Data Structures and Algorithms

SSSP and All Paths Shortest Path

CS 225

Brad Solomon

November 11, 2026



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

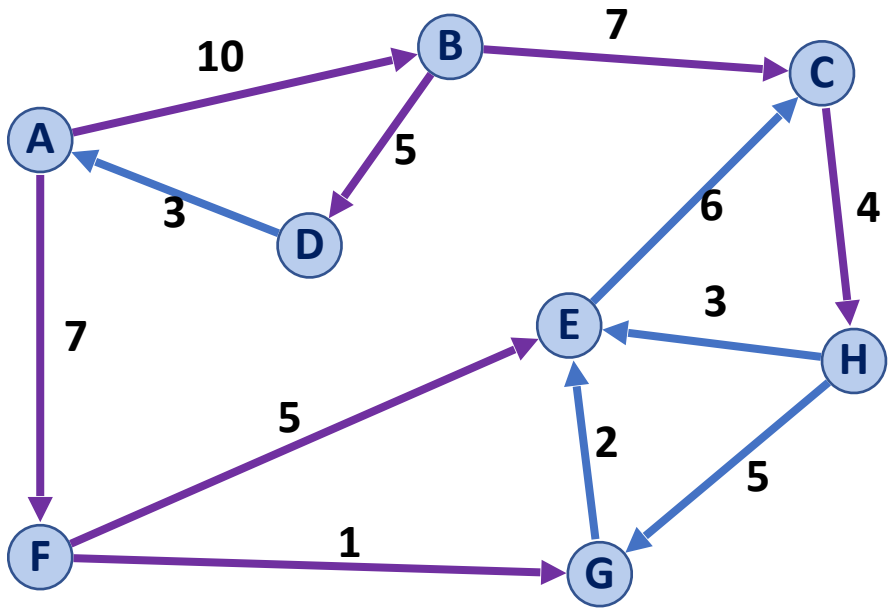
Department of Computer Science

Learning Objectives

Finalize Dijkstra's Algorithm implementation

Introduce and discuss All-Pairs Shortest Path

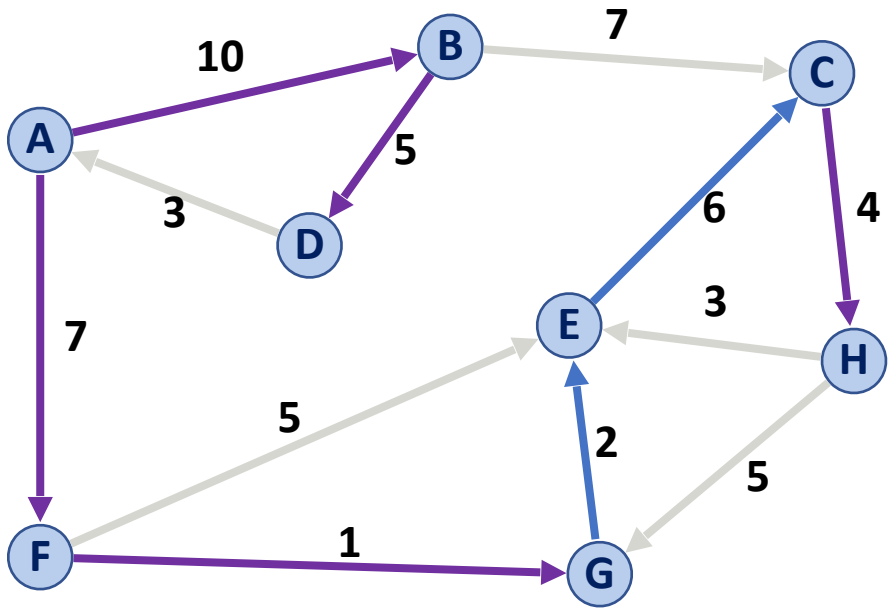
Dijkstra's Algorithm (SSSP)




```
    DijkstraSSSP(G, s):
 6    foreach (Vertex v : G.vertices()):
 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:
16      Vertex u = Q.removeMin()
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]
21          p[v] = u
```

A	B	C	D	E	F	G	H
--							
0							

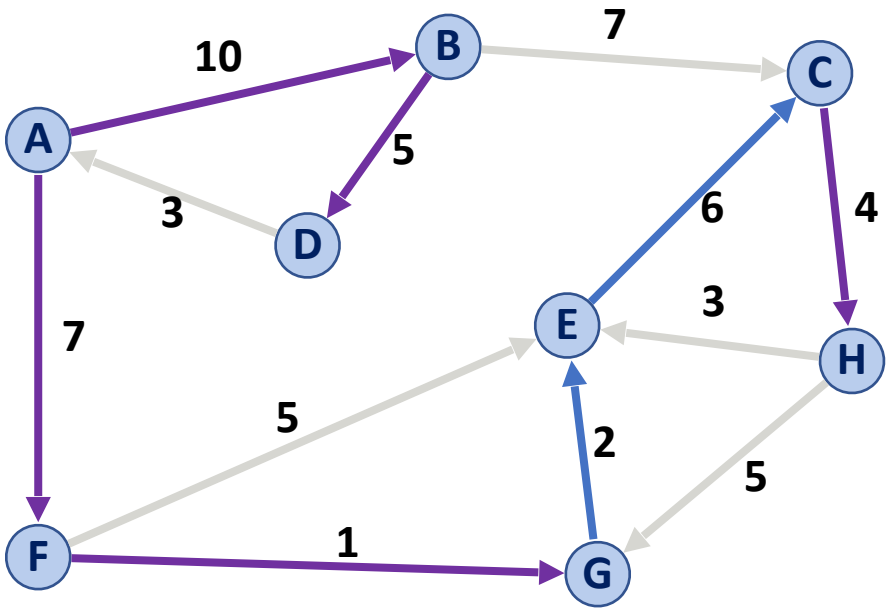
Dijkstra's Algorithm (SSSP)




```
    DijkstraSSSP(G, s):
 6    foreach (Vertex v : G.vertices()):
 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:
16      Vertex u = Q.removeMin()
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]
21          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)



Whats the point of predecessor?

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)

What is the running time of Dijkstra's Algorithm?

