

D&D&G

Dungeons and Dragons  
And Graphs

# Agenda

- About me
- About you
- What are graphs and why do they matter
- Paths
- Connections
- Community

# About me

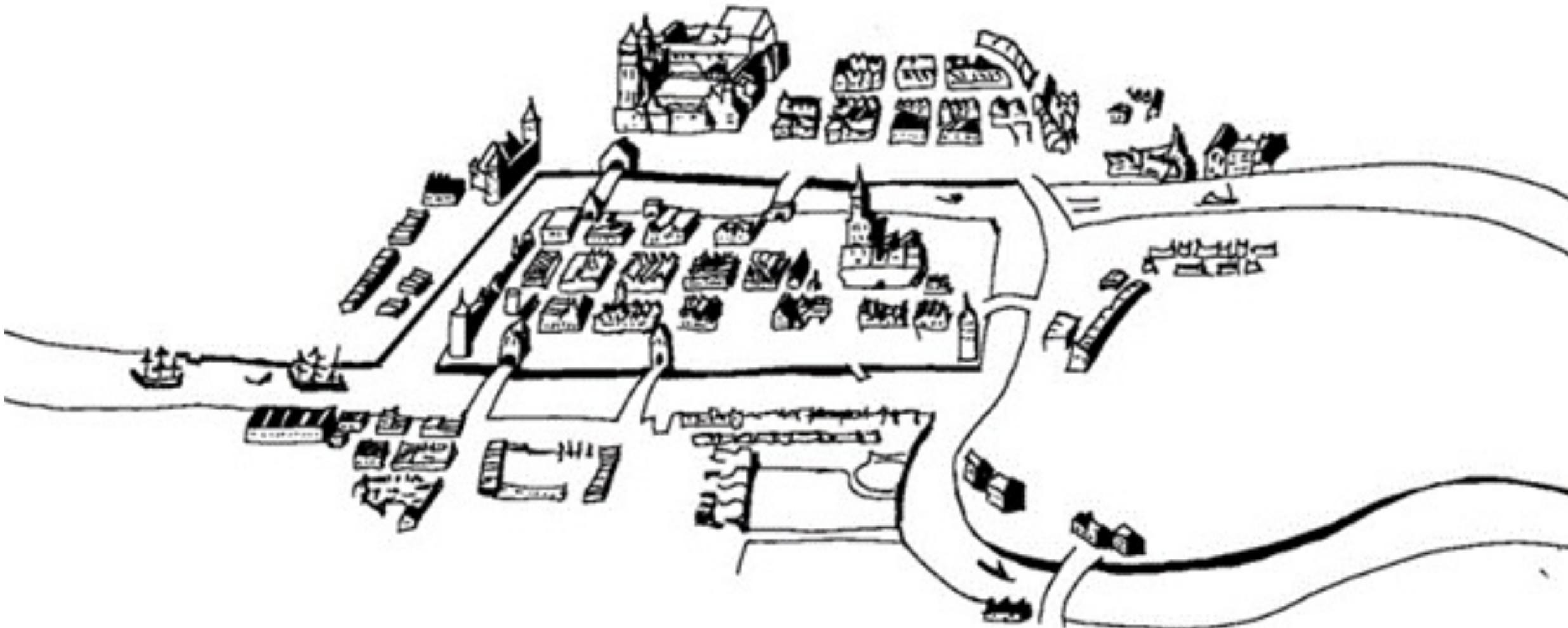
- Graphs
- Python
- Drawing
- Sometimes DMing

# About you

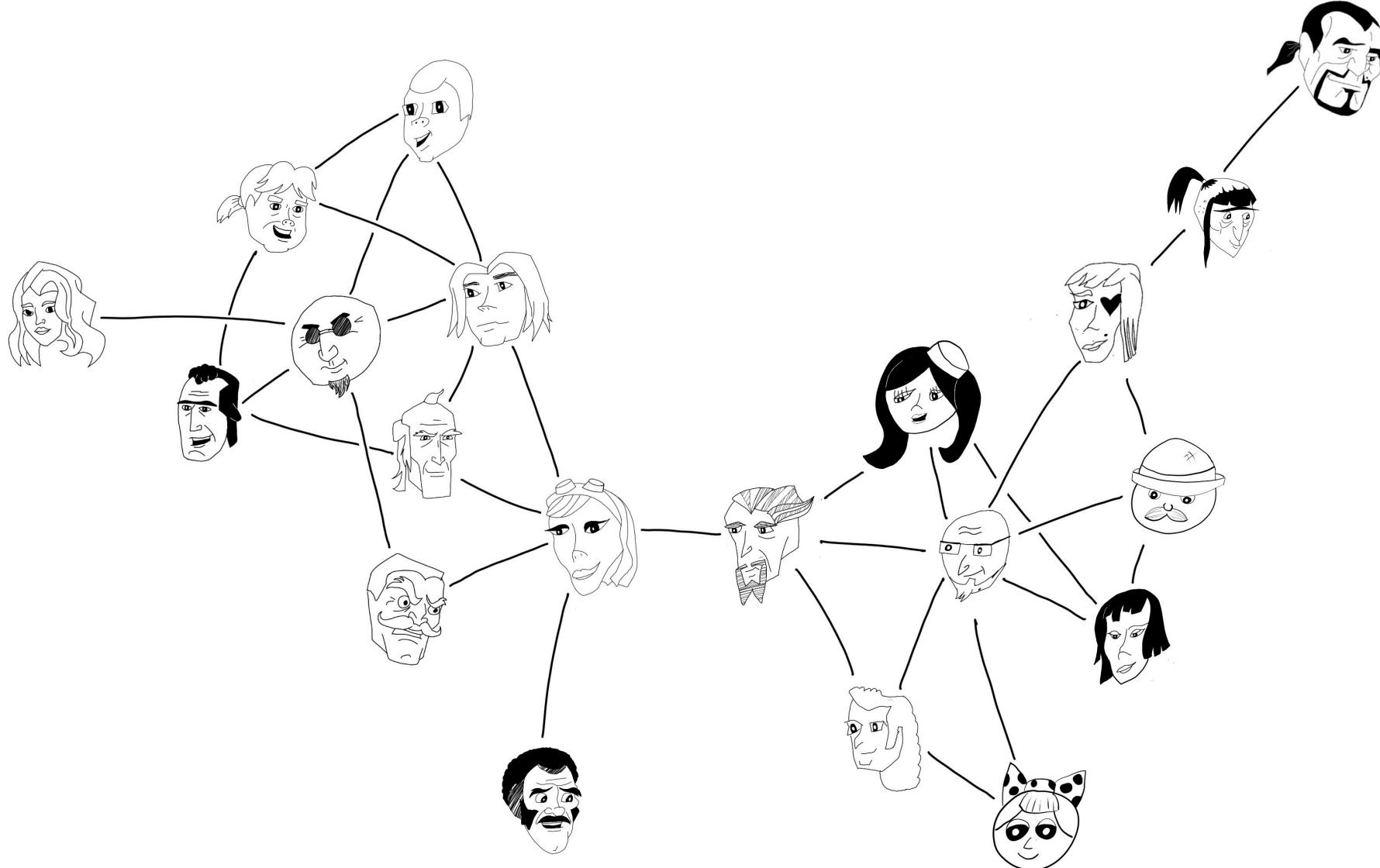
- Some programming
  - Maps/Dictionaries
  - Objects
  - Maybe have heard of or occasionally use graphs and want to dive deeper
- Some math
  - Matrices
  - Logarithms

What are graphs?

# What are graphs

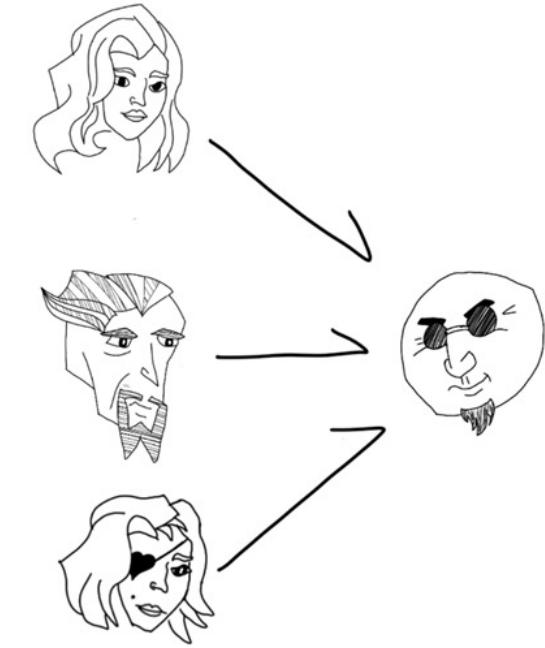


# What are graphs

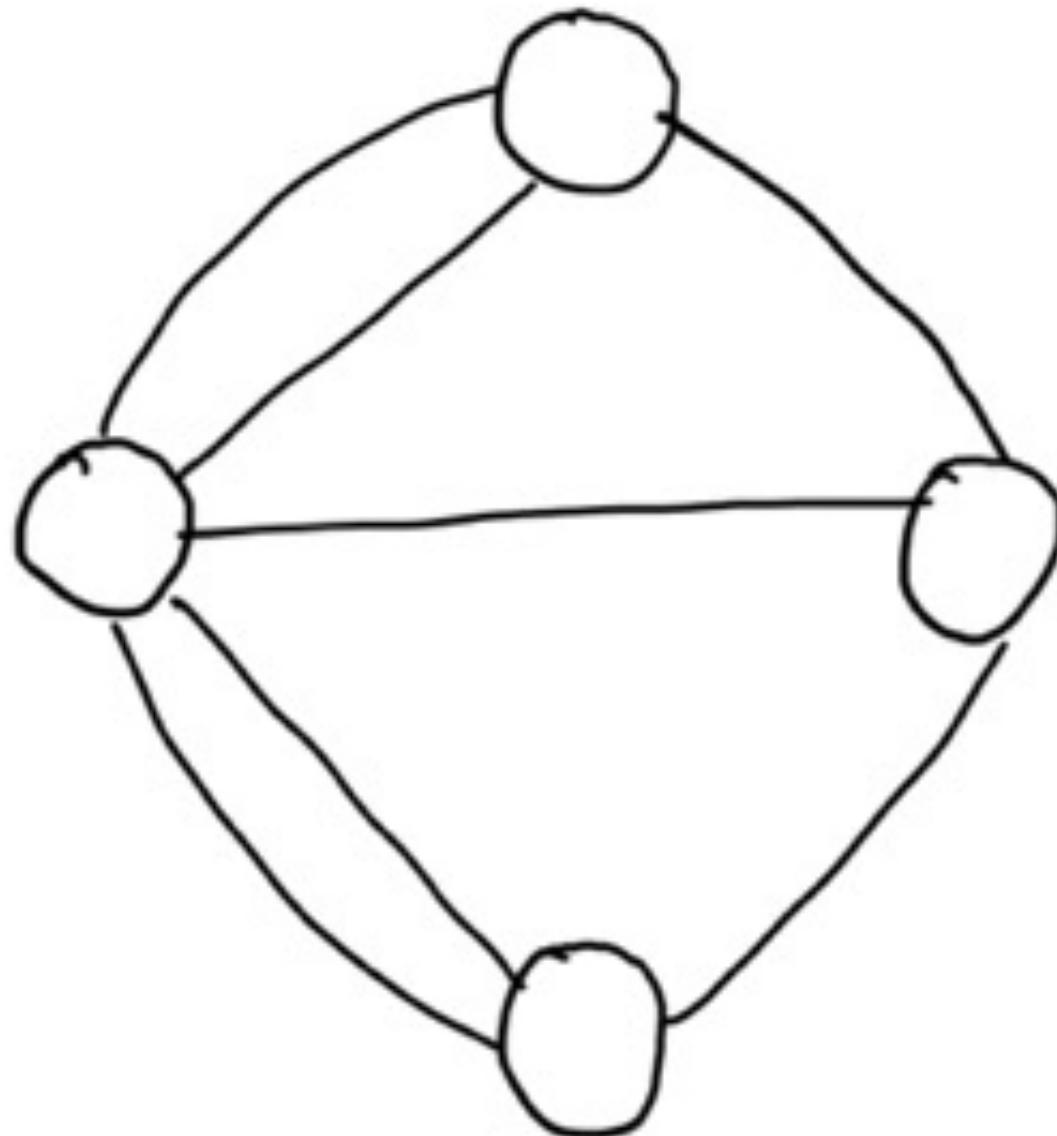


# What are graphs

- Graphs are a data structure that codifies systems as **entities** and **relationships**
- Treats relationships as first-class members
  - Directional
  - Transitive

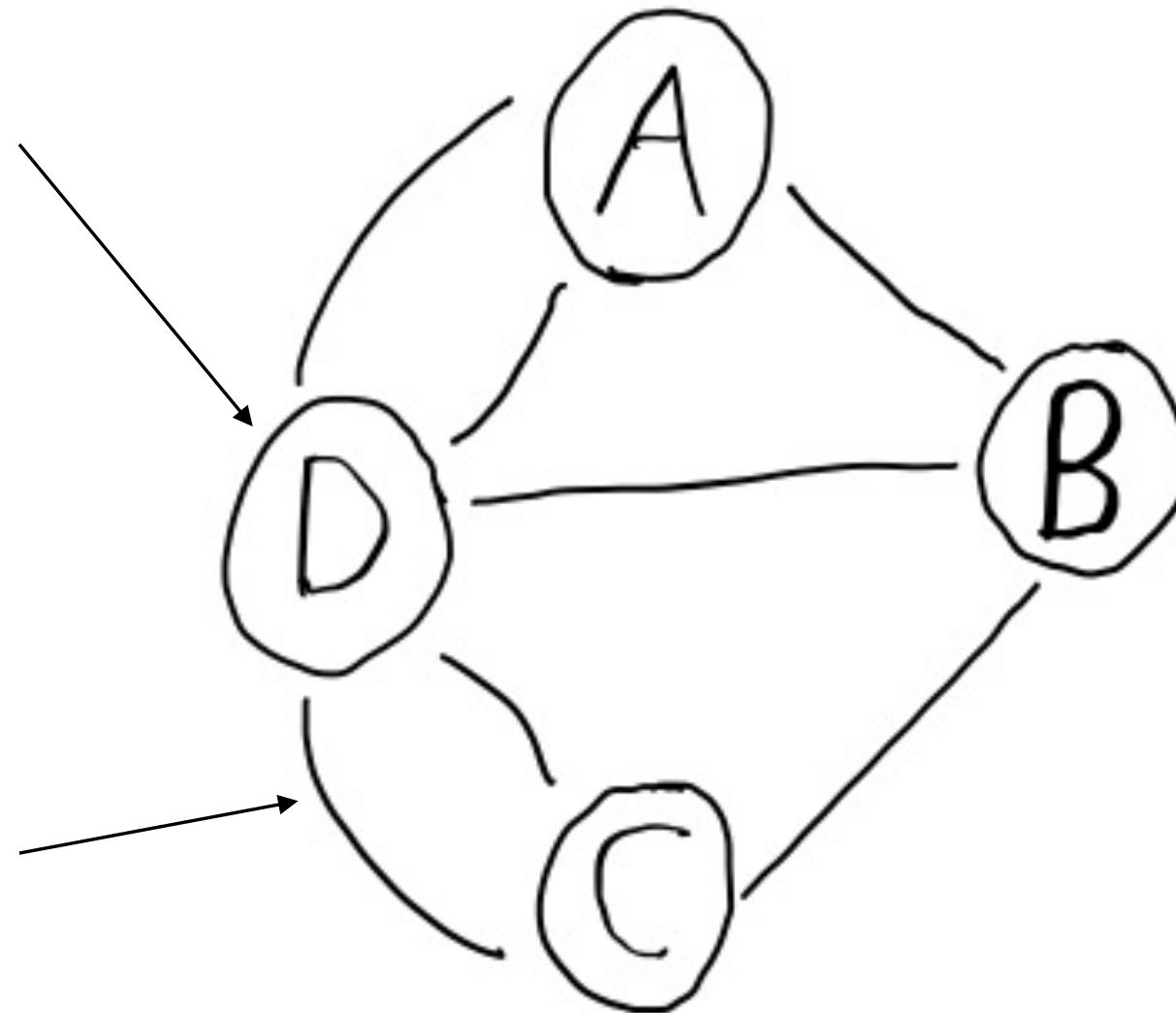


# What are graphs?



# What are graphs?

Node: Fundamental indivisible unit from which graphs are formed.

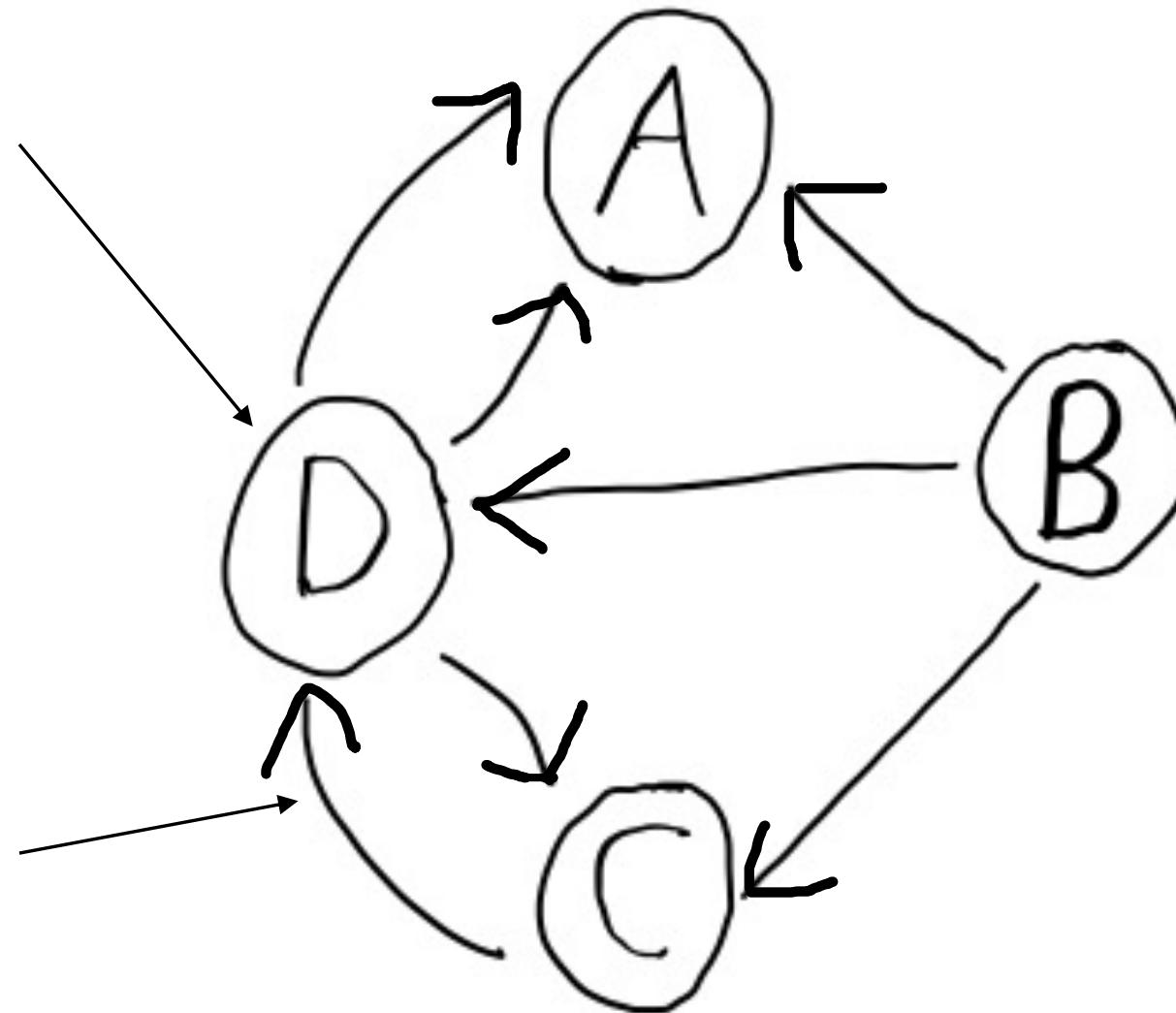


Edge: (un)ordered pairs of nodes representing a connection between two nodes.

A graph is a collection of nodes and edges

# What are graphs?

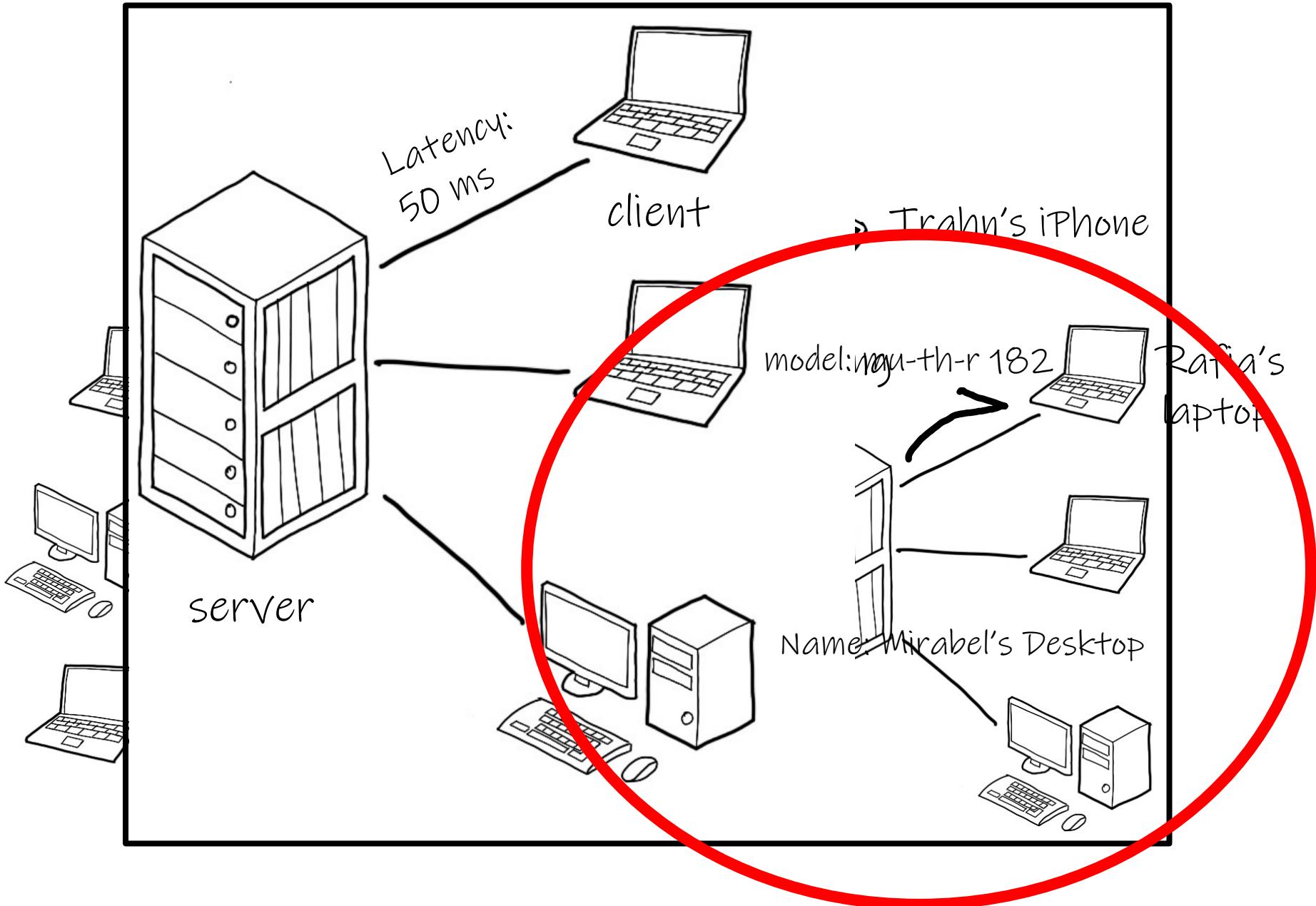
Node: Fundamental indivisible unit from which graphs are formed.



Edge: (un)ordered pairs of nodes representing a connection between two nodes.

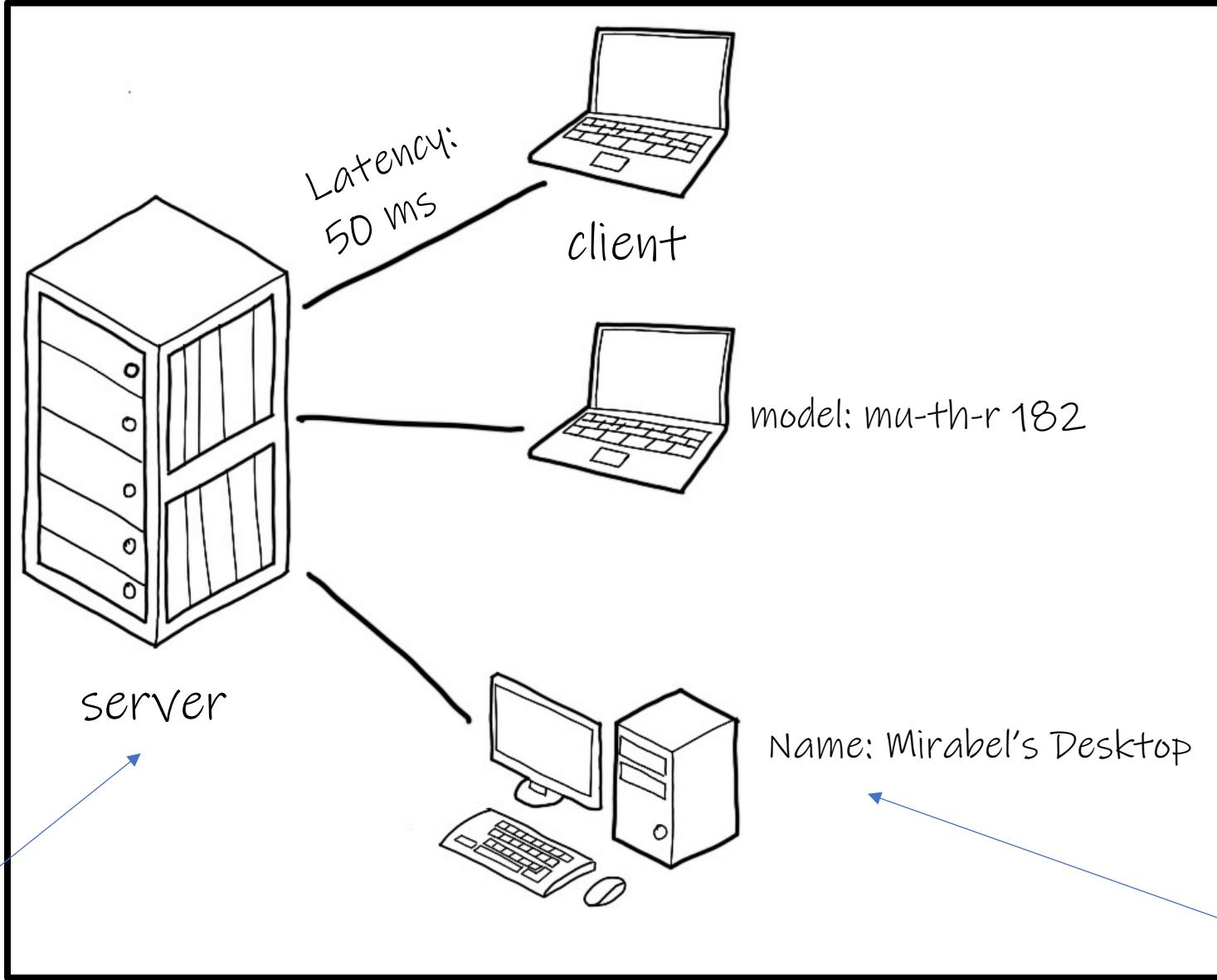
Directed graphs have one-way edges

# Paths & subgraphs



# Paths & subgraphs

Labels:  
“class” of a  
node



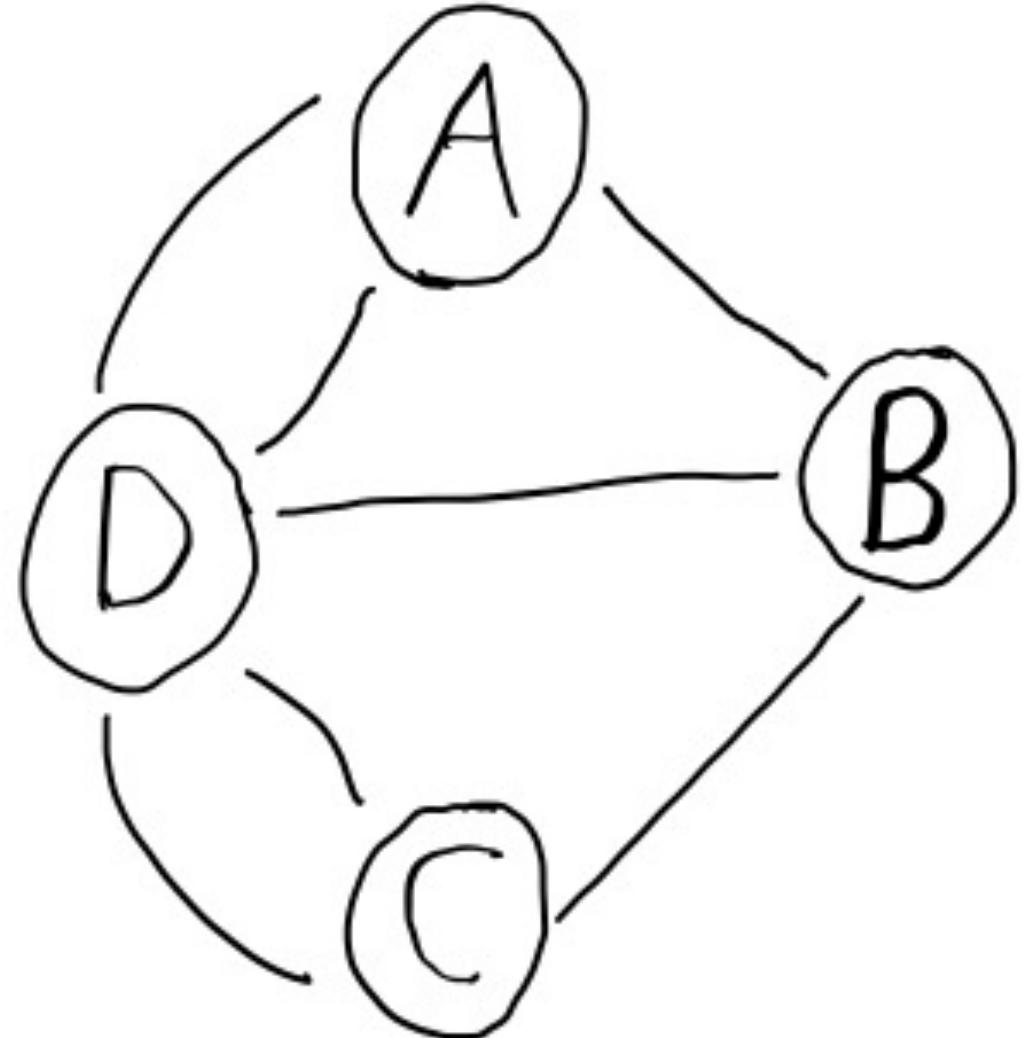
Property:  
Any  
additional  
information  
attached to  
a node/edge

# Applications

- Maps
- Social networks
- State spaces (puzzle solving)
- Disease propagation
- Community identification
- Chemical reactions
- Tensorflow
- Search engines
- Fraud detection
- Air/vehicle/internet traffic
- Document analysis
- Image analysis

# Representing graphs in Python

- Matrices- adjacency, degree
- Edge list
- Dictionary
- NetworkX



# Representing graphs in Python

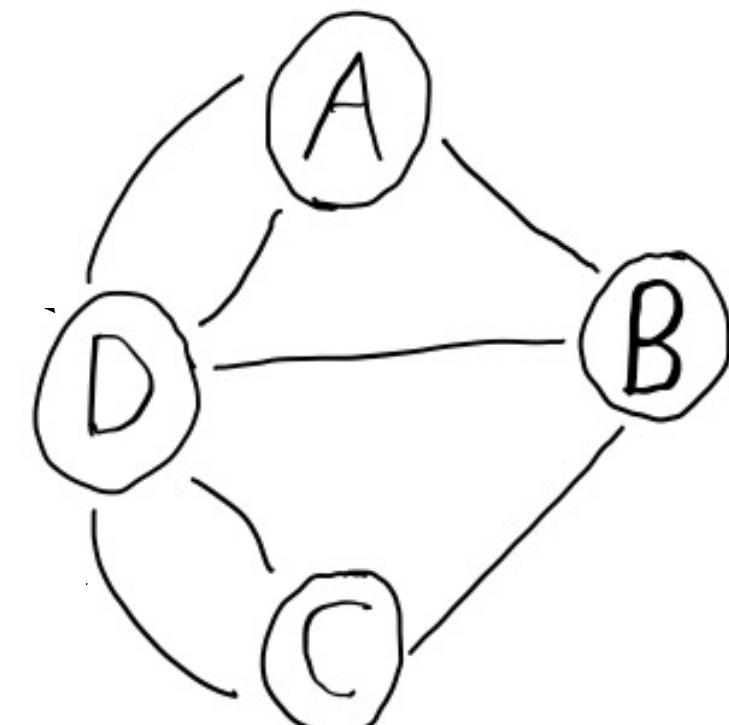
- Matrices

	To			
From	A	B	C	D
A	0	1	0	2
B	1	0	1	1
C	0	1	0	2
D	2	1	2	0

```
K =  
[[0,1,0,2],[1,0,1,1],[0,1,0,2],[2,1,2,0]]  
for k in K: print(k)
```

```
K2 =  
np.array([[0,1,0,2],[1,0,1,1],[0,1,0,2],[2,1,2,0]])  
print(K2)  
[0, 1, 0, 2]  
[1, 0, 1, 1]  
[0, 1, 0, 2]  
[2, 1, 2, 0]
```

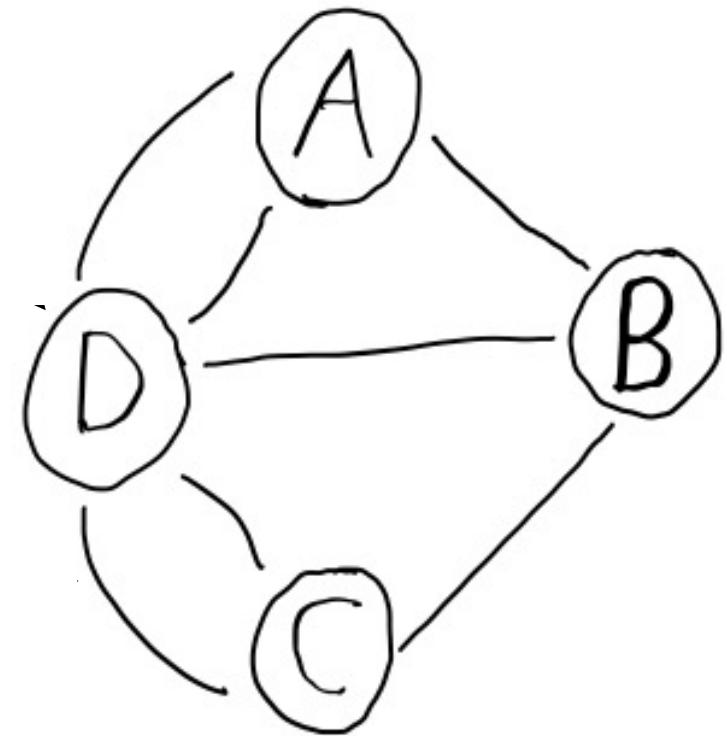
```
[[0 1 0 2]  
 [1 0 1 1]  
 [0 1 0 2]  
 [2 1 2 0]]
```



# Representing graphs in Python

- Edge list

```
[ (A,B), (A,D), (A,D), (B,C),  
  (B,D), (C,D), (C,D) ]
```



K3 =

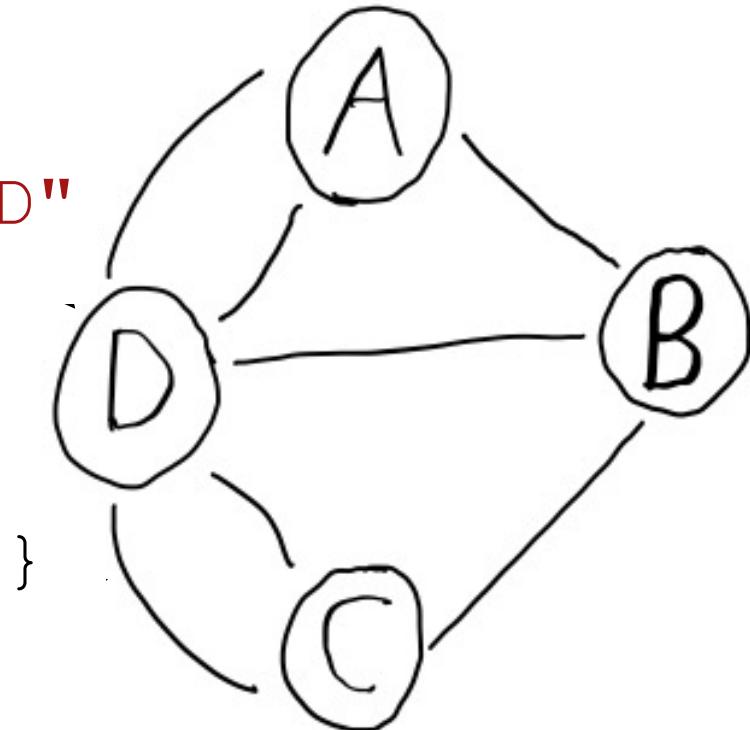
```
[ ("A", "B") , ("A", "D") , ("A", "D") , ("B", "C") , ("B", "D") ,  
  ("C", "D") , ("C", "D") ]  
[ ('A', 'B') , ('A', 'D') , ('A', 'D') , ('B', 'C') ,  
  ('B', 'D') , ('C', 'D') , ('C', 'D') ]
```

# Representing graphs in Python

- Dictionary

```
{ A: [B, D, D],  
  B: [A, C, D],  
  C: [B, D, D],  
  D: [A, A, C, C, B] }
```

```
A, B, C, D = "ABCD"  
K4 = {A: [B, D, D],  
      B: [A, C, D],  
      C: [B, D, D],  
      D: [A, A, C, C, B] }
```



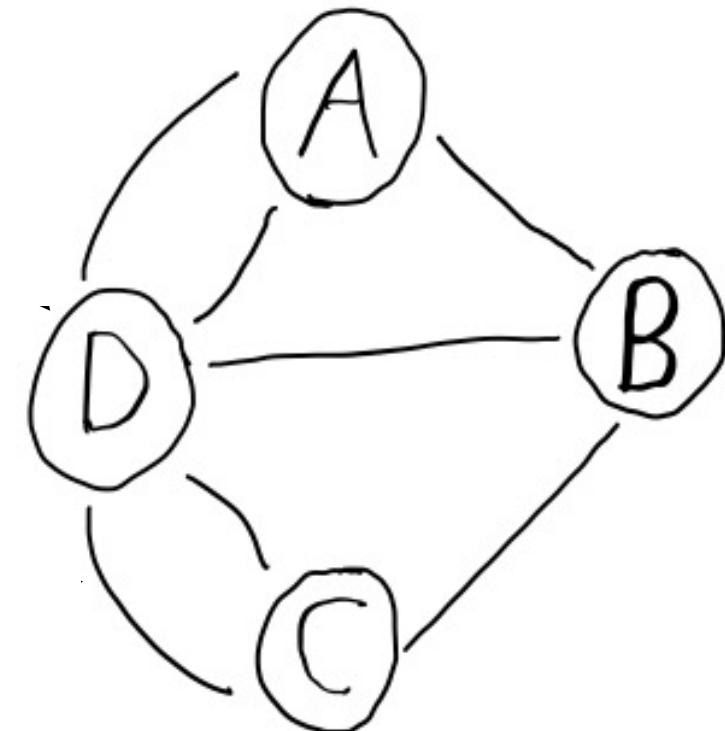
```
{ 'A': ['B', 'D', 'D'],  
  'B': ['A', 'C', 'D'],  
  'C': ['B', 'D', 'D'],  
  'D': ['A', 'A', 'C', 'C', 'B'] }
```

# Representing graphs in Python

- As a custom-made object

```
Class Node():
    def __init__(name):
        self.name = name
        self.neighbors = []
```

```
Class Graph():
    def __init__(adj_dict):
        <some logic>
```



# What are graphs?

- Don't use graphs:
  - Lots of writes
  - Querying all (or very large parts) of a database
- Alternatives:
  - Linear:
    - Lists/arrays
    - Linked list
    - Stacks/Queues
  - Non-linear:
    - Heaps
    - Hashtables
    - Trees?
  - Databases
    - Object, doc, columnar, relational

