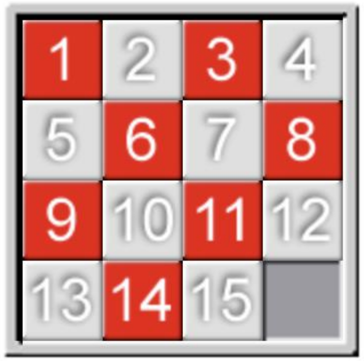
CS2040S
Data Structures and Algorithms

Hashing! (Part 4); Graphs! (Part 1)

Puzzle of the Week:

Can you interchange blocks 14 and 15, leaving everything else the same?



Loyd writes of how he "drove the entire world crazy," and that "A prize of $1,000, offered for the first correct solution to the problem, has never been claimed, although there are thousands of persons who say they performed the required feat." He continues,

> People became infatuated with the puzzle and ludicrous tales are told of shopkeepers who neglected to open their stores; of a distinguished clergyman who stood under a street lamp all through a wintry night trying to recall the way he had performed the feat.... Pilots are said to have wrecked their ships, and engineers rush their trains past stations. A famous Baltimore editor tells how he went for his noon lunch and was discovered by his frantic staff long past midnight pushing little pieces of pie around on a plate! [9]

Plan: this week and next

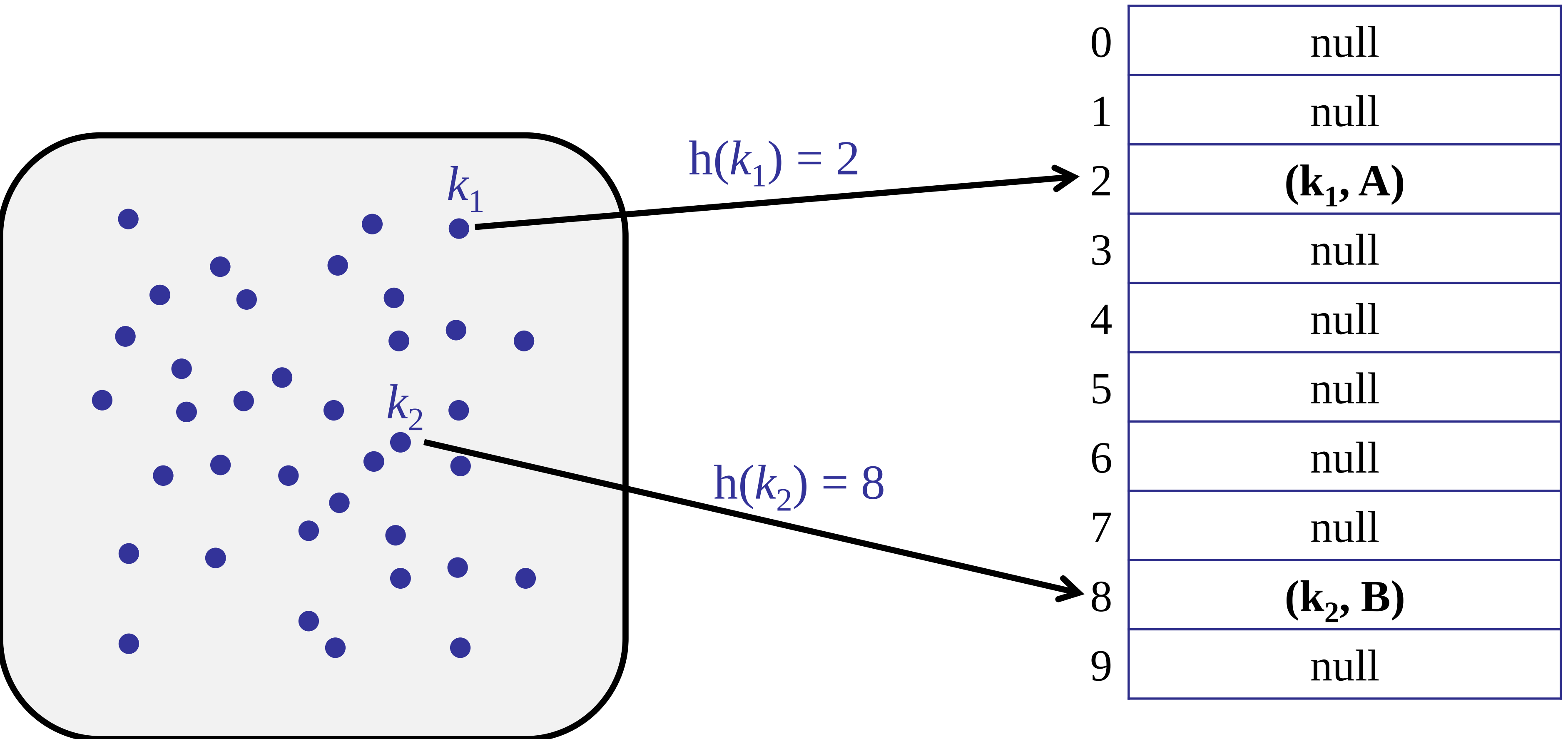
Fourth day of hashing

- Open Addressing

Review

Hash Tables

- Store each item from the symbol table in a table.
- Use hash function to map each key to a bucket.



0	null
1	null
2	**(k_1, A)**
3	null
4	null
5	null
6	null
7	null
8	**(k_2, B)**
9	null

Resolving Collisions

- Basic problem:
  - What to do when two items hash to the same bucket?
- Solution 1: Chaining
  - Insert item into a linked list.
- Solution 2: Open Addressing
  - Find another free bucket.

Open Addressing

Advantages:

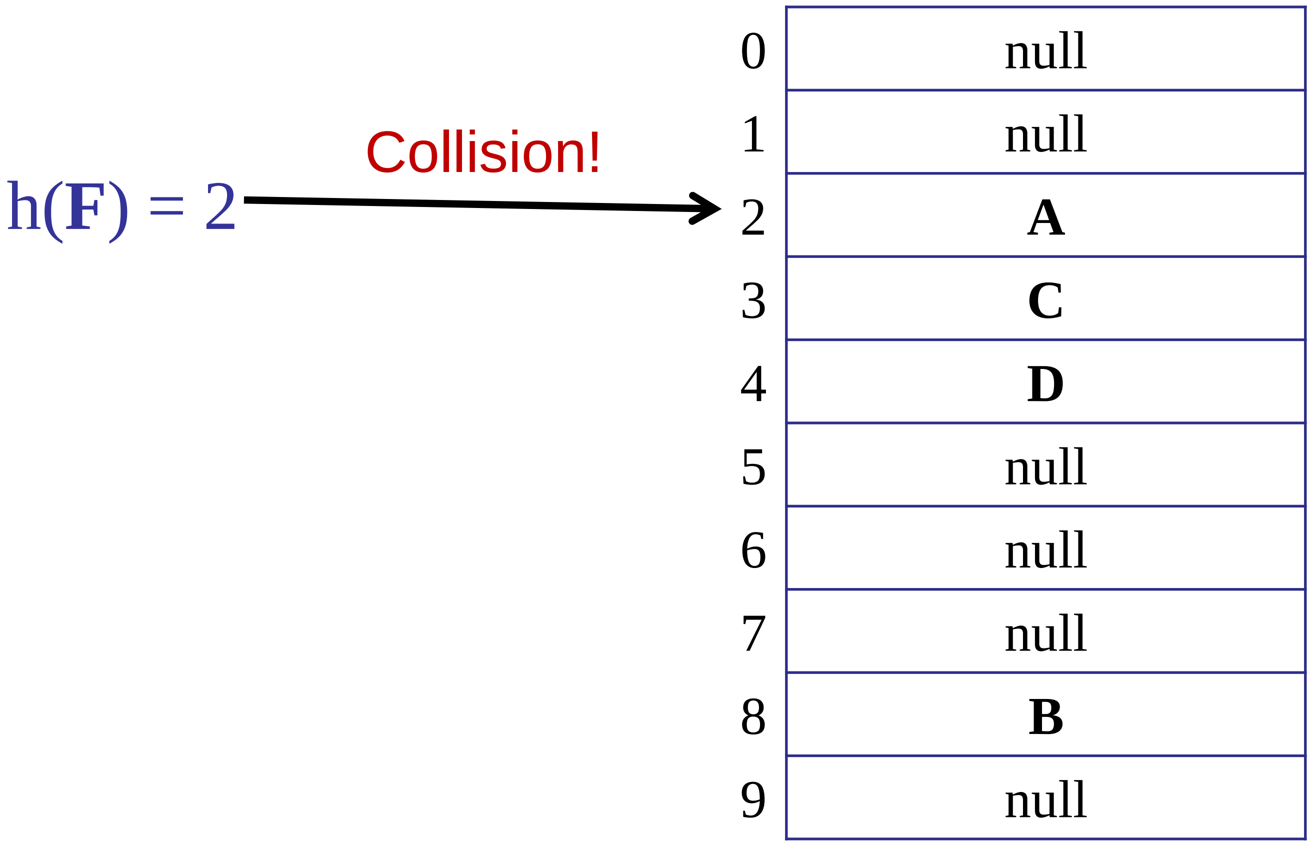
- No linked lists!
- All data directly stored in the table.
- One item per slot.

0	null
1	null
2	**A**
3	null
4	null
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

On collision:

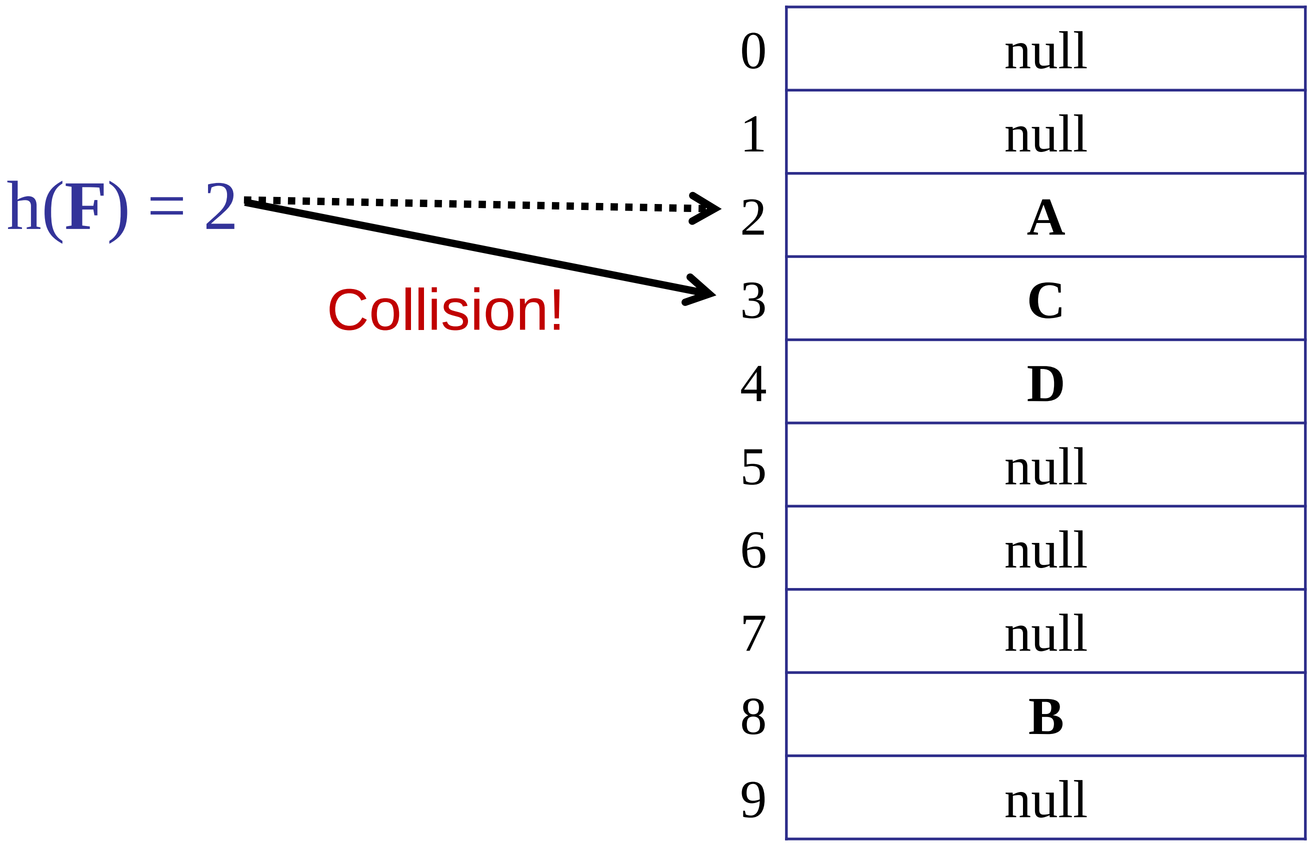
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

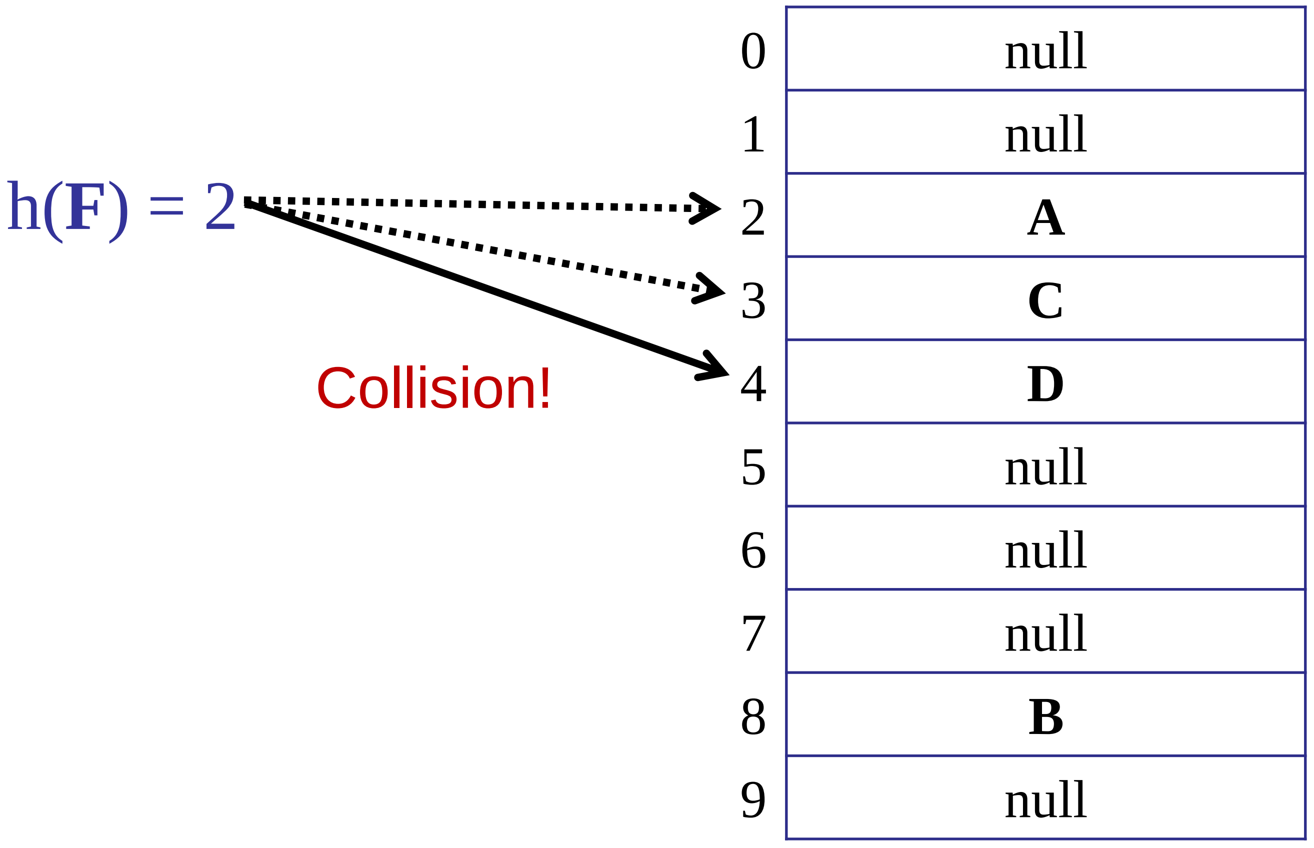
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

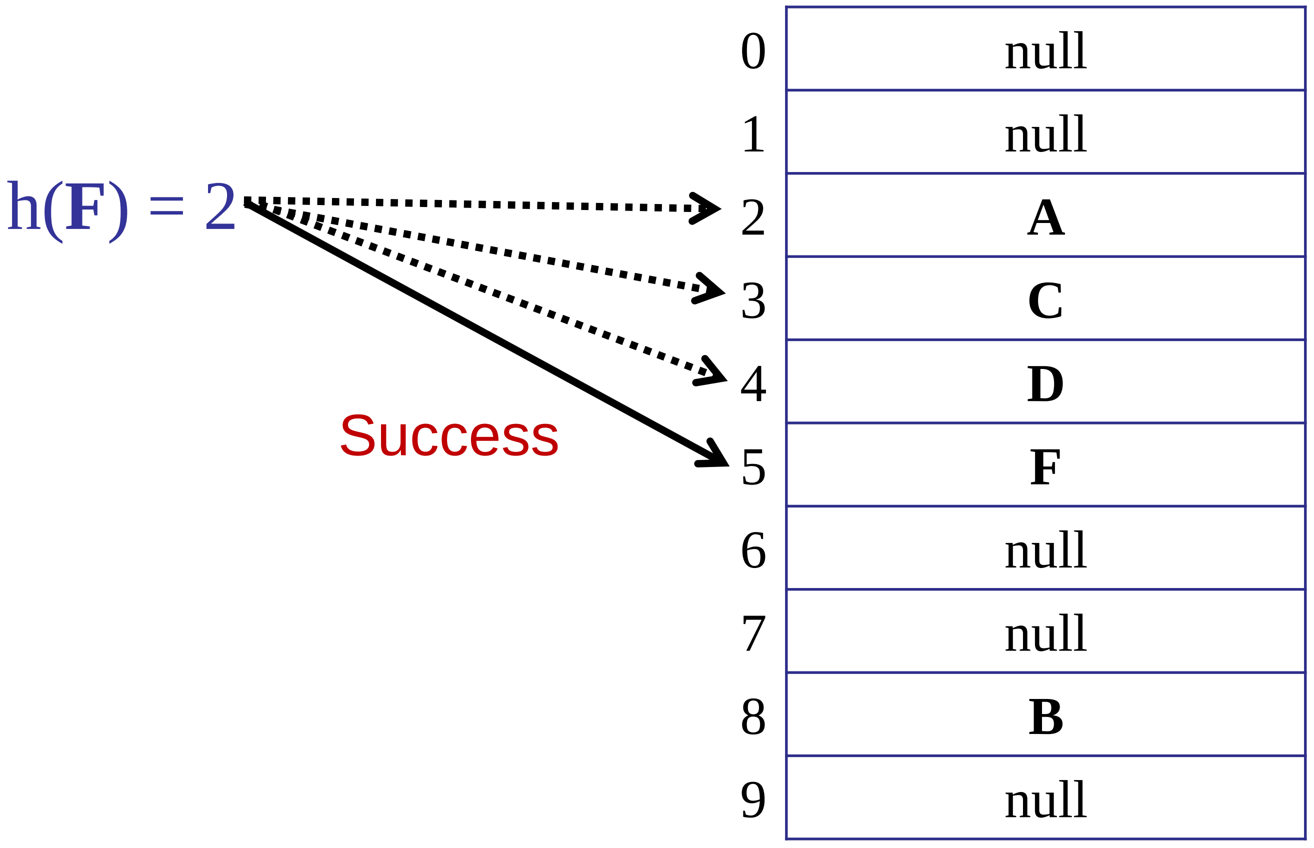
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

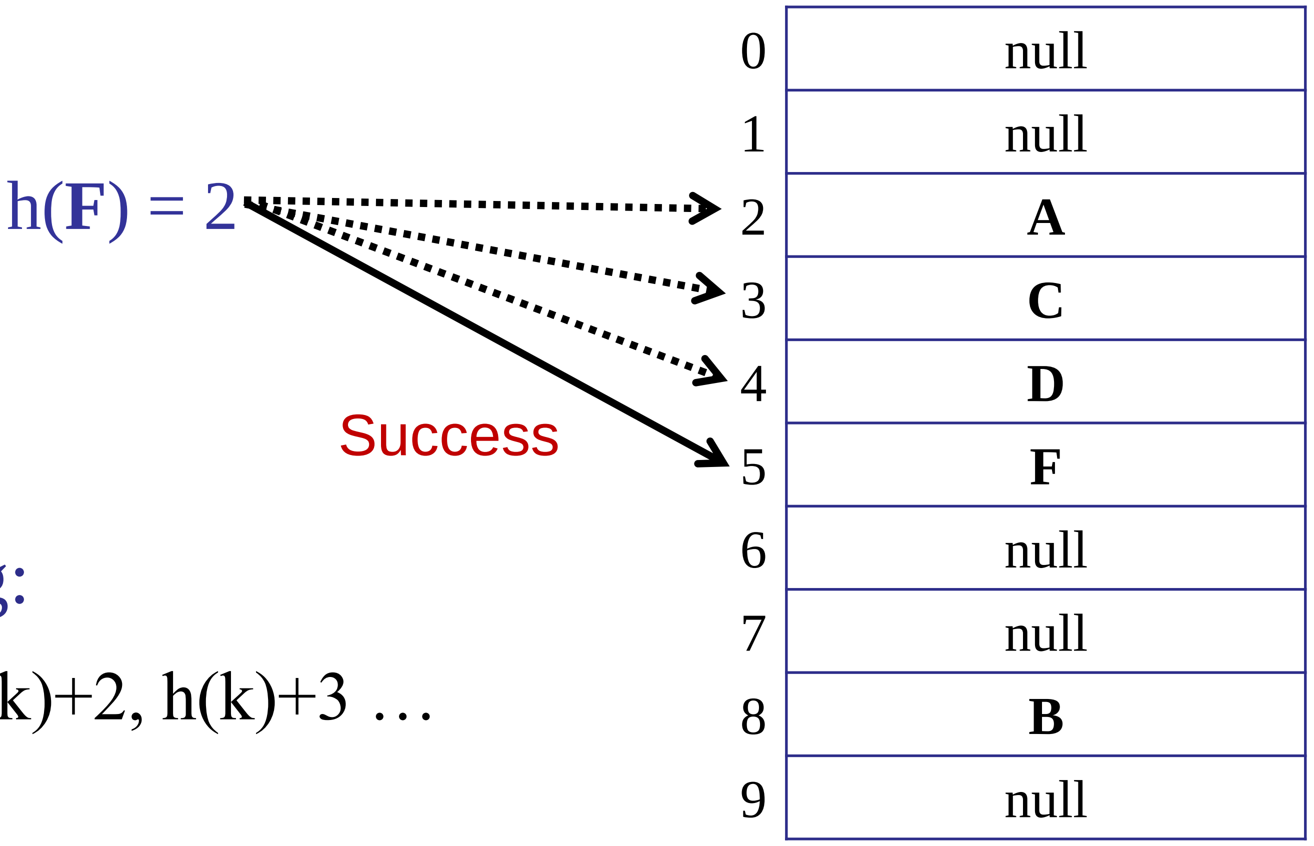
Probe a sequence of buckets until you find an empty one.



Open Addressing

On collision:

Probe a sequence of buckets until you find an empty one.



Linear Probing:

- h(k)+1, h(k)+2, h(k)+3 …

Open Addressing

Hash Function re-defined:

$h(key, i) : U \rightarrow \{1..m\}$

Two parameters:

- key : the thing to map
- i : number of collisions

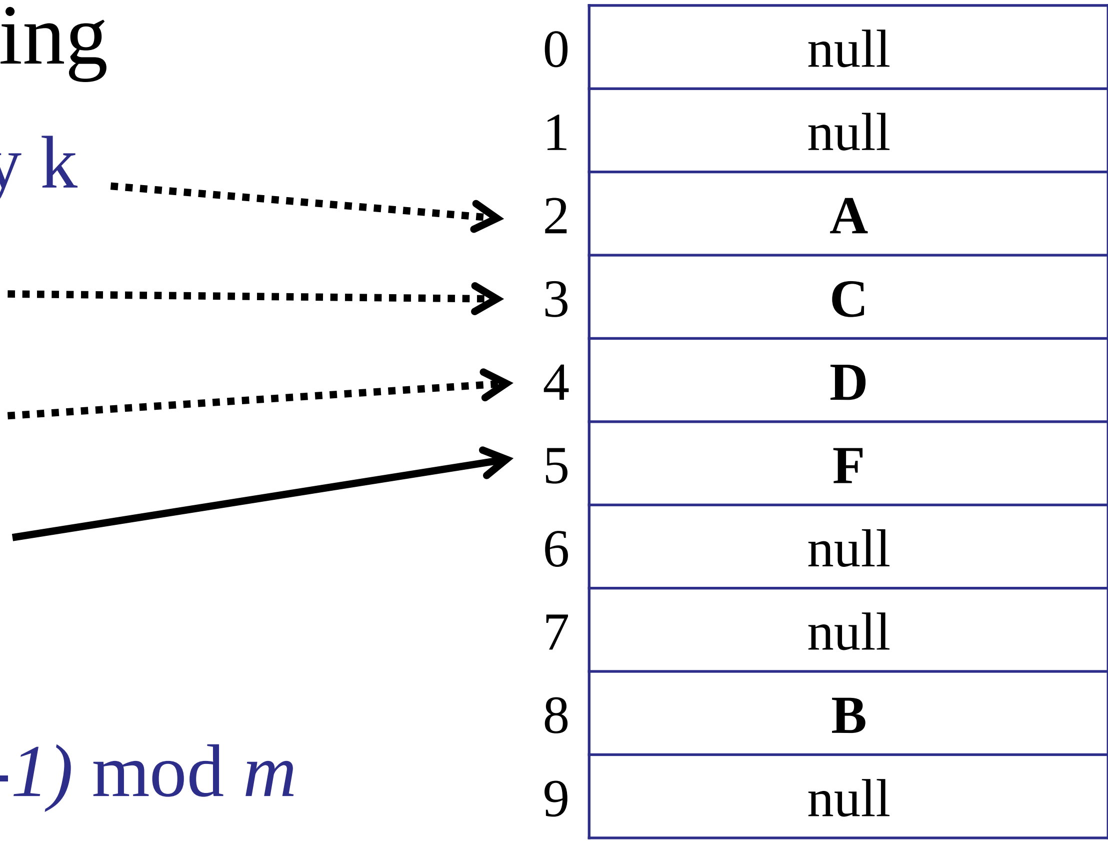
Open Addressing

Hash Function re-defined:

$h(\text{key}, i) : U \rightarrow \{1..m\}$

Example: Linear Probing

- $h(k, 1)$ = hash of key k
- $h(k, 2) = h(k, 1) + 1$
- $h(k, 3) = h(k, 1) + 2$
- $h(k, 4) = h(k, 1) + 3$
- …
- $h(k, i) = h(k, 1) + (i\text{-}1) \bmod m$



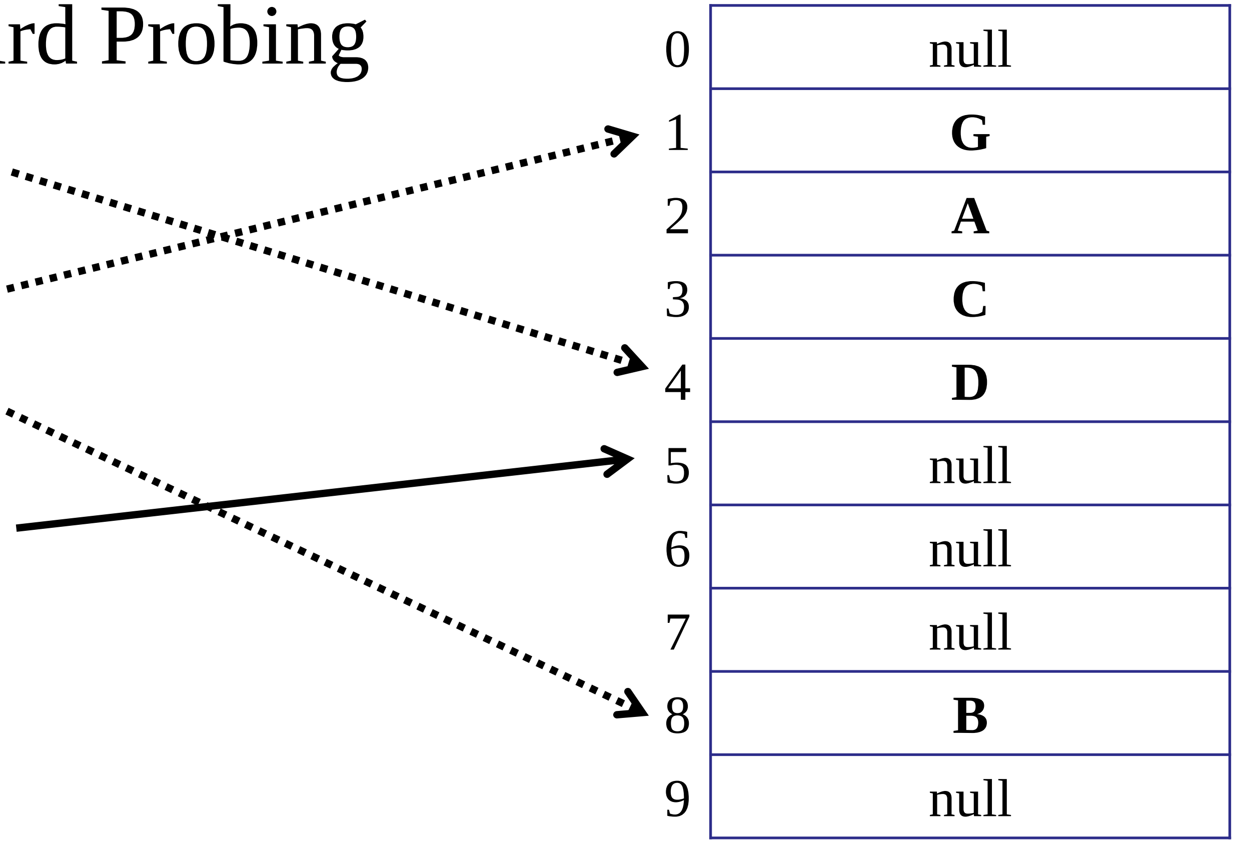
Open Addressing

Hash Function re-defined:

$$h(key, i) : U \rightarrow \{1..m\}$$

Example: Weird Probing

- $h(k, 1) = 4$
- $h(k, 2) = 1$
- $h(k, 3) = 8$
- $h(k, 4) = 5$



0	null
1	**G**
2	**A**
3	**C**
4	**D**
5	null
6	null
7	null
8	**B**
9	null

Open Addressing

```
hash-insert(key, data)
int i = 1;
while (i <= m) {                              // Try every bucket
    int bucket = h(key, i);
    if (T[bucket] == null){                   // Found an empty bucket
        T[bucket] = {key, data};              // Insert key/data
        return success;                       // Return
    }
    i++;
}
throw new TableFullException();               // Table full!
```

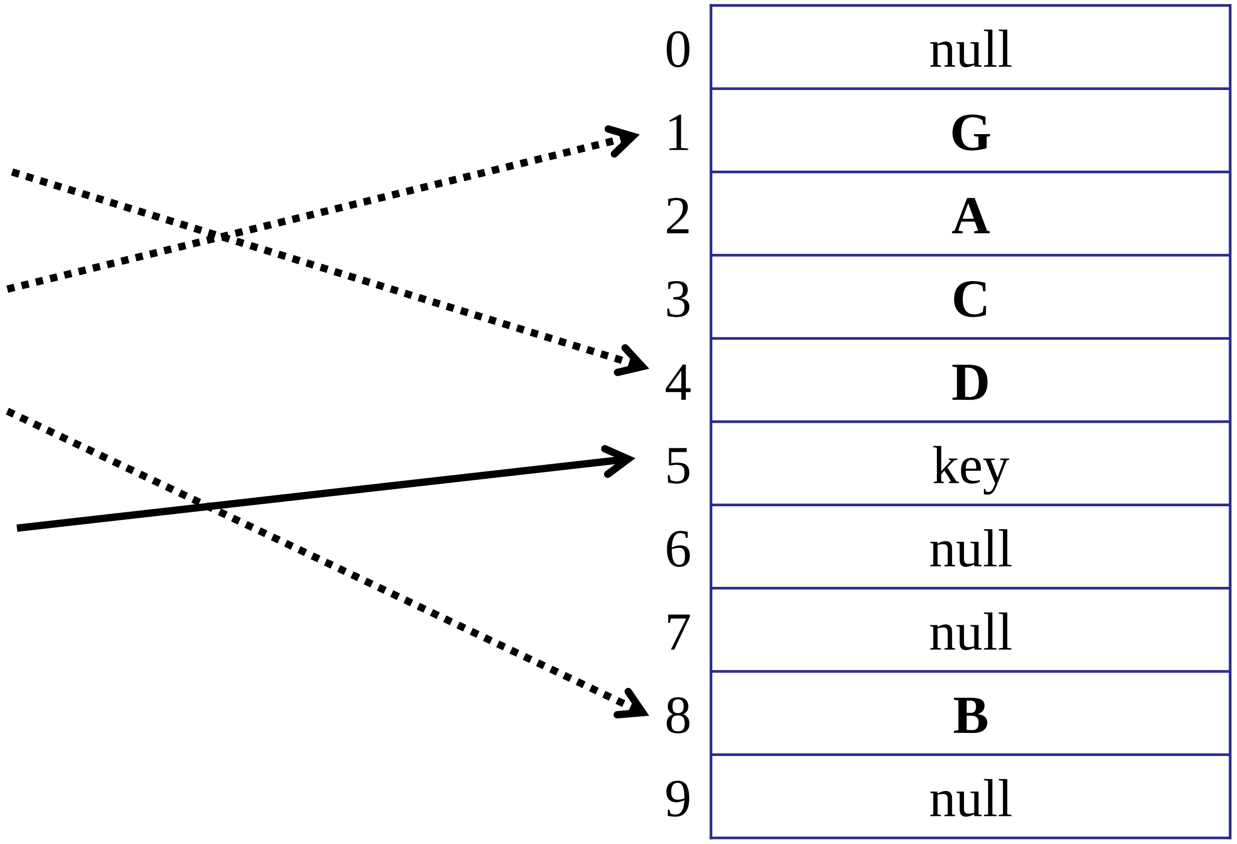
Open Addressing

Hash Function re-defined:

$h(key, i) : U \rightarrow \{1..m\}$

search(key)

- $h(key, 1) = 4$
- $h(key, 2) = 1$
- $h(key, 3) = 8$
- $h(key, 4) = 5$



0	null
1	**G**
2	**A**
3	**C**
4	**D**
5	key
6	null
7	null
8	**B**
9	null

Open Addressing

```
hash-search(key)
1. int i = 1;
2. while (i <= m) {
3.       int bucket = h(key, i);
4.       if (T[bucket] == null)  // Empty bucket!
5.             return key-not-found;
6.       if (T[bucket].key == key)  // Full bucket.
7.                   return T[bucket].data;
8.       i++;
9. }
10. return key-not-found;  // Exhausted entire table.
```

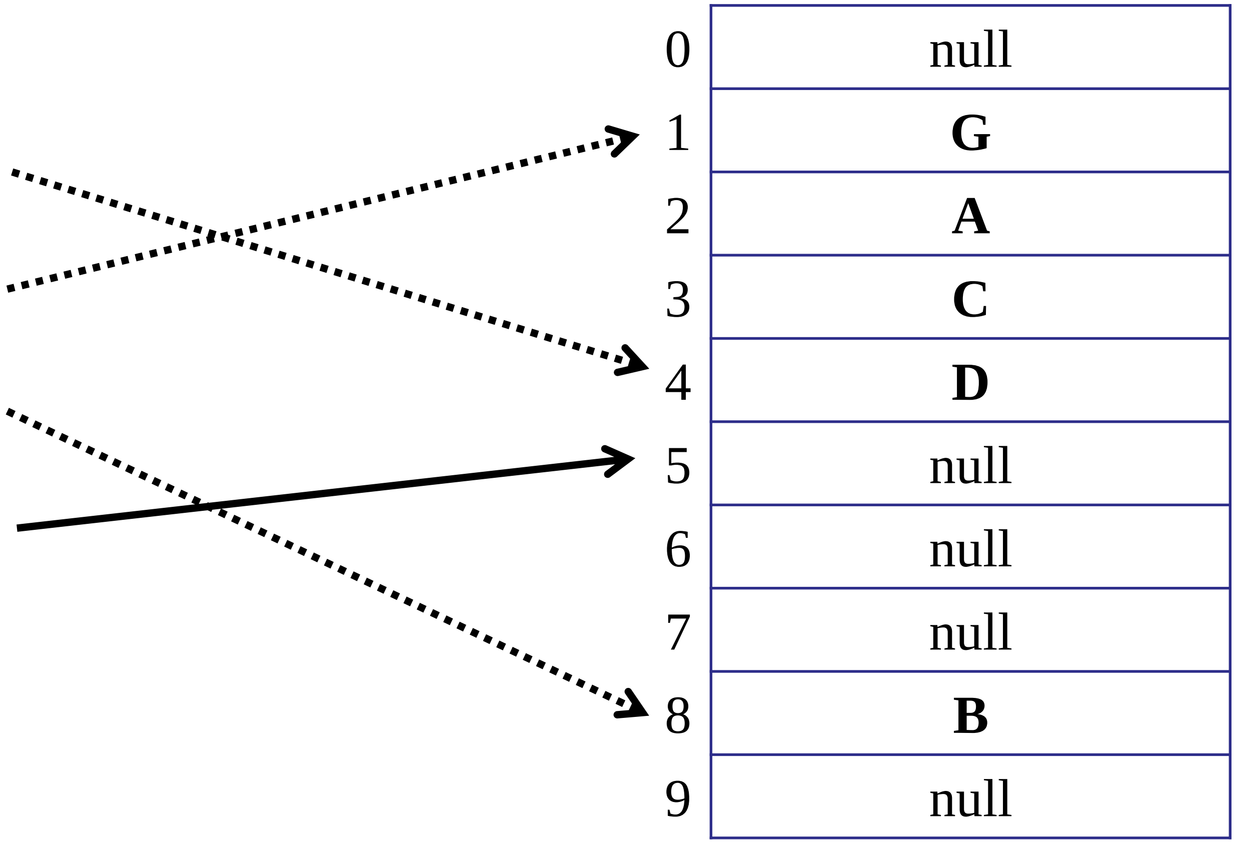
Open Addressing

Hash Function re-defined:

$h(key, i) : U \rightarrow \{1..m\}$

search(key)

- $h(key, 1) = 4$
- $h(key, 2) = 1$
- $h(key, 3) = 8$
- $h(key, 4) = 5$



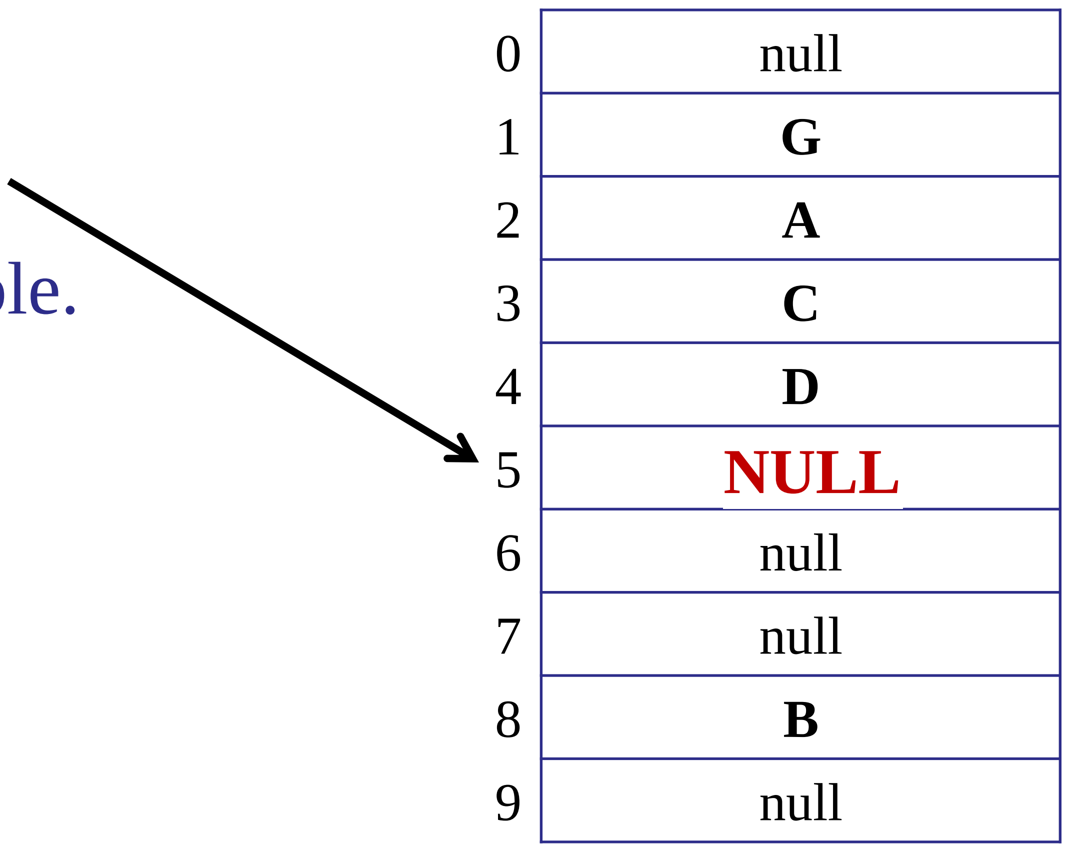
Open Addressing

Hash Function re-defined:

$h(key, i) : U \rightarrow \{1..m\}$

delete(key)

- Find key to delete
- Remove it from table.
- Set bucket to null.



What is wrong with delete?

✓ 1. Search may fail to find an element.
2. The table will have gaps in it.
3. Space is used inefficiently.
4. If the key is inserted again, it may end up in a different bucket.

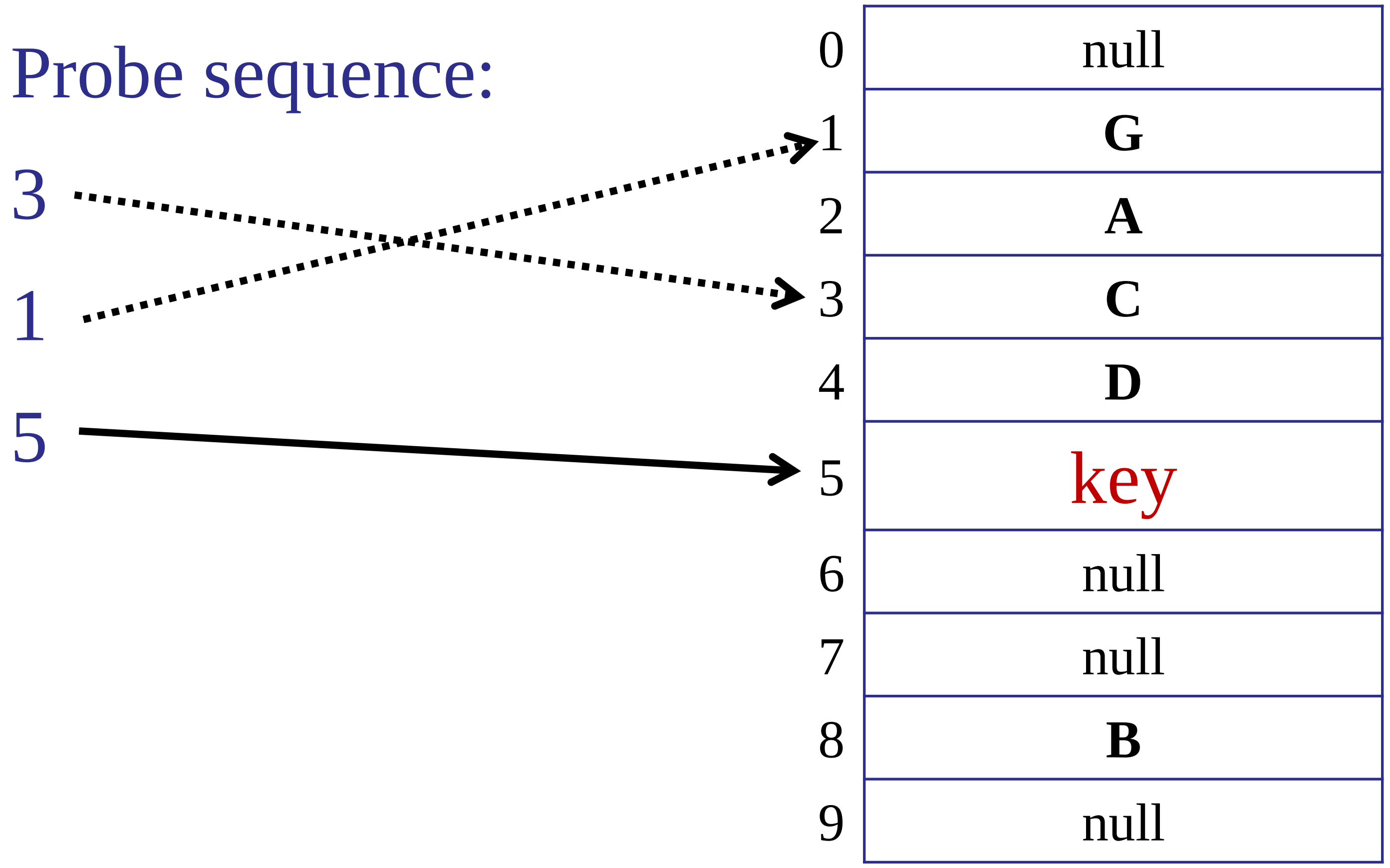
ARCHIPELAGO

is open

Open Addressing

insert(key)

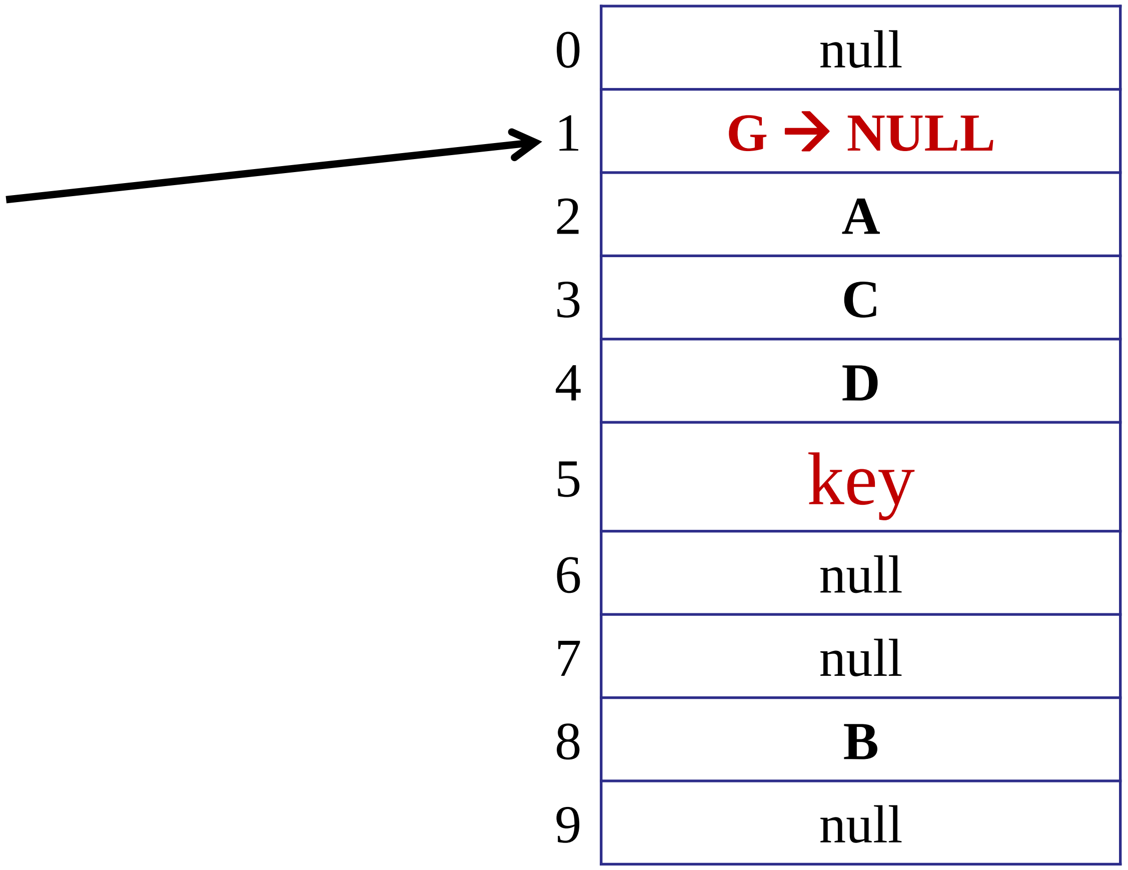
Probe sequence:



Open Addressing

insert(key)

delete(G)



Open Addressing

insert(key)

delete(G)

search(key)

0	null
1	**NULL**
2	**A**
3	**C**
4	**D**
5	key
6	null
7	null
8	**B**
9	null

Open Addressing

insert(key)

delete(G)

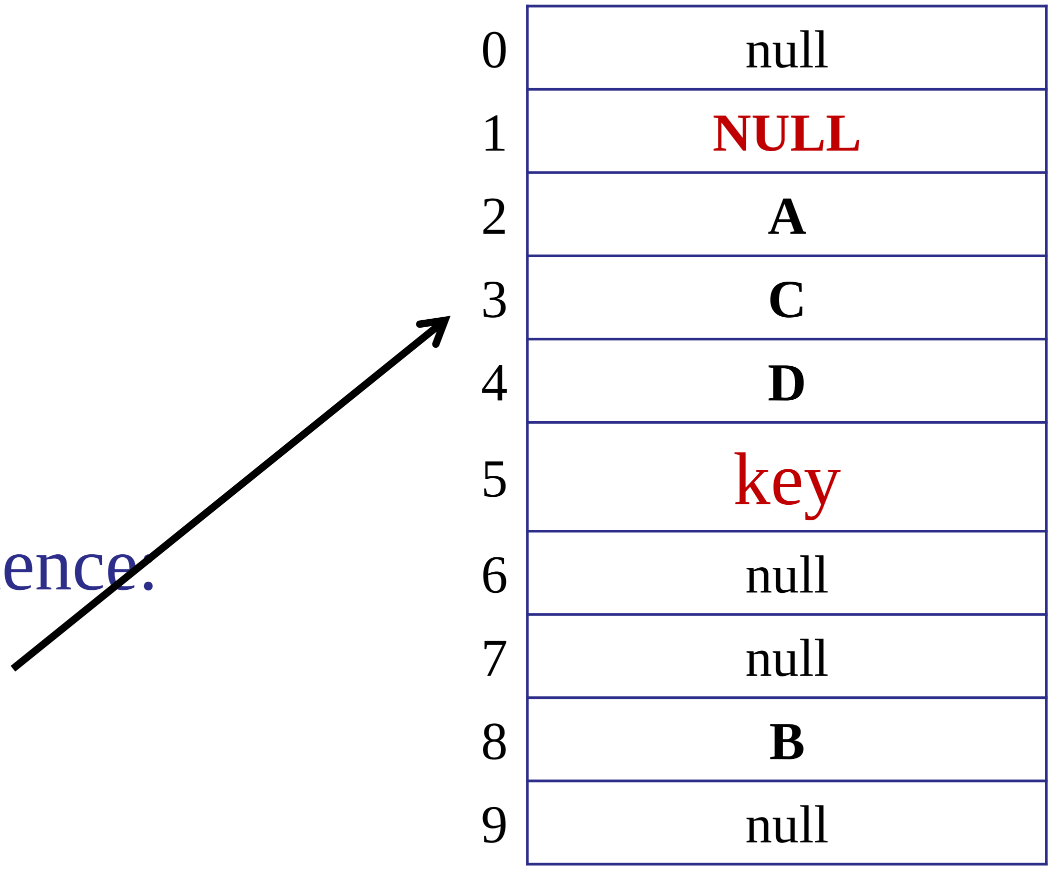
search(key)

Probe sequence:

3

1

5



0	null
1	**NULL**
2	**A**
3	**C**
4	**D**
5	key
6	null
7	null
8	**B**
9	null

Open Addressing

insert(key)

delete(G)

search(key)

Probe sequence:

3

1

Not found!

0	null
1	**NULL**
2	**A**
3	**C**
4	**D**
5	key
6	null
7	null
8	**B**
9	null

Open Addressing

Hash Function re-defined:

$h(key, i) : U \rightarrow \{1..m\}$

delete(key)

- Find key to delete
- Remove it from table.
- Set bucket to DELETED.

(Tombstone value.)

0	null
1	**G**
2	**A**
3	**C**
4	**D**
5	**DELETED**
6	null
7	null
8	**B**
9	null

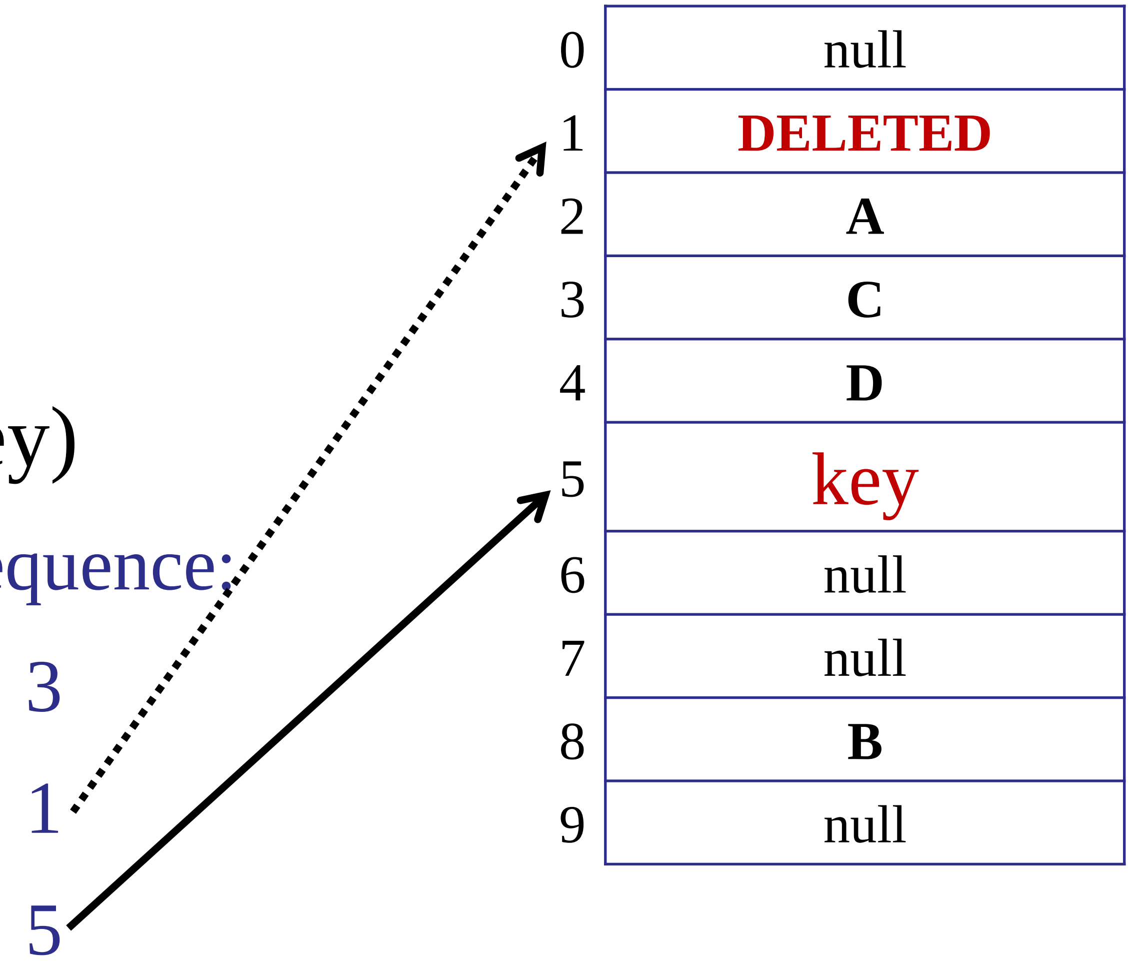
Open Addressing

insert(key)

delete(G)

search(key)

Probe sequence:



What happens when an insert finds a DELETED cell?

1. ✓ Overwrite the deleted cell.
2. Continue probing.
3. Fail.

ARCHIPELAGO is open

Hash Functions

Two properties of a good hash function:

1. h(*key*, *i*) enumerates all possible buckets.
   - For every bucket *j*, there is some *i* such that:

     $$h(key, i) = j$$

   - The hash function is permutation of $\{1..m\}$.
   - For linear probing: true!

What goes wrong if the sequence is not a permutation?

1. Search incorrectly returns key-not-found.
2. Delete fails.
3. Insert puts a key in the wrong place
4. [✓ Returns table-full even when there is still space left.]

ARCHIPELAGO

is open

Hash Functions

Two properties of a good hash function:

2. Simple Uniform Hashing Assumption

   Every key is equally likely to be mapped to every bucket, independently of every other key.

For h(*key*, 1)?

For every h(*key*, *i*)?

Hash Functions

Two properties of a good hash function:

2. Uniform Hashing Assumption

   Every key is equally likely to be mapped to every ***permutation***, independent of every other key.

n! permutations for probe sequence: e.g.,

- 1 2 3 4
- 1 2 4 3
- 1 4 2 3
- 1 4 3 2
- …

Hash Functions

Two properties of a good hash function:

2. Uniform Hashing Assumption

   Every key is equally likely to be mapped to every ***permutation***, independent of every other key.

n! permutations for probe sequence: e.g.,

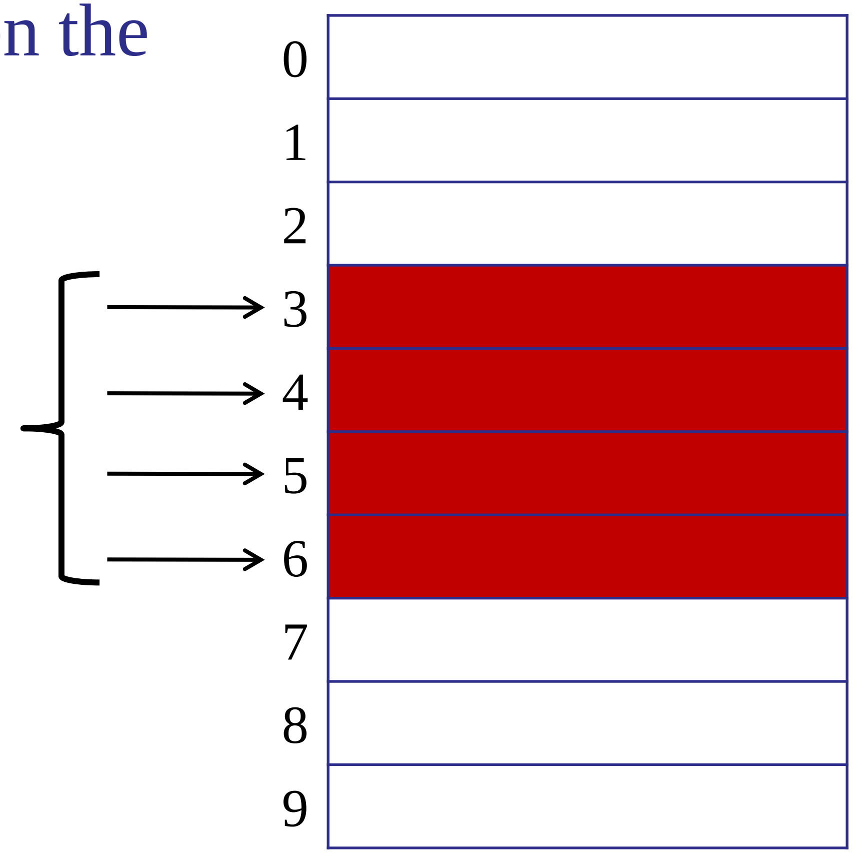
• 1 2 3 4	Pr(1/m)
• 1 2 4 3	Pr(0)
• 1 4 2 3	Pr(0)
• 1 4 3 2	Pr(0)
• …	

NOT Linear Probing

Linear Probing

Problem with linear probing: ***clusters***

- If there is a cluster, then there is a higher probability that the next h(k) will hit the cluster.
- If h(k,1) hits the cluster, then the cluster grows bigger.



- "Rich get richer."

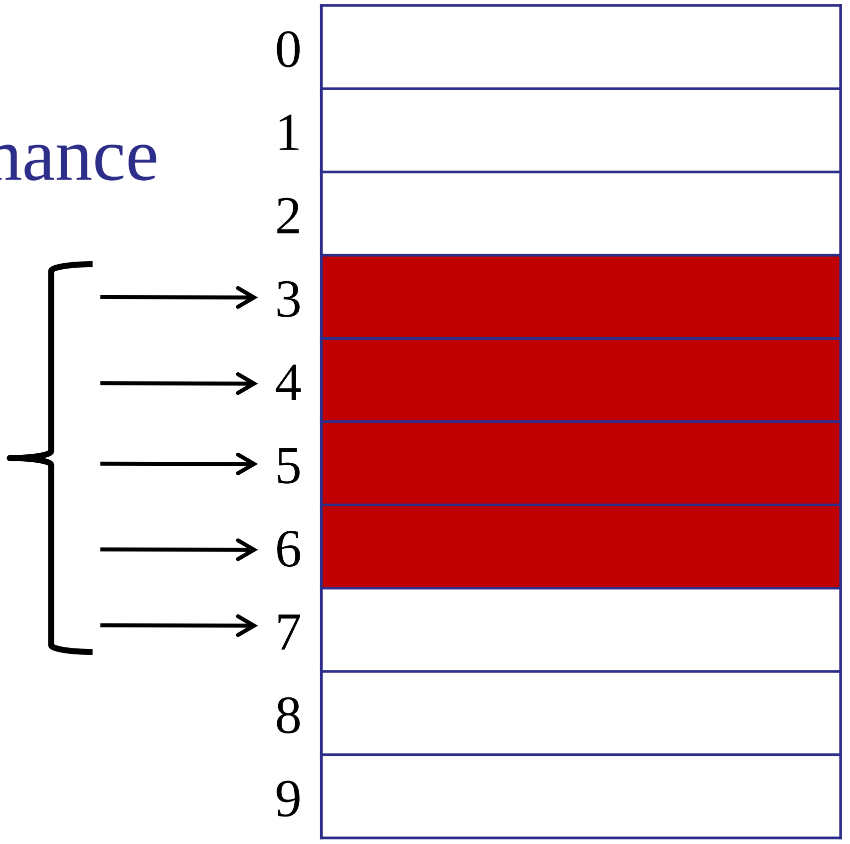
Linear Probing

Problem with linear probing: ***clusters***

- If the table is 1/4 full, then there will be clusters of size:

  $\theta(\log n)$

- Ruins constant-time performance



Linear probing

In practice, linear probing is faster!

- Why? Caching!
- It is *cheap* to access nearby array cells.
  - Example: access T[17]
  - Cache loads: T[10..50]
  - Almost 0 cost to access T[18], T[19], T[20], …
- If the table is 1/4 full, then there will be clusters of size: $\theta(\log n)$
  - Cache may hold entire cluster!
  - No worse than wacky probe sequence.

Double Hashing

- Start with two ordinary hash functions:

$$f(k),\ g(k)$$

- Define new hash function:

$$h(k, i) \;=\; f(k) + i \cdot g(k) \mod m$$

- Note:
  - Since f(k) is good, f(k, 1) is "almost" random.
  - Since g(k) is good, the probe sequence is "almost" random.

Double Hashing

Hash function

$$h(k, i) = f(k) + i \cdot g(k) \mod m$$

Claim: if g(k) is relatively prime to m, then h(k, i) hits all buckets.

- Assume not: then for some distinct $i, j < m$:

$$f(k) + i \cdot g(k) = f(k) + j \cdot g(k) \mod m$$

$\Rightarrow$ $i \cdot g(k) = j \cdot g(k) \mod m$

$\Rightarrow$ $(i - j) \cdot g(k) = 0 \mod m$

$\Rightarrow$ g(k) not relatively prime to m, since (i-j $\neq$ 0 mod m)

Double Hashing

Hash function

$$h(k, i) = f(k) + i \cdot g(k) \mod m$$

Claim: if g(k) is relatively prime to m, then $h(k, i)$ hits all buckets.

Example: if $(m = 2^r)$, then choose $g(k)$ odd.

Performance of Open Addressing

If (m==n), what is the expected insert time, under uniform hashing assumption?

1. O(1)
2. O(log n)
3. O(n)
4. O(n^2)
5. [✓ None of the above.]

ARCHIPELAGO

is open

Performance of Open Addressing

- Chaining:
  - When (m==n), we can still add new items to the hash table.
  - We can still search efficiently.

- Open addressing:
  - When (m==n), the table is full.
  - We cannot insert any more items.
  - We cannot search efficiently.

Performance of Open Addressing

Define:

- Load $\alpha = n / m$

Average # items / bucket

- Assume $\alpha < 1$.

Performance of Open Addressing

Define:

- Load α = *n* / *m* ← Average # items / bucket
- Assume α < 1.

Claim:

Type equation here.

For *n* items, in a table of size *m*, assuming *uniform hashing*, the expected cost of an operation is:

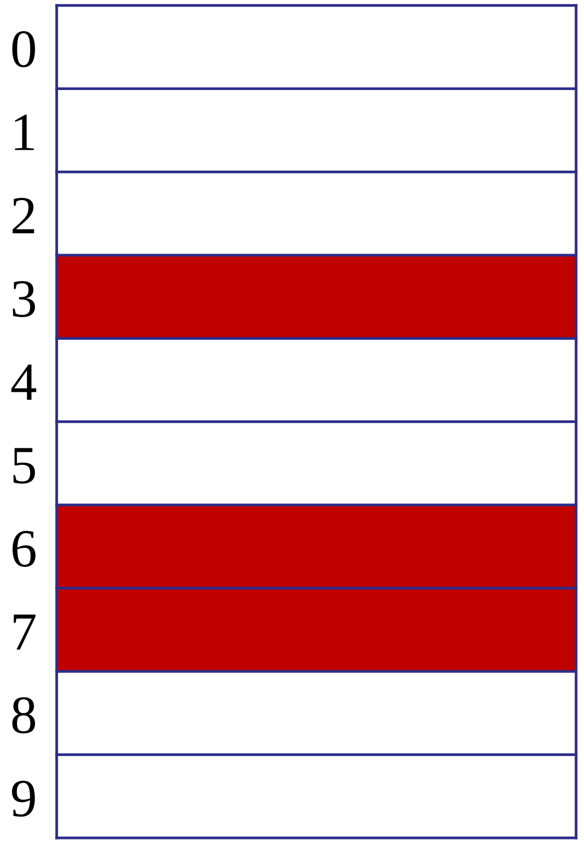
$$\frac{1}{1-\alpha}$$

Example: if (α=90%), then E[# probes] = 10

Performance of Open Addressing

Proof of Claim:

- First probe: probability that first bucket is full is: *n*/*m*

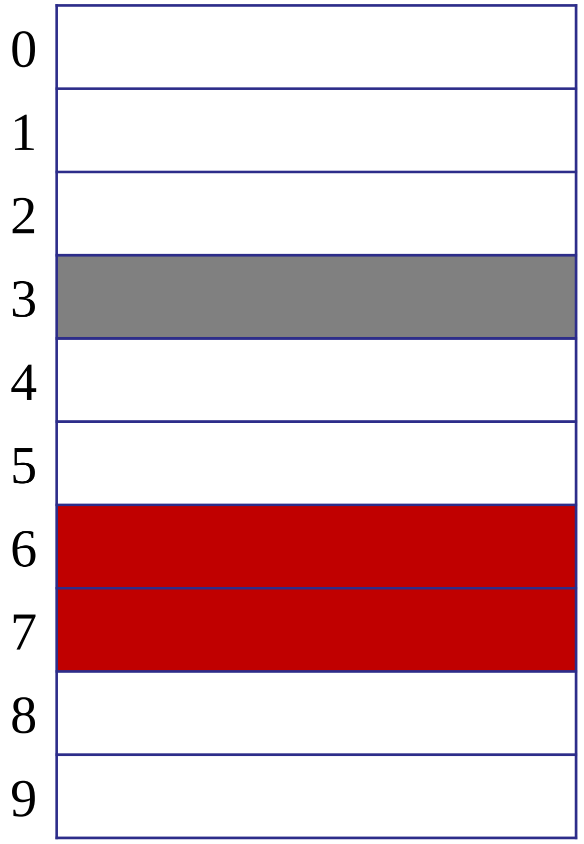


Performance of Open Addressing

Proof of Claim:

- First probe: probability that first bucket is full is: n/m

- Second probe: if first bucket is full, then the probability that the second bucket is also full: $(n-1)/(m-1)$



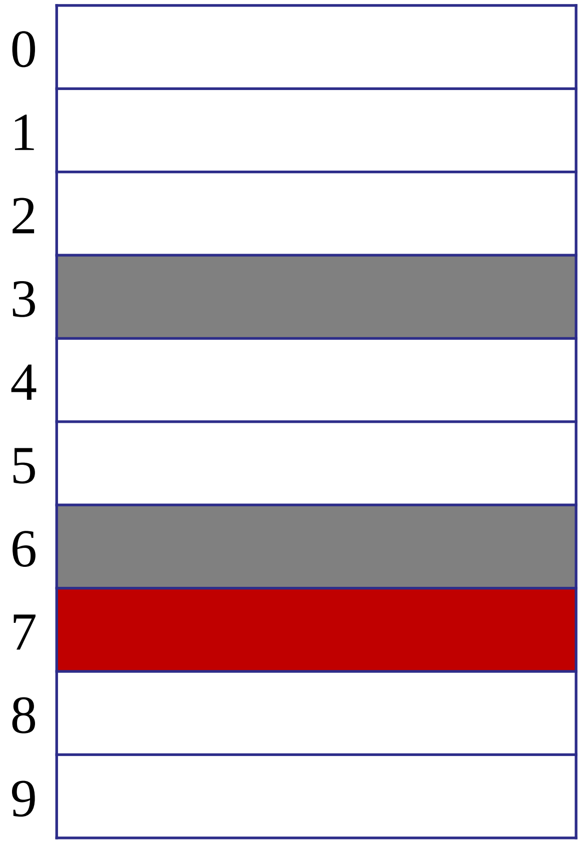
Performance of Open Addressing

Proof of Claim:

- First probe: probability that first bucket is full is: n/m

- Second probe: if first bucket is full, then the probability that the second bucket is also full: $(n - 1) / (m - 1)$

- Third probe: probability is full: $(n - 2) / (m - 2)$



Performance of Open Addressing

Proof of Claim:

- Expected cost:

$$\underbrace{1}_{\text{First probe}} + \underbrace{\frac{n}{m}}_{\text{Probability of collision on first probe}} \left(\boxed{\text{Expected cost of remaining probes}} \right)$$

Performance of Open Addressing

Proof of Claim:

– Expected cost:

$$\underbrace{1}_{\text{First probe}} + \underbrace{\frac{n}{m}}_{\text{Probability of collision on first probe}}\left(1 + \underbrace{\frac{n-1}{m-1}}_{\text{Probability of collision on second probe}}\left(\boxed{\text{Expected cost of remaining probes}}\right)\right)$$

Performance of Open Addressing

Proof of Claim:

- Expected cost:

$$1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(\boxed{\text{Expected cost of remaining probes}}\right)\right)$$

- Note that for small i:

$$\frac{n-i}{m-i}\approx\frac{n}{m}=\alpha$$

Performance of Open Addressing

Proof of Claim:

– Expected cost:

$$1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\cdots\right)\right)\right)$$

Expected cost of remaining probes

$$\approx 1+\alpha\left(1+\alpha\left(1+\alpha\left(\cdots\right)\right)\right)$$

Performance of Open Addressing

Proof of Claim:

– Expected cost:

$$1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\cdots\right)\right)\right)$$

Expected cost of remaining probes

$$\approx 1+\alpha\left(1+\alpha\left(1+\alpha(\cdots)\right)\right)$$

$$= 1+\alpha+\alpha^2+\alpha^3+\cdots$$

Performance of Open Addressing

Proof of Claim:

– Expected cost:

$$1+\frac{n}{m}\left(1+\frac{n-1}{m-1}\left(1+\frac{n-2}{m-2}\left(\cdots\right)\right)\right)$$

Expected cost of remaining probes

$$\approx 1+\alpha\left(1+\alpha\left(1+\alpha(\cdots)\right)\right)$$

$$= 1+\alpha+\alpha^2+\alpha^3+\cdots$$

$$= \frac{1}{1-\alpha}$$

Performance of Open Addressing

Define:

- Load $\alpha = n / m$ ← Average # items / bucket
- Assume $\alpha < 1$.

Claim:

For n items, in a table of size m, assuming *uniform hashing*, the expected cost of an operation is:

$$\frac{1}{1-\alpha}$$

Example: if (α=90%), then E[# probes] = 10

Advantages…

Open addressing:

- Saves space
  - Empty slots vs. linked lists.
- Rarely allocate memory
  - No new list-node allocations.
- Better cache performance
  - Table all in one place in memory
  - Fewer accesses to bring table into cache.
  - Linked lists can wander all over the memory.

Disadvantages…

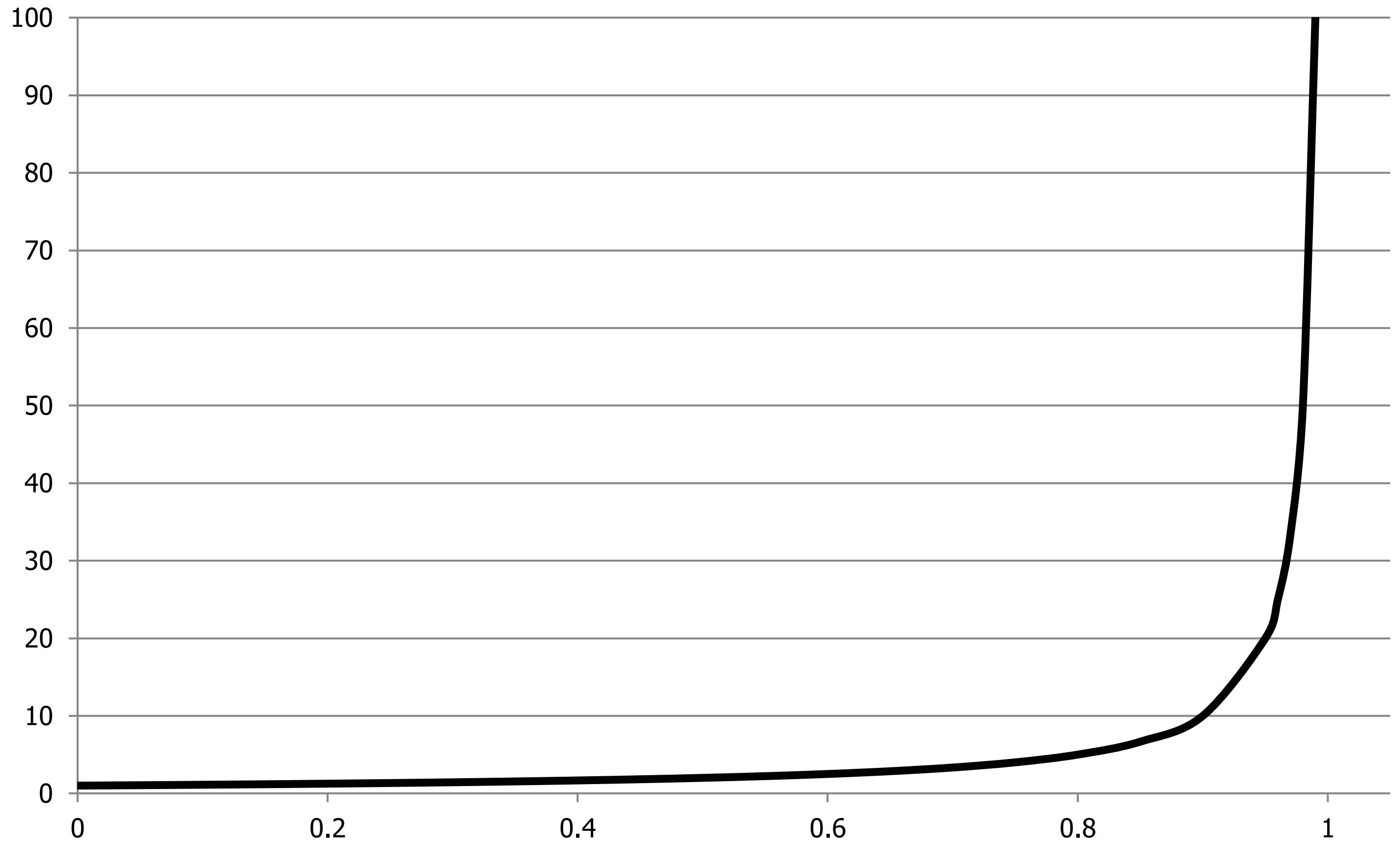
Open addressing:

- More sensitive to choice of hash functions.
  - Clustering is a common problem.
  - See issues with linear probing.
- More sensitive to load.
  - Performance degrades badly as $\alpha \rightarrow 1$.

Disadvantages…

Open addressing:

– Performance degrades badly as $\alpha \rightarrow 1$.



Roadmap

Today: Graph Basics

- What is a graph?
- Modeling problems as graphs.
- Graph representations (list vs. matrix)

Roadmap

Next: Searching Graphs

- Searching graphs
- Shortest path problem
- Bellman-Ford Algorithm
- Dijkstra’s Algorithm

Roadmap

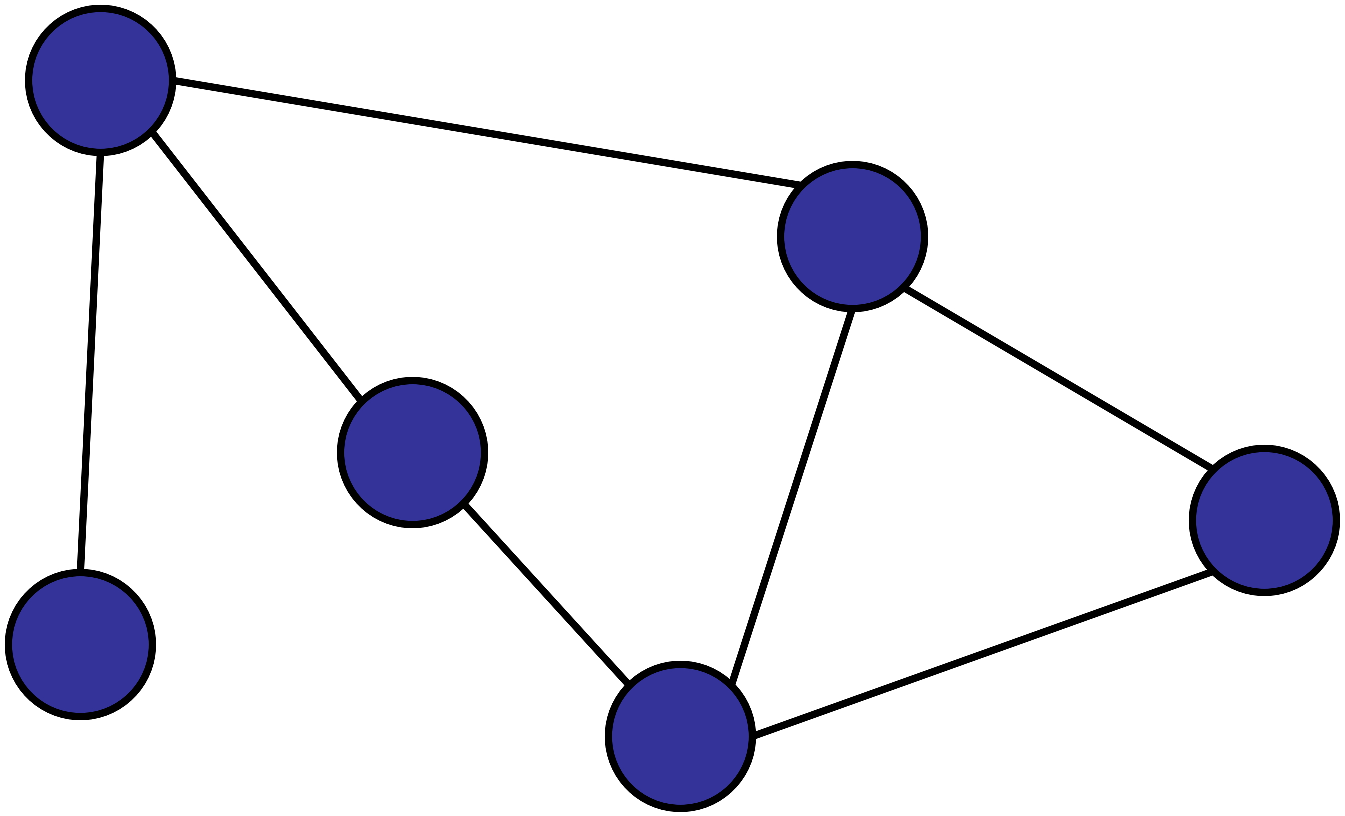
Next next:

- The Minimum Spanning Tree Problem
    - Kruskal’s Algorithm
    - Prim’s Algorithm

What is a graph?

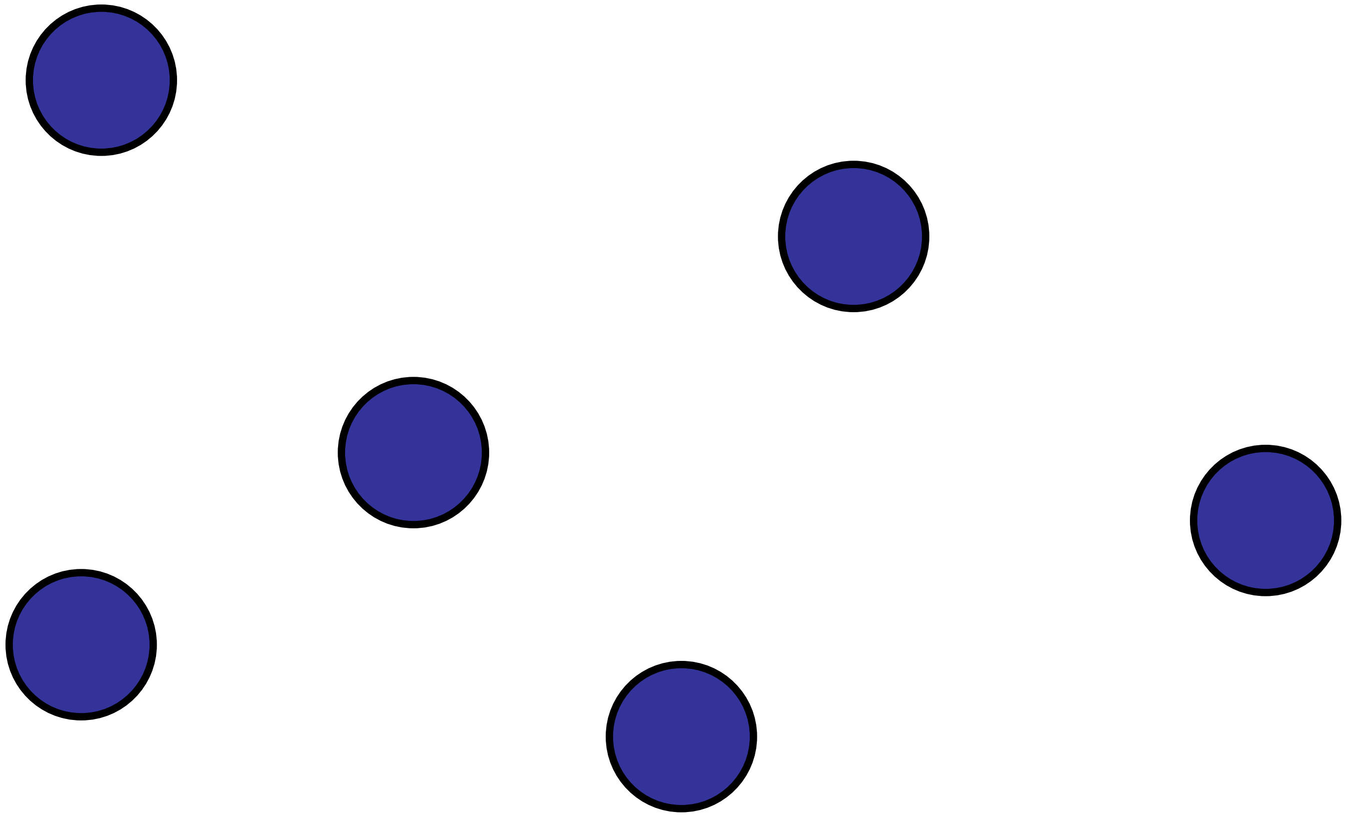
Is it a graph?

1. [✓ Yes]
2. No.



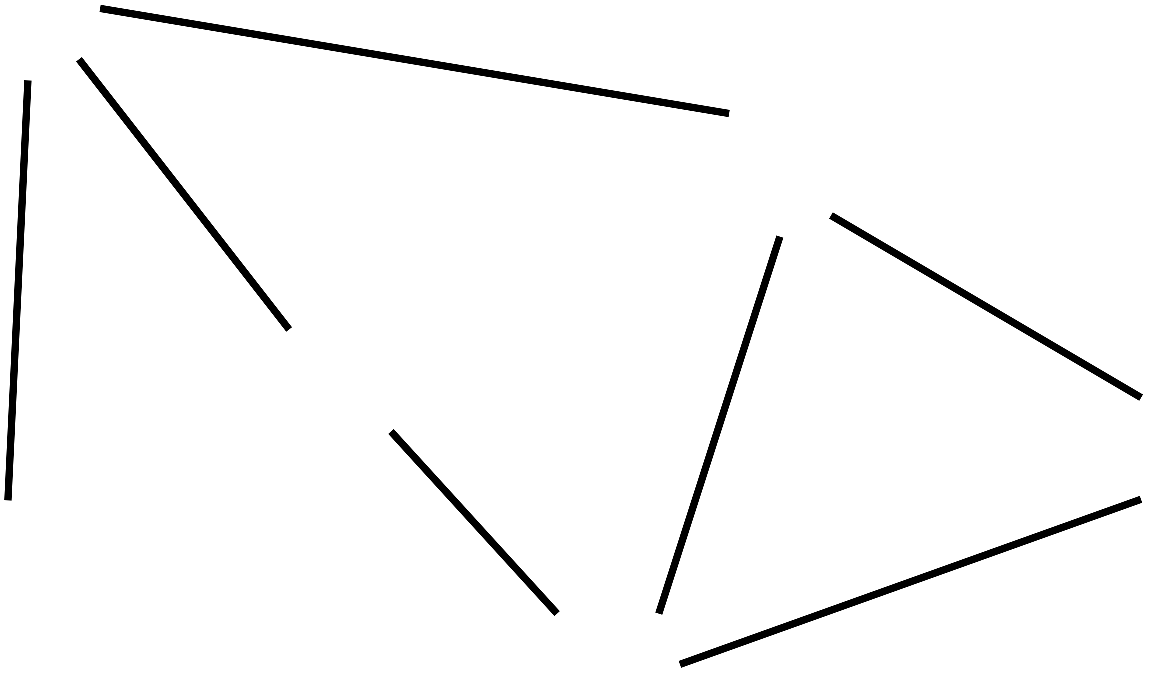
Is it a graph?

1. [✓ Yes]
2. No.



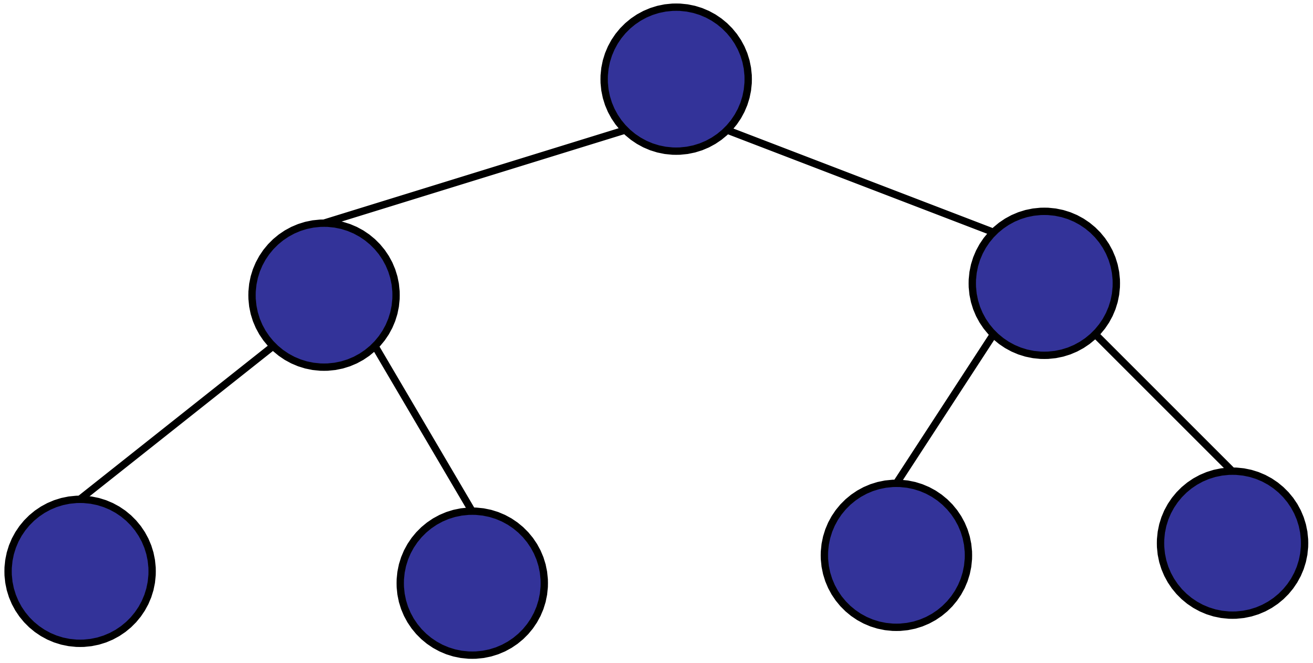
Is it a graph?

1. Yes
2. [✓ No.]



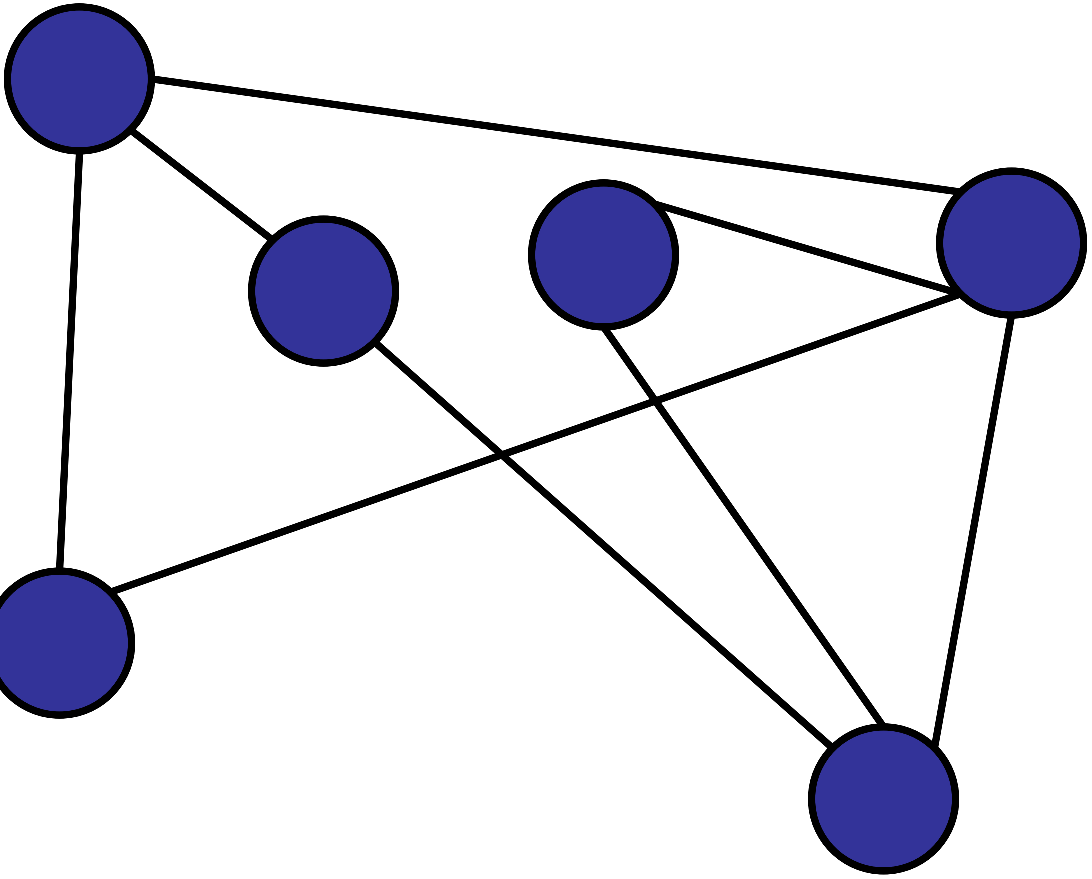
Is it a graph?

1. [✓ Yes]
2. No.



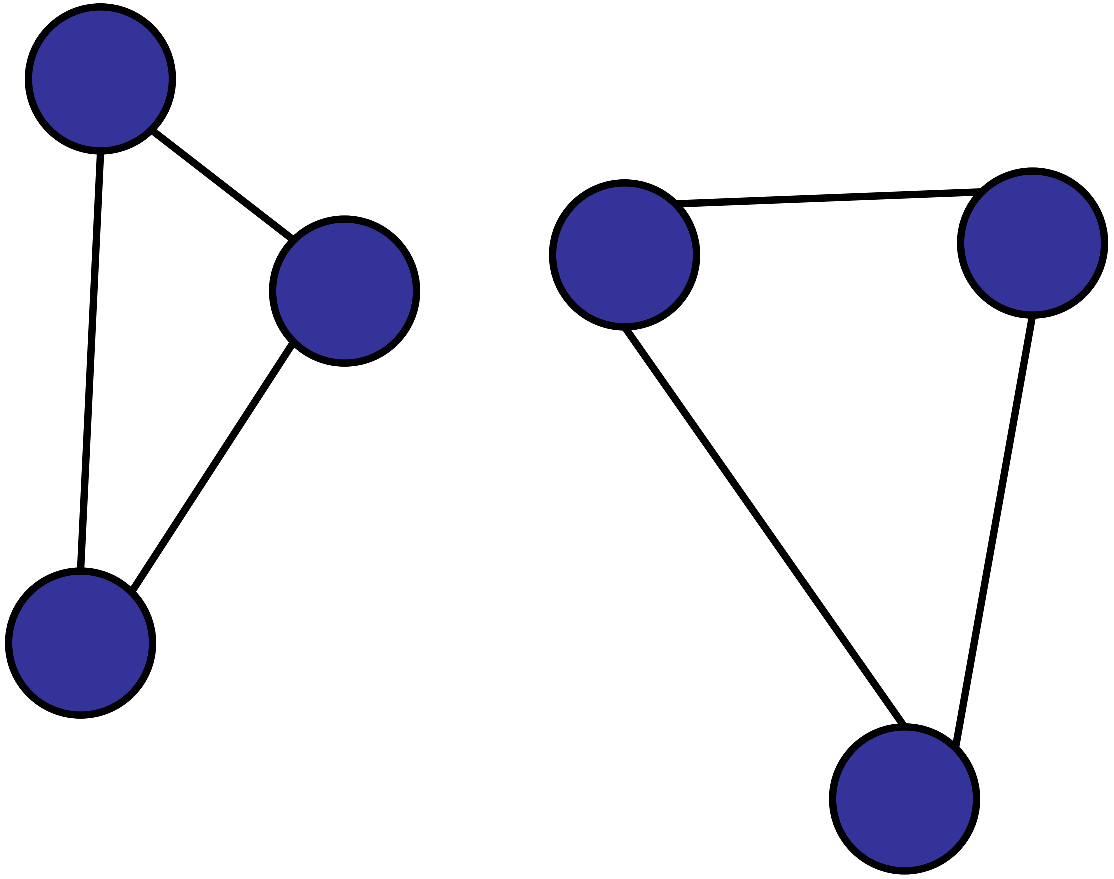
Is it a graph?

1. [✓ Yes]
2. No.



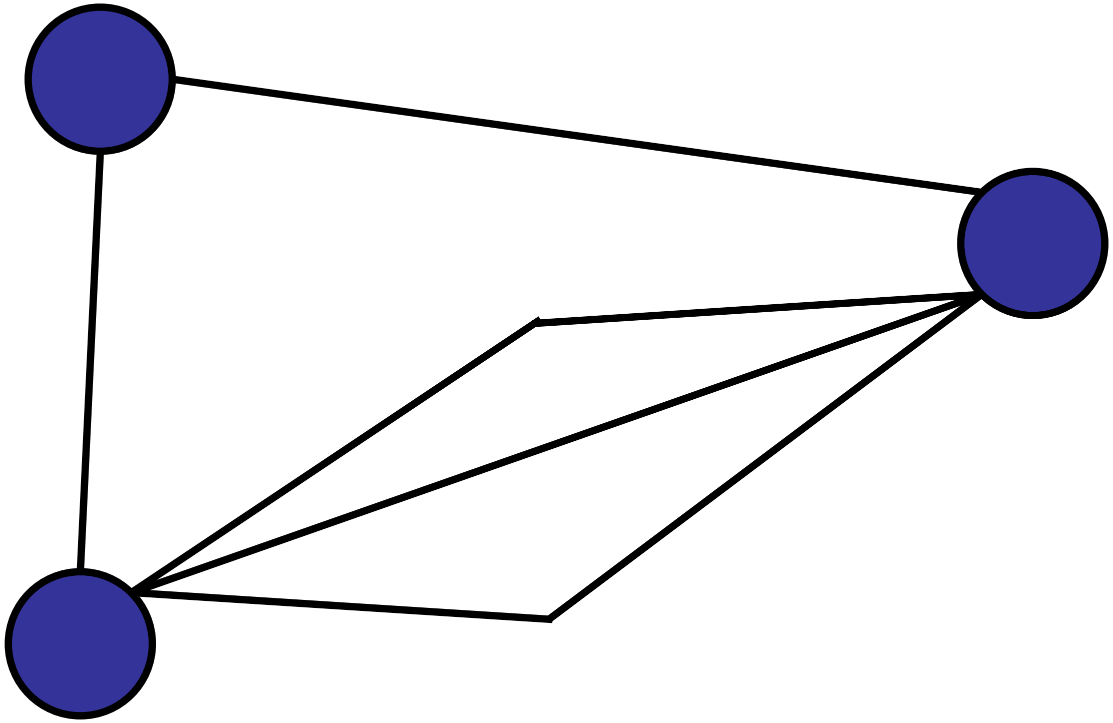
Is it a graph?

1. [✓ Yes]
2. No.



Is it a graph?

1. Yes
2. [✓ No.]



Is it a graph?

1. [✓ Yes]
2. No.

What is a graph?

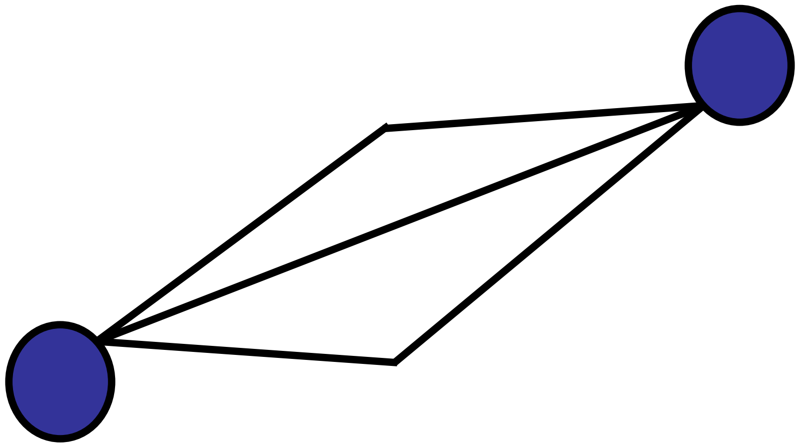
Graph consists of two types of elements:

- Nodes (or vertices)
  - At least one.
- Edges (or arcs)
  - Each edge connects two nodes in the graph
  - Each edge is unique.

What is a multigraph?

Graph consists of two types of elements:

- Nodes (or vertices)
  - At least one.



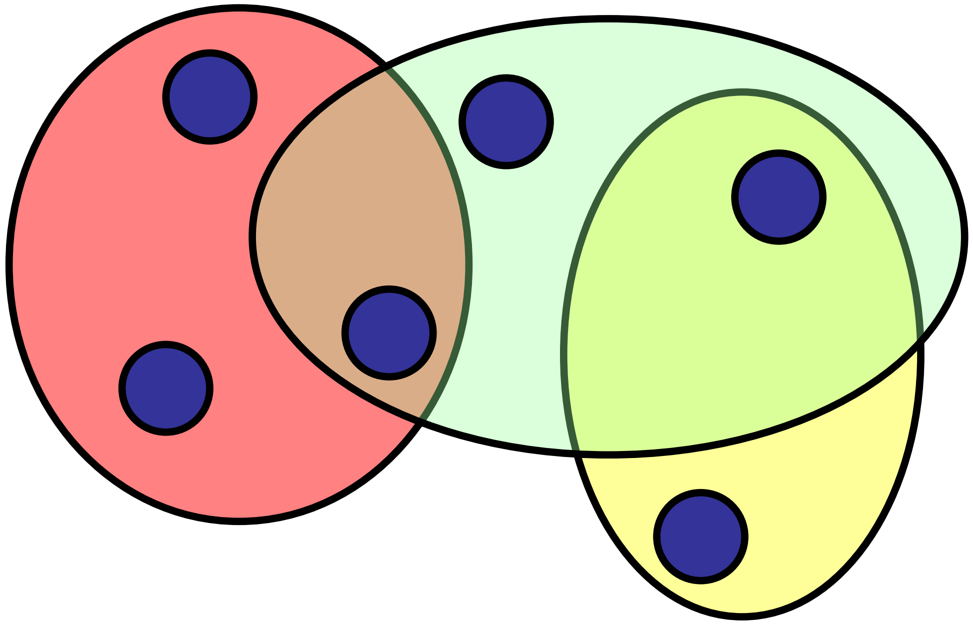
- Edges (or arcs)
  - Each edge connects two nodes in the graph
  - Two nodes may be connected by more than one edge.

(Rare in CS2040S.)

What is a hypergraph?

Graph consists of two types of elements:

- Nodes (or vertices)
  - At least one.
- Edges (or arcs)
  - Each edge connects >= 2 nodes in the graph
  - Each edge is unique.



(Not common in CS2040S)

What is a graph?

Graph $G = \langle V, E \rangle$

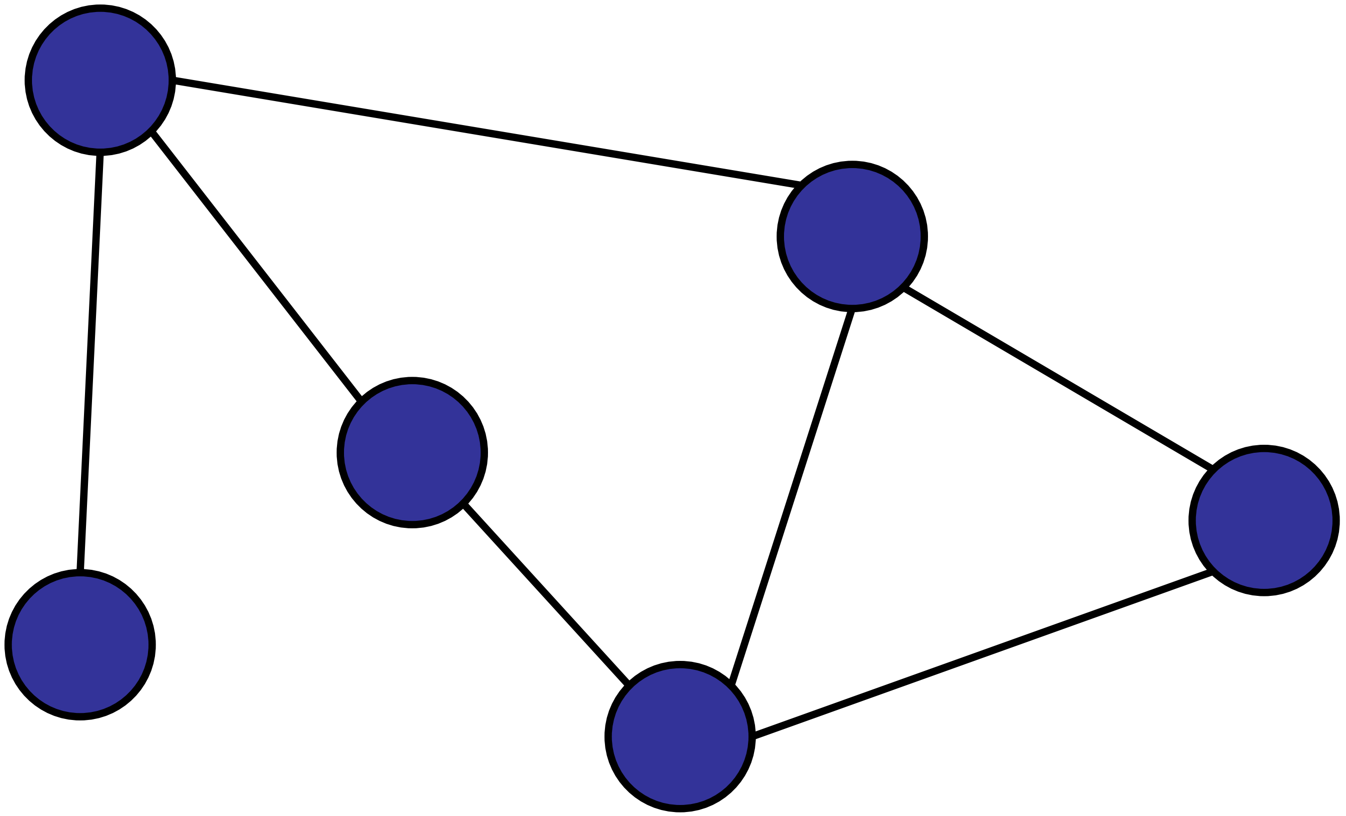
V is a set of nodes

- At least one: $|V| > 0$.

E is a set of edges:

- $E \subseteq \{ (v,w) : (v \in V), (w \in V) \}$
- $e = (v,w)$
- For all $e_1, e_2 \in E : e_1 \neq e_2$

Graph Terminology

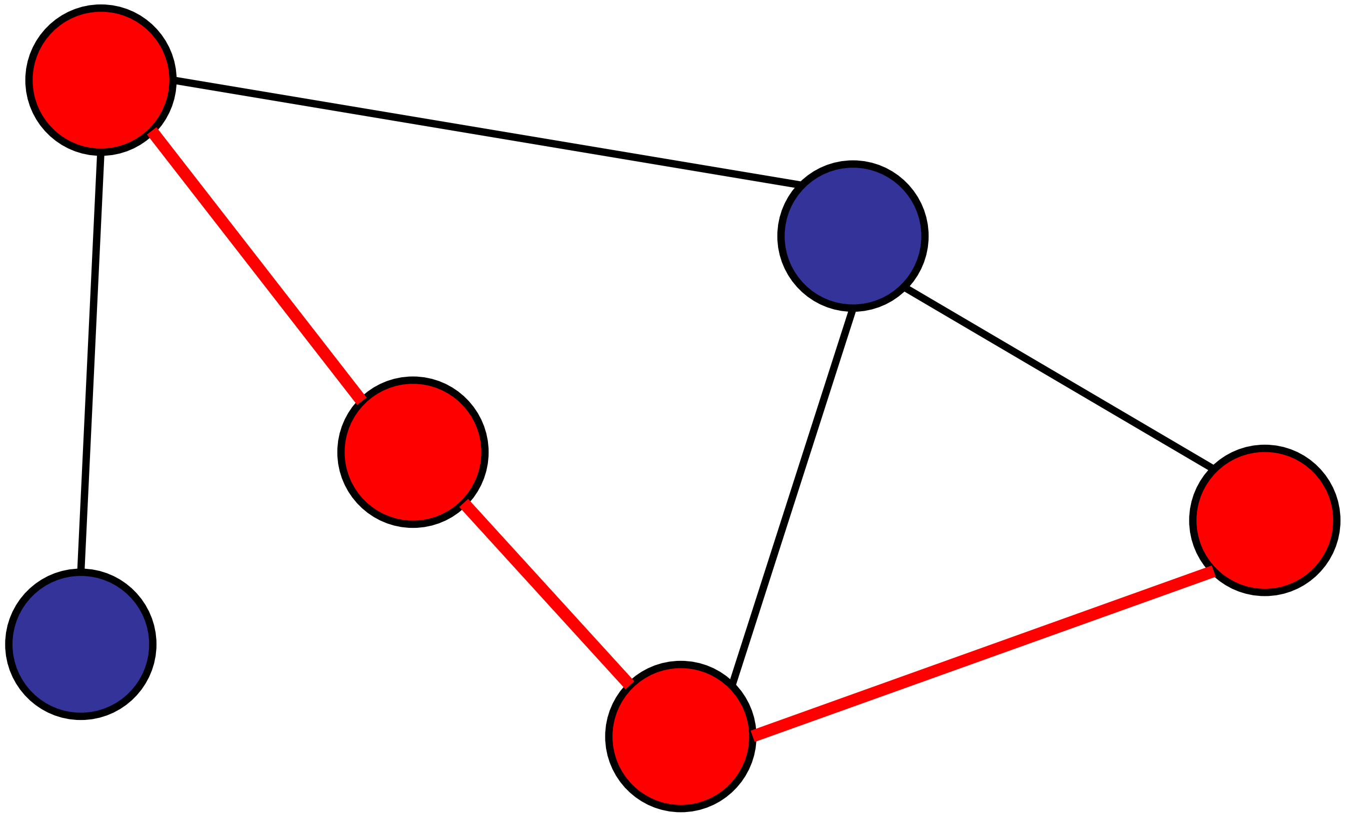


Graph Terminology

(Simple) Path:

Set of edges connecting two nodes.

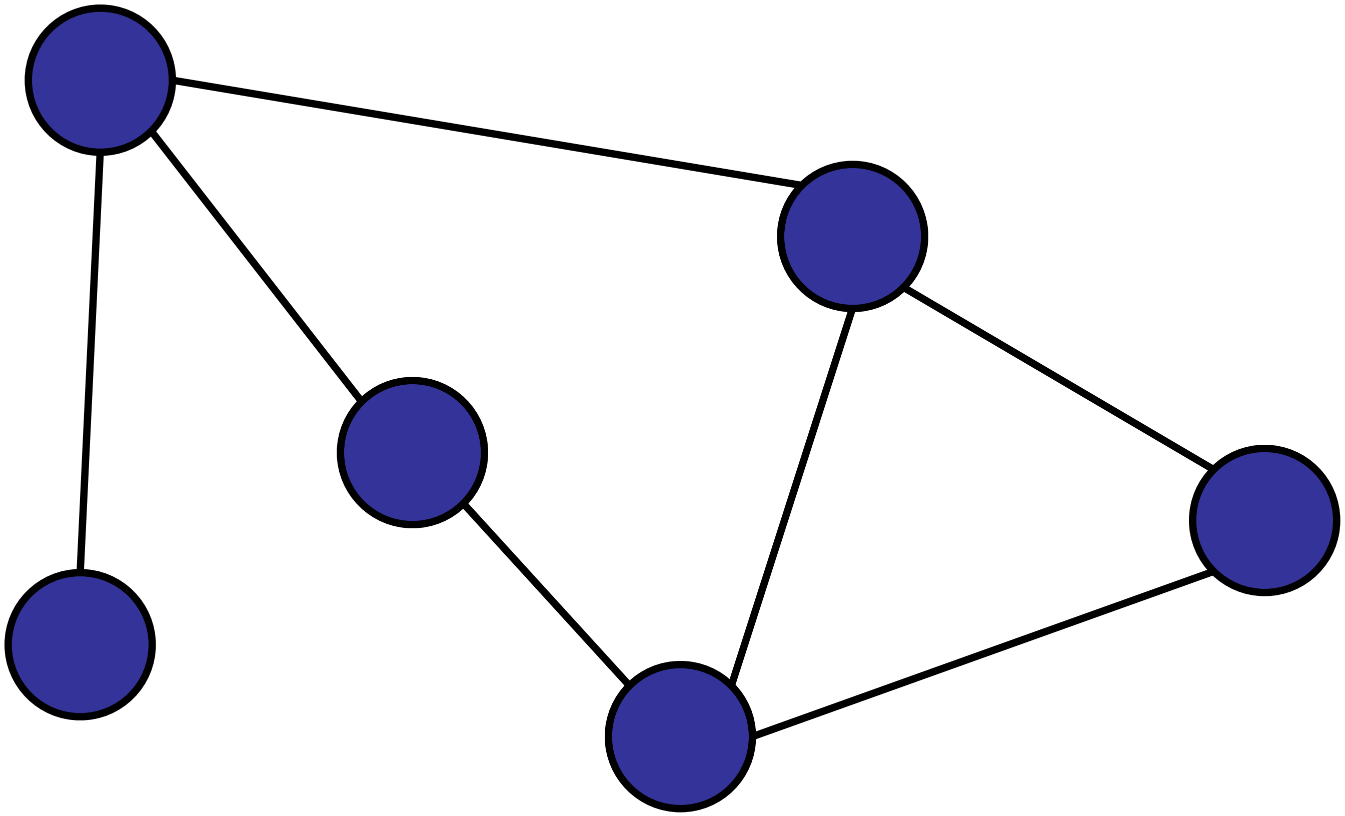
Path intersects each node at most once.



Graph Terminology

Connected:

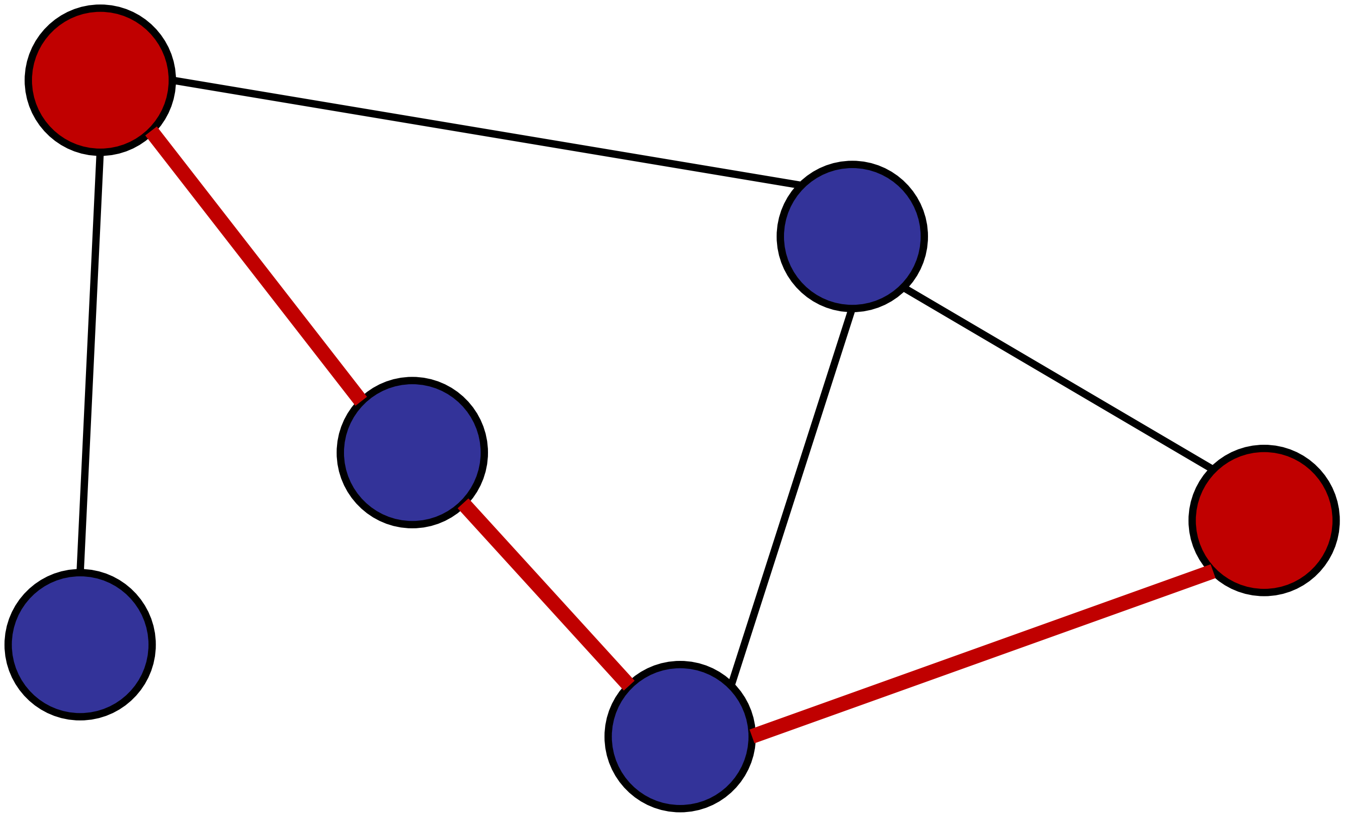
Every pair of nodes is connected by a path.



Graph Terminology

Connected:

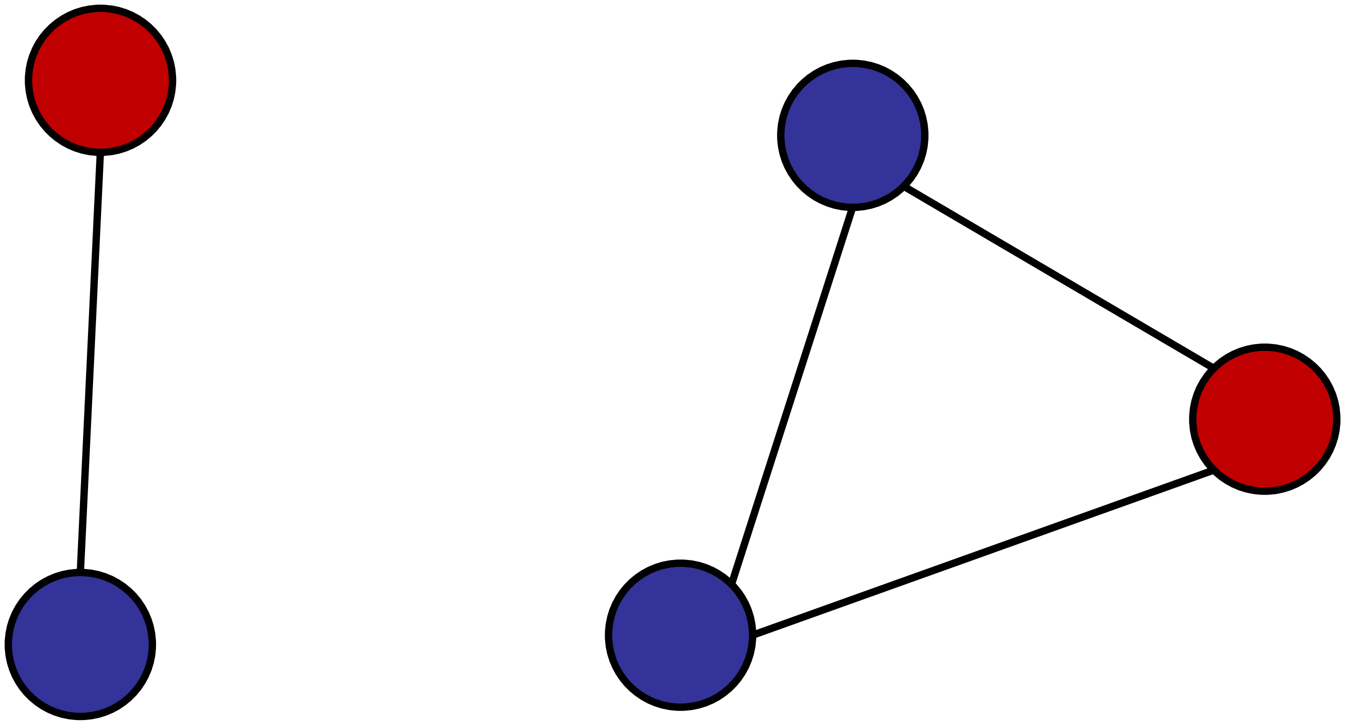
Every pair of nodes is connected by a path.



Graph Terminology

Disconnected:

- Some pair of nodes is not connected by a path.

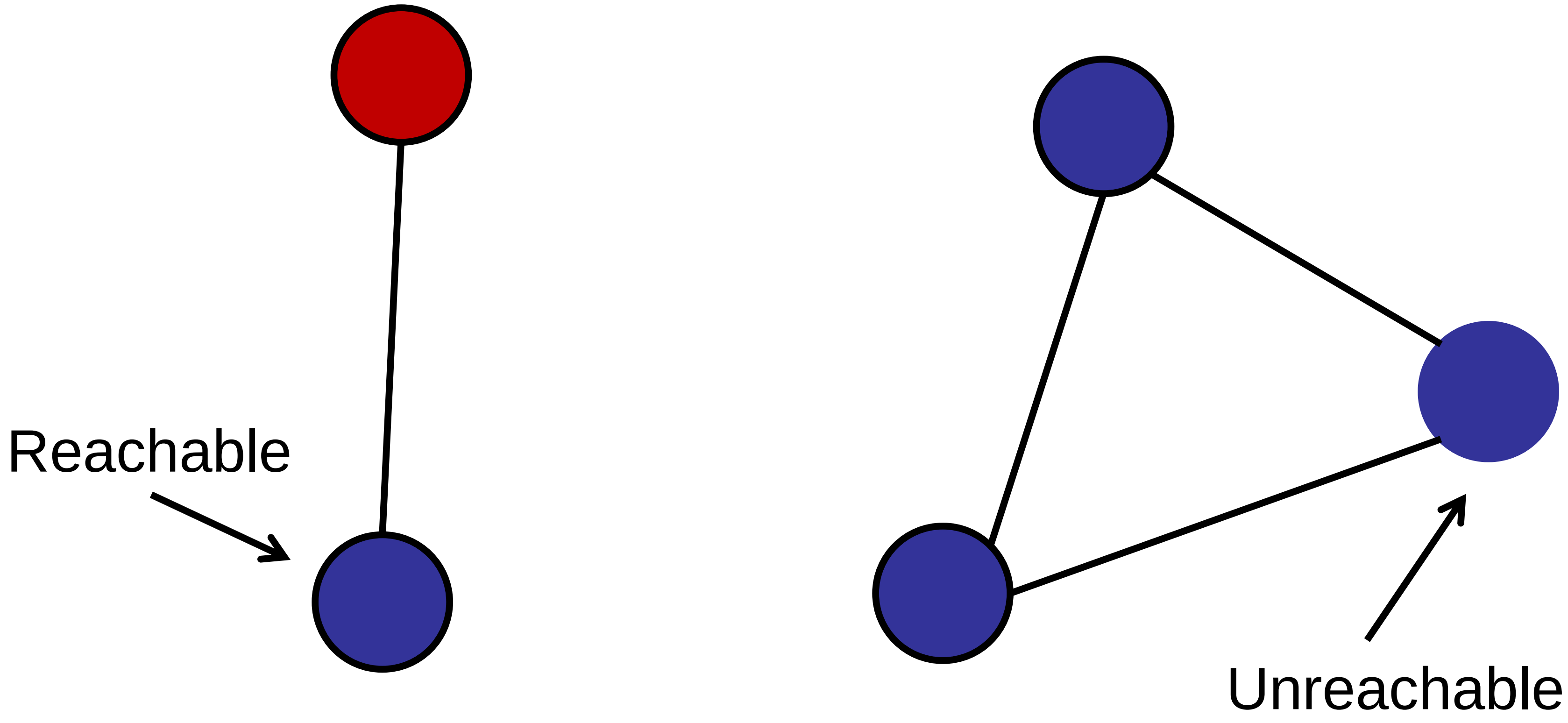


- Two **connected components.**

Graph Terminology

Disconnected:

- Some pair of nodes is not connected by a path.

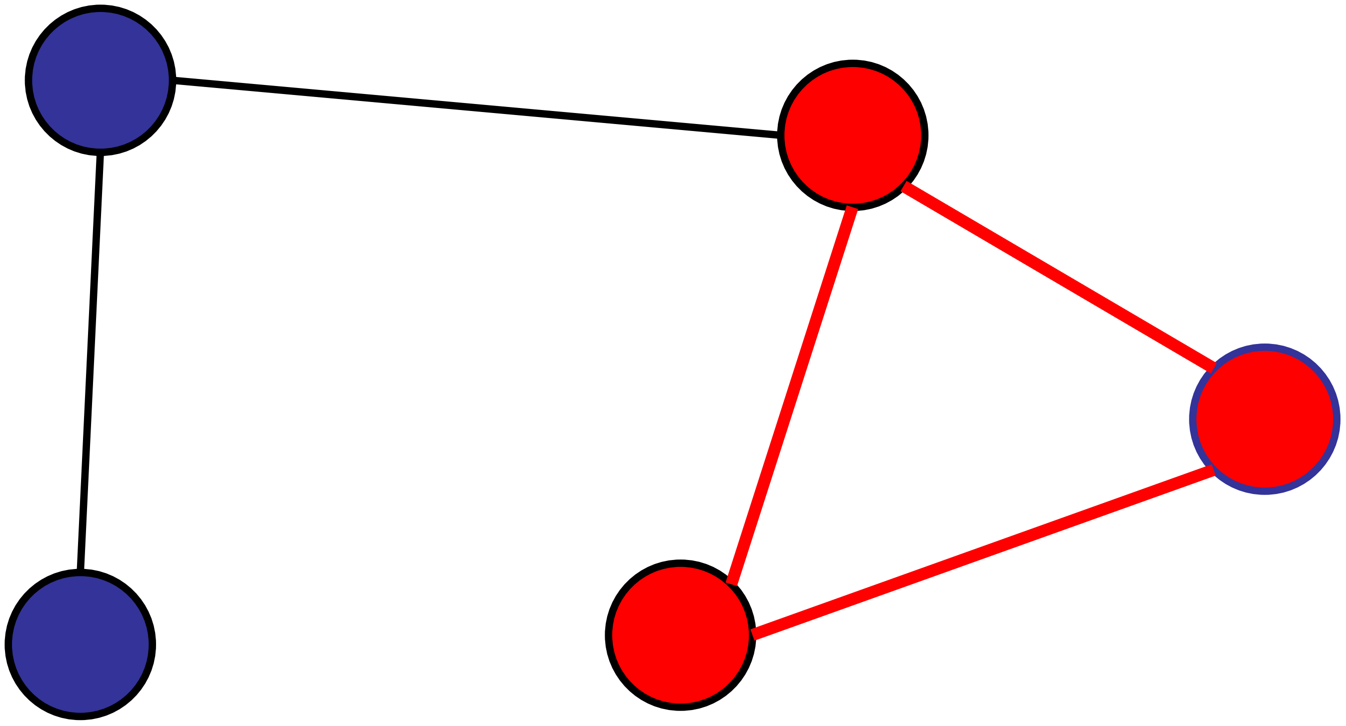


- Two **connected components.**

Graph Terminology

Cycle:

“Path” where first and last node are the same.

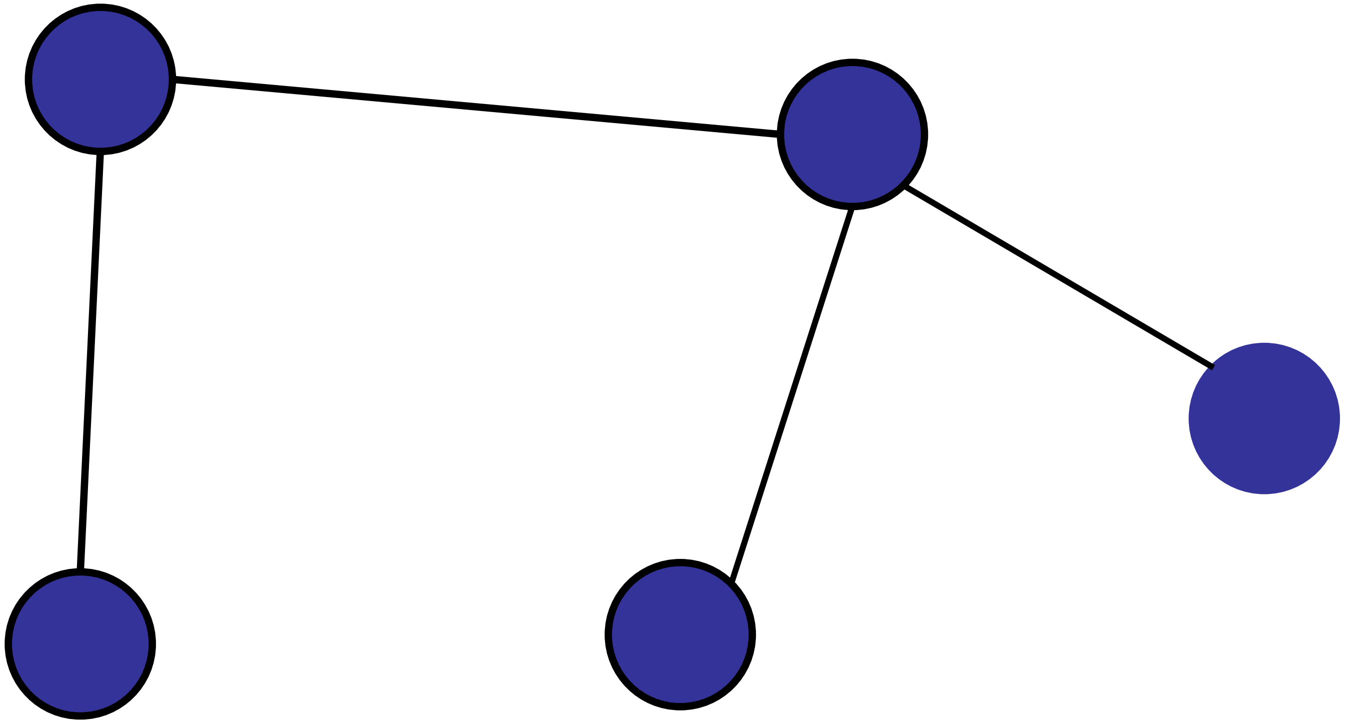


(**Not actually a path, since one node appears twice.)

Graph Terminology

(Unrooted) Tree:

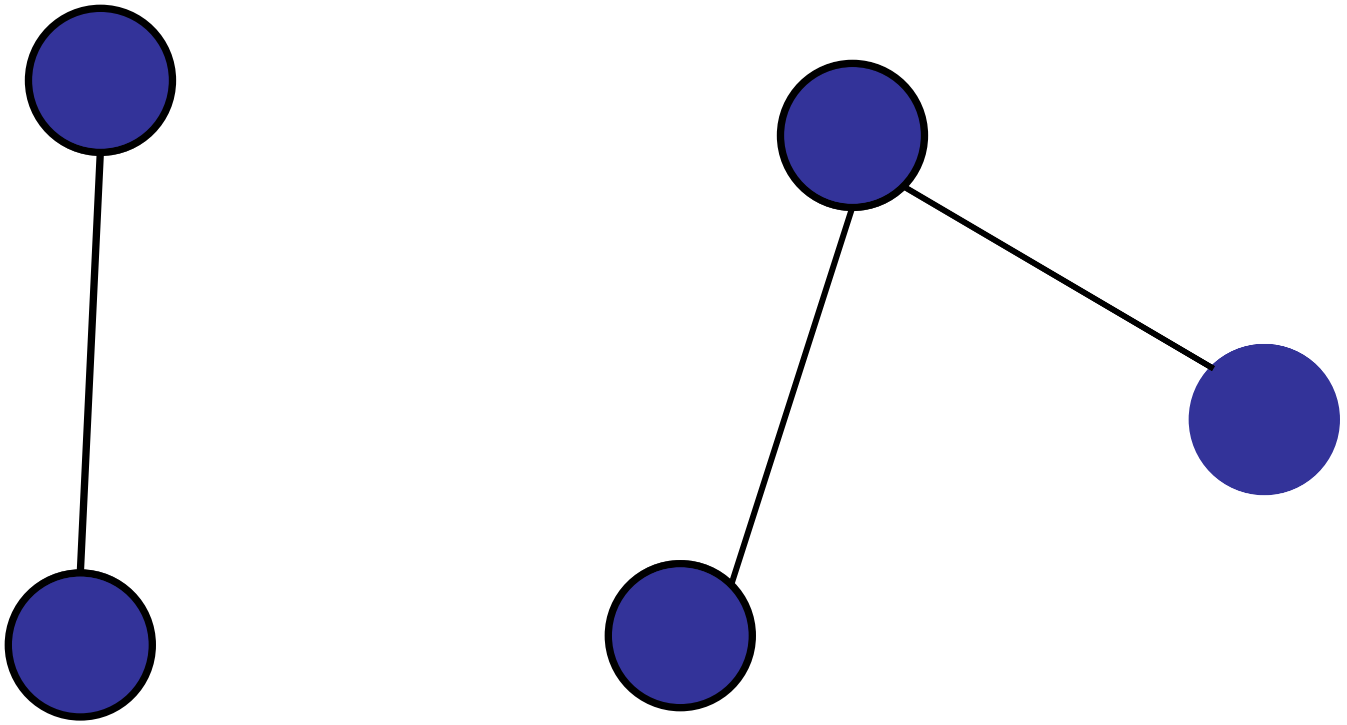
Connected graph with no cycles.



Graph Terminology

Forest:

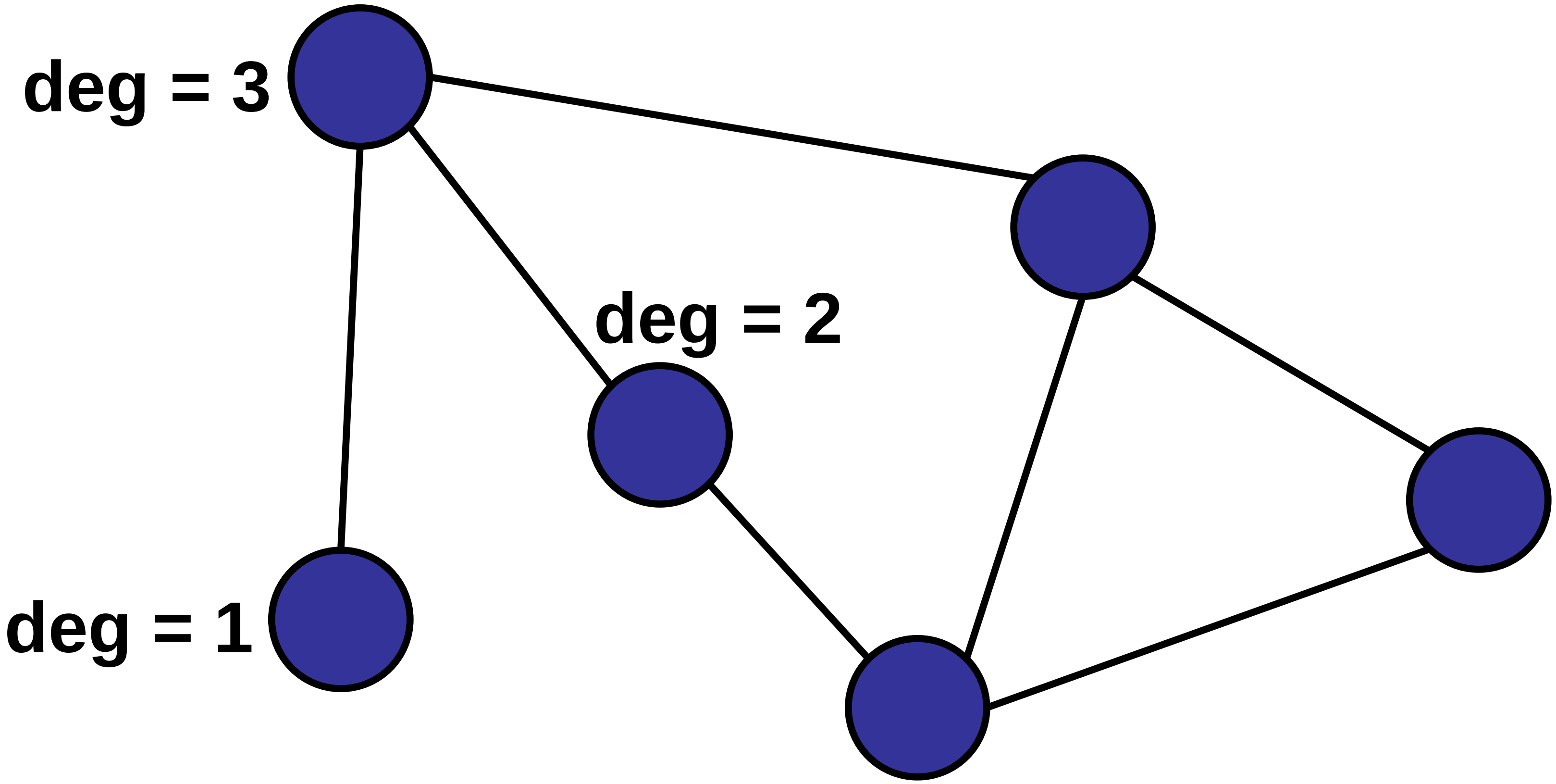
Graph with no cycles.



Graph Terminology

Degree of a node:

- Number of **adjacent** edges.

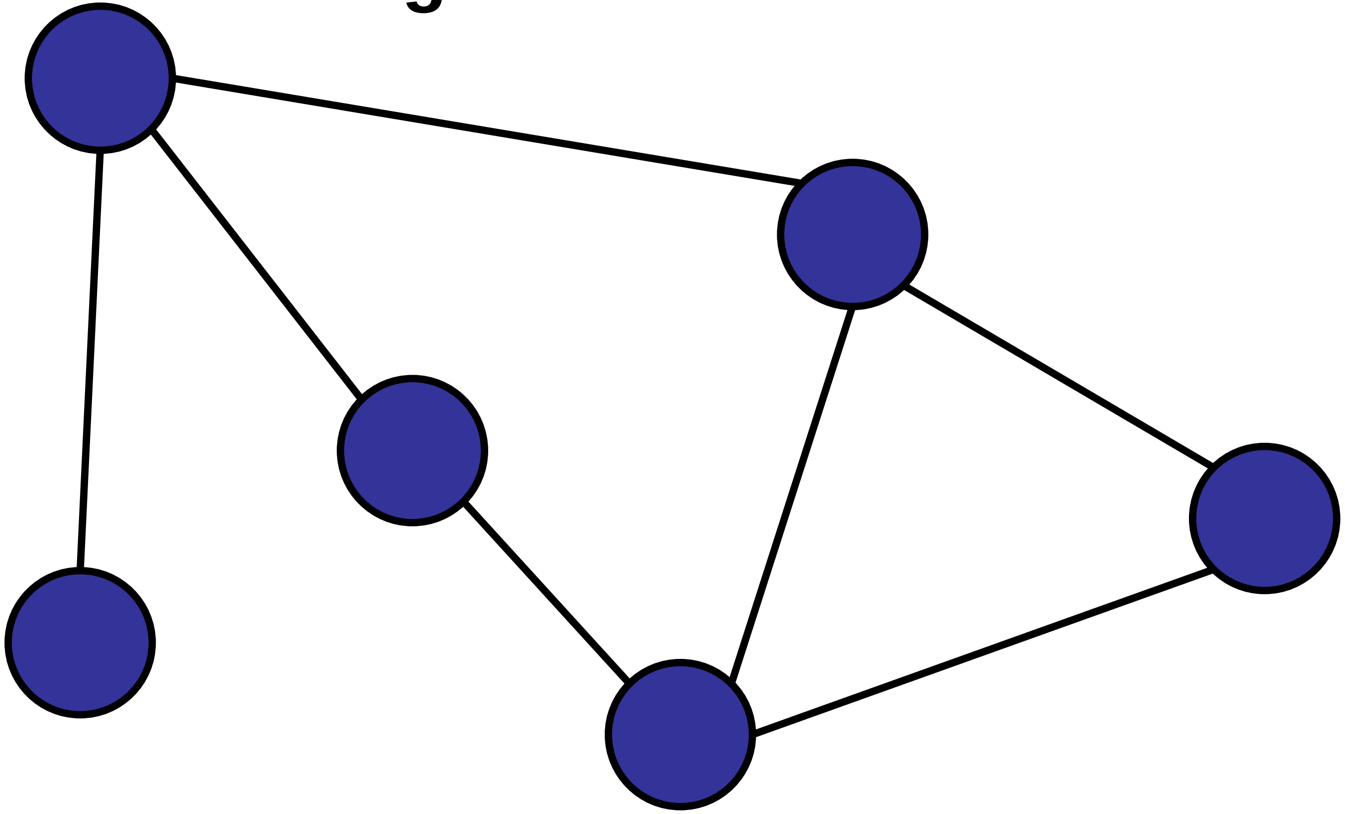


Graph Terminology

Degree of a graph:

– Maximum number of **adjacent** edges.

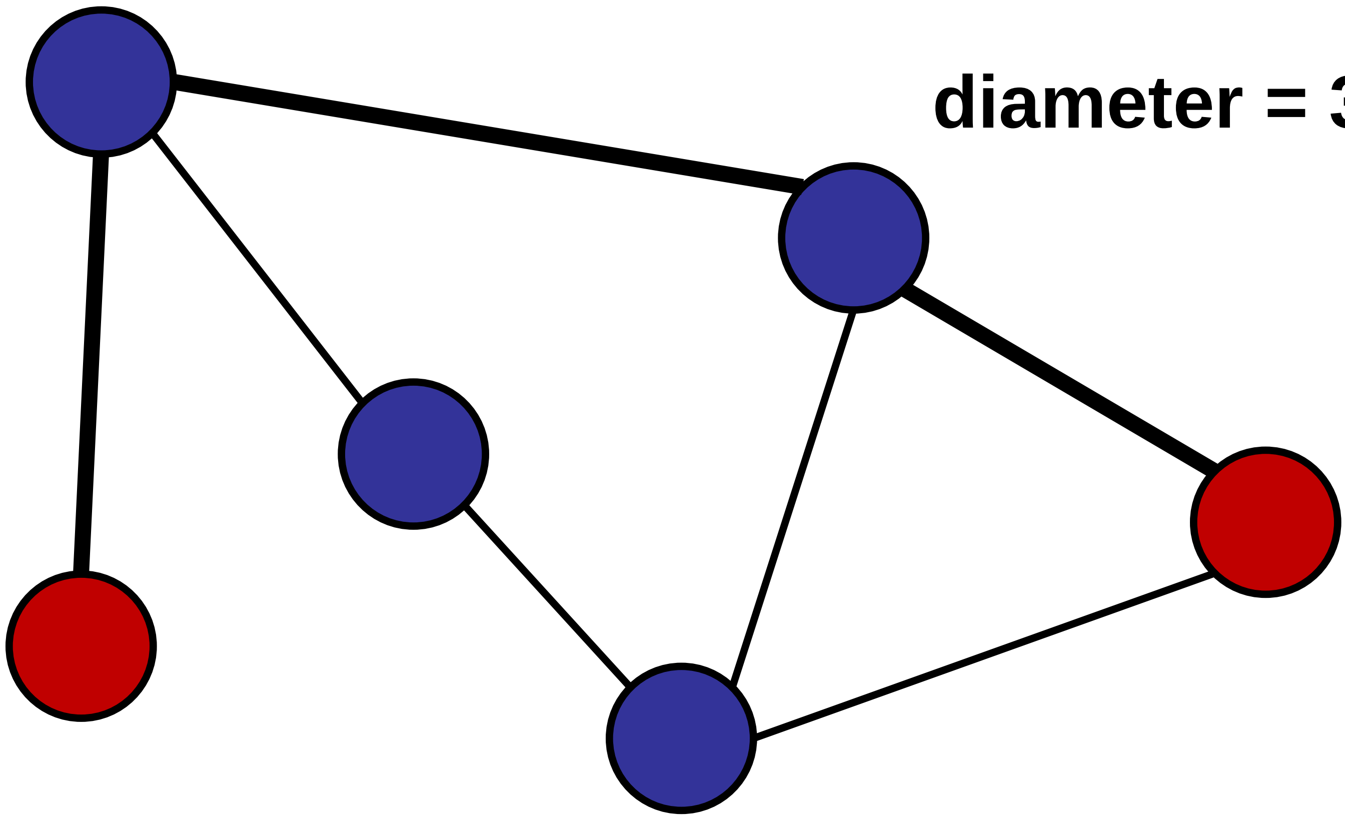
degree = 3



Graph Terminology

Diameter:

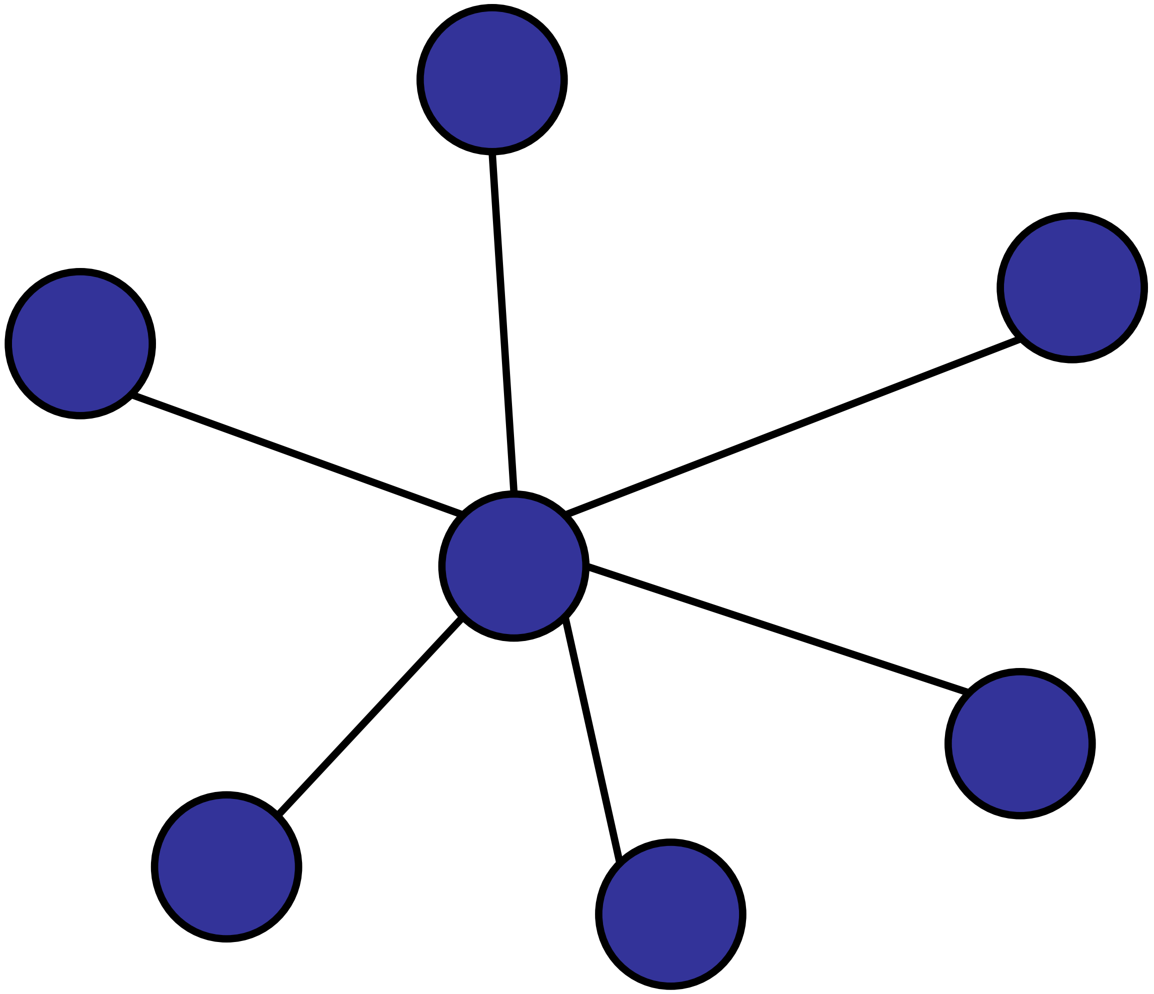
– Maximum distance between two nodes, following the shortest path.



Special Graphs

Special Graphs

Star



One central node, all edges connect center to edges.

Degree of n-node star is:

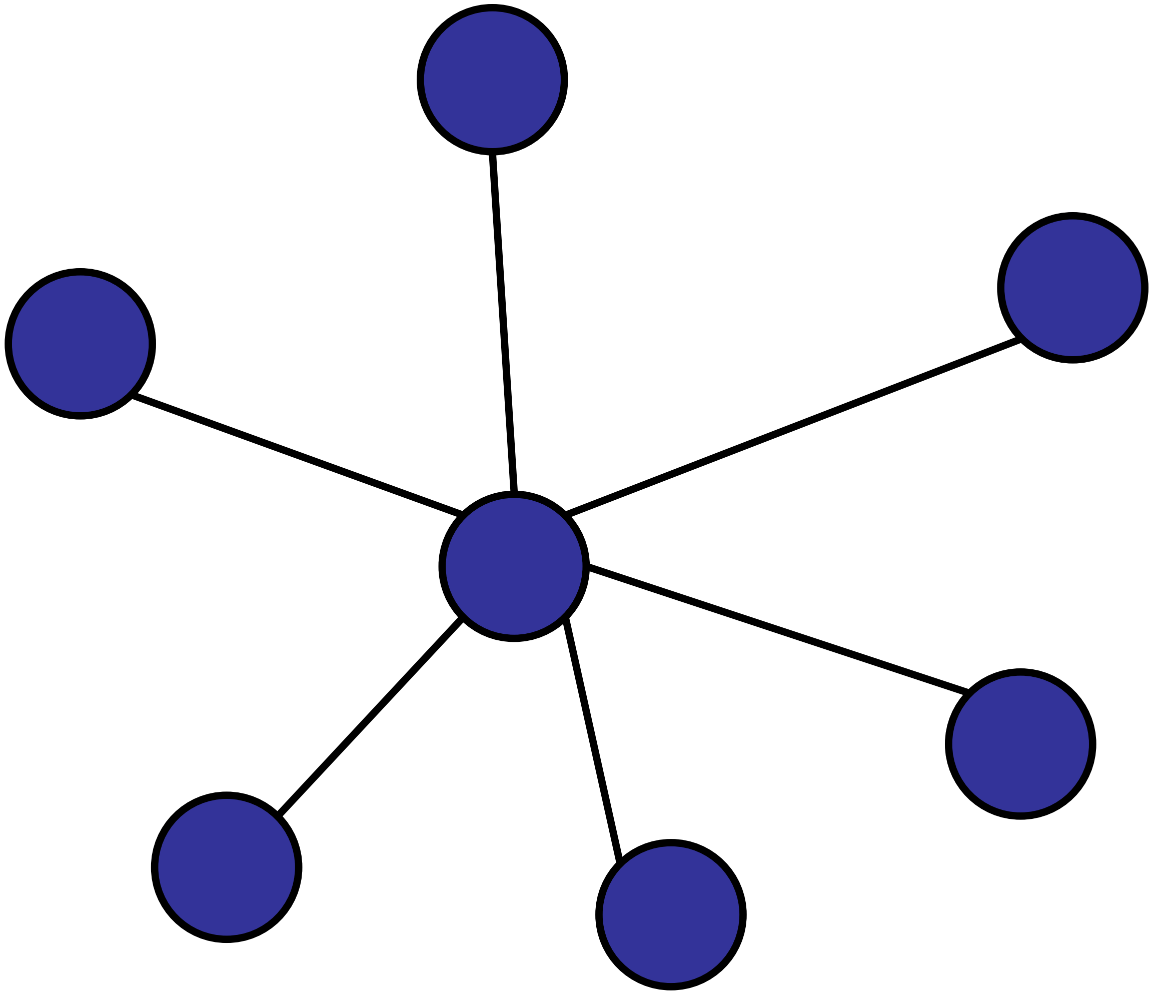
1. 1
2. 2
3. n/2
4. n-2
5. [✓ n-1]
6. n

ARCHIPELAGO

is open

Special Graphs

Star



One central node, all edges connect center to edges.

Diameter of n-node star:

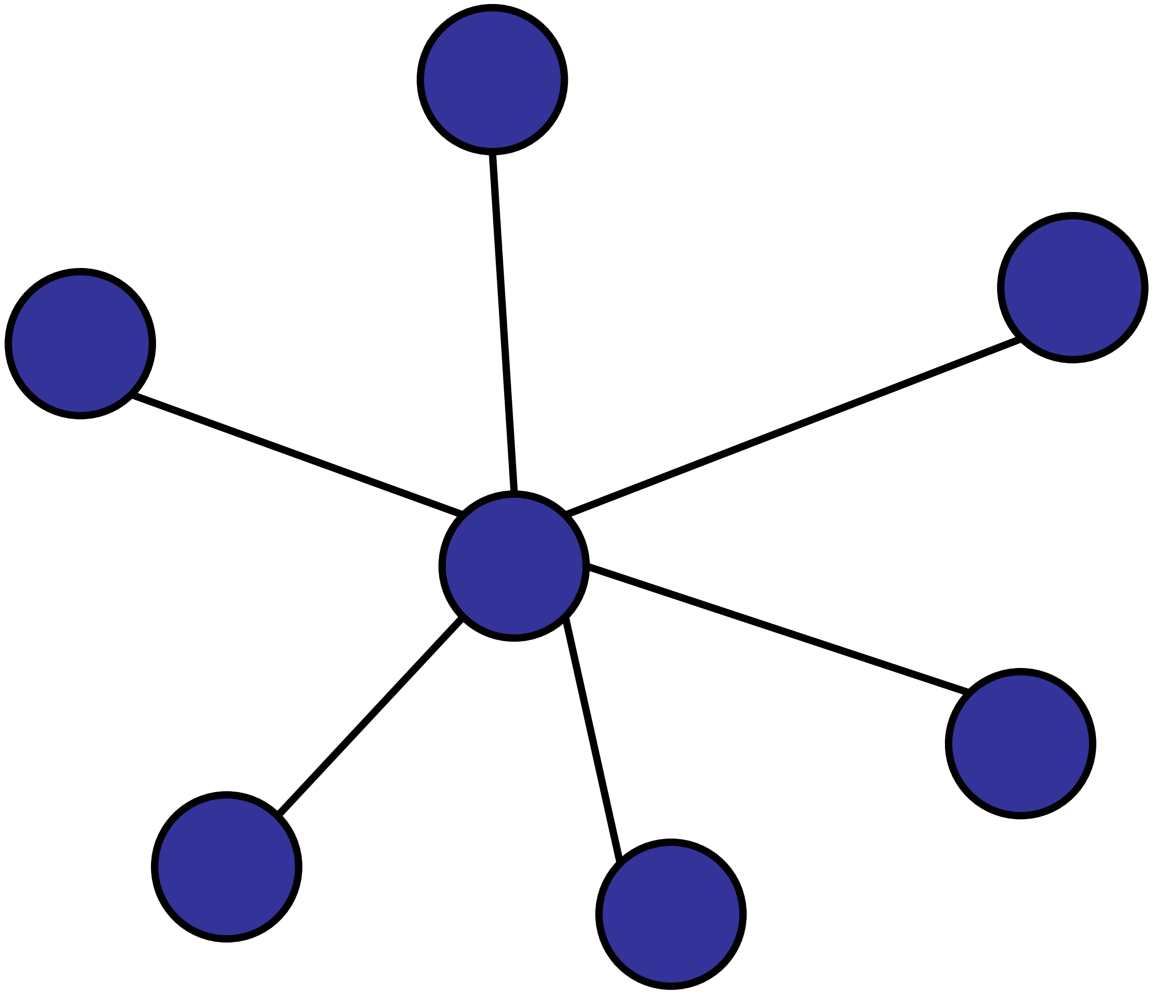
1. 1
2. [✓ 2]
3. n/2
4. n-2
5. n-1
6. n

ARCHIPELAGO

is open

Special Graphs

Star



One central node, all edges connect center to edges.

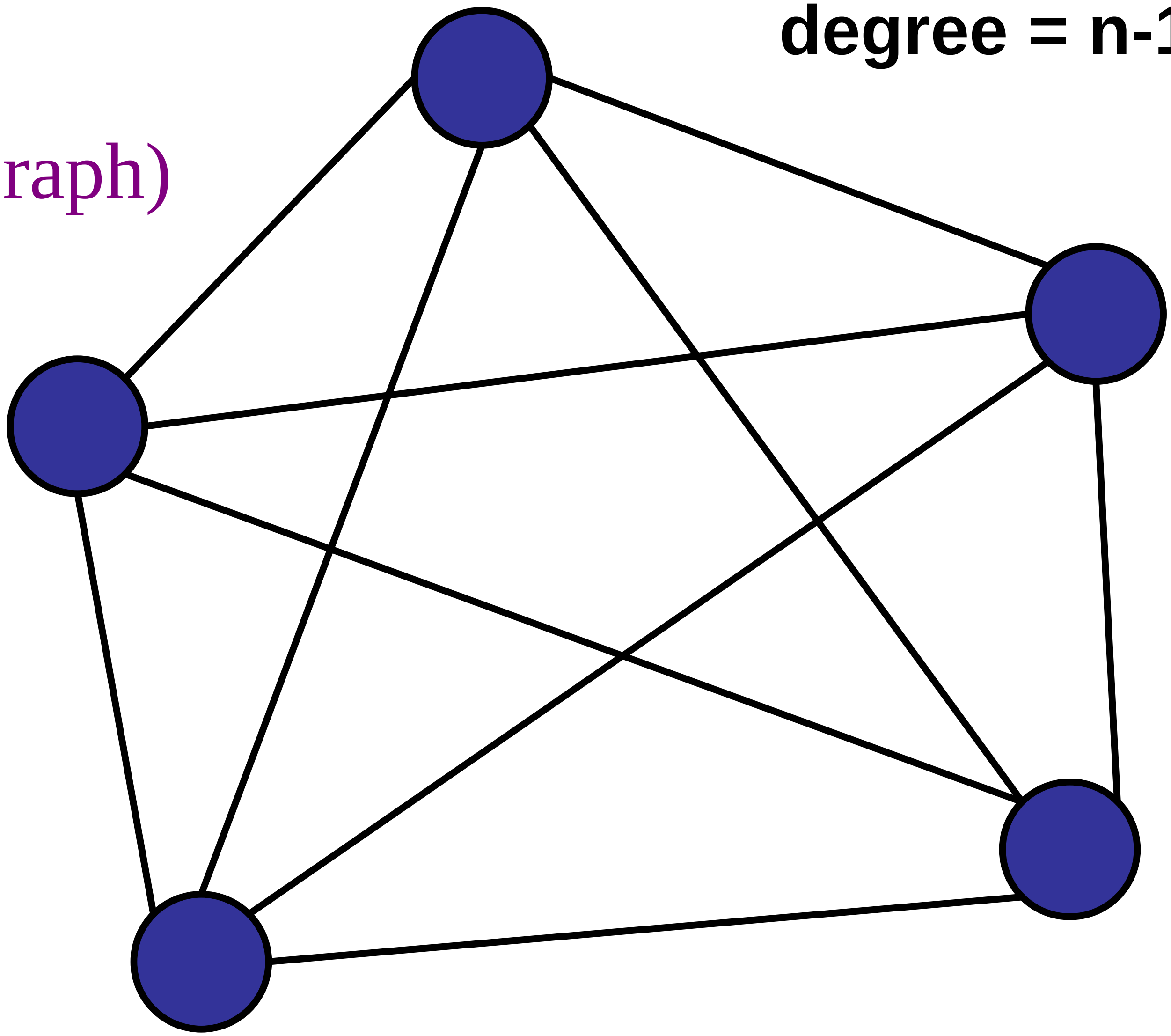
Special Graphs

Clique

(Complete Graph)

diameter = 1

degree = n-1



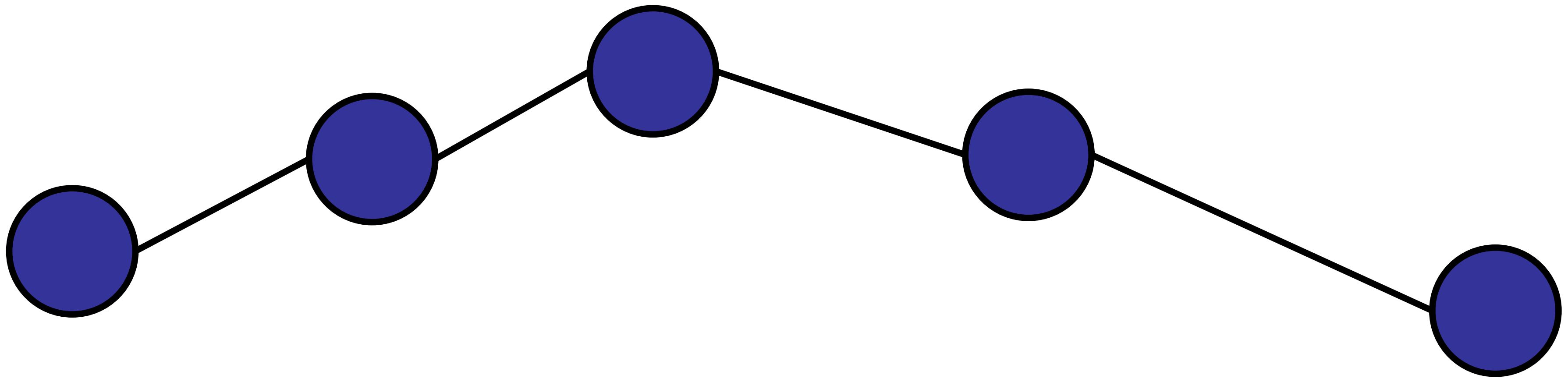
All pairs connected by edges.

Special Graphs

diameter = n-1

degree = 2

Line (or path)



Special Graphs

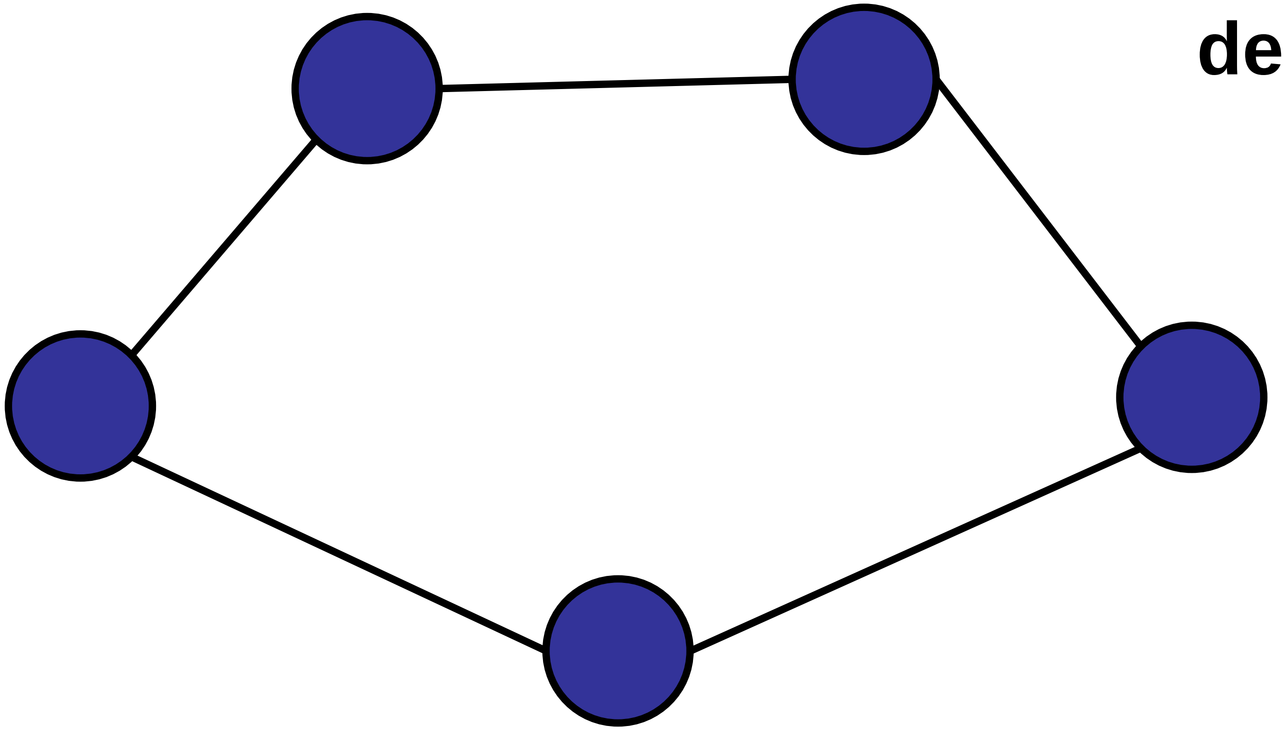
Cycle

diameter = n/2

or

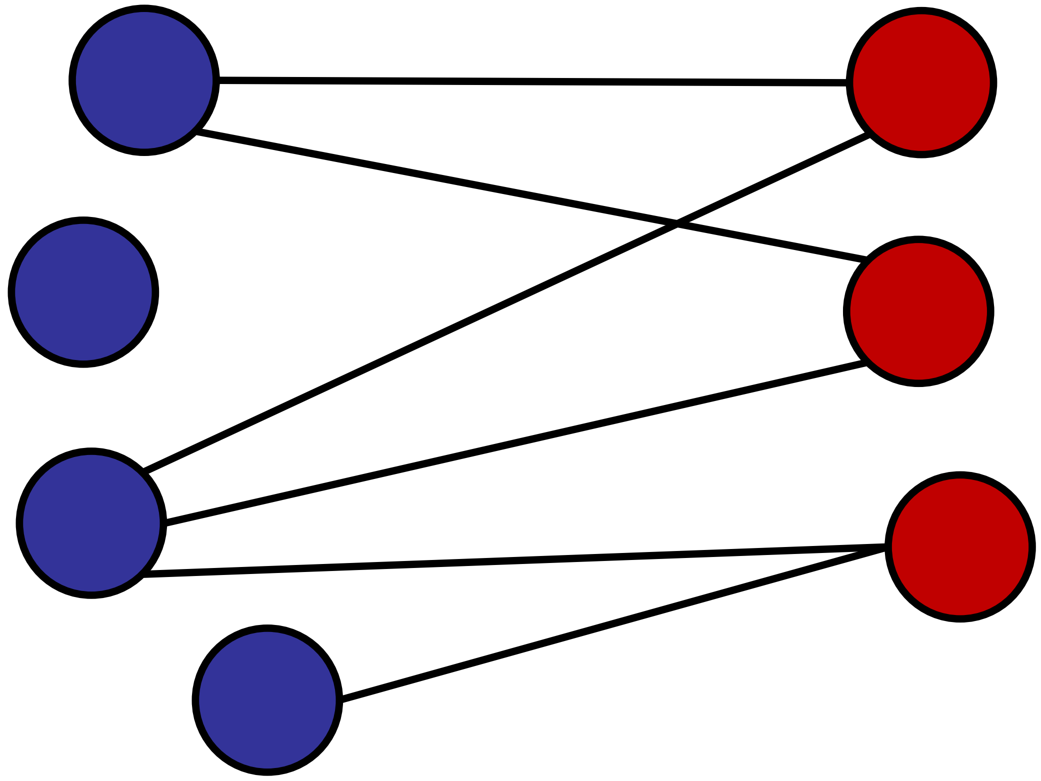
diameter = n/2-1

degree = 2



Special Graphs

Bipartite Graph



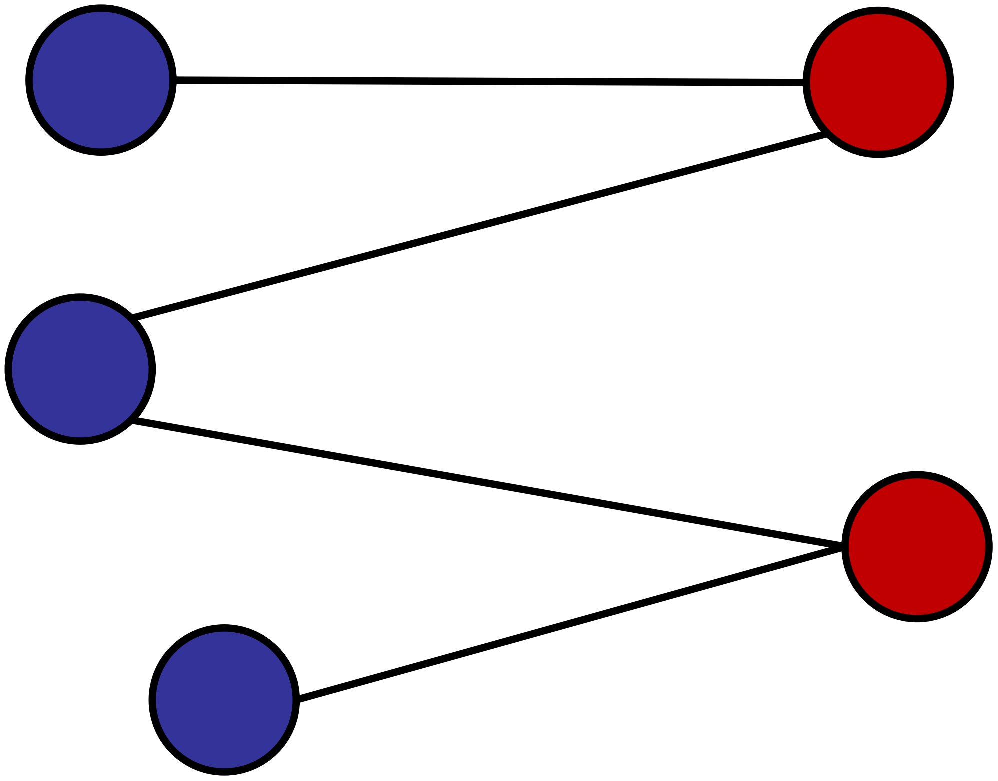
Nodes divided into two sets with no edges between nodes in the same set.

Max. diameter of n-node bipartite graph is:

1. 1
2. 2
3. n/2-1
4. n/2
5. [✓ n-1]
6. n

Special Graphs

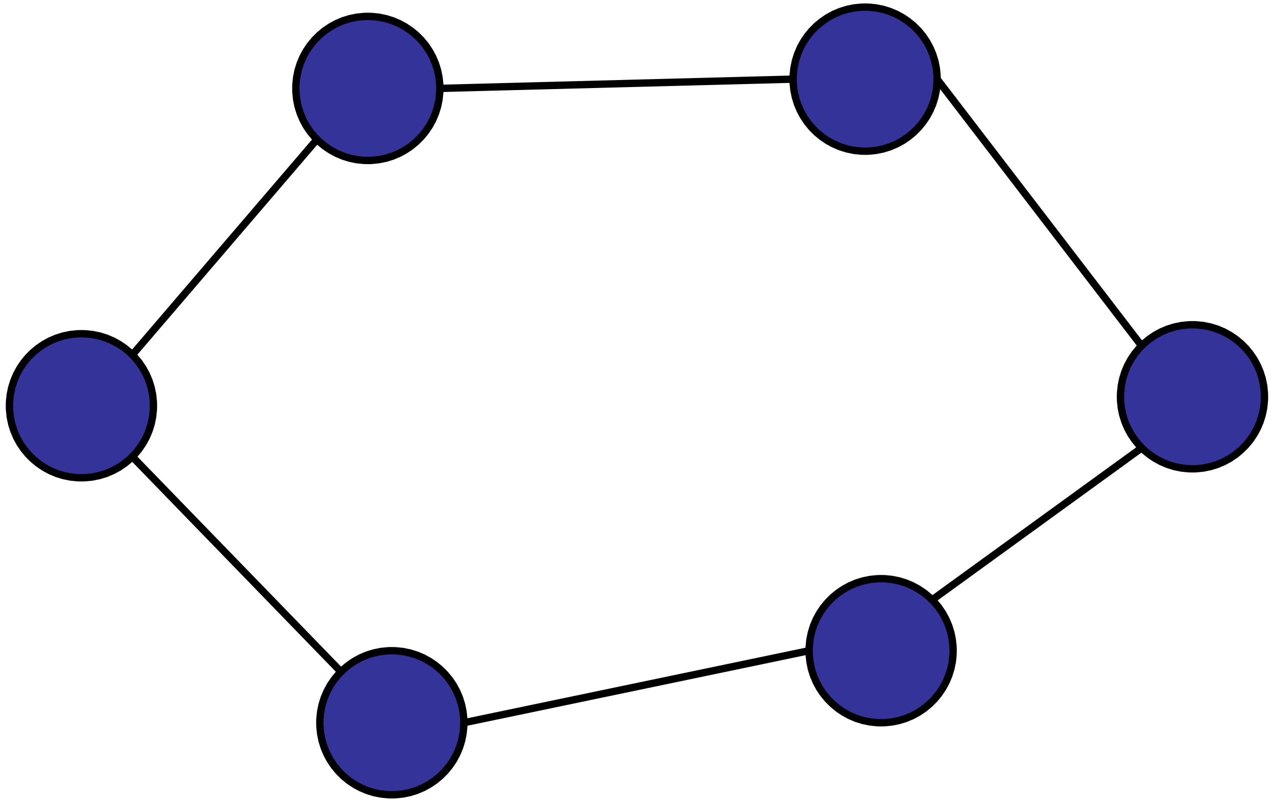
Bipartite Graph



Nodes divided into two sets with no edges between nodes in the same set.

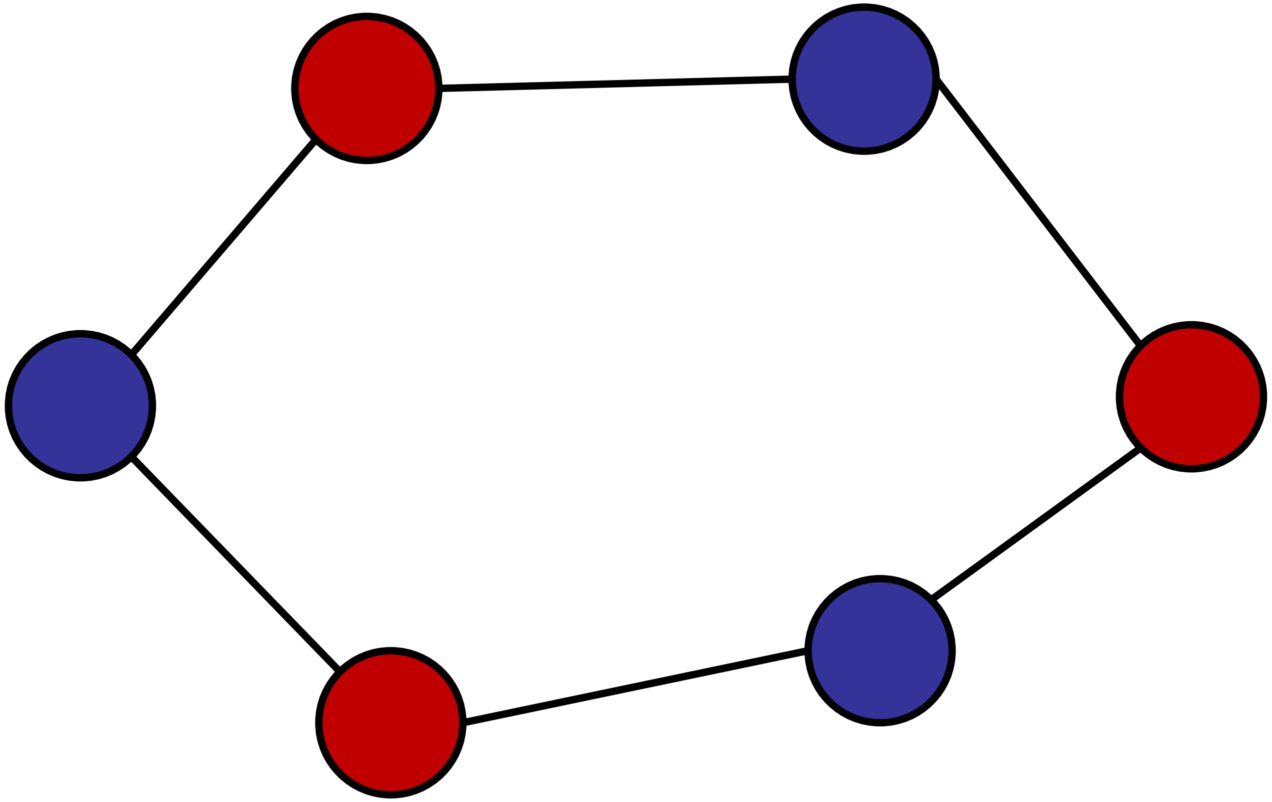
Is it bipartite?

1. Yes
2. No



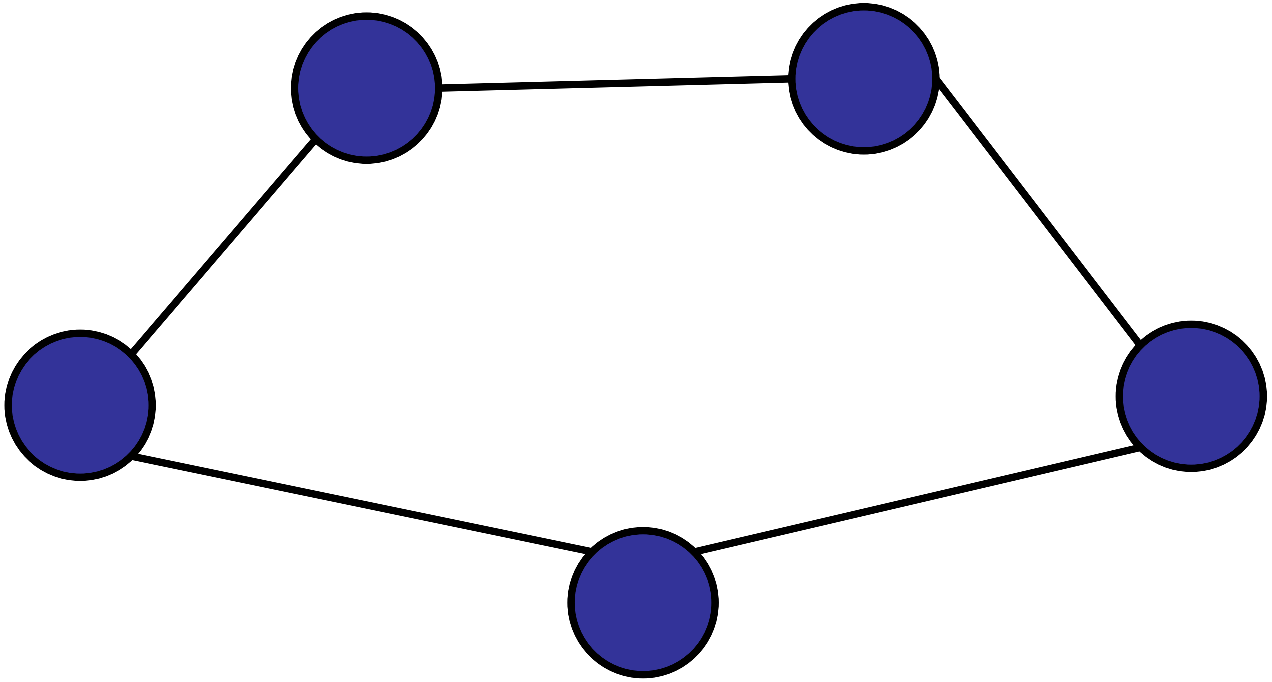
Is it bipartite?

[✓ 1. Yes]
2. No



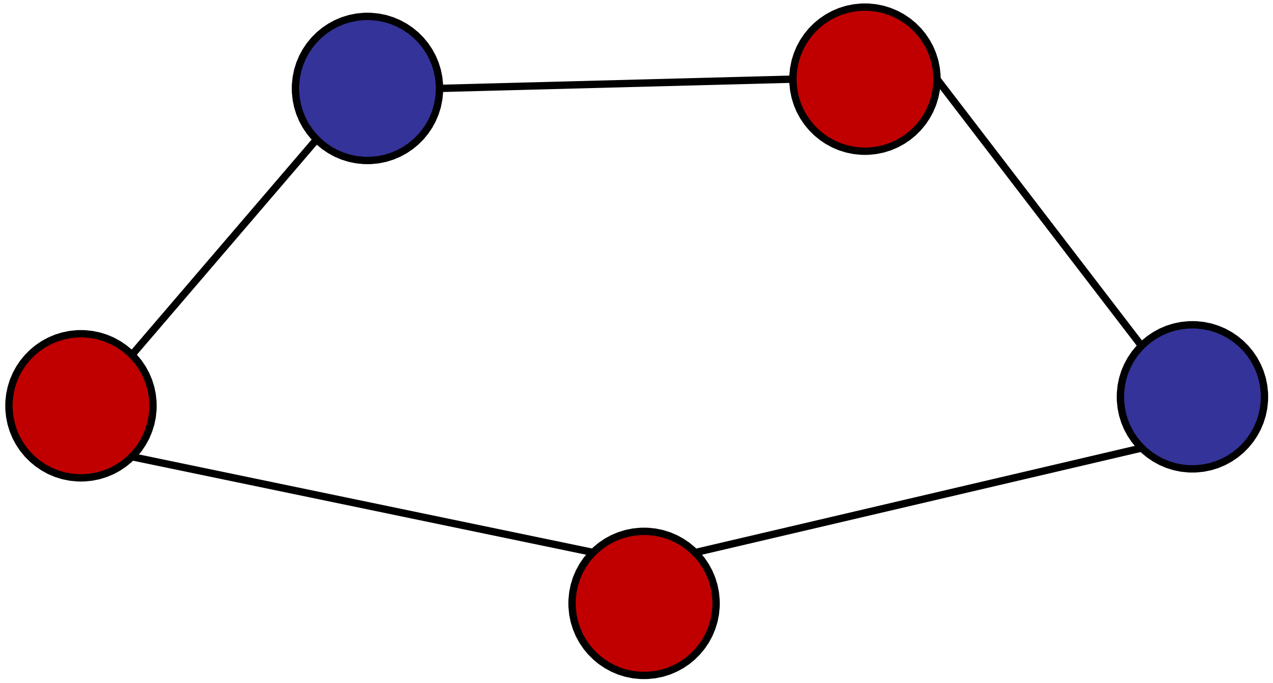
Is it bipartite?

1. Yes
2. No



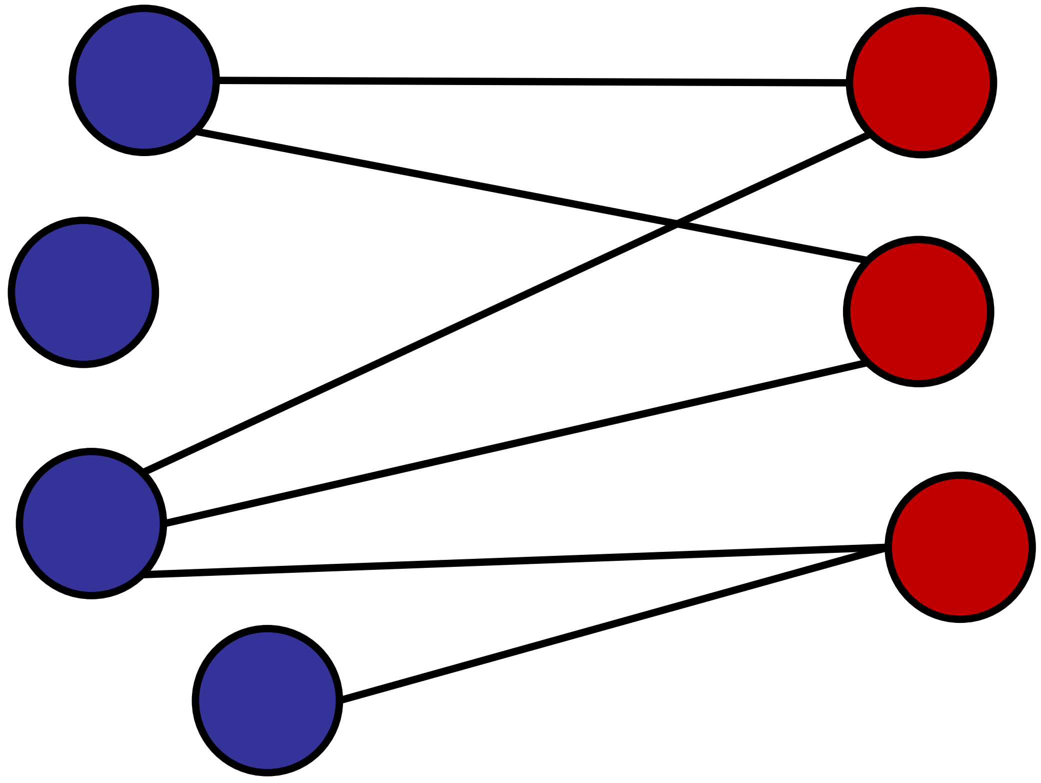
Is it bipartite?

1. Yes
2. [✓ No]



Special Graphs

Bipartite Graph



Nodes divided into two sets with no edges between nodes in the same set.

Roadmap

Today: Graph Basics

- What is a graph?
- Modeling problems as graphs.
- Graph representations (list vs. matrix)

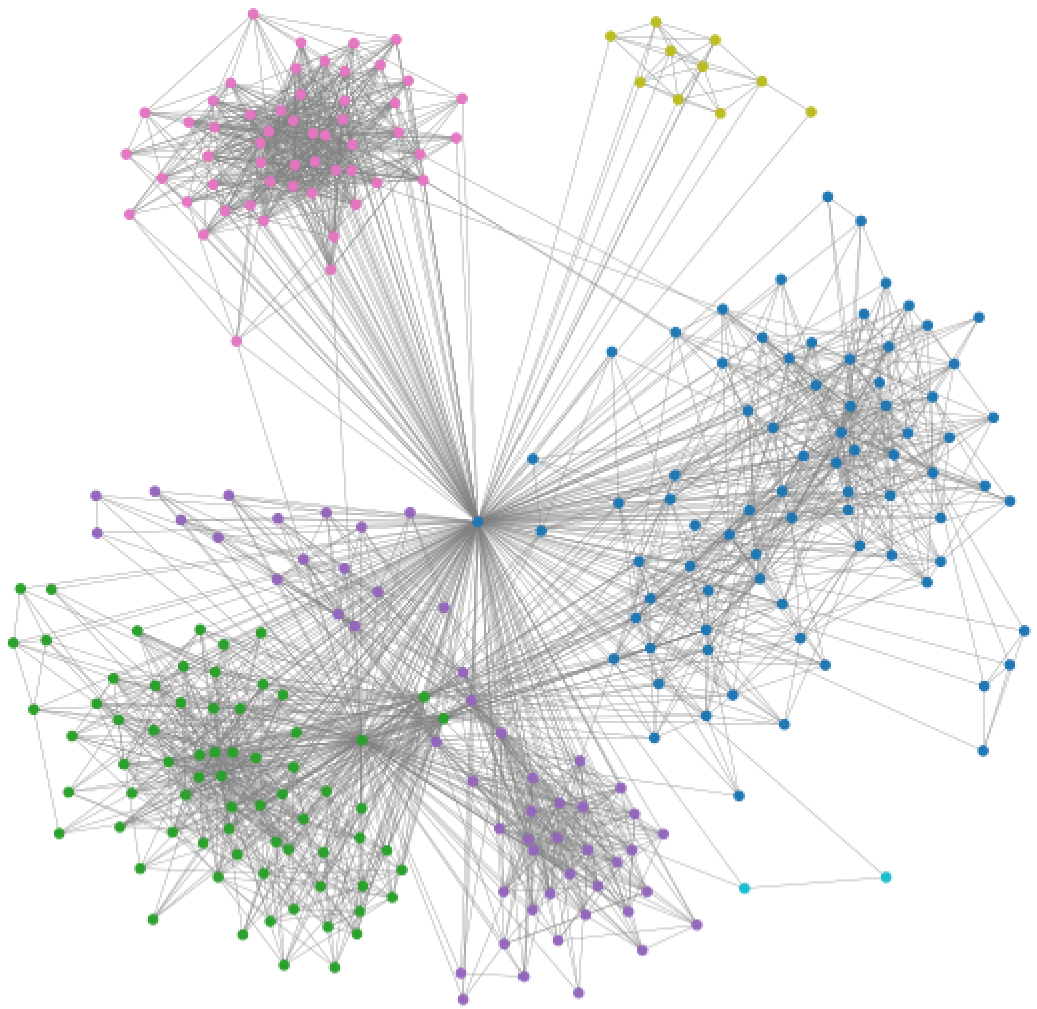
Where do we find graphs?

(How to model real problems as a graph!)

Where do we find graphs?

Social network:

- Nodes are people
- Edge = friendship



facebook

Source: https://www.databentobox.com/2019/07/28/facebook-friend-graph/

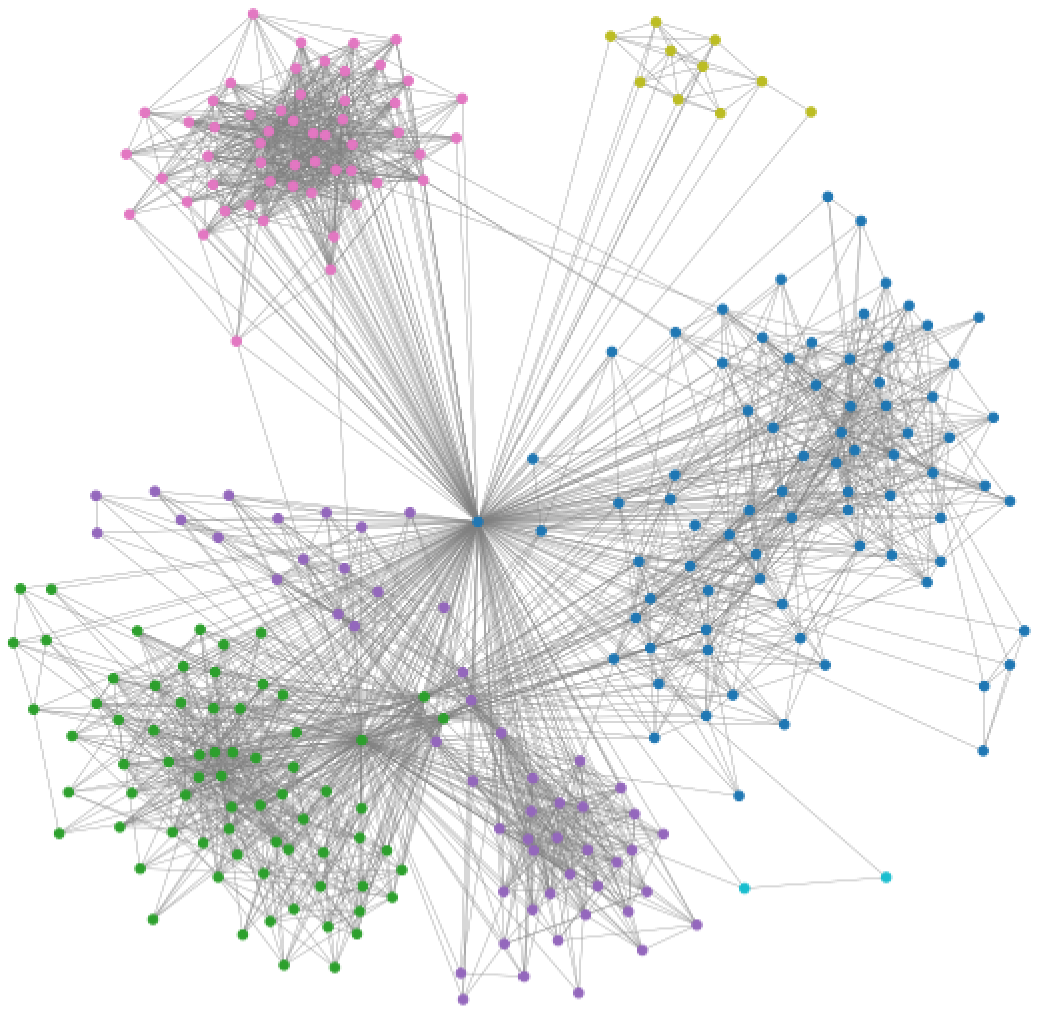
Where do we find graphs?

Social network:

- Nodes are people
- Edge = friendship

Questions:

- Connected?
- Diameter?
- Degree?



Source: https://www.databentobox.com/2019/07/28/facebook-friend-graph/

Where do we find graphs?

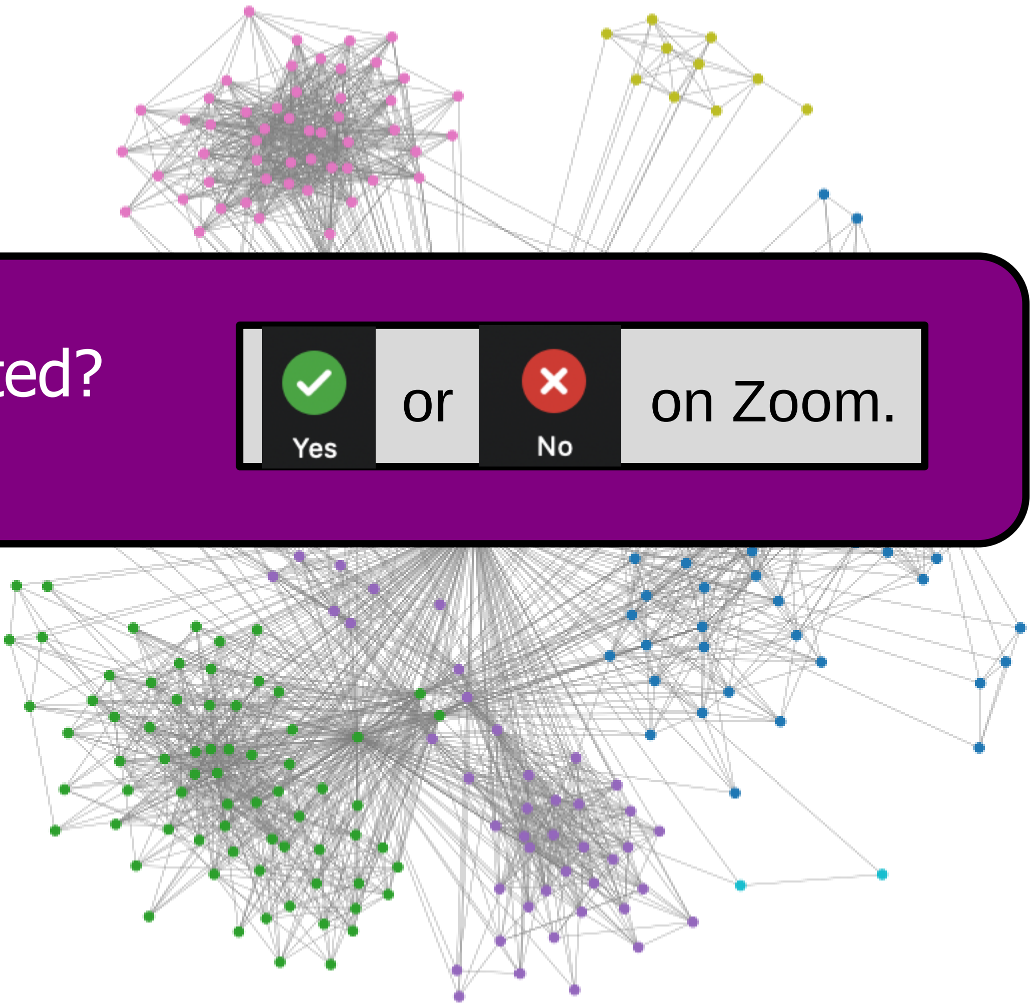
Social network:

- Nodes are people
- Edge =

Is it connected? Yes or No on Zoom.

Questions:

- Connected?
- Diameter?
- Degree?



Source: https://www.databentobox.com/2019/07/28/facebook-friend-graph/

Where do we find graphs?

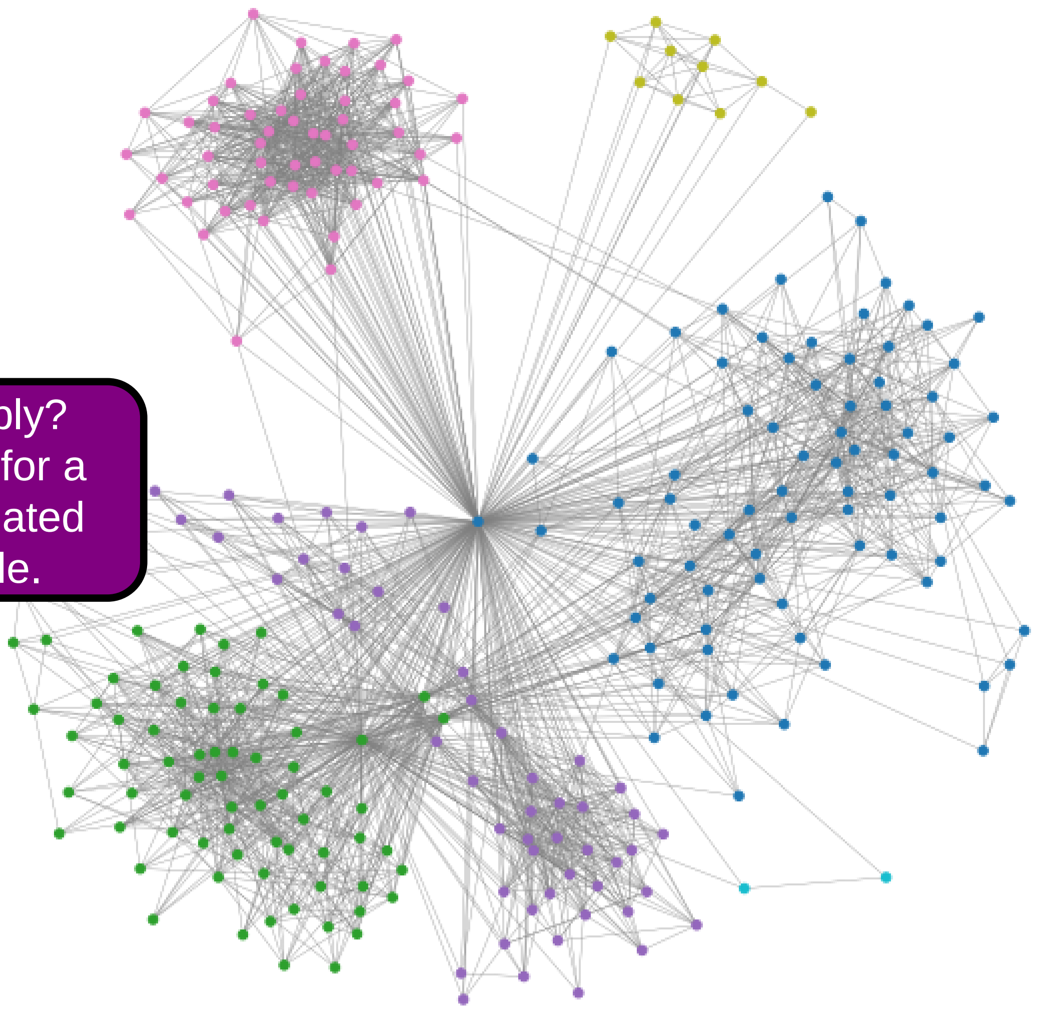
Social network:

- Nodes are people
- Edge = friendship

Questions:

- Connected?
- Diameter?
- Degree?

Probably? Except for a few isolated people.

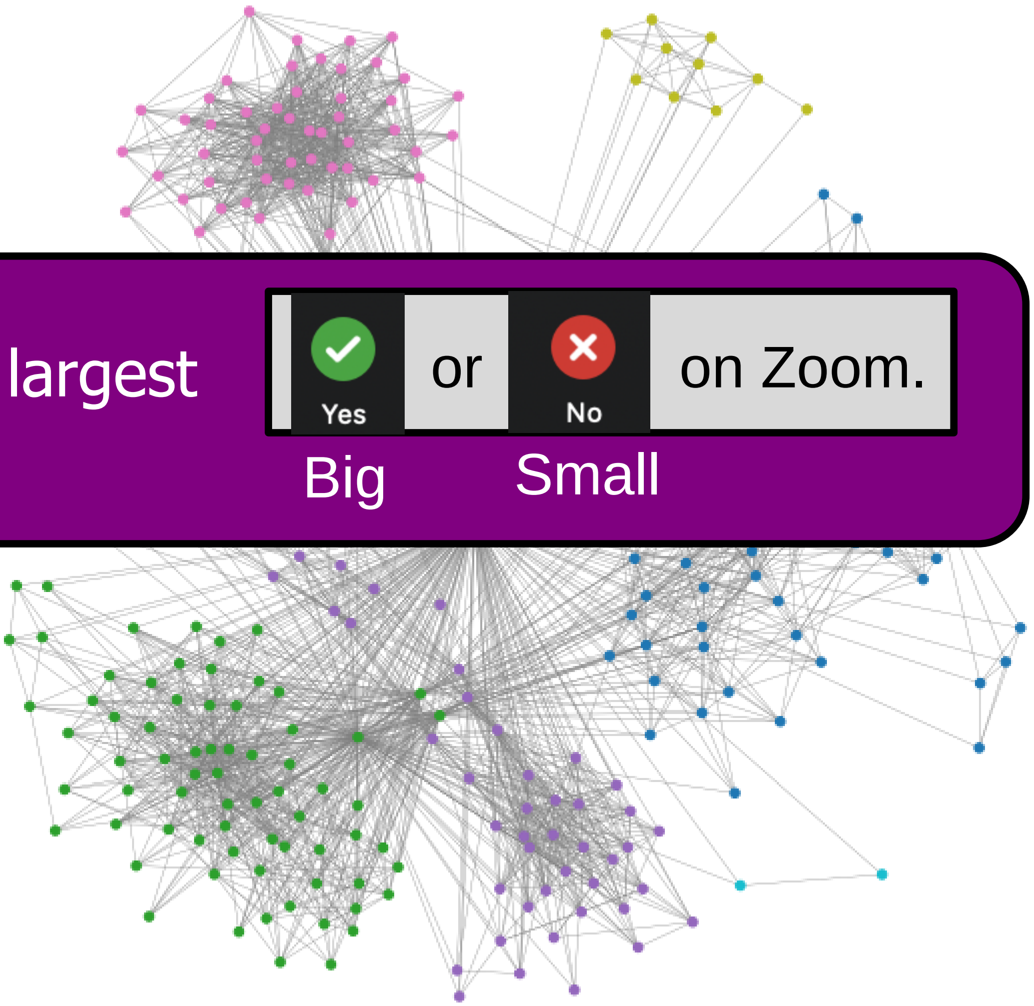


Source: https://www.databentobox.com/2019/07/28/facebook-friend-graph/

Where do we find graphs?

Social network:

- Nodes are people
- Edge =



Diameter of largest component?

Yes or No on Zoom.

Big Small

Questions:

- Connected?
- Diameter?
- Degree?

Source: https://www.databentobox.com/2019/07/28/facebook-friend-graph/

Where do we find graphs?

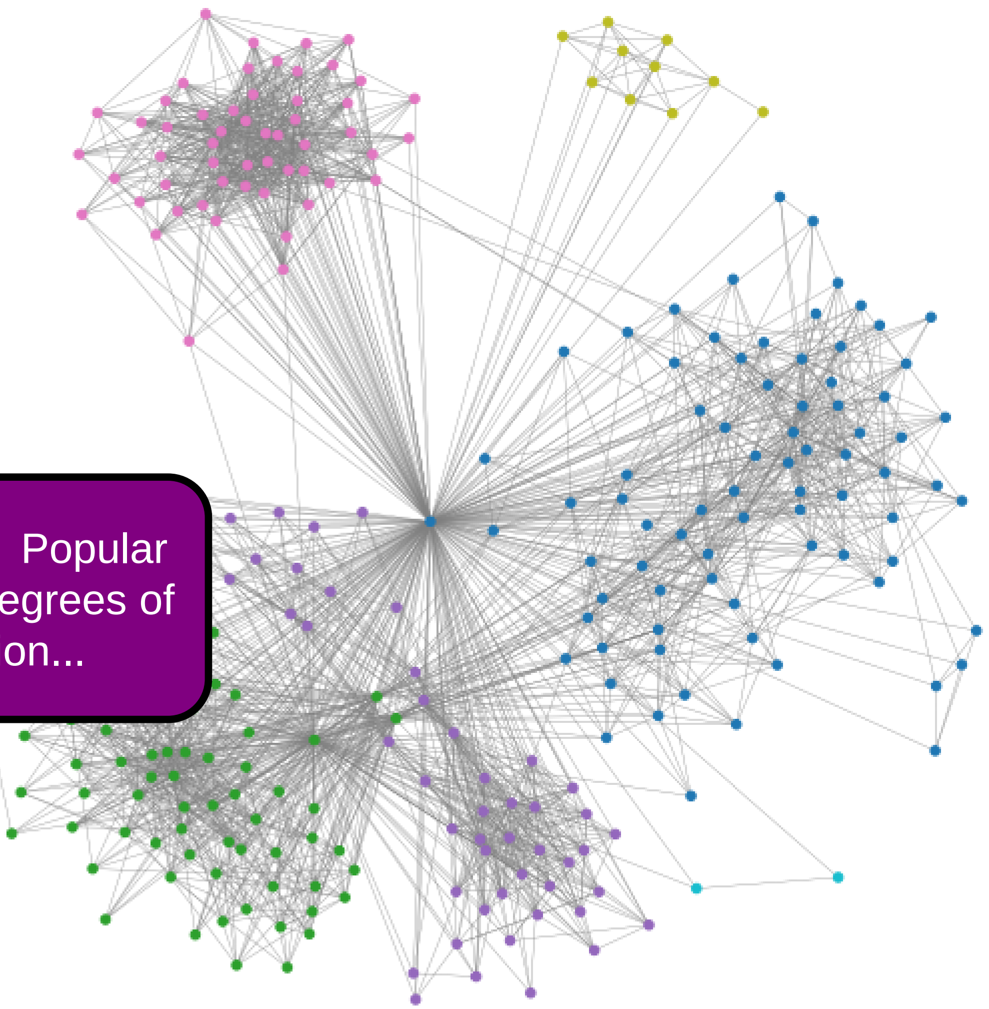
Social network:

- Nodes are people
- Edge = friendship

Questions:

- Connected?
- Diameter?
- Degree?

Fairly small! Popular saying: six degrees of separation...



Source: https://www.databentobox.com/2019/07/28/facebook-friend-graph/

Where do we find graphs?

Social network:

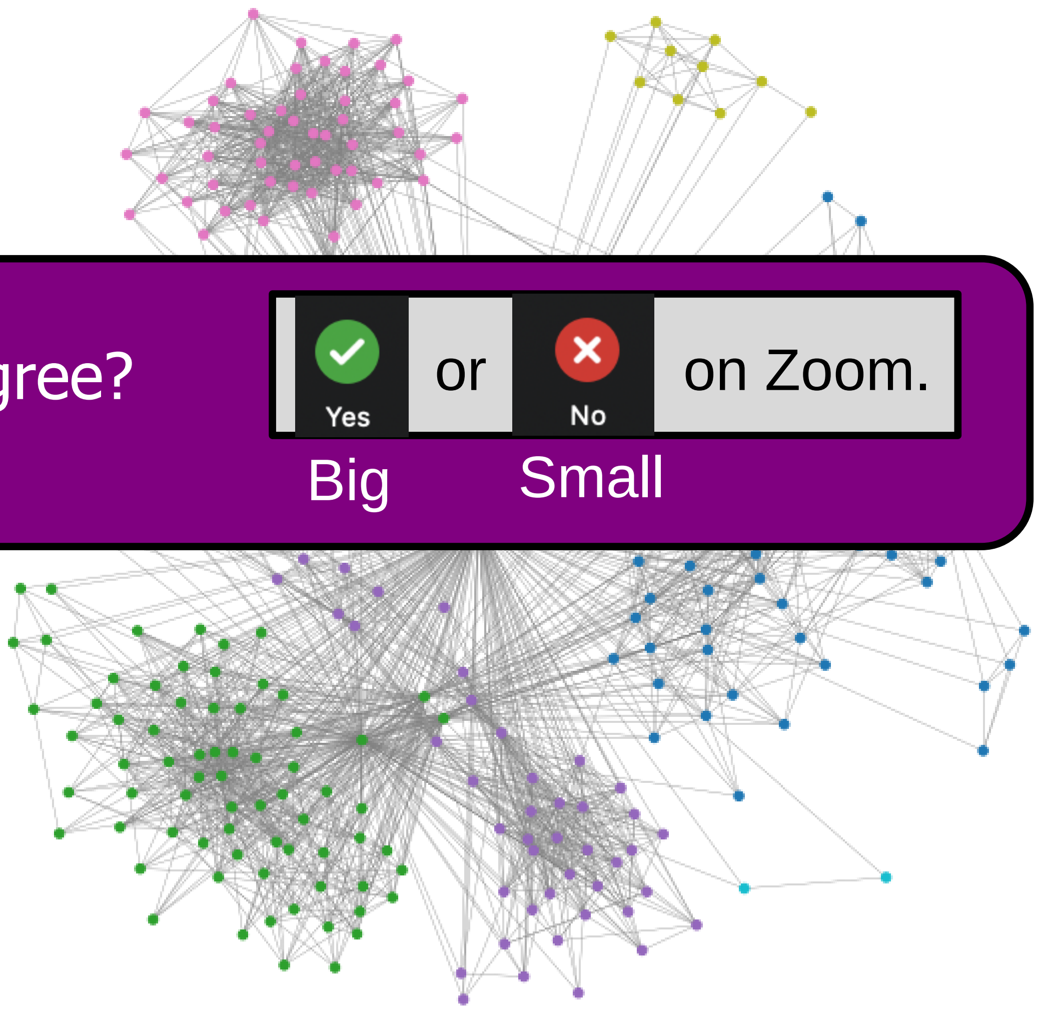
- Nodes are people
- Edge =

Average degree? ✓ Yes or ✗ No on Zoom.

Big Small

Questions:

- Connected?
- Diameter?
- Degree?

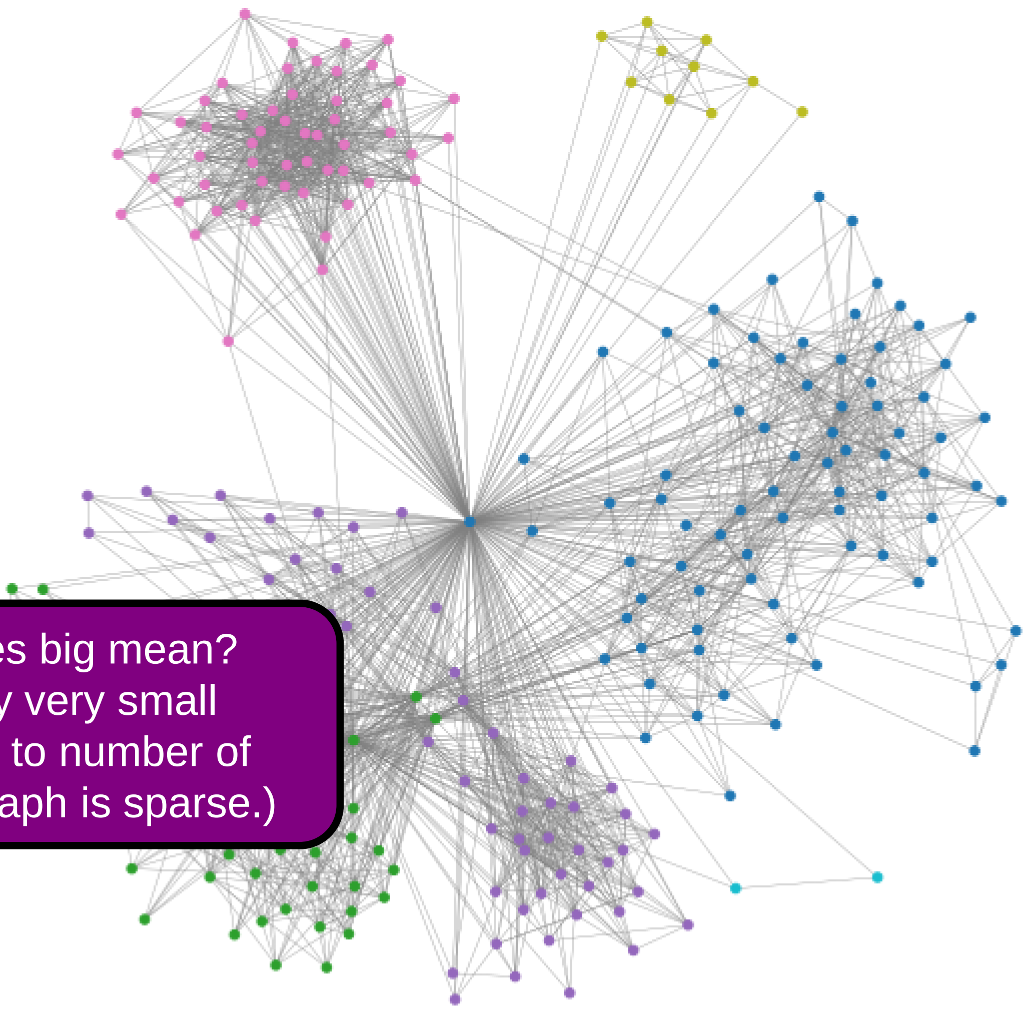


Source: https://www.databentobox.com/2019/07/28/facebook-friend-graph/

Where do we find graphs?

Social network:

- Nodes are people
- Edge = friendship



Questions:

- Connected?
- Diameter?
- Degree?

What does big mean? Probably very small compared to number of nodes. (Graph is sparse.)

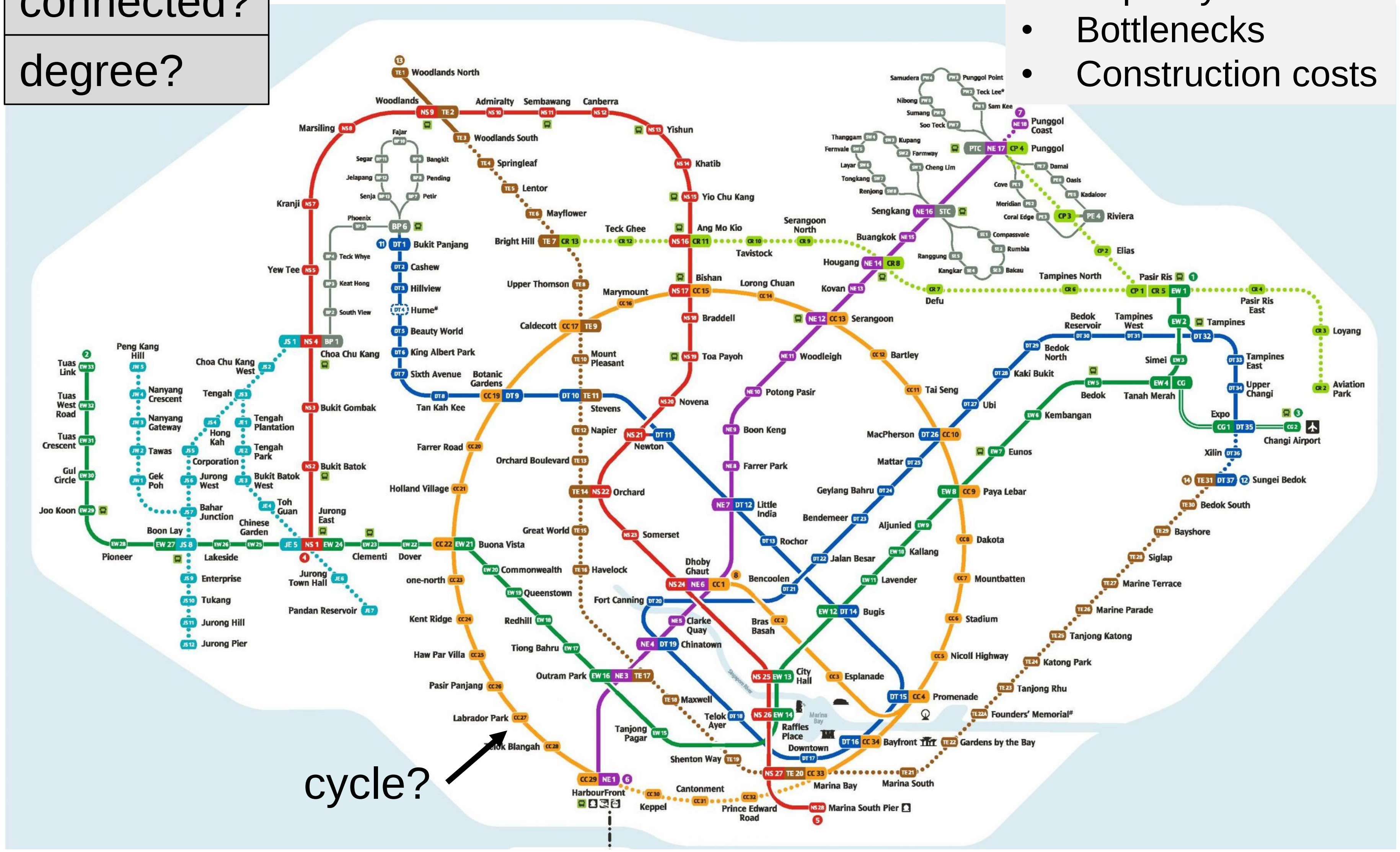
Source: https://www.databentobox.com/2019/07/28/facebook-friend-graph/

Transportation Network

connected?

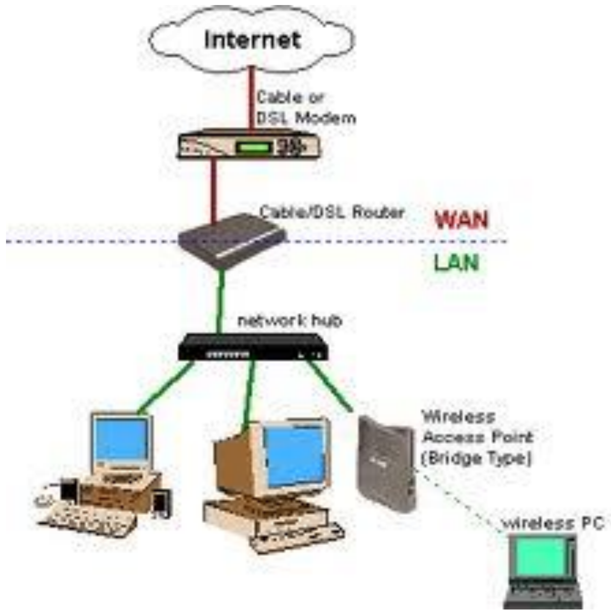
degree?

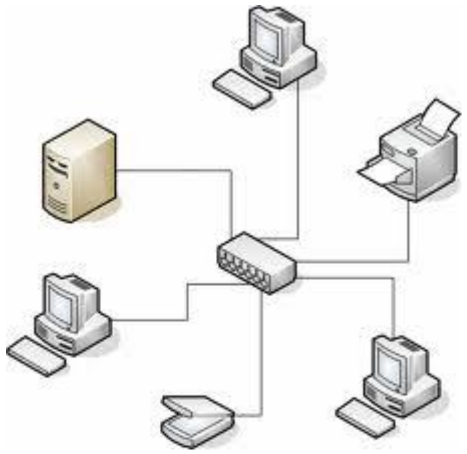
Questions:
- Shortest path
- Capacity
- Bottlenecks
- Construction costs



cycle?

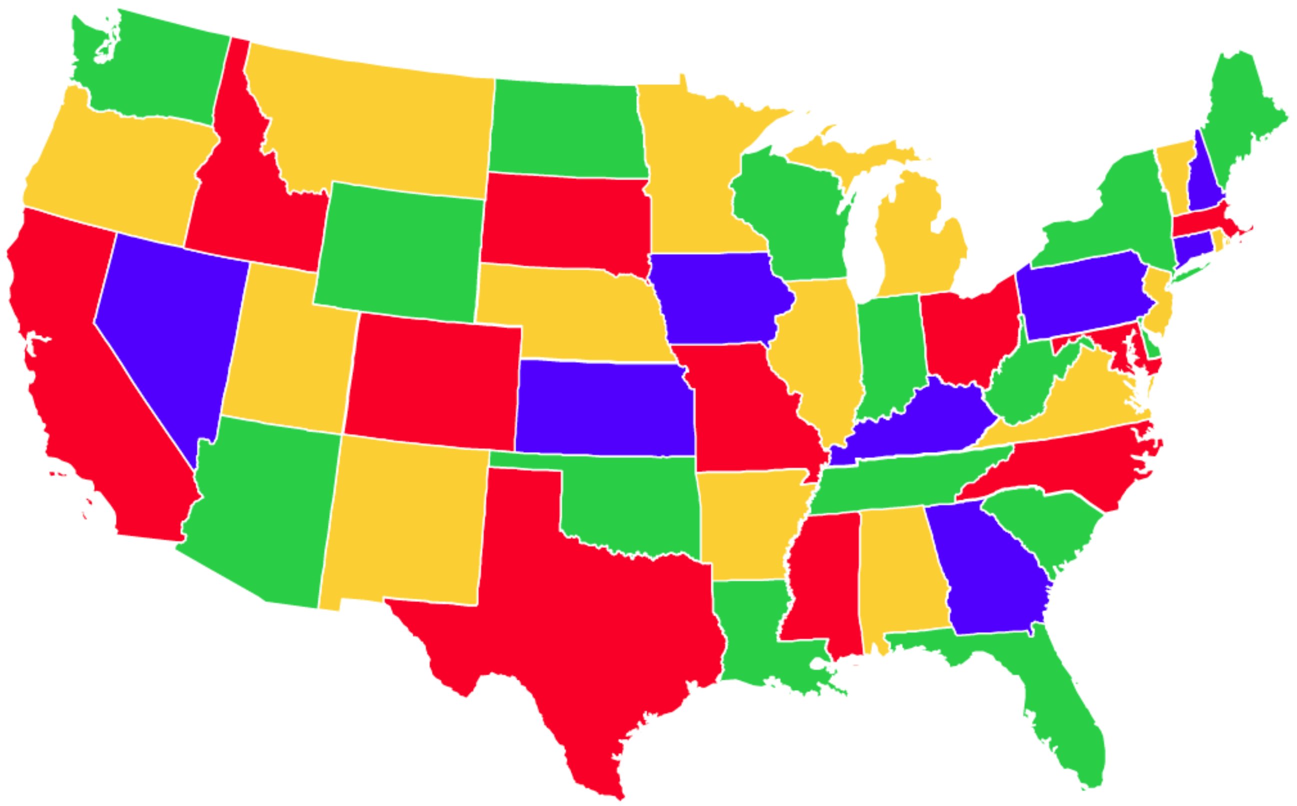
Internet / Computer Networks





Communication Network

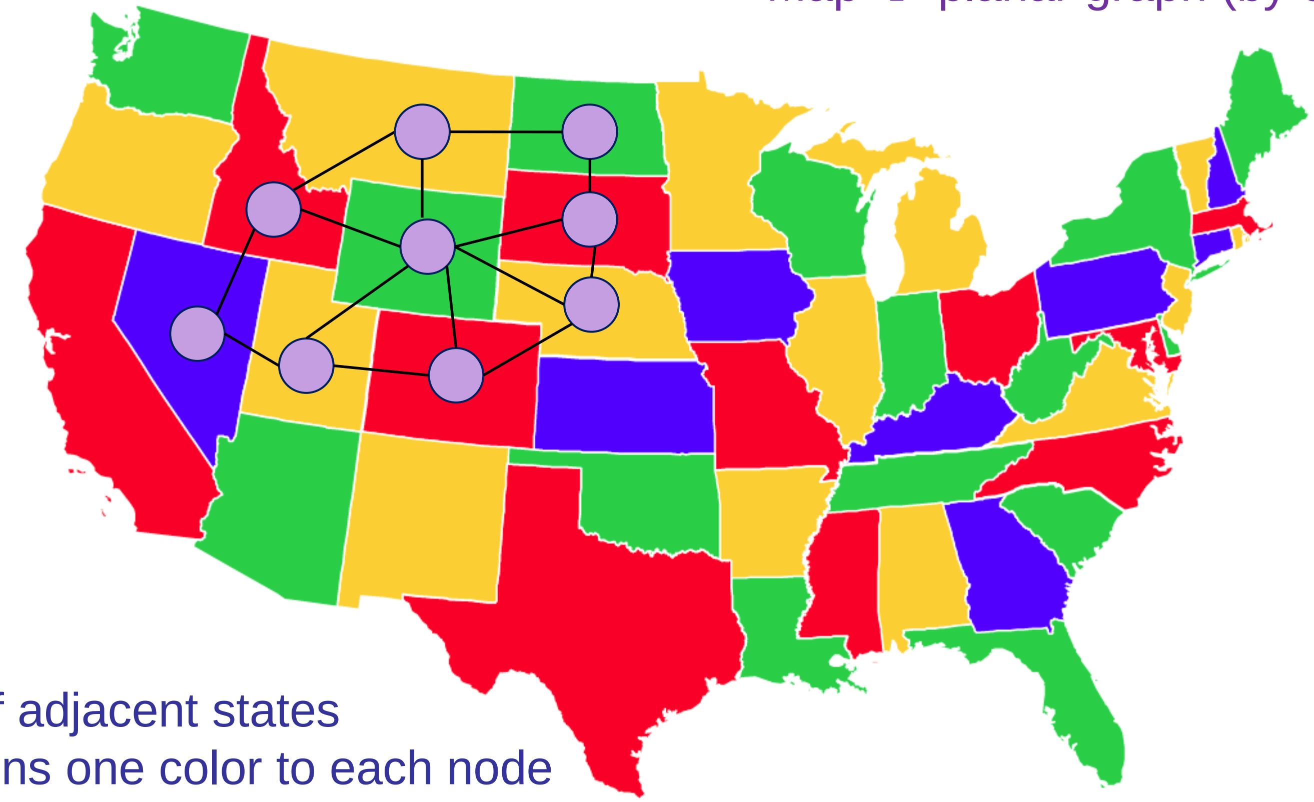
Optimization: 4-Coloring



Can you color a map using only 4-colors so that no two adjacent countries/states have the same color?

4-Coloring Theorem:

Map ➔ planar graph (by construction)

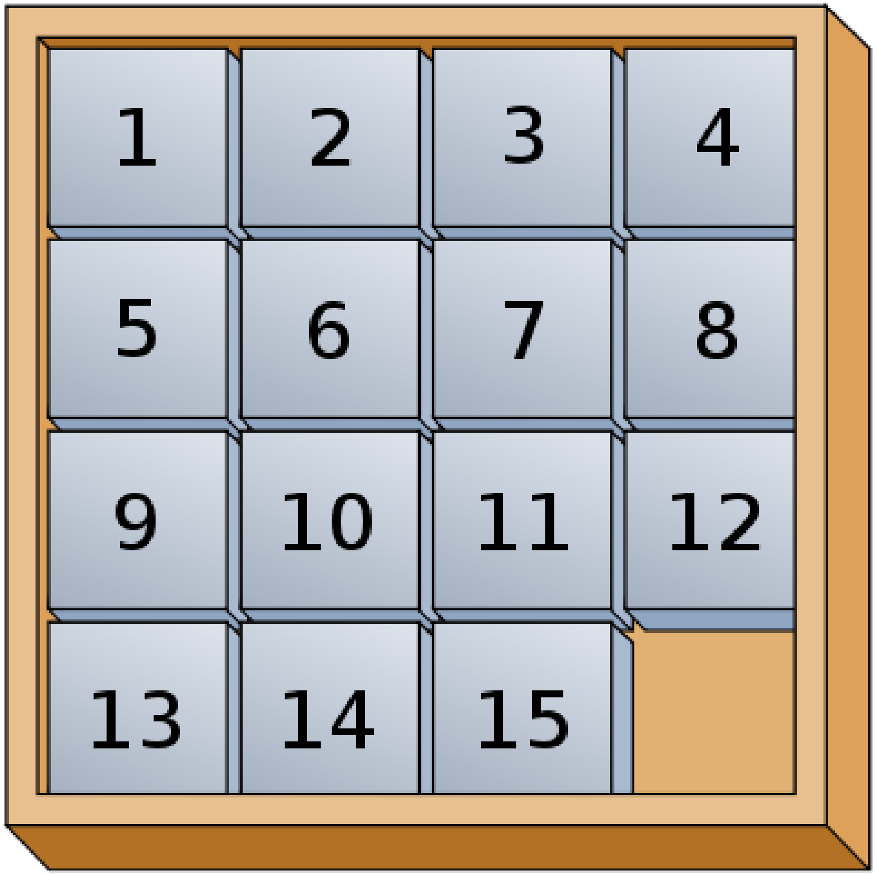


Nodes: states
Edges: pairs of adjacent states
Coloring: assigns one color to each node

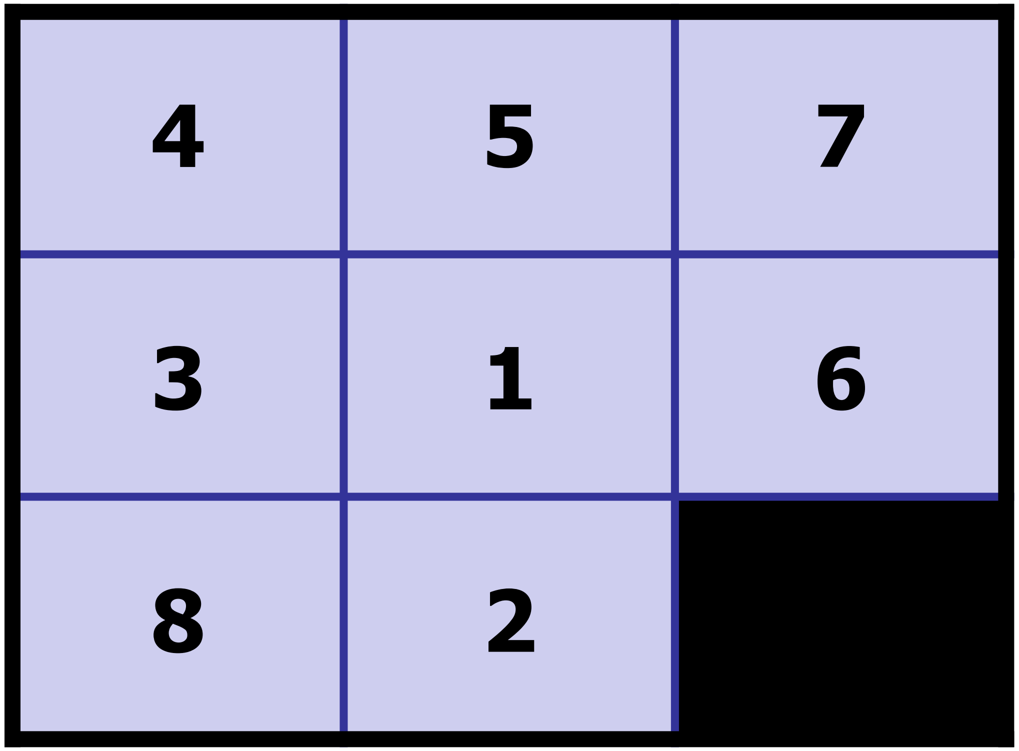
For any planar graph, you can color it using only 4-colors
so that no two adjacent countries/states have the same color?

Can be drawn on a 2d-plane with no crossing edges.

Sliding Puzzle



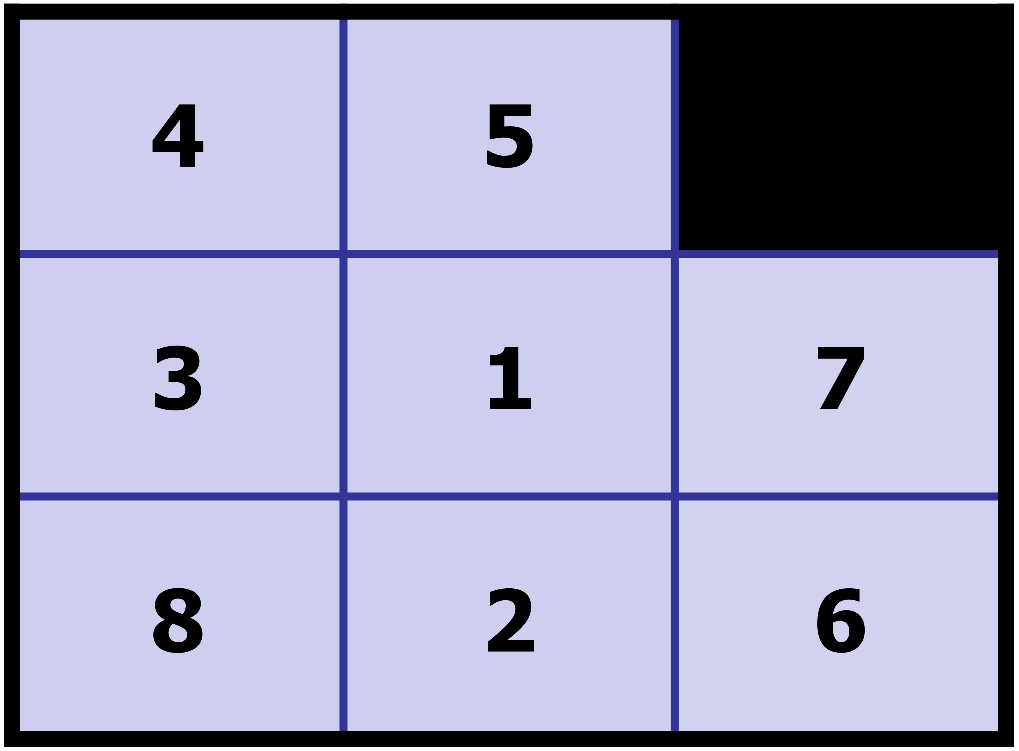
Sliding Puzzle



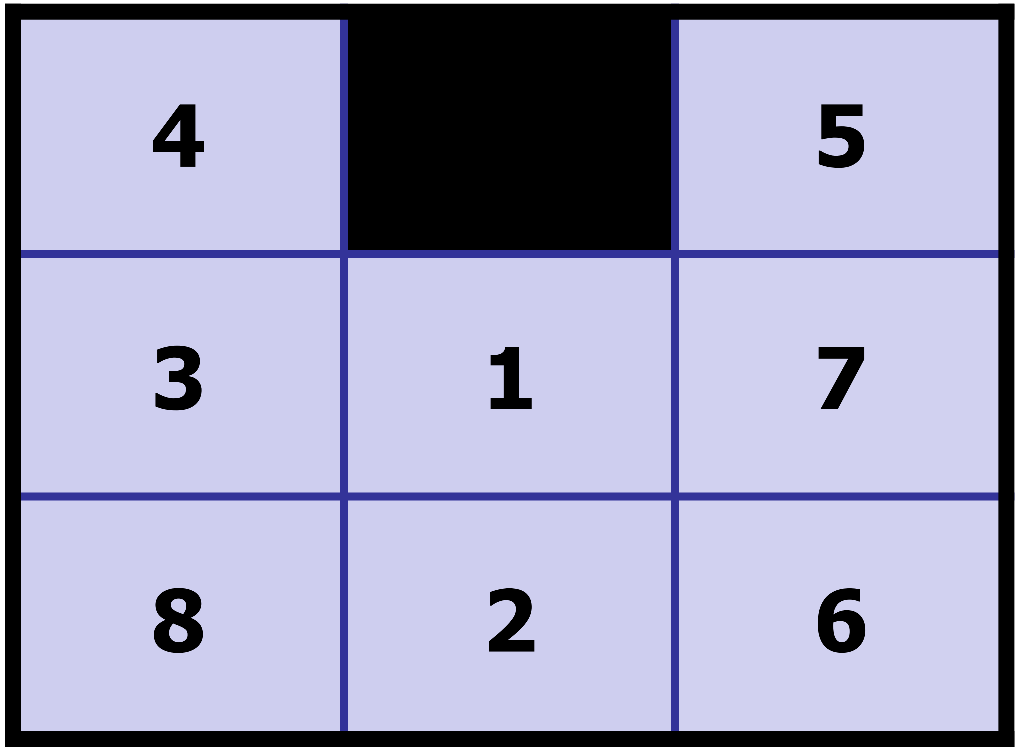
Sliding Puzzle

4	5	7
3	1	■
8	2	6

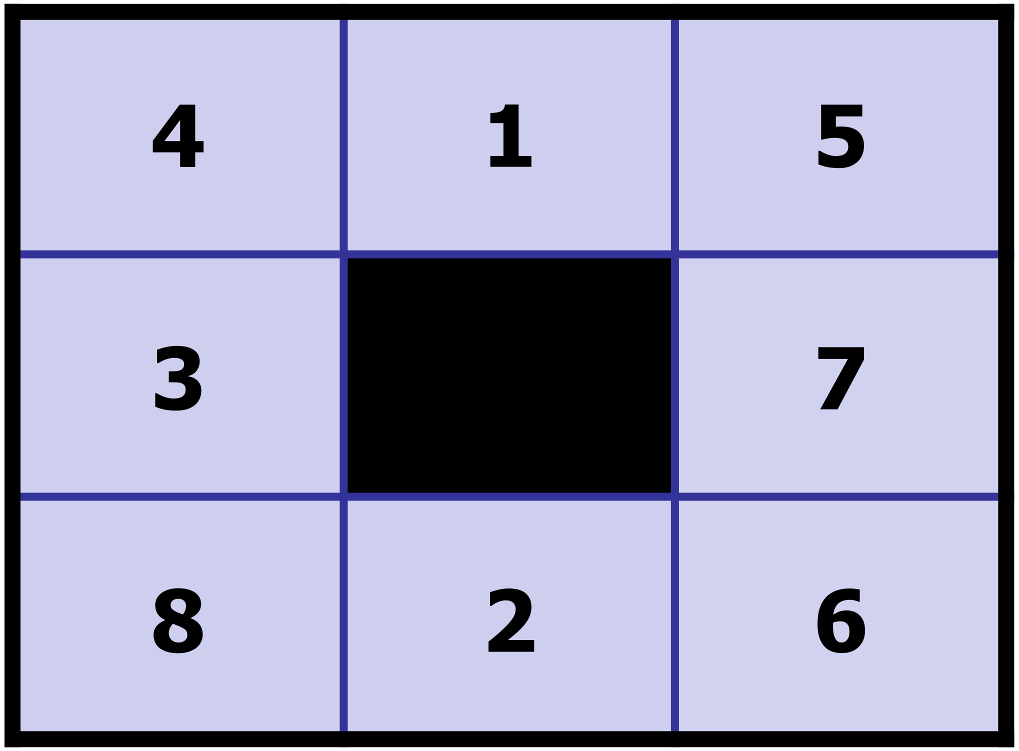
Sliding Puzzle



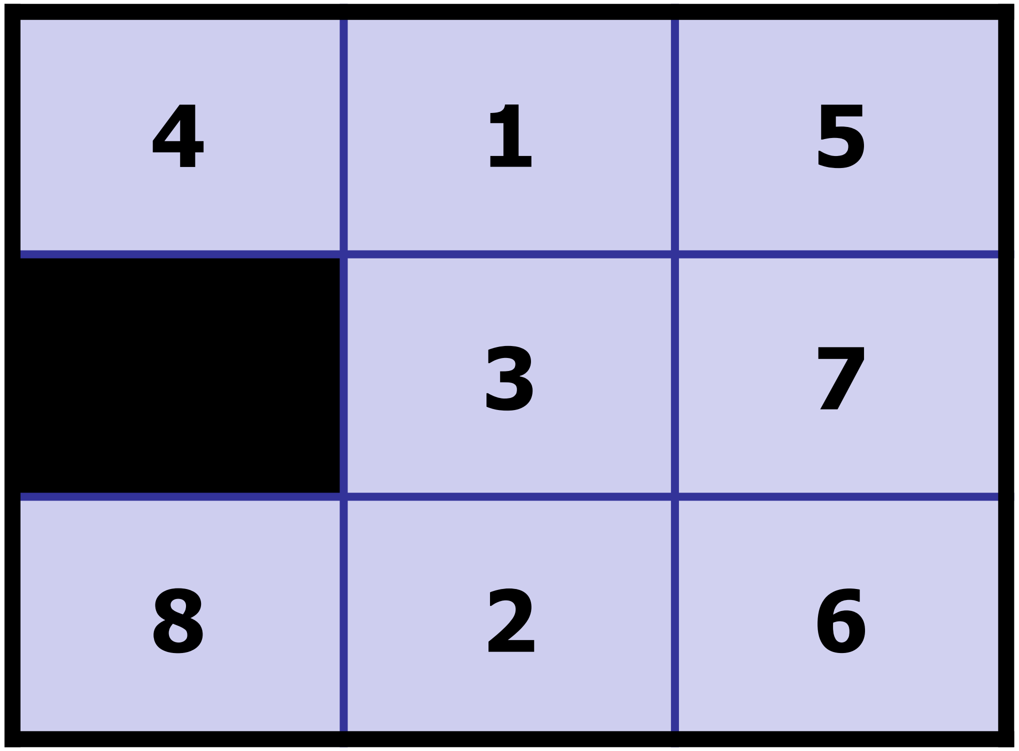
Sliding Puzzle



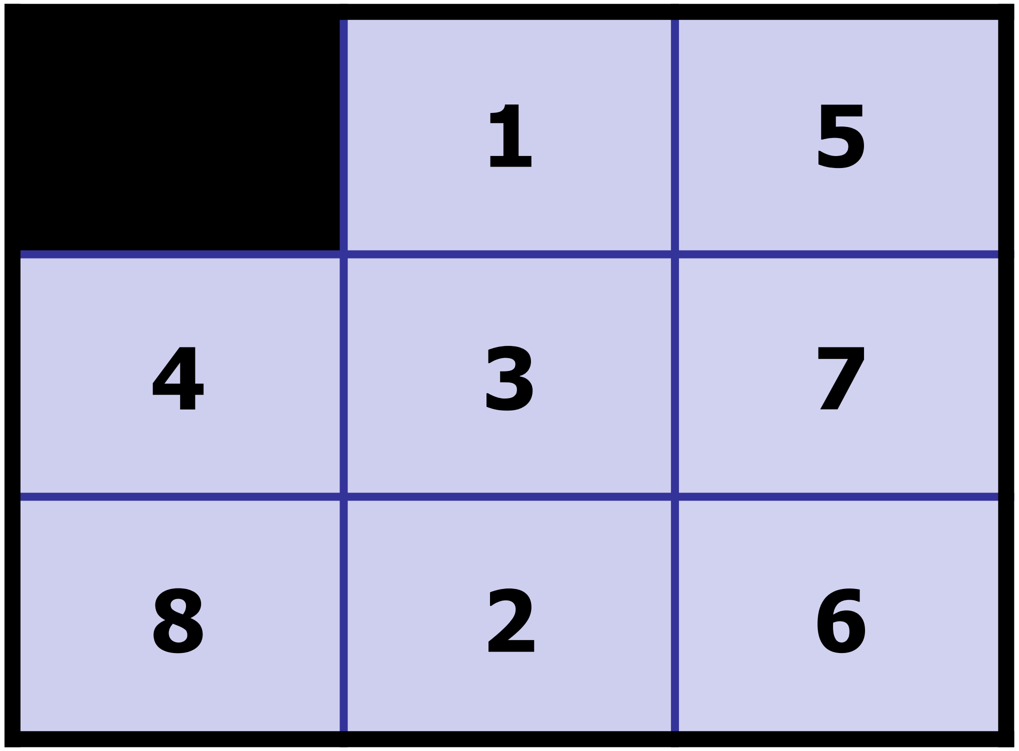
Sliding Puzzle



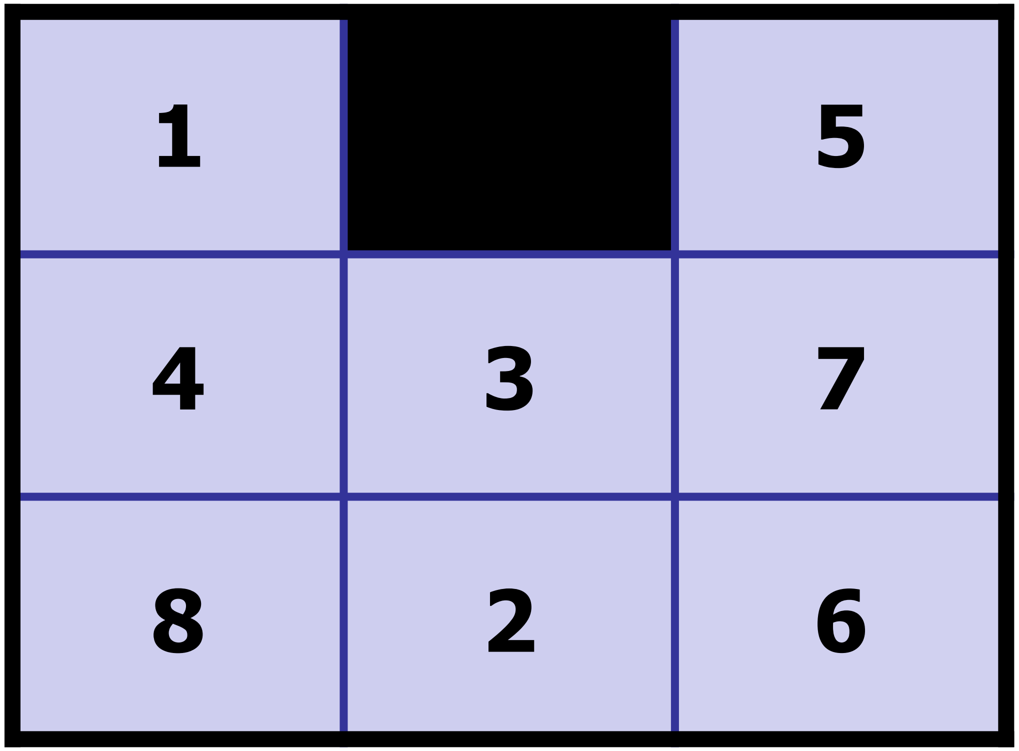
Sliding Puzzle



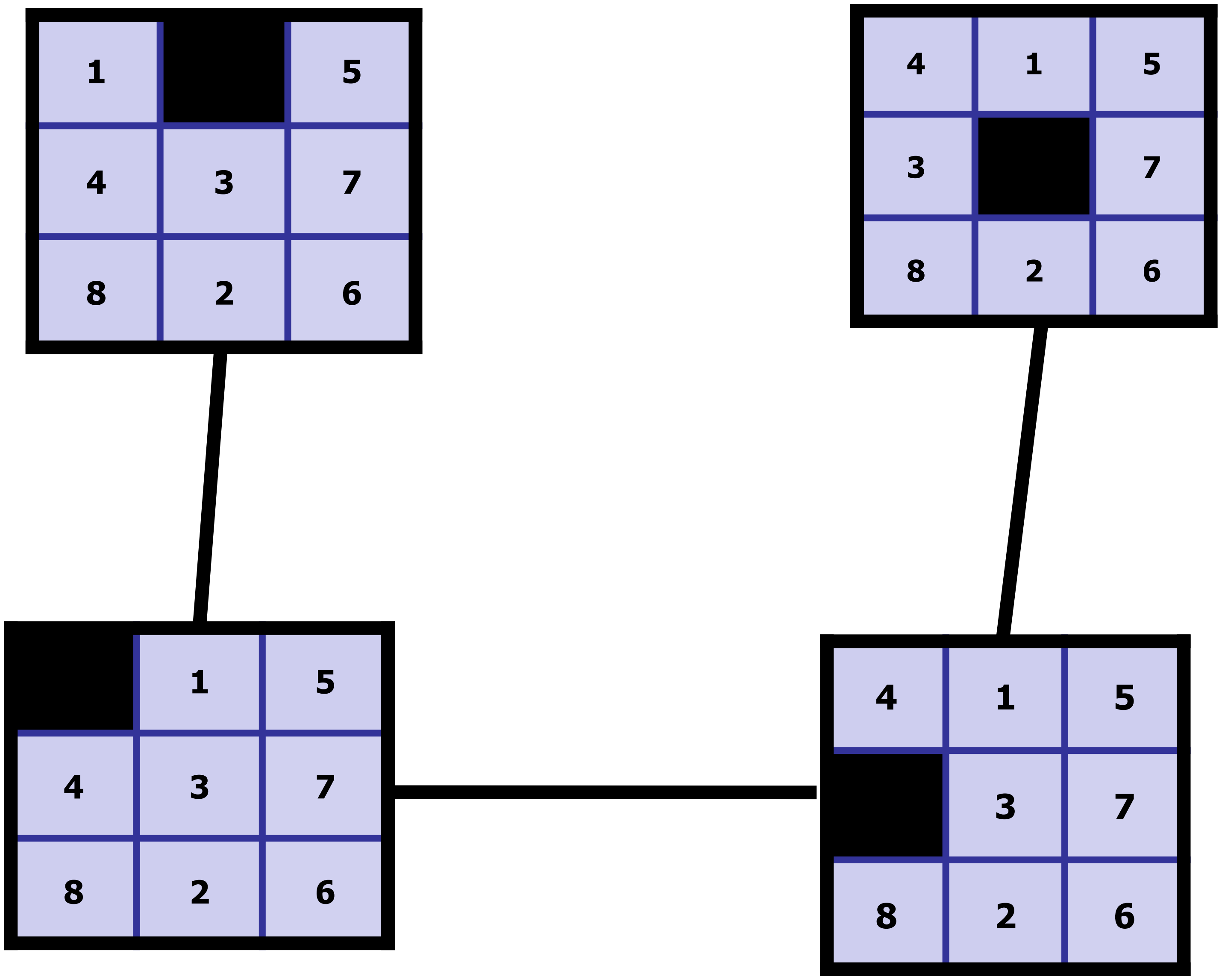
Sliding Puzzle



Sliding Puzzle



Sliding Puzzle is a Graph



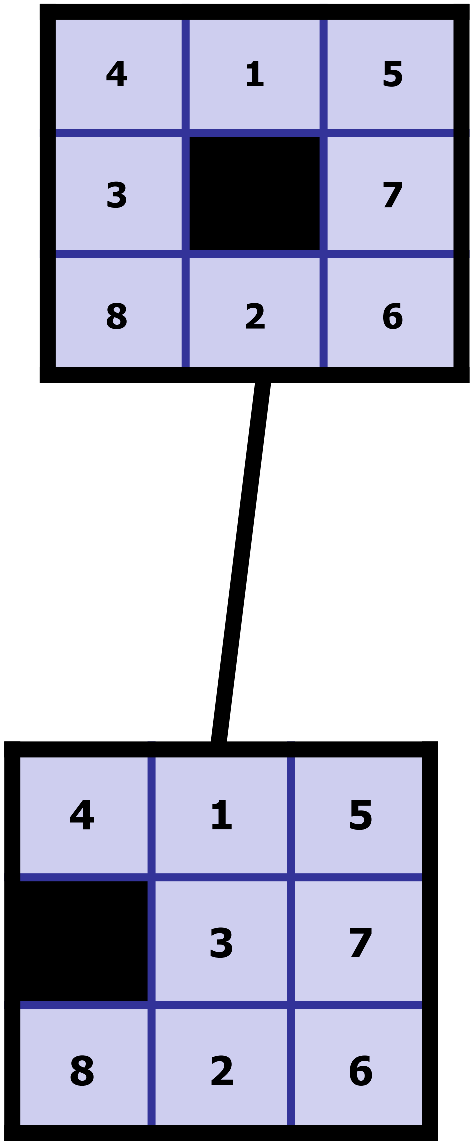
Sliding Puzzle

Nodes:

- State of the puzzle
- Permutation of nine tiles

Edges:

- Two states are edges if they differ by only one move.



What is the maximum degree of the Sliding Puzzle graph?

1. 1
2. 2
3. 3
4. [✓ 4]
5. n/2
6. n
7. n!

ARCHIPELAGO
is open

Sliding Puzzle

Nodes:

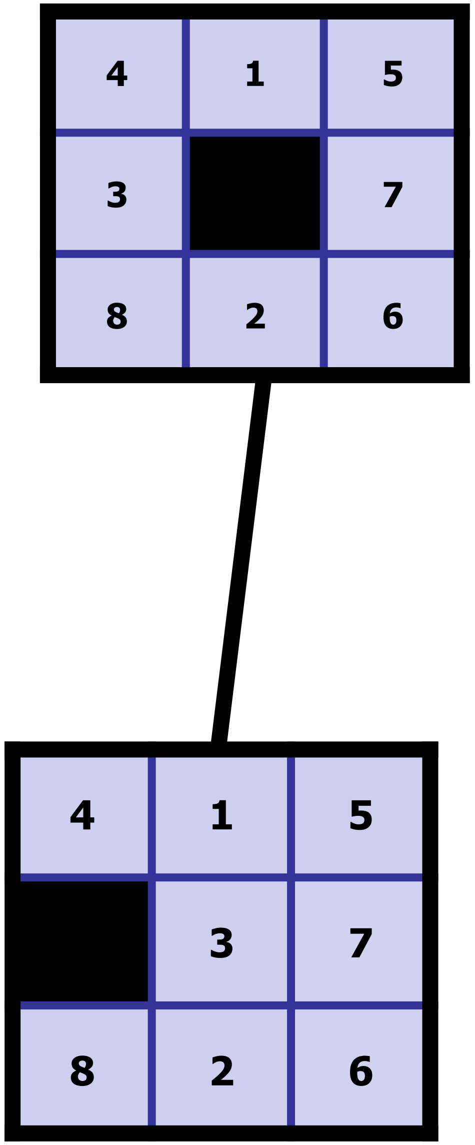
- State of the puzzle
- Permutation of nine tiles

Edges:

- Two states are edges if they differ by only one move.

Nodes = 9! = 362,880

Edges < 4*9! < 1,451,520

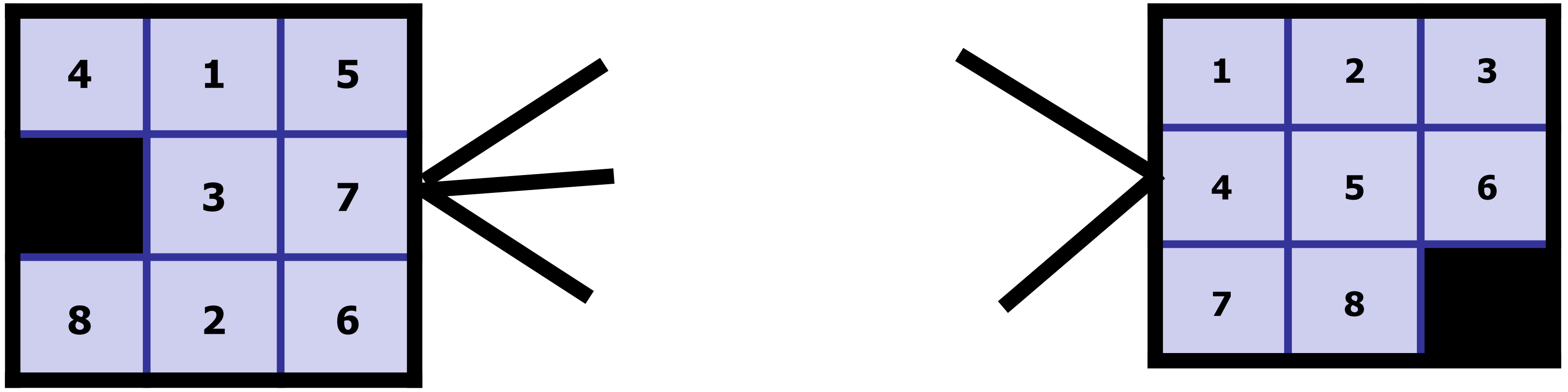


Sliding Puzzle

Number of moves to solve the puzzle?

Initial, scrambled state:

Final, unscrambled state:

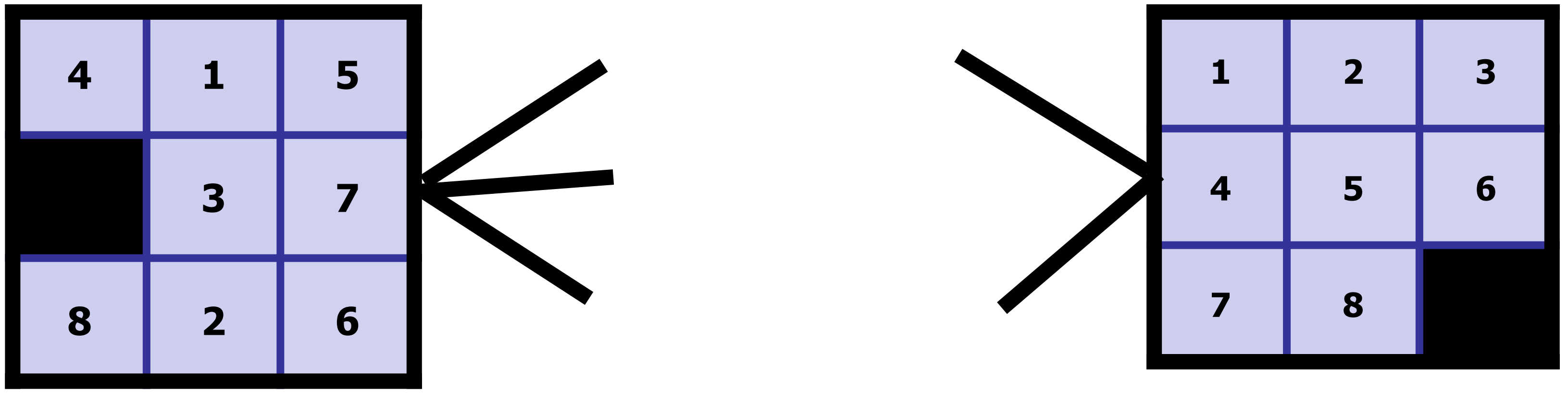


Sliding Puzzle

Number of moves <= Diameter

Initial, scrambled state:

Final, unscrambled state:



2 x 2 x 2 Rubik’s Cube

2 x 2 x 2 Rubik's Cube

Record solve time: 0.49 seconds

2 x 2 x 2 Rubik's Cube

Configuration Graph

- Vertex for each possible state
- Edge for each basic move
  - 90 degree turn
  - 180 degree turn

Puzzle: given initial state, find a path to the solved state.

2 x 2 x 2 Rubik’s Cube

How many vertices?

$$8! \cdot 3^8 = 264{,}539{,}520$$

# cubelets

Each cubelet is
in one of 8 positions.

Each of the 8 cubelets
can be in one of three
orientations

2 x 2 x 2 Rubik's Cube

How many vertices?

$$7! \cdot 3^7 = 11{,}022{,}480$$

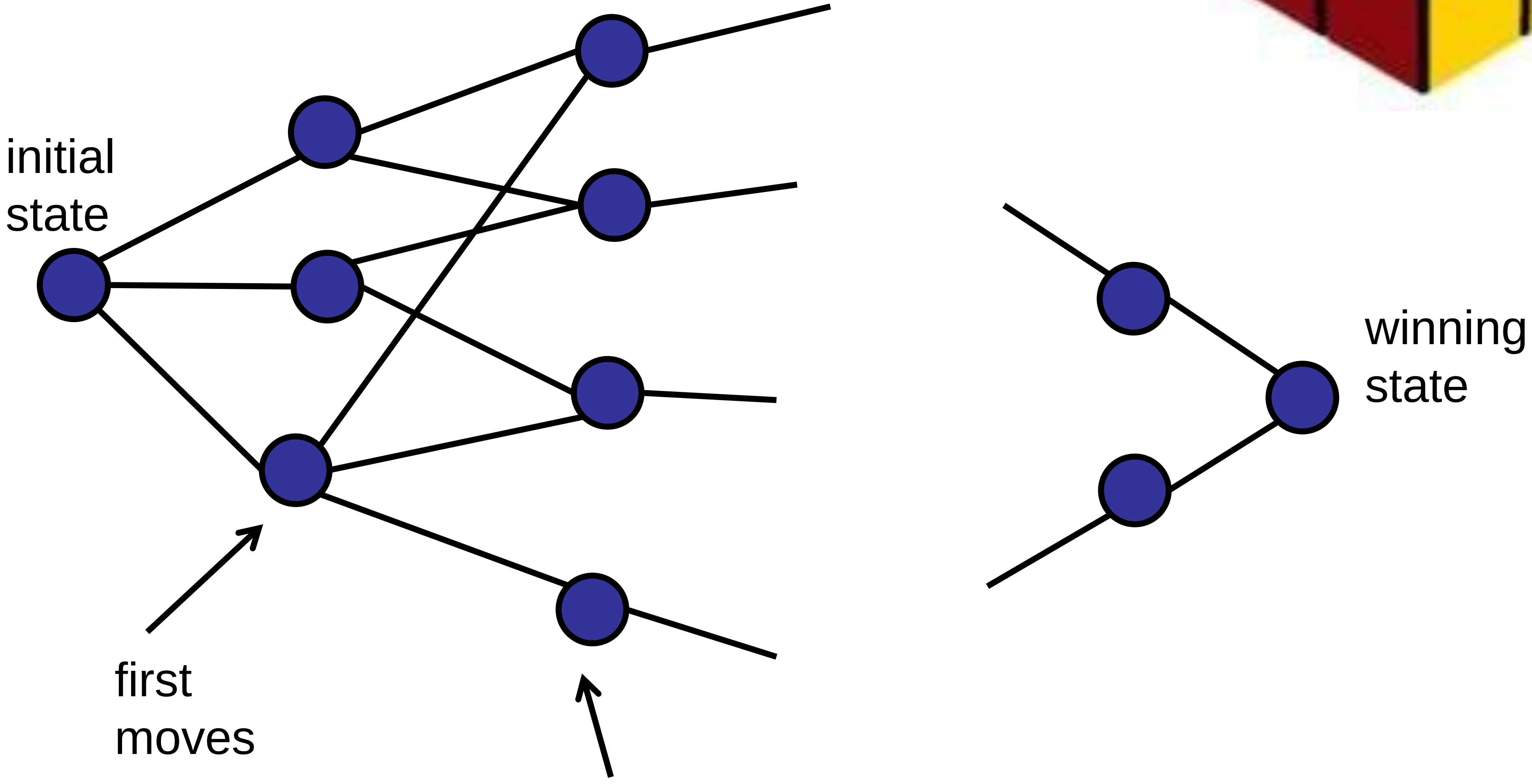
Symmetry:

Fix one cubelet.

Each of the 8 cubelets can be in one of three orientations

2 x 2 x 2 Rubik's Cube

Geography of Rubik's configurations:



reachable in two moves, but not one

Reachable configurations

Distance	90 deg. turns	90/180 deg. turns
0	1	1
1	6	9
2	27	54
3	120	321
4	534	1,847
5	2,256	9,992
6	8,969	50,136
7	33,058	227,536
8	114,149	870,072
9	360,508	1,887,748
0	930,588	623,800
11	1,350,852	2,644
12	782,536	
13	90,280	
14	276	

diameter

Reachable configurations

Distance	90 deg. turns	90/120 deg. turns
0	1	1
1	6	9
2	27	54
9	360,508	1,887,748
0	930,588	623,800
11	1,350,852	2,644
12	782,536	
13	90,280	
14	276	

Challenge:
How do you generate this table?

diameter

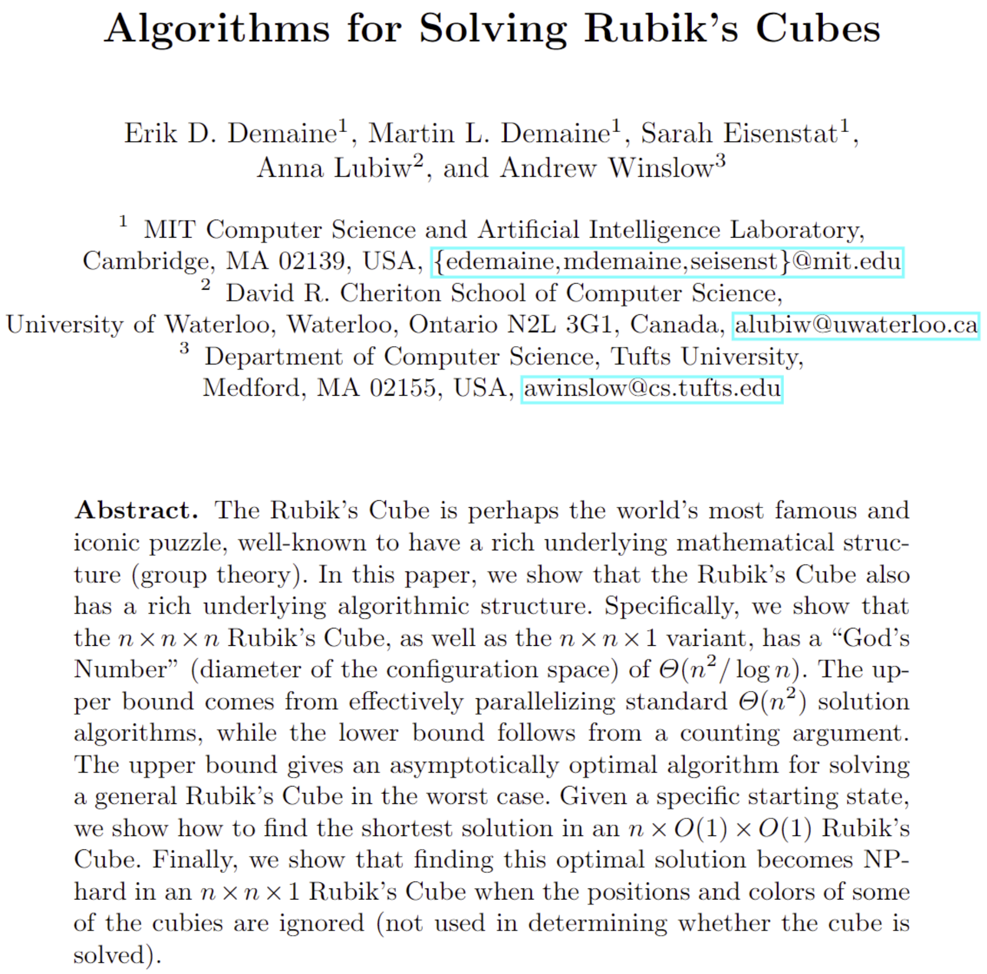
3 x 3 x 3 Rubik's Cube

<u>Configuration Graph</u>

- 43 quintillion vertices (approximately)
- Diameter: 20
  - 1995: require at least 20 moves.
  - 2008: 20 moves is enough from every position.
  - Using Google server farm.
  - 35 CPU-years of computation.
  - 20 seconds / set of 19.5 billion positions.
  - Lots of mathematical and programming tricks.

3 x 3 x 3 Rubik's Cube

What is the diameter of an $n \times n \times n$ cube?



Algorithms for Solving Rubik's Cubes

Erik D. Demaine[1], Martin L. Demaine[1], Sarah Eisenstat[1], Anna Lubiw[2], and Andrew Winslow[3]

[1] MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA 02139, USA, {edemaine,mdemaine,seisenst}@mit.edu
[2] David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, alubiw@uwaterloo.ca
[3] Department of Computer Science, Tufts University, Medford, MA 02155, USA, awinslow@cs.tufts.edu

Abstract. The Rubik's Cube is perhaps the world's most famous and iconic puzzle, well-known to have a rich underlying mathematical structure (group theory). In this paper, we show that the Rubik's Cube also has a rich underlying algorithmic structure. Specifically, we show that the $n \times n \times n$ Rubik's Cube, as well as the $n \times n \times 1$ variant, has a "God's Number" (diameter of the configuration space) of $\Theta(n^2/\log n)$. The upper bound comes from effectively parallelizing standard $\Theta(n^2)$ solution algorithms, while the lower bound follows from a counting argument. The upper bound gives an asymptotically optimal algorithm for solving a general Rubik's Cube in the worst case. Given a specific starting state, we show how to find the shortest solution in an $n \times O(1) \times O(1)$ Rubik's Cube. Finally, we show that finding this optimal solution becomes NP-hard in an $n \times n \times 1$ Rubik's Cube when the positions and colors of some of the cubies are ignored (not used in determining whether the cube is solved).

$$\Theta\left(\frac{n^2}{\log n}\right)$$

Roadmap

Today: Graph Basics

- What is a graph?
- Modeling problems as graphs.
- Graph representations (list vs. matrix)

Representing a Graph

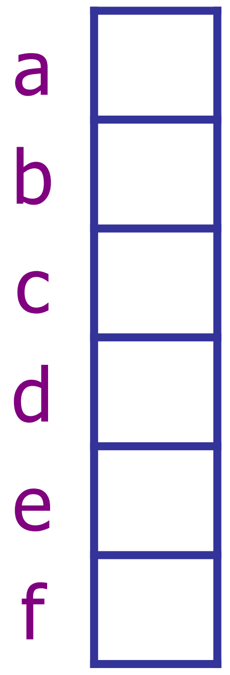
Graph consists of:

- Nodes
- Edges

Representing a Graph

Graph consists of:

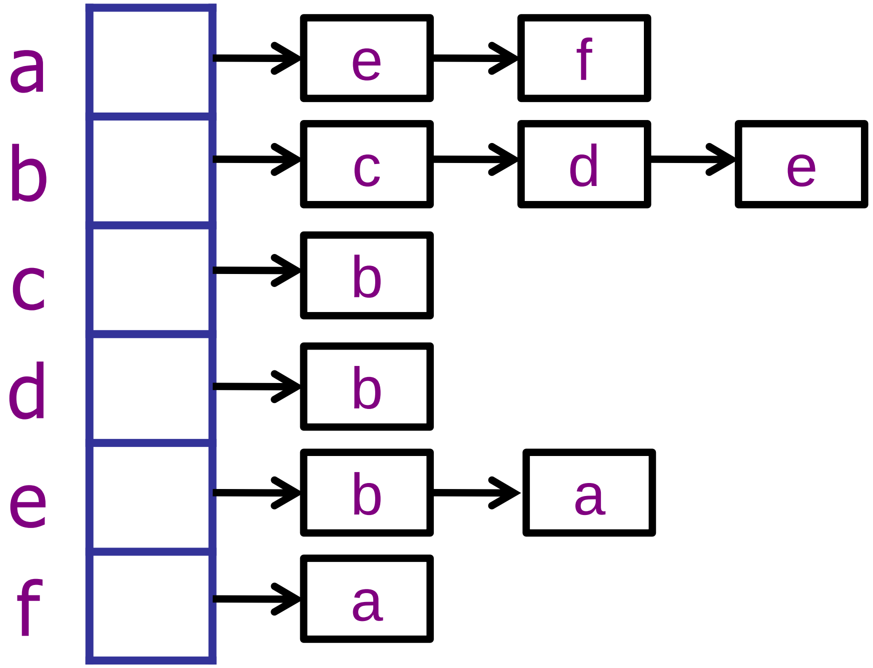
- Nodes: stored in an array
- Edges

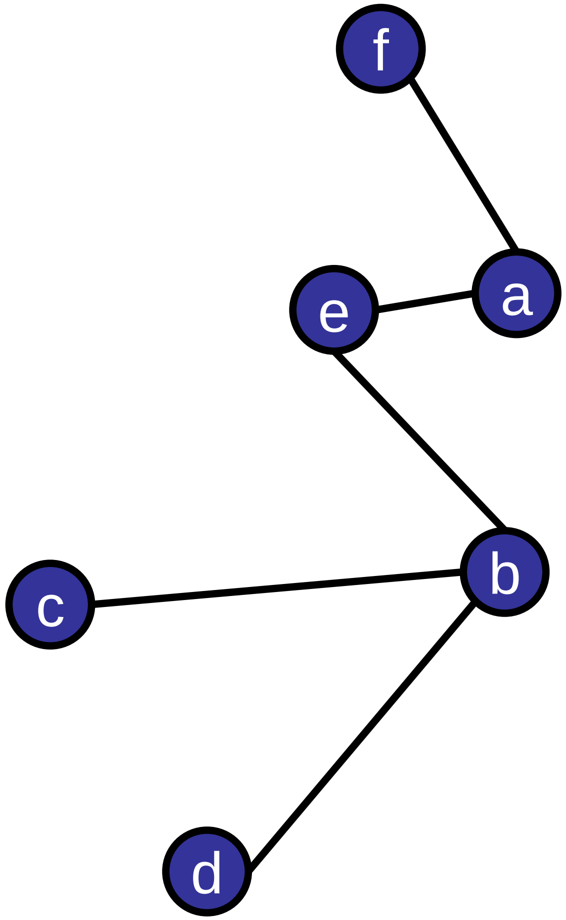


Adjacency List

Graph consists of:

- Nodes: stored in an array
- Edges: linked list per node





Adjacency List in Java

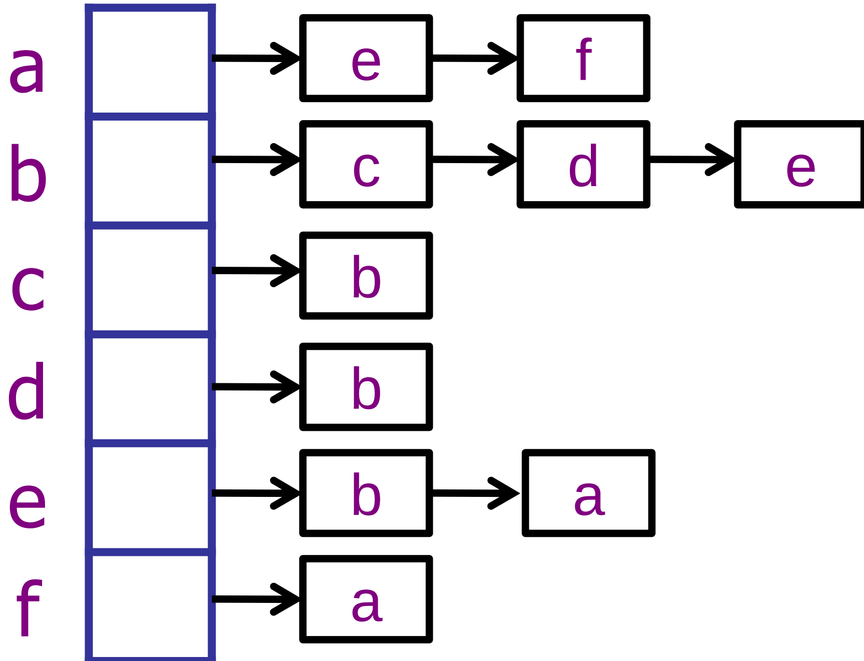
```
class NeighborList extends LinkedList<Integer> {
}


class Node {
  int key;
  NeighborList nbrs;
}


class Graph {
  Node[] nodeList;



}
```



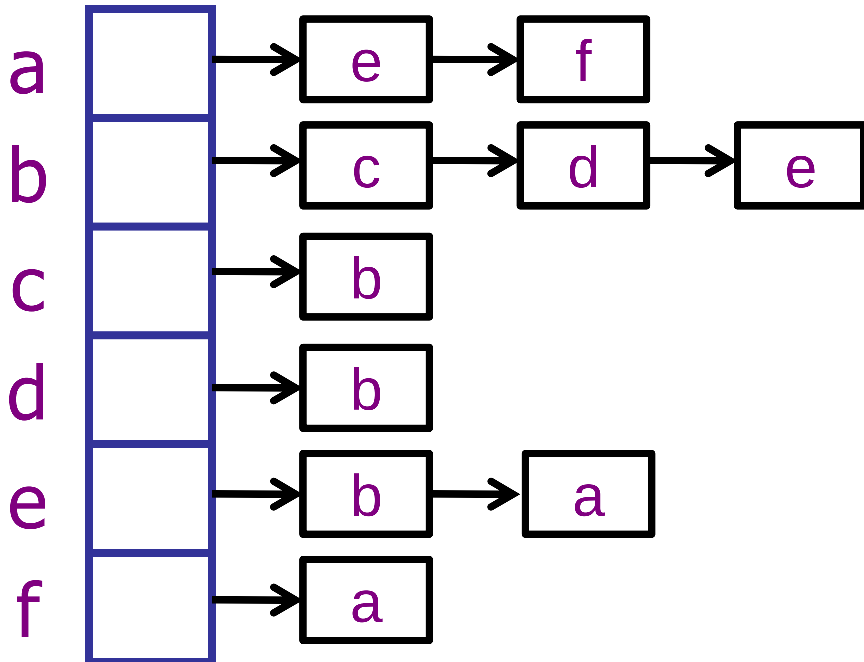
Adjacency List in Java

```
class Graph{
    List<List<Integer>> nodes;


}
```

More concise code is not *always* better…

- Harder to read
- Harder to debug
- Harder to extend



Representing a Graph

Graph consists of:

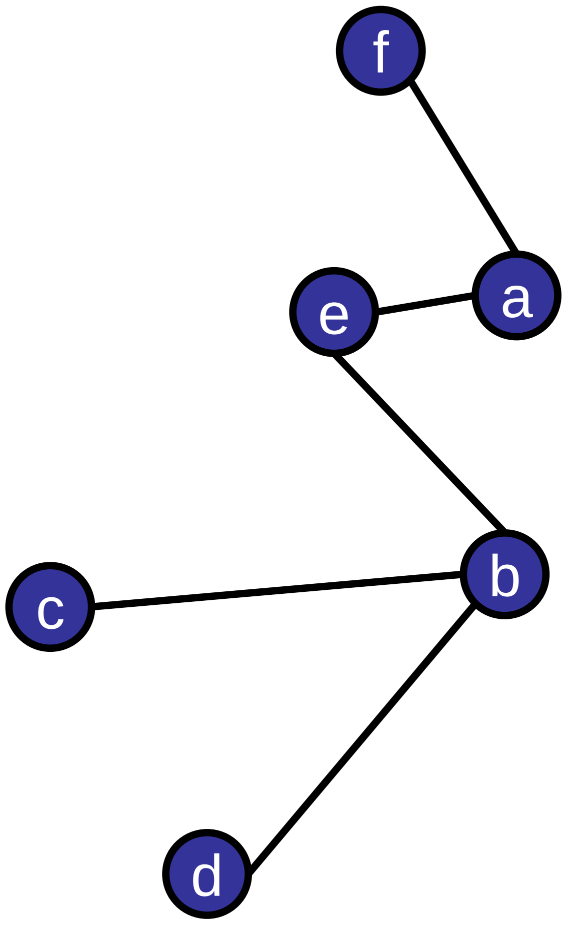
- Nodes
- Edges = pairs of nodes

Adjacency Matrix

Graph consists of:

- Nodes
- Edges = pairs of nodes

	a	b	c	d	e	f
a	0	0	0	0	1	1
b	0	0	1	1	1	0
c	0	1	0	0	0	0
d	0	1	0	0	0	0
e	1	1	0	0	0	0
f	1	0	0	0	0	0



Adjacency Matrix

Graph represented as:

A[v][w] = 1 iff (v,w) ∈ E

Neat property:

- A^2 = length 2 paths

	a	b	c	d	e	f
a	0	0	0	0	**1**	**1**
b	0	0	**1**	**1**	**1**	0
c	0	**1**	0	0	0	0
d	0	**1**	0	0	0	0
e	**1**	**1**	0	0	0	0
f	**1**	0	0	0	0	0

Adjacency Matrix

To find out if c and d are 2-hop neighbors:

- Let $B = A^2$
- $B[c, d] = A[c, .] \bullet A[., d]$

- $B[c, d] >= 1$ iff

  $A[c, x] = A[x, d] = 1$

  for some x.

	a	b	c	d	e	f
a	0	0	0	0	1	1
b	0	0	1	1	1	0
c	0	1	0	0	0	0
d	0	1	0	0	0	0
e	1	1	0	0	0	0
f	1	0	0	0	0	0

Adjacency Matrix

Graph represented as:

$A[v][w] = 1$ iff $(v,w) \in E$

Neat properties:

- A^2 = length 2 paths
- A^4 = length 4 paths

	a	b	c	d	e	f
a	0	0	0	0	1	1
b	0	0	1	1	1	0
c	0	1	0	0	0	0
d	0	1	0	0	0	0
e	1	1	0	0	0	0
f	1	0	0	0	0	0

Neat way to figure out connectivity…
Neat way to figure out diameter…
Not always the most efficient…
Parallelizes well….

Adjacency Matrix

Graph represented as:

$A[v][w] = 1$ iff $(v,w) \in E$

Neat properties:

- A^2 = length 2 paths
- A^4 = length 4 paths
- $A^\infty \approx$ Google pagerank?

(Simulate random walk by replacing '1' with probability.)

	a	b	c	d	e	f
a	0	0	0	0	1	1
b	0	0	1	1	1	0
c	0	1	0	0	0	0
d	0	1	0	0	0	0
e	1	1	0	0	0	0
f	1	0	0	0	0	0

Adjacency Matrix in Java

Graph represented as:

A[v][w] = 1 iff (v,w) ∈ E

```
class Graph {

  boolean[][] adjMatrix;


}
```

	a	b	c	d
a	0	0	0	0
b	0	0	**1**	**1**
c	0	**1**	0	0
d	0	**1**	0	0
e	**1**	**1**	0	0
f	**1**	0	0	0

Adjacency Matrix in Java

Graph represented as:

A[v][w] = 1 iff (v,w) ∈ E

```
class Graph {

  Node[][] adjMatrix;


}
```

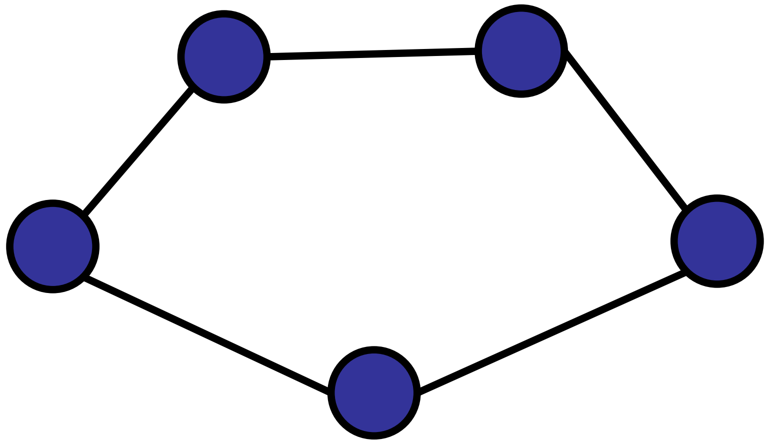
	a	b	c	d
a	0	0	0	0
b	0	0	1	1
c	0	1	0	0
d	0	1	0	0
e	1	1	0	0
f	1	0	0	0

Trade-offs

Adjacency Matrix vs. Array?

For a cycle, which representation is better?

1. [✓ Adjacency list]
2. Adjacency matrix
3. Equivalent



ARCHIPELAGO

is open

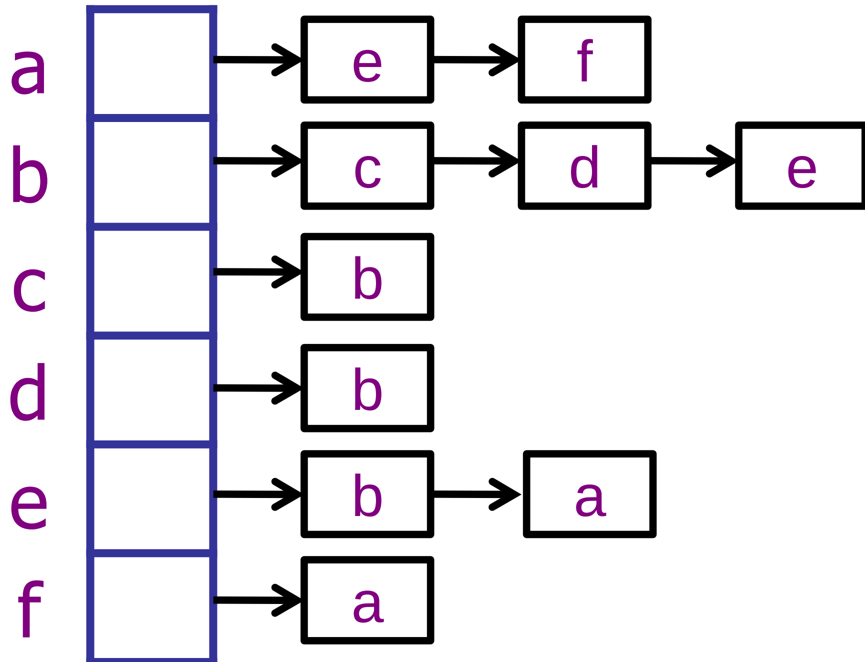
Adjacency List

Memory usage for graph G = (V, E):

- array of size |V|
- linked lists of size |E|

Total: O(V + E)

For a cycle: O(V)



Adjacency Matrix

Memory usage for graph G = (V, E):

- array of size |V|*|V|

Total: $O(V^2)$

For a cycle: $O(V^2)$

	a	b	c	d	e	f
a	0	0	0	0	**1**	**1**
b	0	0	**1**	**1**	**1**	0
c	0	**1**	0	0	0	0
d	0	**1**	0	0	0	0
e	**1**	**1**	0	0	0	0
f	**1**	0	0	0	0	0

For a clique, which representation is better?

1. Adjacency matrix
2. Adjacency list
3. Equivalent

ARCHIPELAGO

is open

Adjacency List vs. Matrix

Memory usage for graph $G = (V, E)$:

- Adjacency List: $O(V + E)$
- Adjacency Matrix: $O(V^2)$

For a cycle: $O(V)$ vs. $O(V^2)$

For a clique: $O(V + E) = O(V^2)$ vs. $O(V^2)$

Adjacency List vs. Matrix

Memory usage for graph G = (V, E):

- Adjacency List: $O(V + E)$
- Adjacency Matrix: $O(V^2)$

For a cycle: $O(V)$ vs. $O(V^2)$

For a clique: $O(V + E) = O(V^2)$ vs. $O(V^2)$

Base rule: if graph is dense then use an adjacency matrix; else use an adjacency list.

dense: $|E| = \theta(V^2)$

Which representation for Facebook Graph? Query: Are Bob and Joe friends?

1. Adjacency List
2. [✓ Adjacency Matrix]
3. Equivalent

List: (much) better space.
Matrix: somewhat faster

Which representation for Facebook Graph? Query: List all my friends?

1. [✓ Adjacency List]
2. Adjacency Matrix
3. Equivalent

Trade-offs

Adjacency Matrix:

- Fast query: are v and w neighbors?
- Slow query: find me any neighbor of v.
- Slow query: enumerate all neighbors.

Adjacency List:

- Fast query: find me any neighbor.
- Fast query: enumerate all neighbors.
- Slower query: are v and w neighbors?

Graph Representations

Key questions to ask:

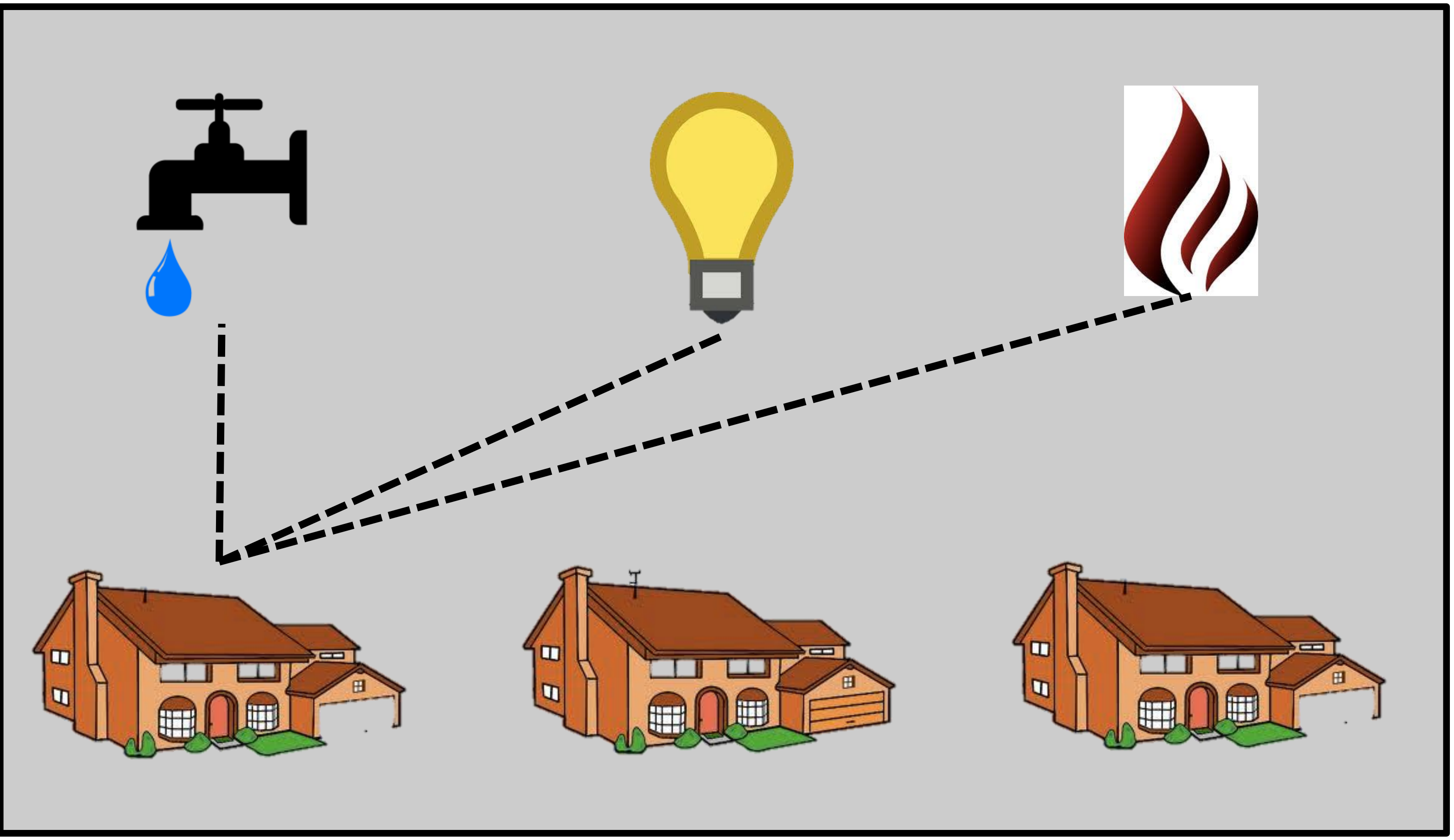
- Space usage: is graph dense or sparse?
- Queries: what type of queries do I need?
  - Enumerate neighbors?
  - Query relationship?

Roadmap

Today: Graph Basics

- What is a graph?
- Modeling problems as graphs.
- Graph representations (list vs. matrix)

Puzzle



Connect each house to all three utilities (water, electricity, gas).
Do not let any of the cables or pipes cross.
(Or show that it is impossible.)