Data Structures and Algorithms

All Paths Shortest Path (Plus Review)

CS 225
Brad Solomon

November 6, 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 on PL

Topics covered can be found on website

Registration started October 31

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

Learning Objectives

Introduce and discuss All-Paths Shortest Path

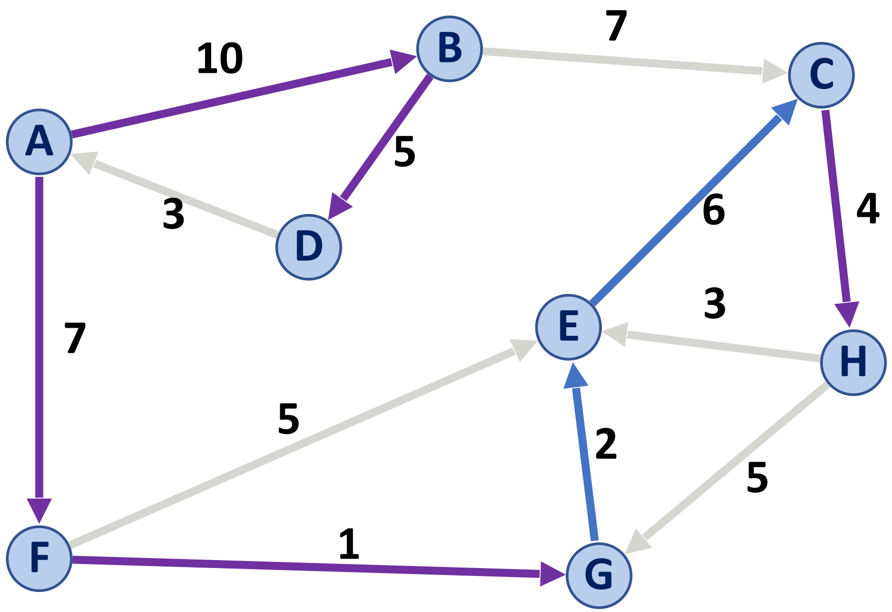
Graphs

Review deterministic data structures in CS

An opportunity for Q&A for exam 4

Dijkstra's Algorithm (SSSP)

Prims equivalent




```
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
```

init

heap

update costs

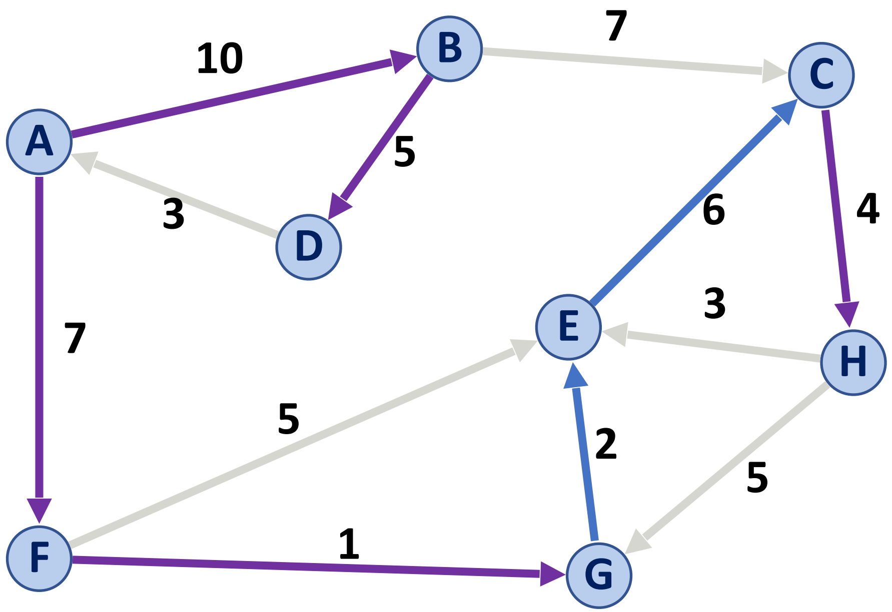
A	B	C	D	E	F	G	H
--	A	E	B	G	A	F	C
0	10	16	15	10	7	8	20

↳ One change from Prim

↳ Prim only single edge weight

↳ Dijkstras sum of edges

Dijkstra's Algorithm (SSSP)



Whats the point of predecessor?

shortest Path from A to H is dist 20

A → F → G → E → C → H

A

pred

Backtracking!