YOU MEET IN A TAVERN

~~YOU MEET IN A TAVERN~~

disclaimer

GET AN ALE LAGER





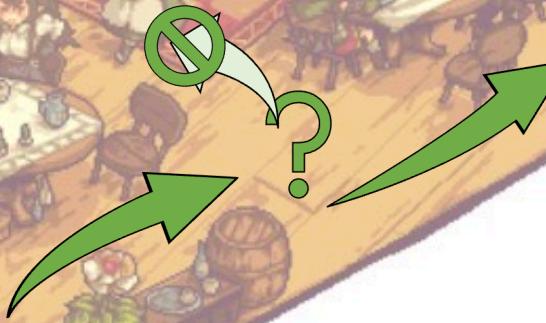








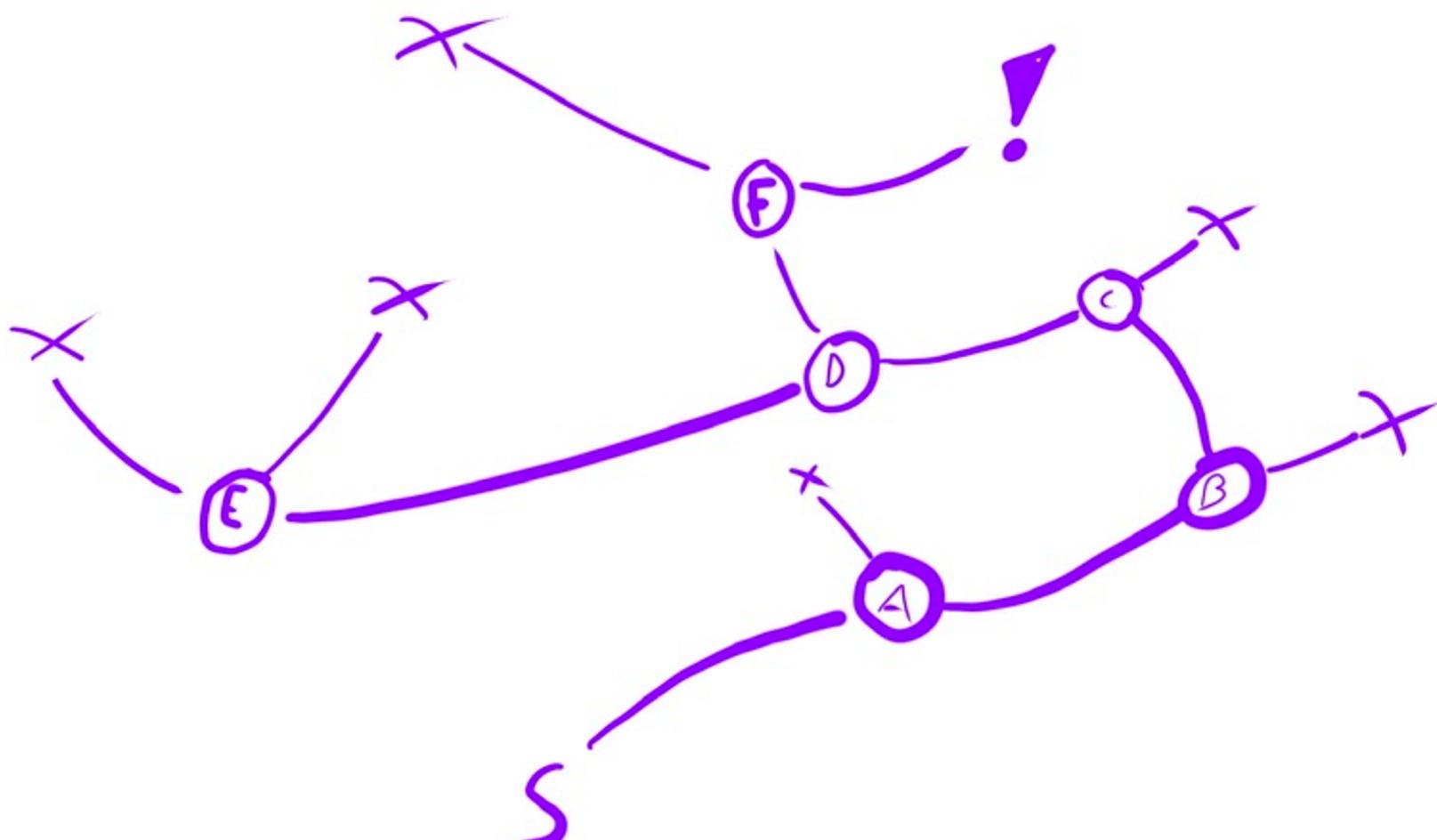






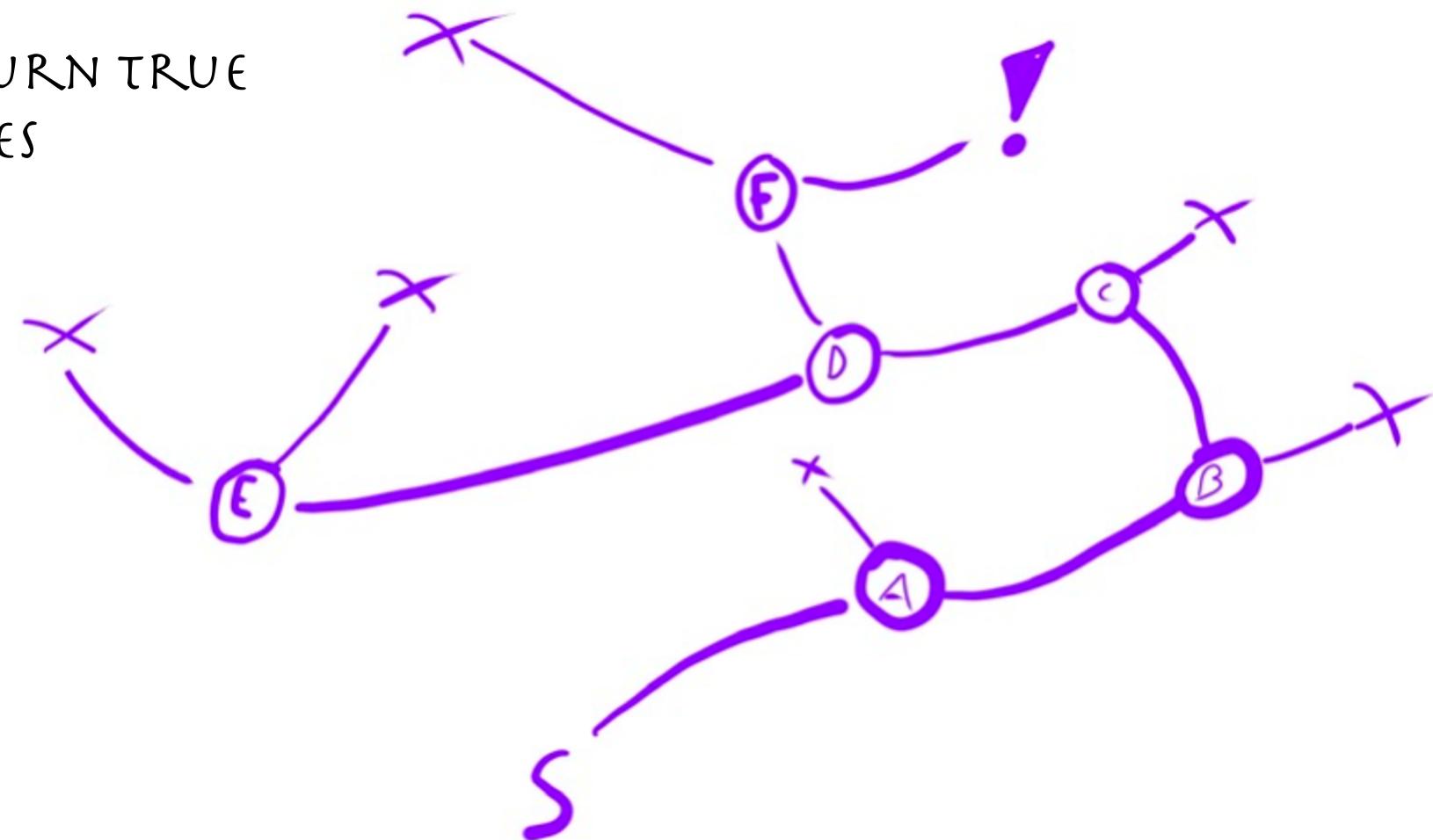
# MAZES/MAPS CAN BE REPRESENTED AS GRAPHS!

- ENTITIES: INTERSECTIONS
- RELATIONSHIPS: PATHS BETWEEN INTERSECTIONS



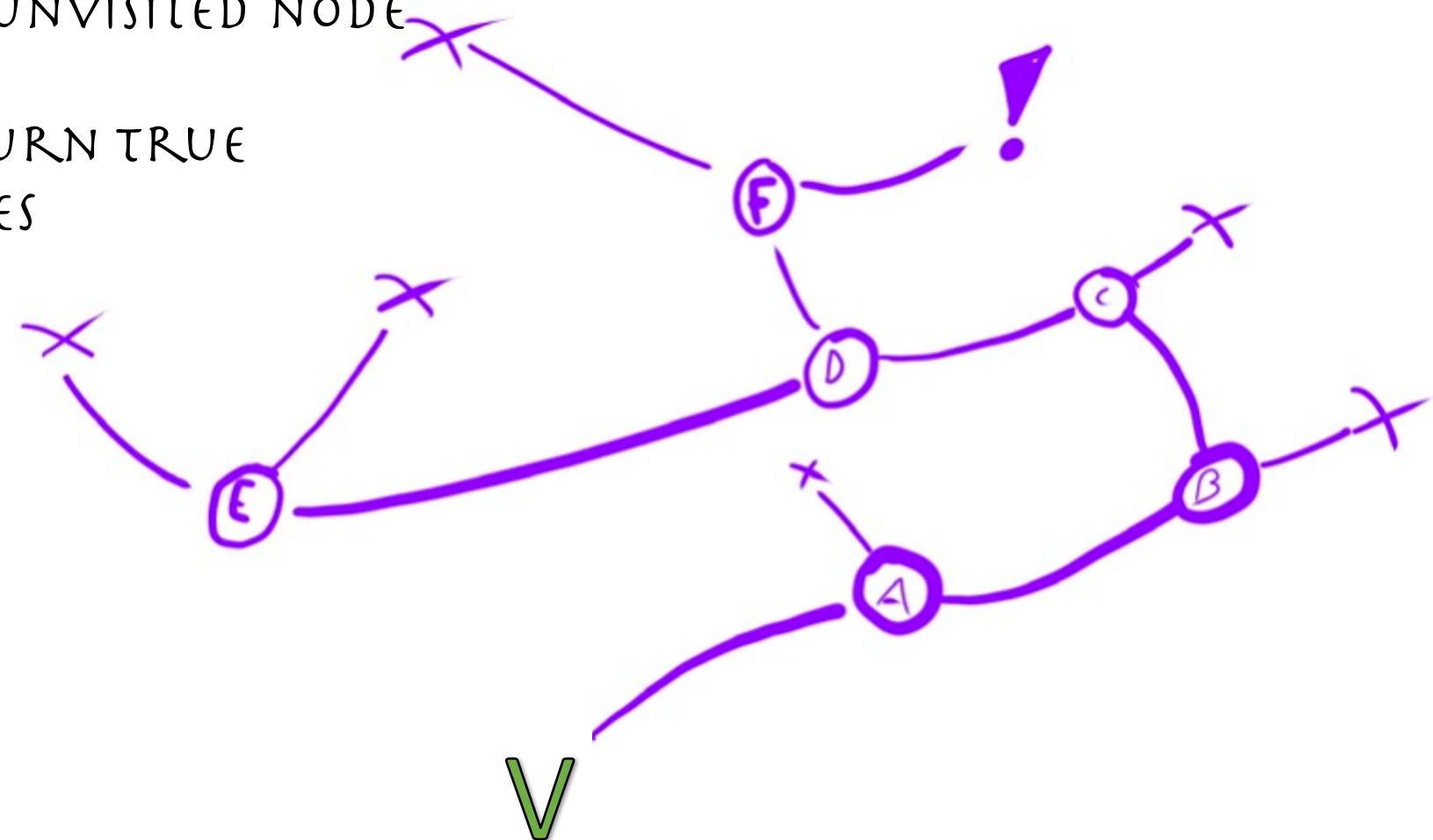
## DEPTH FIRST SEARCH

- IF YOU'VE FOUND THE END POINT:
  - RETURN TRUE #SUCCESS!
- MARK YOUR CURRENT NODE AS VISITED
- FOR EVERY NEIGHBORING UNVISITED NODE
  - #PERFORM DFS
  - IF DFS(NEIGHBOR): RETURN TRUE
- #YOU'VE RUN OUT OF NODES
- RETURN FALSE



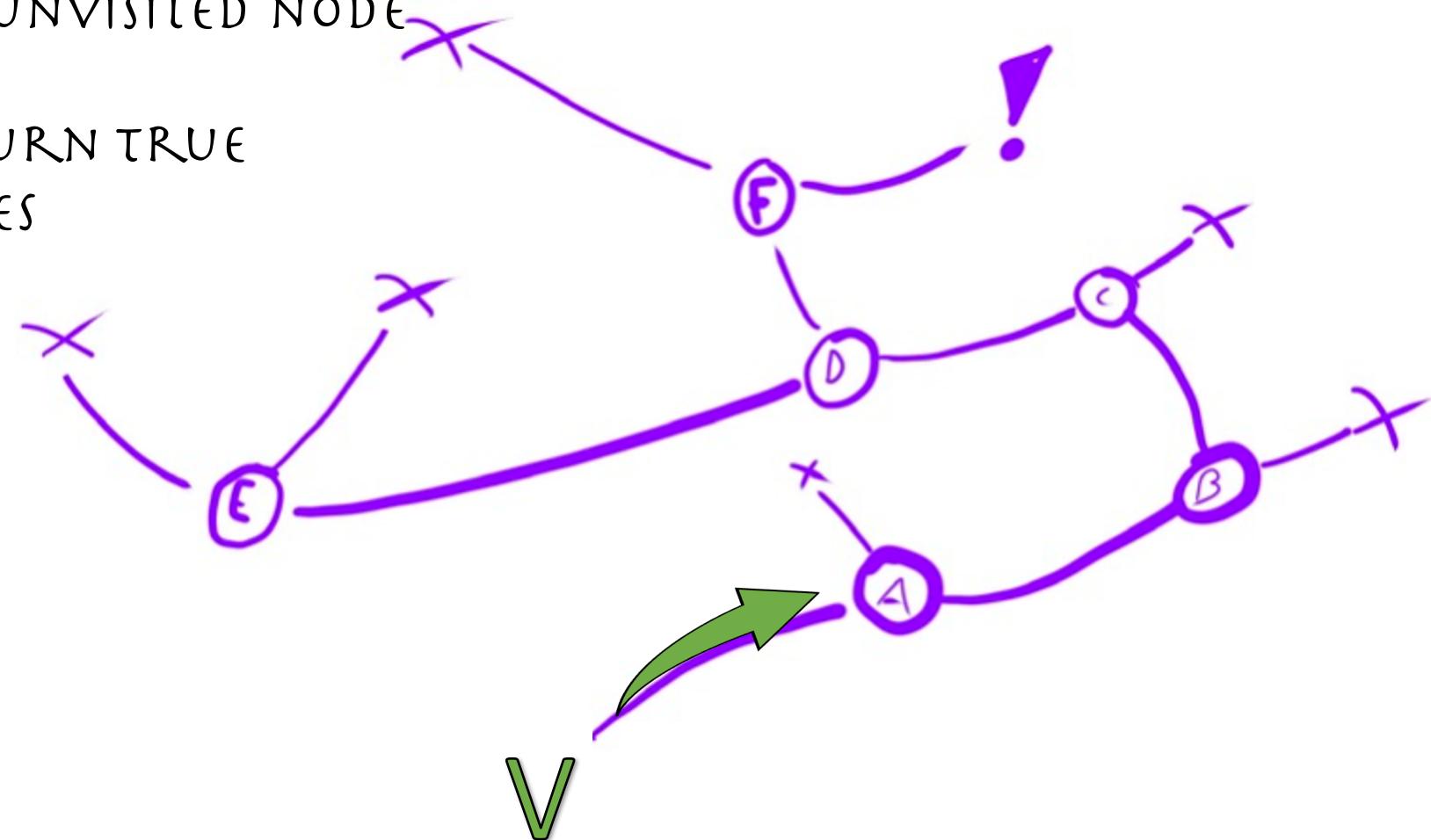
## DEPTH FIRST SEARCH

- IF YOU'VE FOUND THE END POINT:
  - #SUCCESS!
  - RETURN TRUE
- MARK YOUR CURRENT NODE AS VISITED
- FOR EVERY NEIGHBORING UNVISITED NODE
  - #PERFORM DFS
  - IF DFS(NEIGHBOR): RETURN TRUE
- #YOU'VE RUN OUT OF NODES
- RETURN FALSE



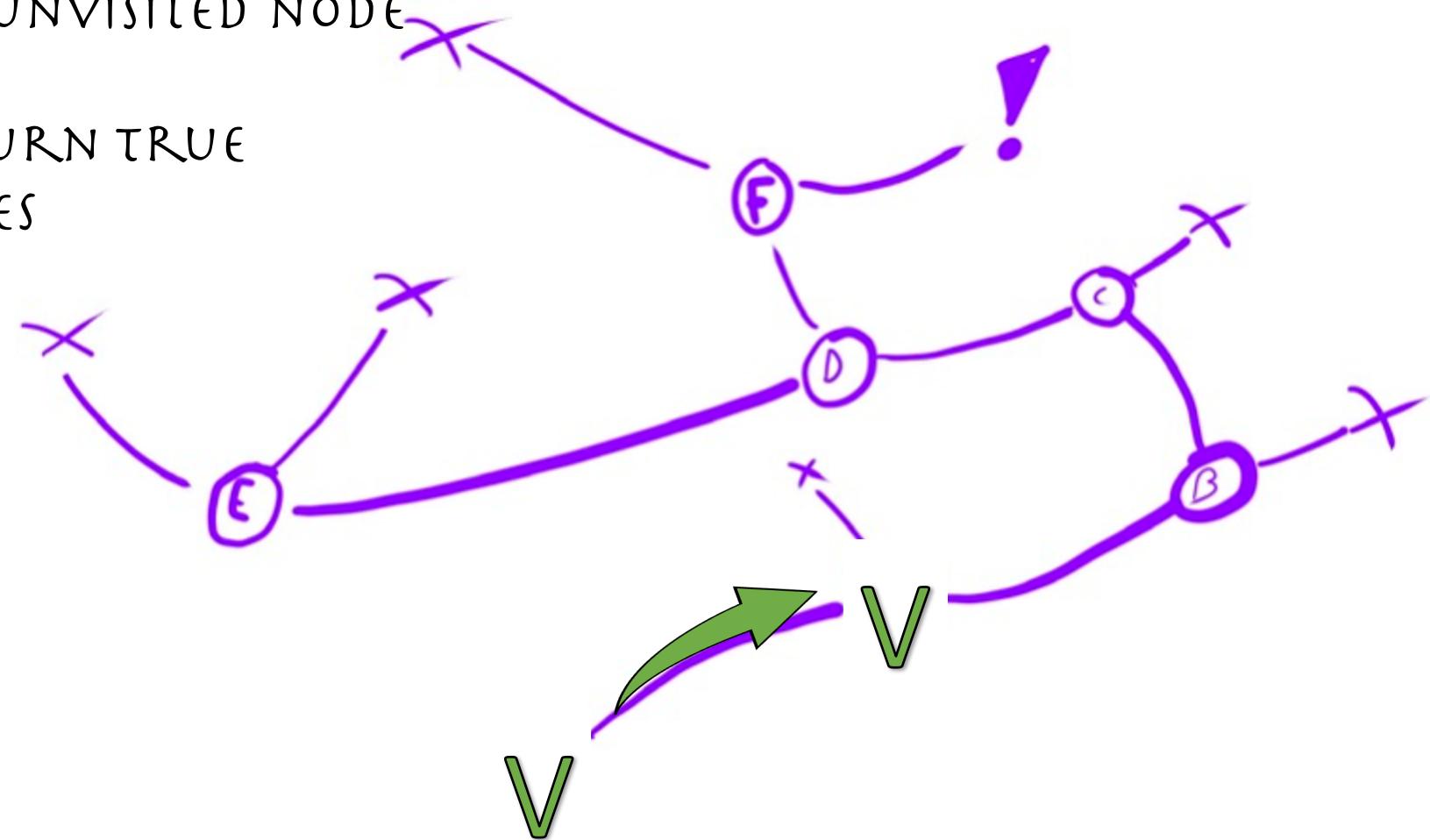
## DEPTH FIRST SEARCH

- IF YOU'VE FOUND THE END POINT:
  - #SUCCESS!
  - RETURN TRUE
- MARK YOUR CURRENT NODE AS VISITED
- FOR EVERY NEIGHBORING UNVISITED NODE
  - #PERFORM DFS
  - IF DFS(NEIGHBOR): RETURN TRUE
- #YOU'VE RUN OUT OF NODES
- RETURN FALSE



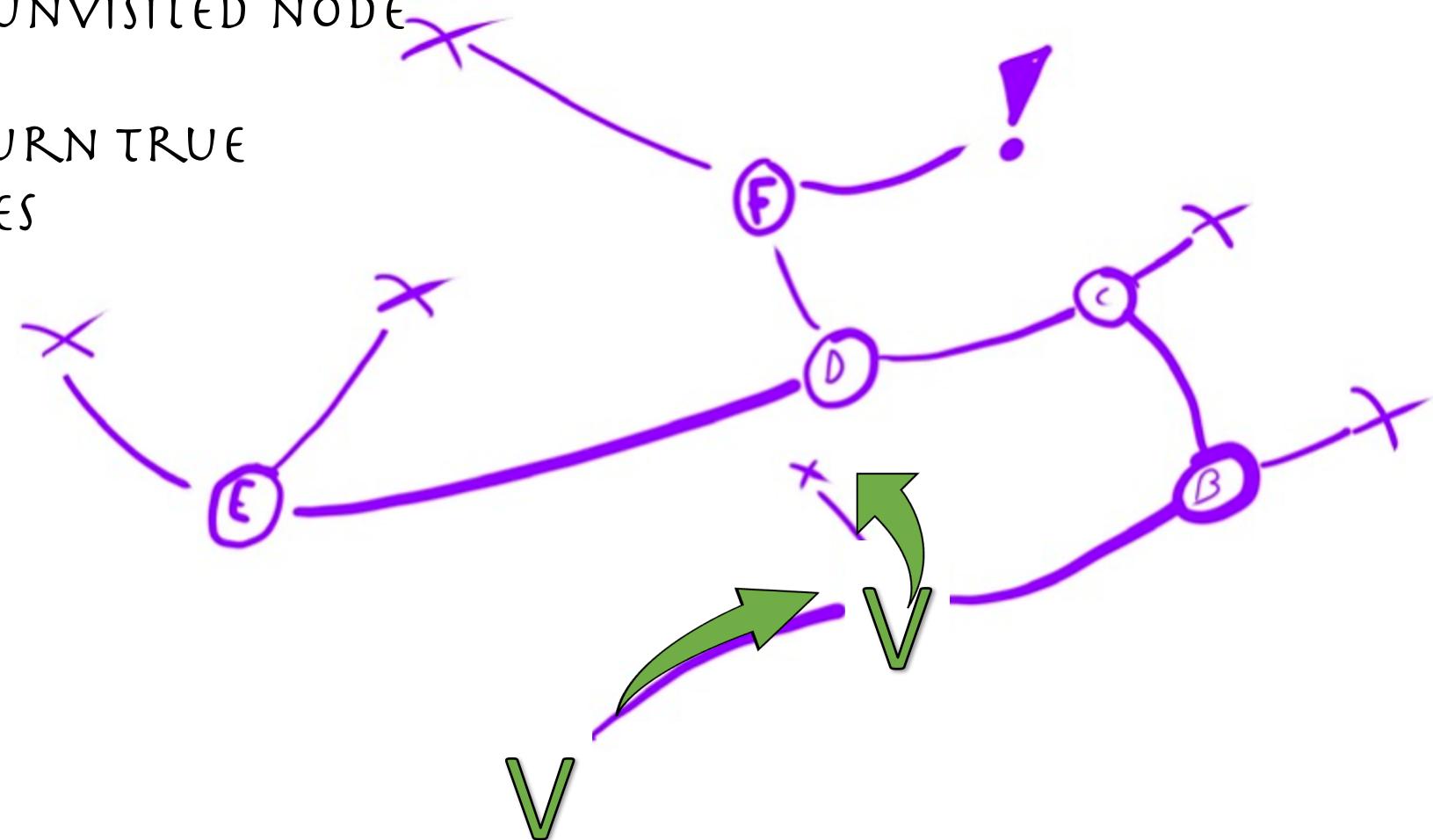
## DEPTH FIRST SEARCH

- IF YOU'VE FOUND THE END POINT:
  - #SUCCESS!
  - RETURN TRUE
- MARK YOUR CURRENT NODE AS VISITED
- FOR EVERY NEIGHBORING UNVISITED NODE
  - #PERFORM DFS
  - IF DFS(NEIGHBOR): RETURN TRUE
- #YOU'VE RUN OUT OF NODES
- RETURN FALSE



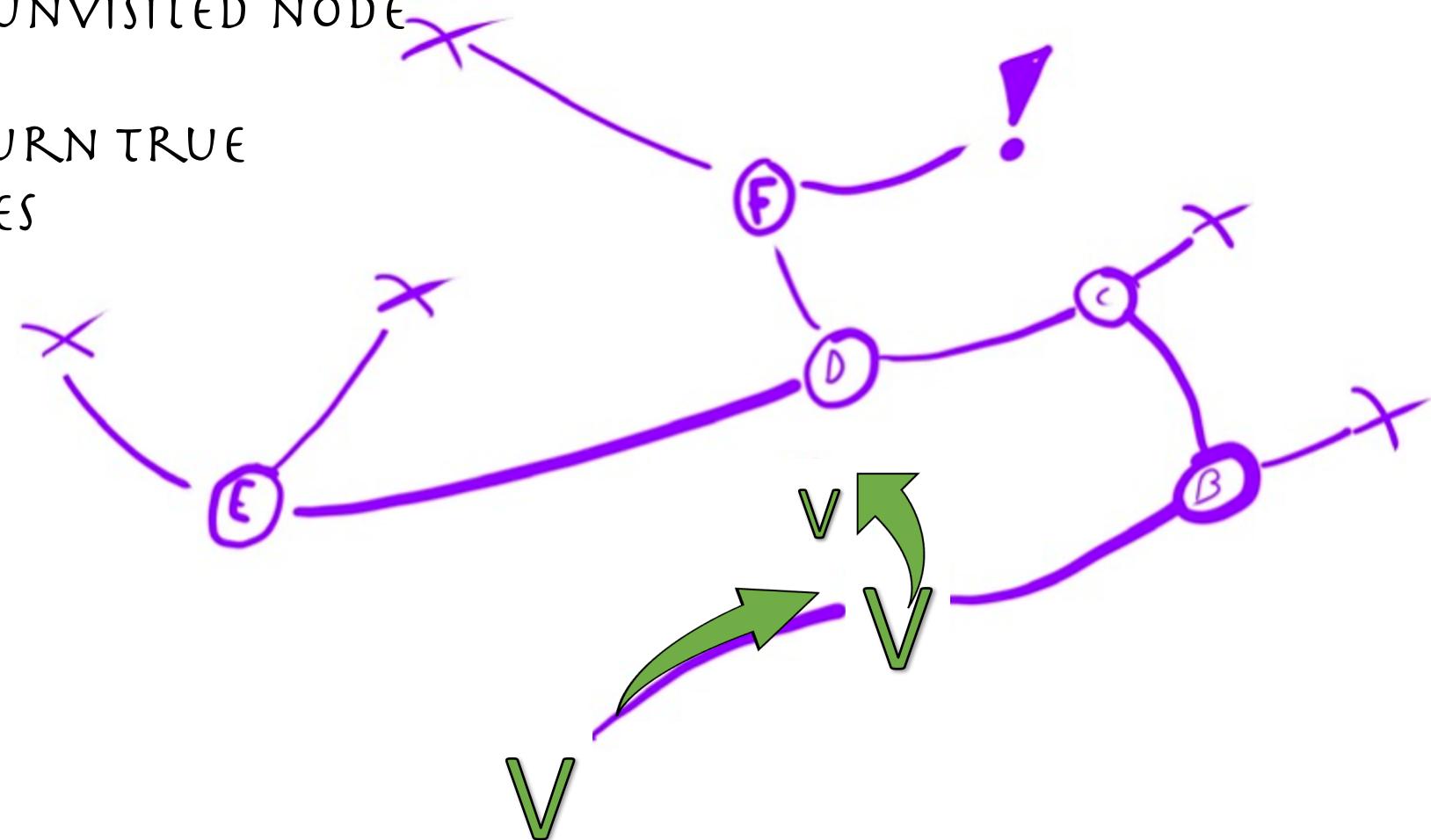
## DEPTH FIRST SEARCH

- IF YOU'VE FOUND THE END POINT:
  - #SUCCESS!
  - RETURN TRUE
- MARK YOUR CURRENT NODE AS VISITED
- FOR EVERY NEIGHBORING UNVISITED NODE
  - #PERFORM DFS
  - IF DFS(NEIGHBOR): RETURN TRUE
- #YOU'VE RUN OUT OF NODES
- RETURN FALSE



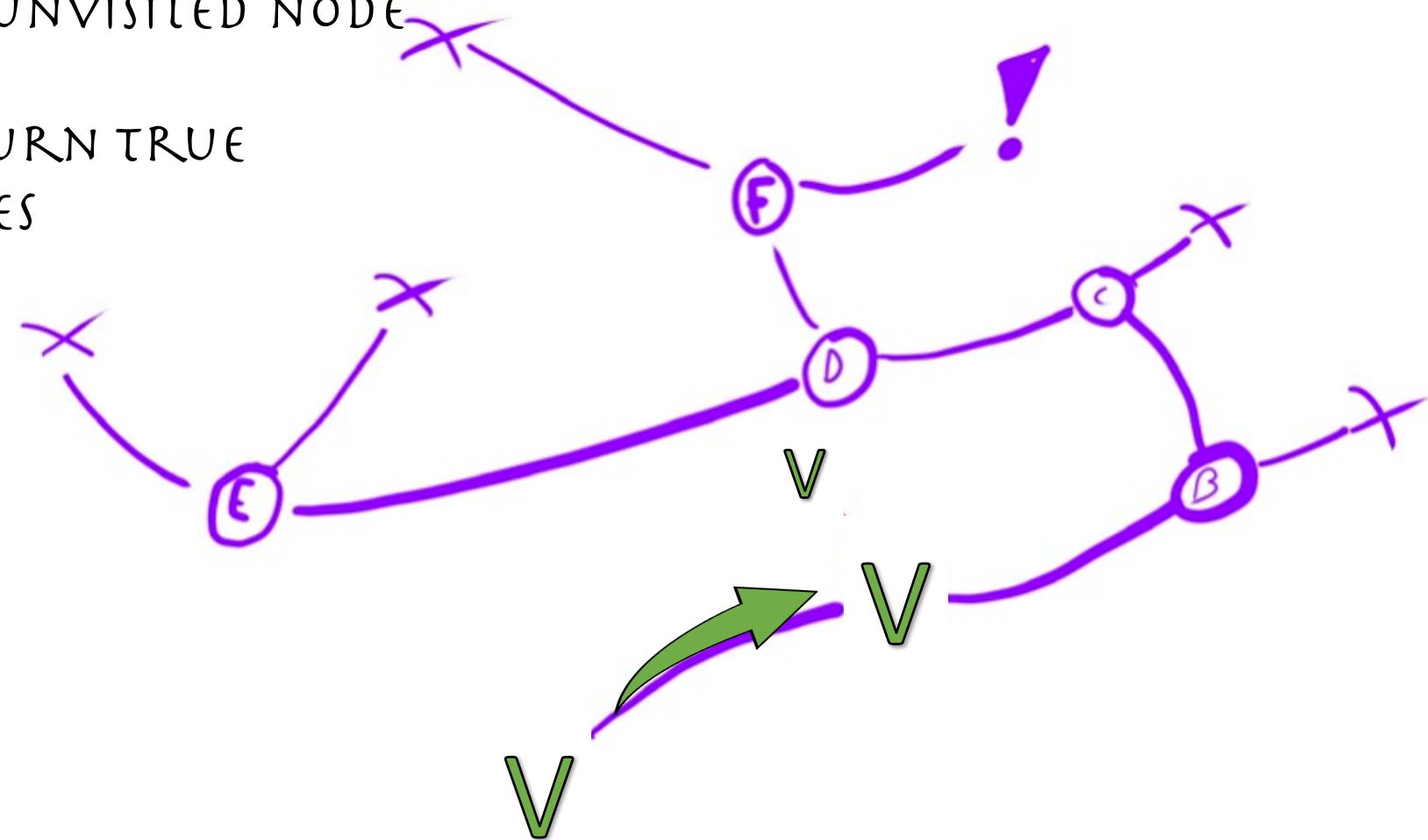
## DEPTH FIRST SEARCH

- IF YOU'VE FOUND THE END POINT:
  - #SUCCESS!
  - RETURN TRUE
- MARK YOUR CURRENT NODE AS VISITED
- FOR EVERY NEIGHBORING UNVISITED NODE
  - #PERFORM DFS
  - IF DFS(NEIGHBOR): RETURN TRUE
- #YOU'VE RUN OUT OF NODES
- RETURN FALSE



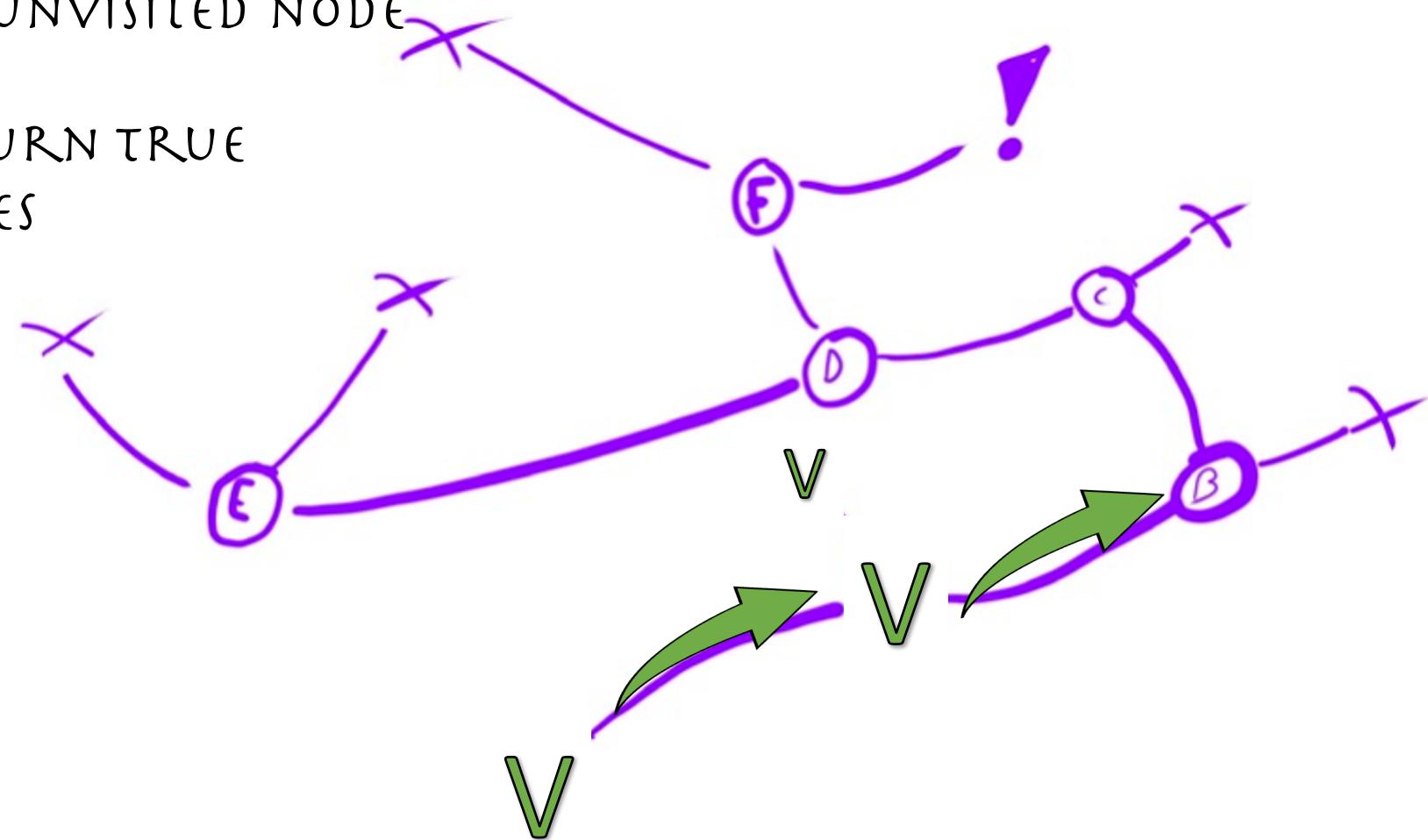
## DEPTH FIRST SEARCH

- IF YOU'VE FOUND THE END POINT:
  - #SUCCESS!
  - RETURN TRUE
- MARK YOUR CURRENT NODE AS VISITED
- FOR EVERY NEIGHBORING UNVISITED NODE
  - #PERFORM DFS
  - IF DFS(NEIGHBOR): RETURN TRUE
- #YOU'VE RUN OUT OF NODES
- RETURN FALSE



## DEPTH FIRST SEARCH

- IF YOU'VE FOUND THE END POINT:
  - #SUCCESS!
  - RETURN TRUE
- MARK YOUR CURRENT NODE AS VISITED
- FOR EVERY NEIGHBORING UNVISITED NODE
  - #PERFORM DFS
  - IF DFS(NEIGHBOR): RETURN TRUE
- #YOU'VE RUN OUT OF NODES
- RETURN FALSE



ANSWERING THE CALL TO ADVENTURE



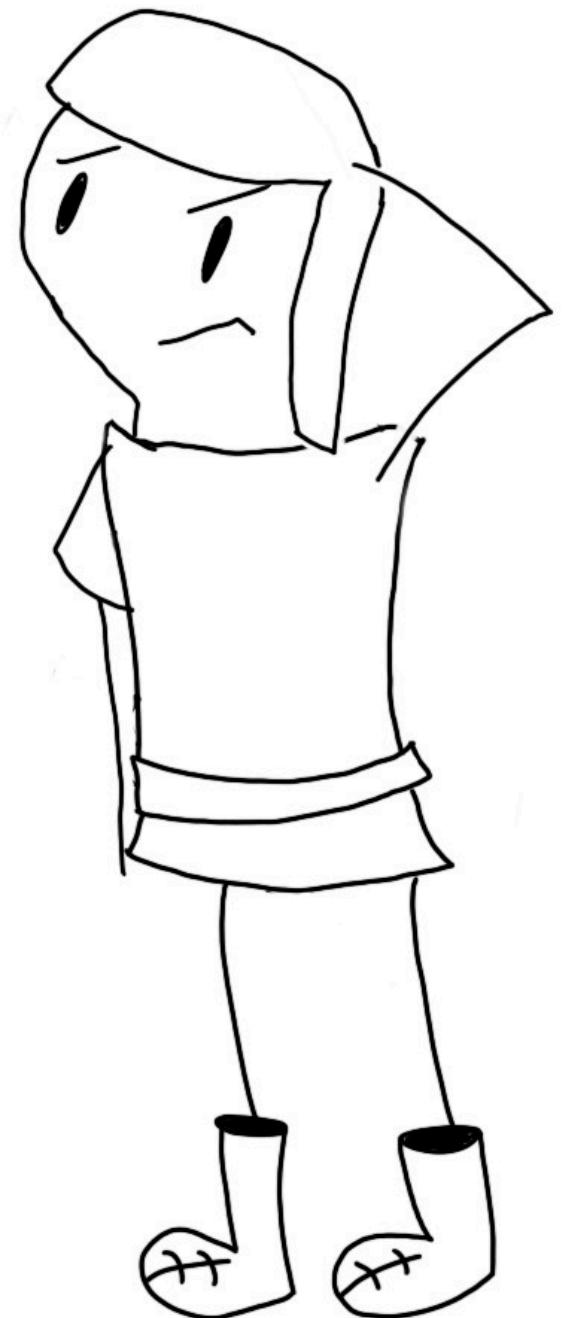
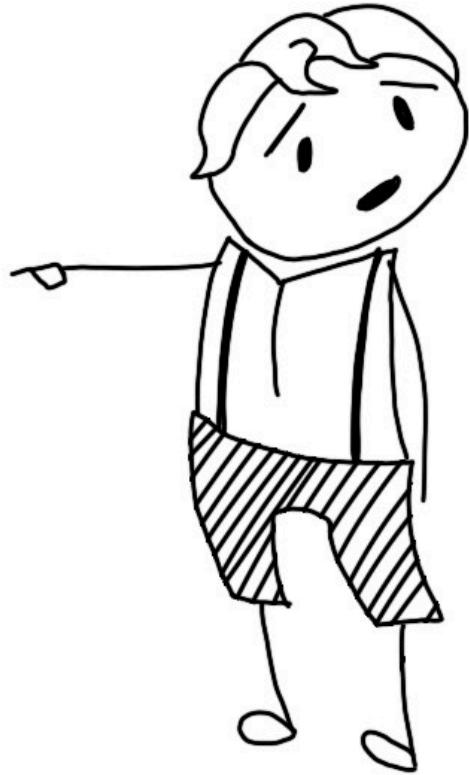
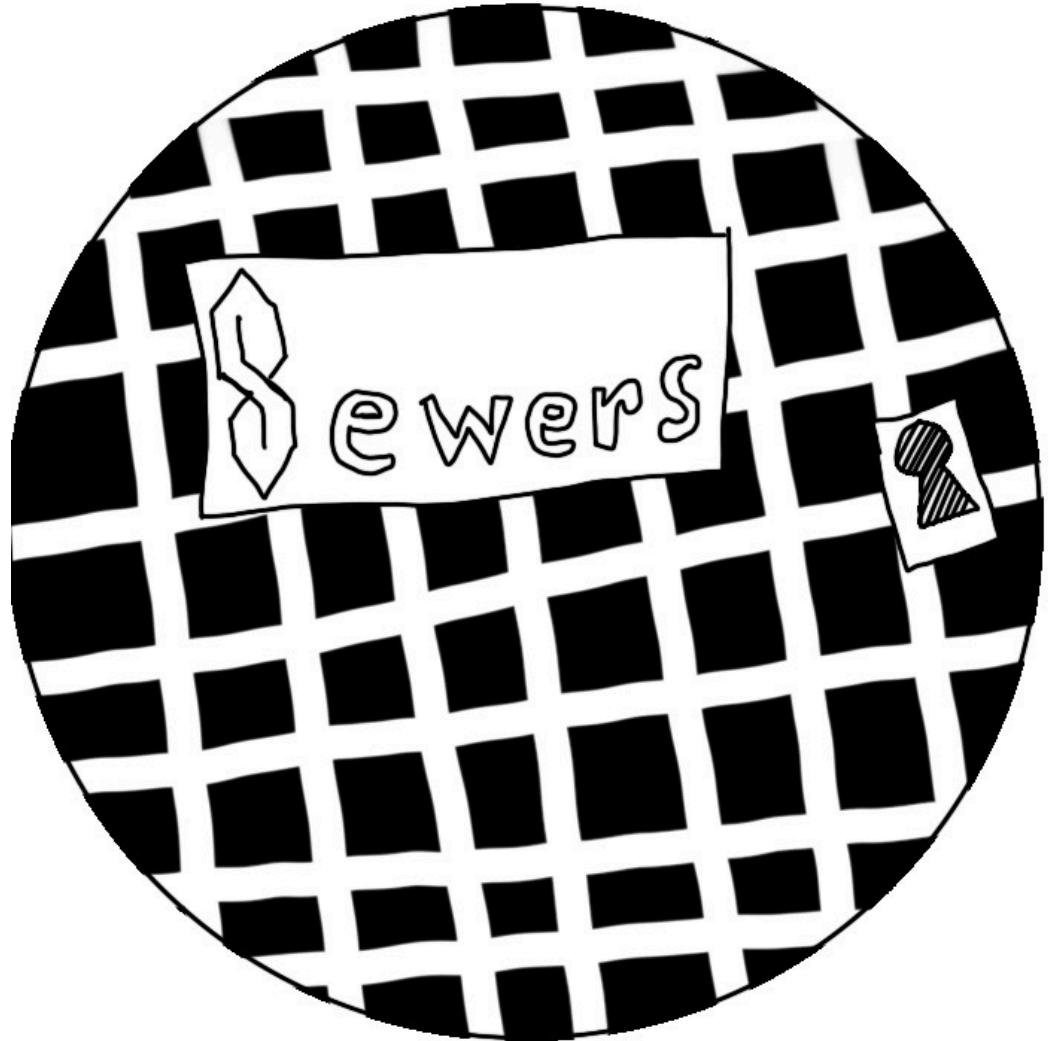
# ANSWERING THE CALL TO ADVENTURE



ANSWERING THE CALL TO ADVENTURE



# ANSWERING THE CALL TO ADVENTURE



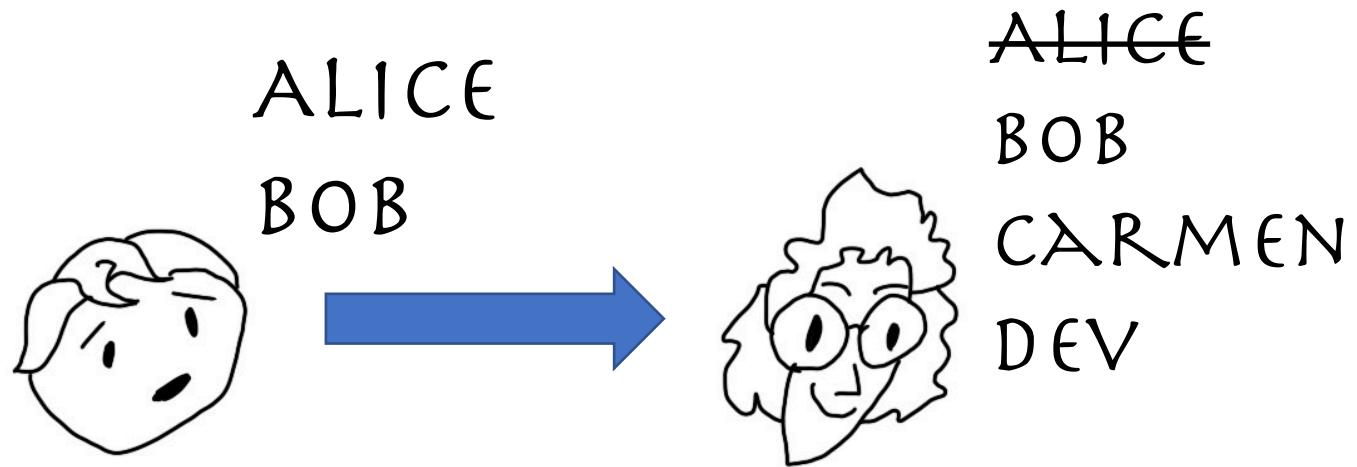
# ANSWERING THE CALL TO ADVENTURE



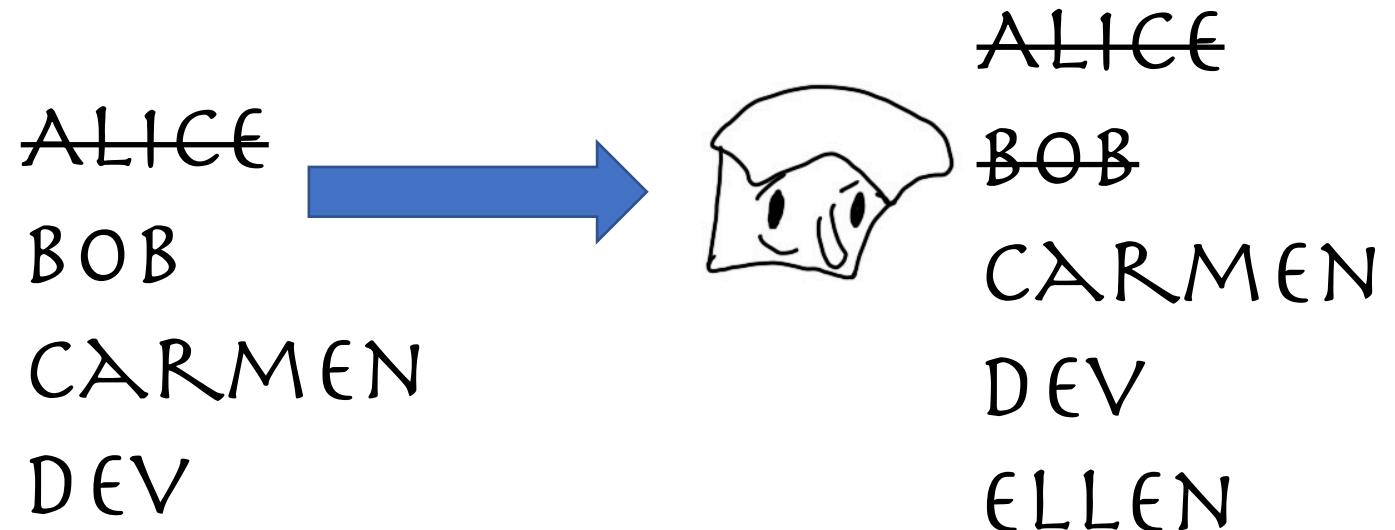
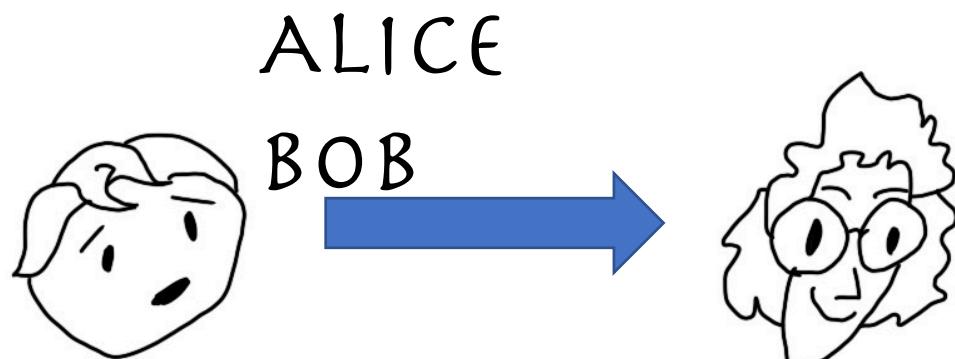
ALICE

BOB

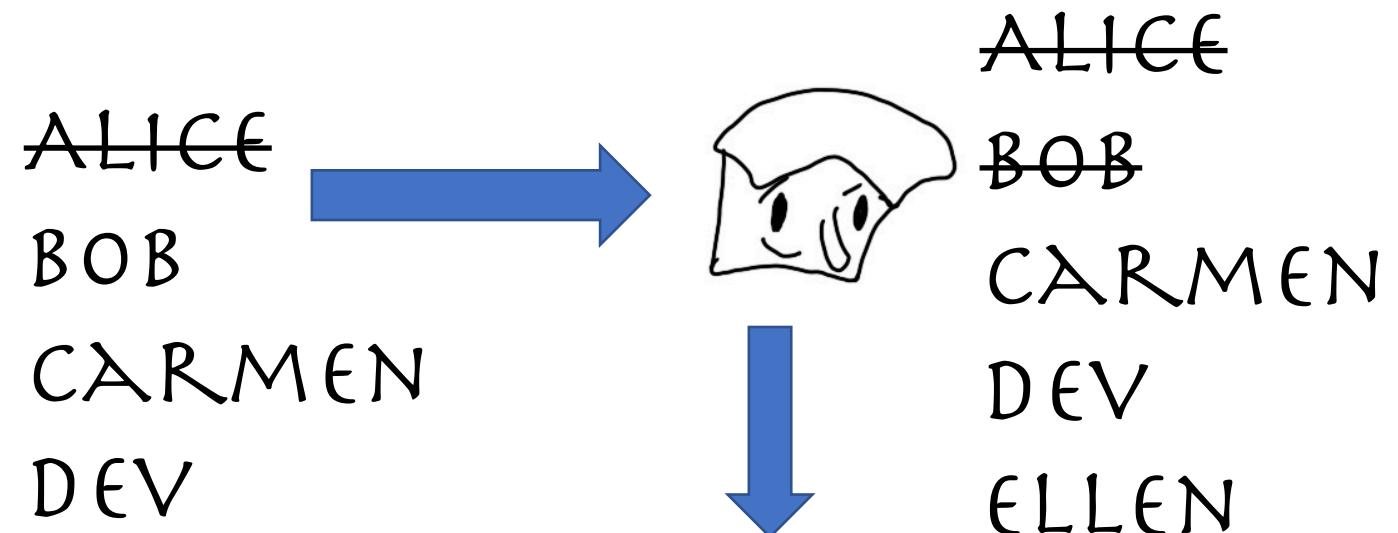
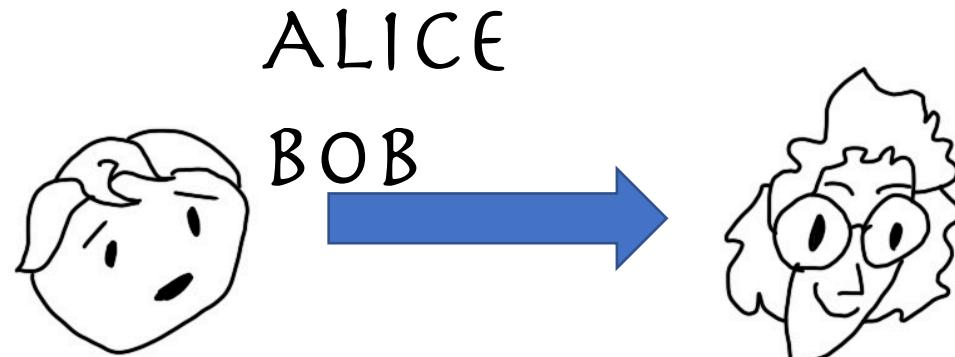
# ANSWERING THE CALL TO ADVENTURE



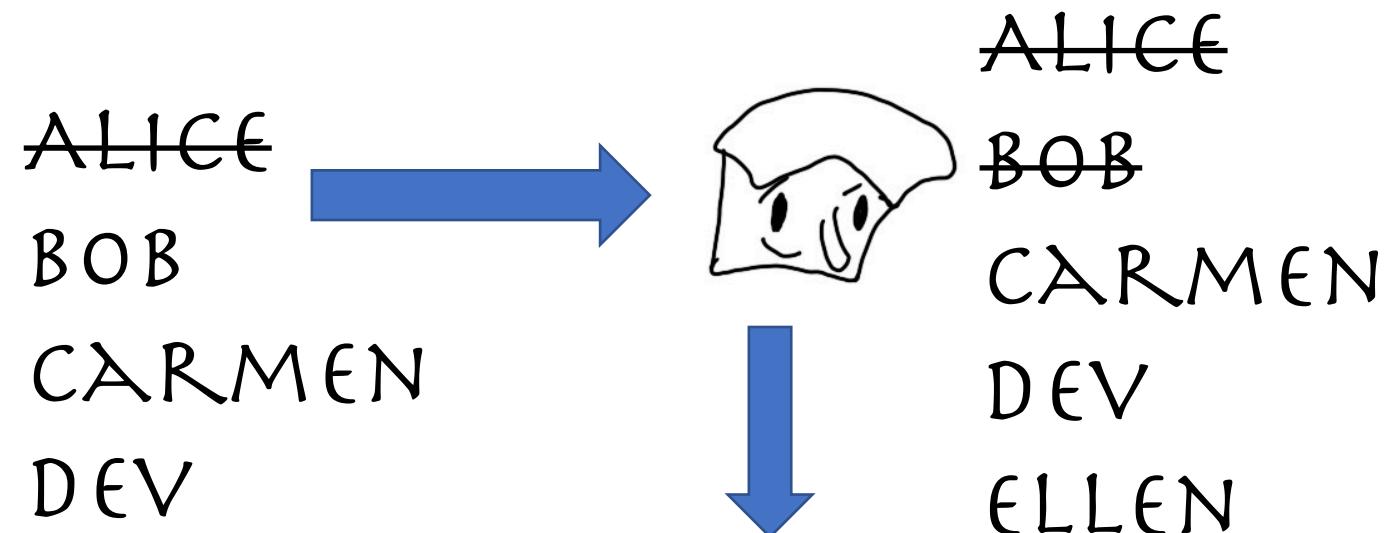
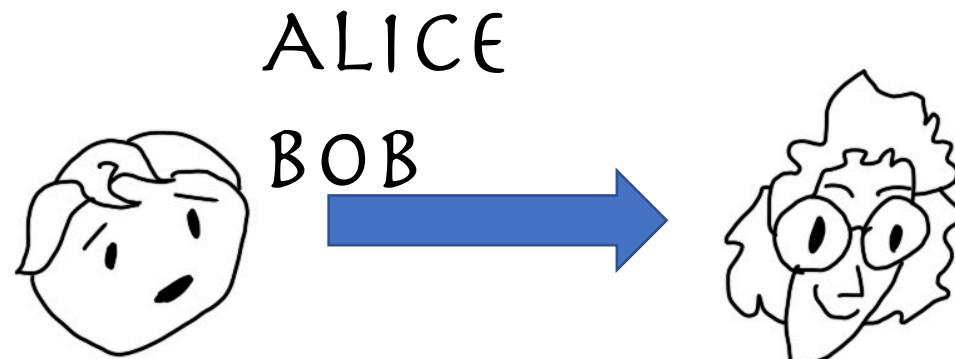
# ANSWERING THE CALL TO ADVENTURE



# ANSWERING THE CALL TO ADVENTURE



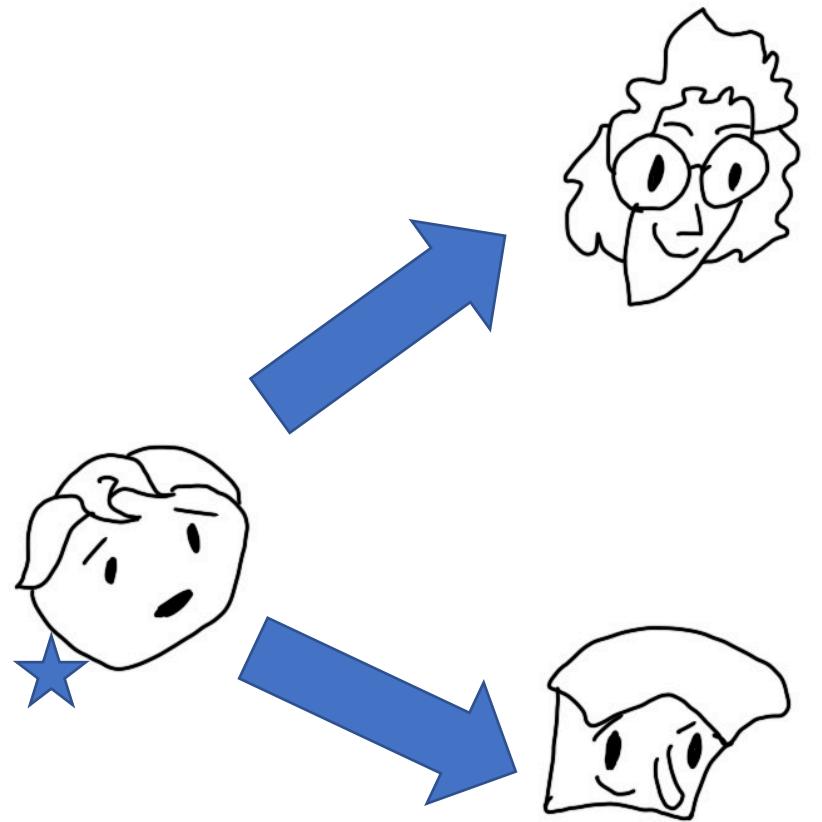
# ANSWERING THE CALL TO ADVENTURE



## ANSWERING THE CALL TO ADVENTURE

- NETWORKS OF PEOPLE CAN BE REPRESENTED AS GRAPHS
- ENTITIES: PEOPLE
- RELATIONSHIP: THE RELATIONSHIP BETWEEN PEOPLE
  - ALICE KNOWS BOB
  - ALICE FOLLOWS BOB
  - ALICE AND BOB ARE FRIENDS WITH EACH OTHER
  - ALICE THINKS BOB HAS THE KEY

# ANSWERING THE CALL TO ADVENTURE

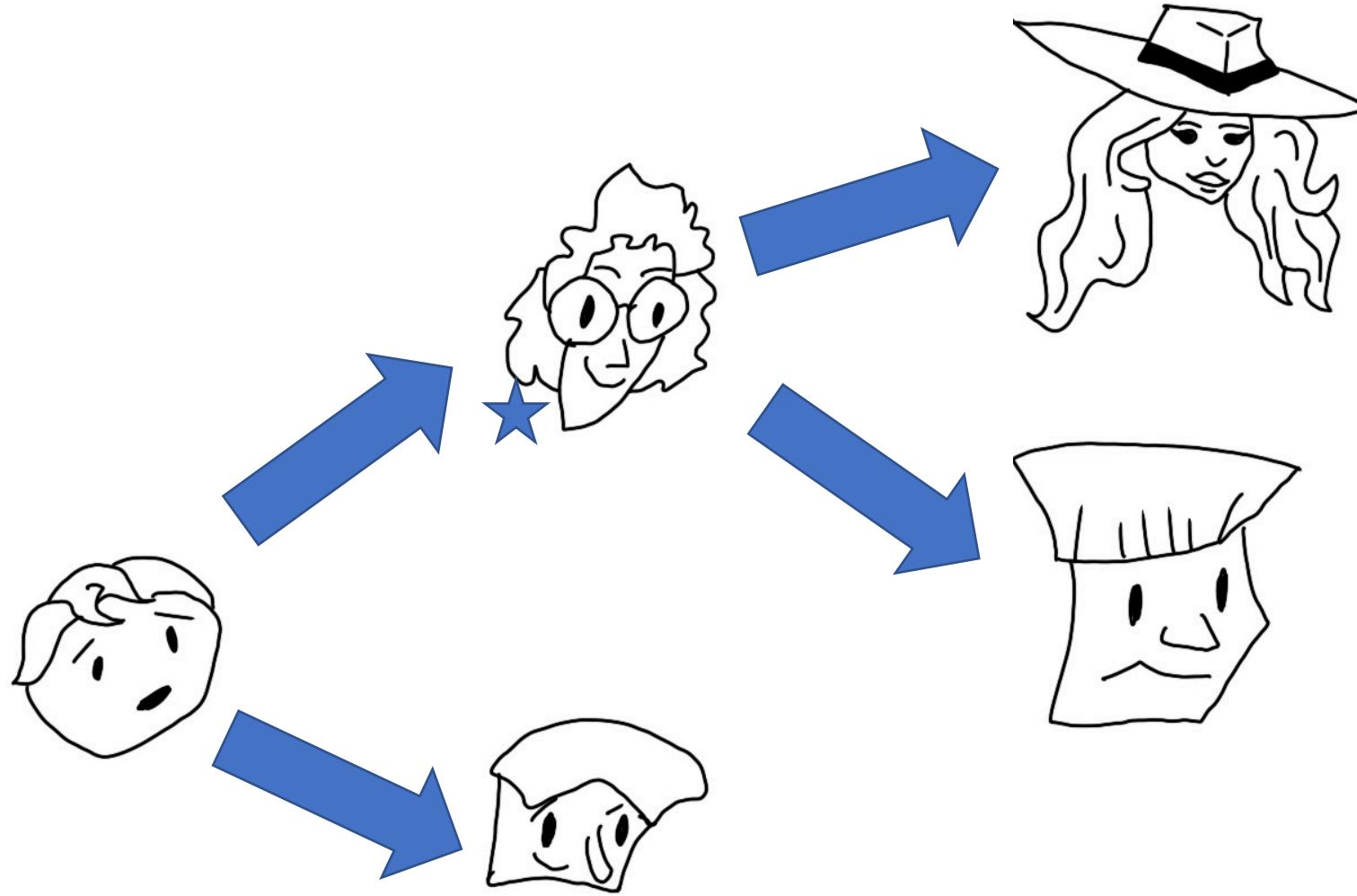


PEOPLE TO ASK

ALICE  
BOB

X THINKS Y HAS THE KY

# ANSWERING THE CALL TO ADVENTURE



PEOPLE TO ASK

ALICE

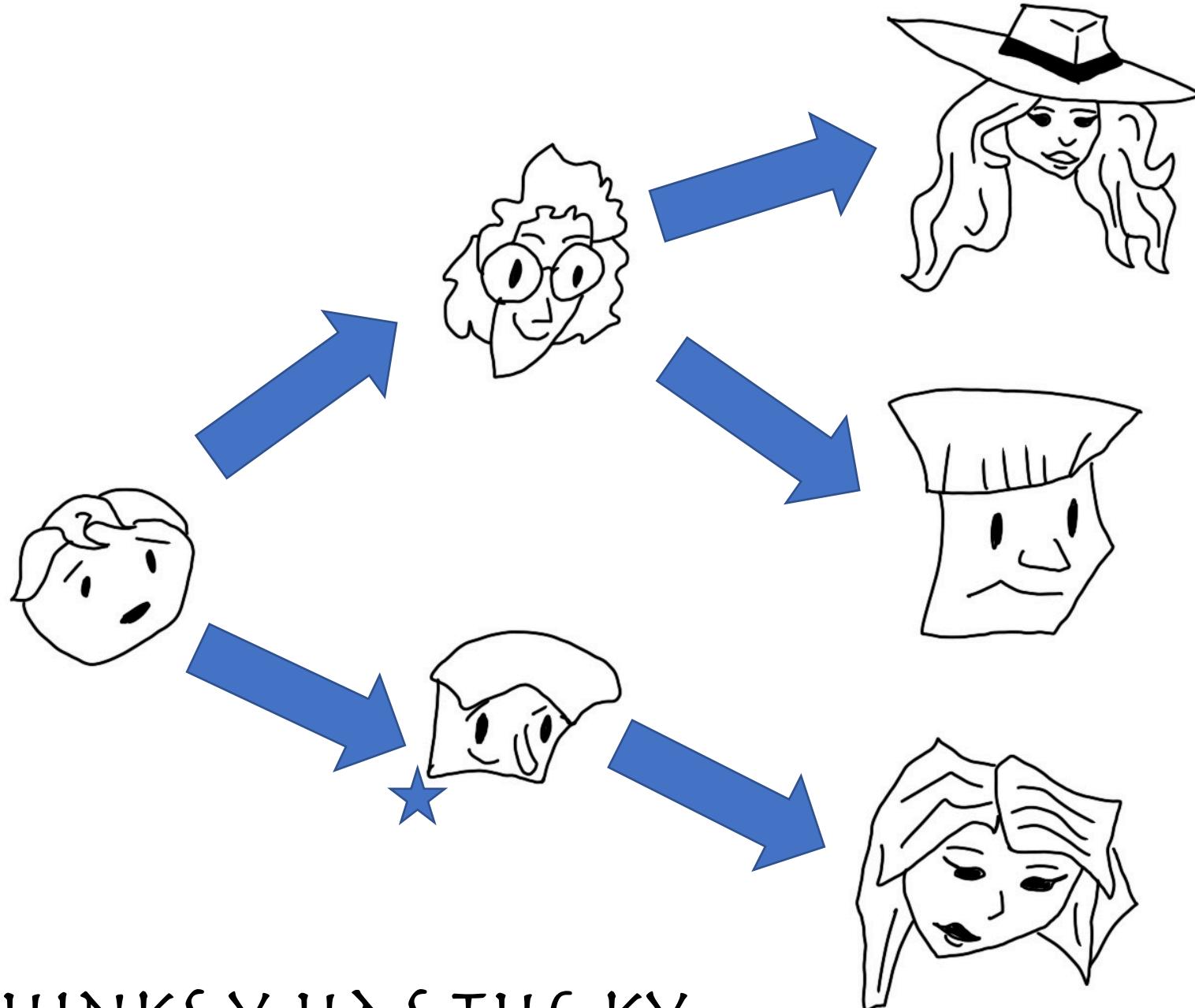
BOB

CARMEN

DAN

X THINKS Y HAS THE KY

# ANSWERING THE CALL TO ADVENTURE



X THINKS Y HAS THE KY

PEOPLE TO ASK

~~ALICE~~

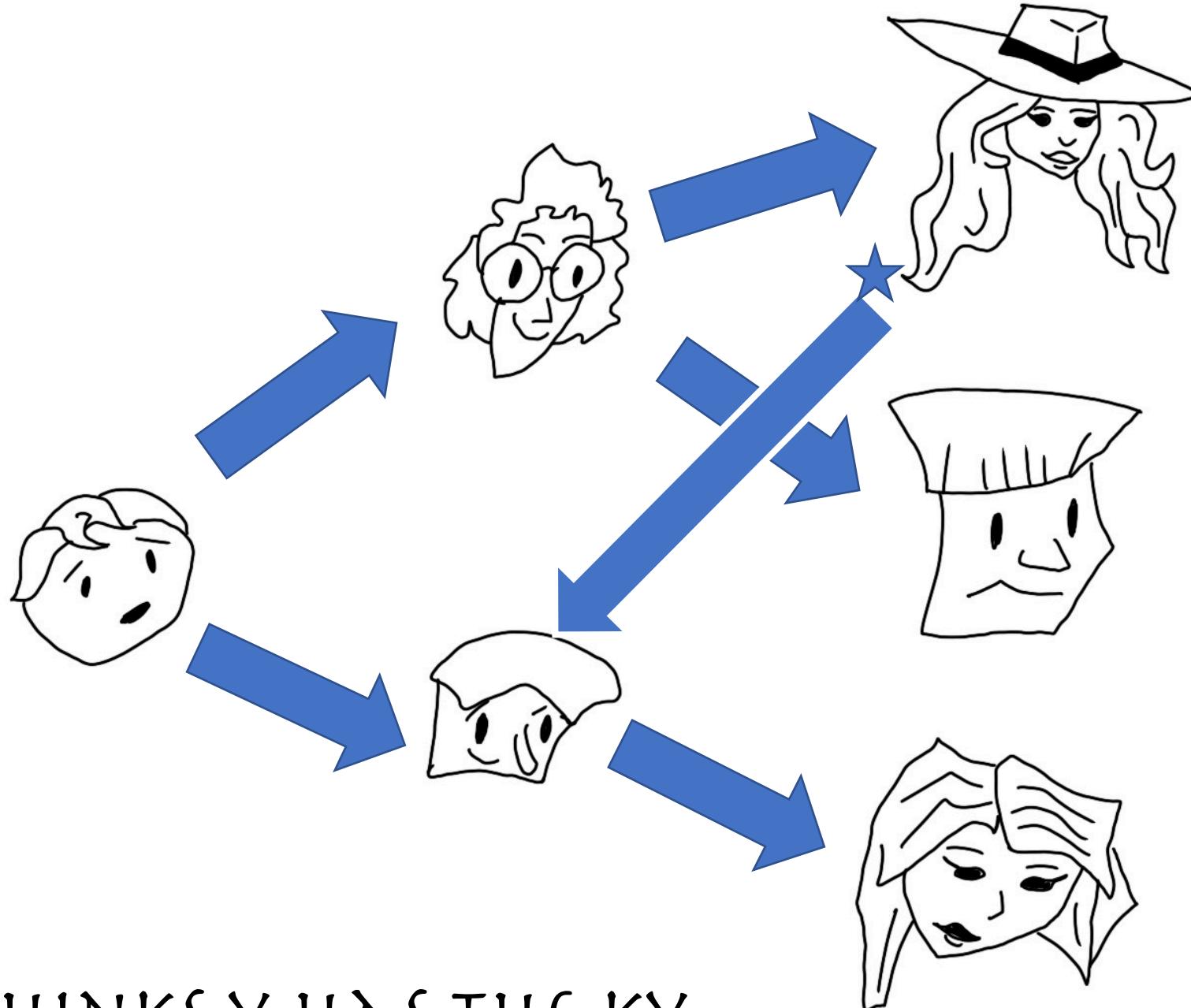
~~BOB~~

CARMEN

DAN

ELLEN

# ANSWERING THE CALL TO ADVENTURE



PEOPLE TO ASK

~~ALICE~~

~~BOB~~

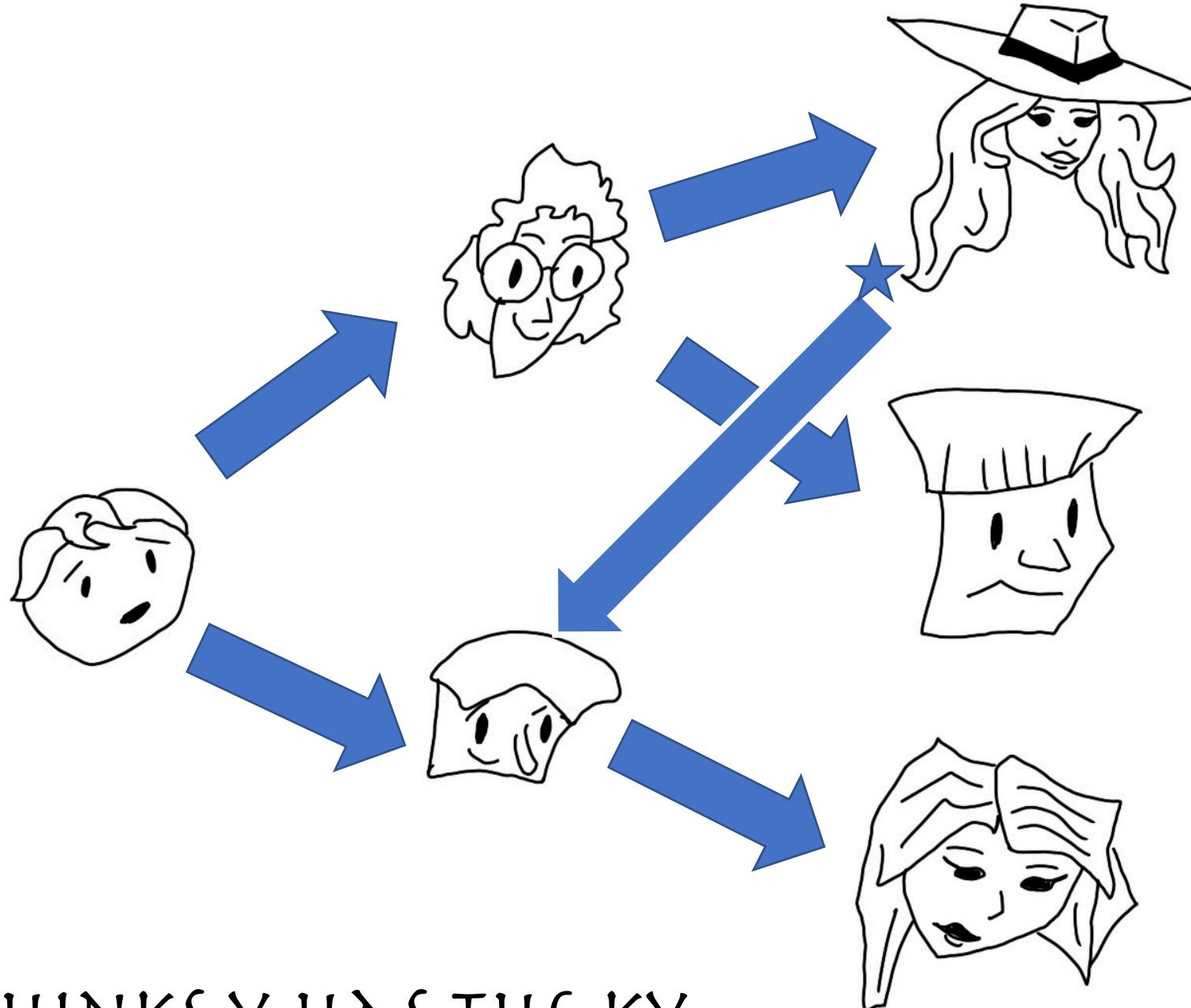
~~CARMEN~~

DAN

ELLEN

BOB

# ANSWERING THE CALL TO ADVENTURE



PEOPLE TO ASK

~~ALICE~~

~~BOB~~

~~CARMEN~~

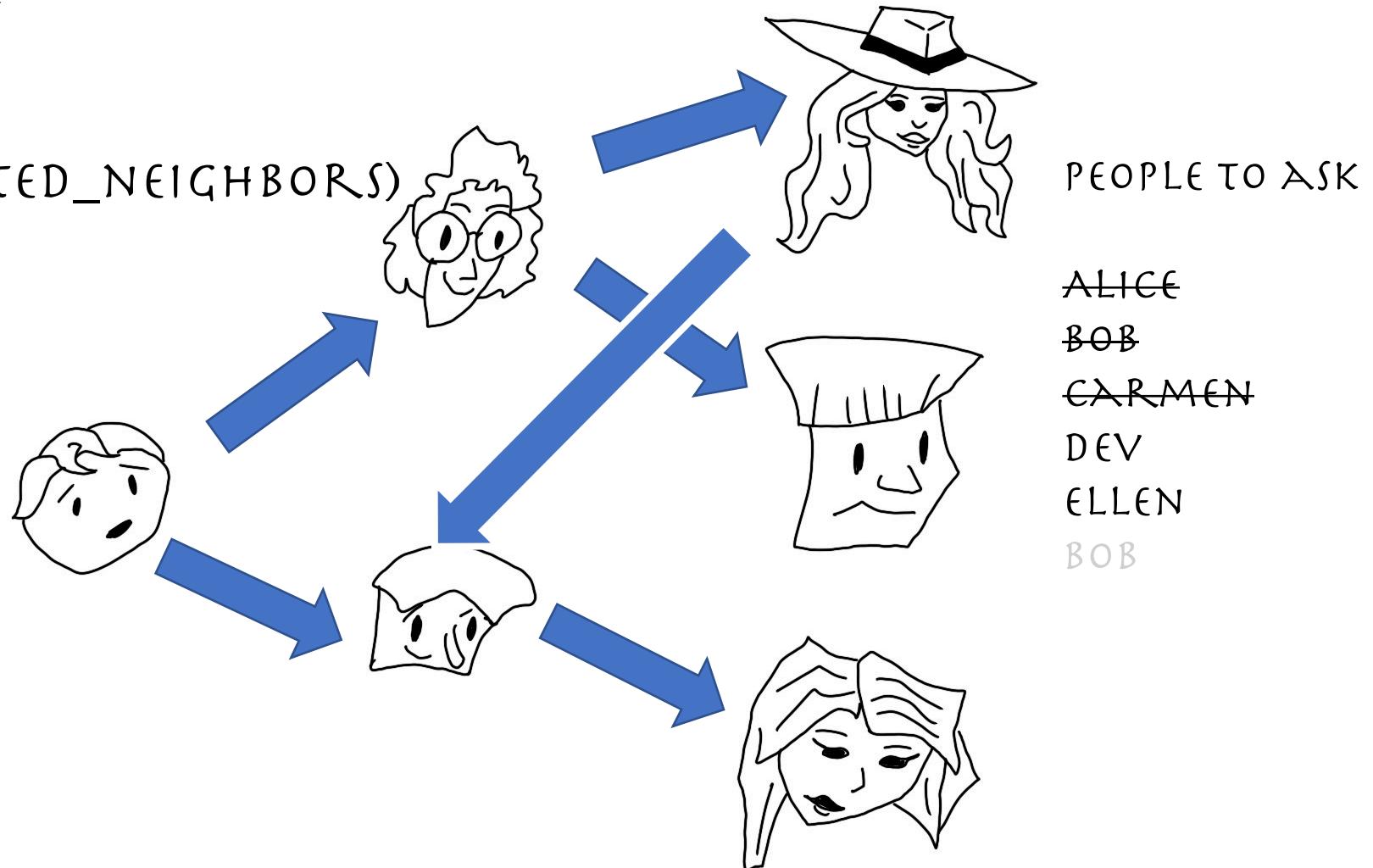
DAN

ELLEN

BOB

# BREADTH FIRST SEARCH

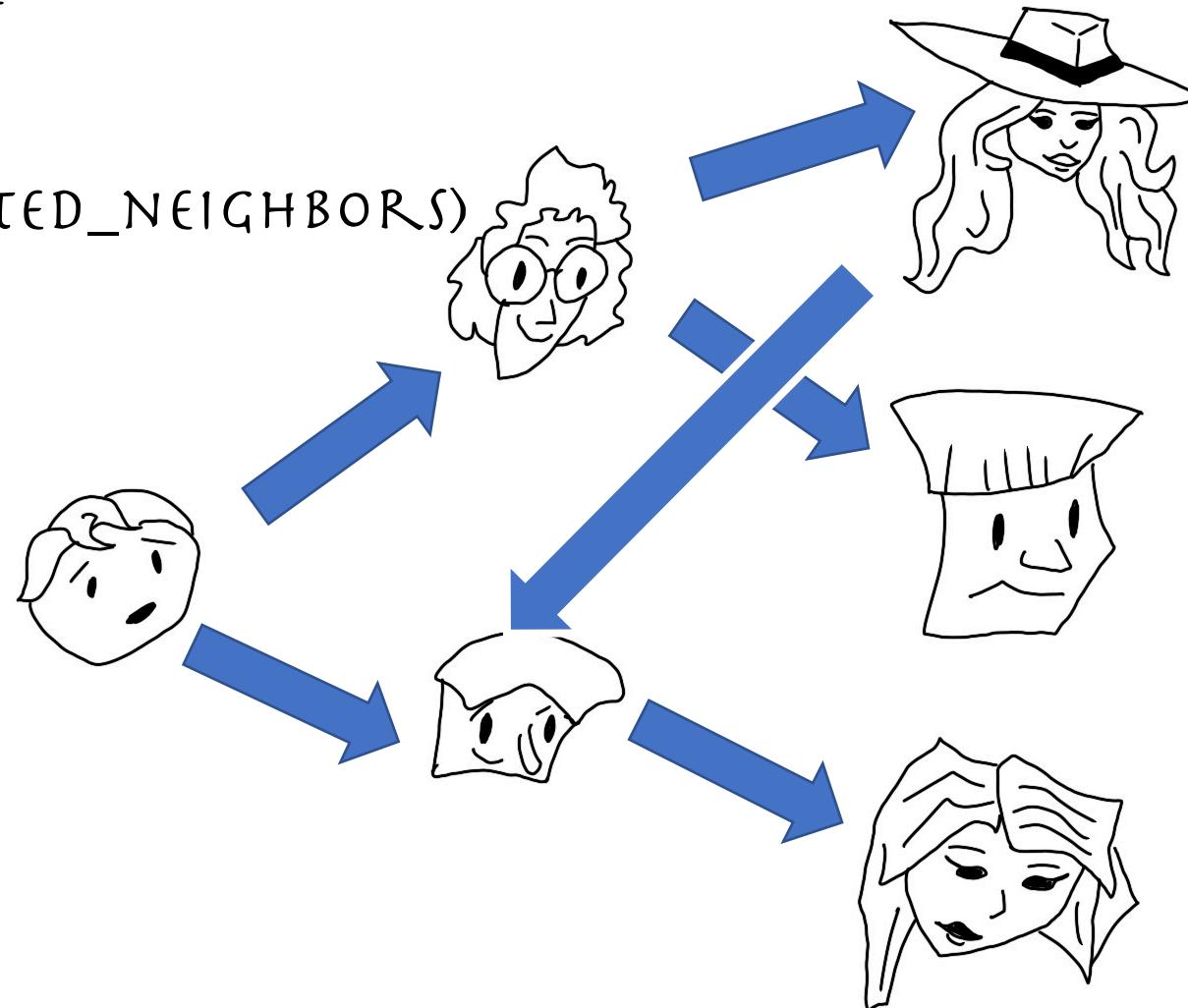
- ADD THE STARTING NODE TO Q (A QUEUE)
- WHILE Q
  - CURRENT = Q.POPELEFT()
  - CURRENT.VISITED = TRUE
  - IF CURRENT == TGT:
    - SUCCESS!
  - Q.APPEND(CURR.UNVISITED\_NEIGHBORS)



# BREADTH FIRST SEARCH

- ADD THE STARTING NODE TO Q (A QUEUE)
- WHILE Q
  - CURRENT = Q.POPELEFT()
  - CURRENT.VISITED = TRUE
  - IF CURRENT == TGT:
    - SUCCESS!
  - Q.APPEND(CURR.UNVISITED\_NEIGHBORS)

QUEUE:  
KID

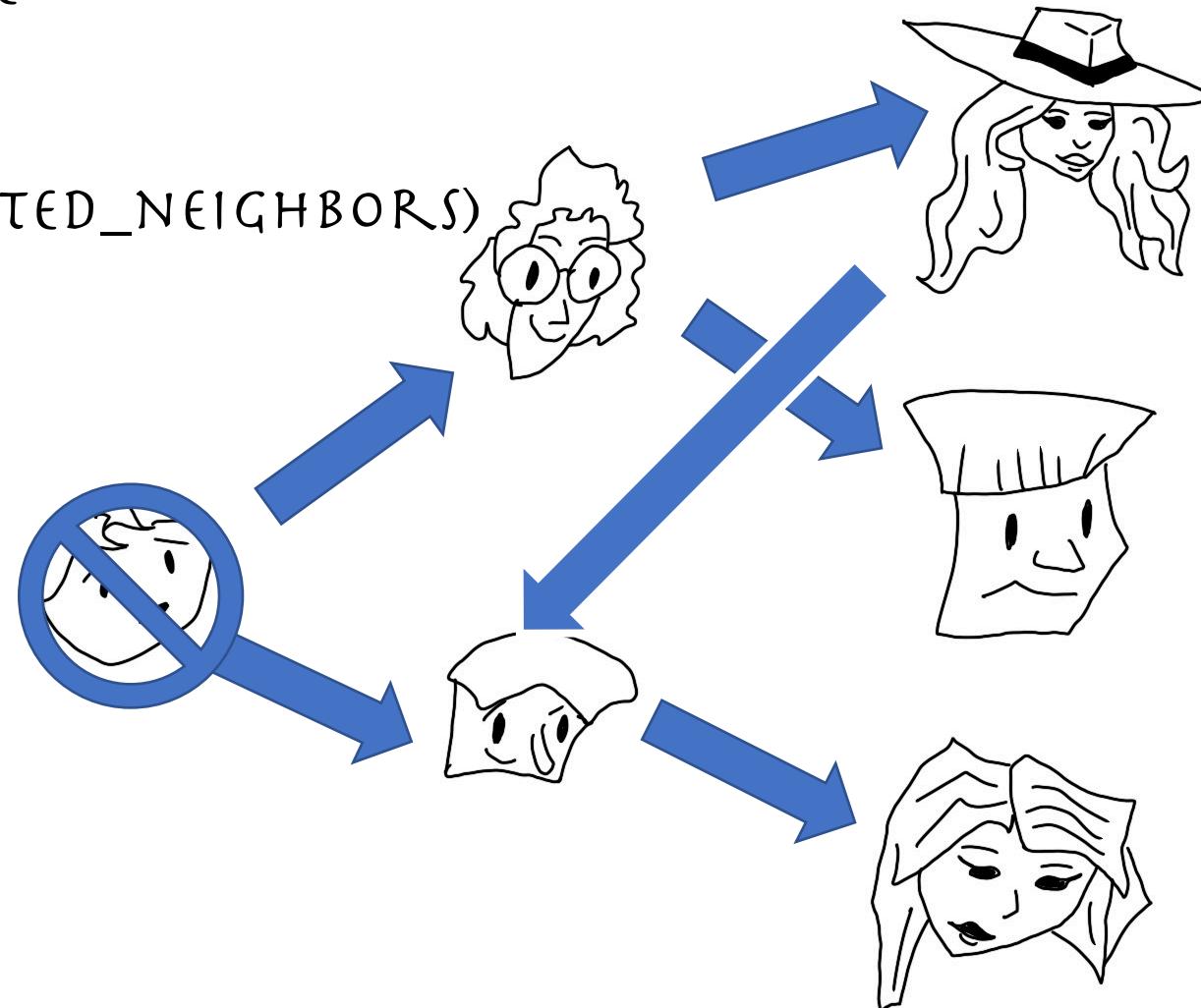


# BREADTH FIRST SEARCH

- ADD THE STARTING NODE TO Q (A QUEUE)
- WHILE Q
  - CURRENT = Q.POPELEFT()
  - CURRENT.VISITED = TRUE
  - IF CURRENT == TGT:
    - SUCCESS!
  - Q.APPEND(CURR.UNVISITED\_NEIGHBORS)

CURRENT: KID

QUEUE:  
<EMPTY>



# BREADTH FIRST SEARCH

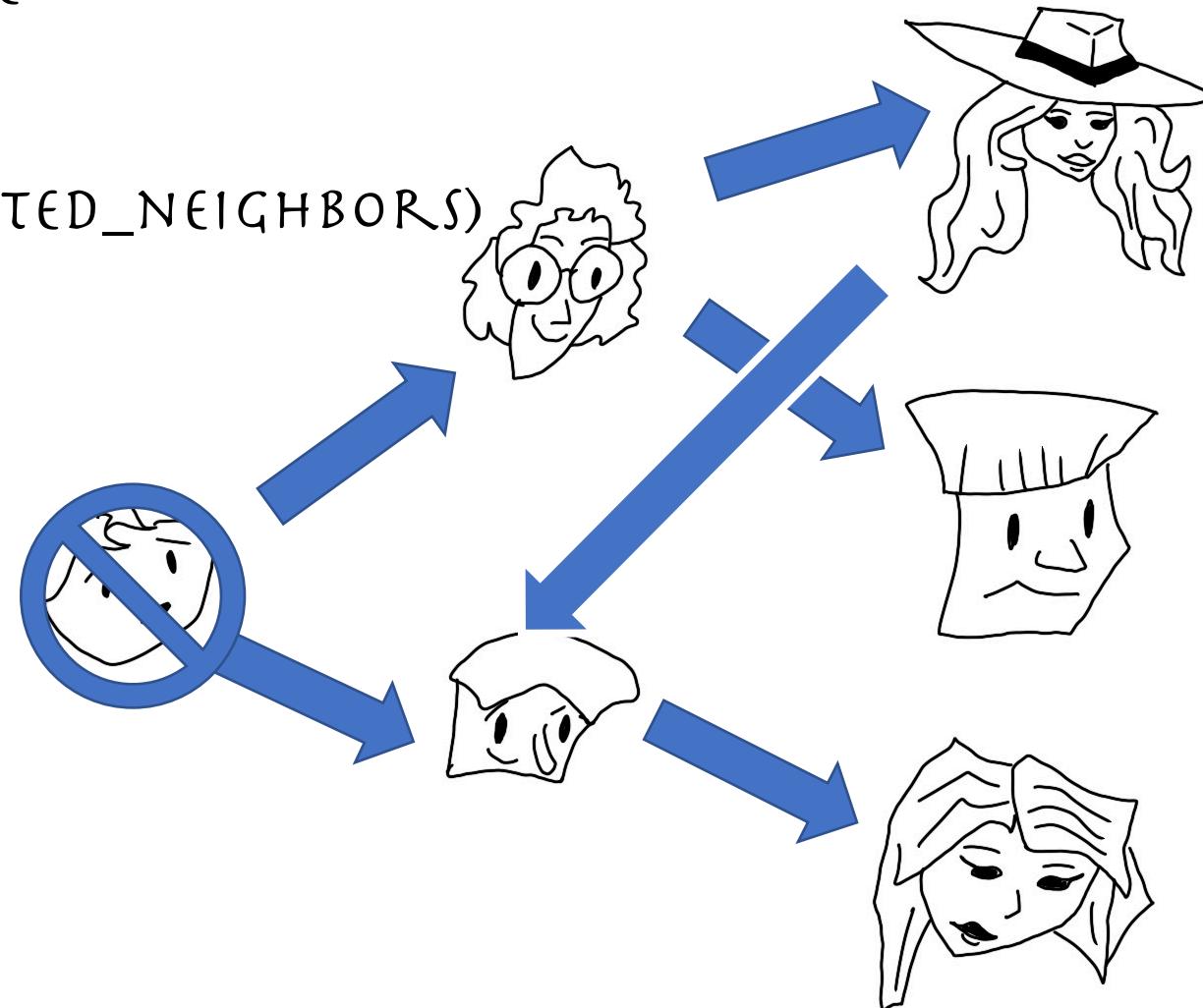
- ADD THE STARTING NODE TO Q (A QUEUE)
- WHILE Q
  - CURRENT = Q.POPELEFT()
  - CURRENT.VISITED = TRUE
  - IF CURRENT == TGT:
    - SUCCESS!
  - Q.APPEND(CURR.UNVISITED\_NEIGHBORS)

CURRENT: KID

QUEUE:

ALICE

BOB

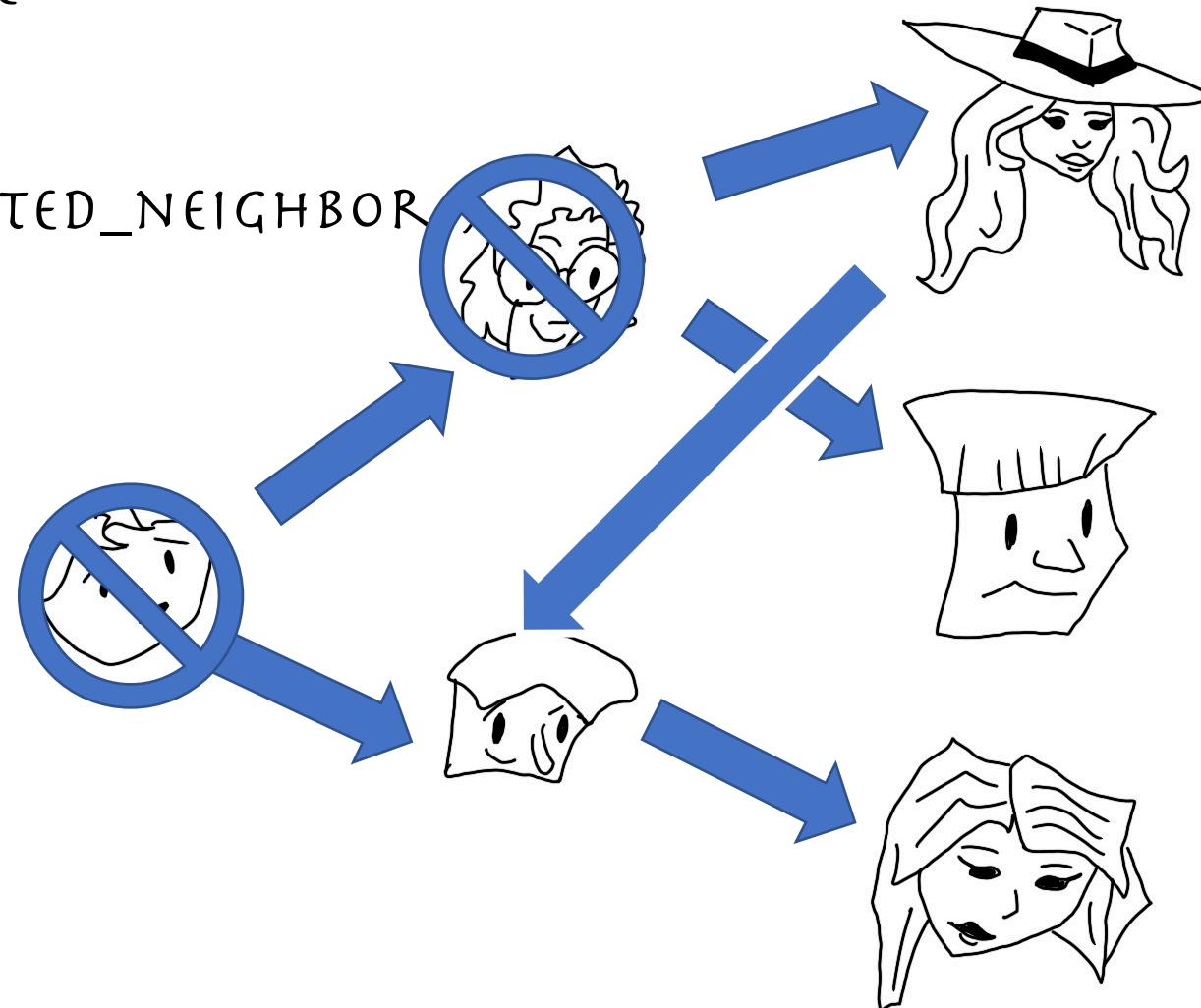


# BREADTH FIRST SEARCH

- ADD THE STARTING NODE TO Q (A QUEUE)
- WHILE Q
  - CURRENT = Q.POPELEFT()
  - CURRENT.VISITED = TRUE
  - IF CURRENT == TGT:
    - SUCCESS!
  - Q.APPEND(CURR.UNVISITED\_NEIGHBOR)

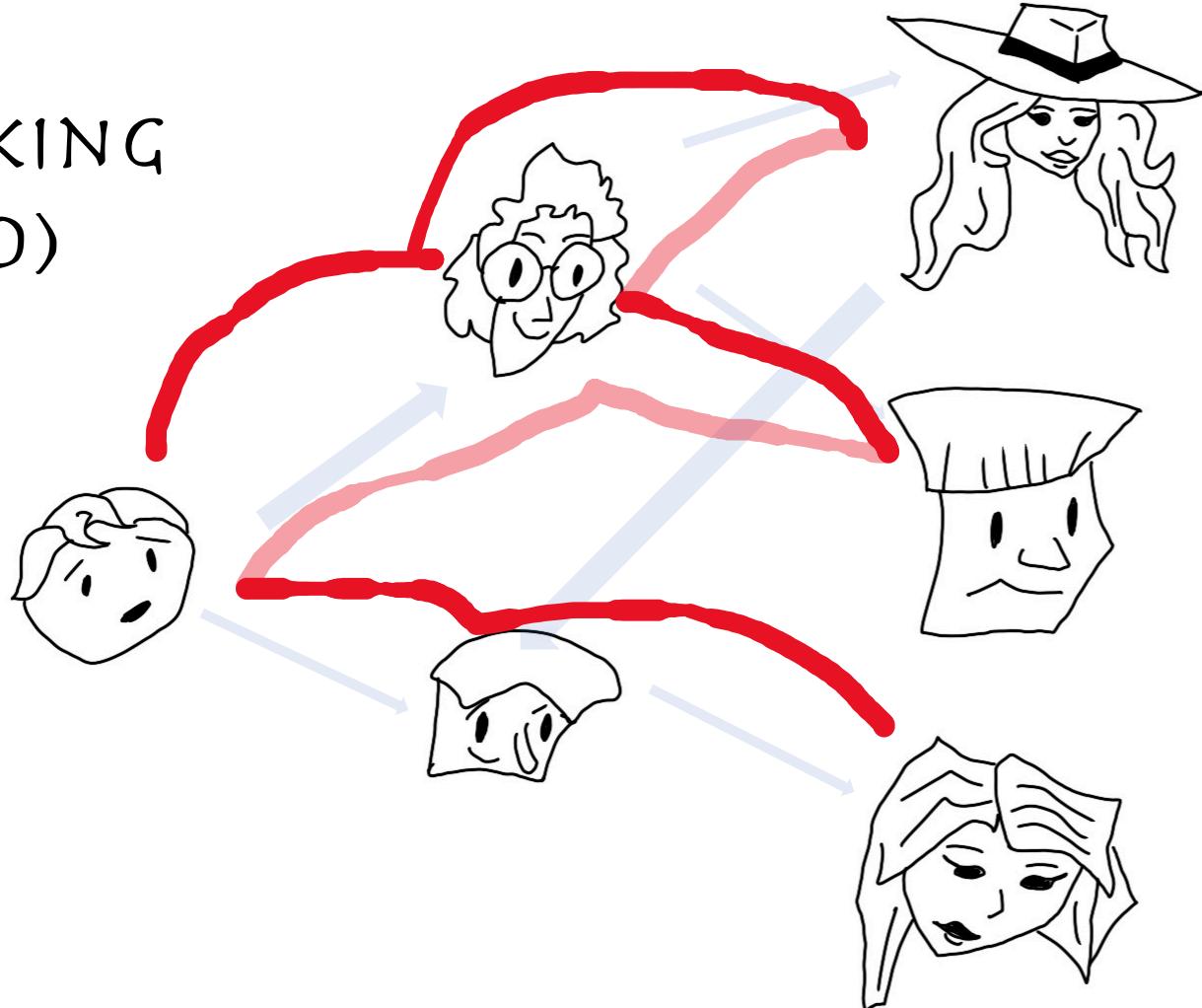
CURRENT: ALICE

QUEUE:  
BOB



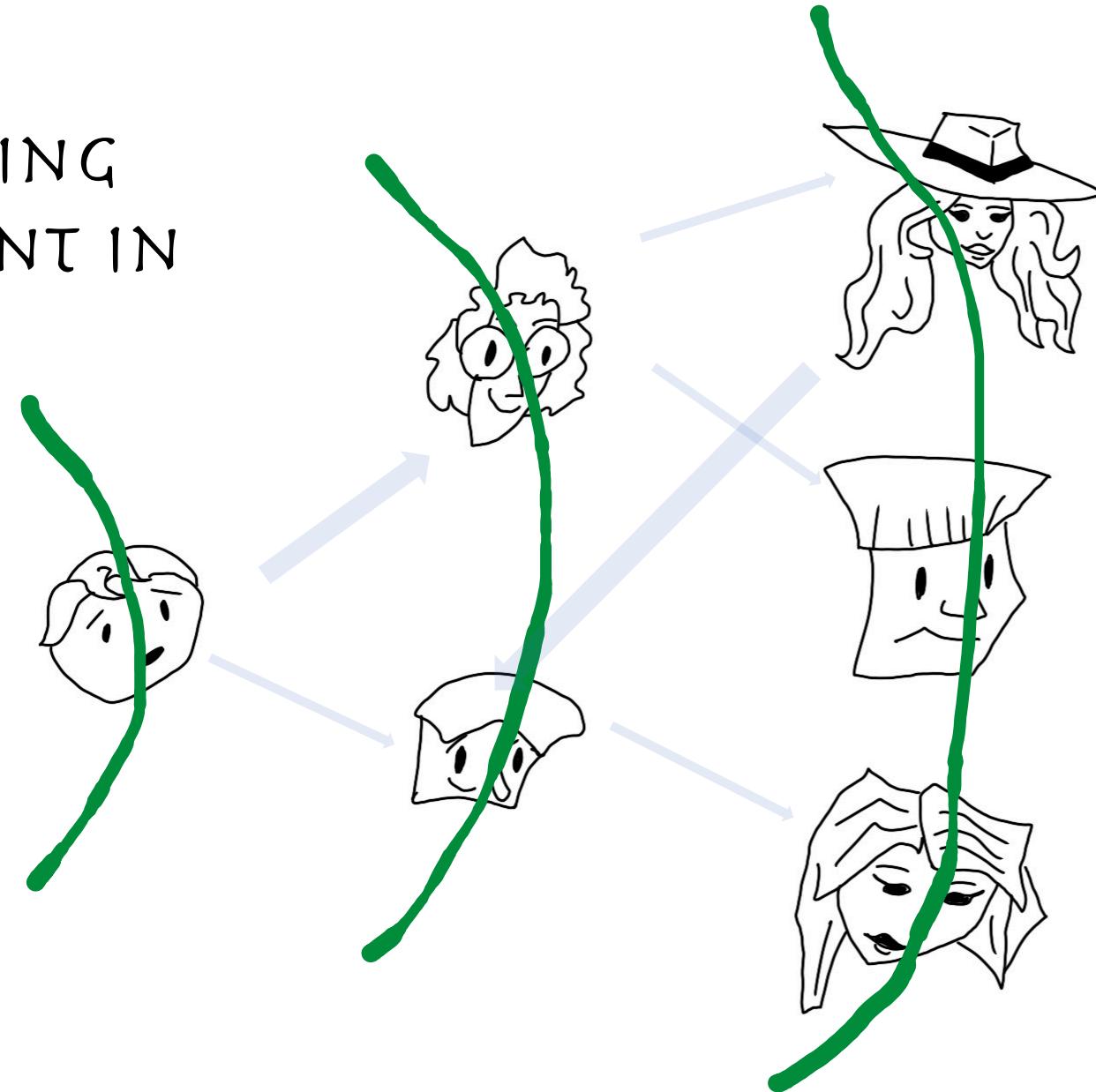
# A QUICK NOTE ON BFS VS DFS

DFS: RED  
(BACKTRACKING  
IN LIGHT RED)



# A QUICK NOTE ON BFS VS DFS

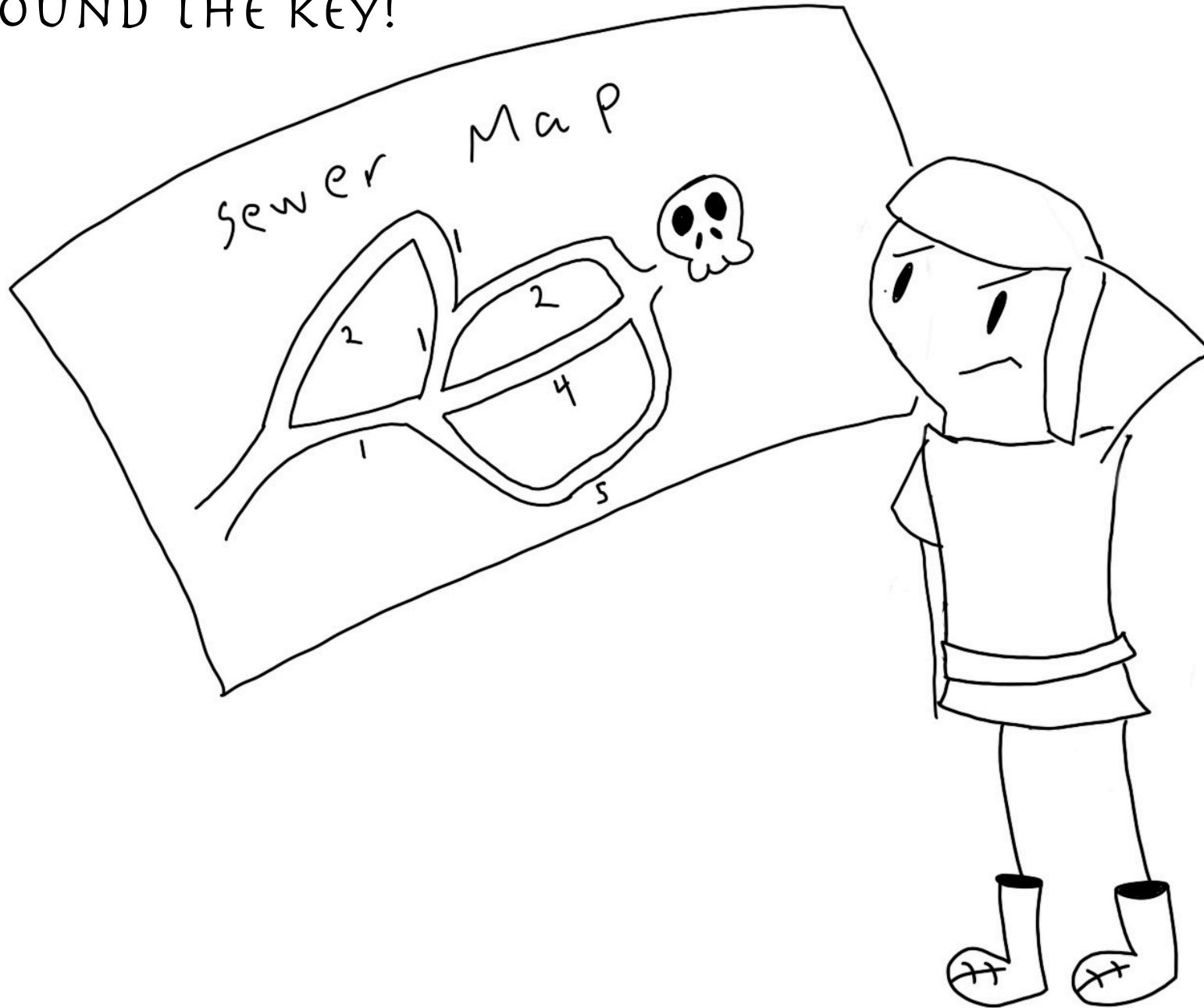
BFS: EXPANDING  
SEARCH FRONT IN  
GREEN



YOU FOUND THE KEY!

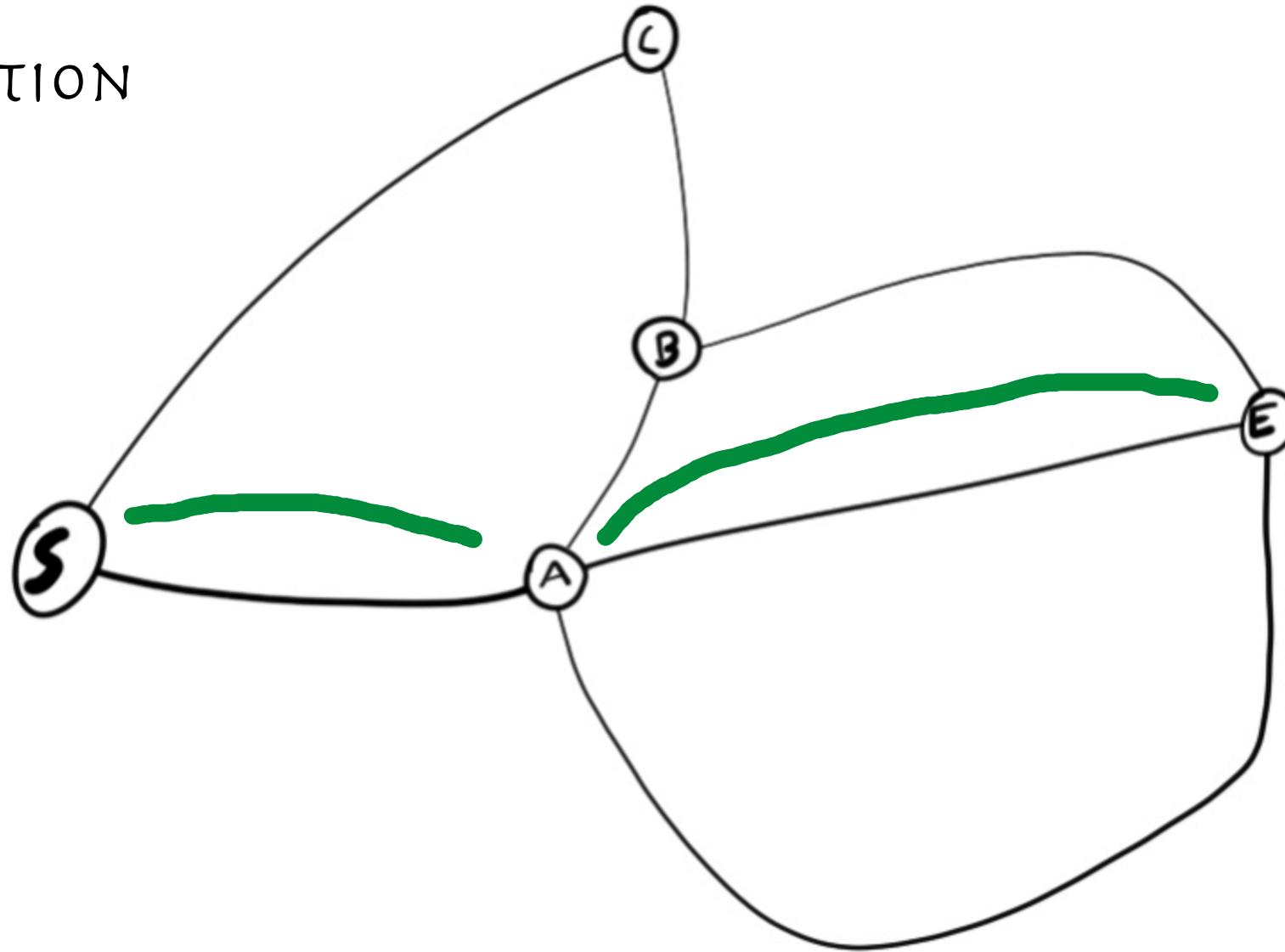


YOU FOUND THE KEY!

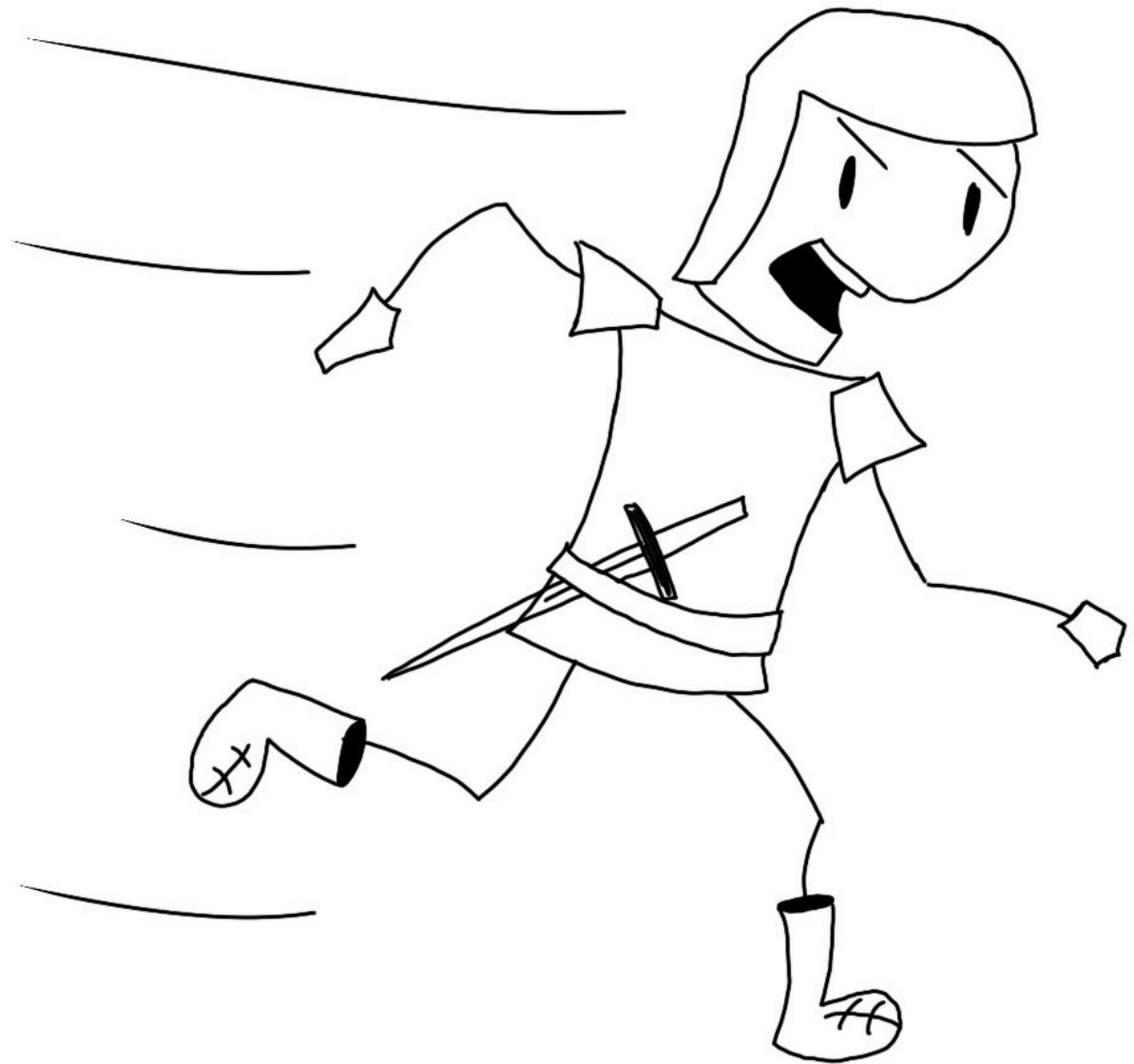
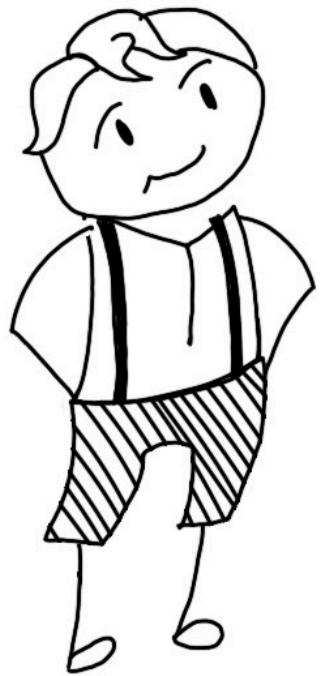


MAP -> GRAPH

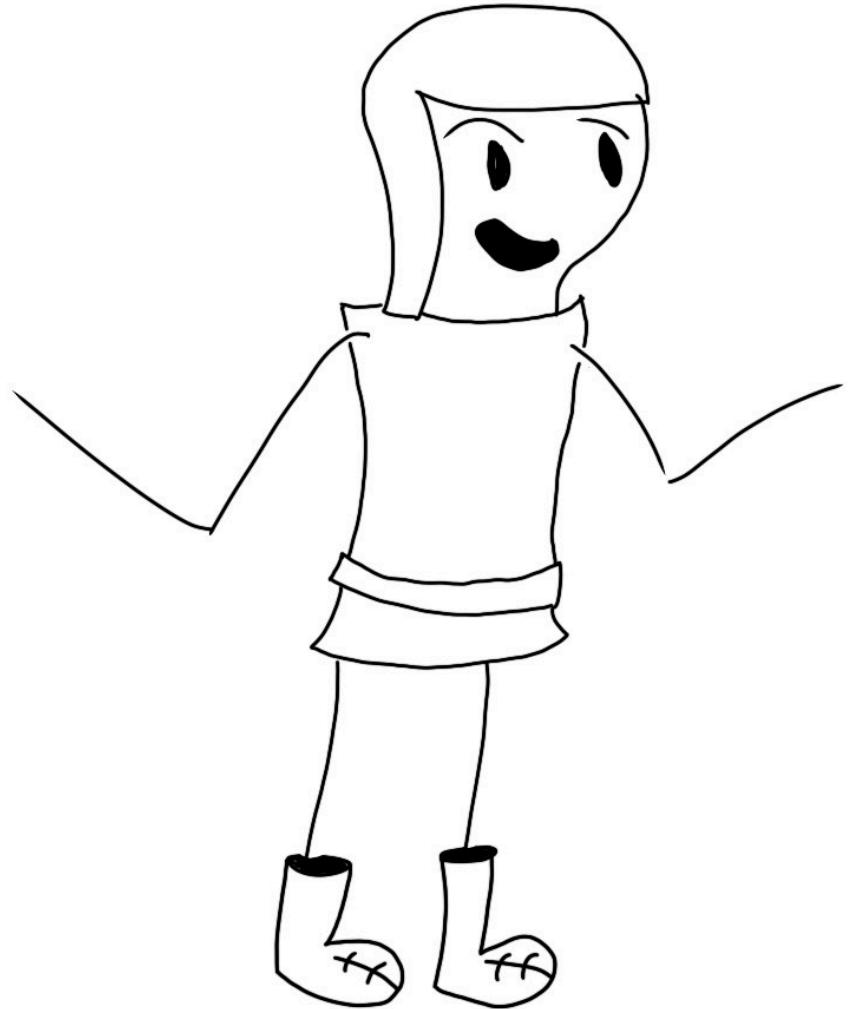
BFS SOLUTION



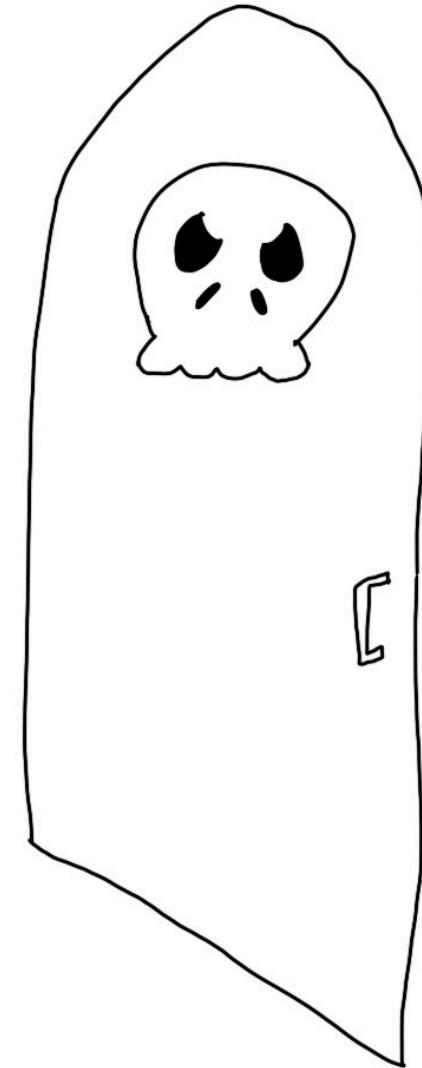
YOU FOUND THE KEY!



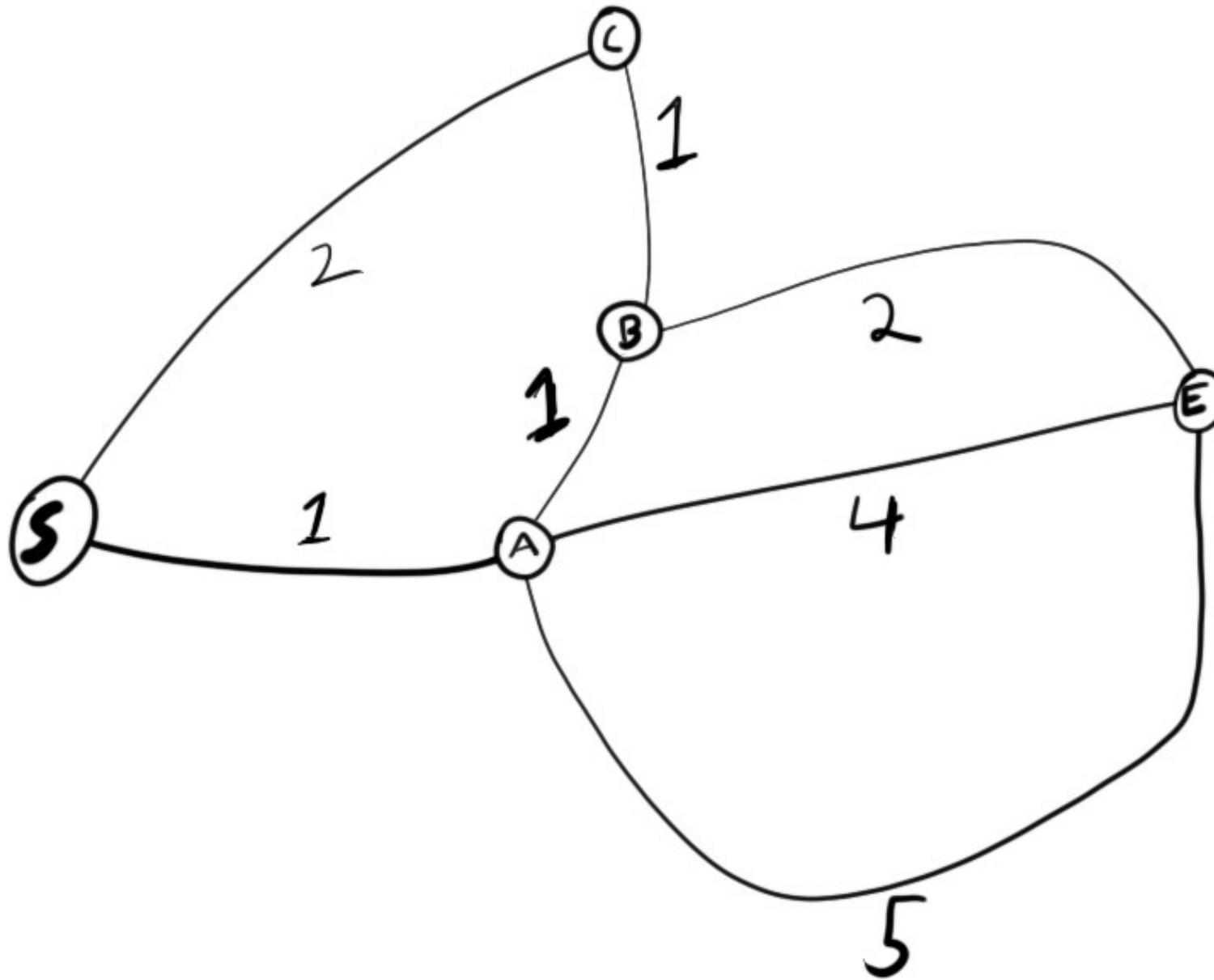
YOU FOUND THE KEY!



A WIZARD  
NAMED  
DIJKSTRA



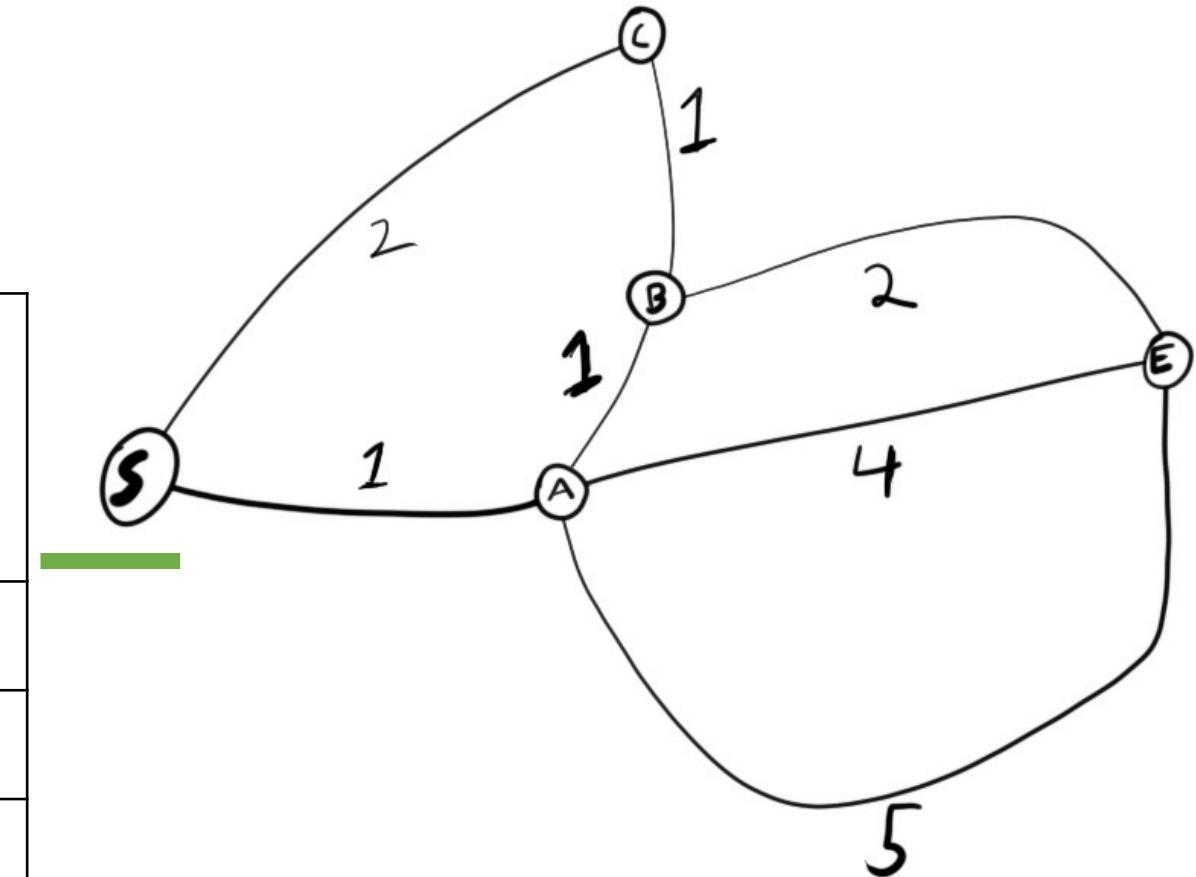
YOU FOUND THE KEY!



# DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM  
PREVIOUS NODES

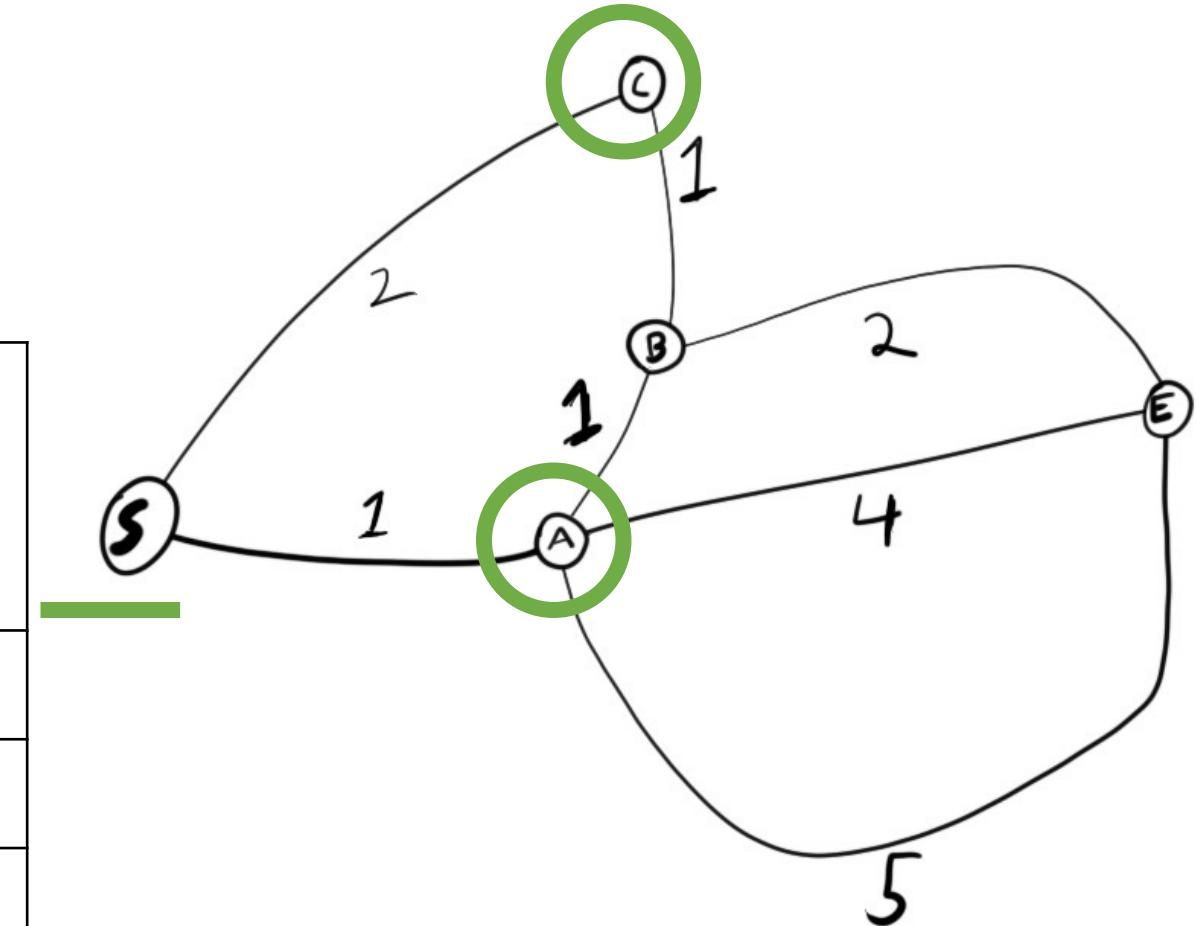
NODES	DISTANCE FROM START	PREVIOUS
START	0	$\emptyset$
A	$\infty$	$\emptyset$
B	$\infty$	$\emptyset$
C	$\infty$	$\emptyset$
END	$\infty$	$\emptyset$



# DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM  
PREVIOUS NODES

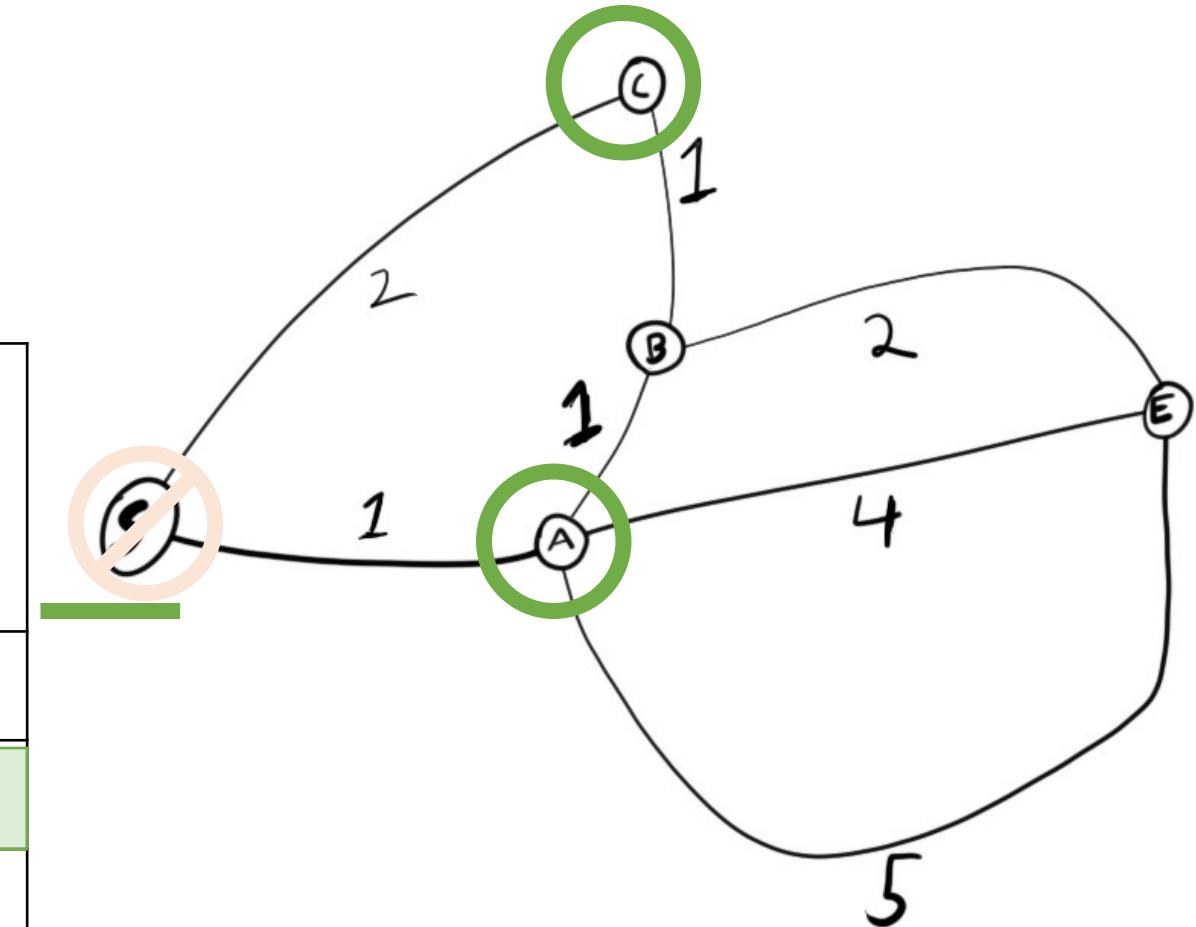
NODES	DISTANCE FROM START	PREVIOUS
START	0	$\emptyset$
A	$\infty$	$\emptyset$
B	$\infty$	$\emptyset$
C	$\infty$	$\emptyset$
END	$\infty$	$\emptyset$



# DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM  
PREVIOUS NODES

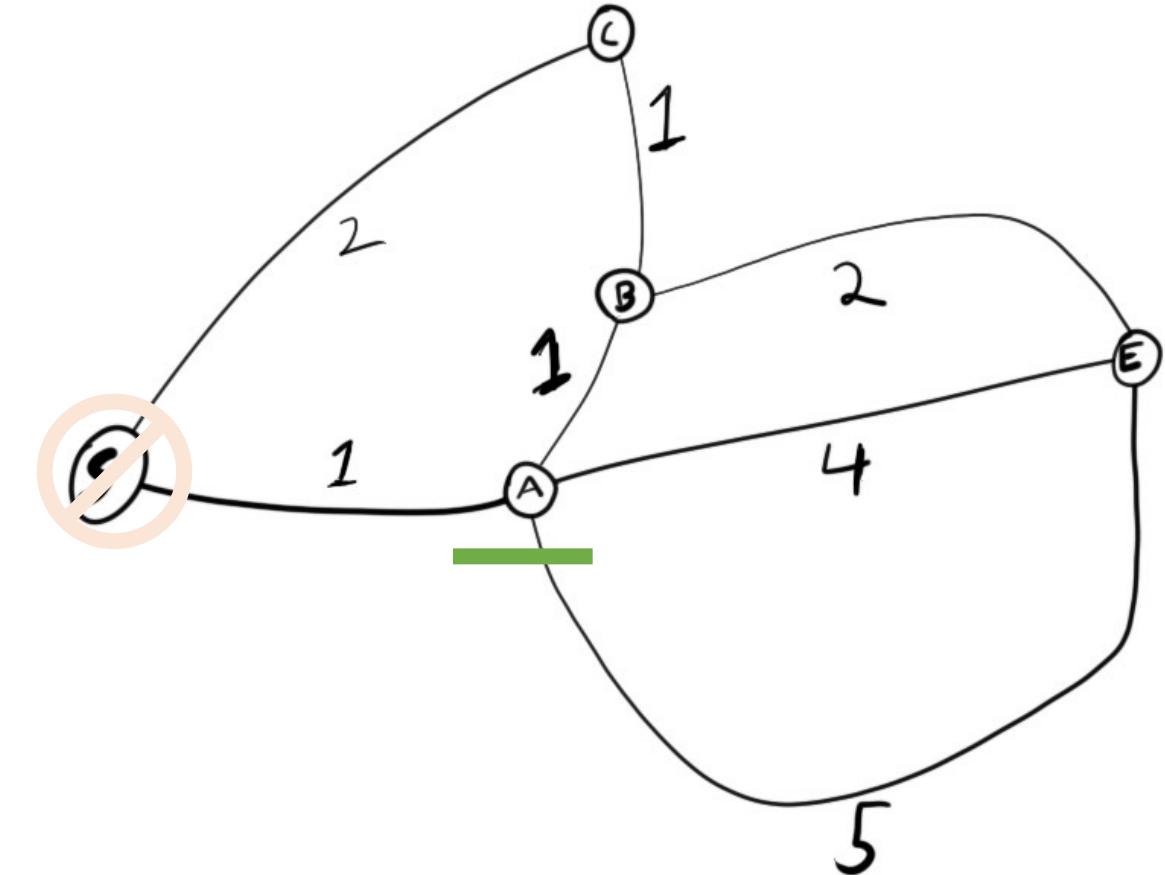
NODES	DISTANCE FROM START	PREVIOUS
START	0	$\emptyset$
A	1	START
B	$\infty$	$\emptyset$
C	2	START
END	$\infty$	$\emptyset$



# DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM  
PREVIOUS NODES

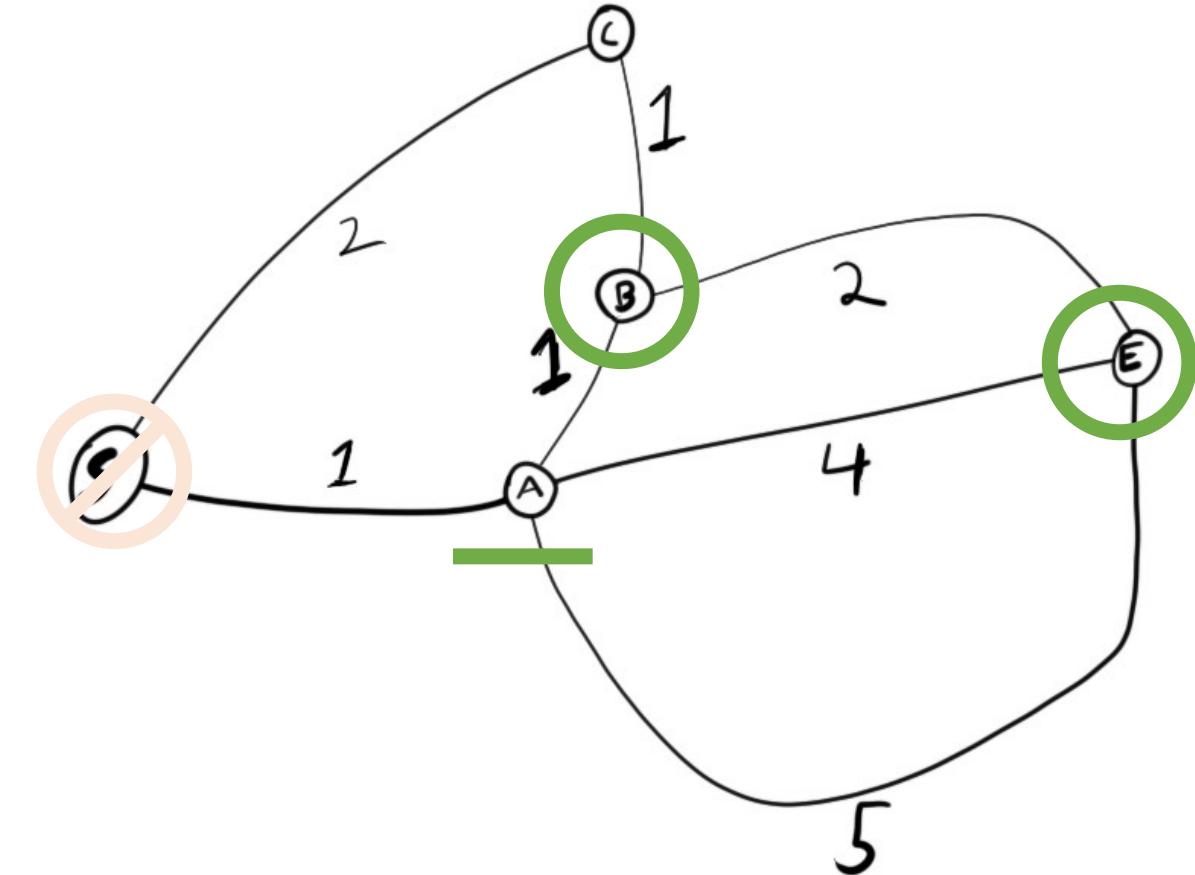
NODES	DISTANCE FROM START	PREVIOUS
START	0	$\emptyset$
A	1	START
B	$\infty$	$\emptyset$
C	2	START
END	$\infty$	$\emptyset$



# DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM  
PREVIOUS NODES

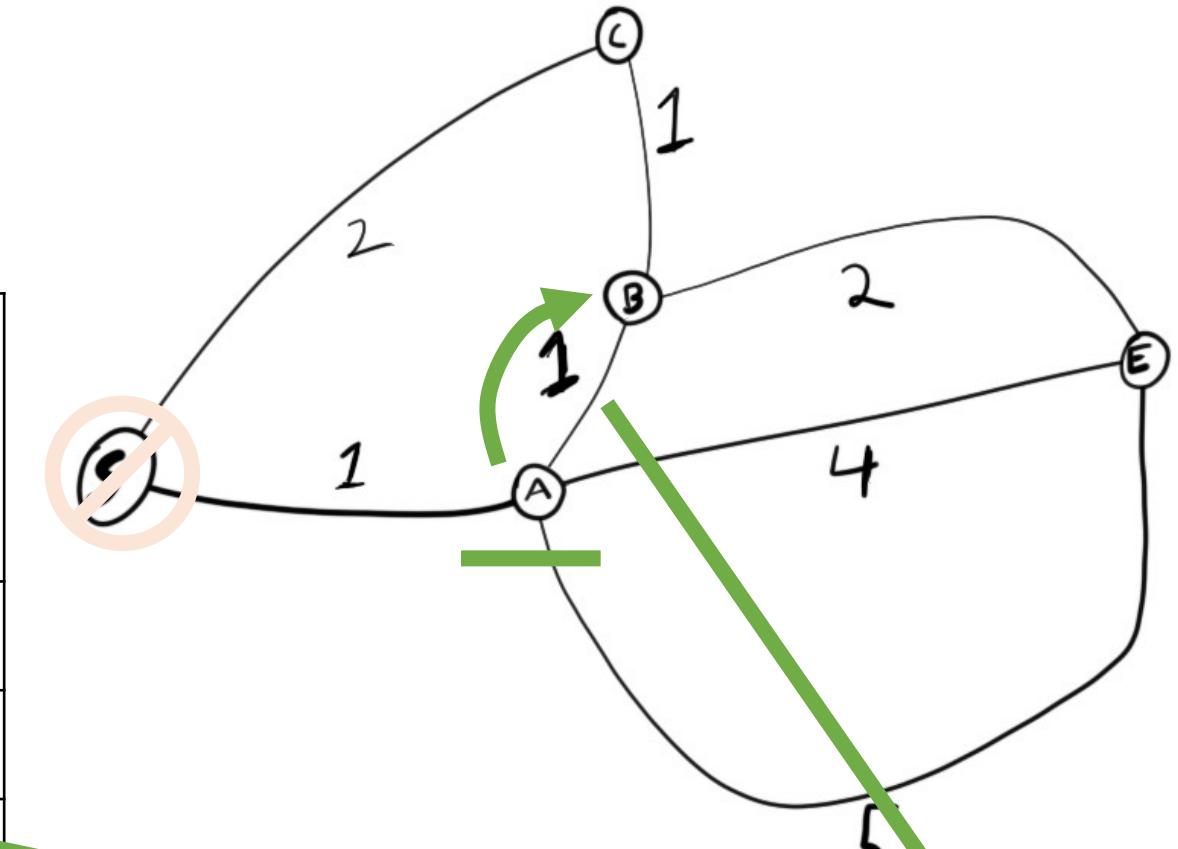
NODES	DISTANCE FROM START	PREVIOUS
START	0	$\emptyset$
A	1	START
B	$\infty$	$\emptyset$
C	2	START
END	$\infty$	$\emptyset$



# DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM  
PREVIOUS NODES

NODES	DISTANCE FROM START	PREVIOUS
START	0	$\emptyset$
A	1	START
B	$\infty$	$\emptyset$
C	2	START
END	$\infty$	$\emptyset$

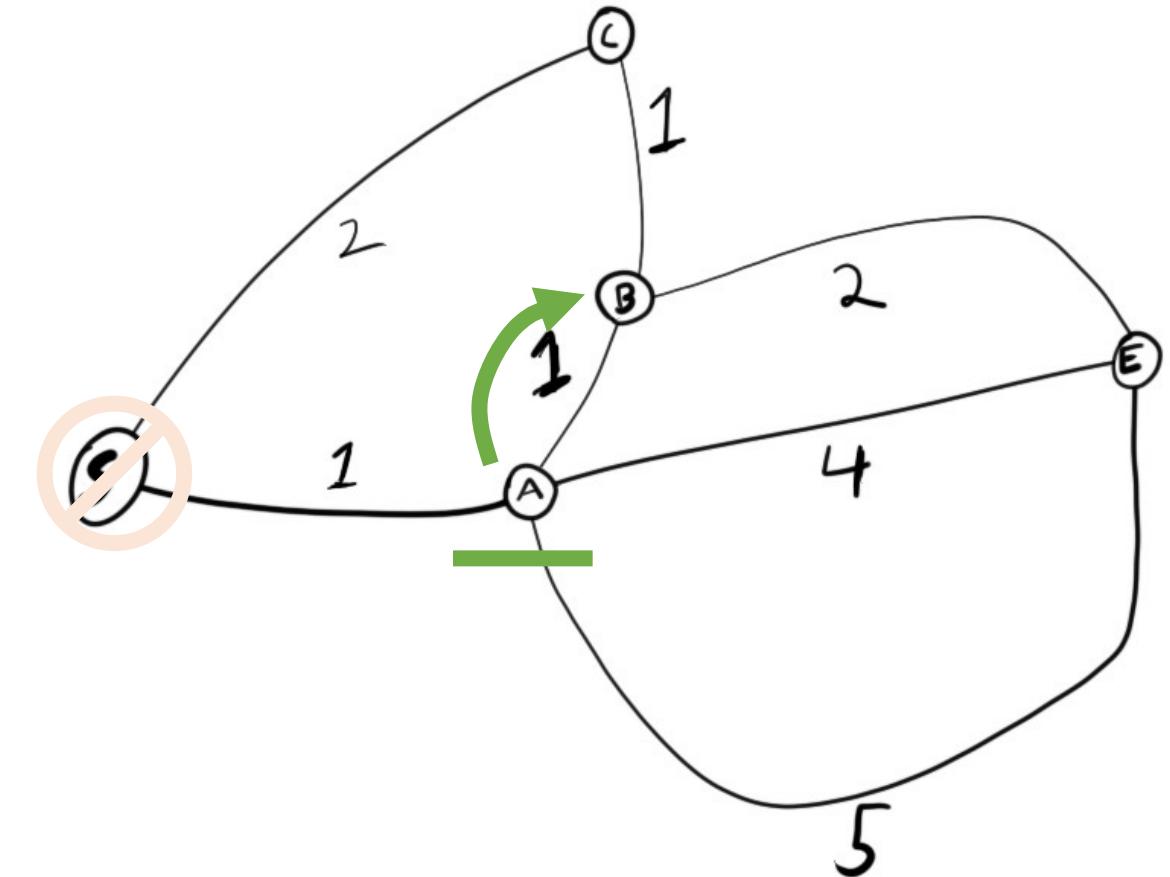


DISTANCE FROM START TO B  
 $D_{SB} = D_{SA} + D_{AB} = 1 + 1$

# DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM  
PREVIOUS NODES

NODES	DISTANCE FROM START	PREVIOUS
START	0	$\emptyset$
A	1	START
B	$\infty$	$\emptyset$
C	2	START
END	$\infty$	$\emptyset$

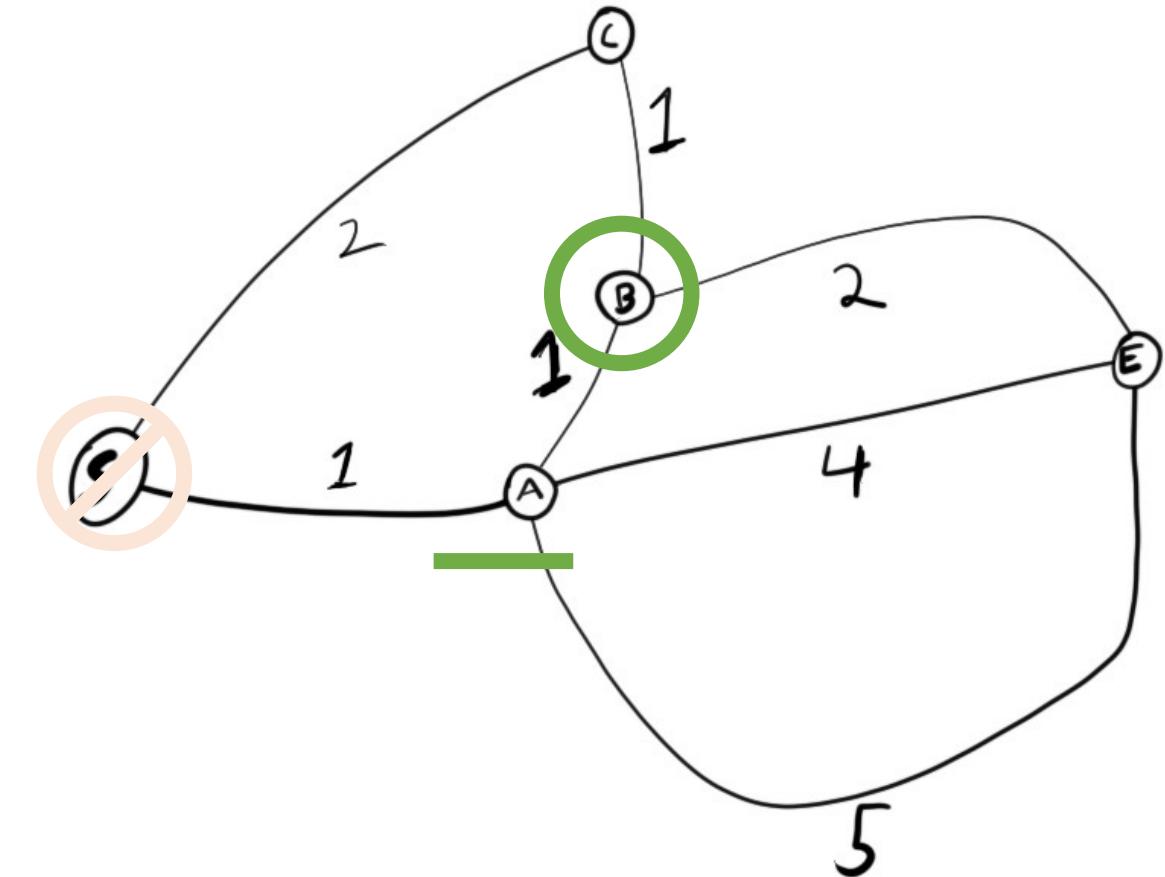


DISTANCE FROM START TO B  
 $D_{SB} = D_{SA} + D_{AB} = 1 + 1$

# DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM  
PREVIOUS NODES

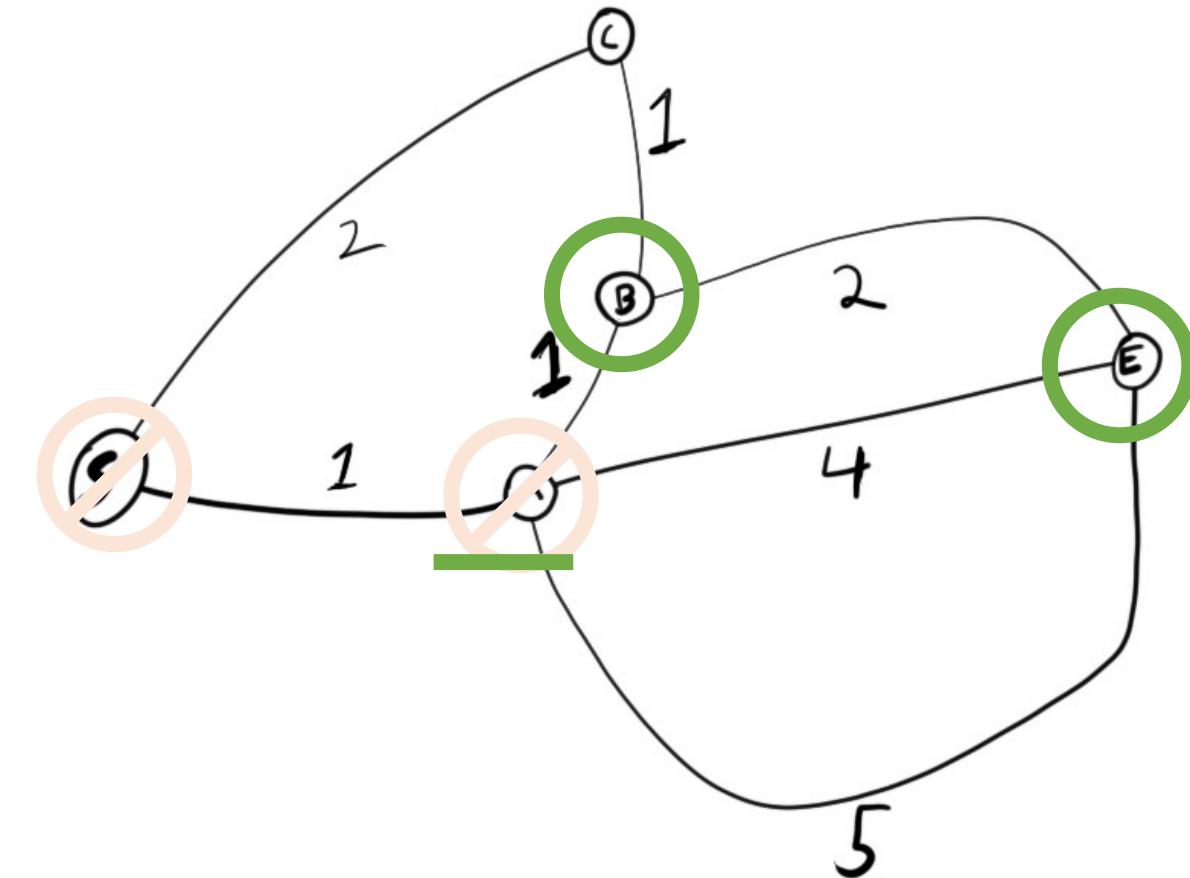
NODES	DISTANCE FROM START	PREVIOUS
START	0	$\emptyset$
A	1	START
B	2	A
C	2	START
END	5	A



# DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM  
PREVIOUS NODES

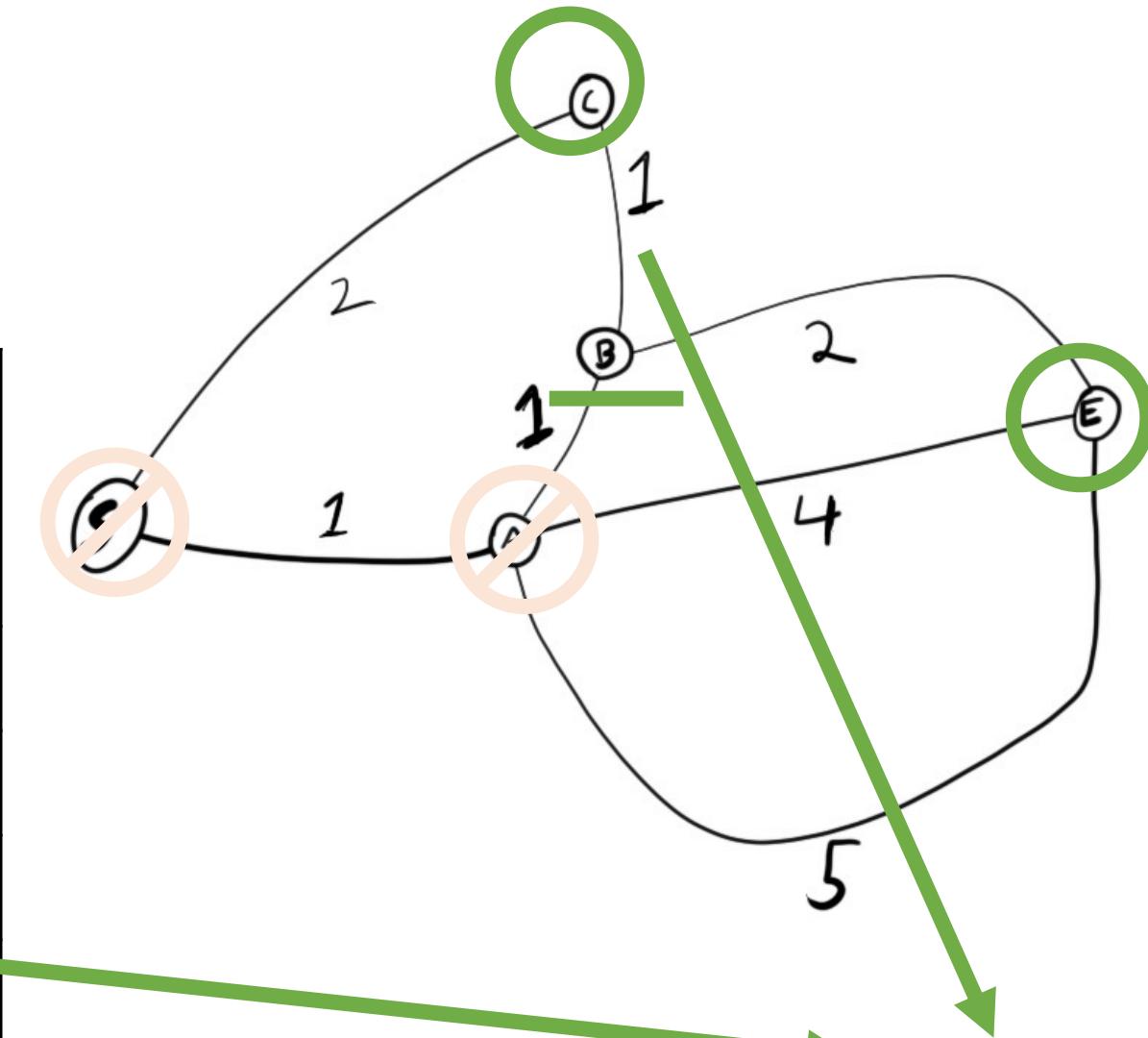
NODES	DISTANCE FROM START	PREVIOUS
START	0	$\emptyset$
A	1	START
B	2	A
C	2	START
END	5	A



# DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM  
PREVIOUS NODES

NODES	DISTANCE FROM START	PREVIOUS
START	0	$\emptyset$
A	1	START
B	2	A
C	2	START
END	5	A

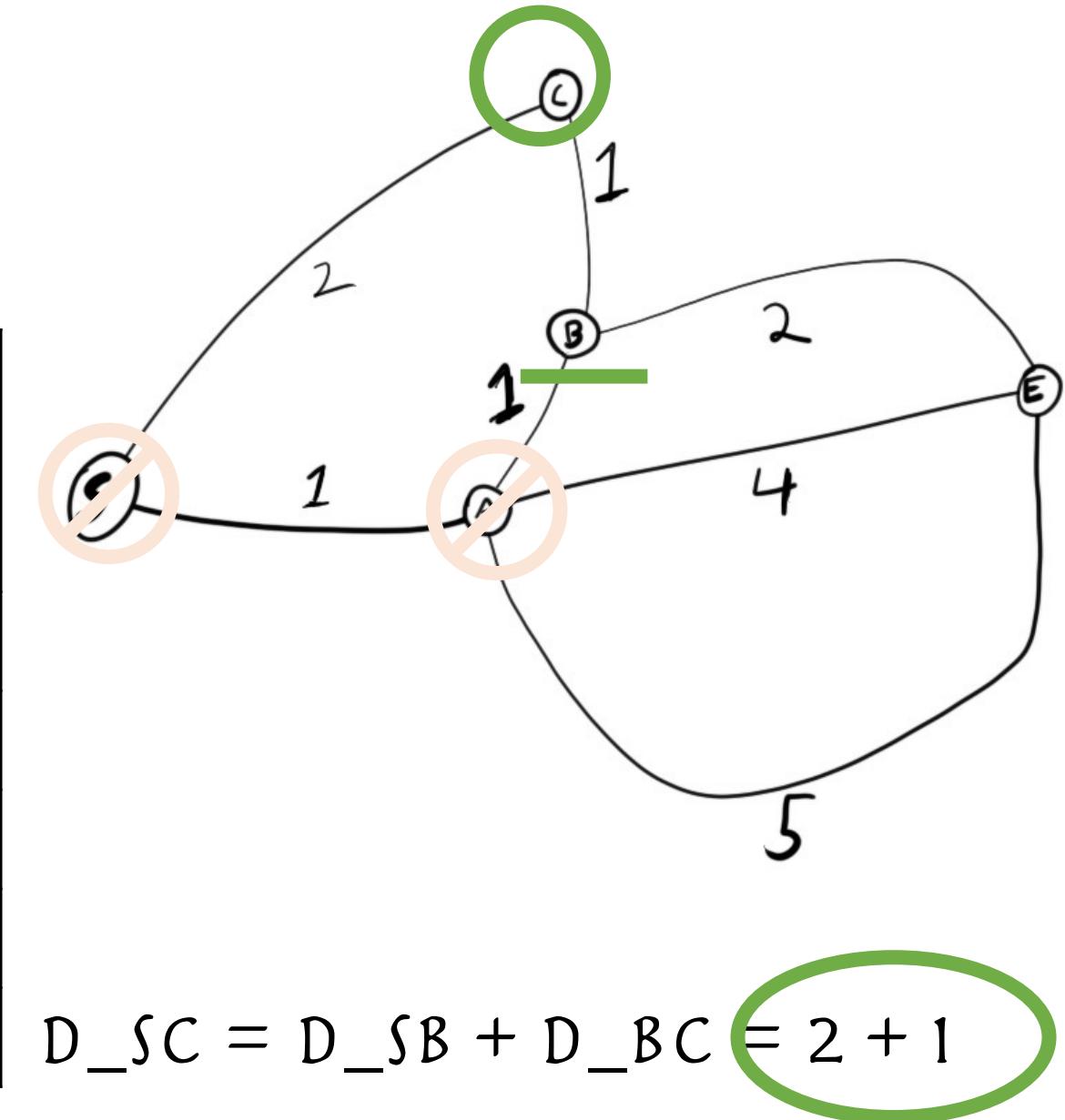


$$D_{SC} = D_{SB} + D_{BC} = 2 + 1$$

# DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM  
PREVIOUS NODES

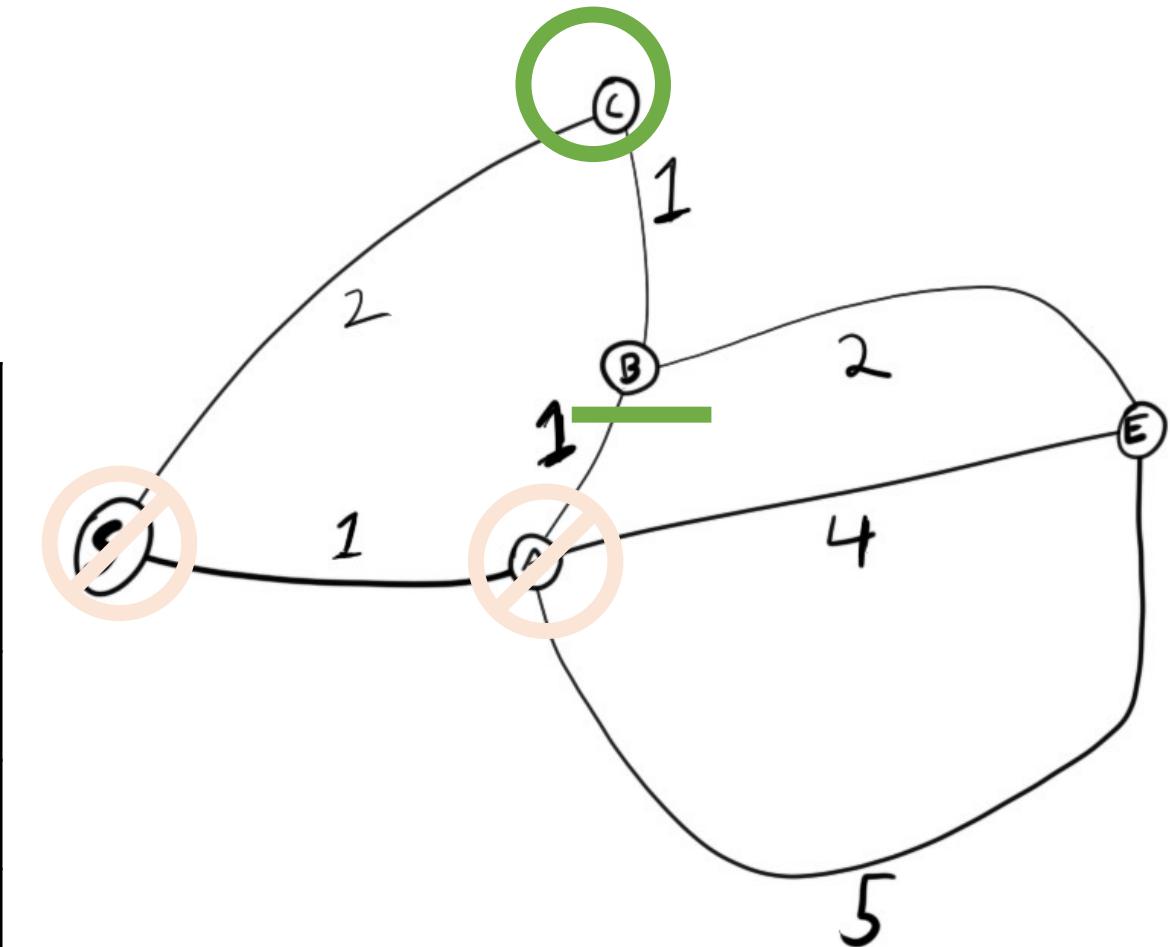
NODES	DISTANCE FROM START	PREVIOUS
START	0	$\emptyset$
A	1	START
B	2	A
C	2	START
END	5	A



# DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM  
PREVIOUS NODES

NODES	DISTANCE FROM START	PREVIOUS
START	0	$\emptyset$
A	1	START
B	2	A
C	2	START
END	5	A

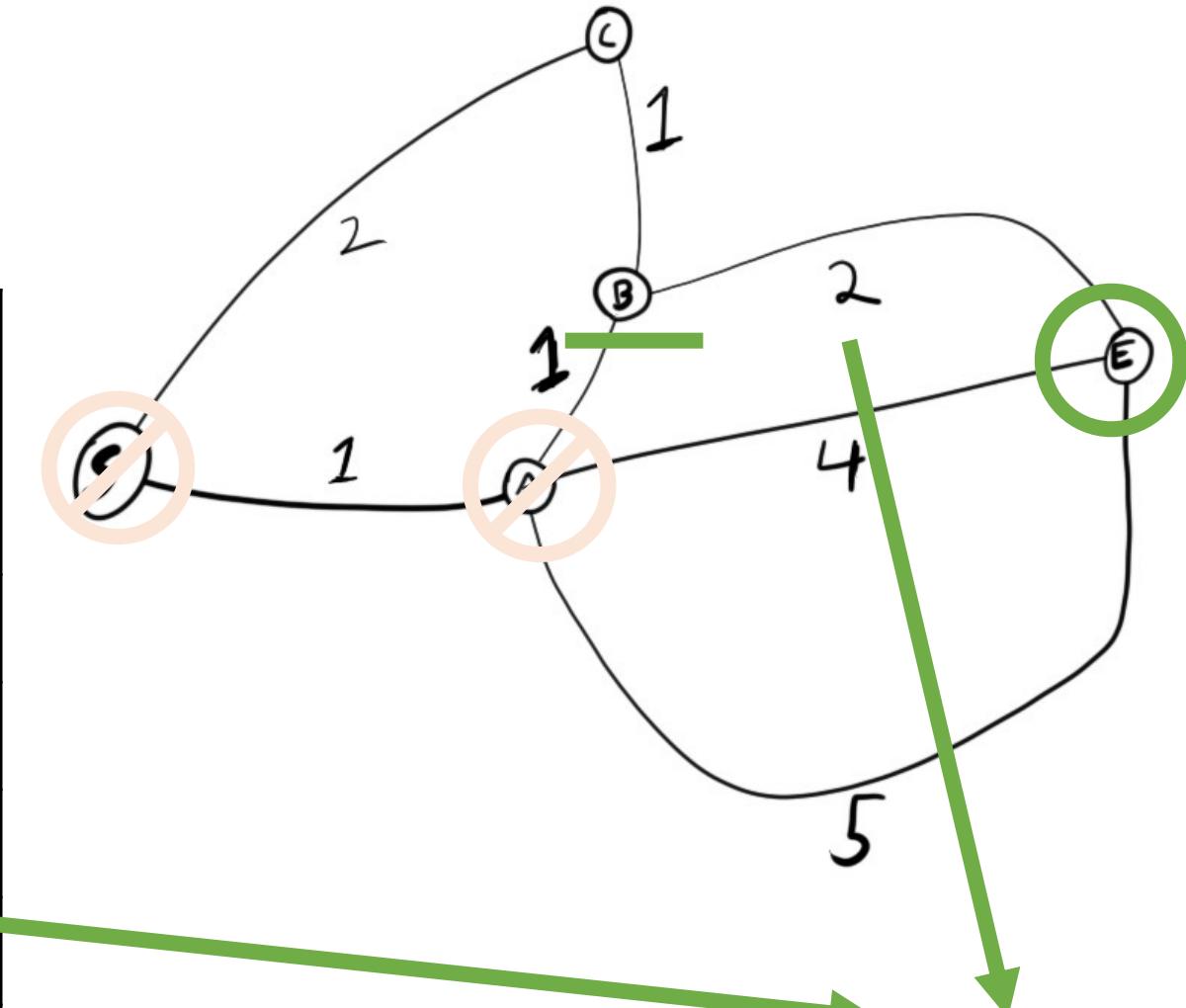


$$D_{SC} = D_{SB} + D_{BC} = 2 + 1$$

# DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM  
PREVIOUS NODES

NODES	DISTANCE FROM START	PREVIOUS
START	0	$\emptyset$
A	1	START
B	2	A
C	2	START
END	5	A

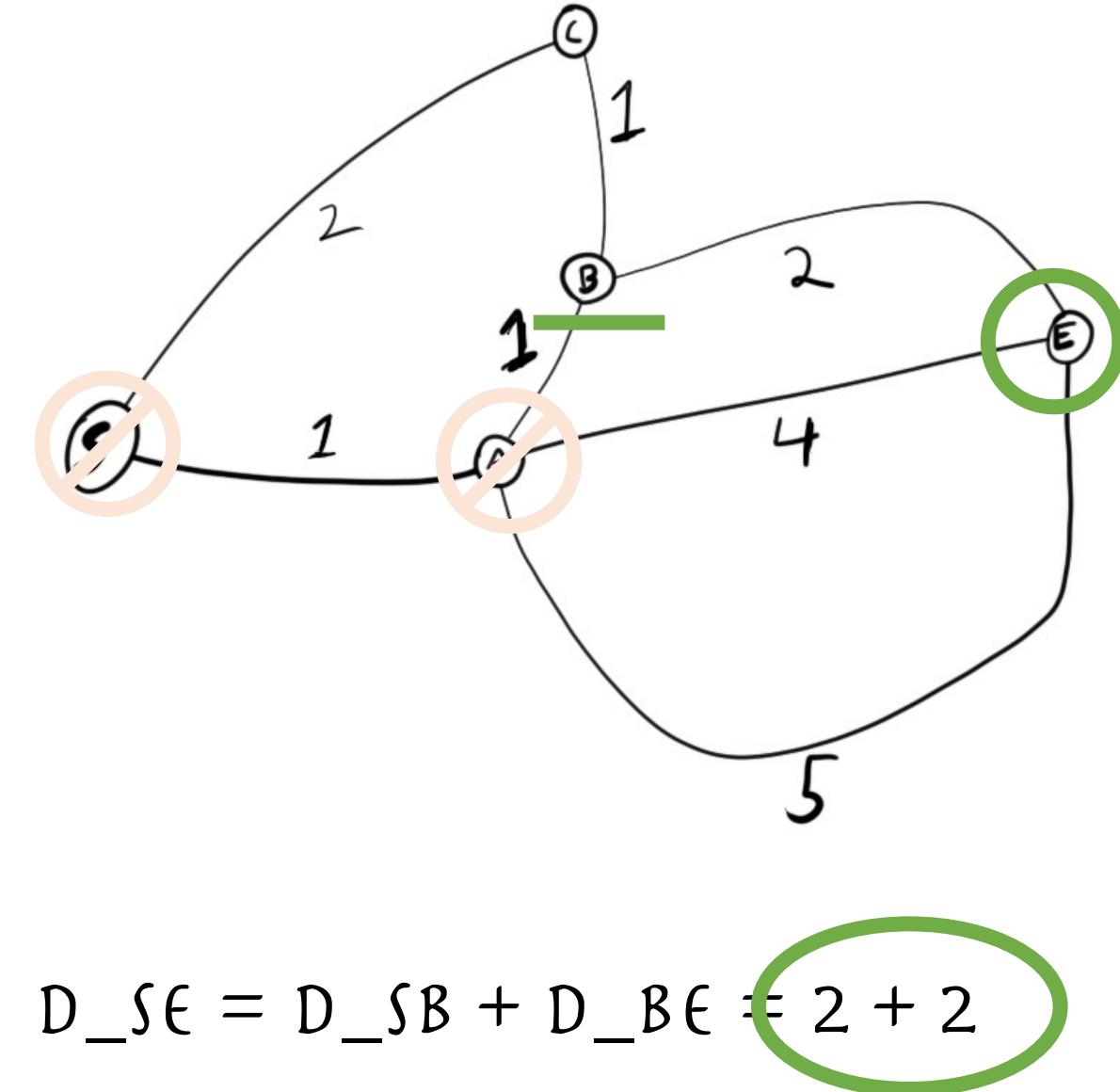


$$D_{SE} = D_{SB} + D_{BE} = 2 + 2$$

# DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM  
PREVIOUS NODES

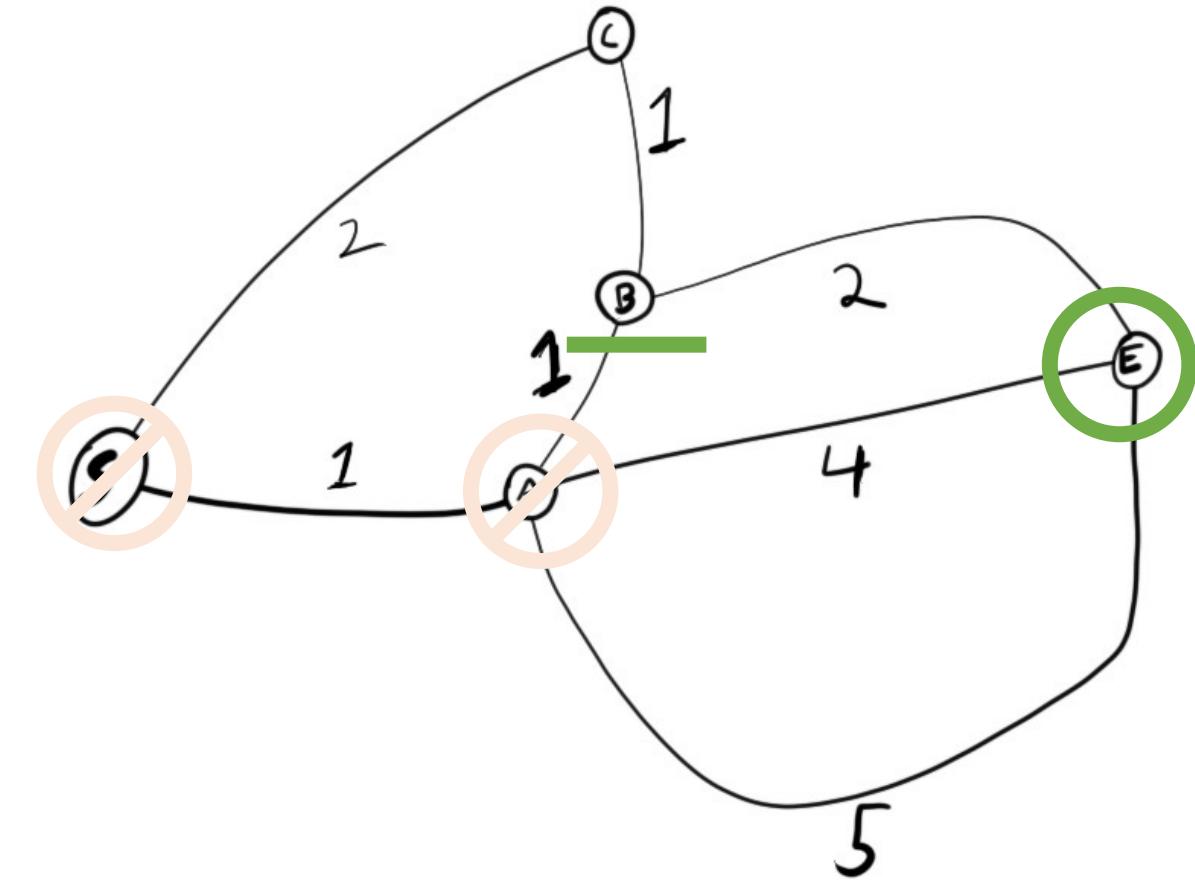
NODES	DISTANCE FROM START	PREVIOUS
START	0	$\emptyset$
A	1	START
B	2	A
C	2	START
END	5	A



# DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM  
PREVIOUS NODES

NODES	DISTANCE FROM START	PREVIOUS
START	0	$\emptyset$
A	1	START
B	2	A
C	2	START
END	4	B

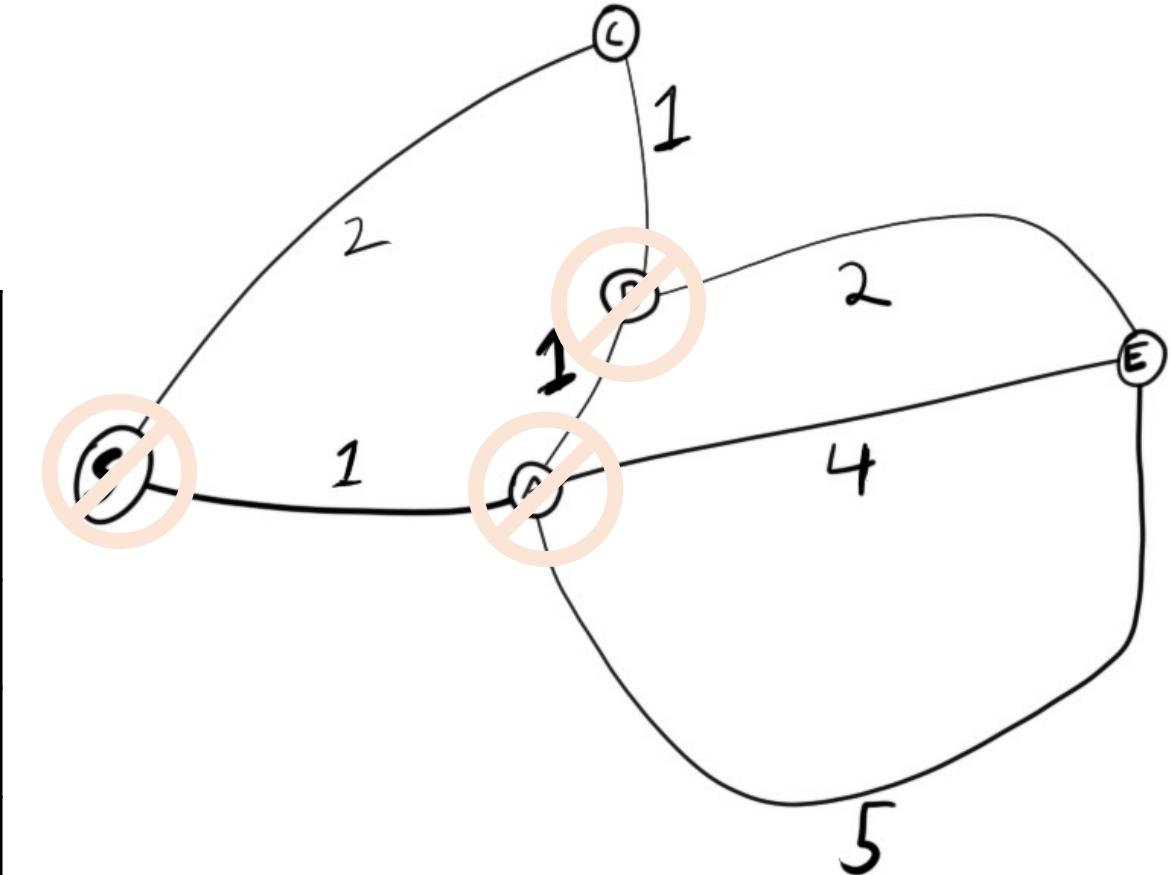


$$D_{SE} = D_{SB} + D_{BE} = 2 + 2$$

# DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM  
PREVIOUS NODES

NODES	DISTANCE FROM START	PREVIOUS
START	0	$\emptyset$
A	1	START
B	2	A
C	2	START
END	4	B

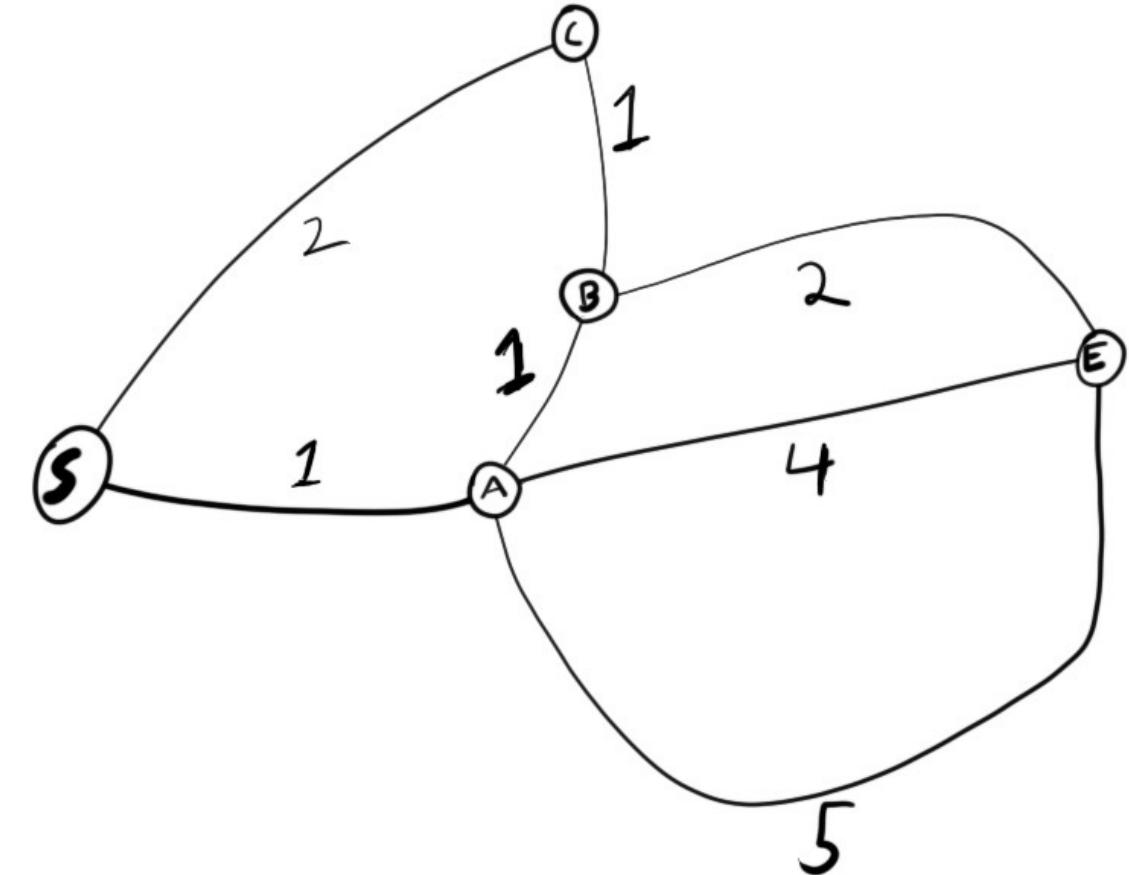


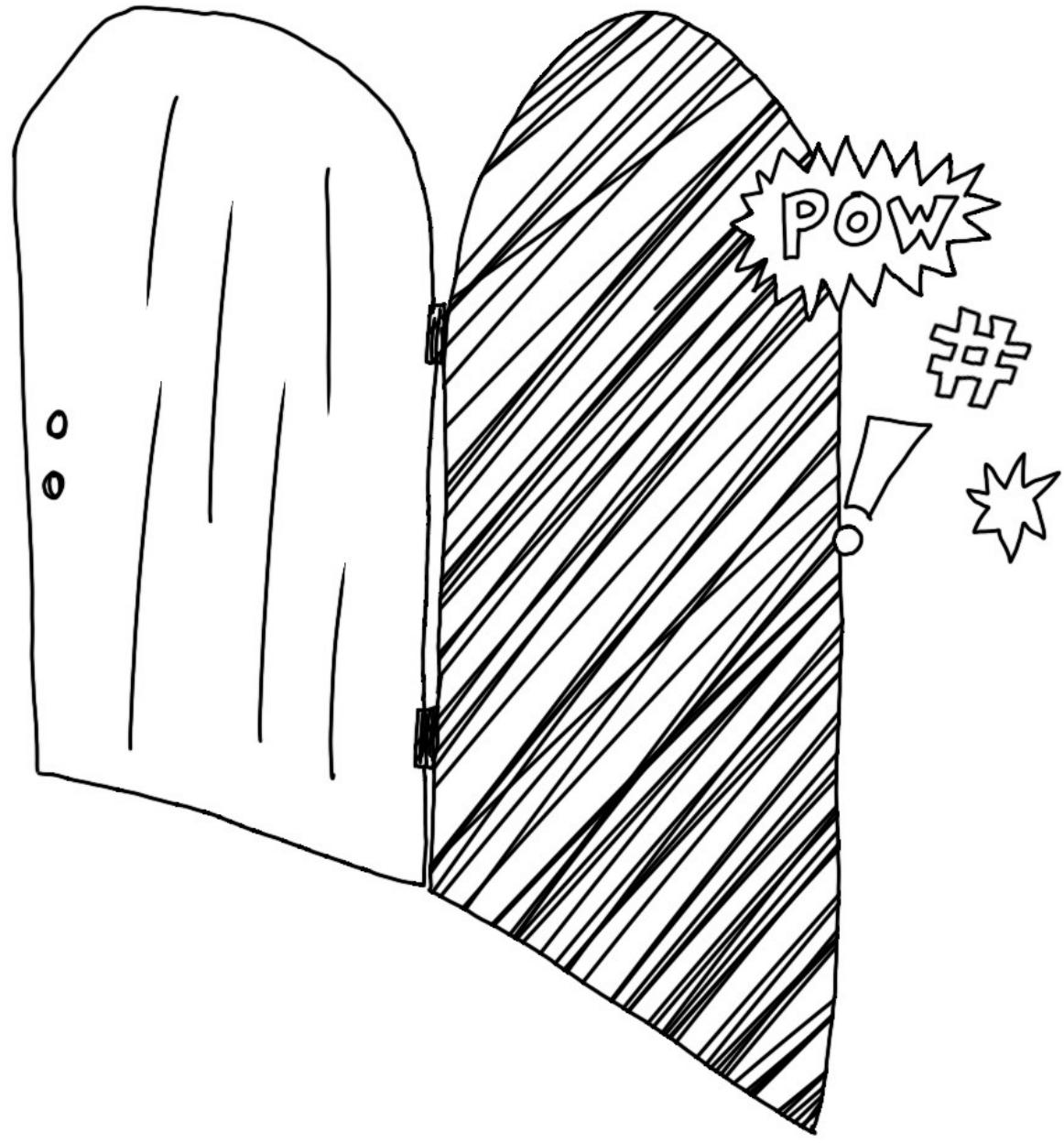
REPEAT UNTIL TARGET  
NODE IS VISITED

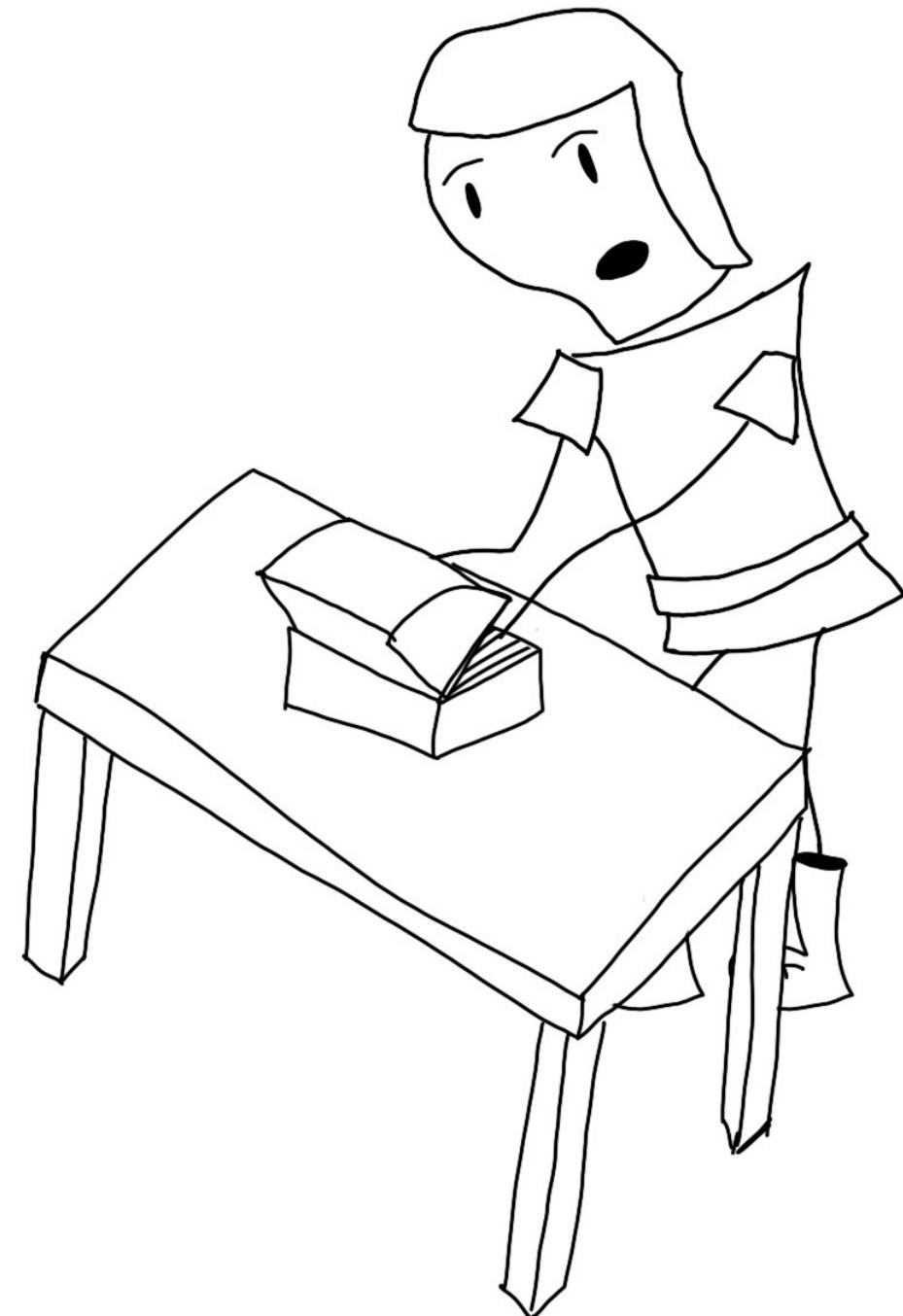
# DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM  
PREVIOUS NODES

NODES	DISTANCE FROM START	PREVIOUS
START	0	$\emptyset$
A	1	START
B	2	A
C	2	START
END	4	B

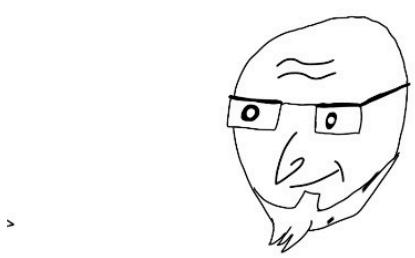


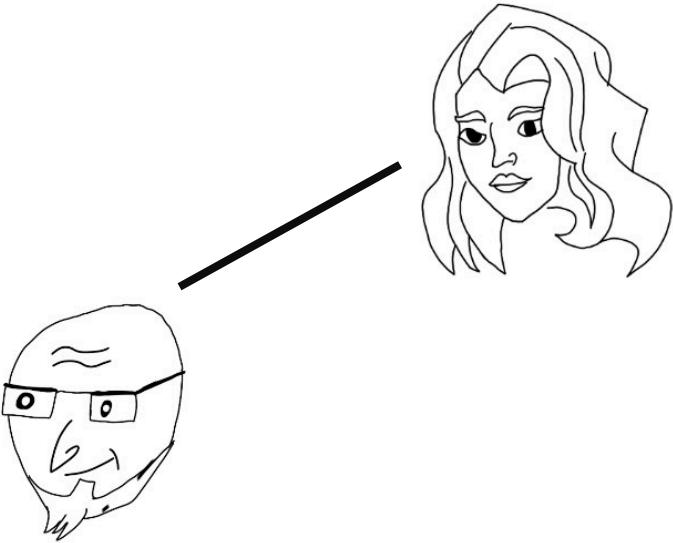


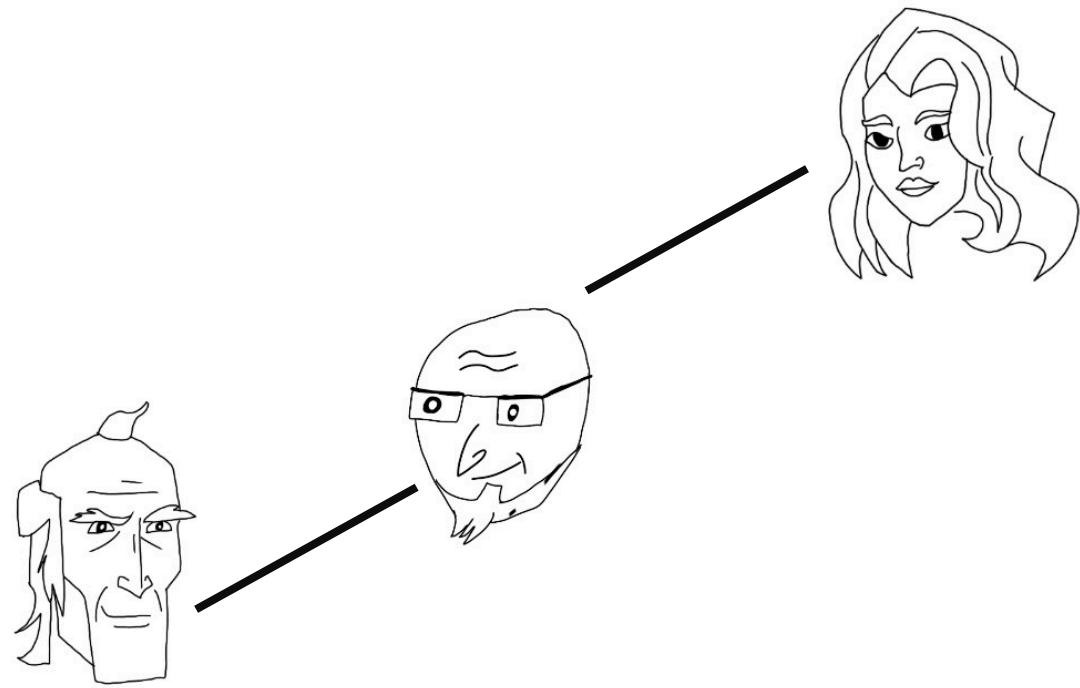


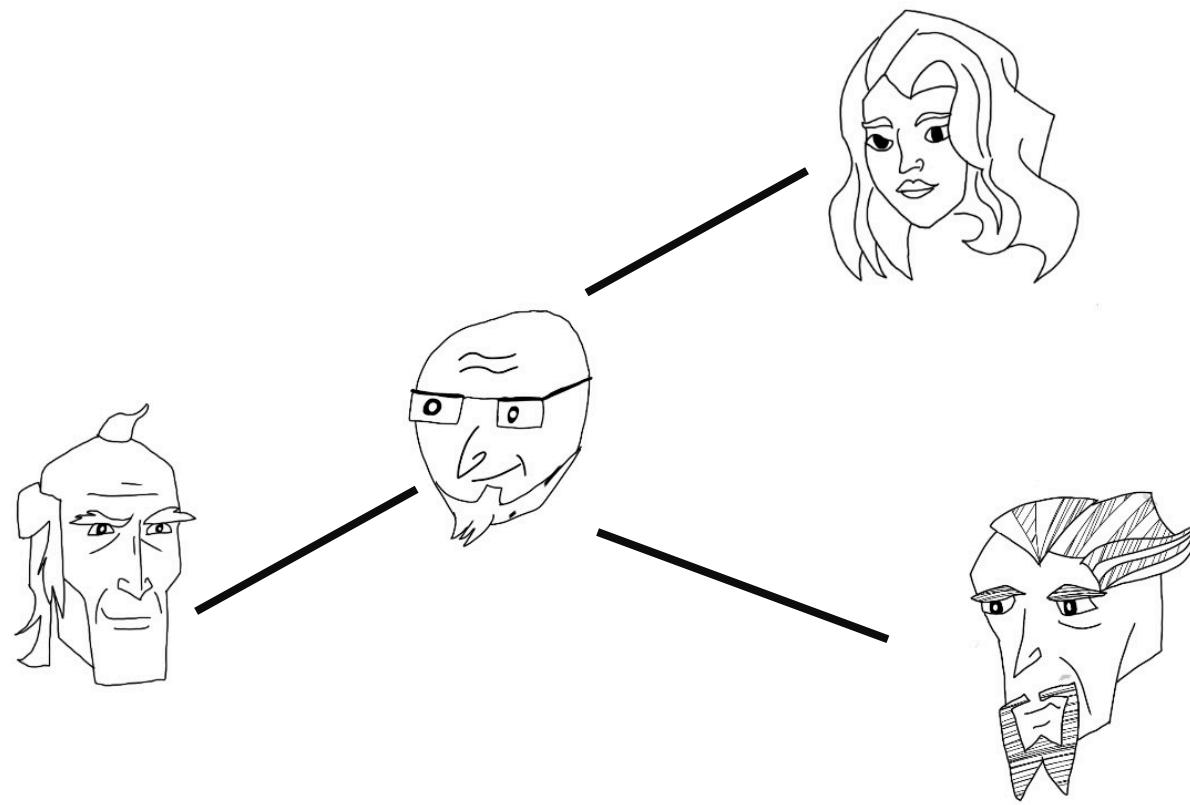
# WHO'S REALLY IN CHARGE?!

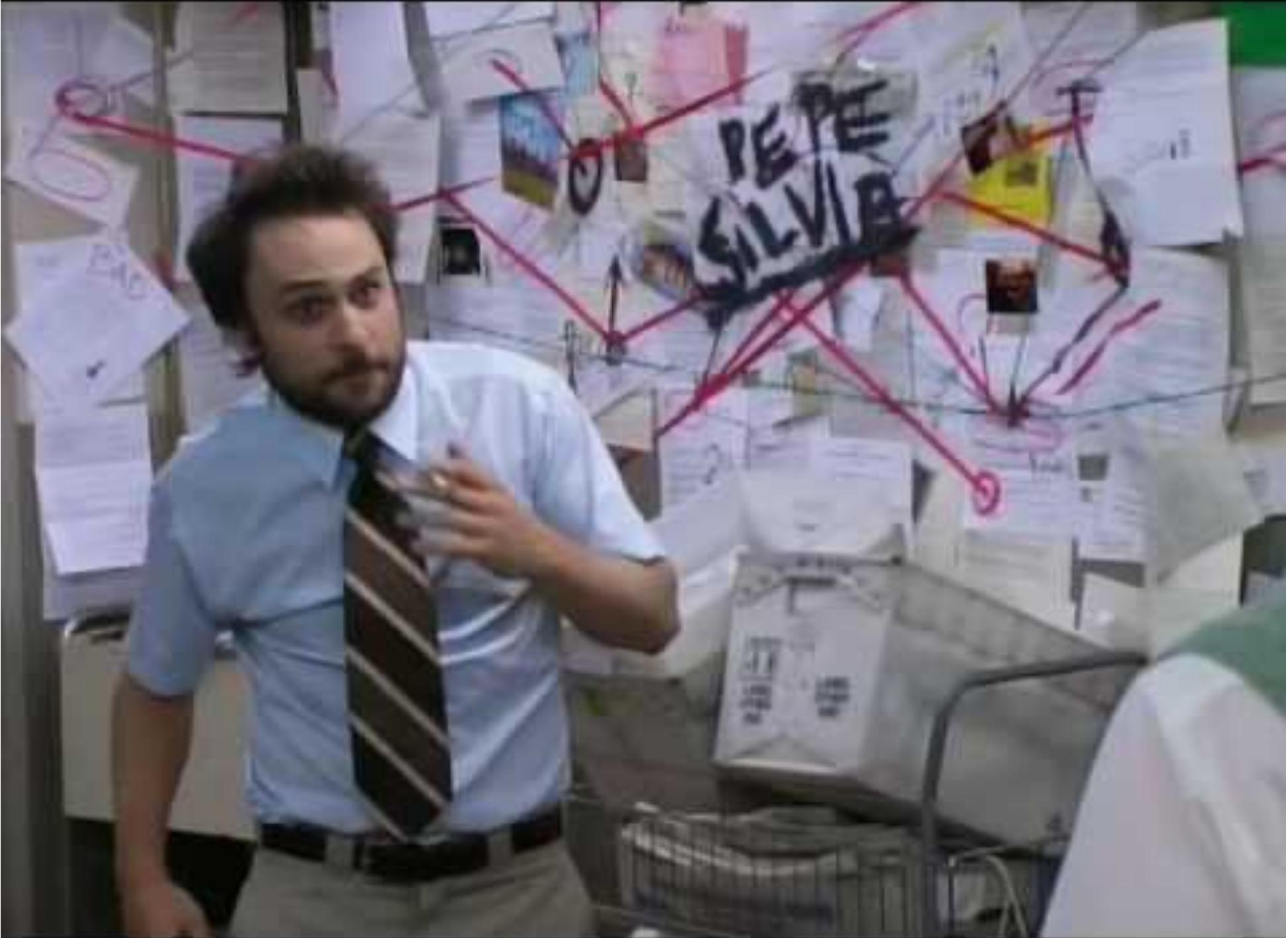
- YOU HAVE A LIST OF MESSAGES
- FRANK -> GINA
- GINA -> HECTOR
- HECTOR -> FRANK
- ?!?!?!?!

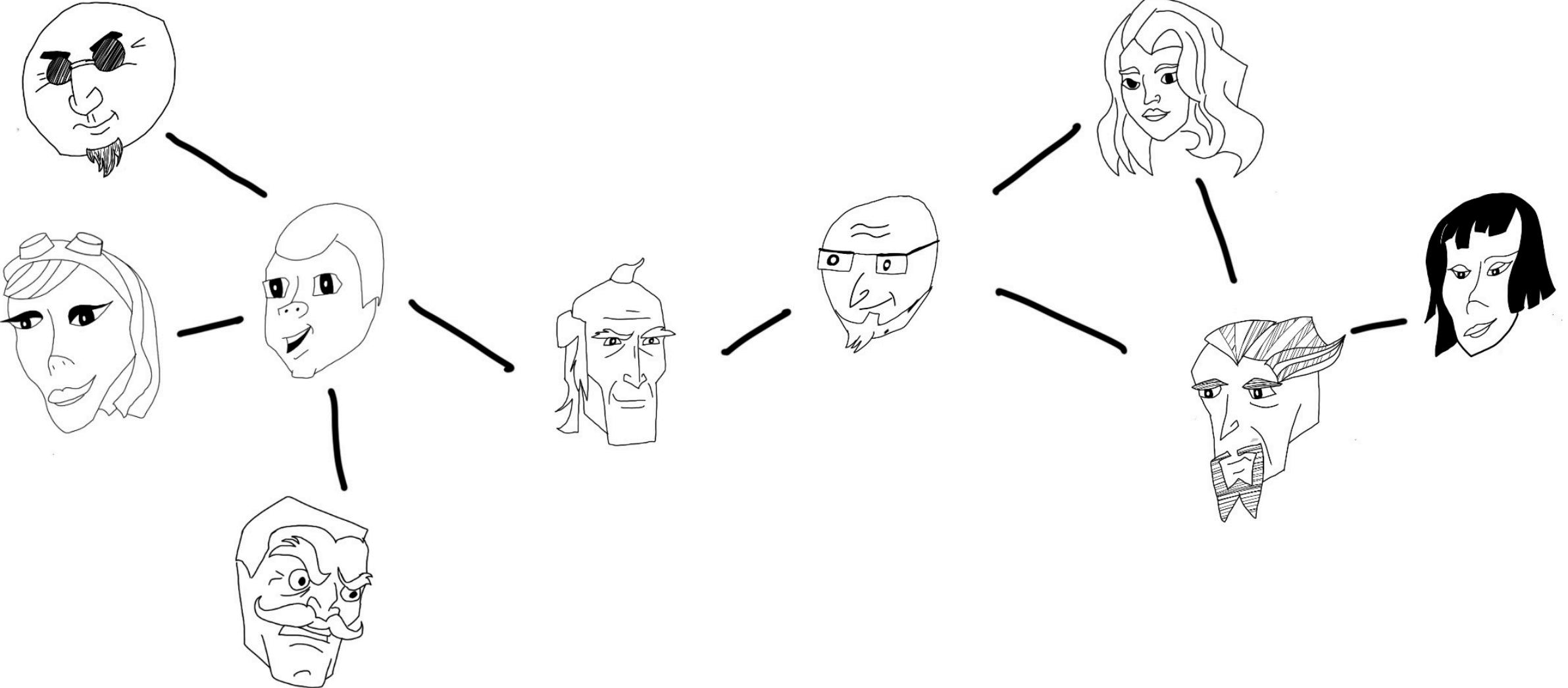












# WHO'S REALLY IN CHARGE?!

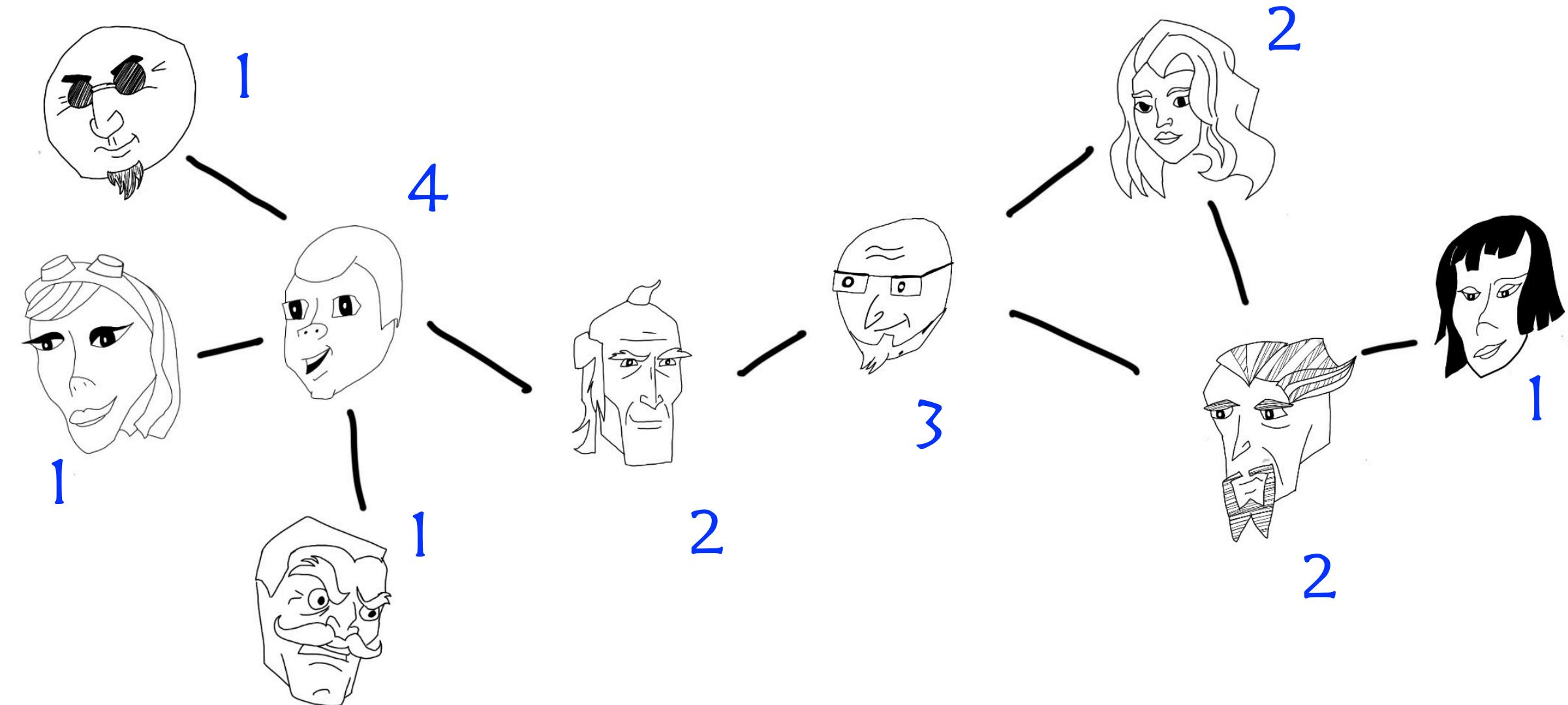
- GIVEN A GRAPH OF PEOPLE SENDING MESSAGES, HOW DO YOU DETERMINE IMPORTANT PEOPLE?
  - ARTICULATION POINTS
  - NODES IN A GRAPH THAT REDUCE CONNECTIVITY

# WHO'S REALLY IN CHARGE?!

- GIVEN A GRAPH OF PEOPLE SENDING MESSAGES, HOW DO YOU DETERMINE IMPORTANT PEOPLE?
  - ~~ARTICULATION POINTS~~
    - ~~NODES IN A GRAPH THAT REDUCE CONNECTIVITY~~
  - CENTRALITY
    - DETERMINING A NODE'S INFLUENCE

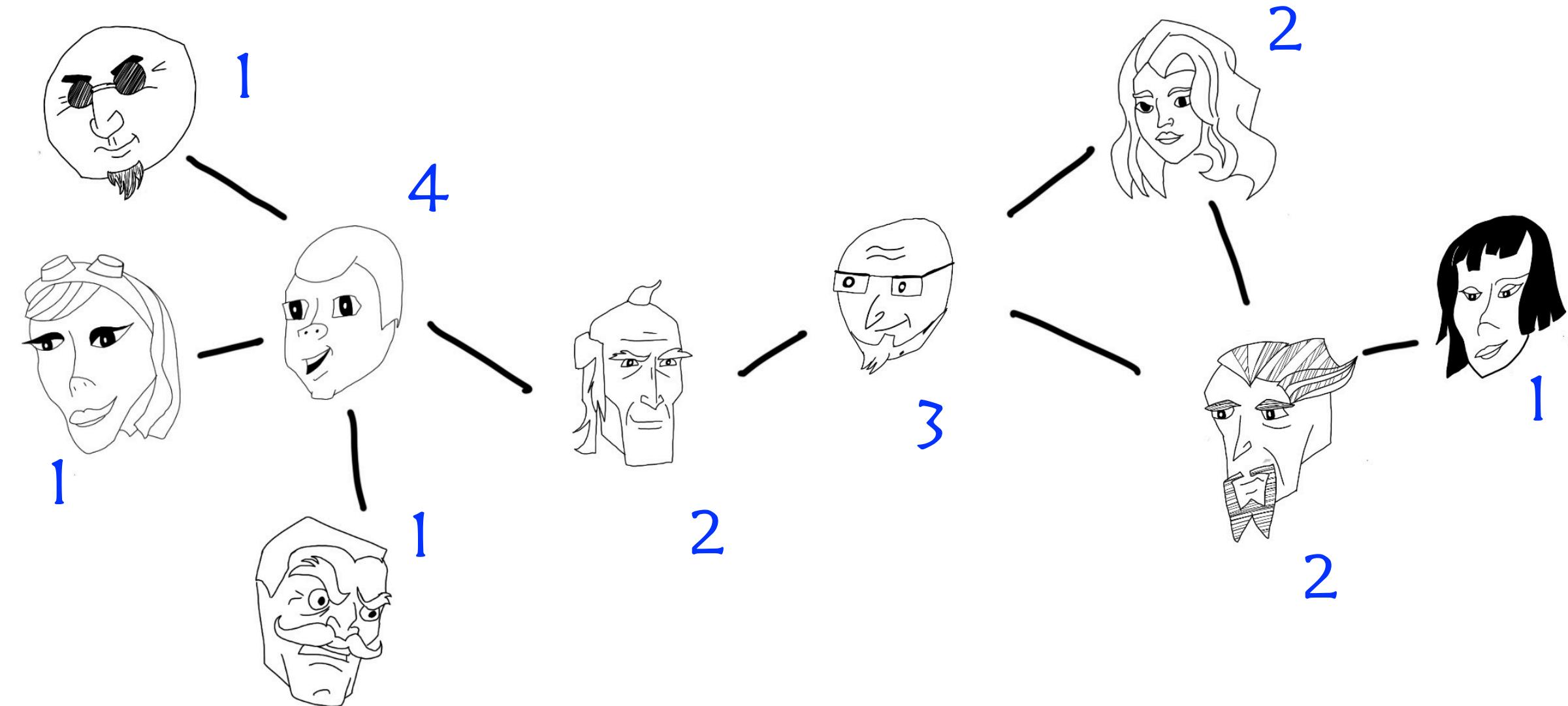
# DEGREE CENTRALITY

- JUST COUNT THE DEGREES PER NODE



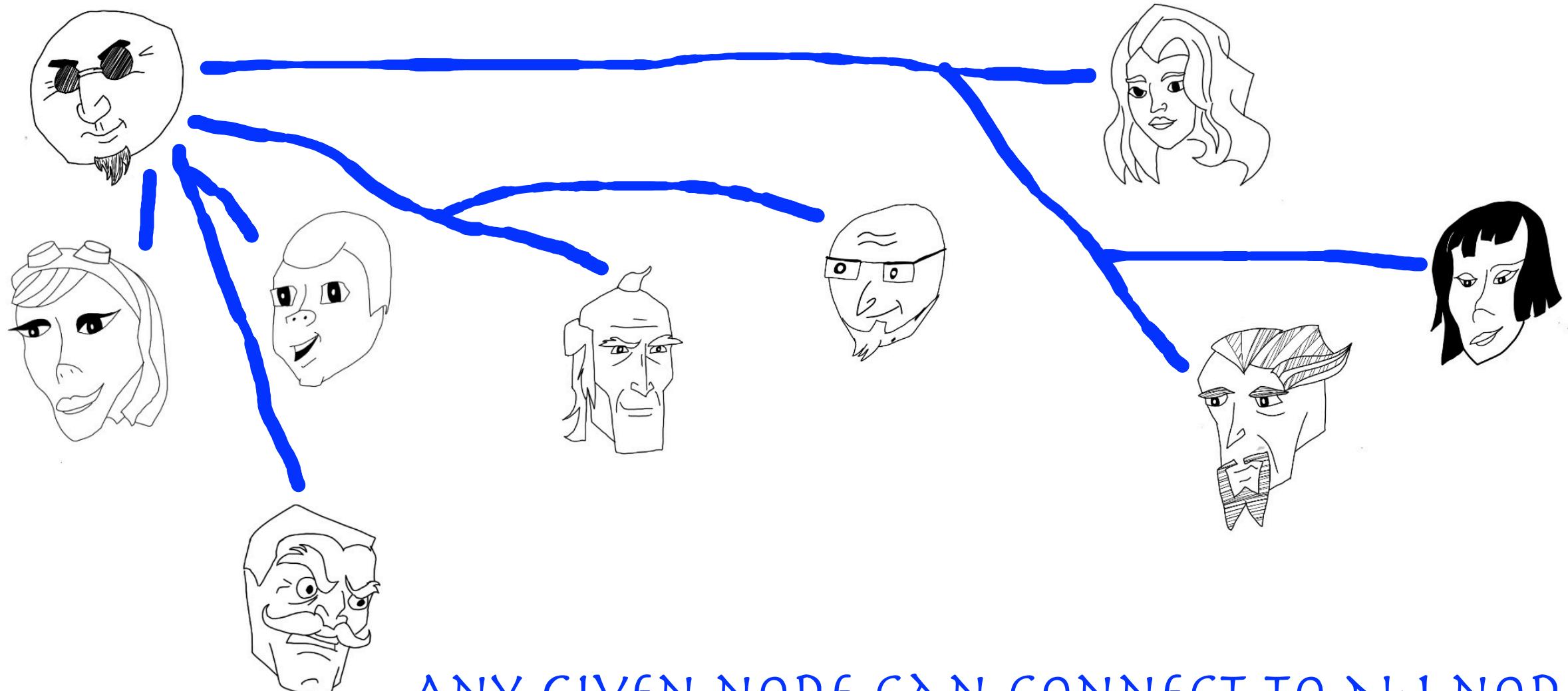
# DEGREE CENTRALITY

- ...AND NORMALIZE THEM BY ...?



# DEGREE CENTRALITY

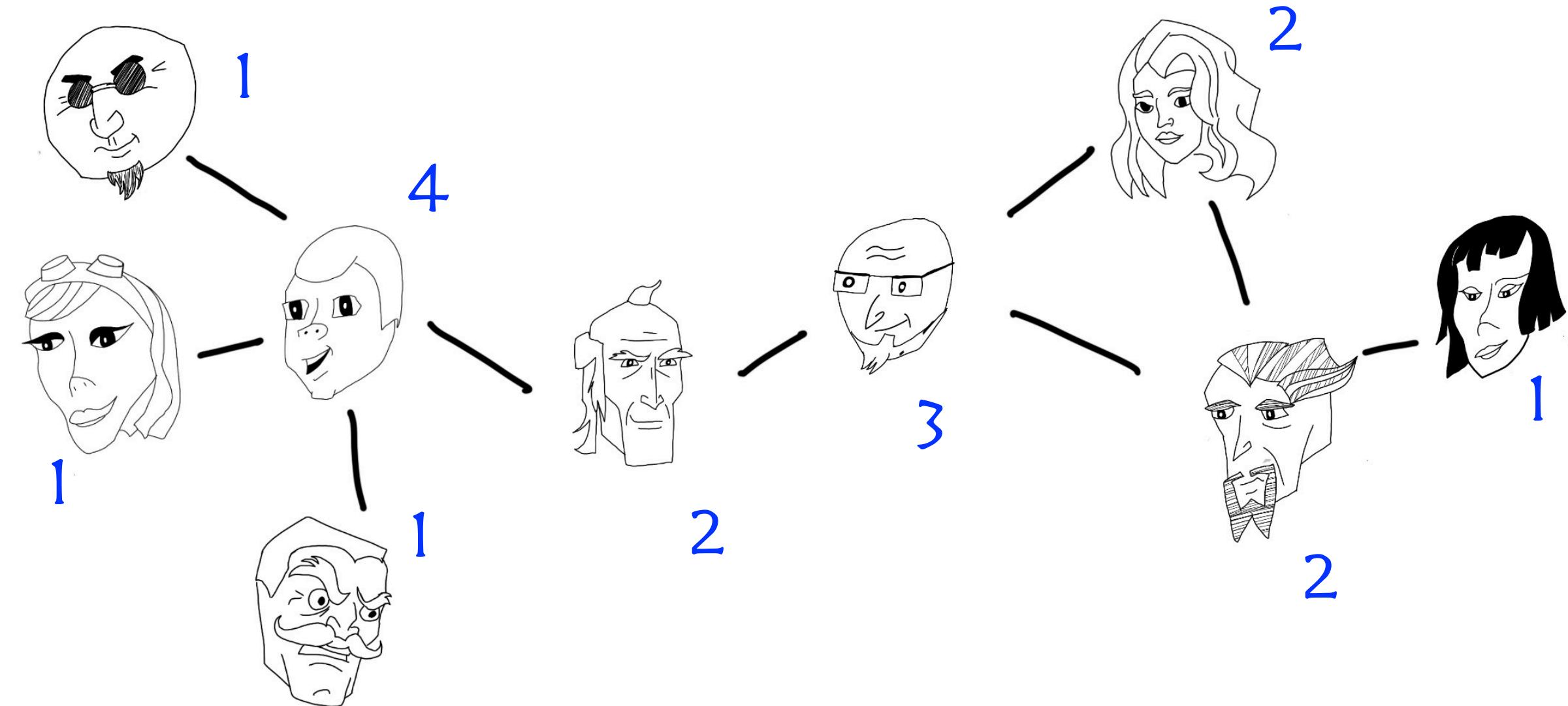
- ...AND NORMALIZE THEM BY MAX POSSIBLE DEGREE



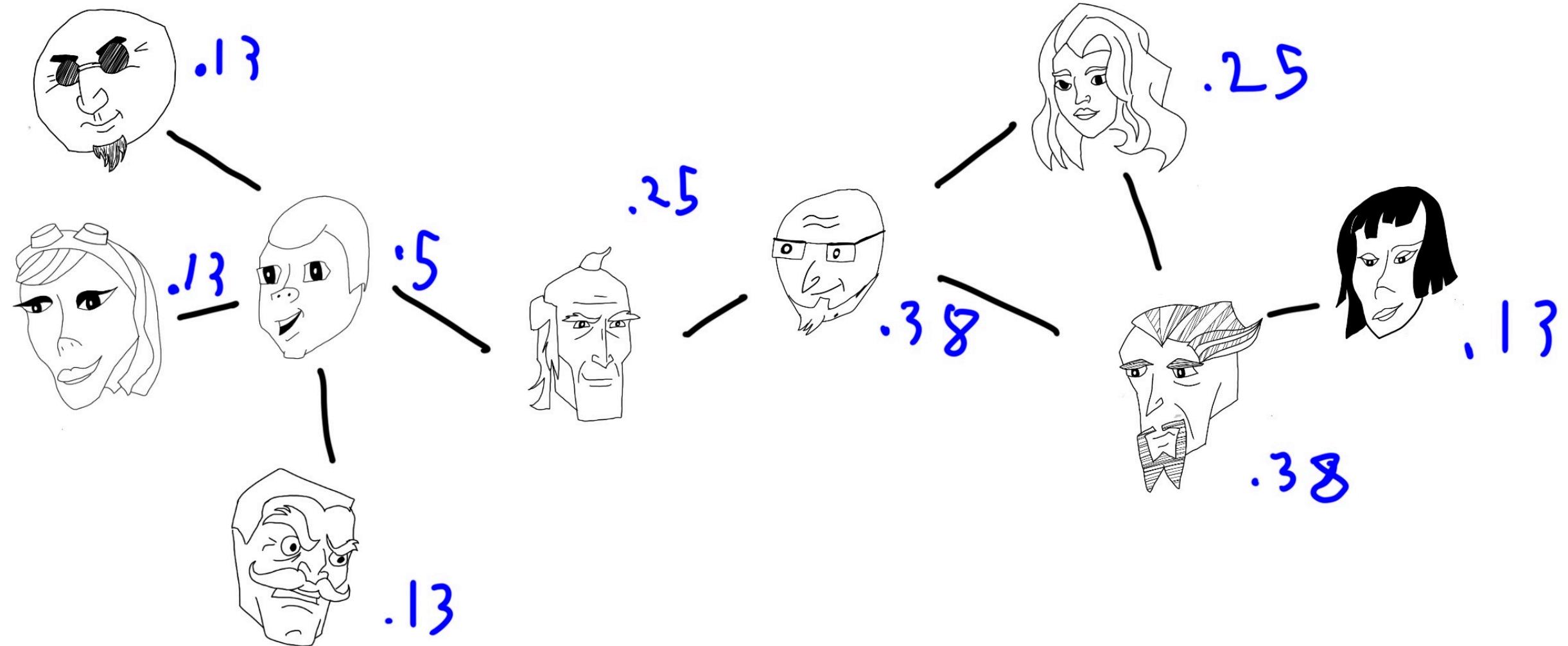
ANY GIVEN NODE CAN CONNECT TO  $N-1$  NODES

# DEGREE CENTRALITY

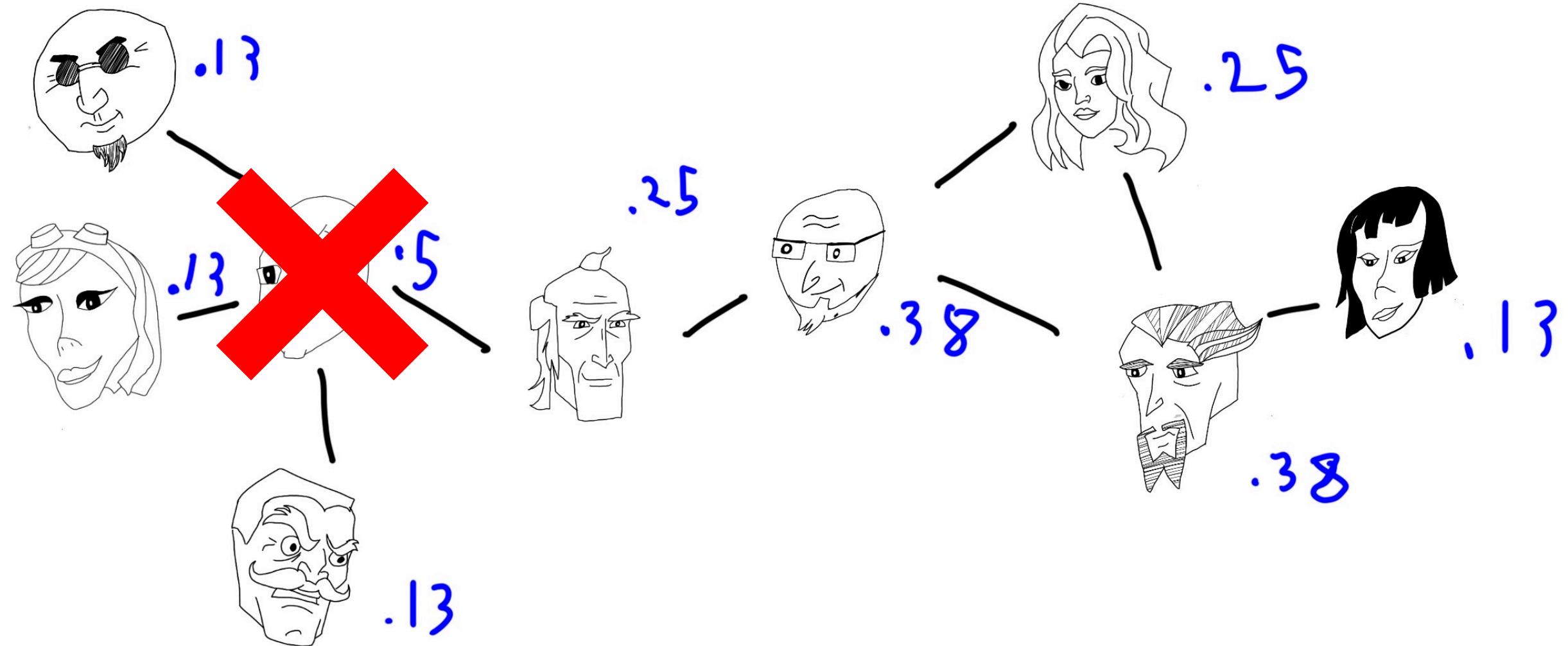
- ...AND NORMALIZE THEM BY MAX POSSIBLE DEGREE

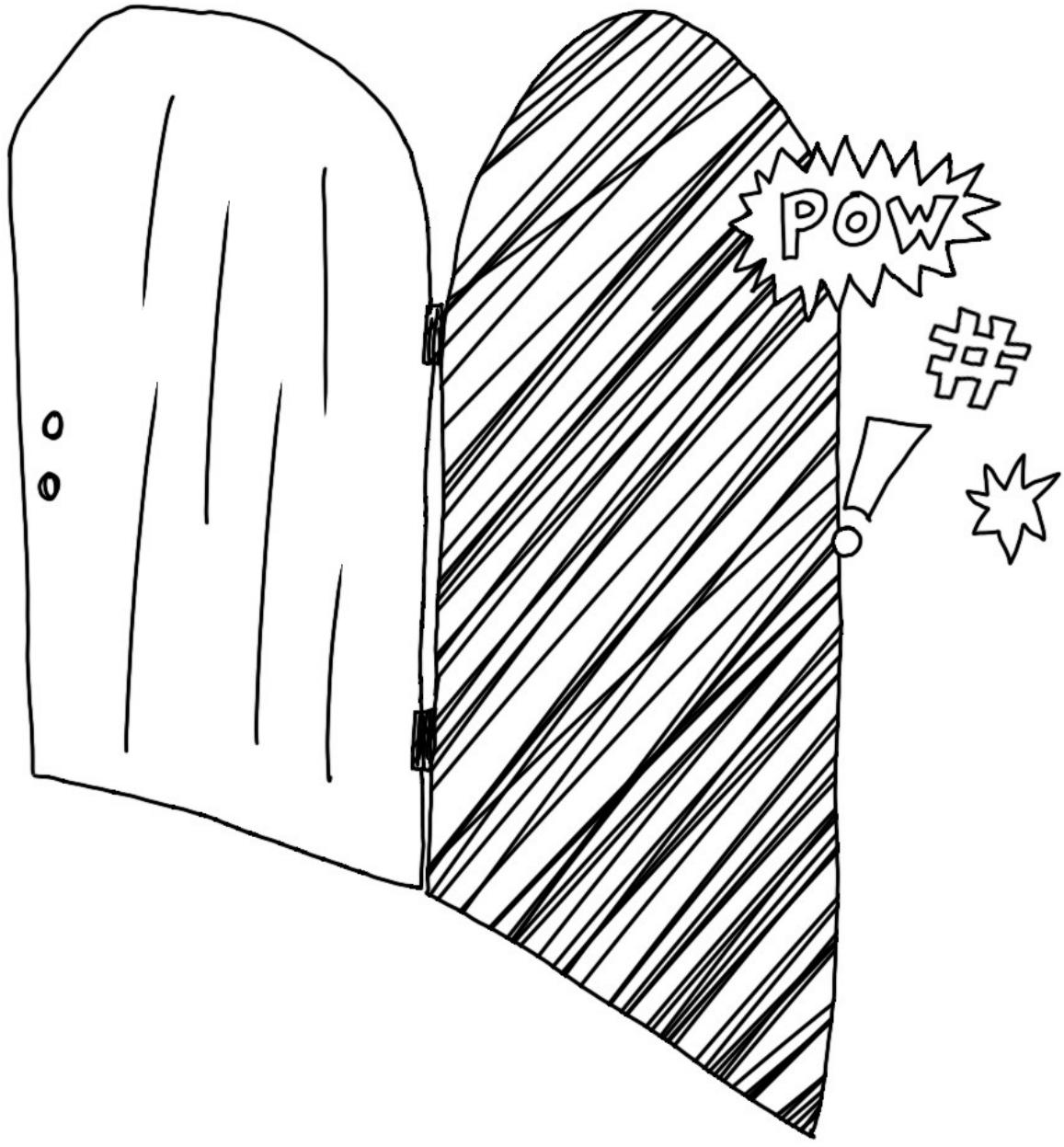


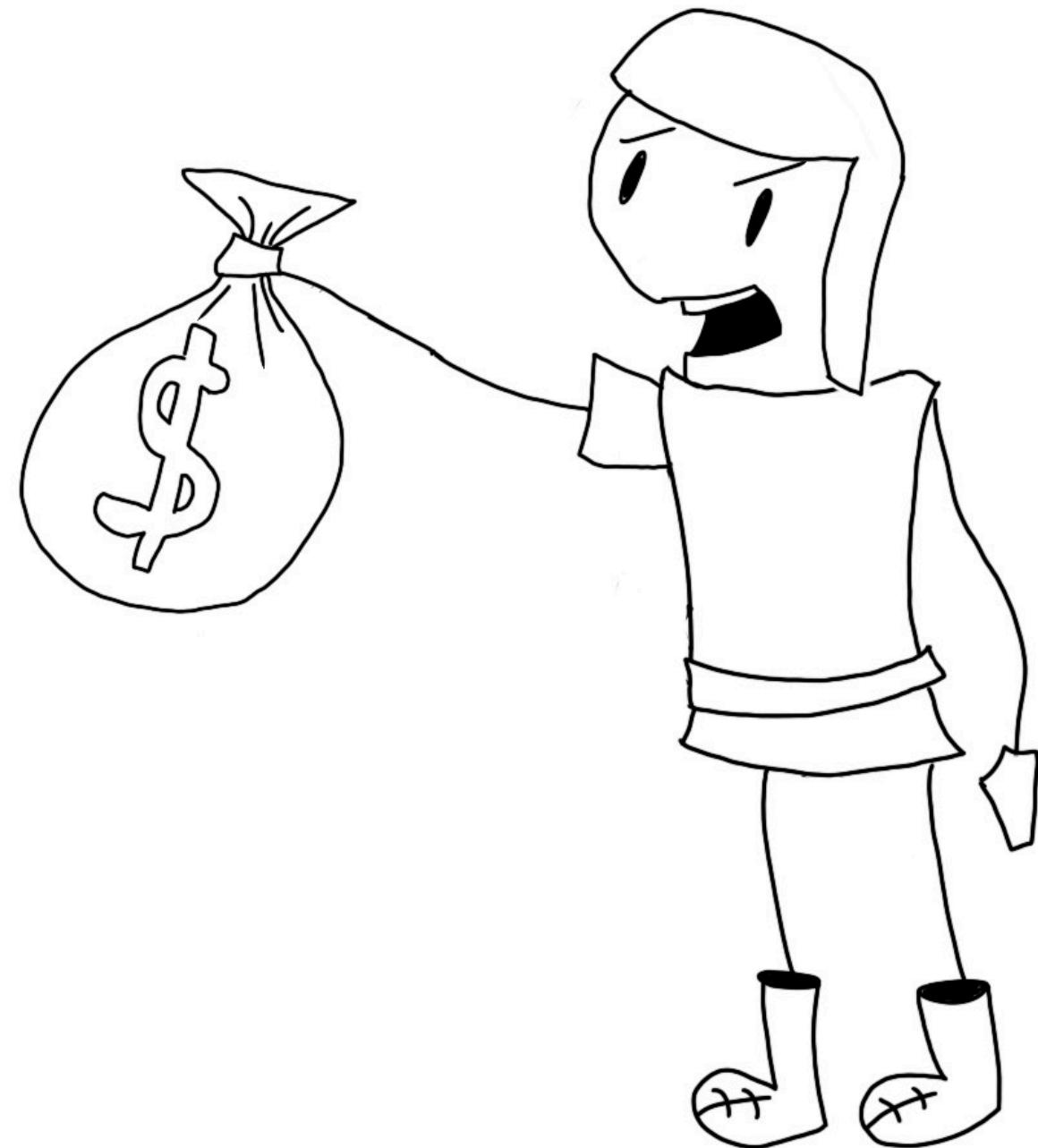
# DEGREE CENTRALITY

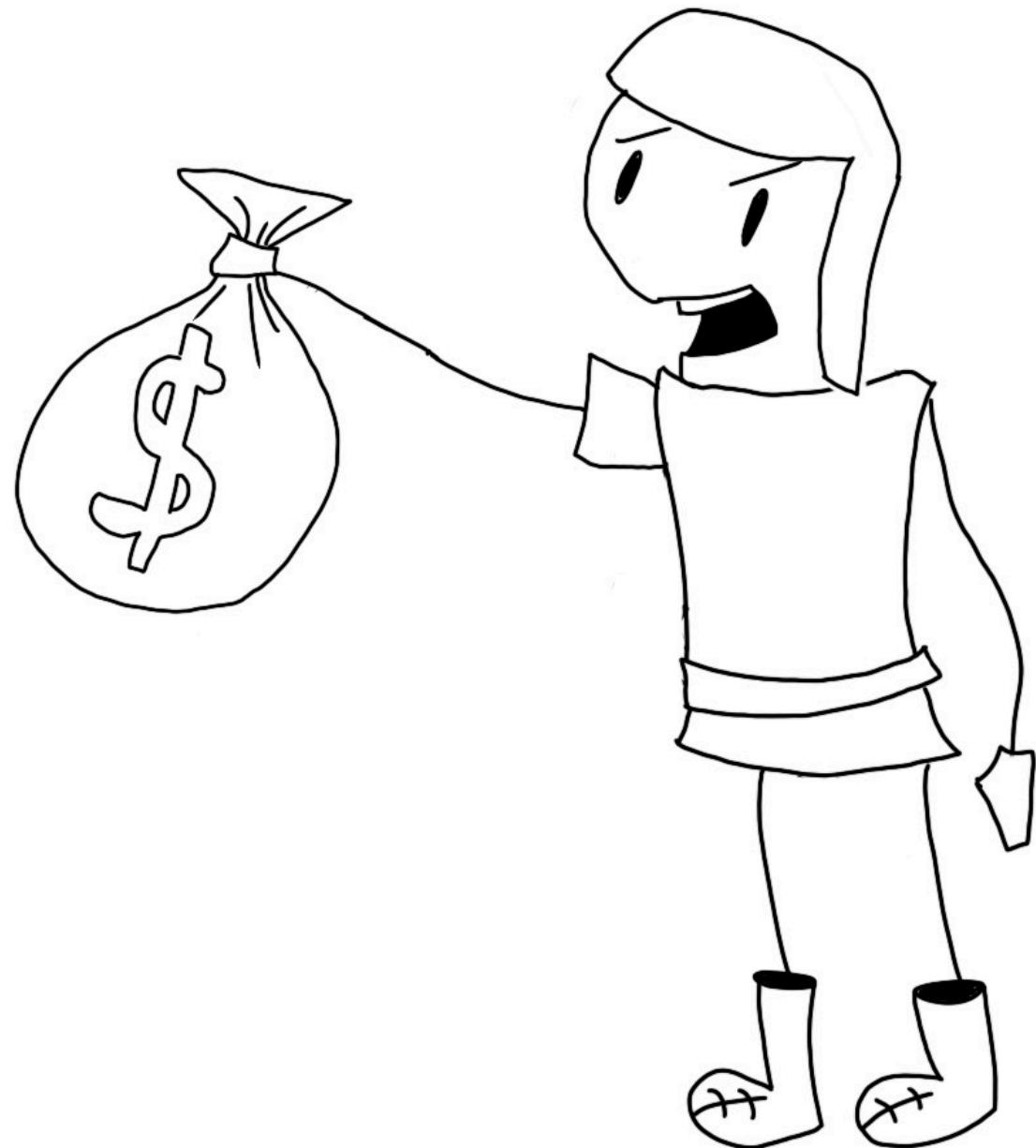


# DEGREE CENTRALITY









# TOWN CHARTER

## I. LEGISLATIVE DEPARTMENT

### I. MAYOR

## II. ADMINISTRATIVE DEPARTMENT

### I. MAYOR'S AIDE

## III. PUBLIC WORKS

### I. WATER WELL MAINTENANCE

### II. ROADS

### III. ANIMAL CONTROL

## IV. EMERGENCY SERVICES

### I. FIRE

### II. MEDICAL

### III. HAZARDOUS MAGIC

# COMMUNITY DETECTION

- PRIM'S ALGORITHM

I.I

IV.I

IV.III

II.I

IV.II

III.I

III.II

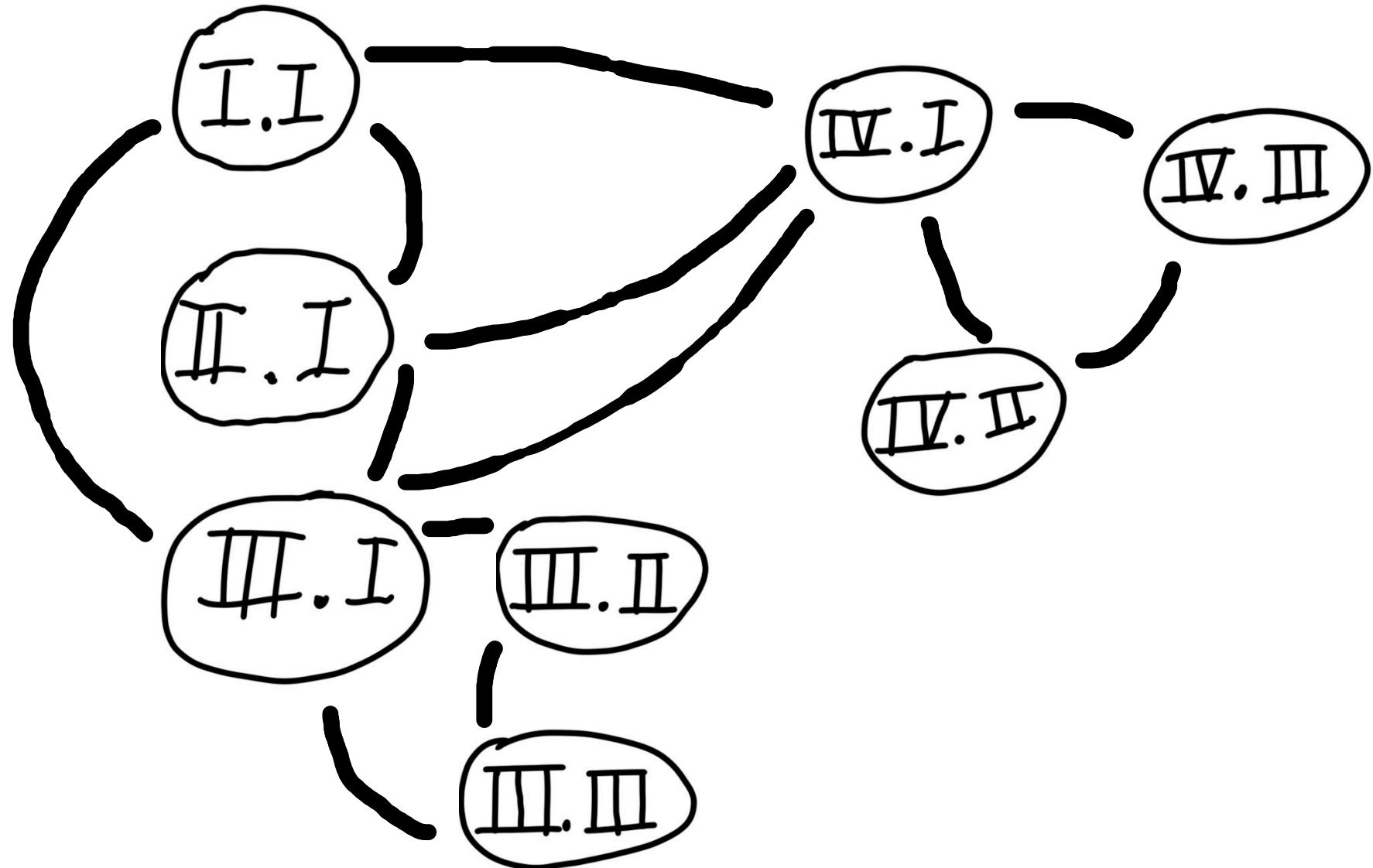
III.III

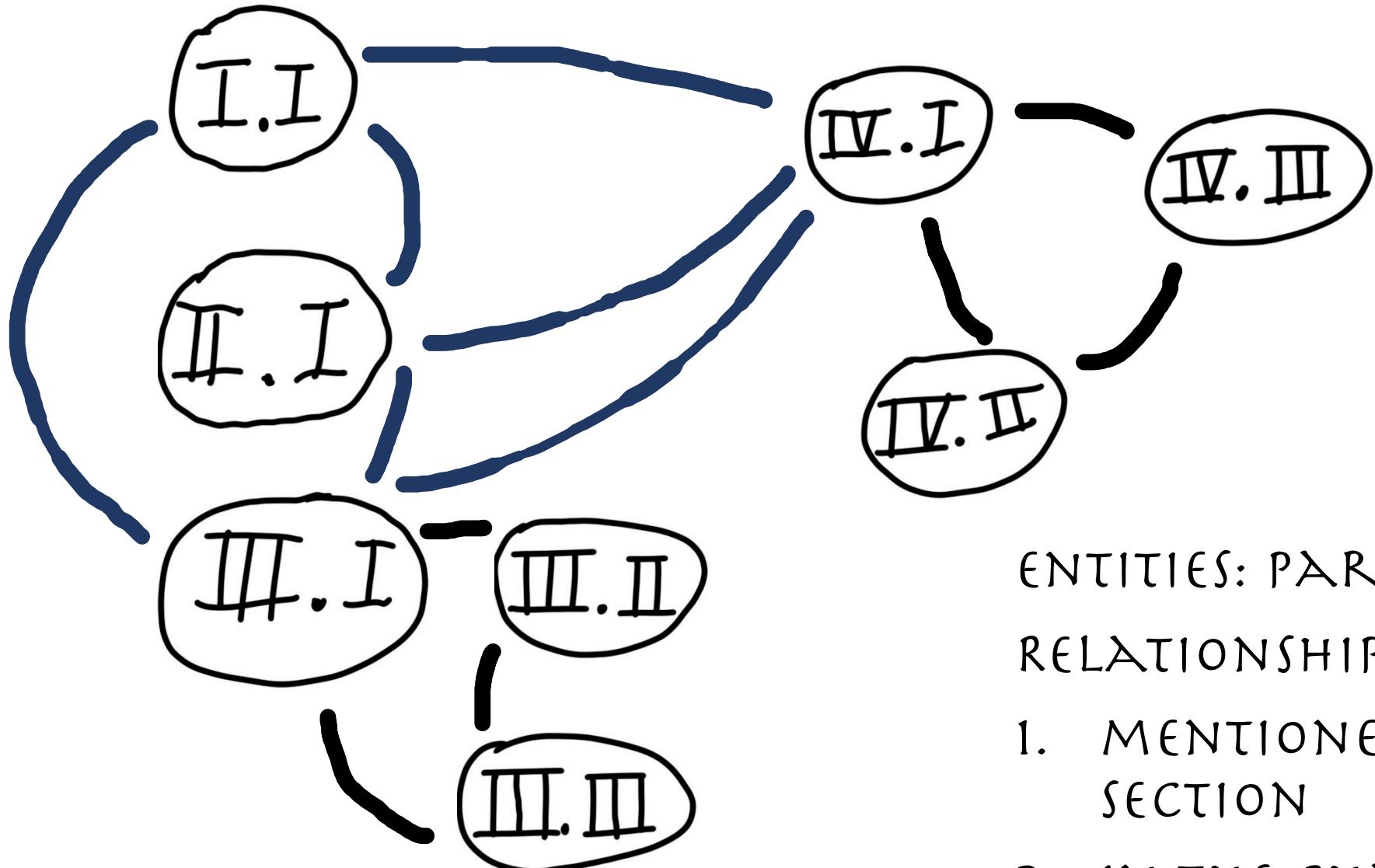
I.I

II.I

III.I  
III.II  
III.III

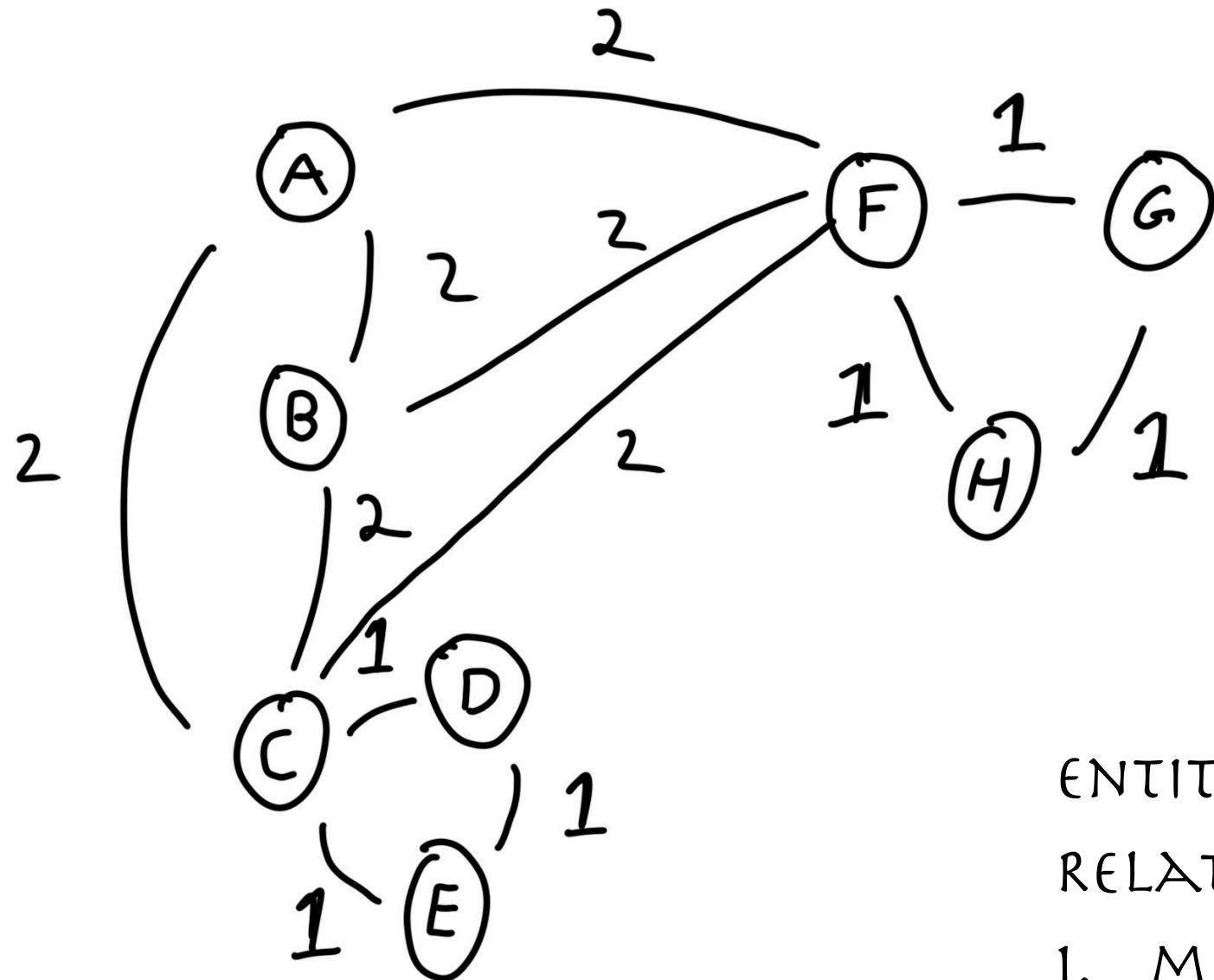
IV.I  
IV.II  
IV.III





ENTITIES: PARAGRAPH  
RELATIONSHIPS:

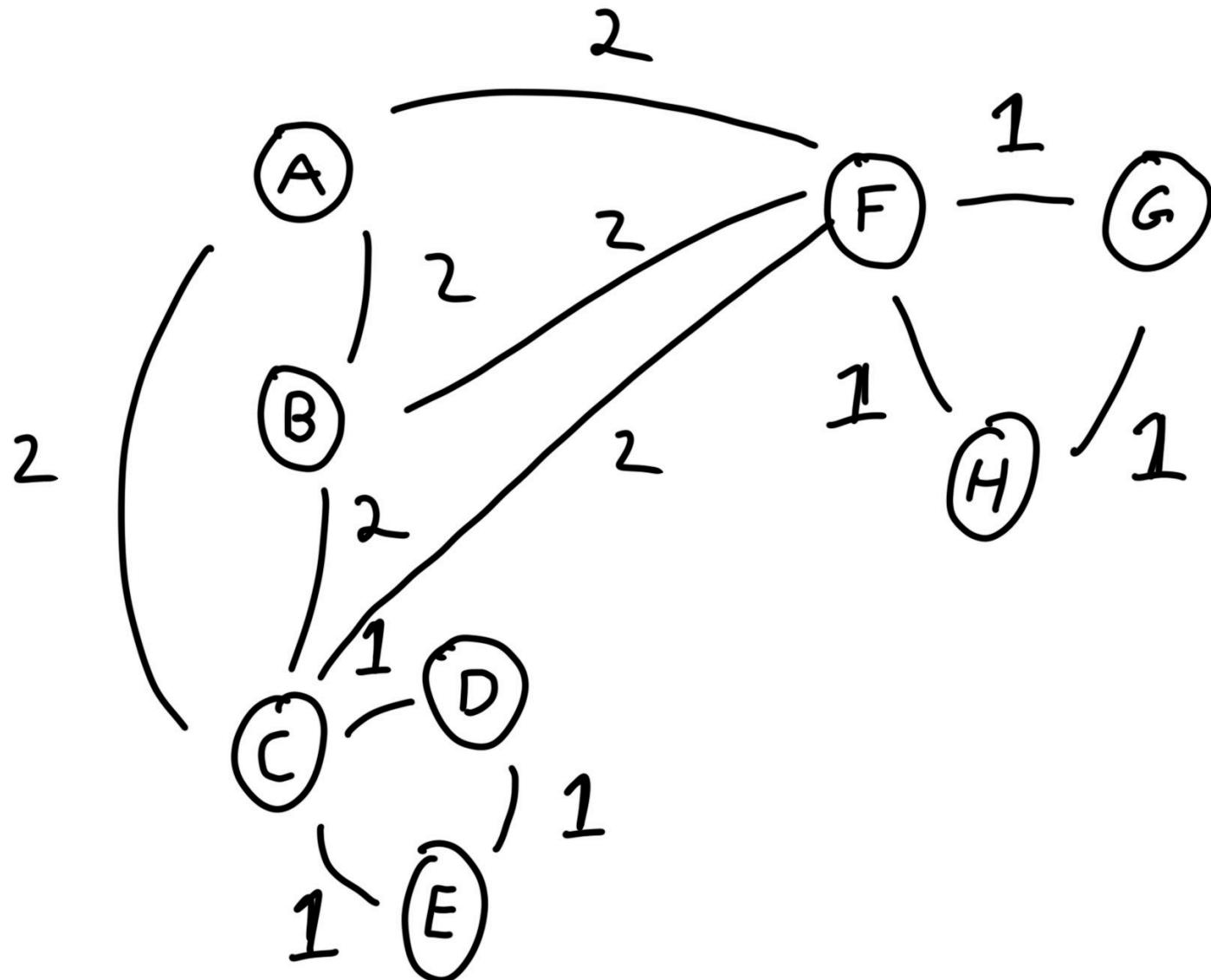
1. MENTIONED IN THE SAME SECTION
2. IN THE CHARTER



ENTITIES: PARAGRAPH

RELATIONSHIPS:

1. MENTIONED IN THE SAME SECTION
2. IN THE CHARTER



$AB$	2
$AC$	2
$AF$	2
$BC$	2
$BF$	2
$CF$	2
$CD$	1
$CE$	1
$DE$	1
$FG$	1
$FH$	1
$GH$	1

(A)

(F)

(G)

(B)

RECONNECT THE GRAPH,  
WITH AS FEW CONNECTIONS  
(BY WEIGHT) AS POSSIBLE

(C)

(D)

(E)

AB	2
AC	2
AF	2
BC	2
BF	2
CF	2
CD	1
CE	1
DE	1
FG	1
FH	1
GH	1

(A)

(B)

(C)

(D)

(E)

(F)

(G)

(H)

A B	2
A C	2
A F	2
B C	2
B F	2
C F	2
C D	1
C E	1
D E	1
F G	1
F H	1
<u>G H</u>	1

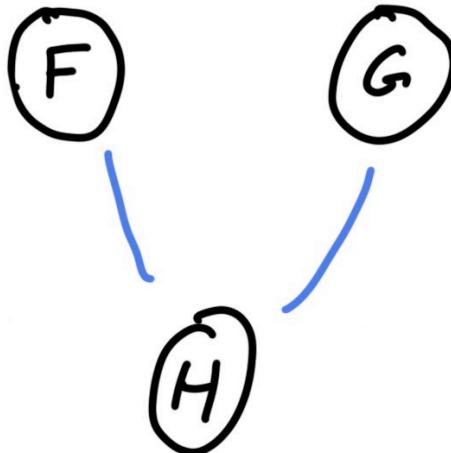
(A)

(B)

(C)

(D)

(E)



A B	2
A C	2
A F	2
B C	2
B F	2
C F	2
C D	1
C E	1
D E	1
F G	1
<u>F H</u>	1
<u>G H</u>	1

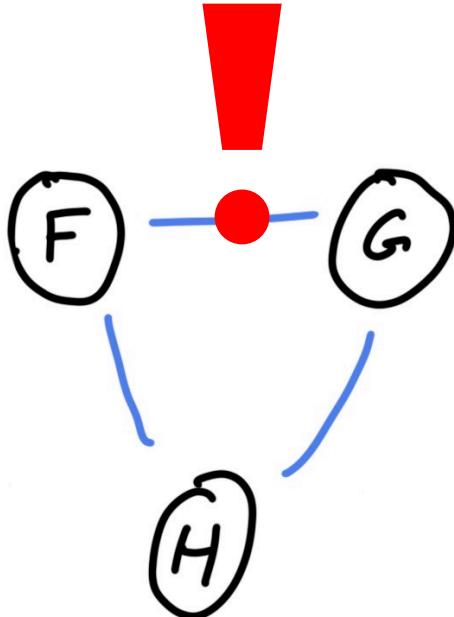
(A)

(B)

(C)

(D)

(E)



$AB$	2
$AC$	2
$AF$	2
$BC$	2
$BF$	2
$CF$	2
$CD$	1
$CE$	1
$DE$	1
$\cancel{FG}$	1
$\cancel{FH}$	1
$\cancel{GH}$	1

(A)

(B)

(C)

(D)

(E)

(F)

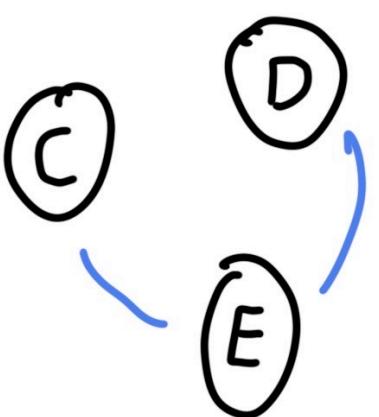
(G)

(H)

$AB$	2
$AC$	2
$AF$	2
$BC$	2
$\beta F$	2
$CF$	2
$CD$	1
$CE$	1
$DE$	1
$\times FG$	1
<del><math>FH</math></del>	1
<del><math>GH</math></del>	1

(A)

(B)

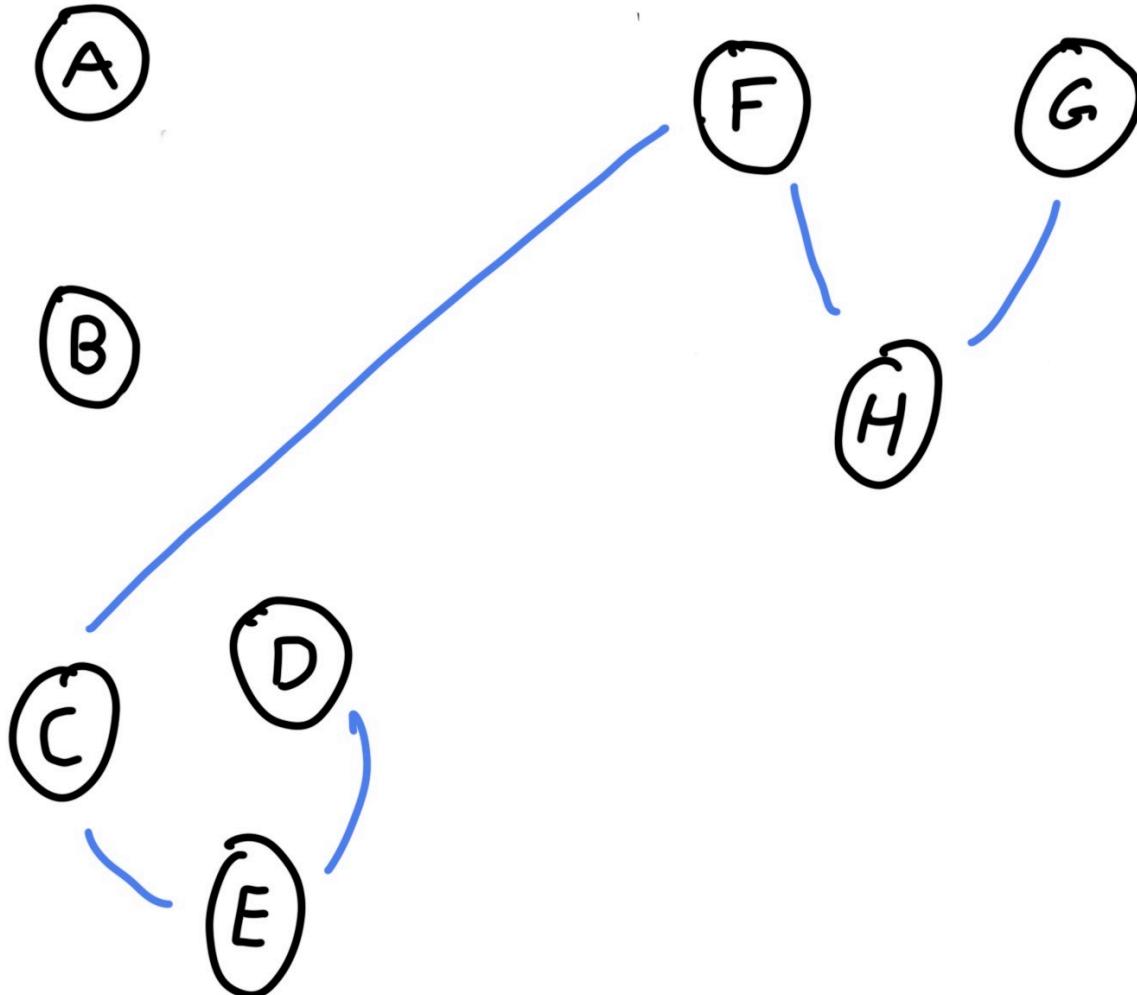


(F)

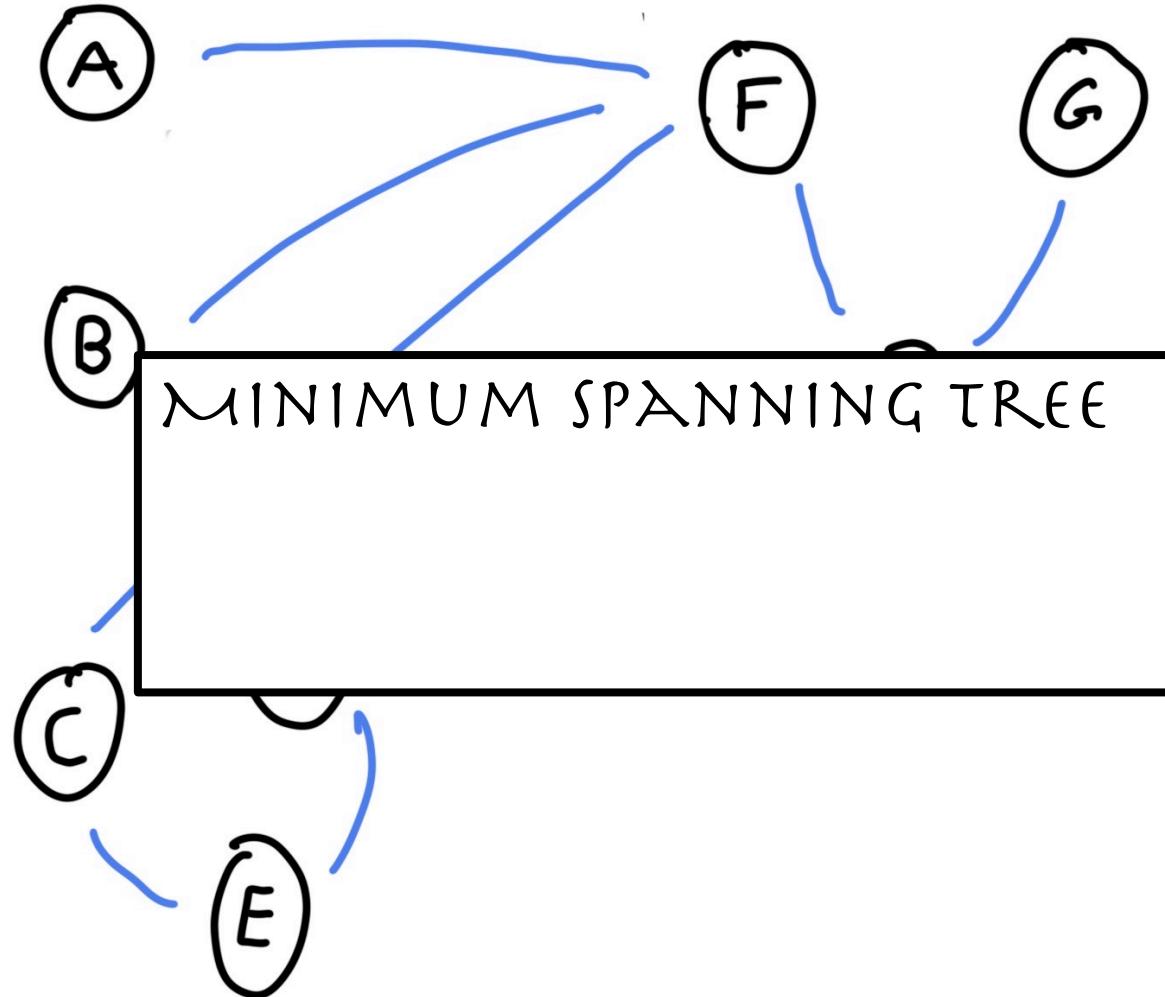
(G)

(H)

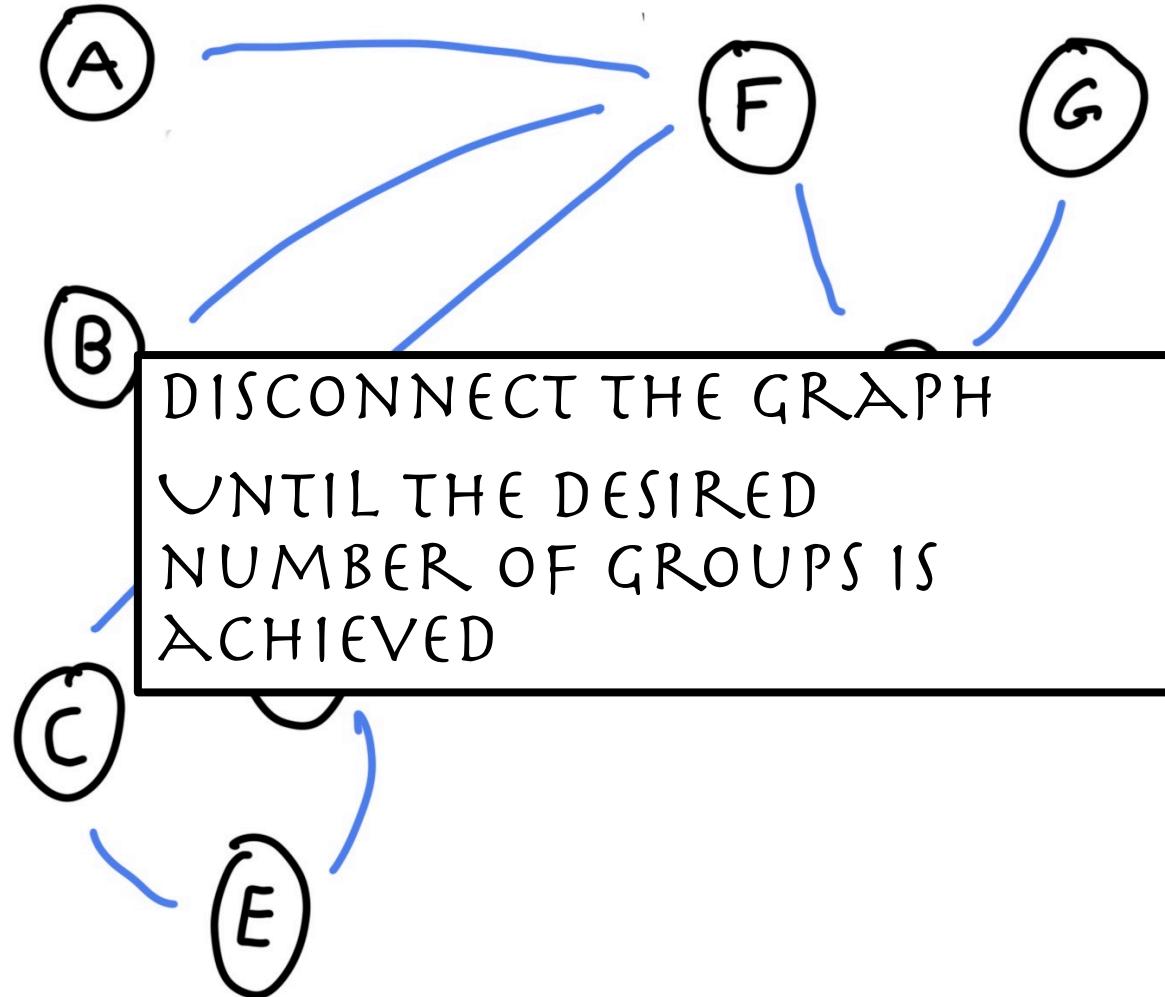
$AB$	2
$AC$	2
$AF$	2
$BC$	2
$\beta F$	2
$CF$	2
<del><math>CD</math></del>	1
<del><math>CE</math></del>	1
<del><math>DE</math></del>	1
<del><math>FG</math></del>	1
<del><math>FH</math></del>	1
<del><math>GH</math></del>	1



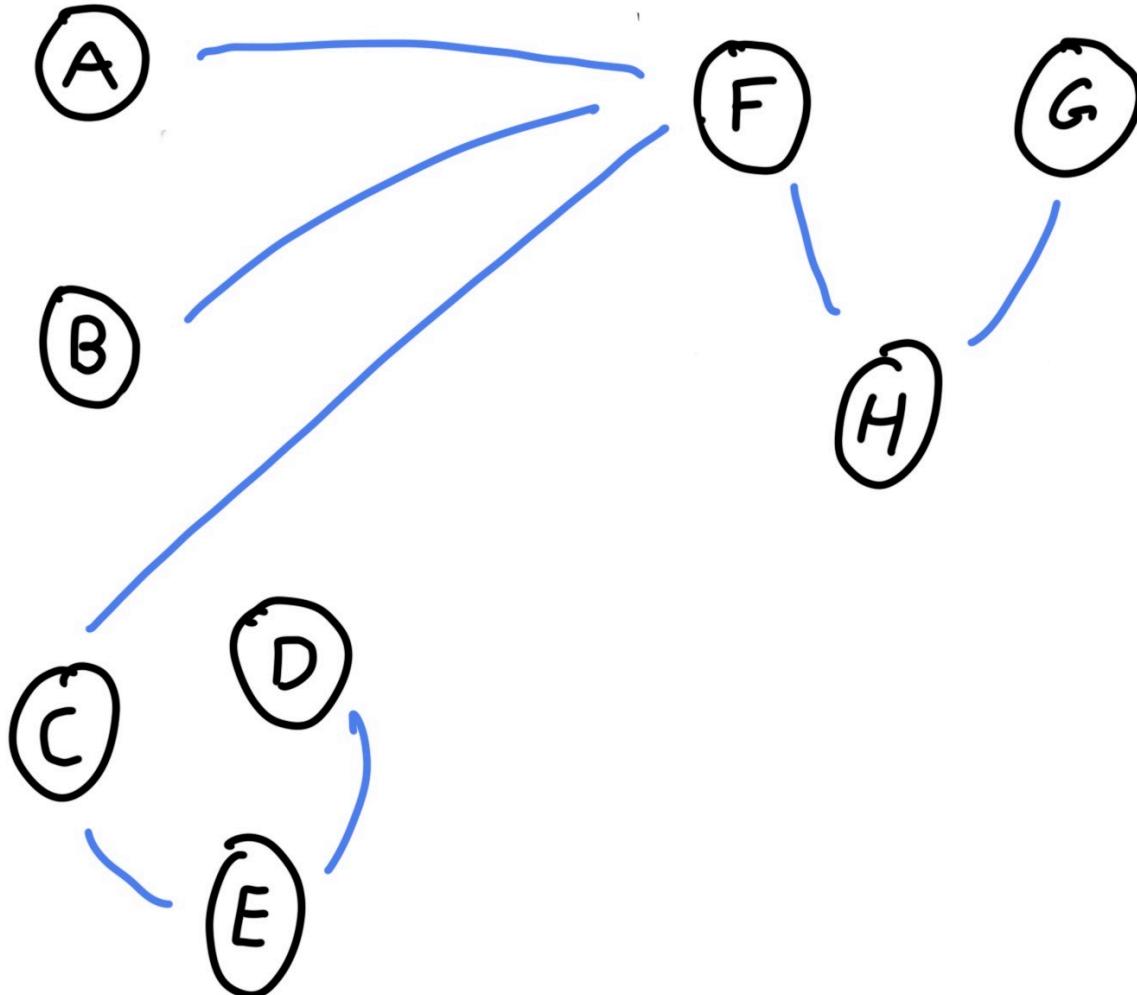
$AB$	2
$AC$	2
$AF$	2
$BC$	2
$\beta F$	2
<del><math>CF</math></del>	2
<del><math>CD</math></del>	1
<del><math>CE</math></del>	1
<del><math>DE</math></del>	1
<del><math>FG</math></del>	1
<del><math>FH</math></del>	1
<del><math>GH</math></del>	1



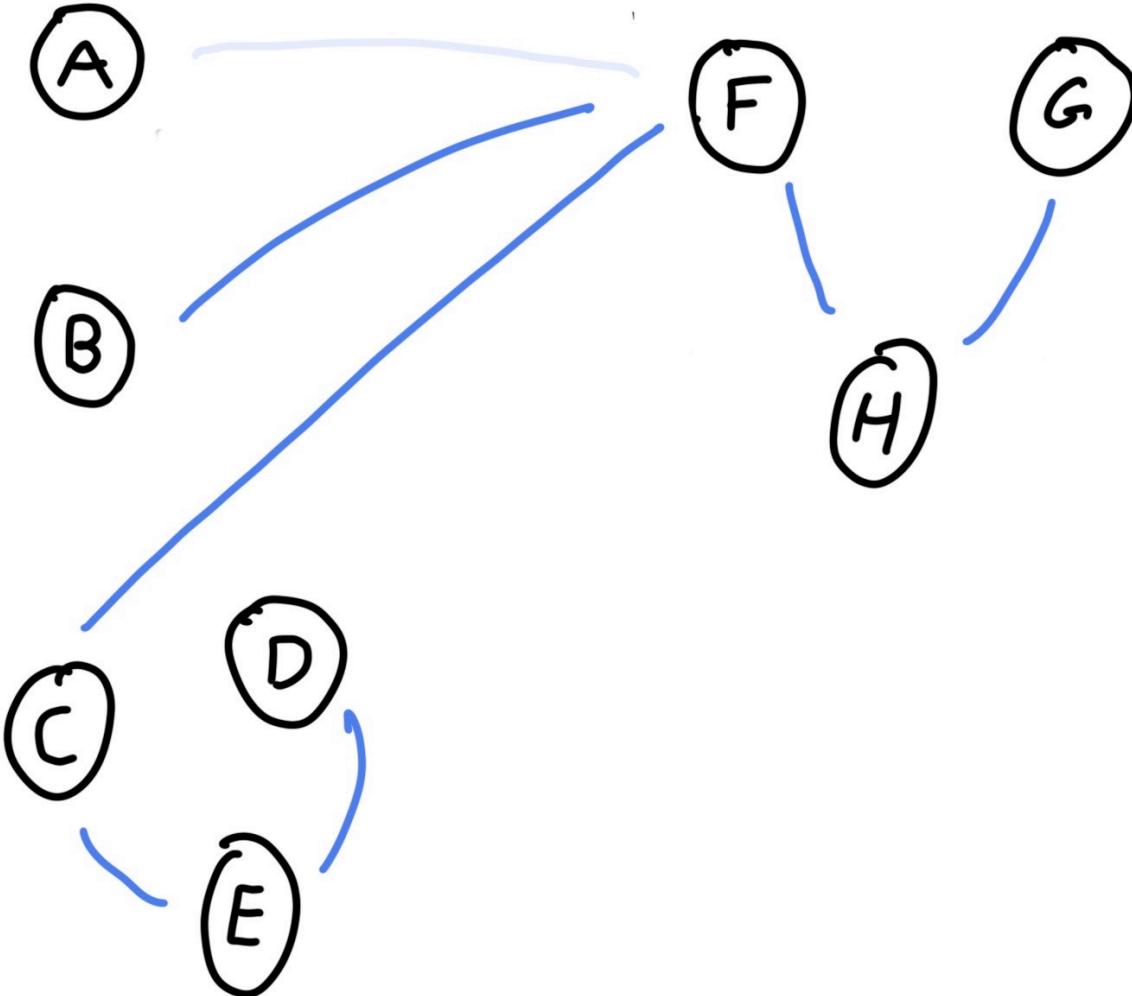
<del>AB</del>	2
<del>AC</del>	2
<del>AF</del>	2
<del>BC</del>	2
<del>BF</del>	2
<del>CF</del>	2
<del>CD</del>	1
<del>CE</del>	1
<del>DE</del>	1
<del>FG</del>	1
<del>FH</del>	1
<del>GH</del>	1



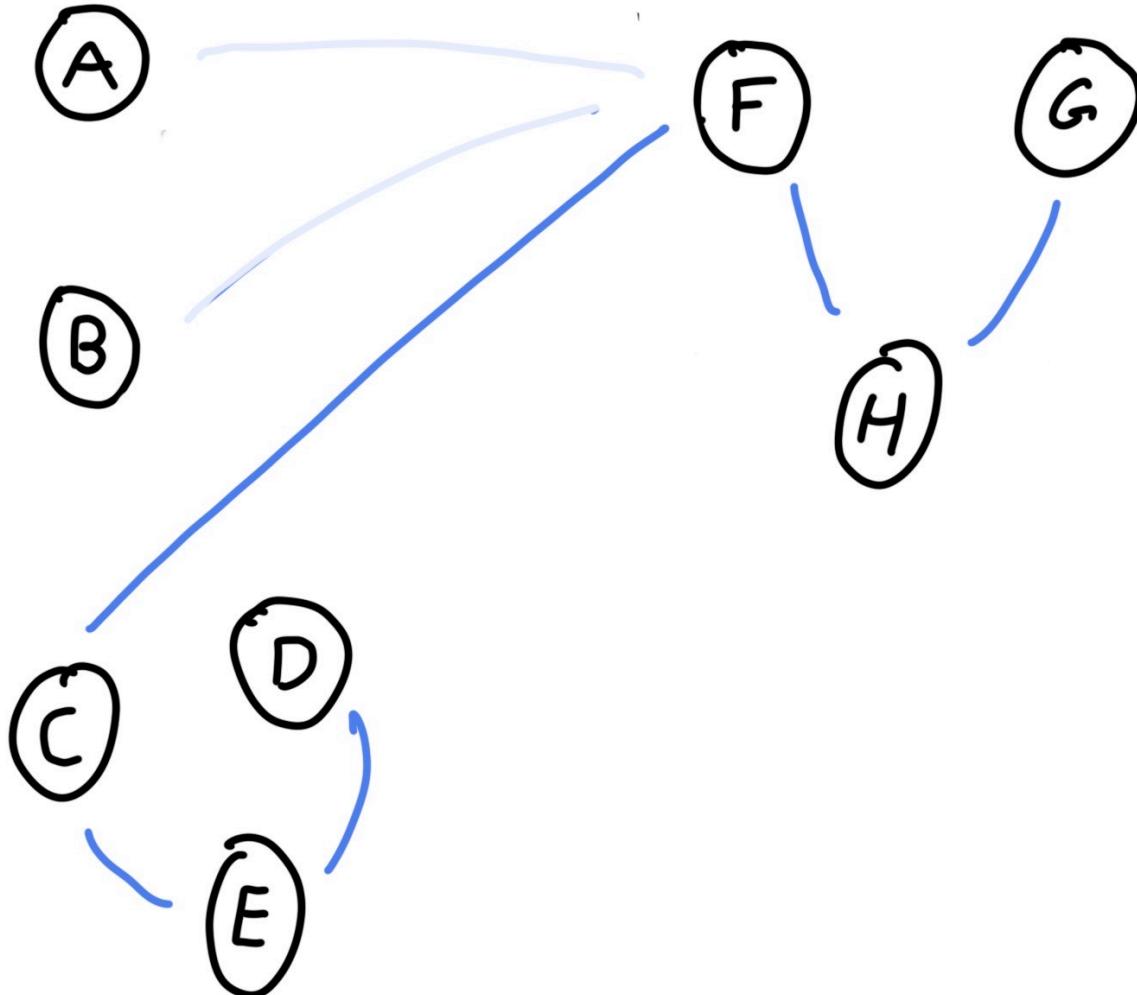
<del>X</del> AB	2
<del>X</del> AC	2
<del>AF</del>	2
<del>X</del> BC	2
<del>BF</del>	2
<del>CF</del>	2
<del>X</del> CD	1
<del>CE</del>	1
<del>DE</del>	1
<del>X</del> FG	1
<del>FH</del>	1
<del>GH</del>	1



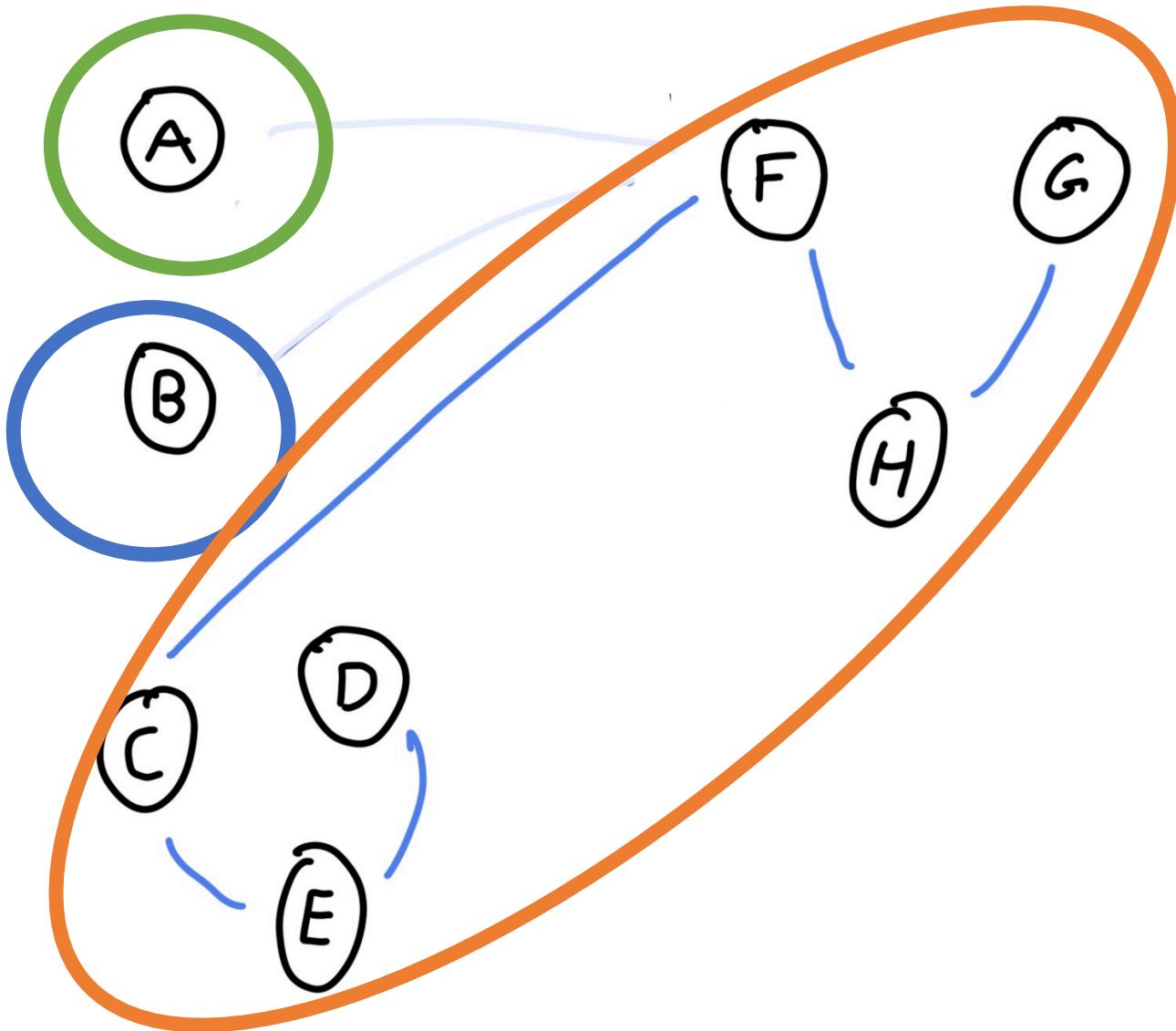
<del>X</del> AB	2
<del>X</del> AC	2
<del>AF</del>	2
<del>X</del> BC	2
<del>BF</del>	2
<del>CF</del>	2
<del>X</del> CD	1
<del>CE</del>	1
<del>DE</del>	1
<del>X</del> FG	1
<del>FH</del>	1
<del>GH</del>	1



<del>X</del> AB	2
<del>X</del> AC	2
<del>AF</del>	2
<del>X</del> BC	2
<del>BF</del>	2
<del>CF</del>	2
<del>X</del> CD	1
<del>CE</del>	1
<del>DE</del>	1
<del>X</del> FG	1
<del>FH</del>	1
<del>GH</del>	1



<del>XAB</del>	2
<del>XAC</del>	2
<del>AF</del>	2
<del>XBC</del>	2
<del>BF</del>	2
<del>CF</del>	2
<del>XCD</del>	1
<del>CE</del>	1
<del>DE</del>	1
<del>XFG</del>	1
<del>FH</del>	1
<del>GH</del>	1



X AB	2
X AC	2
AF	2
X BC	2
BF	2
CF	2
X CD	1
CE	1
DE	1
X FG	1
FH	1
GH	1

# TOWN CHARTER

## I. LEGISLATIVE DEPARTMENT

### I. MAYOR

## II. ADMINISTRATIVE DEPARTMENT

### I. MAYOR'S AIDE

## III. PUBLIC WORKS

### I. WATER WELL MAINTENANCE

### II. ROADS

### III. ANIMAL CONTROL

## IV. EMERGENCY SERVICES

### I. FIRE

### II. MEDICAL

### III. HAZARDOUS MAGIC

# WHAT HAPPENS WHEN GRAPH ALGORITHMS FAIL?

- EXPECTATIONS
  - WHAT DOES SUCCESS/FAILURE LOOK LIKE?
- ALGORITHM
  - IS YOUR ALGORITHM IMPLEMENTED CORRECTLY
  - ARE THERE SIMILAR ALGORITHMS
- ENTITIES/RELATIONSHIPS
  - REDEFINE WHAT A NODE/EDGE IS

Graphs are a ~~collection of nodes and edges~~

- Graphs encode systems as Entities and Relationships
- Breadth First Search finds nodes in the fewest hops
  - Remember to track visited nodes!
- Graph databases are resilient to on-the-fly changes
- Paths
  - Dfs/BFS/Dijkstra
- Connections
  - Centrality (degree, eigenvector, closeness, betweenness)
- Communities
  - Prim's
  - Louvain, K-L partition,

