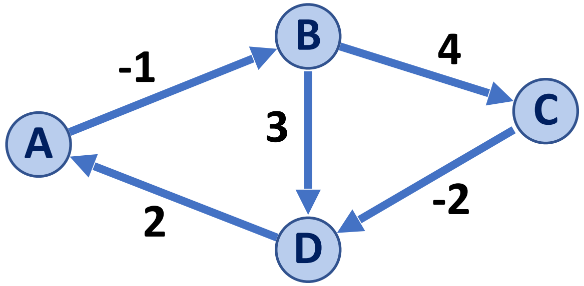
Let us consider comparisons where w = A:

u=A, v=A

u=A, v=B

Don’t waste time if u=w or v=w!

	A	B	C	D
A	0	-1	∞	∞
B	∞	0	4	3
C	∞	∞	0	-2
D	2	∞	∞	0



Floyd-Warshall Algorithm

```
foreach (Vertex w : G):
  foreach (Vertex u : G):
    foreach (Vertex v : G):
      if (d[u, v] > d[u, w] + d[w, v])
        d[u, v] = d[u, w] + d[w, v]
```

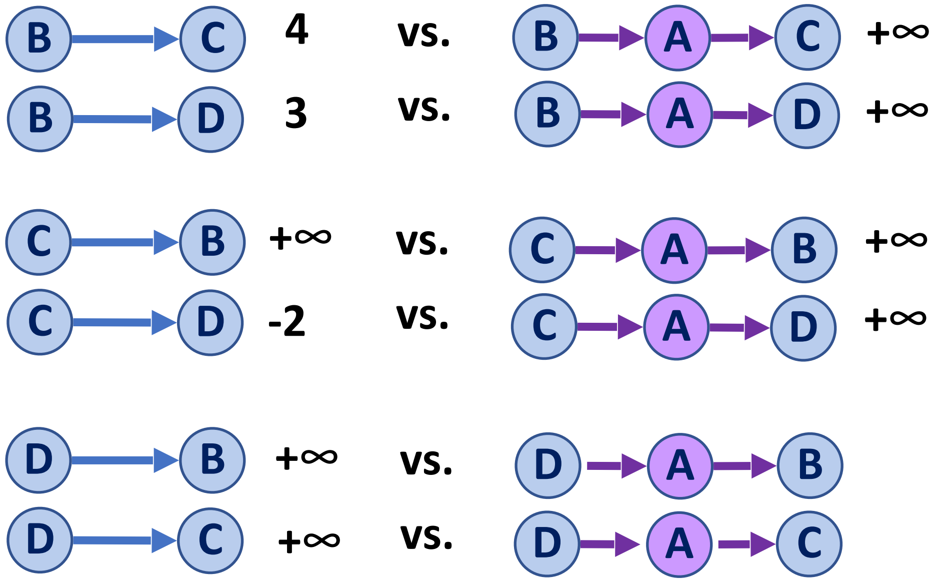
Let **w** be midpoint

Let **u** be start point

Let **v** be end point

Is our distance shorter now?

Let us consider w = A (and u != w and v != w):



B → C 4 vs. B → A → C +∞

B → D 3 vs. B → A → D +∞

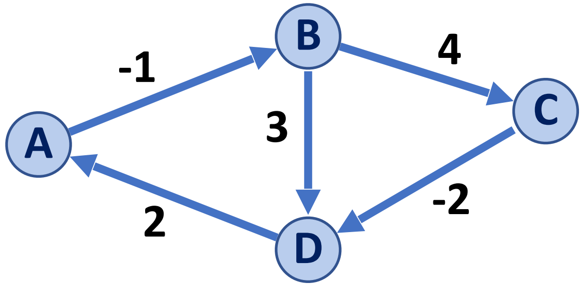
C → B +∞ vs. C → A → B +∞

C → D -2 vs. C → A → D +∞

D → B +∞ vs. D → A → B

D → C +∞ vs. D → A → C

	A	B	C	D
A	0	-1	∞	∞
B	∞	0	4	3
C	∞	∞	0	-2
D	2	∞	∞	0



Floyd-Warshall Algorithm

```
foreach (Vertex w : G):
  foreach (Vertex u : G):
    foreach (Vertex v : G):
      if (d[u, v] > d[u, w] + d[w, v])
        d[u, v] = d[u, w] + d[w, v]
```

Let **w** be midpoint

Let **u** be start point

Let **v** be end point

Is our distance shorter now?

Let us consider w = A (and u != w and v != w):

B → C 4 vs. B → A → C +∞

B → D 3 vs. B → A → D +∞

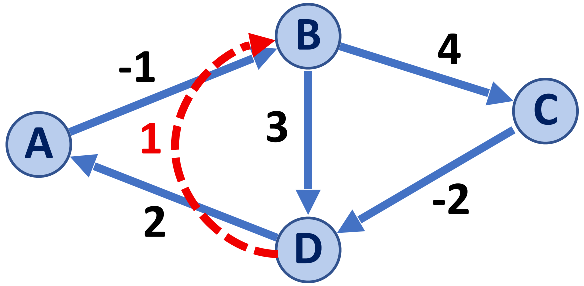
C → B +∞ vs. C → A → B +∞

C → D -2 vs. C → A → D +∞

D → B +∞ vs. D → A → B 1

D → C +∞ vs. D → A → C +∞

	A	B	C	D
A	0	-1	∞	∞
B	∞	0	4	3
C	∞	∞	0	-2
D	2	1	∞	0

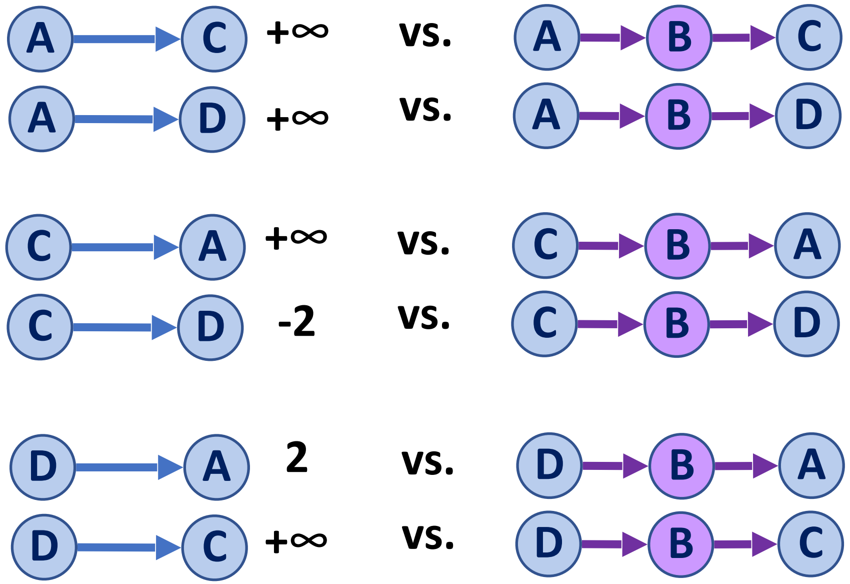


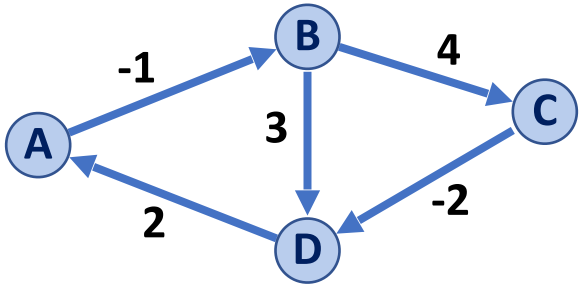
Floyd-Warshall Algorithm

```
foreach (Vertex w : G):
  foreach (Vertex u : G):
    foreach (Vertex v : G):
      if (d[u, v] > d[u, w] + d[w, v])
        d[u, v] = d[u, w] + d[w, v]
```

	A	B	C	D
A	0	-1	∞	∞
B	∞	0	4	3
C	∞	∞	0	-2
D	2	1	∞	0

Let us consider w = B (and u != w and v != w):





Floyd-Warshall Algorithm

```
foreach (Vertex w : G):
  foreach (Vertex u : G):
    foreach (Vertex v : G):
      if (d[u, v] > d[u, w] + d[w, v])
        d[u, v] = d[u, w] + d[w, v]
```

	A	B	C	D
A	0	-1	3	2
B	∞	0	4	3
C	∞	∞	0	-2
D	2	1	5	0

Let us consider w = C (and u != w and v != w):

A → B -1 vs. A → C → B +∞

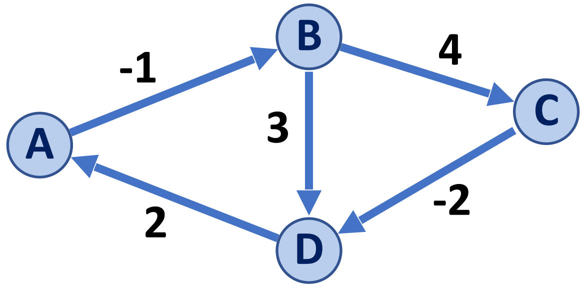
A → D 2 vs. A → C → D

B → A +∞ vs. B → C → A +∞

B → D 3 vs. B → C → D

D → A 2 vs. D → C → A +∞

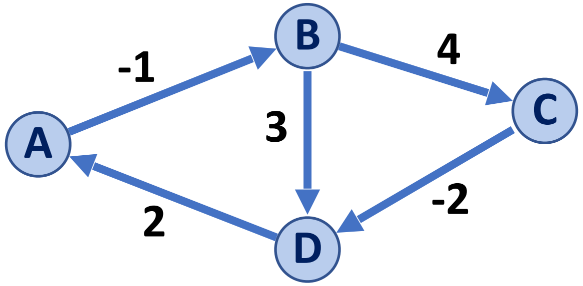
D → B 1 vs. D → C → B +∞



Floyd-Warshall Algorithm

```
FloydWarshall(G):
  Let d be a adj. matrix initialized to +inf
  foreach (Vertex v : G):
    d[v][v] = 0
  foreach (Edge (u, v) : G):
    d[u][v] = cost(u, v)

  foreach (Vertex u : G):
    foreach (Vertex v : G):
      foreach (Vertex w : G):
        if (d[u, v] > d[u, w] + d[w, v])
          d[u, v] = d[u, w] + d[w, v]
```

	A	B	C	D
A	0	-1	3	1
B	5	0	4	2
C	0	-1	0	-2
D	2	1	5	0



Floyd-Warshall Algorithm

Running time?

```
FloydWarshall(G):
  Let d be a adj. matrix initialized to +inf
  foreach (Vertex v : G):
    d[v][v] = 0
  foreach (Edge (u, v) : G):
    d[u][v] = cost(u, v)

  foreach (Vertex u : G):
    foreach (Vertex v : G):
      foreach (Vertex w : G):
        if d[u, v] > d[u, w] + d[w, v]:
          d[u, v] = d[u, w] + d[w, v]
```

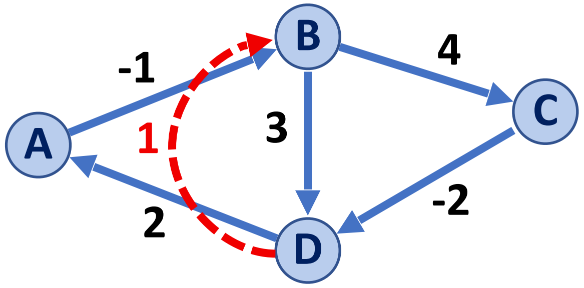
Floyd-Warshall Algorithm

We aren’t storing path information! Can we fix this?

```
FloydWarshall(G):
  Let d be a adj. matrix initialized to +inf
  foreach (Vertex v : G):
    d[v][v] = 0
  foreach (Edge (u, v) : G):
    d[u][v] = cost(u, v)

  foreach (Vertex w : G):
    foreach (Vertex u : G):
      foreach (Vertex v : G):
        if (d[u, v] > d[u, w] + d[w, v])
          d[u, v] = d[u, w] + d[w, v]
```

Floyd-Warshall Algorithm

```
FloydWarshall(G):
  Let d be a adj. matrix initialized to +inf
  foreach (Vertex v : G):
    d[v][v] = 0
    s[v][v] = 0
  foreach (Edge (u, v) : G):
    d[u][v] = cost(u, v)
    s[u][v] = v

  foreach (Vertex w : G):
    foreach (Vertex u : G):
      foreach (Vertex v : G):
        if (d[u, v] > d[u, w] + d[w, v])
          d[u, v] = d[u, w] + d[w, v]
          s[u, v] = s[u, w]
```



	A	B	C	D
A	0	-1	∞	∞
B	∞	0	4	3
C	∞	∞	0	-2
D	2	∞	∞	0

	A	B	C	D
A		B		
B			C	D
C				D
D	A			