```
    DijkstraSSSP(G, s):
 6    foreach (Vertex v : G):
 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:
16      Vertex u = Q.removeMin()
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]
21          p[v] = m
22
23    return T
```

Dijkstra's Algorithm (SSSP)

What is the running time of Dijkstra's Algorithm?

O(m + n log n)

The same as Prim's!

(Times here are minheap)

6-9: O(n)

11-12: O(n)

15: repeat below n x

16-22: O(log n)

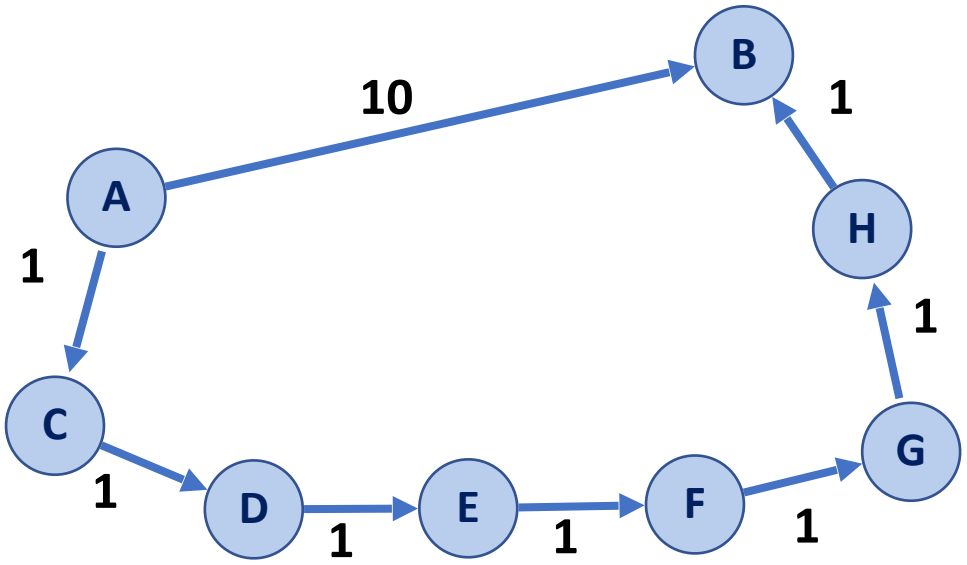
[w/ Fib Heap O(1) updates]

```
    DijkstraSSSP(G, s):
 6    foreach (Vertex v : G):
 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:
16      Vertex u = Q.removeMin()
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]
21          p[v] = m
22
23    return T
```

Dijkstra's Algorithm (SSSP)

Claim: Dijkstras will always visit a node through its optimal shortest path.

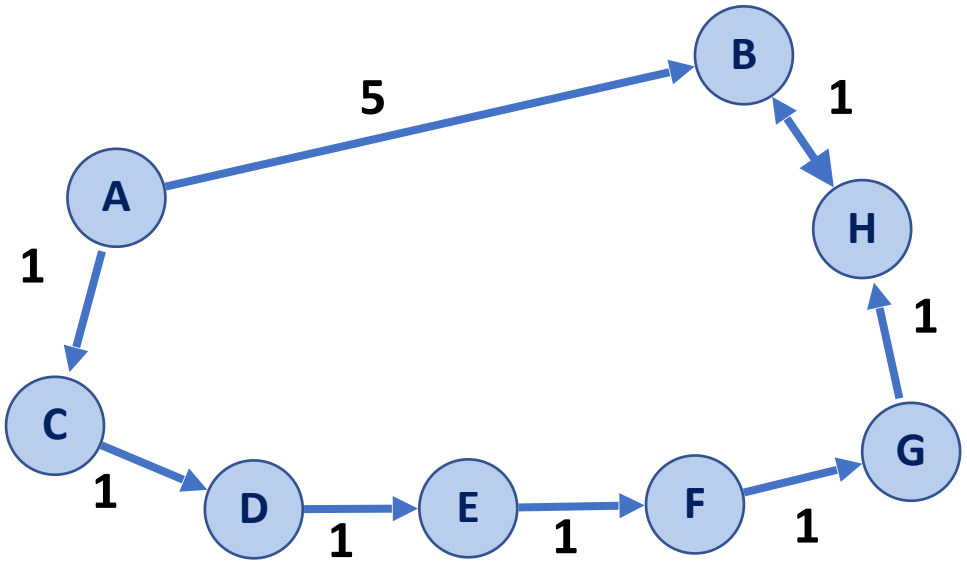
When we will visit B in the following graph?



Dijkstra's Algorithm (SSSP)

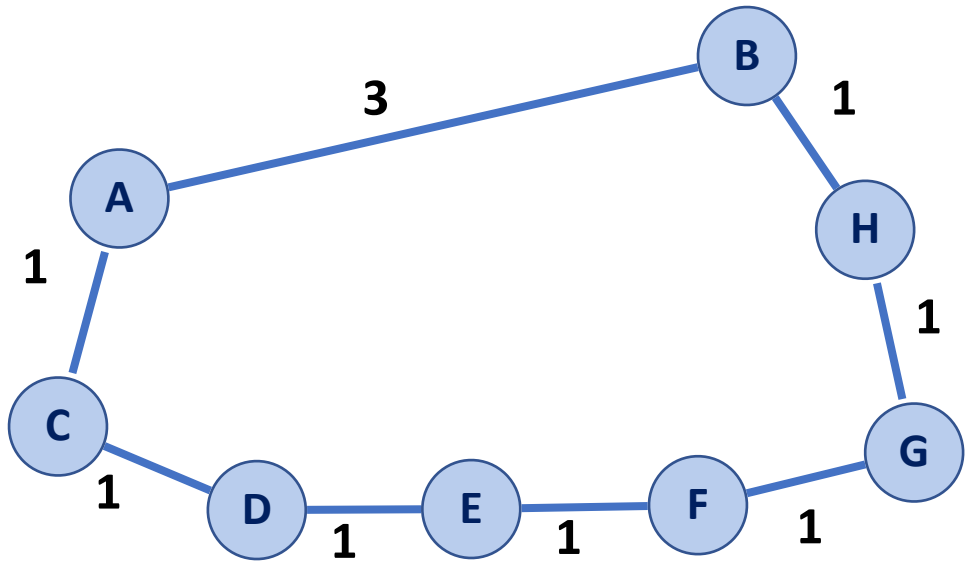
Claim: Dijkstras will always visit a node through its optimal shortest path.

When we will visit H in the following graph?



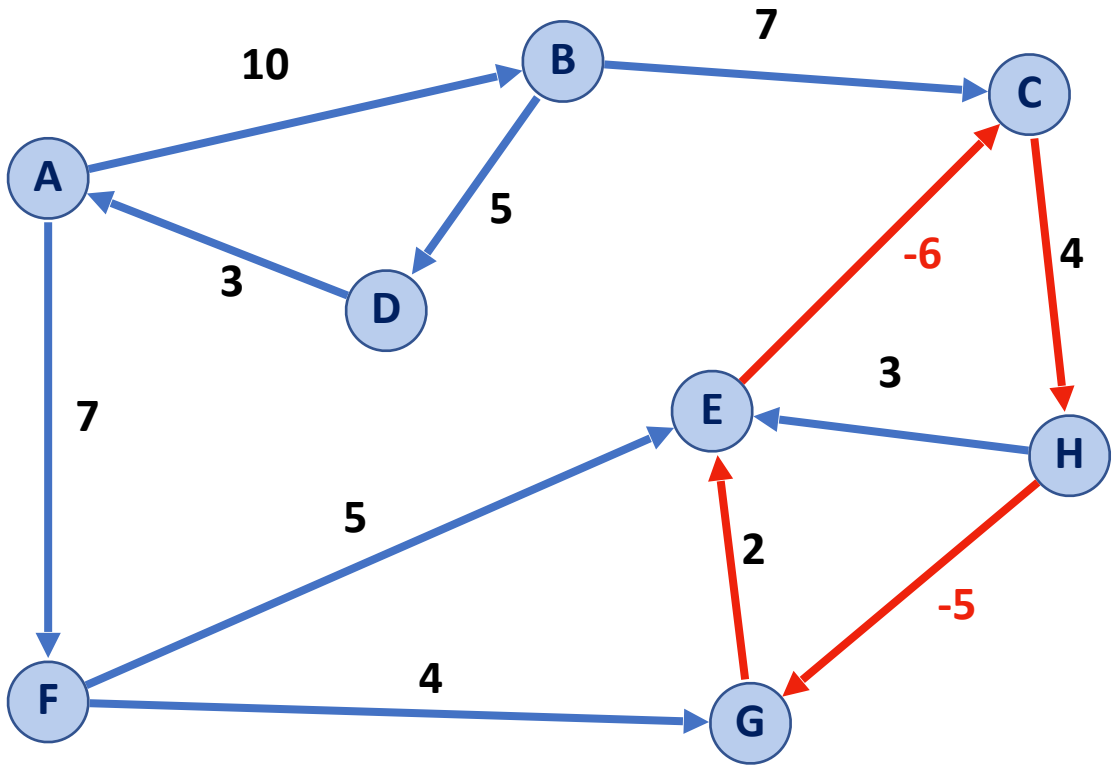
Dijkstra's Algorithm (SSSP)

How does Dijkstra's algorithm handle undirected graphs?



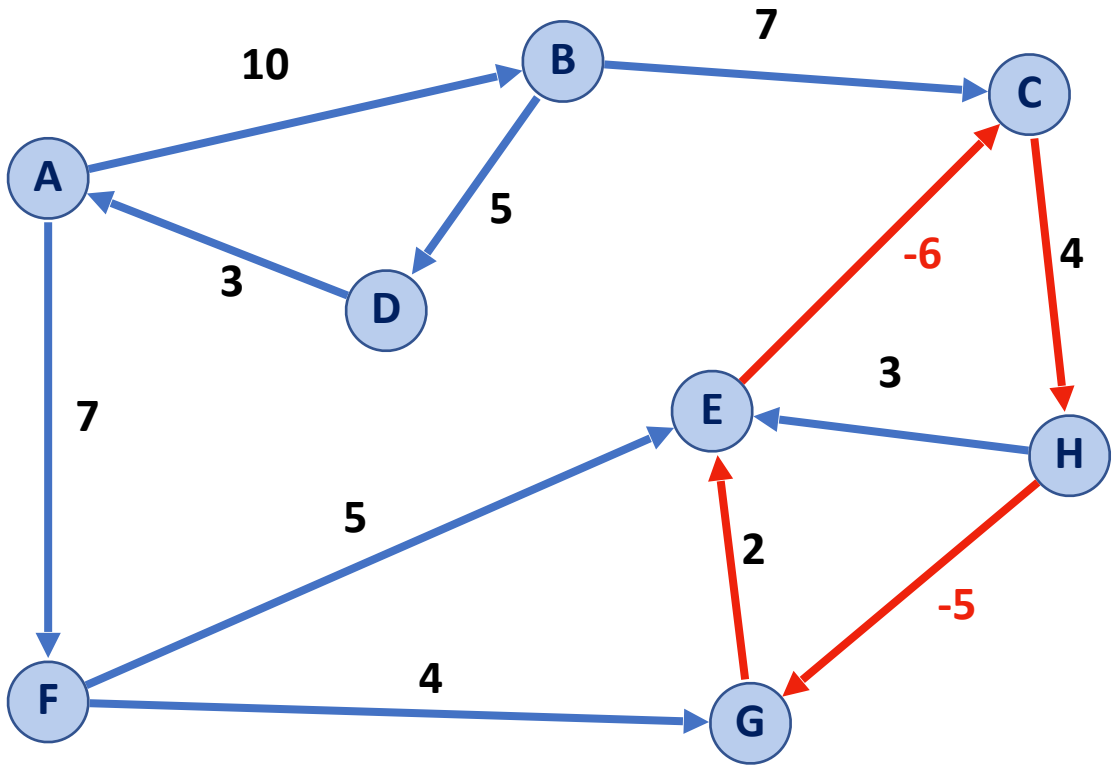
Dijkstra's Algorithm (SSSP)

How does Dijkstras handle a negative weight cycle?



Dijkstra's Algorithm (SSSP)

How does Dijkstras handle a negative weight cycle?



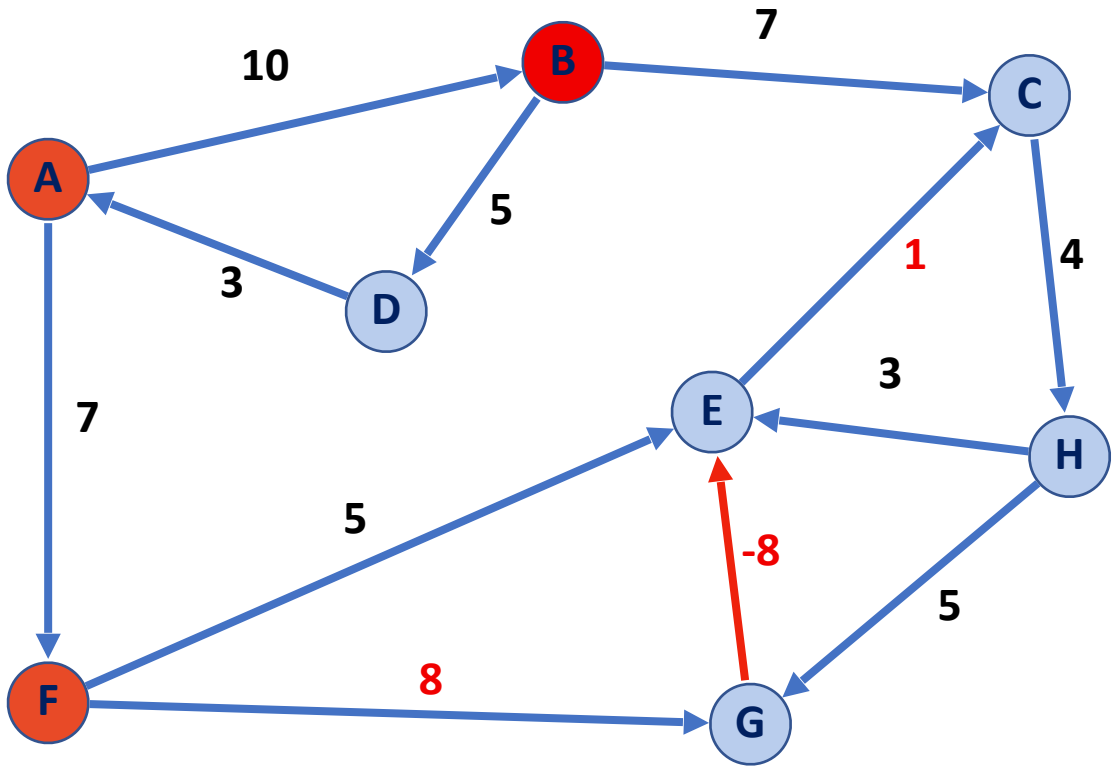
Shortest Path (A → E): A → F → E → (C → H → G → E)*

Length: 12

Length: -5 (repeatable)

Dijkstra's Algorithm (SSSP)

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



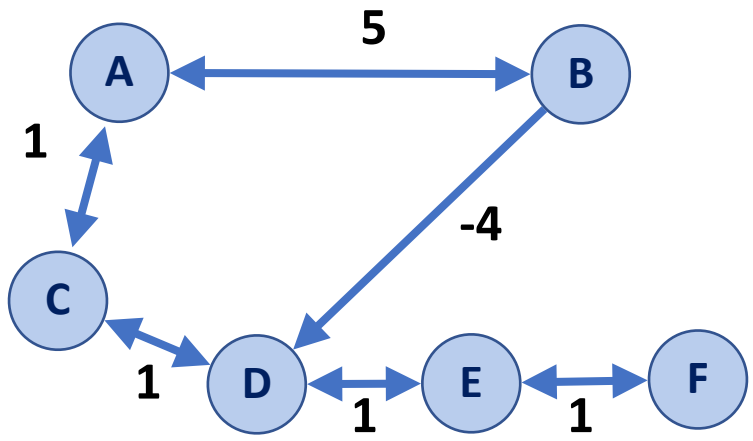
A	B	C	D	E	F	G	H
--	A	B	B	F	A	F	--
0	10	17	15	12	7	15	∞

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
--					
0					



Dijkstra's Algorithm (SSSP)

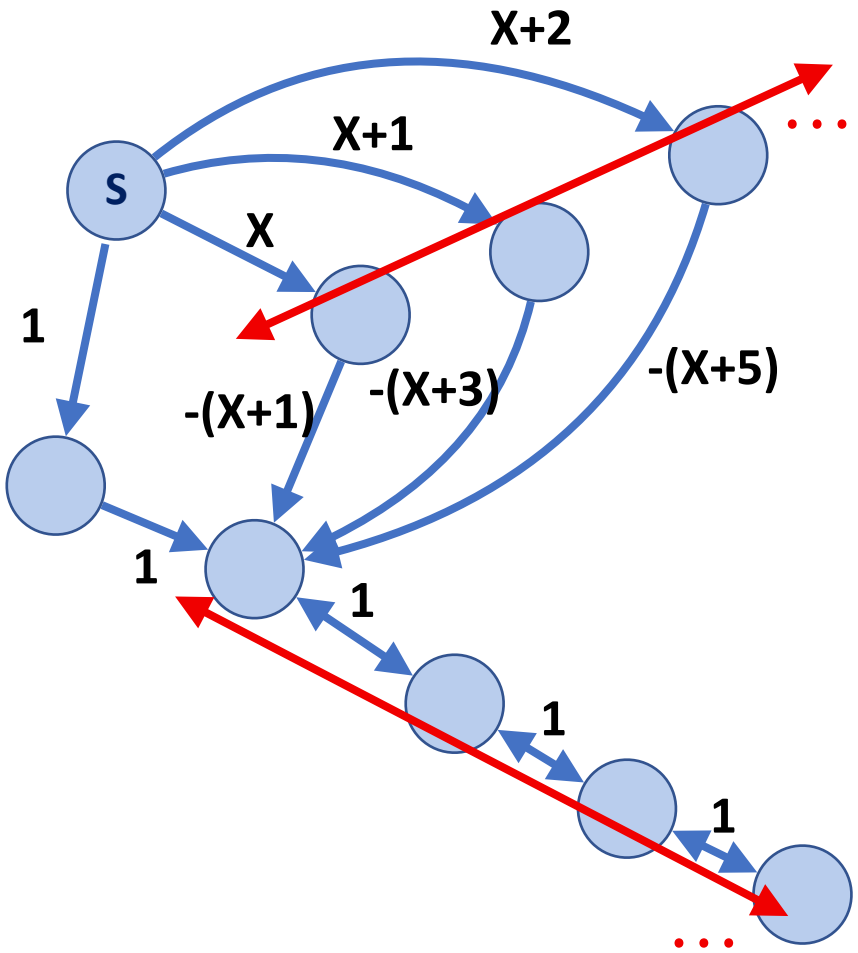
Recalculating all distances is possible, but algorithm runtime is very bad!

```
    DijkstraSSSP(G, s):
 6    foreach (Vertex v : G):
 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 until Q.empty():
16      Vertex u = Q.removeMin()
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]
21          p[v] = m
22          if v not in Q:
23             Q.push(v)
24    return T
```

Dijkstra's Algorithm (SSSP)

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

Worst case: ~n/2 nodes each updating ~n/2 nodes distances




```
    DijkstraSSSP(G, s):
 6    foreach (Vertex v : G):
 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 until Q.empty():
16      Vertex u = Q.removeMin()
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]
21          p[v] = m
22          if v not in Q:
23             Q.push(v)
24    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:

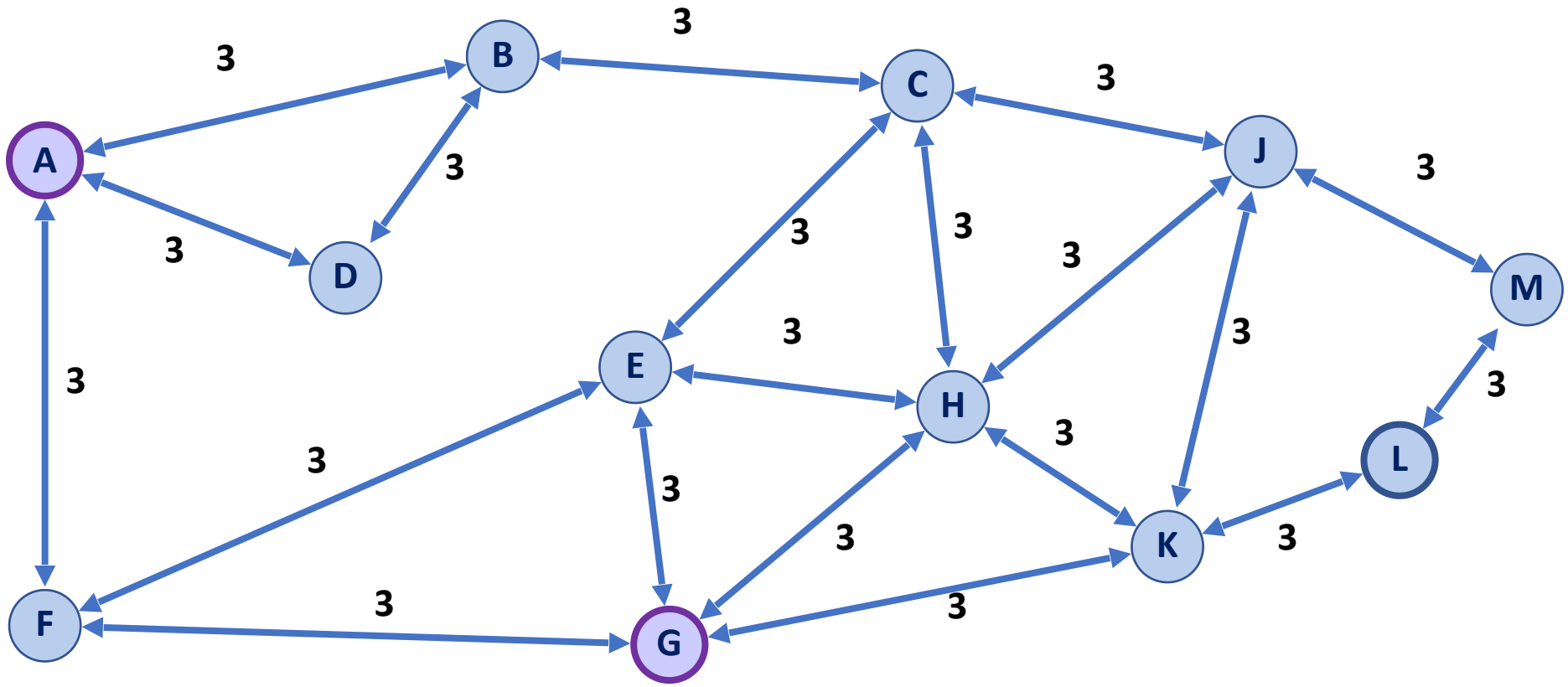
Sparse: O(m + n log n)

Dense: $O(n^2)$

```
    DijkstraSSSP(G, s):
 6    foreach (Vertex v : G):
 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:
16      Vertex u = Q.removeMin()
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]
21          p[v] = m
22
23    return T
```

Landmark Path Problem

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



Floyd-Warshall Algorithm

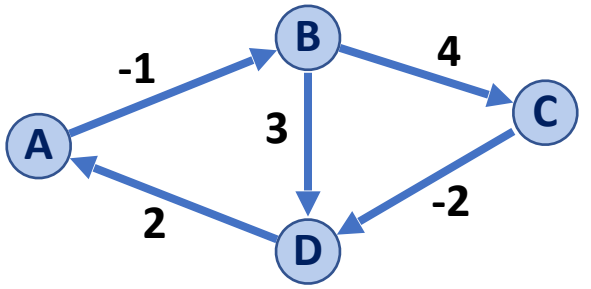
Floyd-Warshall's Algorithm is an alternative to Dijkstra in the presence of negative-weight edges (not negative weight cycles).

```
 1  FloydWarshall(G):
 2    Let d be a adj. matrix initialized to +inf
 3    foreach (Vertex v : G):
 4      d[v][v] = 0
 5    foreach (Edge (u, v) : G):
 6      d[u][v] = cost(u, v)
 7
 8    foreach (Vertex u : G):
 9      foreach (Vertex v : G):
10        foreach (Vertex w : G):
11          if (d[u, v] > d[u, w] + d[w, v])
12            d[u, v] = d[u, w] + d[w, v]
```

Floyd-Warshall Algorithm

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

	A	B	C	D
A				
B				
C				
D				

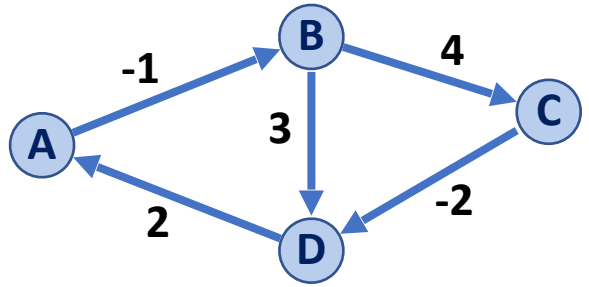


Floyd-Warshall Algorithm

```
 8    foreach (Vertex w : G):
 9      foreach (Vertex u : G):
10        foreach (Vertex v : G):
11          if (d[u, v] > d[u, w] + d[w, v])
12            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

```
 8   foreach (Vertex w : G):
 9     foreach (Vertex u : G):
10       foreach (Vertex v : G):
11         if (d[u, v] > d[u, w] + d[w, v])
12           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

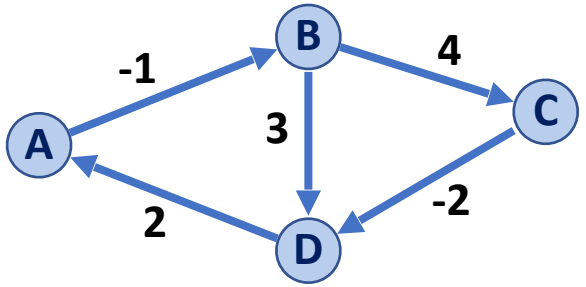


u=A, v=B



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

```
 8   foreach (Vertex w : G):
 9     foreach (Vertex u : G):
10       foreach (Vertex v : G):
11         if (d[u, v] > d[u, w] + d[w, v])
12           d[u, v] = d[u, w] + d[w, v]
```

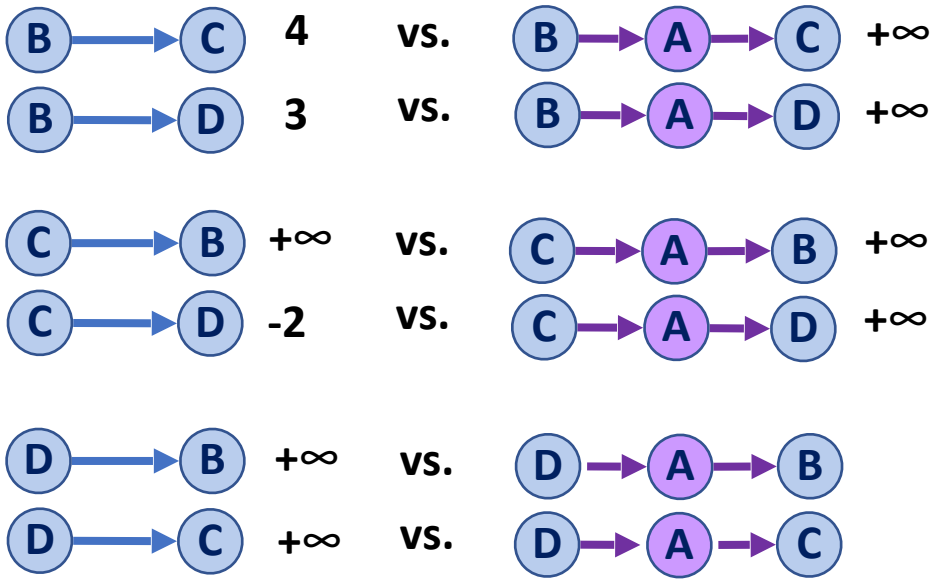
Let **w** be midpoint

Let **u** be start point

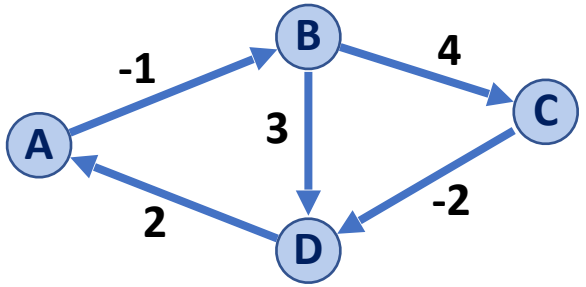
Let **v** be end point

Is our distance shorter now?

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



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



Floyd-Warshall Algorithm

```
 8   foreach (Vertex w : G):
 9     foreach (Vertex u : G):
10       foreach (Vertex v : G):
11         if (d[u, v] > d[u, w] + d[w, v])
12           d[u, v] = d[u, w] + d[w, v]
```

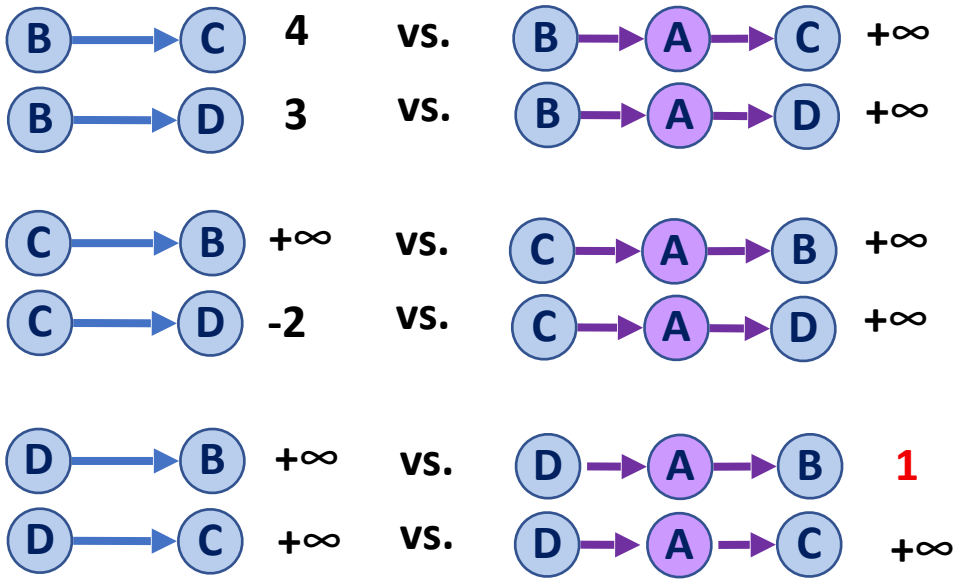
Let **w** be midpoint

Let **u** be start point

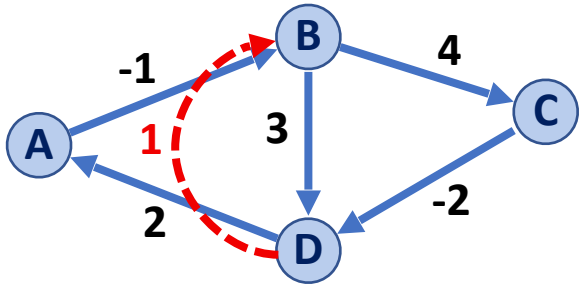
Let **v** be end point

Is our distance shorter now?

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



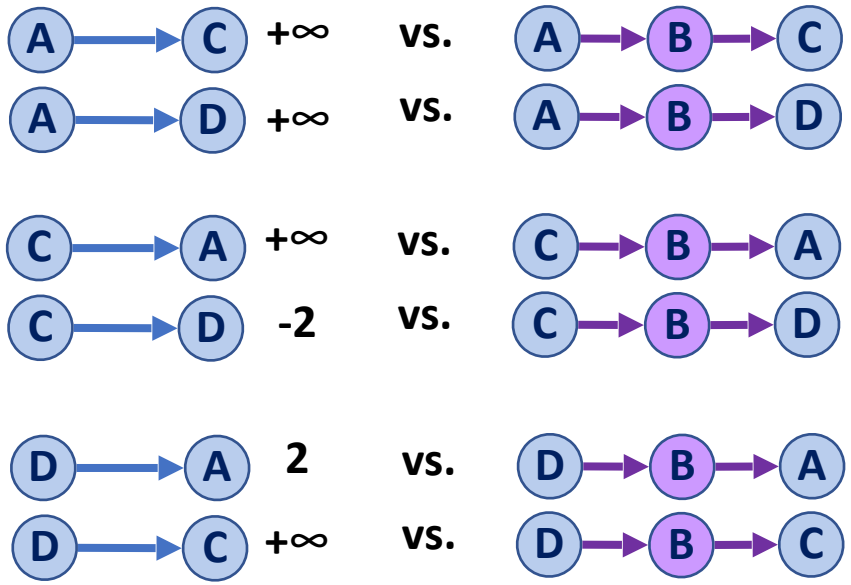
	A	B	C	D
A	0	-1	∞	∞
B	∞	0	4	3
C	∞	∞	0	-2
D	2	1	∞	0



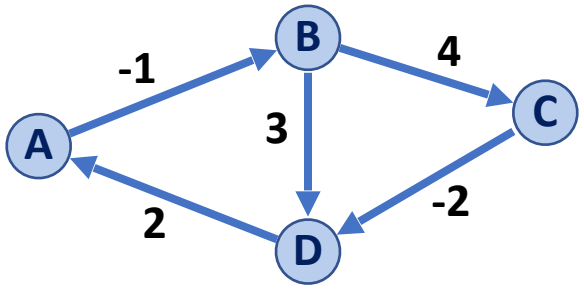
Floyd-Warshall Algorithm

```
 8   foreach (Vertex w : G):
 9     foreach (Vertex u : G):
10       foreach (Vertex v : G):
11         if (d[u, v] > d[u, w] + d[w, v])
12           d[u, v] = d[u, w] + d[w, v]
```

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



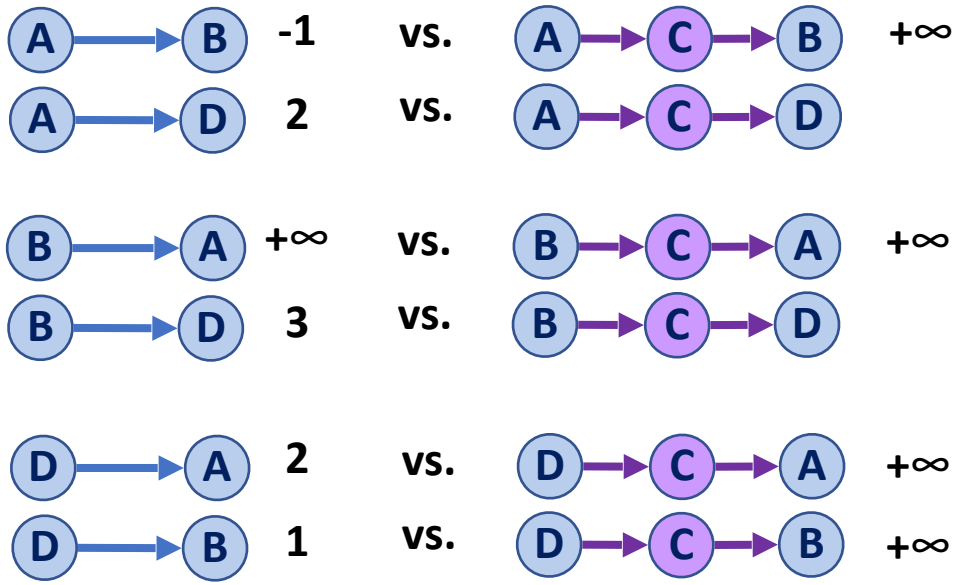
	A	B	C	D
A	0	-1	∞	∞
B	∞	0	4	3
C	∞	∞	0	-2
D	2	1	∞	0



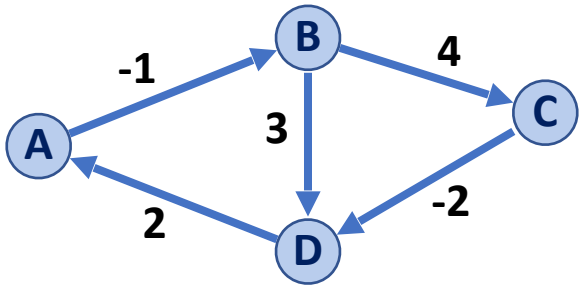
Floyd-Warshall Algorithm

```
 8   foreach (Vertex w : G):
 9     foreach (Vertex u : G):
10       foreach (Vertex v : G):
11         if (d[u, v] > d[u, w] + d[w, v])
12           d[u, v] = d[u, w] + d[w, v]
```

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



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

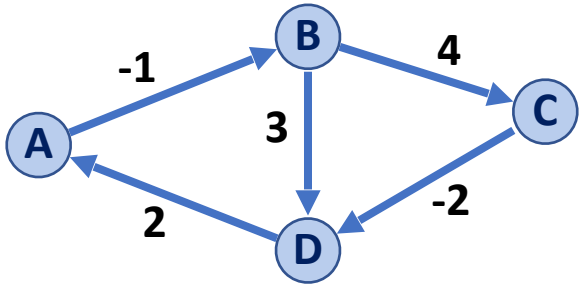


Floyd-Warshall Algorithm



```
 1  FloydWarshall(G):
 2    Let d be a adj. matrix initialized to +inf
 3    foreach (Vertex v : G):
 4      d[v][v] = 0
 5    foreach (Edge (u, v) : G):
 6      d[u][v] = cost(u, v)
 7
 8    foreach (Vertex u : G):
 9      foreach (Vertex v : G):
10        foreach (Vertex w : G):
11          if (d[u, v] > d[u, w] + d[w, v])
12            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):
 6    Let d be a adj. matrix initialized to +inf
 7    foreach (Vertex v : G):
 8      d[v][v] = 0
 9    foreach (Edge (u, v) : G):
10      d[u][v] = cost(u, v)
11
12    foreach (Vertex u : G):
13      foreach (Vertex v : G):
14        foreach (Vertex w : G):
15          if d[u, v] > d[u, w] + d[w, v]:
16            d[u, v] = d[u, w] + d[w, v]
```

We have only scratched the surface on graphs!

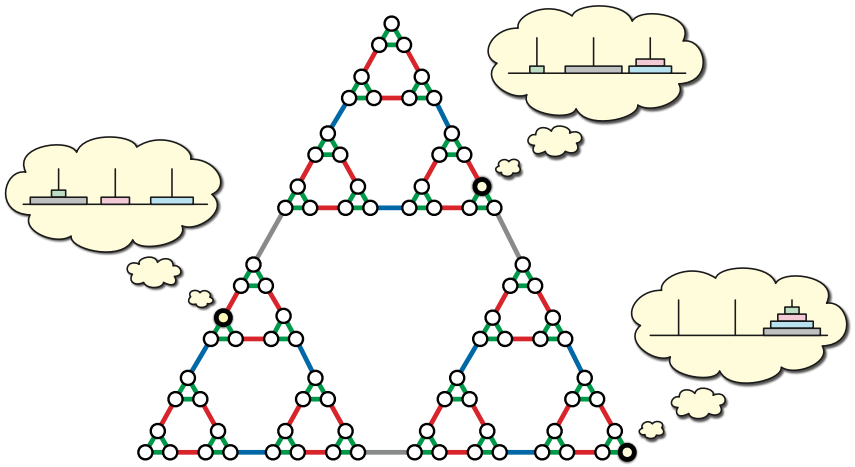


Image from Jeff Erickson Algorithms First Edition

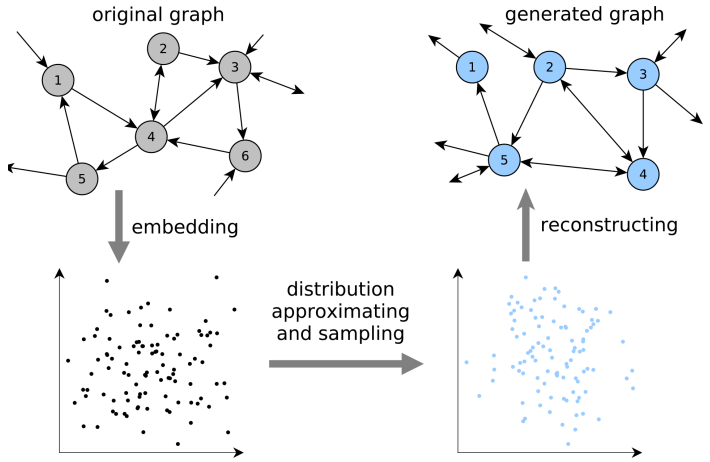
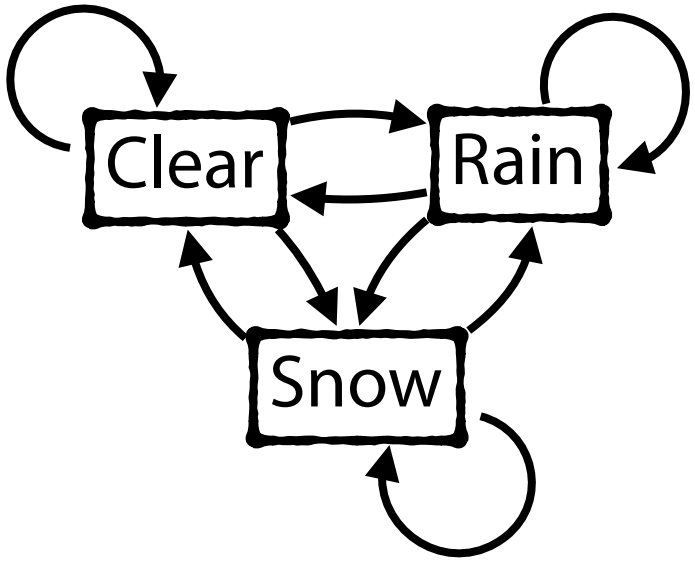


Image from Drobyshevskiy et al. **Random graph modeling: A survey of the concepts.** 2019



$$M = \begin{pmatrix} .5 & .3 & .2 \\ .5 & .4 & .1 \\ .2 & .1 & .7 \end{pmatrix}$$

Randomized Algorithms

A **randomized algorithm** is one which uses a source of randomness somewhere in its implementation.

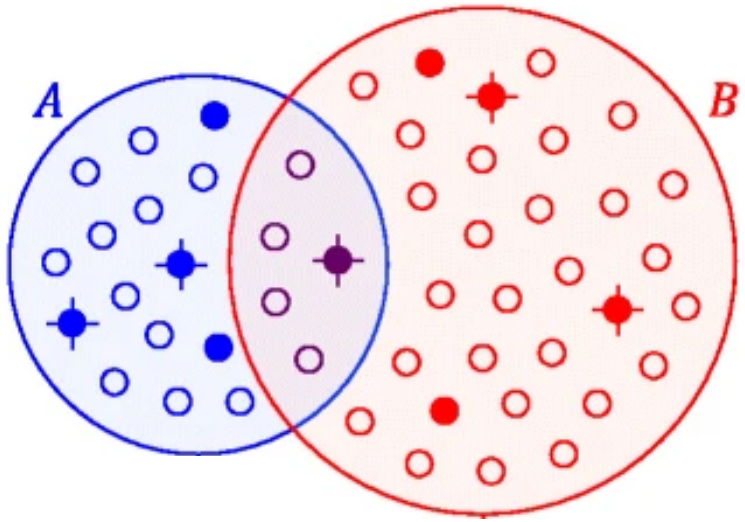
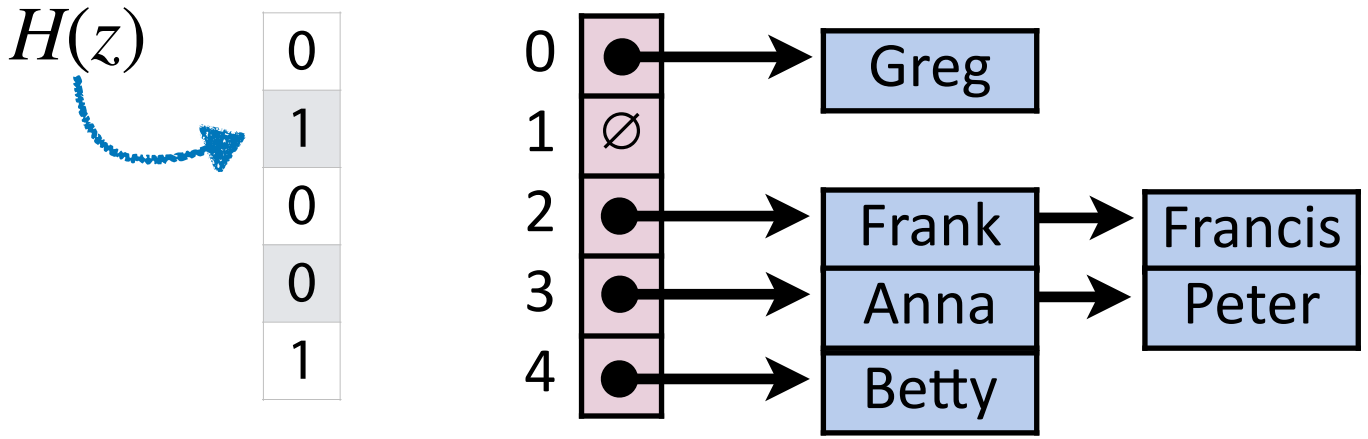


Figure from Ondov et al 2016



$H(x)$	0	2	1	0	0	4	0	2	0	6
$H(y)$	1	0	2	3	1	0	3	4	0	1
$H(z)$	2	1	0	2	0	1	0	0	7	2

A faulty list

Imagine you have a list ADT implementation ***except***…

Every time you called **insert**, it would fail 50% of the time.

Quick Primes with Fermat's Primality Test

If p is prime and a is not divisible by p, then $a^{p-1} \equiv 1 \pmod{p}$

But… ***sometimes*** if n is composite and $a^{n-1} \equiv 1 \pmod{n}$

Probabilistic Accuracy: Fermat primality test

	$a^{p-1} \equiv 1 \pmod{p}$	$a^{p-1} \not\equiv 1 \pmod{p}$
p is prime		
p is not prime		

Probabilistic Accuracy: Fermat primality test

Let's assume $\alpha = .5$

First trial: $a = a_0$ and prime test returns 'prime!'

Second trial: $a = a_1$ and prime test returns 'prime!'

Third trial: $a = a_2$ and prime test returns 'not prime!'

Is our number prime?

What is our **false positive** probability? Our **false negative** probability?

Probabilistic Accuracy: Fermat primality test



Summary: Randomized algorithms can also have fixed (or bounded) runtimes at the cost of probabilistic accuracy.

Randomness:

Assumptions: