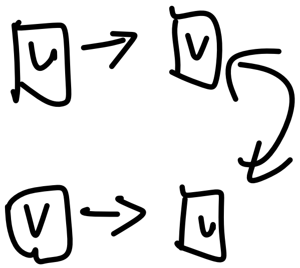
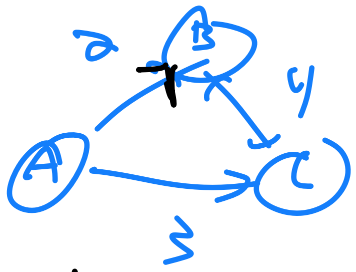


Data Structures

Shortest Path 2 (All Paths!)



CS 225
Brad Solomon & G Carl Evans

December 4, 2023

	A	B	C
A		2	3
B	∞		4
C	∞	4	

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

Last Lecture!

Wednesday is review day. Prepare questions!

Can also post questions ahead of time (so I can prep slides)

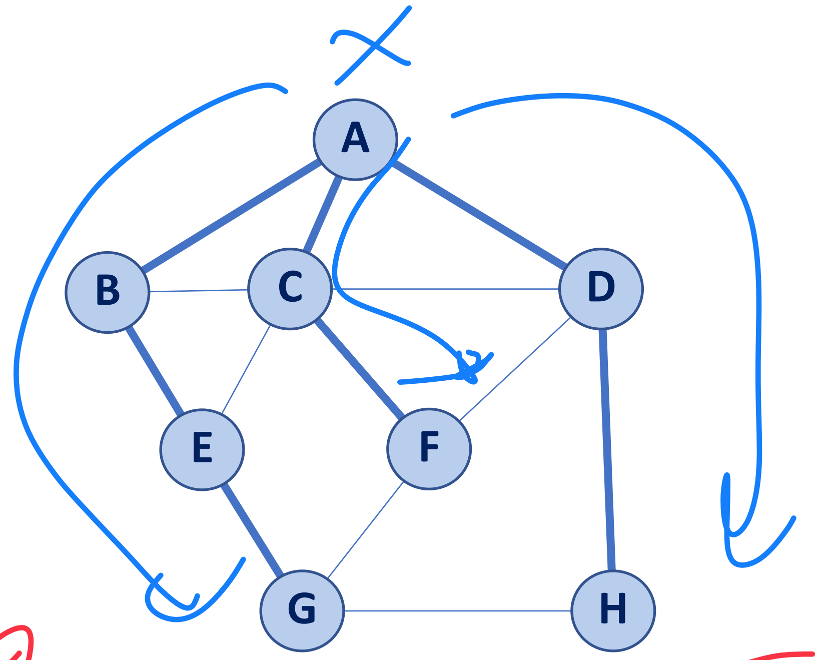
↳ lectures channel

Learning Objectives

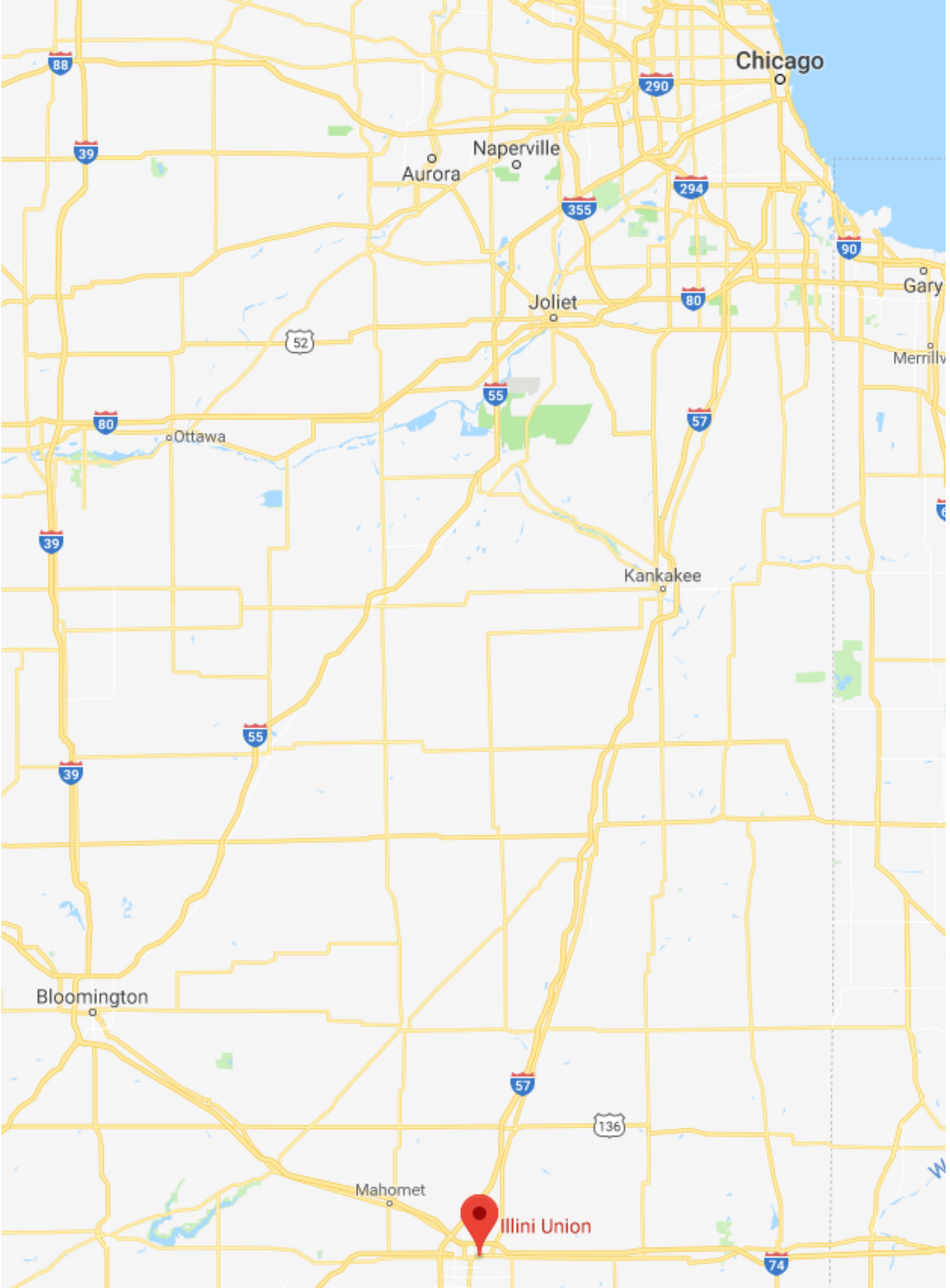
Calculate runtime of Dijkstras Algorithm

Introduce Bellman-Ford as an alternative to shortest path

Shortest Path



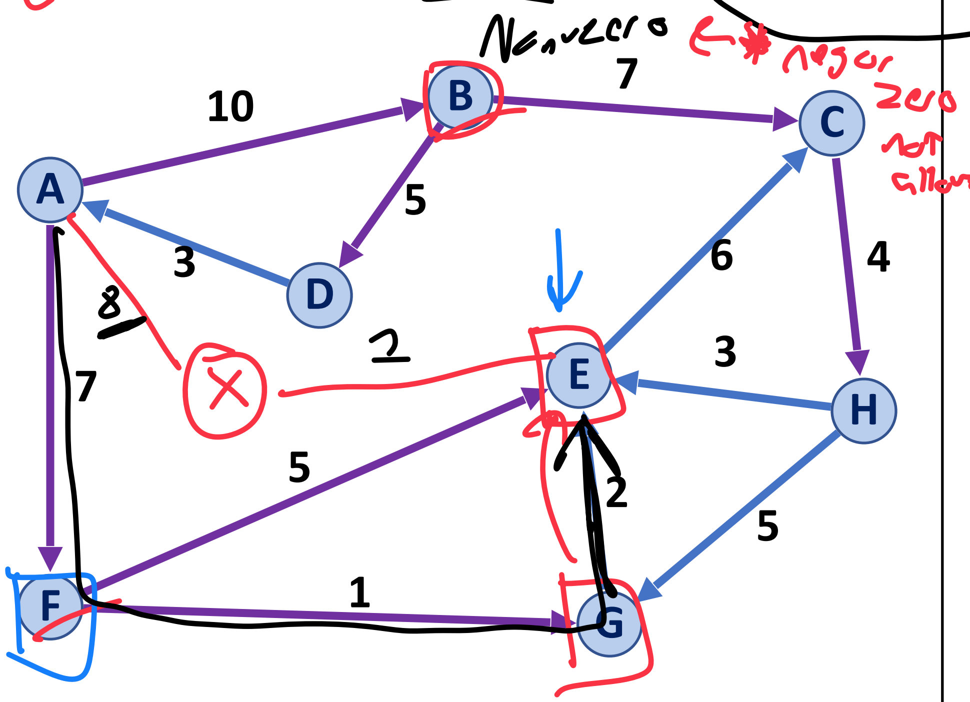
BFS or ~~DFS~~



Dijkstra's Algorithm (SSSP)

Cost(A, x) + cost(X, E) < Cost(A, G) + cost(G, E)

10

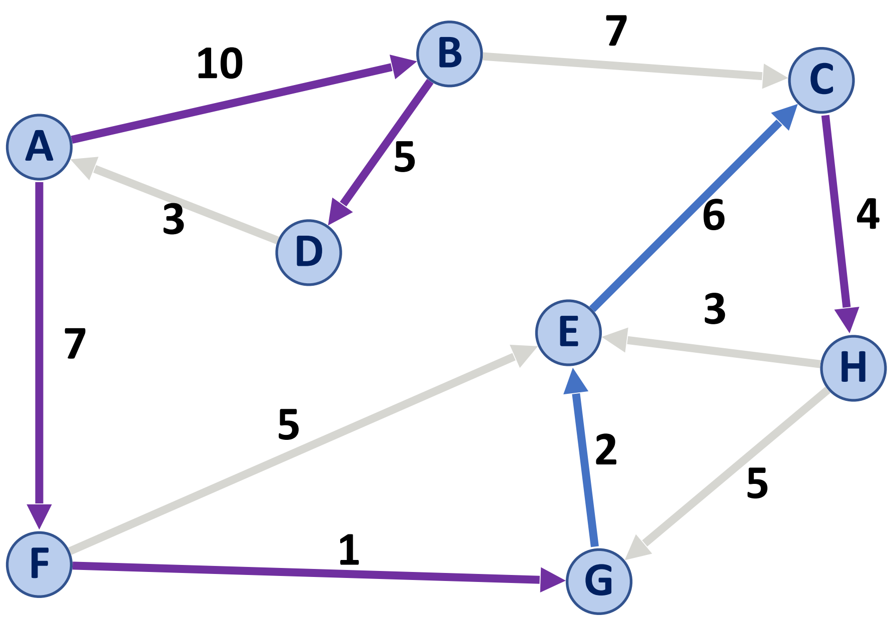



```
DijkstraSSSP(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  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
```

remove the min element

A	B	C	D	E	F	G	H
--	A		B	FG	A	F	
0	10	∞	~~∞~~ 15	~~∞~~ 10	7	~~∞~~ 8	∞

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)

What is the running time of Dijkstra's Algorithm?

Running time for Prim

using Fib heap

$O(m + n \log n)$

$m \log n$ or m

min heap	fib heap
$O(\log n)$	$O(\log n)$
$O(\log n)$	$O(1)$

remove min

updating / decreasing key

```
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
```

$O(n)$

$O(n)$

$n \times$

$O(\log n) \to O(n)$

total of m edges

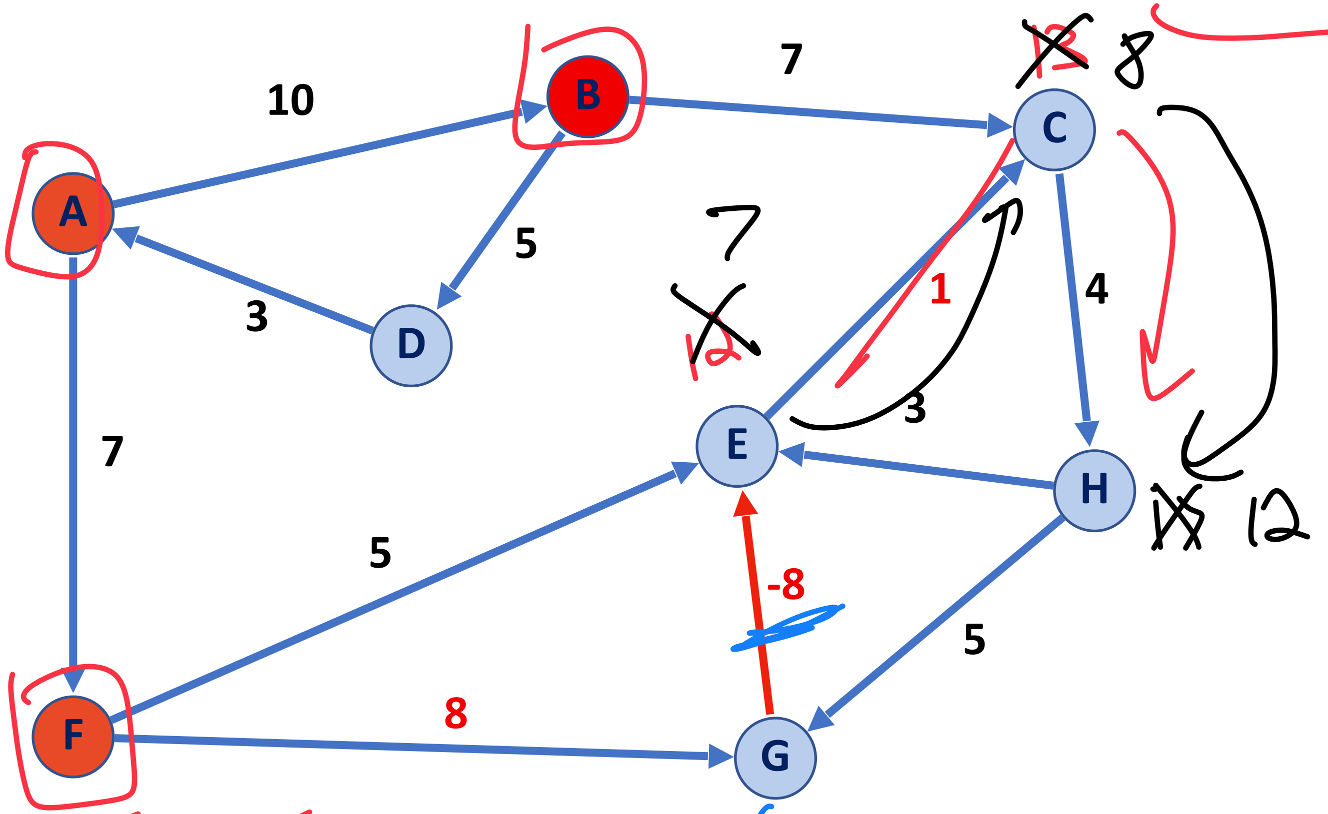
$O(1)$ Fib heap

$O(\log n)$

Dijkstra's Algorithm (SSSP)

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

↳ Not allowed



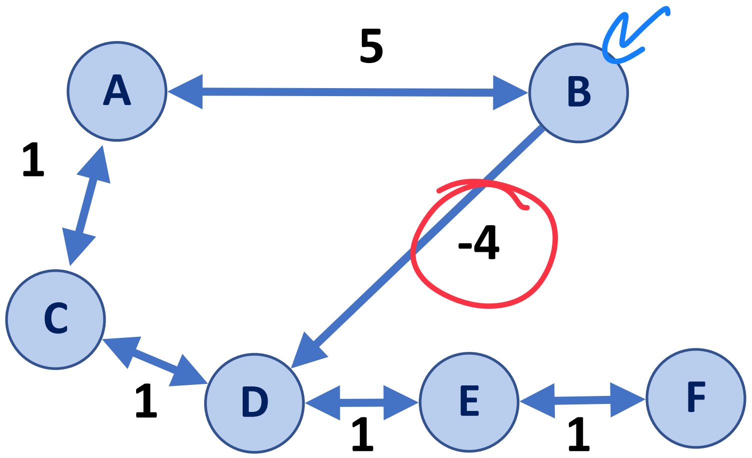
A	B	C	D	E	F	G	H
--	A	~~B~~ E	B	F	A	F	~~--~~ C
0	10	~~17~~ 13	15	12	7	15	~~∞~~ 17

Dijkstra's Algorithm (SSSP)

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

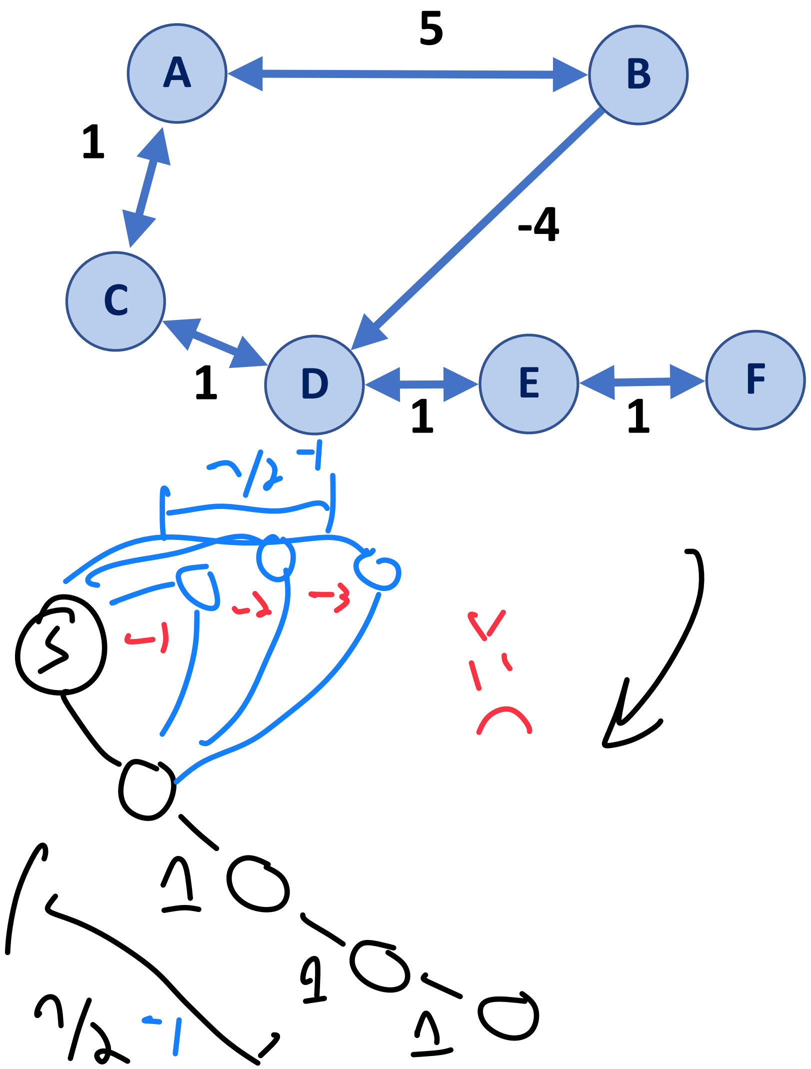
Negative weights break this assumption!

A	B	C	D	E	F
--	A	A	~~C~~ B	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!

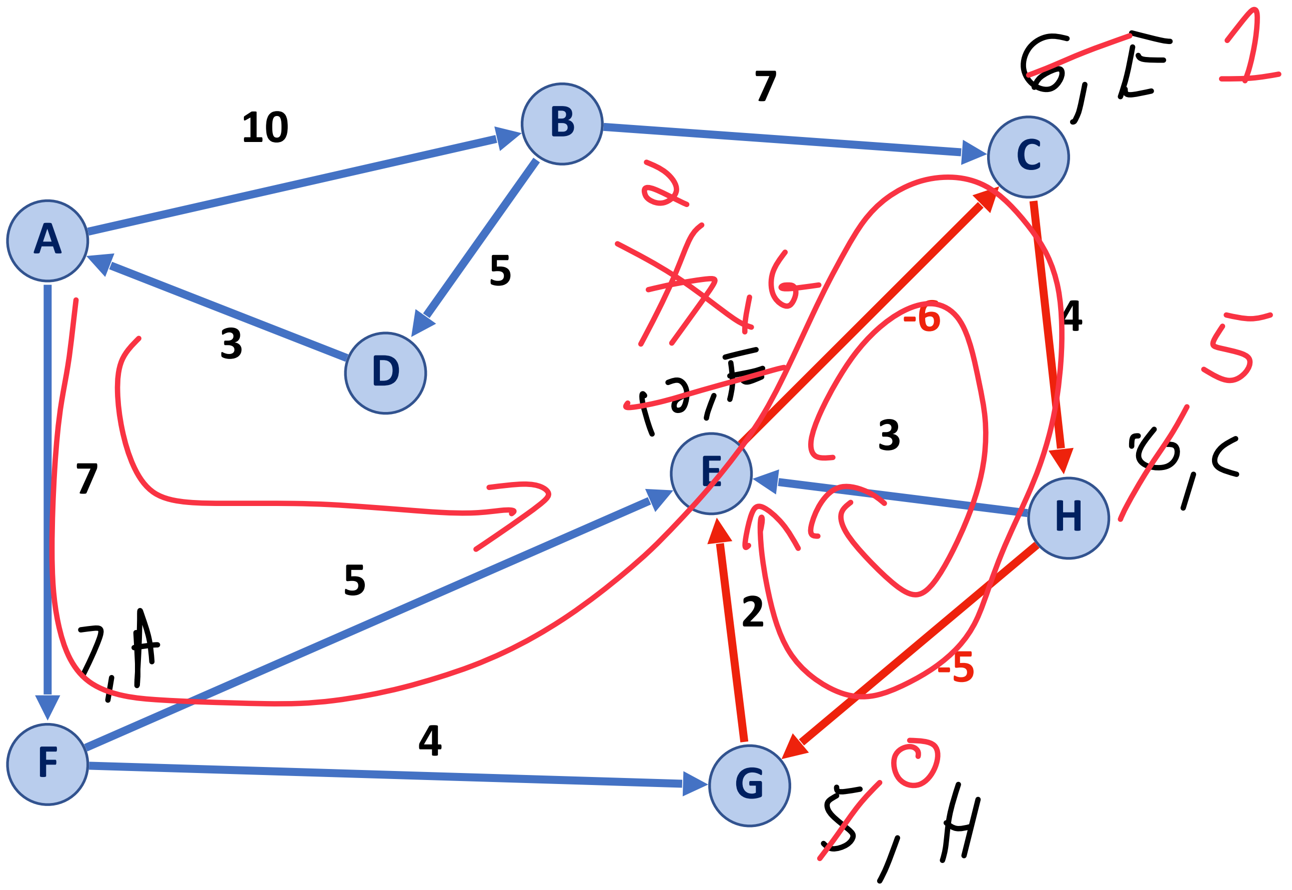



```
DijkstraSSSP(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  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
```

re add to queue

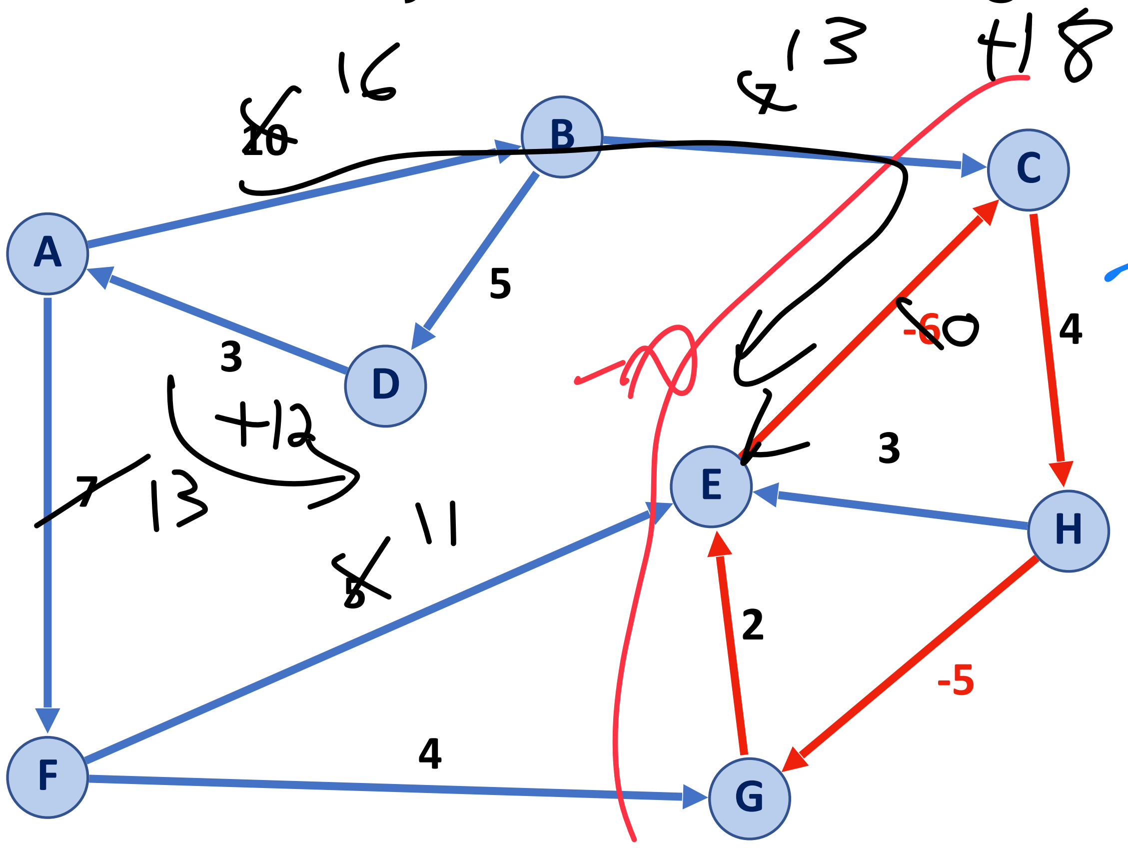
Dijkstra’s Algorithm (SSSP)

How does Dijkstras handle a negative weight cycle?



Dijkstra's Algorithm (SSSP)

How does Dijkstras handle a negative weight cycle?



Can't we normalize edges? → +6 to everything

↳ this is bad b/c it changes performance

↳ Don't do this!

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

Length: 12

Length: -5 (repeatable)

Dijkstra's Algorithm (SSSP)

Dijkstras Algorithm works efficiently only on non-negative weights* No neg cycles allowed

Optimal implementation:

Fib heap

On dense graph ties unsorted list

Optimal runtime:

$O(m + n \log n)$

$O(n^2)$

```
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
```

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 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]
```

All paths shortest Path

Dynamic program

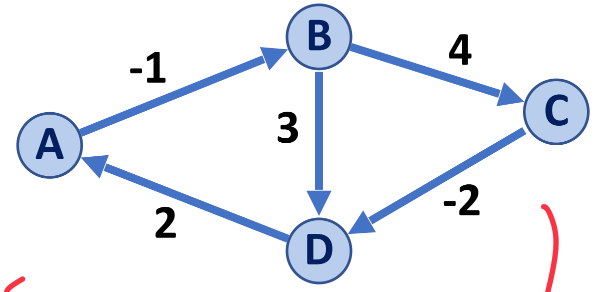
Floyd-Warshall Algorithm

```
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)
```

↳ Adj Matrix

cost(u,v) = ∞
if no edge

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



↳ as adj
matrix

Floyd-Warshall Algorithm

imagine if I took path through w (midpoint)

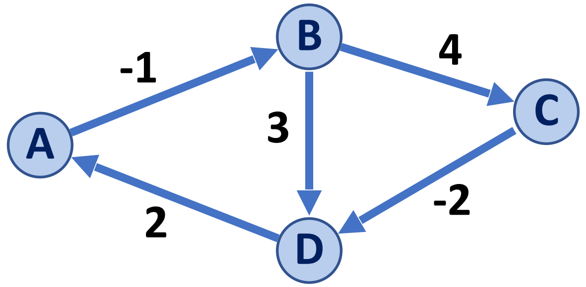
```
foreach (Vertex w : G):
  foreach (Vertex u : G):     — start point
    foreach (Vertex v : G):   — end point
      if (d[u, v] > d[u, w] + d[w, v])
        d[u, v] = d[u, w] + d[w, v]
```

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

Let us consider comparisons where w = A:

w=A, u=A, v=A   0 > 0+0 ✗

, v=B   -1 > 0+-1 ✗

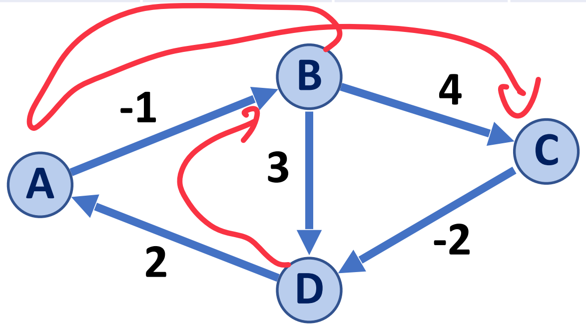


Don't consider cases where w = u or w = v

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 = A (and u != w and v != w):

B → C 4 vs. B $\xrightarrow{\infty}$ A → C ∞

B → D 3 vs. B → A → D ∞

C → B ∞ vs. C $\xrightarrow{\infty}$ A → B ∞

C → D -2 vs. C → A → D ∞

D → B ∞ vs. D $\xrightarrow{2}$ A $\xrightarrow{-1}$ B 1

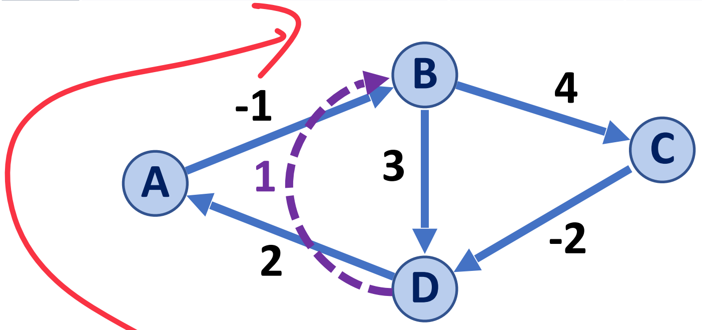
D → C ∞ vs. D → A → C ∞

Start end

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



Memoization

D-A-B

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 +∞

Floyd-Warshall Algorithm

```
foreach (Vertex w : G):        ← 2nd iteration
  foreach (Vertex u : G):      ← start
    foreach (Vertex v : G):    ← end
      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	~~∞~~ C	0

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

A → C +∞ vs. A → B → C -1+4 = 3

A → D +∞ vs. A → B → D -1+3 = 2

C → A +∞ vs. C → B → A ∞ ∞

C → D -2 vs. C → B → D ∞ ∞

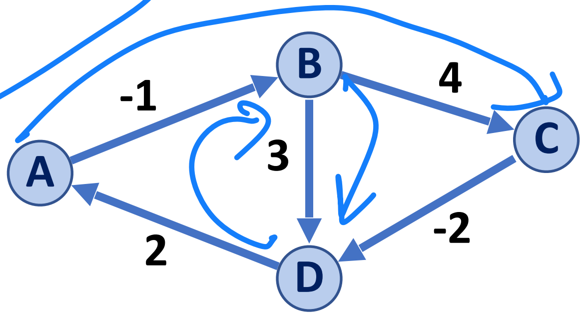
D → A 2 vs. D → B → A 2+∞ ∞ x

D → C +∞ vs. $\boxed{D \to B}$ → C 2 2+4 = 5

D → A → B → C

Not B

u v



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]
```

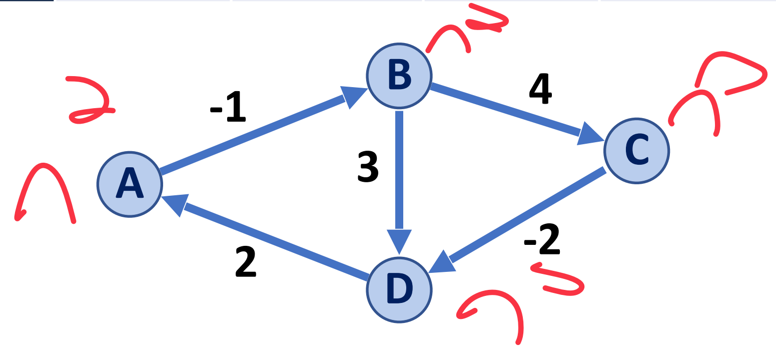
	A	B	C	D
A	0	-1	3	2
B	∞	0	4	3
C	∞	∞	0	-2
D	2	**1**	**5**	0

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 +∞



D iteration not shown
(skip to end)

Floyd-Warshall Algorithm

All path shortest path

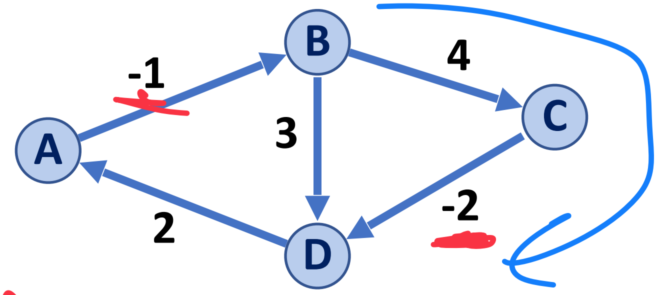
End matrix]

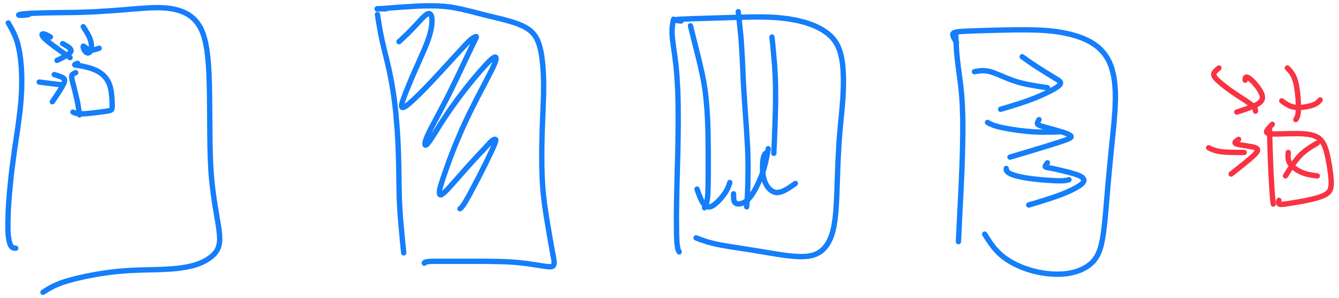
```
FloydWarshall(G):
  Let d be a adj. matrix initialized to +inf
  foreach (Vertex v : G):
    d[v][v] = 0
  foreach (Edge (u, v) : G):
    d[u][v] = cost(u, v)

  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]
```

u v w vs w u 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

↳ The order doesn't matter as long as consistent





Runtime?

Floyd-Warshall Algorithm

Running time?

$O(n^3)$, easy to code!

→ easy to multithread

↳ textbook dynamic program

```
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):                      n x
    foreach (Vertex u : G):                    n x
      foreach (Vertex v : G):                  n x
        if (d[u, v] > d[u, w] + d[w, v])
          d[u, v] = d[u, w] + d[w, v]          O(1)
```

$O(n)$

$O(m)$

$O(n^3)$

Floyd-Warshall Algorithm

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

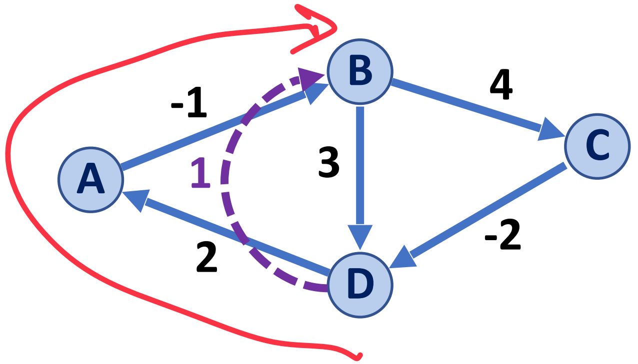
```
FloydWarshall(G):
  Let d be a adj. matrix initialized to +inf
  foreach (Vertex v : G):
    d[v][v] = 0
  foreach (Edge (u, v) : G):
    d[u][v] = cost(u, v)

  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	~~∞~~ 1	∞	0

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

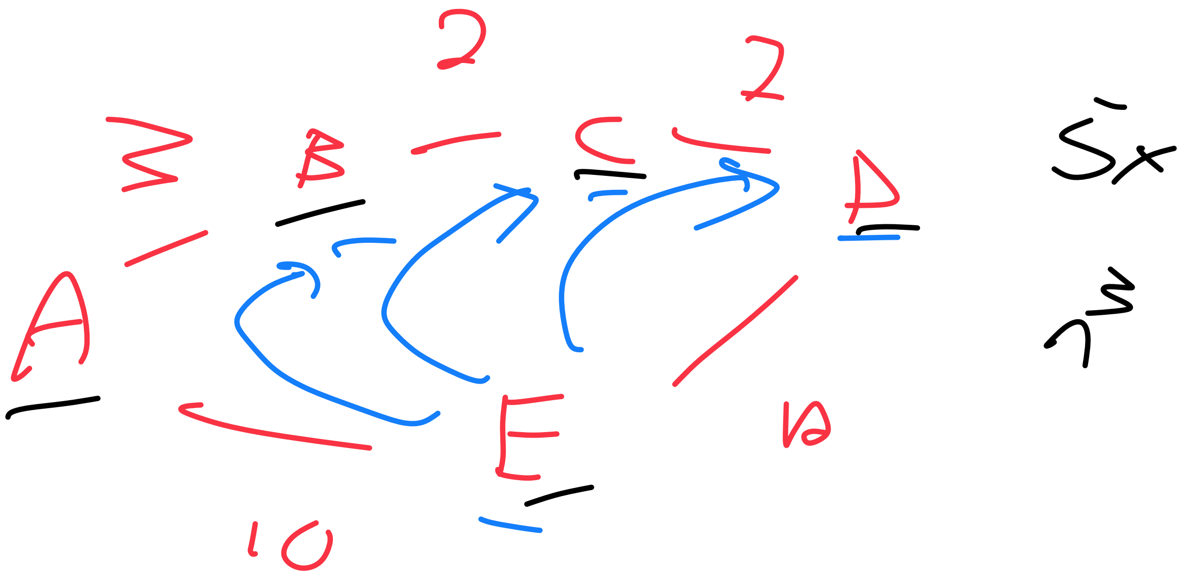


2x space

Trivial???

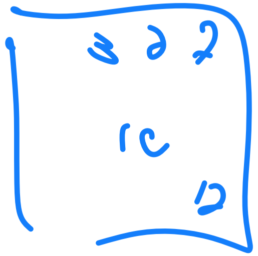
CS 225 In Review → Look at review slide deck

- Lists
- Stacks and Queues
- Trees
- Heaps
- Disjoint Sets
- Probability
- Hash Tables
- Bloom Filters
- MinHash
- Graphs



$w = A$ $\quad$ $w = B$ $\quad$ $w = C$

$u, \forall$ all



The End - Questions? n!

1) Recursion iterator

2) Tree traversals

3) Amortized analysis