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

← Pred gives path

Dijkstra's Algorithm (SSSP)

Dijkstras Algorithm works only on non-negative weights

Optimal implementation:

Fibonacci Heap

If dense, unsorted list ties

Optimal runtime:

n^2 for dense

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
```

Floyd-Warshall Algorithm

All Paths shortest path

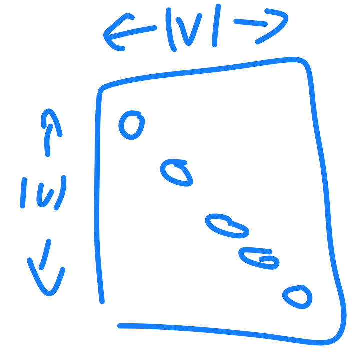
Floyd-Warshall's Algorithm is an alternative to Dijkstra in the presence of negative-weight edges (not negative weight cycles).

"Adj 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]
```

Init

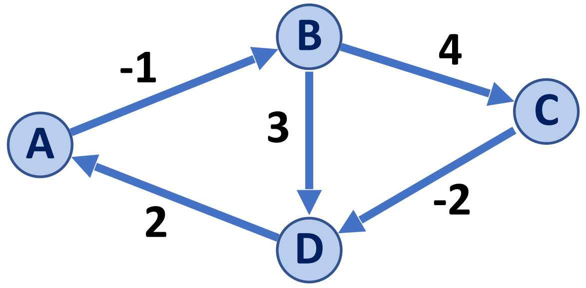


larger than ∃ d(u,w) + d(w,v) is there a better path through intermediate node.

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



Floyd-Warshall Algorithm

u is my start
v is my end
w is proposed midpoint

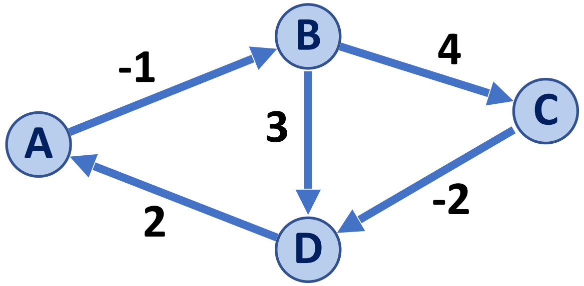
```
foreach (Vertex w : G):
  foreach (Vertex u : G):
    foreach (Vertex v : G):
      if (d[u, v] > d[u, w] + d[w, v])
        d[u, v] = d[u, w] + d[w, v]
```

Let us consider comparisons where w = A:

end

Start

	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

A → A **0** **vs.** A → A → A **0**

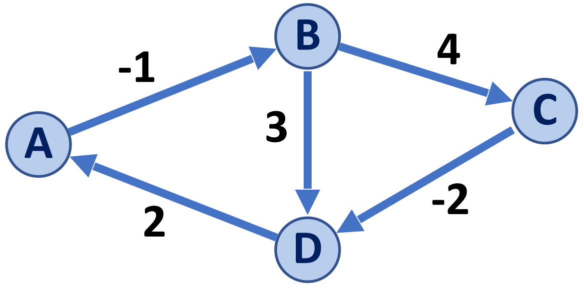
This if case

u=A, v=B

A → B **-1** **vs.** A → A → B **-1**

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 +∞ ✗

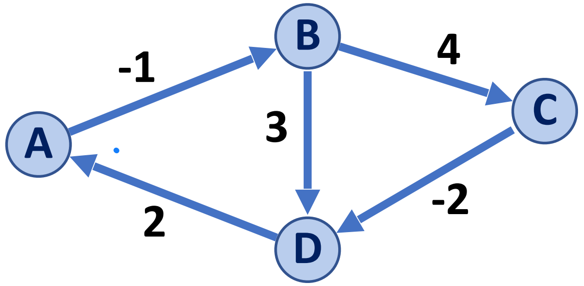
C → D -2 vs. C → A → D +∞ ✗

D → B +∞ vs. D → A → B 2 + -1 = 1

D → C +∞ vs. D → A → C ∞ ✗

2 + ∞

	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]
```

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 +∞

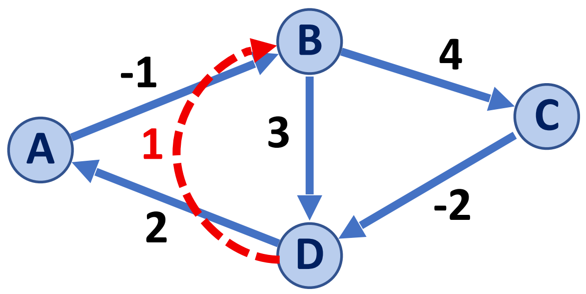
C → B +∞ vs. C → A → B +∞

C → D -2 vs. C → A → D +∞

D → B +∞ vs. D → A → B 1

D → C +∞ vs. D → A → C +∞

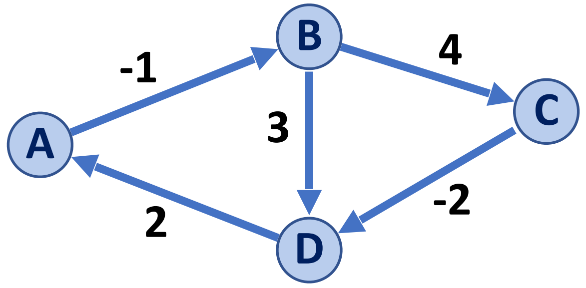
	A	B	C	D
A	0	-1	∞	∞
B	∞	0	4	3
C	∞	∞	0	-2
D	2	1	∞	0



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	~~∞~~ 3	~~∞~~ 2
B	∞	0	4	3
C	∞	∞	0	-2
D	2	1	~~∞~~ 5	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 $\infty + \infty$

C → D -2 vs. C → B → D $\infty + 3$

∞ but thats b/c we havent filled in other parts

ex: B→A exists!

B→D→A 5

we havent seen yet!

D → A 2 vs. D → B → A $1+\infty$

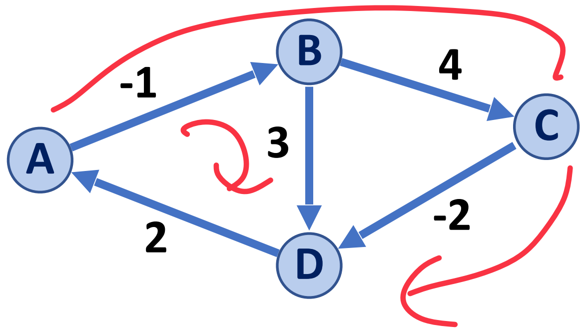
D → C +∞ vs. D → B → C $1+4=5$

from A

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~~ 1
B	∞	0	4	~~3~~ 2
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.** $A \xrightarrow{3} C \xrightarrow{-2} D$ 1

B → A **+∞** **vs.** B → C → A **+∞**

B → D **3** **vs.** $B \xrightarrow{4} C \xrightarrow{-2} D$ 2

D → A **2** **vs.** D → C → A **+∞**

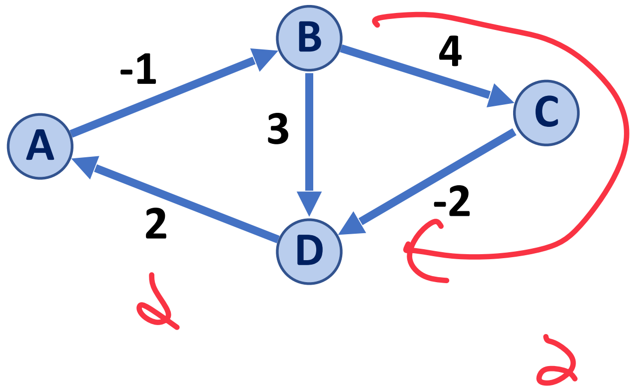
D → B **1** **vs.** D → C → B **+∞**

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	4	0	4	2
C	0	-1	0	-2
D	2	1	5	0



Floyd-Warshall Algorithm

Running time? $O(n^3)$ operation!

↳ Easy to code (multithreadable!)

↳ Handles neg weight (but not cycle)

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

matrix

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]
```

Ignore

Add direct edges store endpoint

extra example:

↳ A → B → C → D (dist 1)

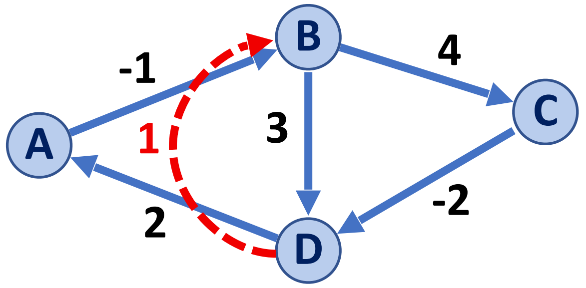
	A	B	C	D
A	0	-1	∞	∞
B	∞	0	4	3
C	∞	∞	0	-2
D	2	1	∞	0

end

	A	B	C	D
A		B		B
B			C	~~D~~ C
C				D
D	$\boxed{A}$	A		

start

storing intermediate steps



We have only scratched the surface on graphs!

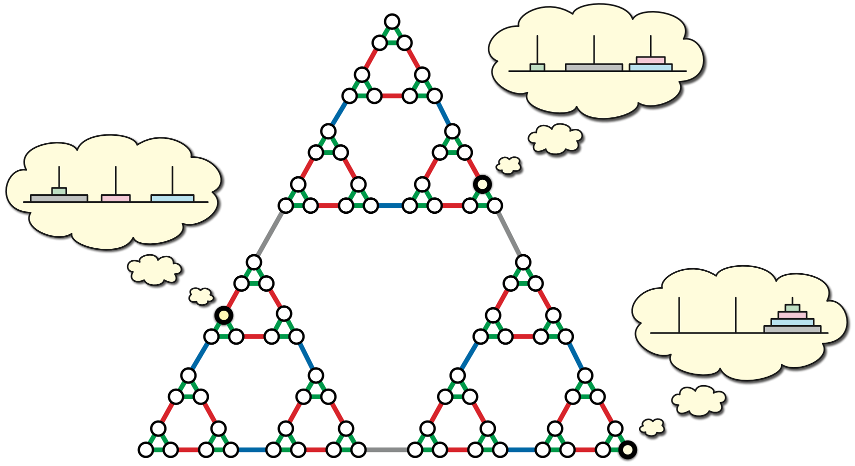


Image from Jeff Erickson Algorithms First Edition

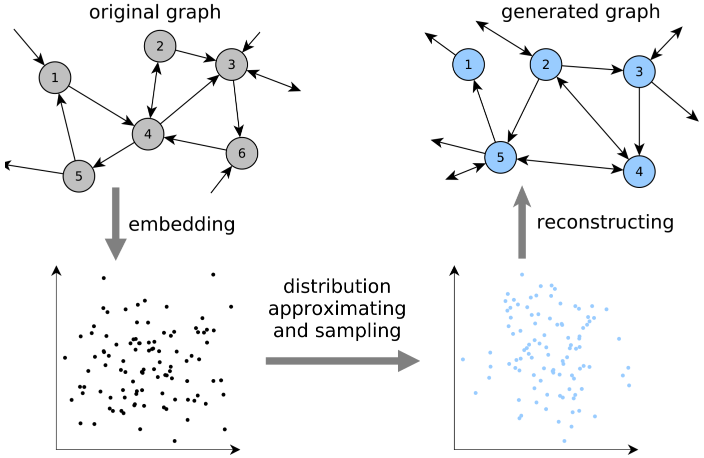
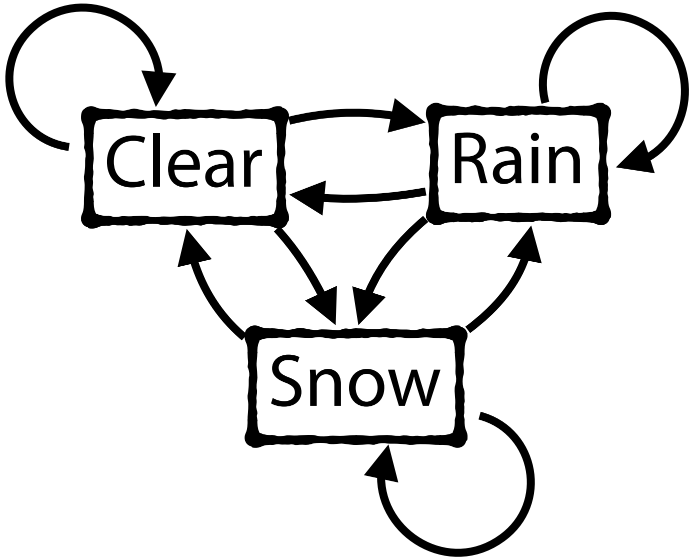


Image from Drobyshevskiy et al. **Random graph modeling: A survey of the concepts.** 2019



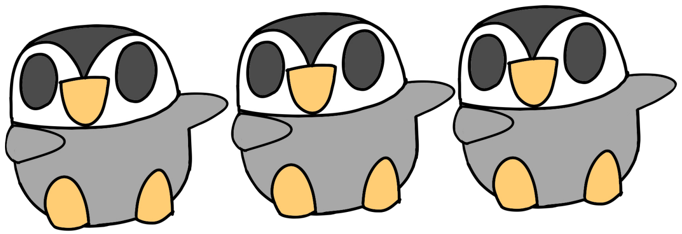
$$M = \begin{pmatrix} .5 & .3 & .2 \\ .5 & .4 & .1 \\ .2 & .1 & .7 \end{pmatrix}$$

Lets review what we’ve seen so far!

Lets review what we’ve seen so far!

Its arrays all the way down.

Lists



The not-so-secret underlying implementation for many things

	Singly Linked List	Array
Look up **arbitrary** location	O(n)	O(1)
Insert after **given** element	O(1)	O(n)
Remove after **given** element	O(1)	O(n)
Insert at **arbitrary** location	O(n)	O(n)
Remove at **arbitrary** location	O(n)	O(n)
Search for an input **value**	O(n)	O(n)

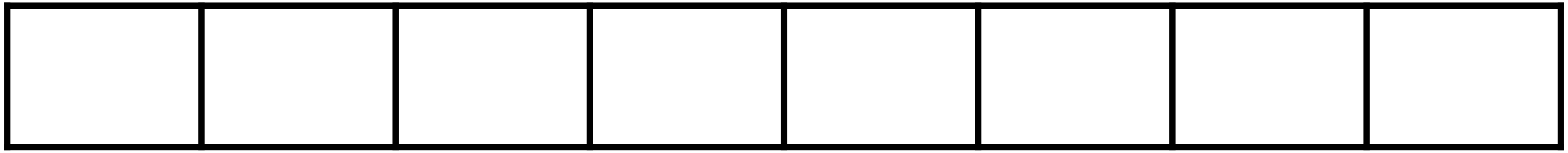
Special Cases: insert Front O(1) Insert Back O(1)*

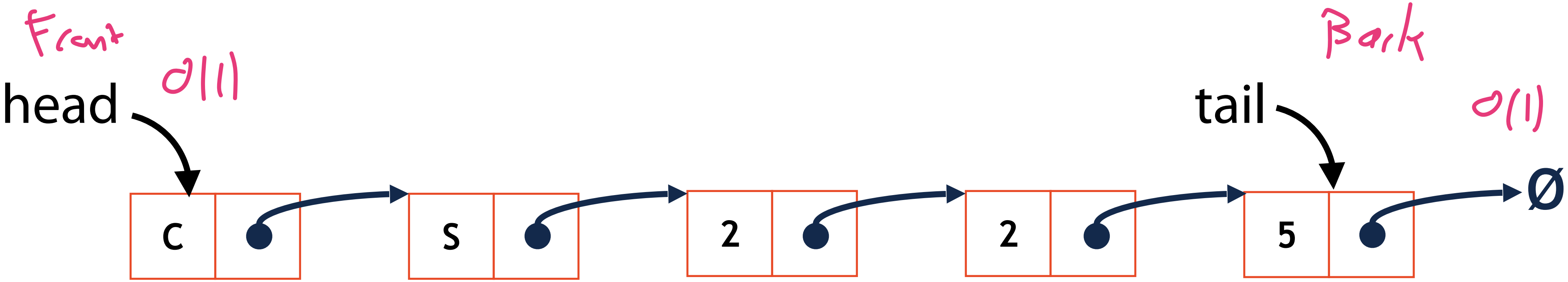
Stack and Queue

Taking advantage of special cases in lists / arrays

insert / remove Back

O(1)*





Heap

Taking advantage of special cases in lists / arrays

insert Back

O(1)*

Array List (Pointer implementation)

T* Start

O(1) lookup

swap

T* Size

T* Capacity

	4	5	6	15	9	7	20	16	25	14	12	11				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

size_t Start

size_t Capacity

Array List (Index implementation)

size_t Size

Disjoint Set Implementation

Taking advantage of array lookup operations

Store an UpTree as an array, canonical items store **height** / **size**

0 1 4

2 7

3 5 6

0	1	2	3	4	5	6	7
-2	0	-2	-2	0	3	3	2
-3		-2	-3				

Find(k): Repeatedly look up values until **negative value**

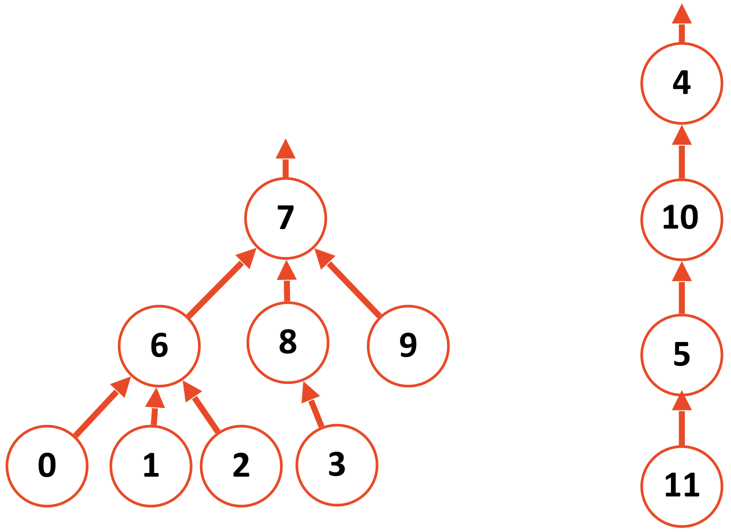
Union(k_1, k_2): Update ***smaller*** canonical item to point to larger

Update value of remaining canonical item

O(1)

Disjoint Sets – Smart Union

Minimizing number of O(1) operations



Union by height

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	**-4**	10	7	**4**	7	7	4	5

Idea: *Keep the height of the tree as small as possible.*

Union by size

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	**7**	10	7	**-12**	7	7	4	5

Idea: *Minimize the number of nodes that increase in height*

Both guarantee the height of the tree is: O(log n). limits find

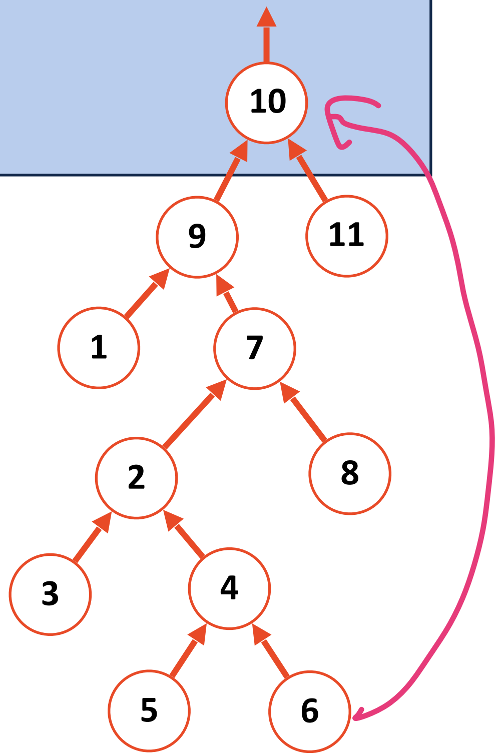
Disjoint Sets Path Compression

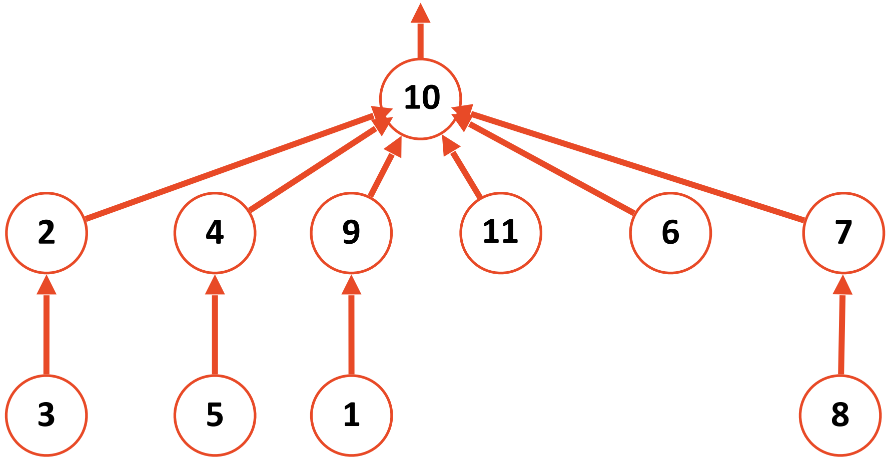
Find(6)

Minimizing number of O(1) operations

```
int DisjointSets::find(int i) {
  if ( s[i] < 0 ) { return i; }
  else {
    int root = find( s[i] );
    s[i] = root;
    return root;
  }
}
```

taking advantage of arrays





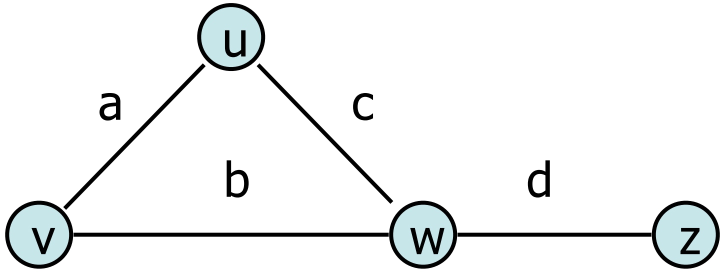
Alternative Not-Actually-A-Proof

Unproven Claim: A disjoint set implemented with smart union and path compression with **m** find calls and **n** items has a worst case running time of **inverse Ackerman.** $\left[O(m \ \alpha(n))\right]$

This grows *very* slowly to the point of being treated a constant in CS.

Just arrays! (kinda)

Graph Implementation: Edge List |V|= n, |E|= m



Literally just arrays

Special case

O(1)*

insertVertex(K key):

insertEdge(Vertex v1, Vertex v2, K key):

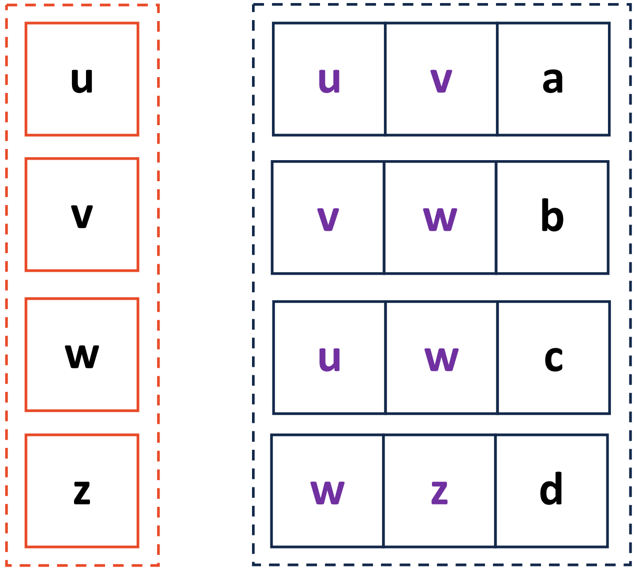
O(m)

removeVertex(Vertex v):

removeEdge(Vertex v1, Vertex v2, K key):

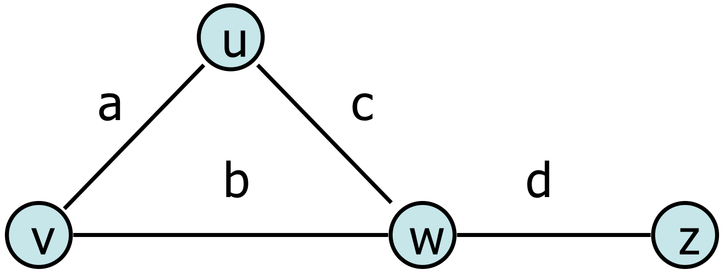
incidentEdges(Vertex v):

areAdjacent(Vertex v1, Vertex v2):



Graph Implementation: Adjacency Matrix

$|V| = n, |E| = m$



Literally just a matrix of arrays

O(1) ← lookups in arrays!

insertEdge(Vertex v1, Vertex v2, K key):

removeEdge(Vertex v1, Vertex v2, K key):

areAdjacent(Vertex v1, Vertex v2):

O(n) " array search ~~lookup~~ on n items

incidentEdges(Vertex v):

← array search

O(n)—O(n^2)

insertVertex(K key):

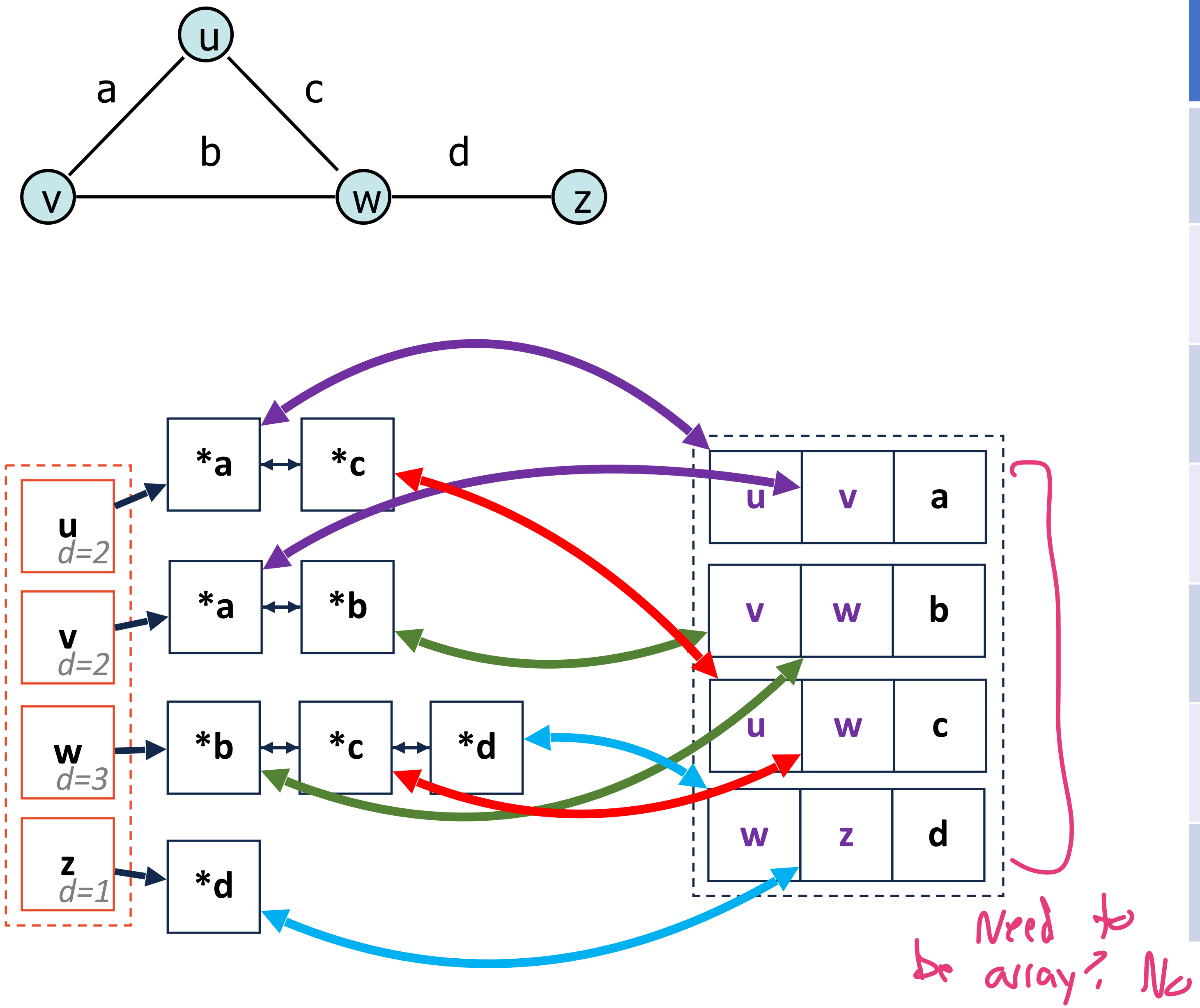
removeVertex(Vertex v):

u
v
w
z

	u	v	w	z
u	-	a	c	0
v		-	b	0
w			-	d
z				-

Adjacency List

Technically linked lists I guess



Expressed as O(f)	Adjacency List
Space	n+m
insertVertex(v)	1*
removeVertex(v)	deg(v)
insertEdge(u, v)	1*
removeEdge(u, v)	min(deg(u), deg(v))
incidentEdges(v)	deg(v)
areAdjacent(u, v)	min(deg(u), deg(v))

… And thats most of exam 4

Randomized Algorithms

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

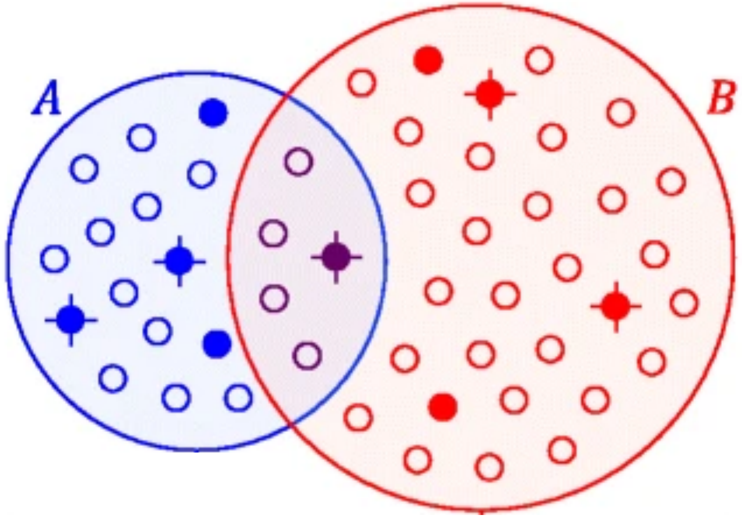
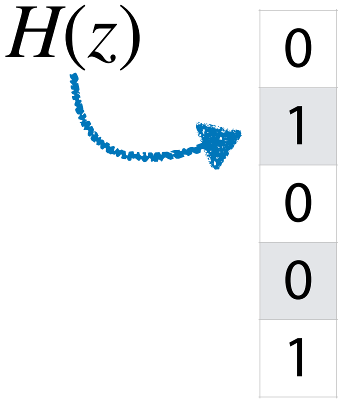
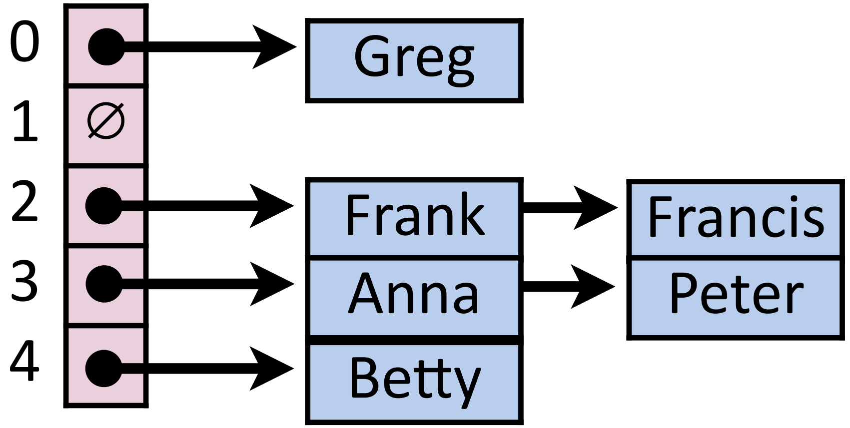


Figure from Ondov et al 2016





$H(x)$	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: