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

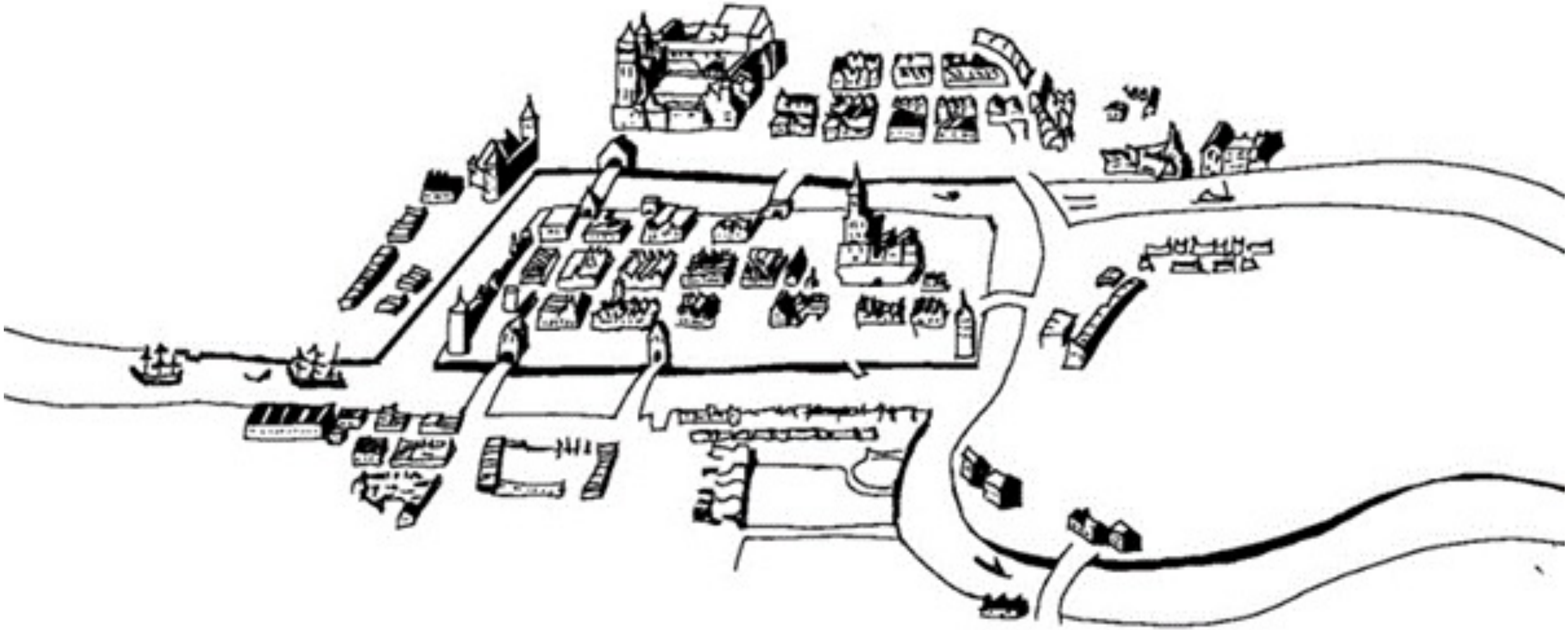
- Graphs
- Python
- Drawing
- Sometimes DMing

About you

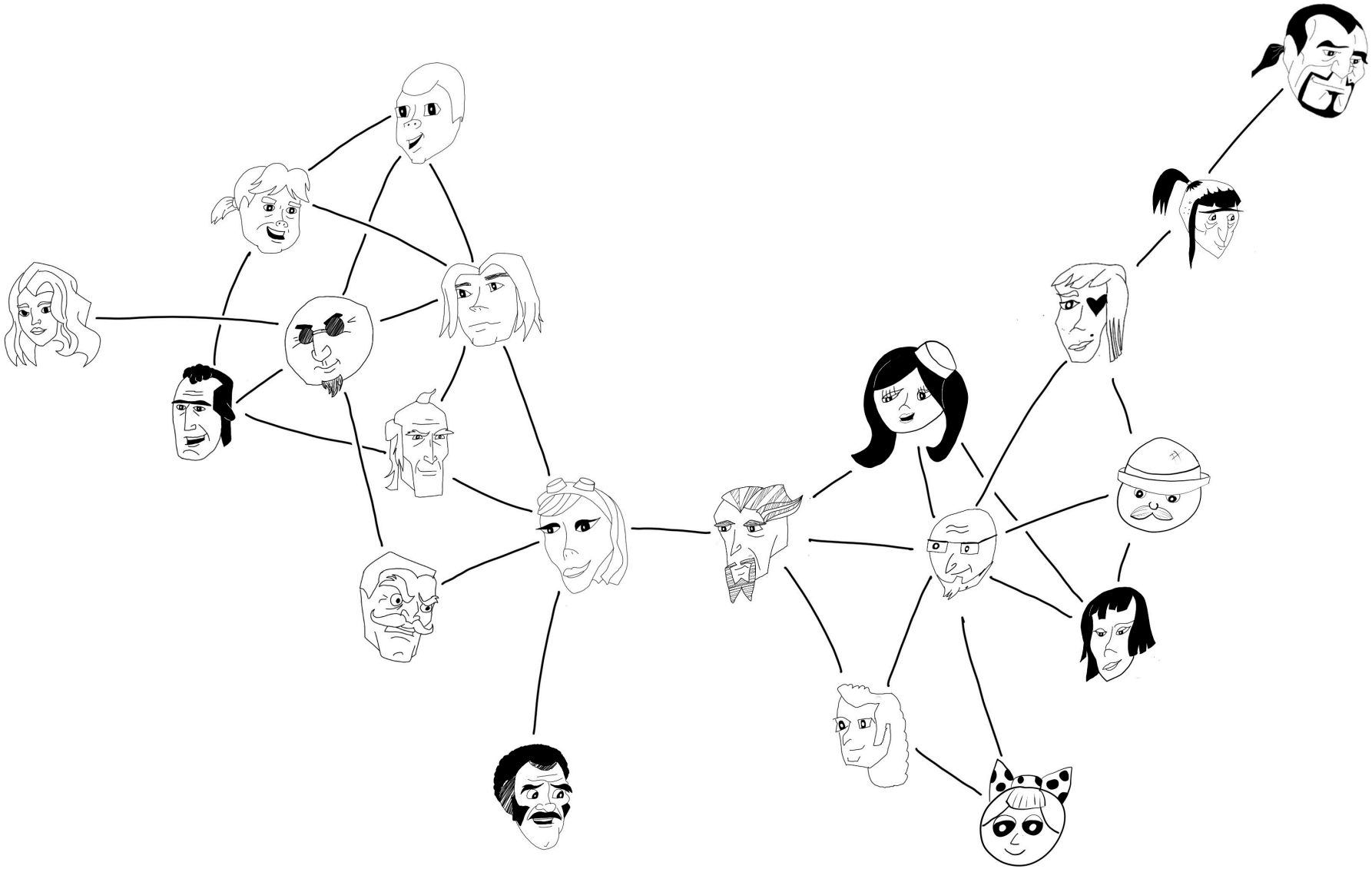
- Some programming
  - Maps/Dictionaries
  - Objects
  - Maybe have heard of or occasionally use graphs and want to dive deeper
- Some math
  - Matrices
  - Logarithms

What are graphs?

What are graphs

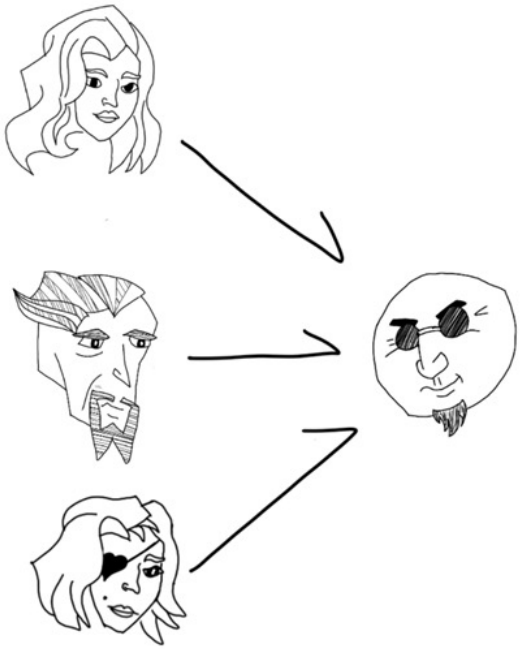


What are graphs

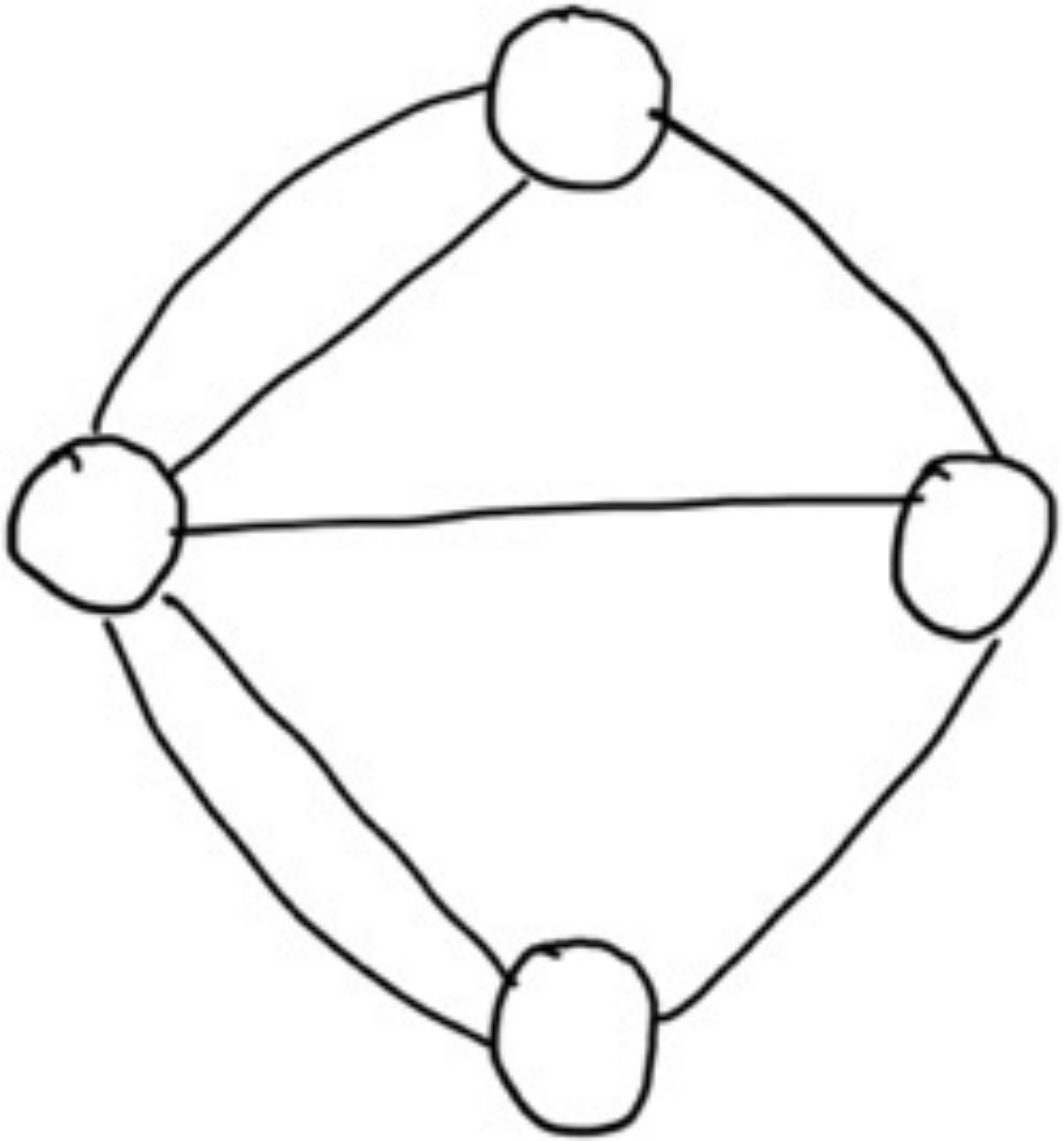


What are graphs

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



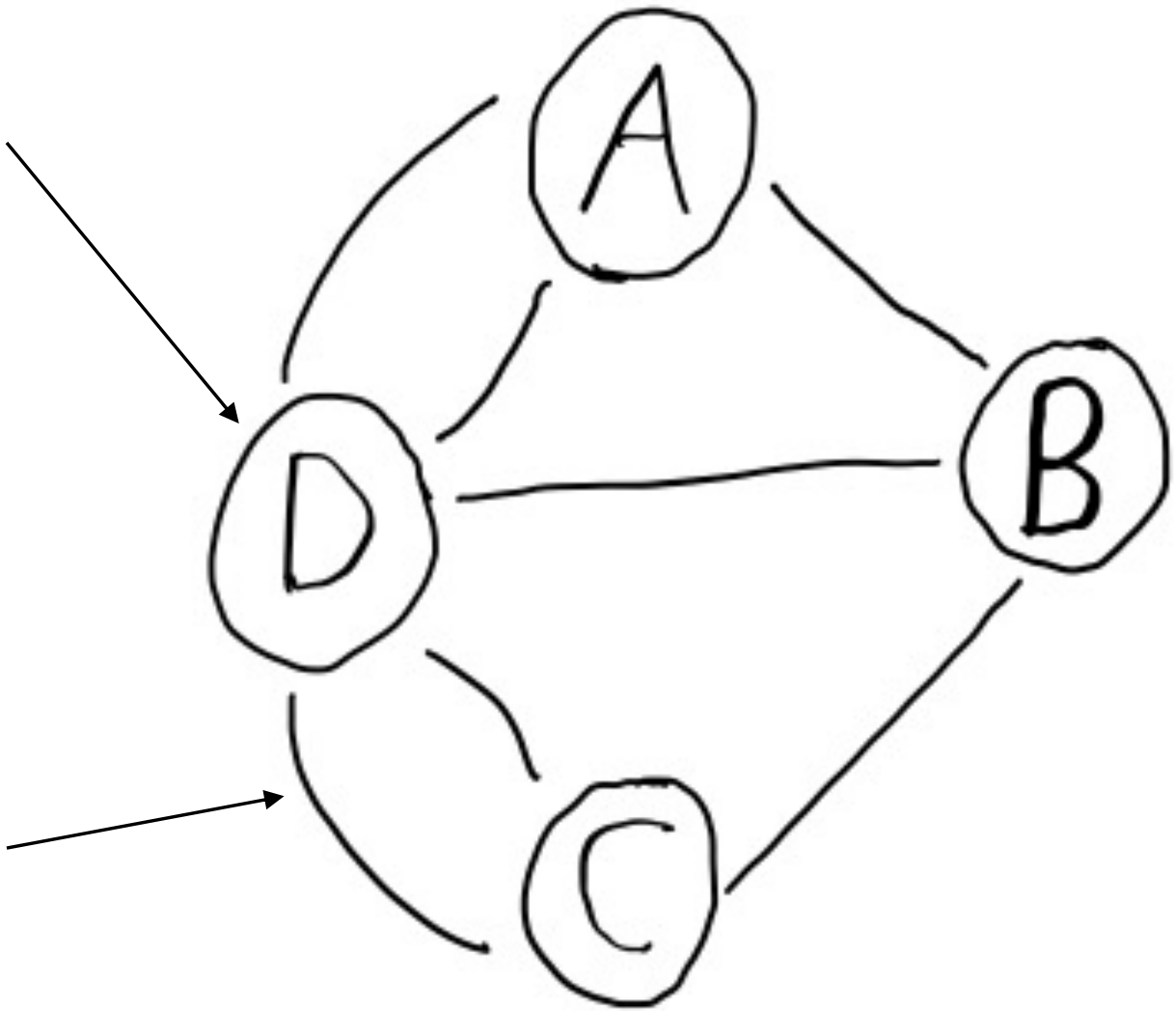
What are graphs?



What are graphs?

Node: Fundamental indivisible unit from which graphs are formed.

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

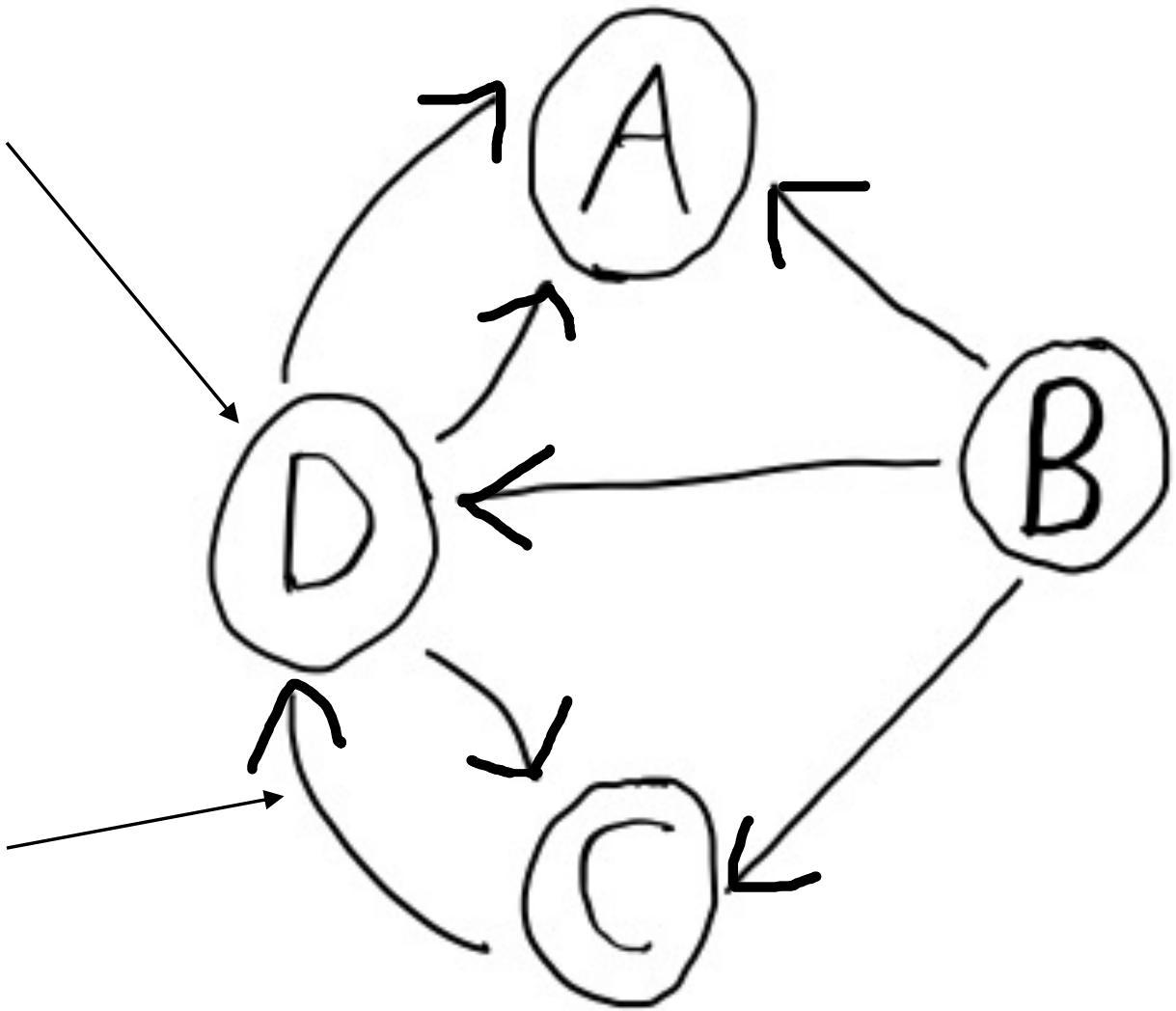


A graph is a collection of nodes and edges

What are graphs?

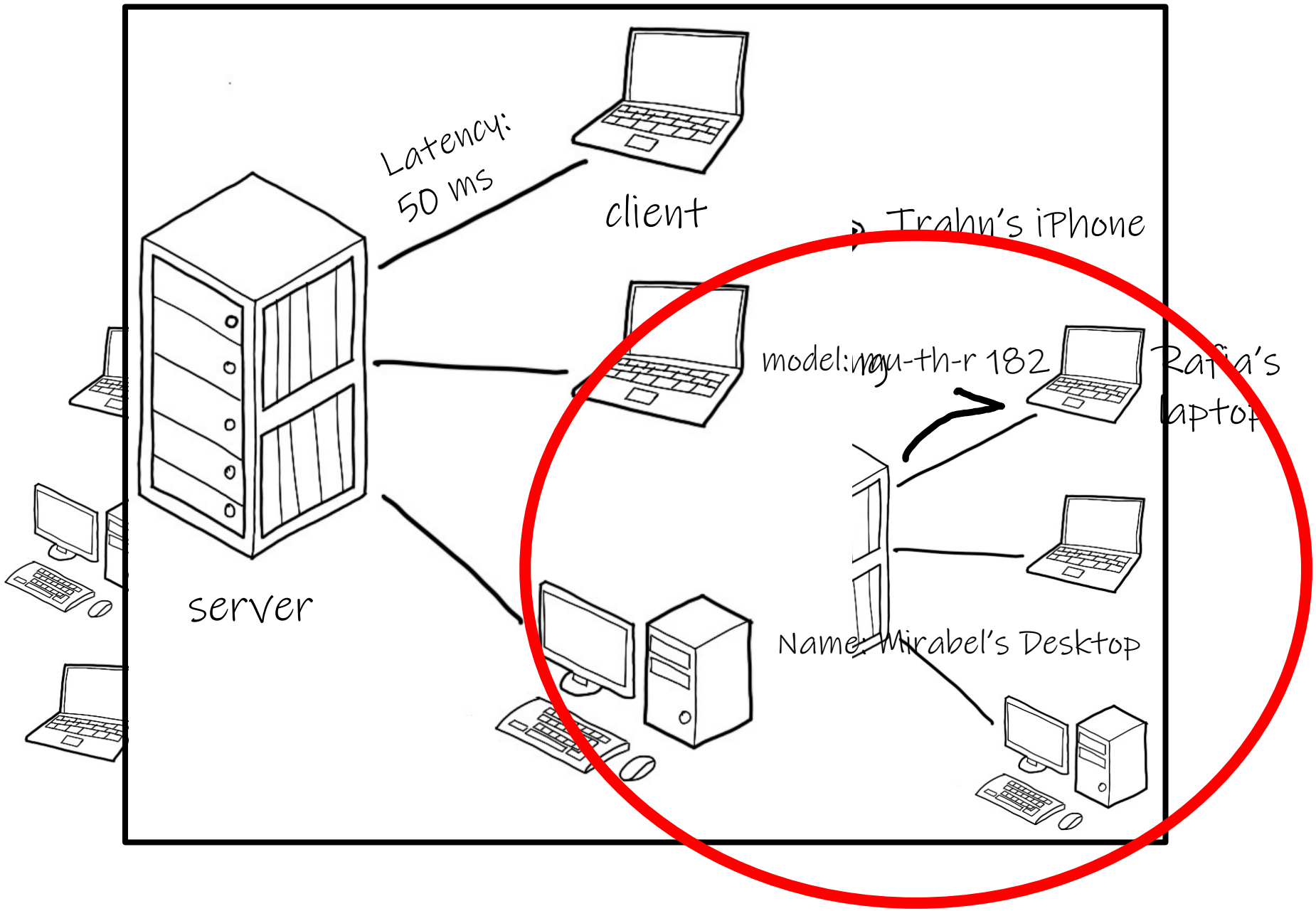
Node: Fundamental indivisible unit from which graphs are formed.

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

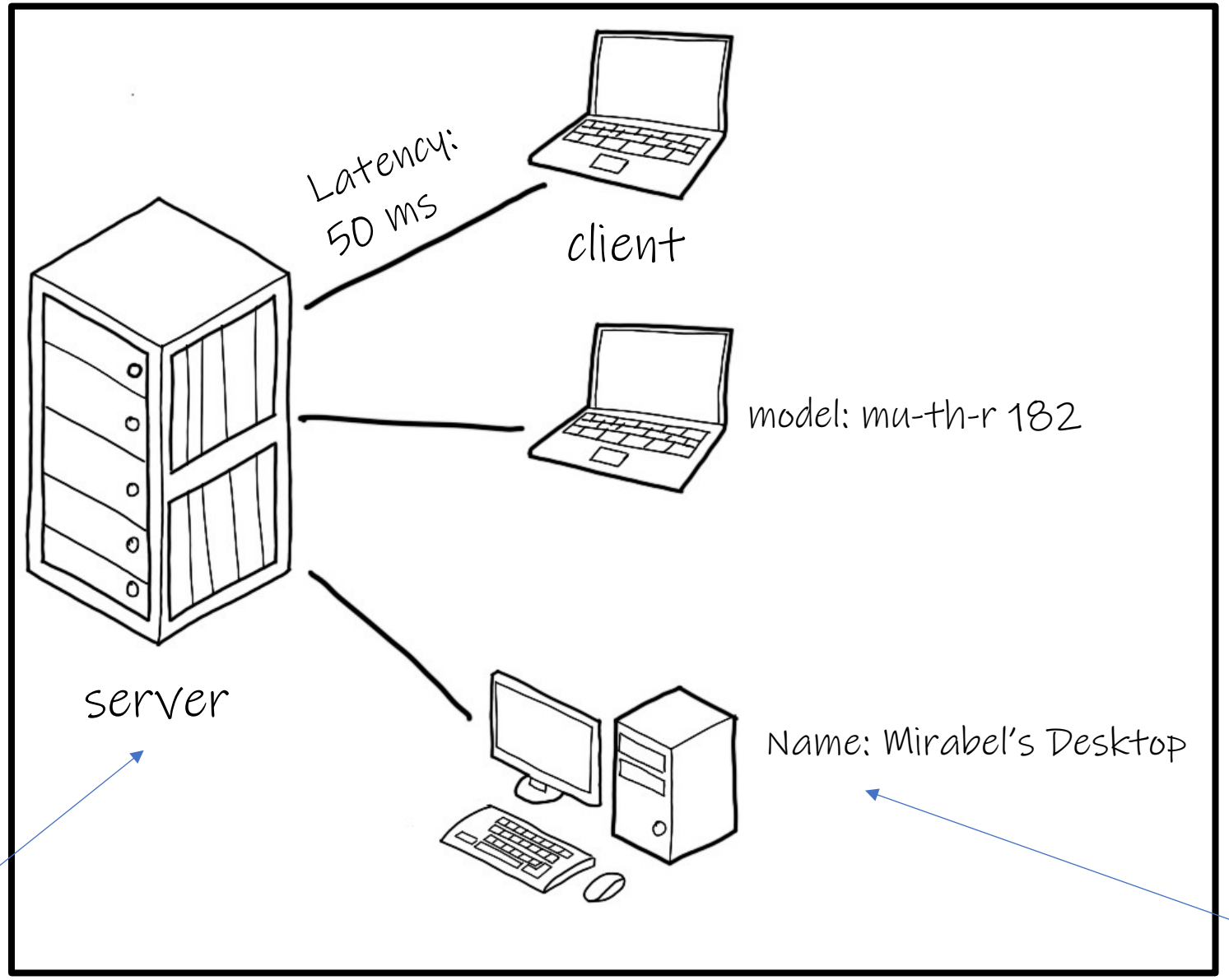


Directed graphs have one-way edges

Paths & subgraphs



Paths & subgraphs



Labels:
"class" of a node

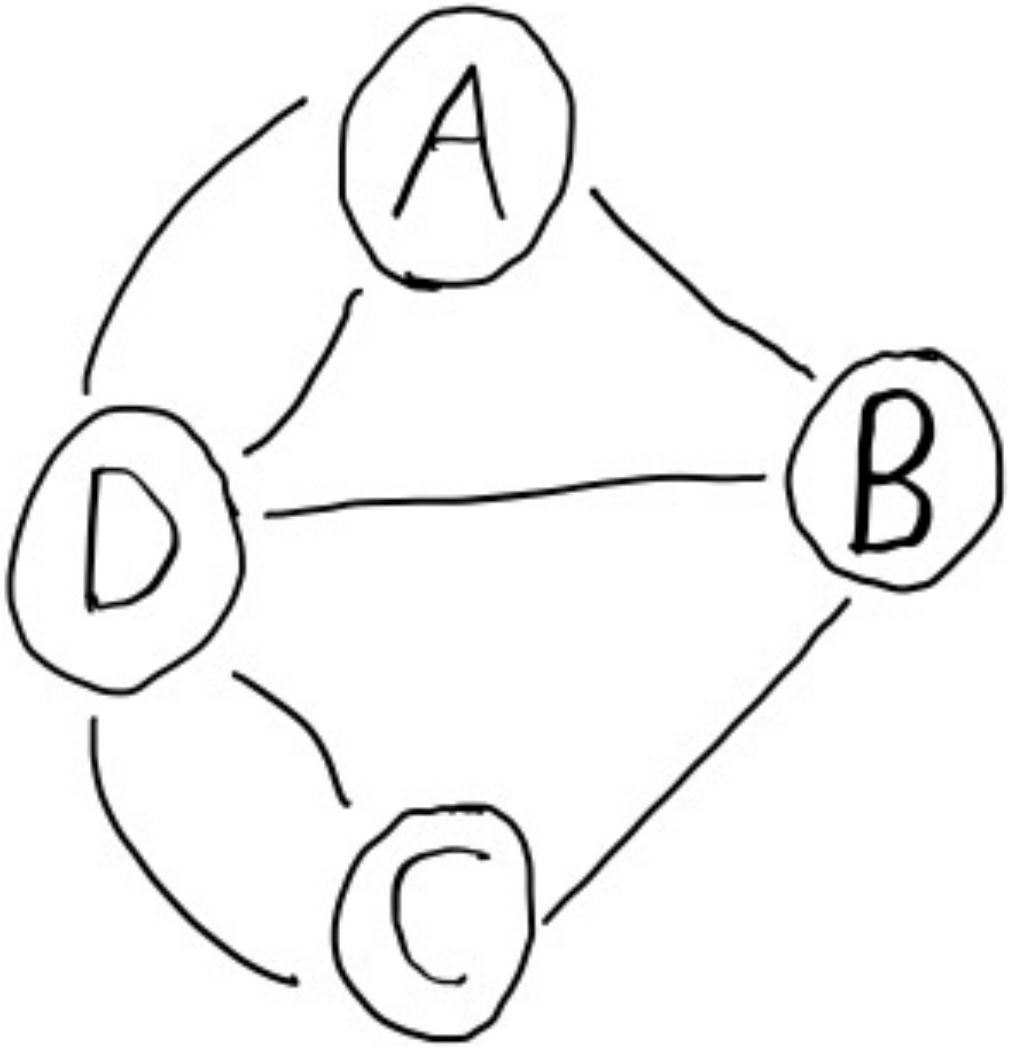
Property:
Any additional information attached to a node/edge

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

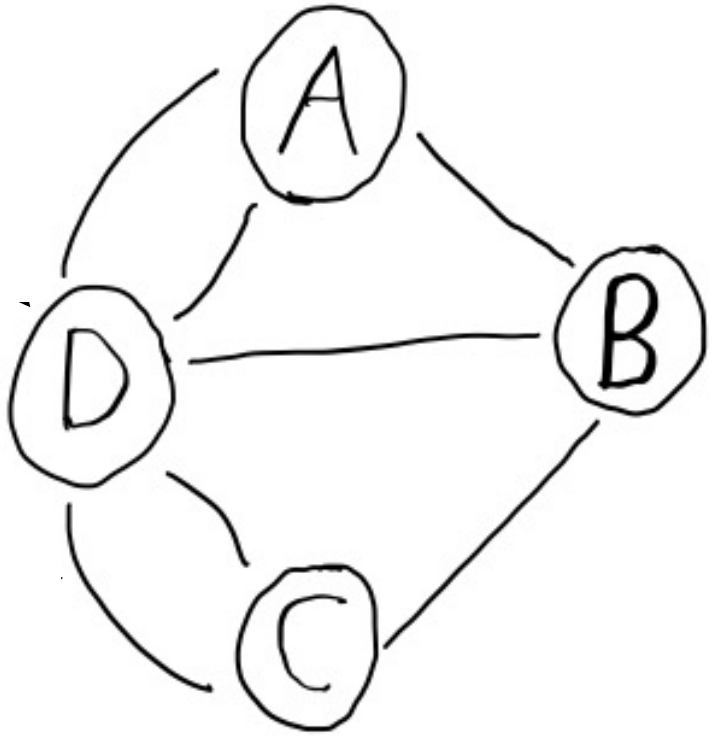
From \ To	A	B	C	D
A	0	1	0	2
B	1	0	1	1
C	0	1	0	2
D	2	1	2	0

```
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]
```

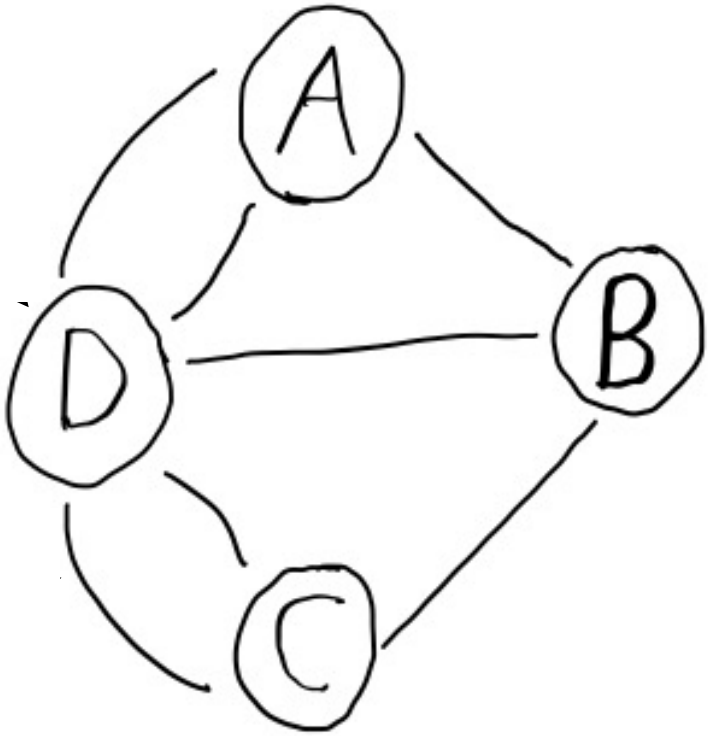
```
[[0 1 0 2]
[1 0 1 1]
[0 1 0 2]
[2 1 2 0]]
```



Representing graphs in Python

- Edge list

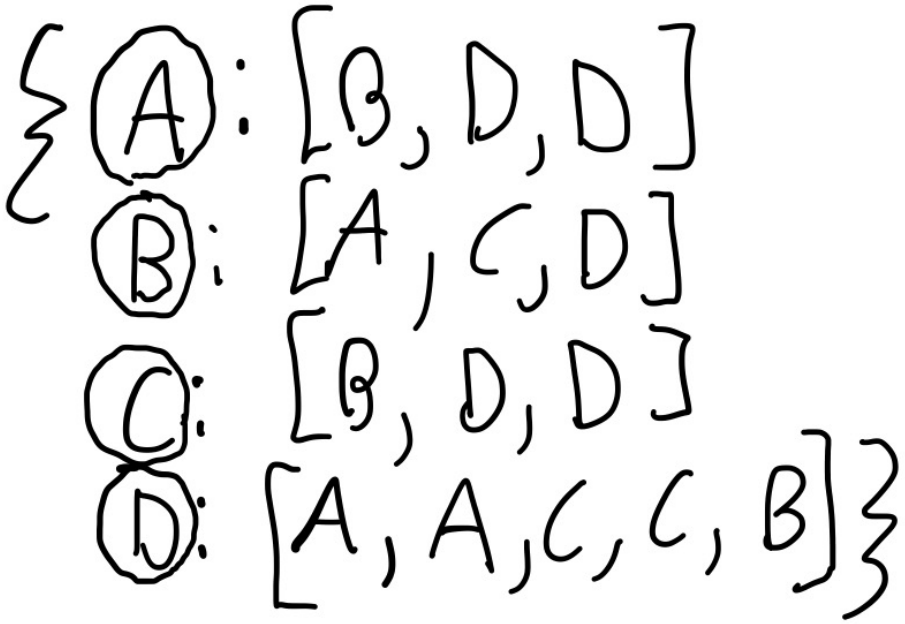
[(A,B),(A,D),(A,D),(B,C),
(B,D),(C,D),(C,D)]



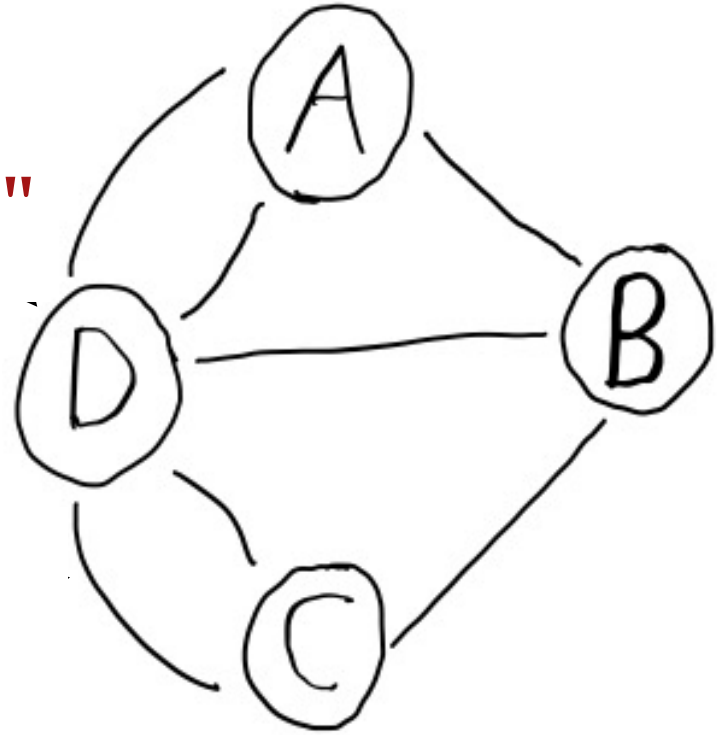

```
K3 =
[("A","B"),("A","D"),("A","D"),("B","C"),("B","D"),
("C","D"),("C","D")]
[('A', 'B'), ('A', 'D'), ('A', 'D'), ('B', 'C'),
('B', 'D'), ('C', 'D'), ('C', 'D')]
```

Representing graphs in Python

- Dictionary




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

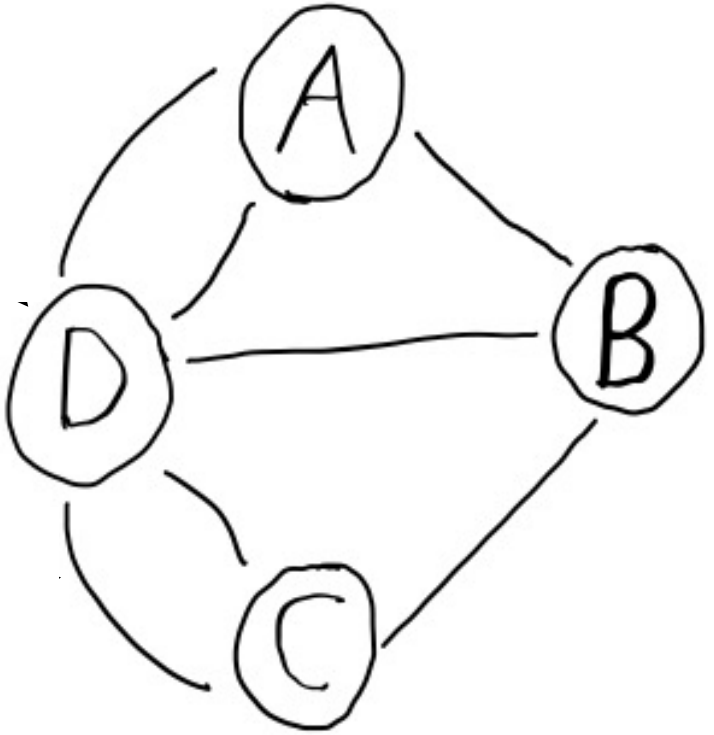



```
{'A': ['B', 'D', 'D'],
'B': ['A', 'C', 'D'],
'C': ['B', 'D', 'D'],
'D': ['A', 'A', 'C', 'C', 'B']}
```

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~~

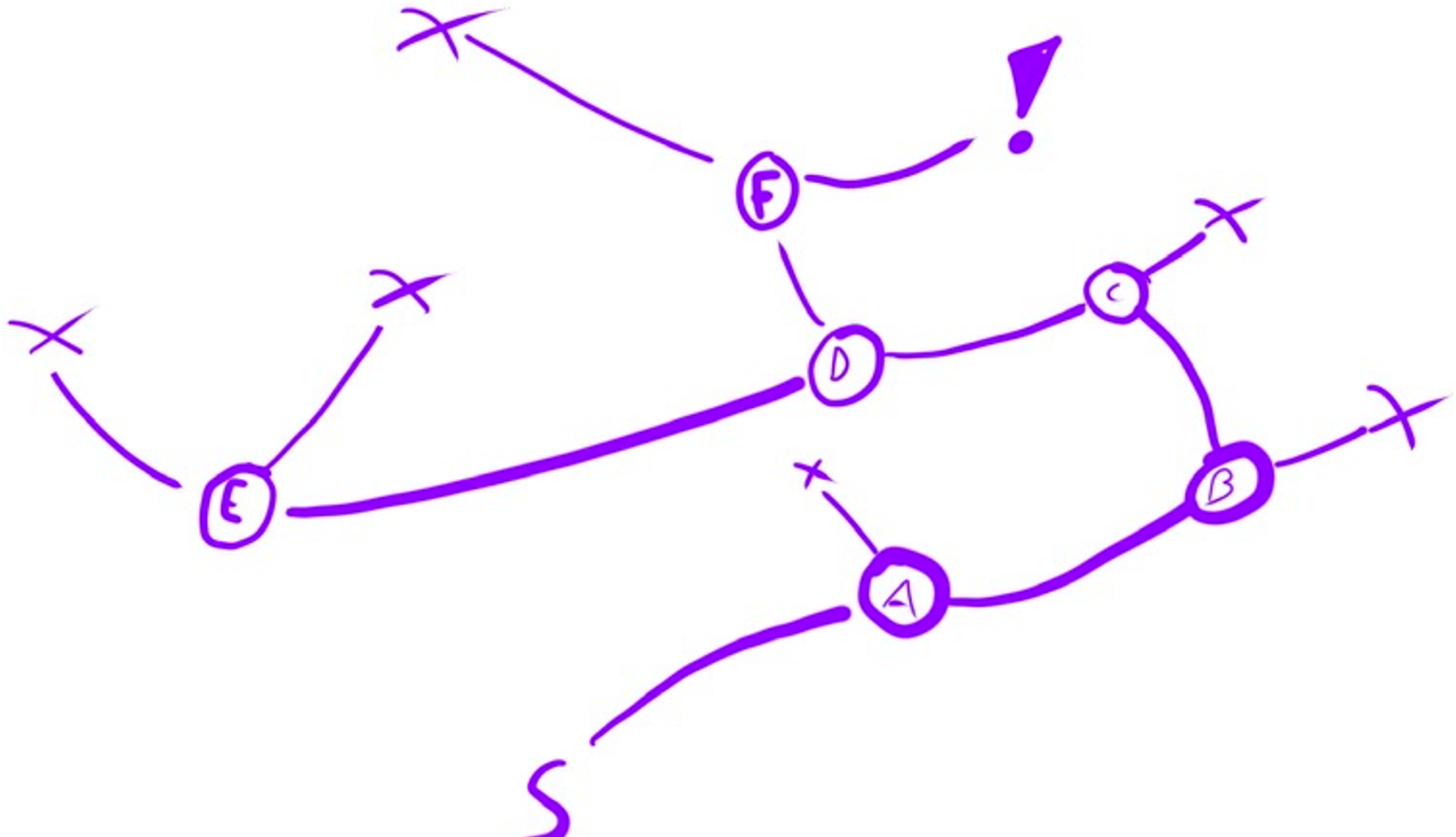
disclaimer

GET AN ~~ALE~~ LAGER

NOAQH, DEVIANTART

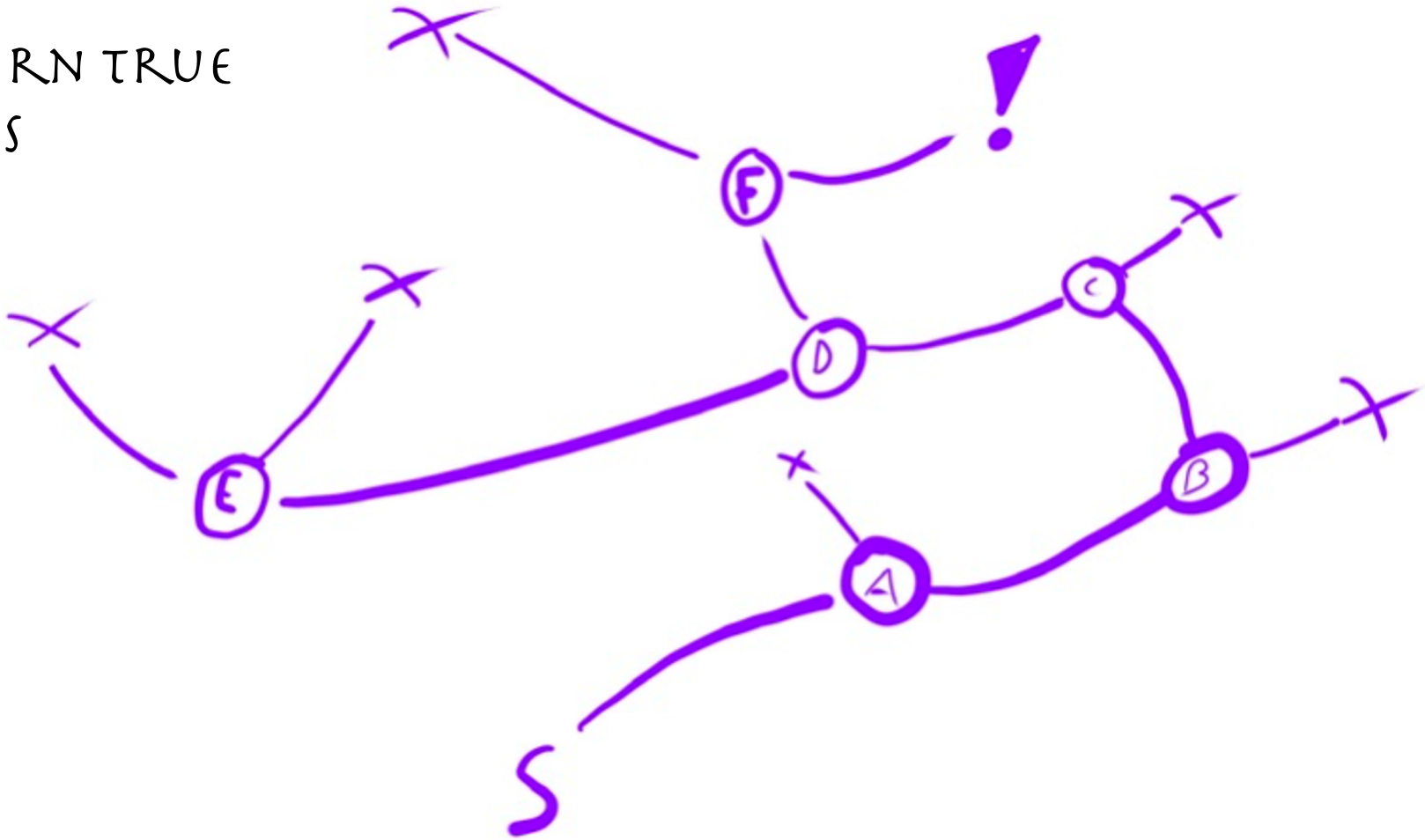
MAZES/MAPS CAN BE REPRESENTED AS GRAPHS!

- ENTITIES: INTERSECTIONS
- RELATIONSHIPS: PATHS BETWEEN INTERSECTIONS



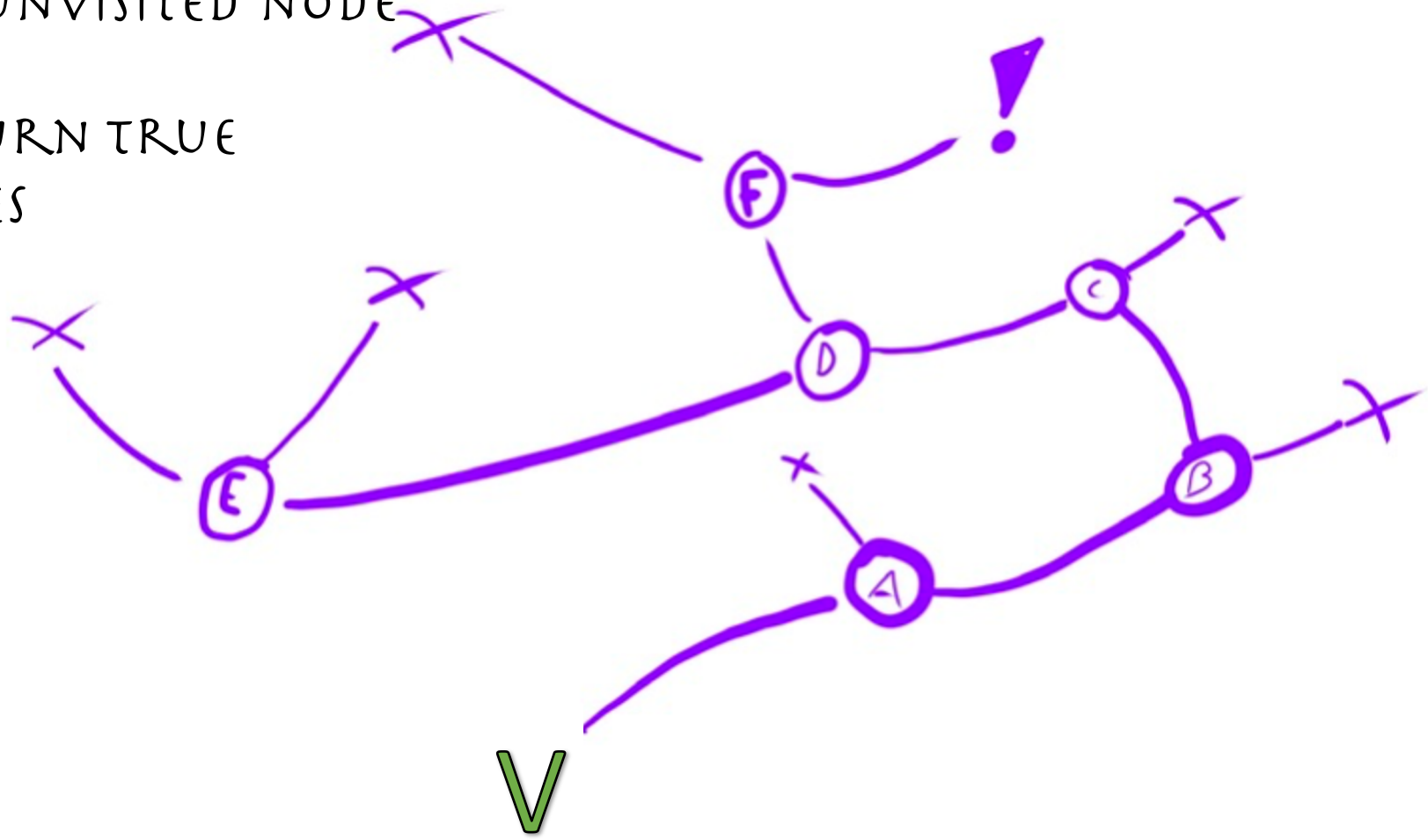
DEPTH FIRST SEARCH

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



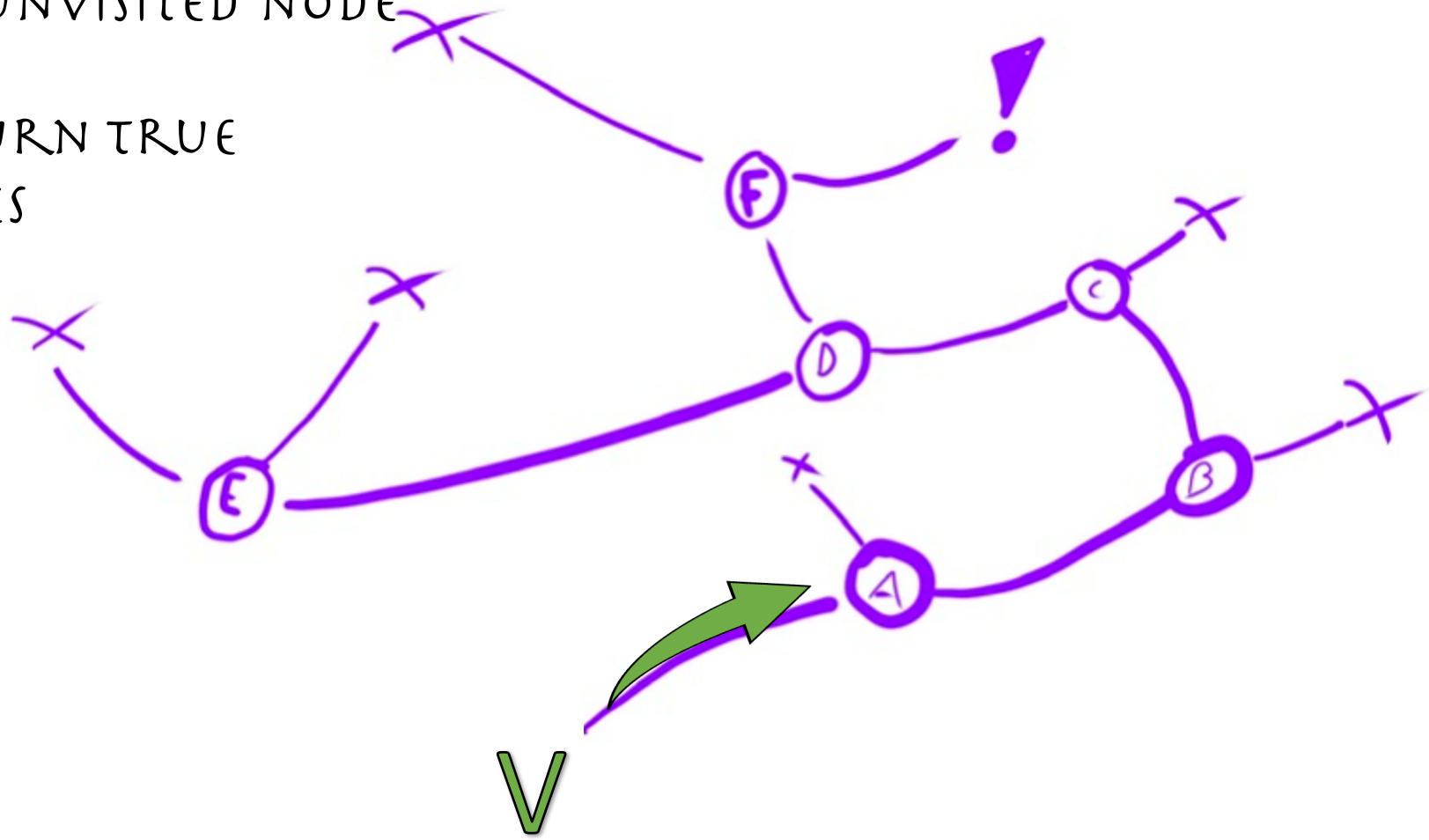
DEPTH FIRST SEARCH

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



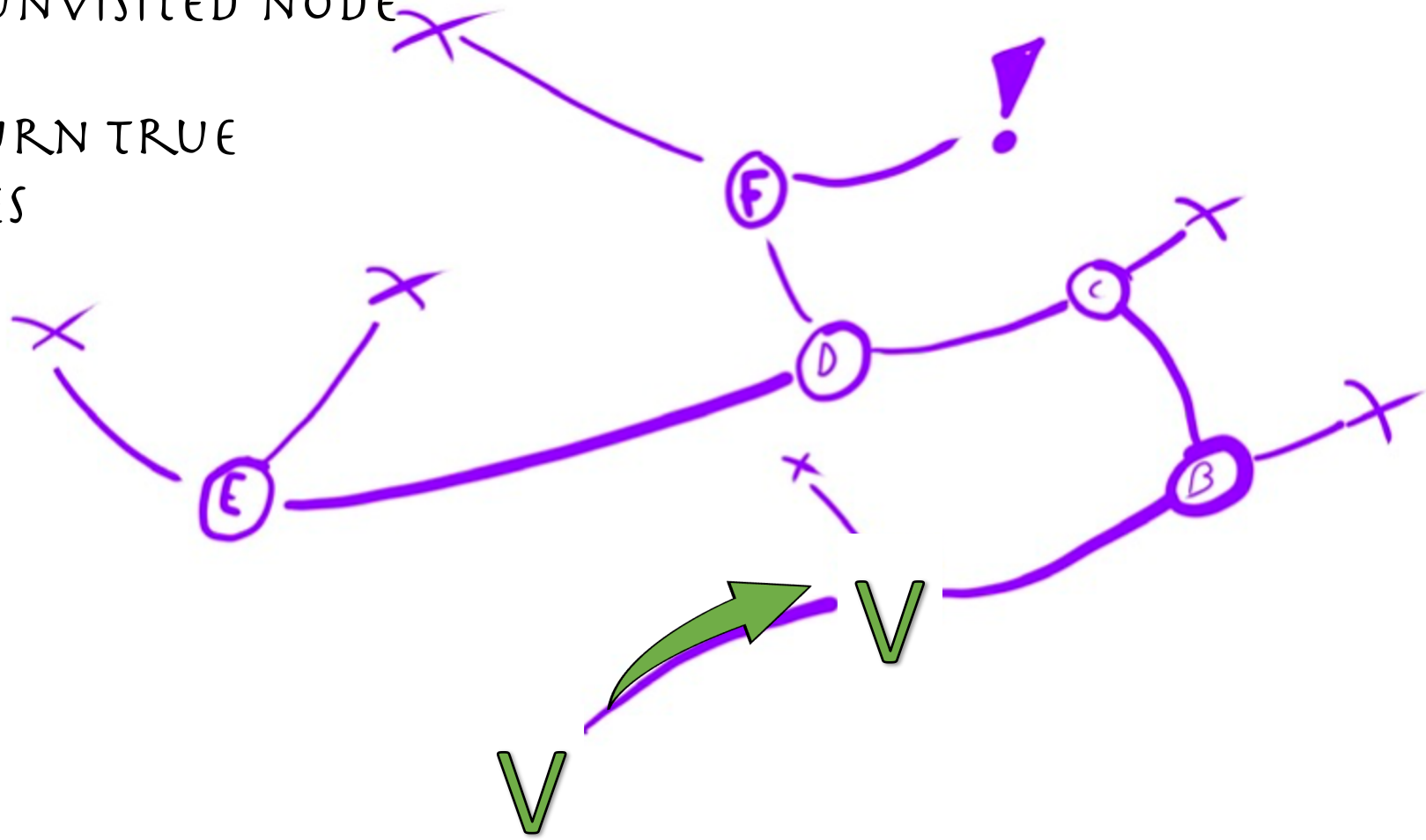
Depth First Search

- If you've found the end point:
  - #success!
  - return true
- Mark you current node as visited
- For every neighboring unvisited node
  - #perform dfs
  - if dfs(neighbor): return true
- #you've run out of nodes
- return false



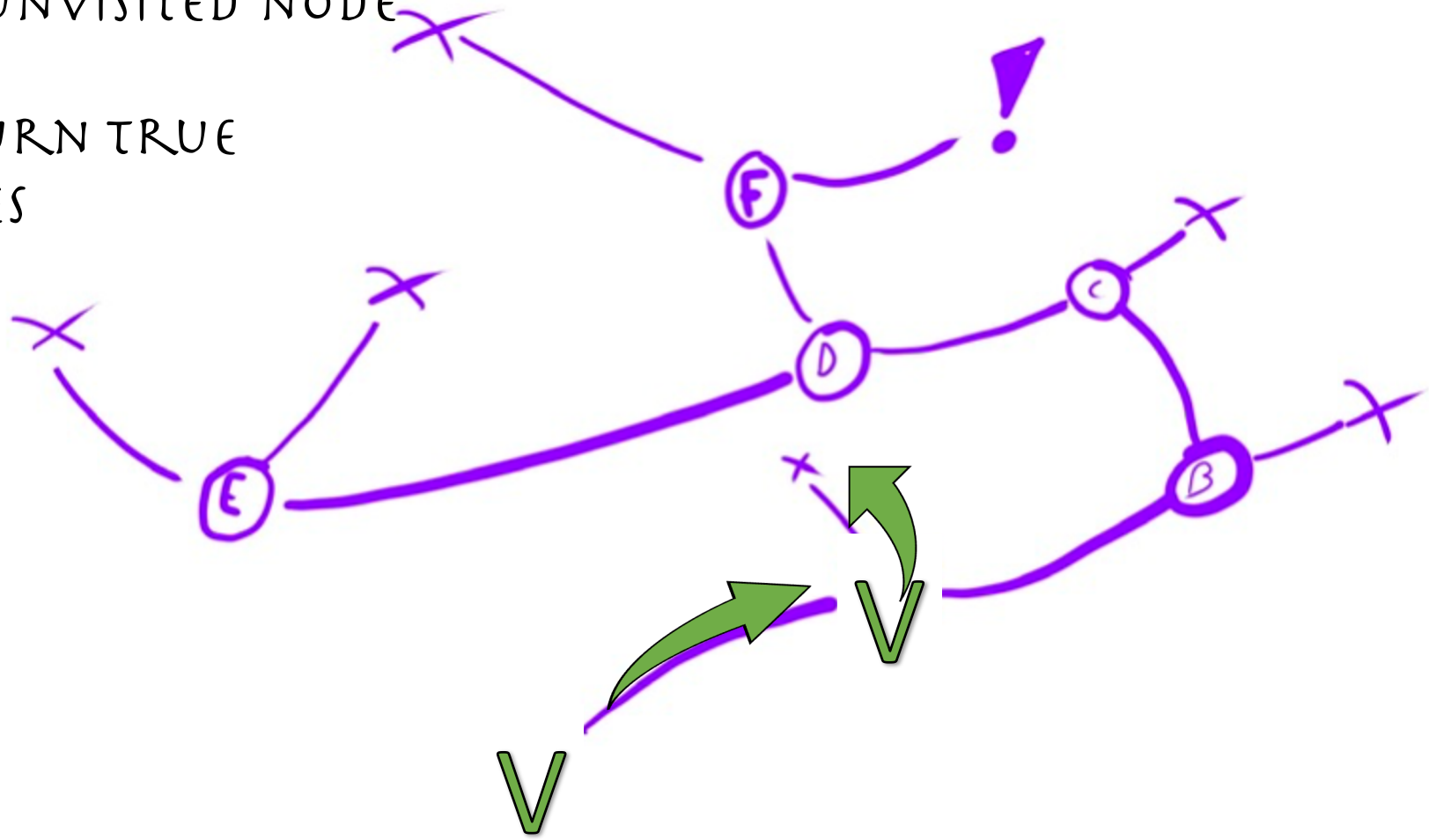
DEPTH FIRST SEARCH

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



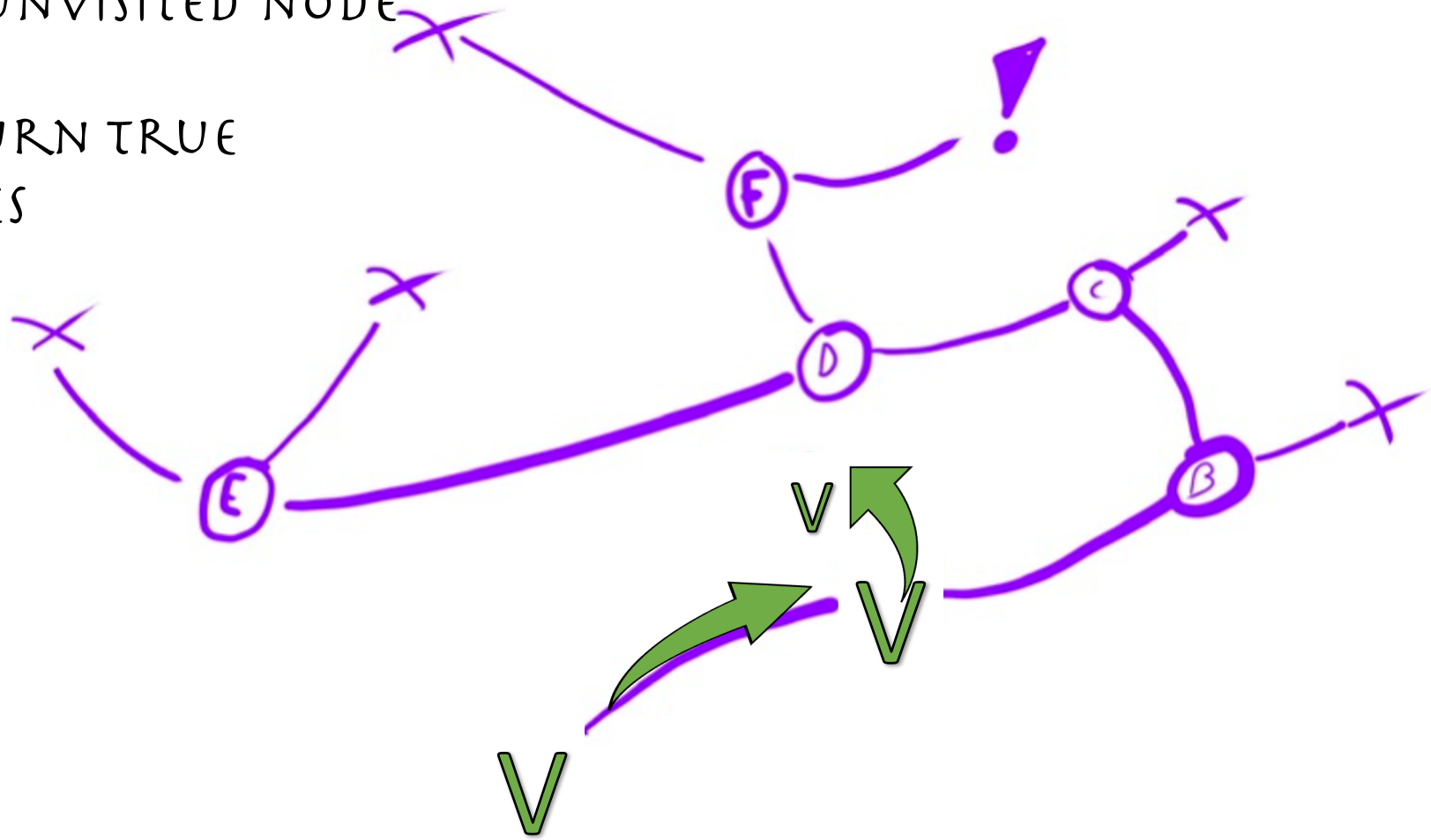
DEPTH FIRST SEARCH

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



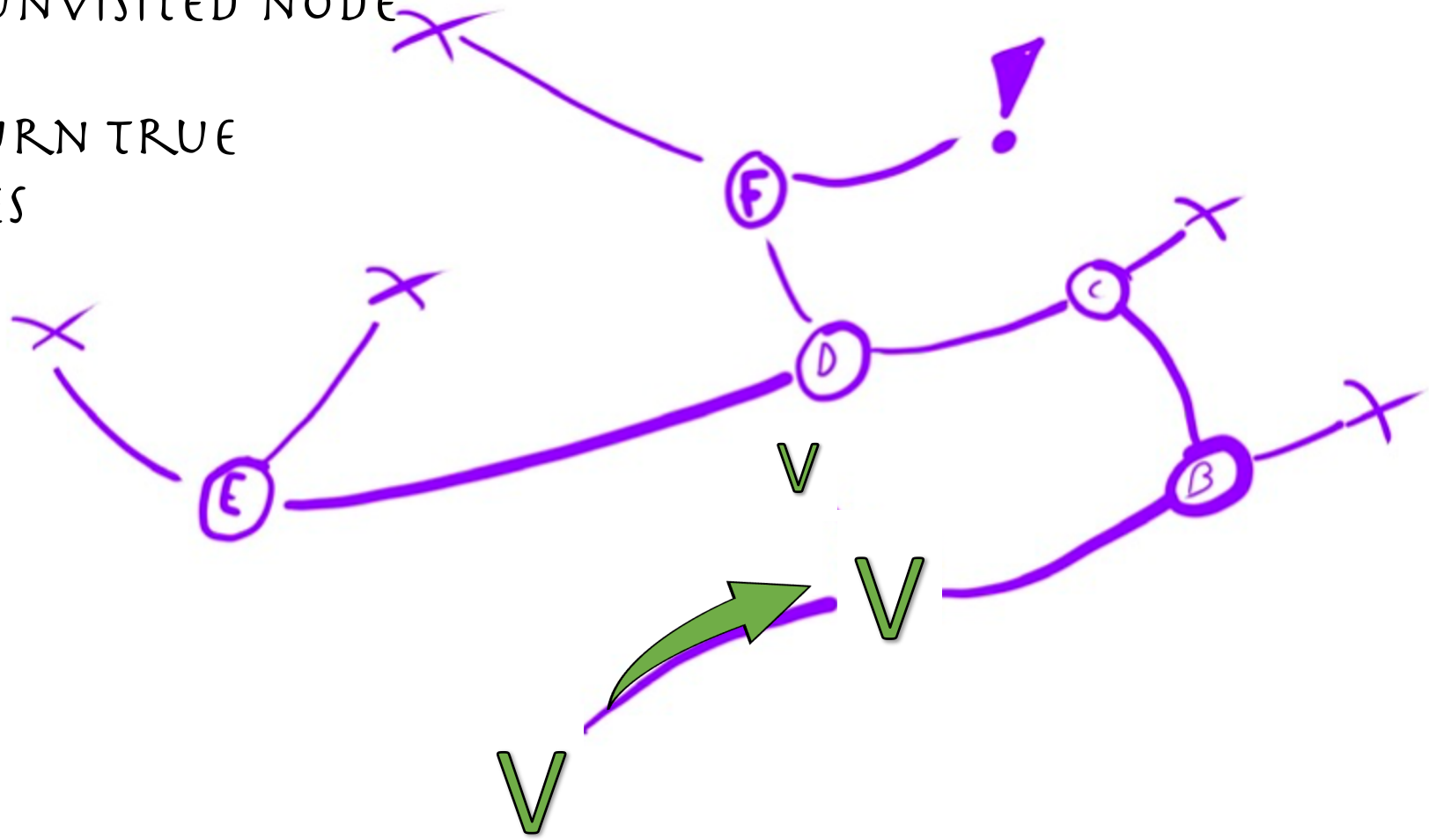
DEPTH FIRST SEARCH

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



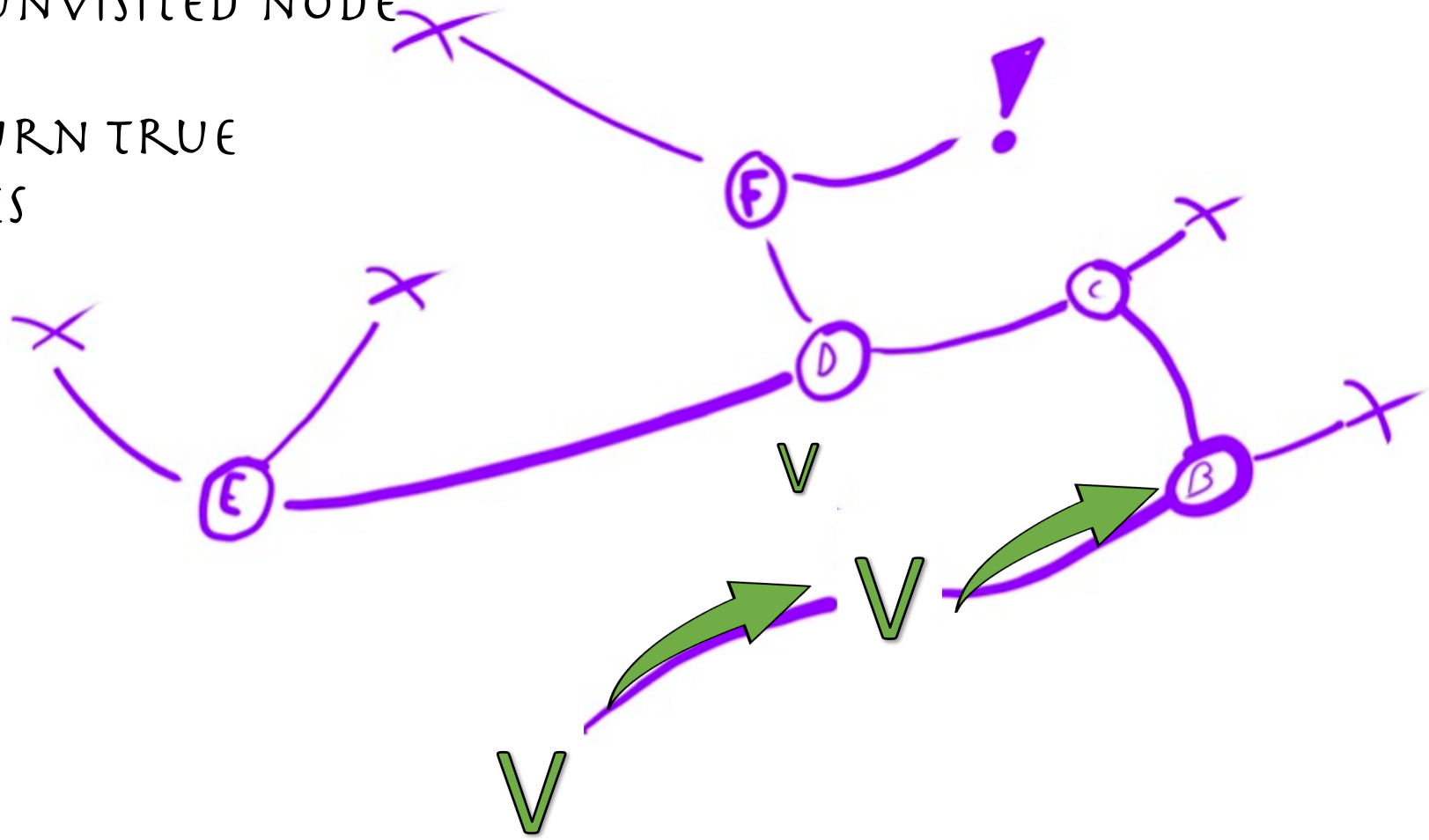
DEPTH FIRST SEARCH

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

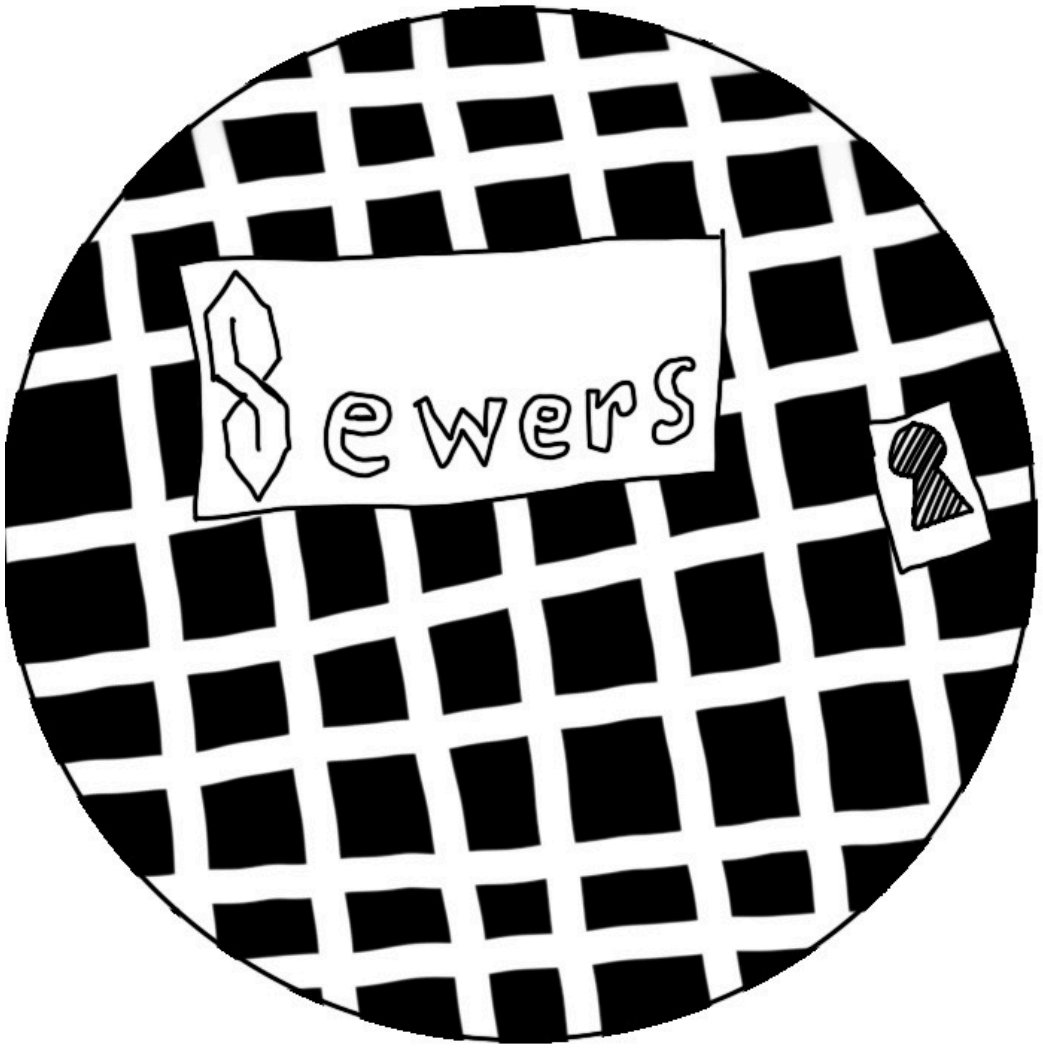


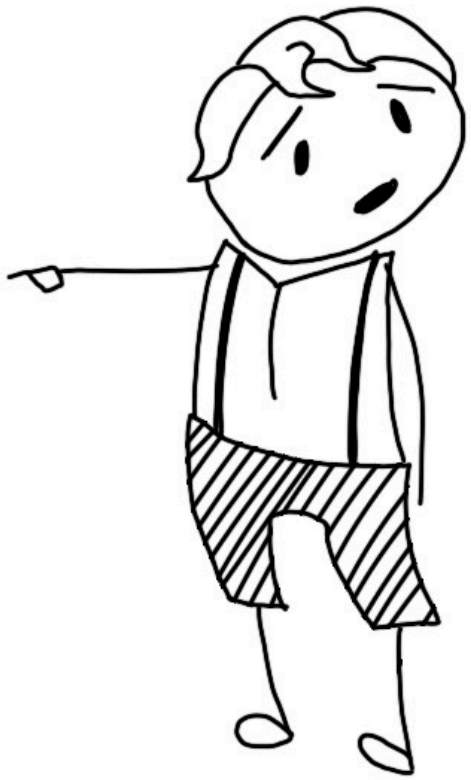
ANSWERING THE CALL TO ADVENTURE

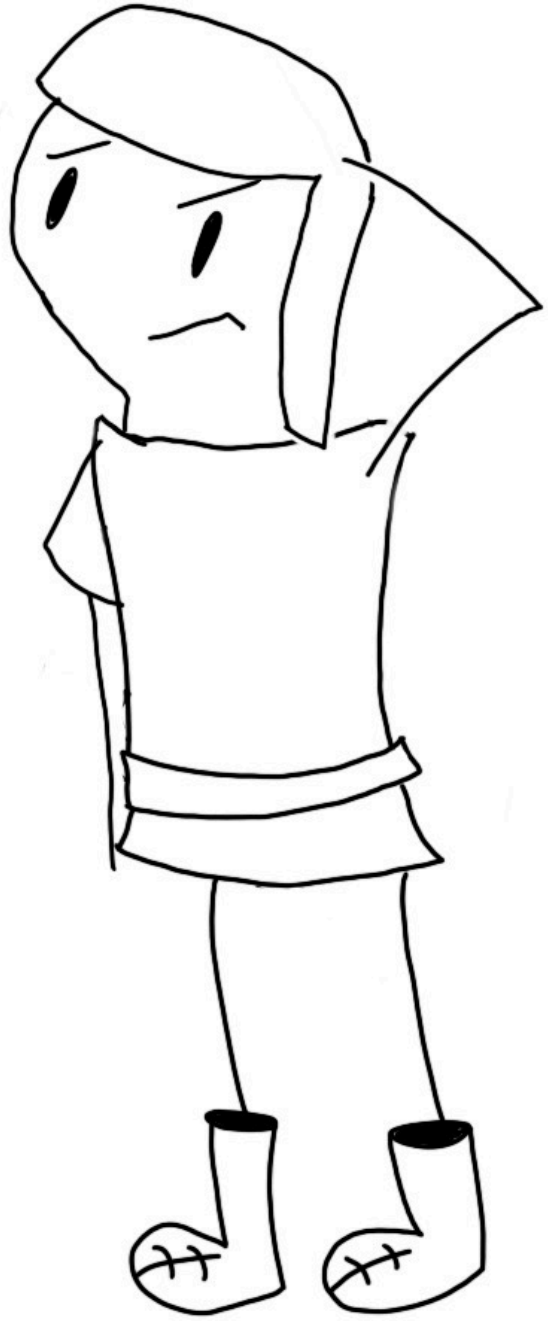
ANSWERING THE CALL TO ADVENTURE

ANSWERING THE CALL TO ADVENTURE

ANSWERING THE CALL TO ADVENTURE







ANSWERING THE CALL TO ADVENTURE

ALICE
BOB

ANSWERING THE CALL TO ADVENTURE

ALICE
BOB

~~ALICE~~
BOB
CARMEN
DEV

ANSWERING THE CALL TO ADVENTURE

ALICE
BOB

~~ALICE~~
BOB
CARMEN
DEV

~~ALICE~~
~~BOB~~
CARMEN
DEV
ELLEN

ANSWERING THE CALL TO ADVENTURE

ALICE
BOB

~~ALICE~~
BOB
CARMEN
DEV

~~ALICE~~
~~BOB~~
CARMEN
DEV
ELLEN

~~ALICE~~
~~BOB~~
CARMEN
DEV
ELLEN
BOB

ANSWERING THE CALL TO ADVENTURE

ALICE
BOB

~~ALICE~~
BOB
CARMEN
DEV

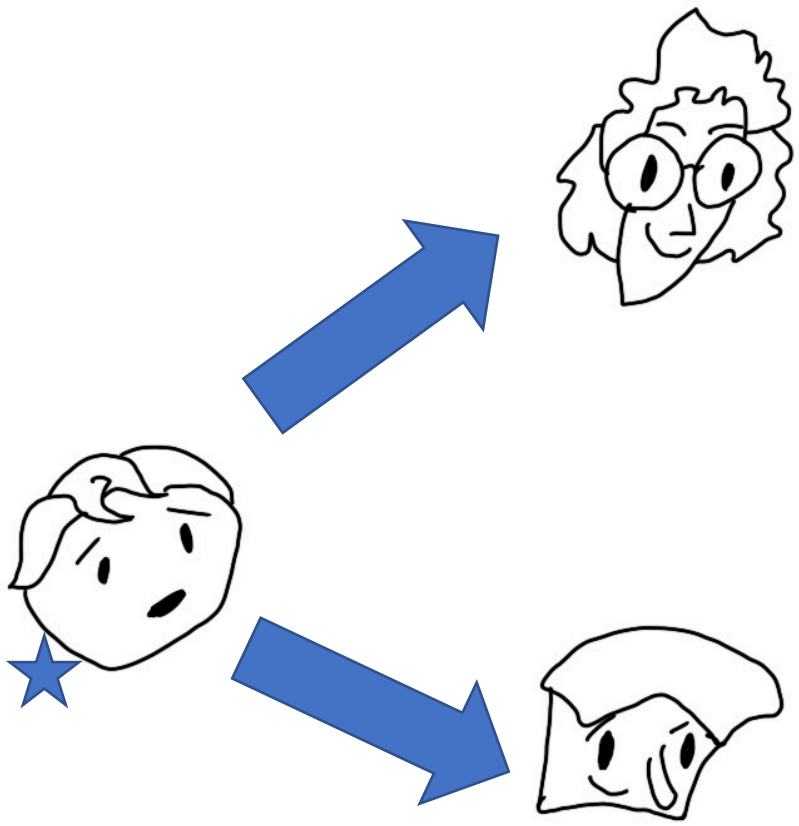
~~ALICE~~
~~BOB~~
CARMEN
DEV
ELLEN

~~ALICE~~
~~BOB~~
CARMEN
DEV
ELLEN
BOB

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

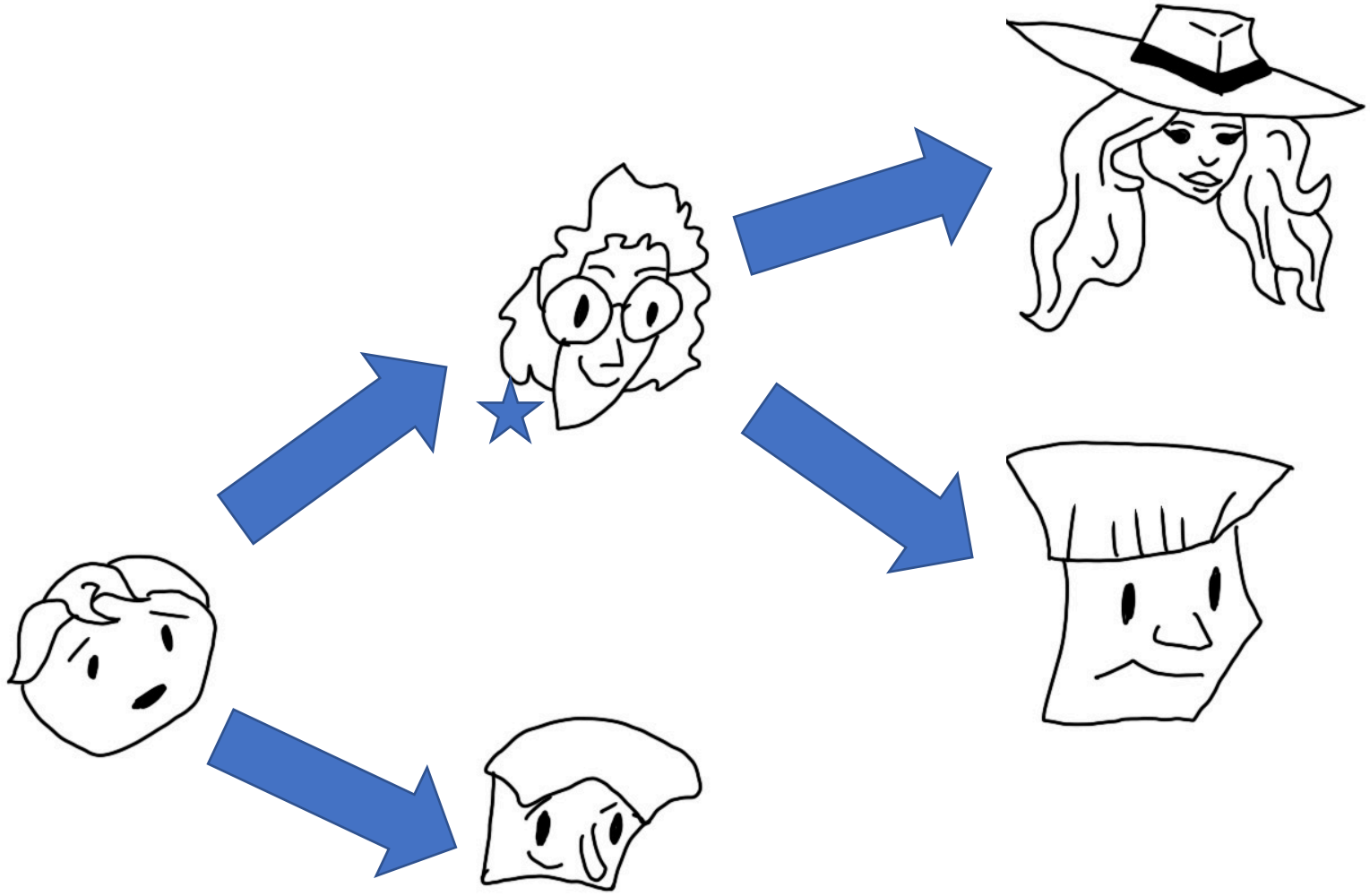


PEOPLE TO ASK

ALICE
BOB

X THINKS Y HAS THE KY

ANSWERING THE CALL TO ADVENTURE

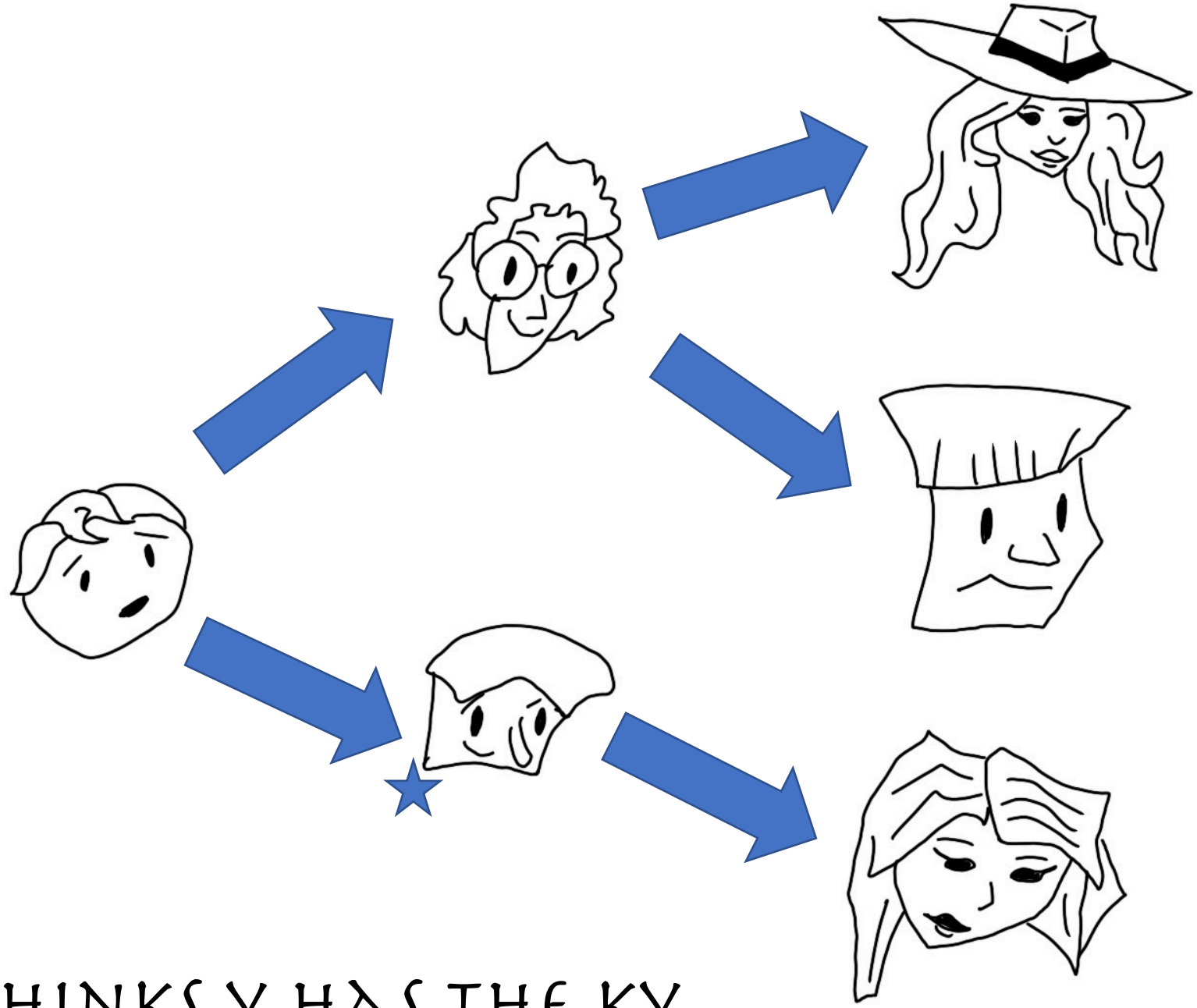


PEOPLE TO ASK

~~ALICE~~
BOB
CARMEN
DAN

X THINKS Y HAS THE KY

ANSWERING THE CALL TO ADVENTURE

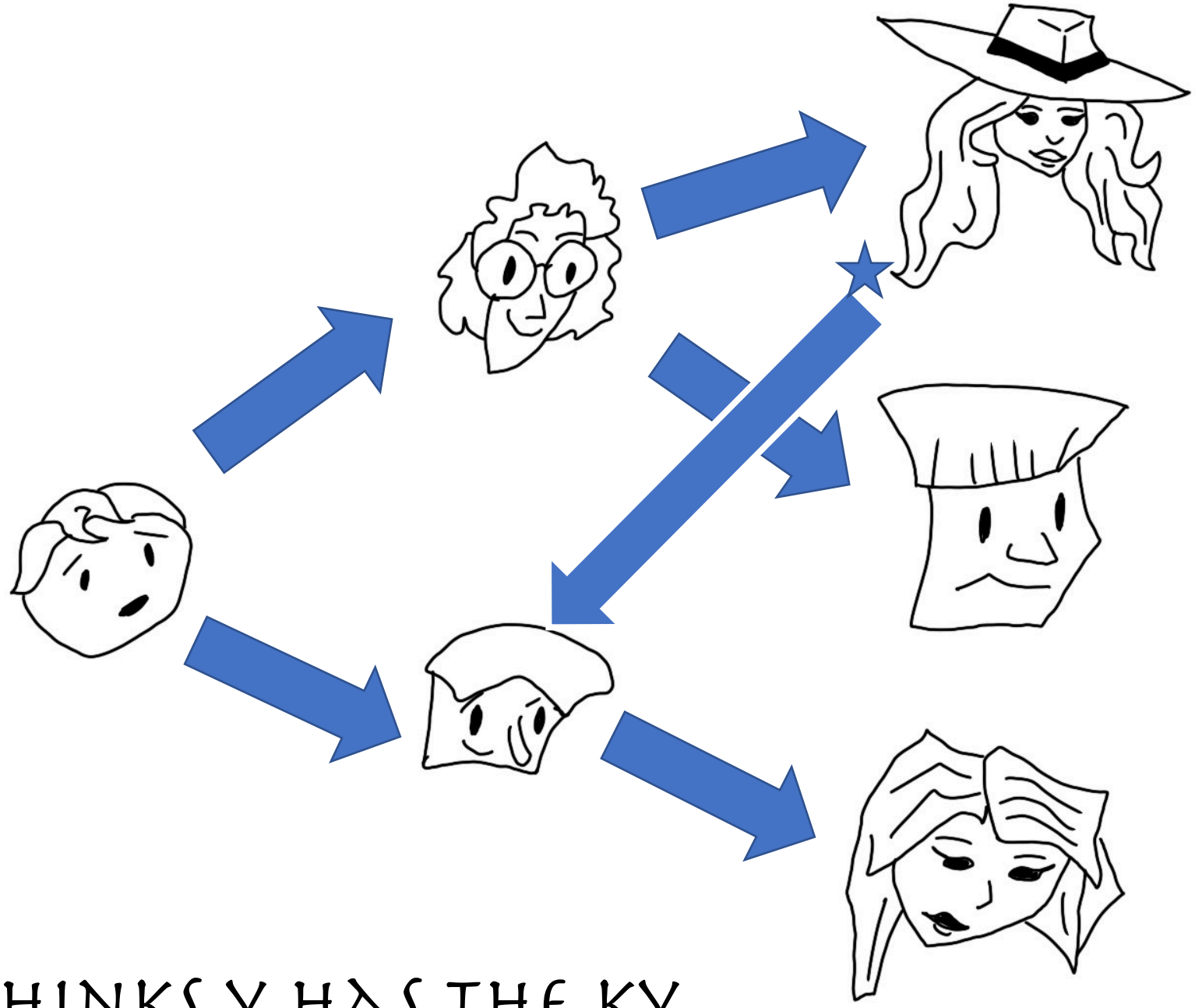


PEOPLE TO ASK

~~ALICE~~
~~BOB~~
CARMEN
DAN
ELLEN

X THINKS Y HAS THE KY

ANSWERING THE CALL TO ADVENTURE

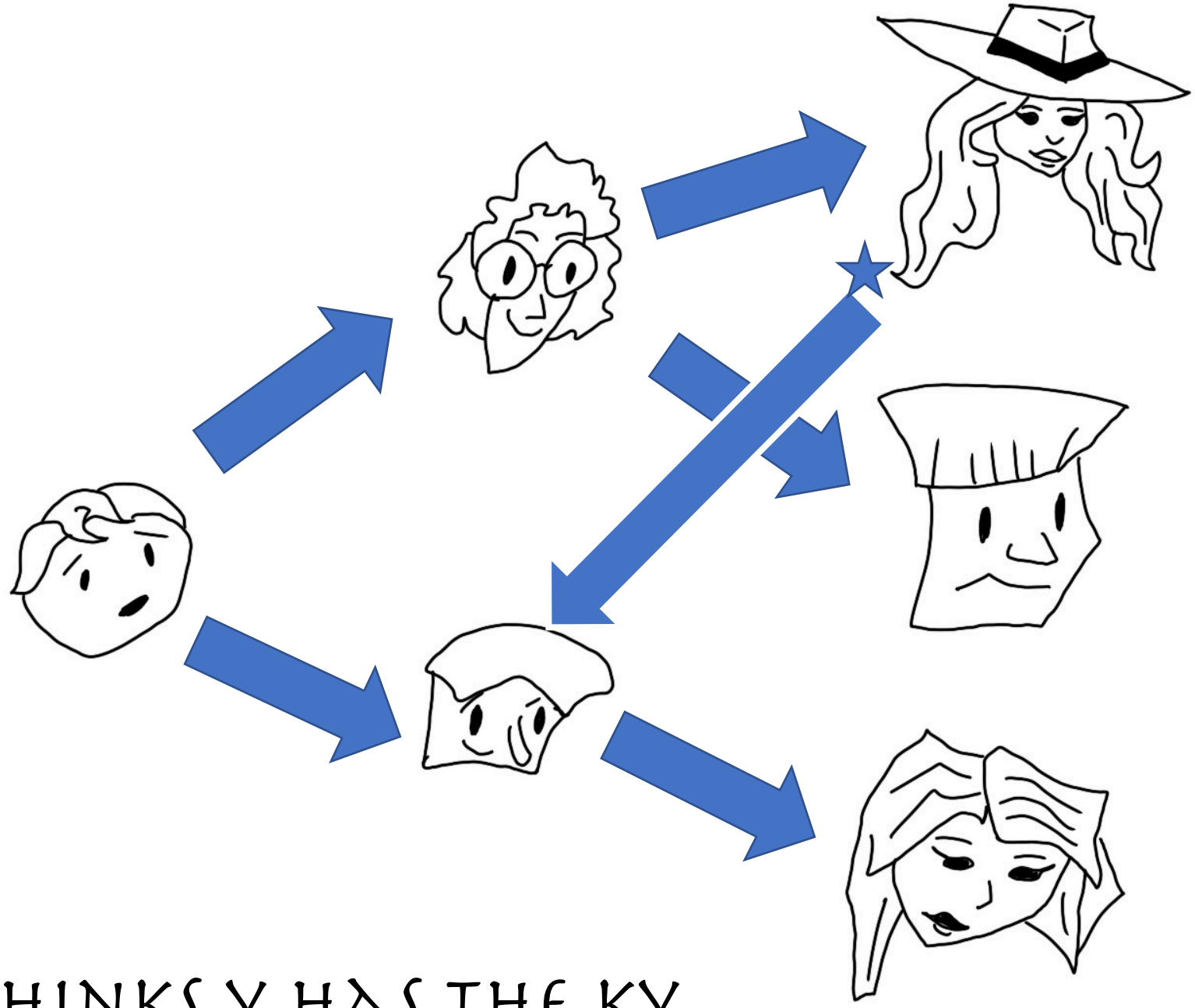


PEOPLE TO ASK

~~ALICE~~
~~BOB~~
~~CARMEN~~
DAN
ELLEN
BOB

X THINKS Y HAS THE KY

ANSWERING THE CALL TO ADVENTURE



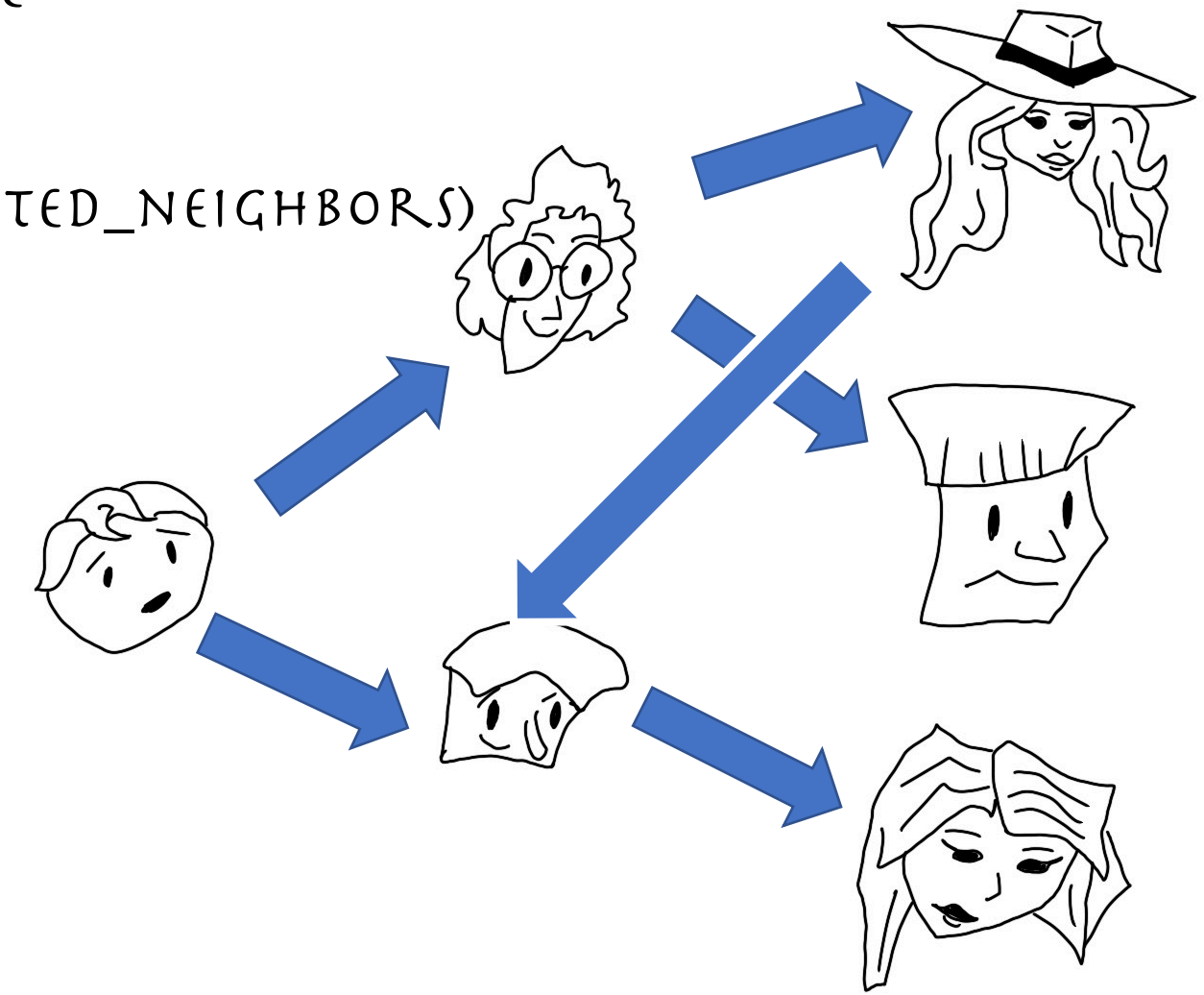
PEOPLE TO ASK

~~ALICE~~
~~BOB~~
~~CARMEN~~
DAN
ELLEN
BOB

X THINKS Y HAS THE KY

BREADTH FIRST SEARCH

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



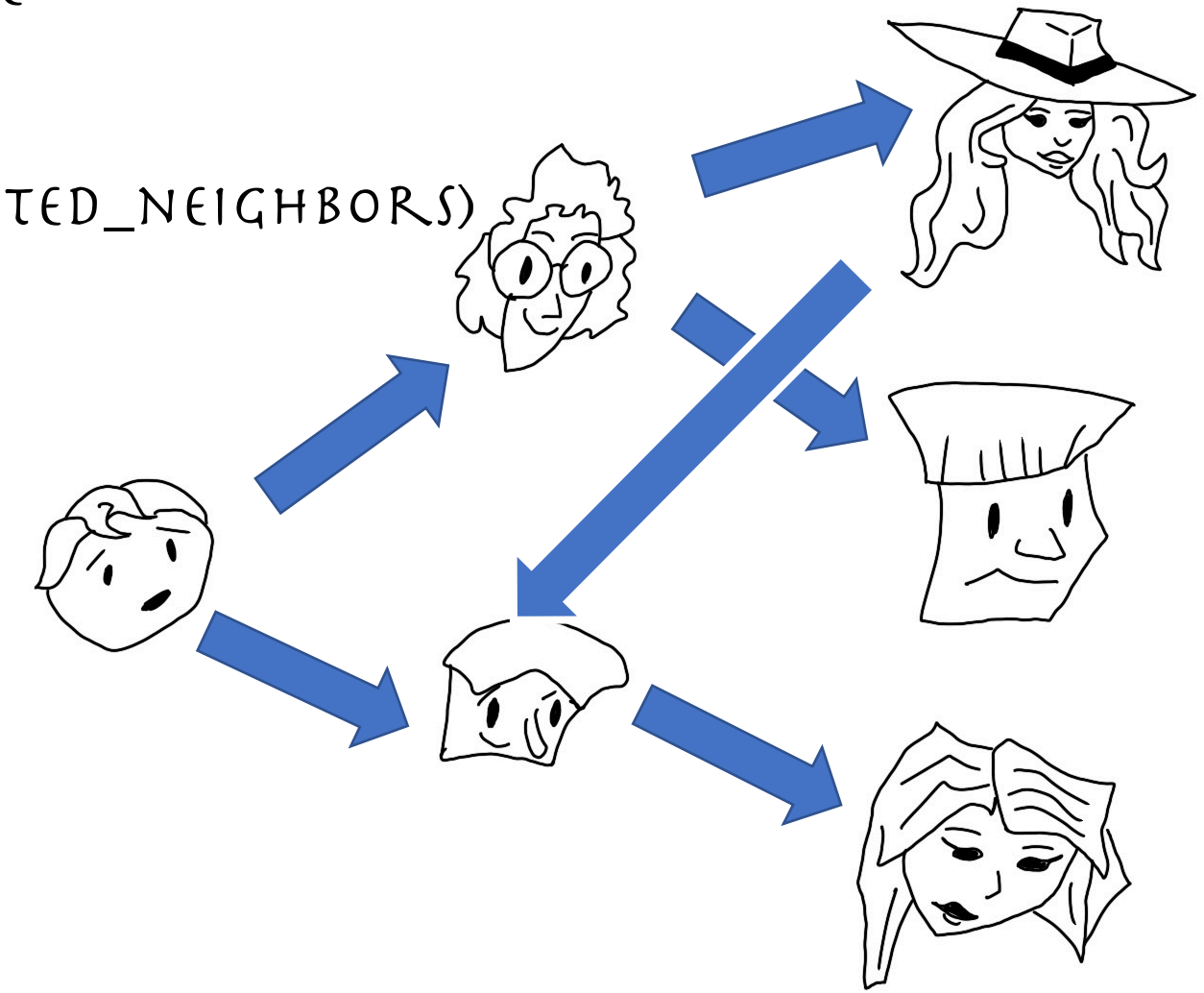
PEOPLE TO ASK

~~ALICE~~
~~BOB~~
~~CARMEN~~
DEV
ELLEN
BOB

BREADTH FIRST SEARCH

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

QUEUE:
KID

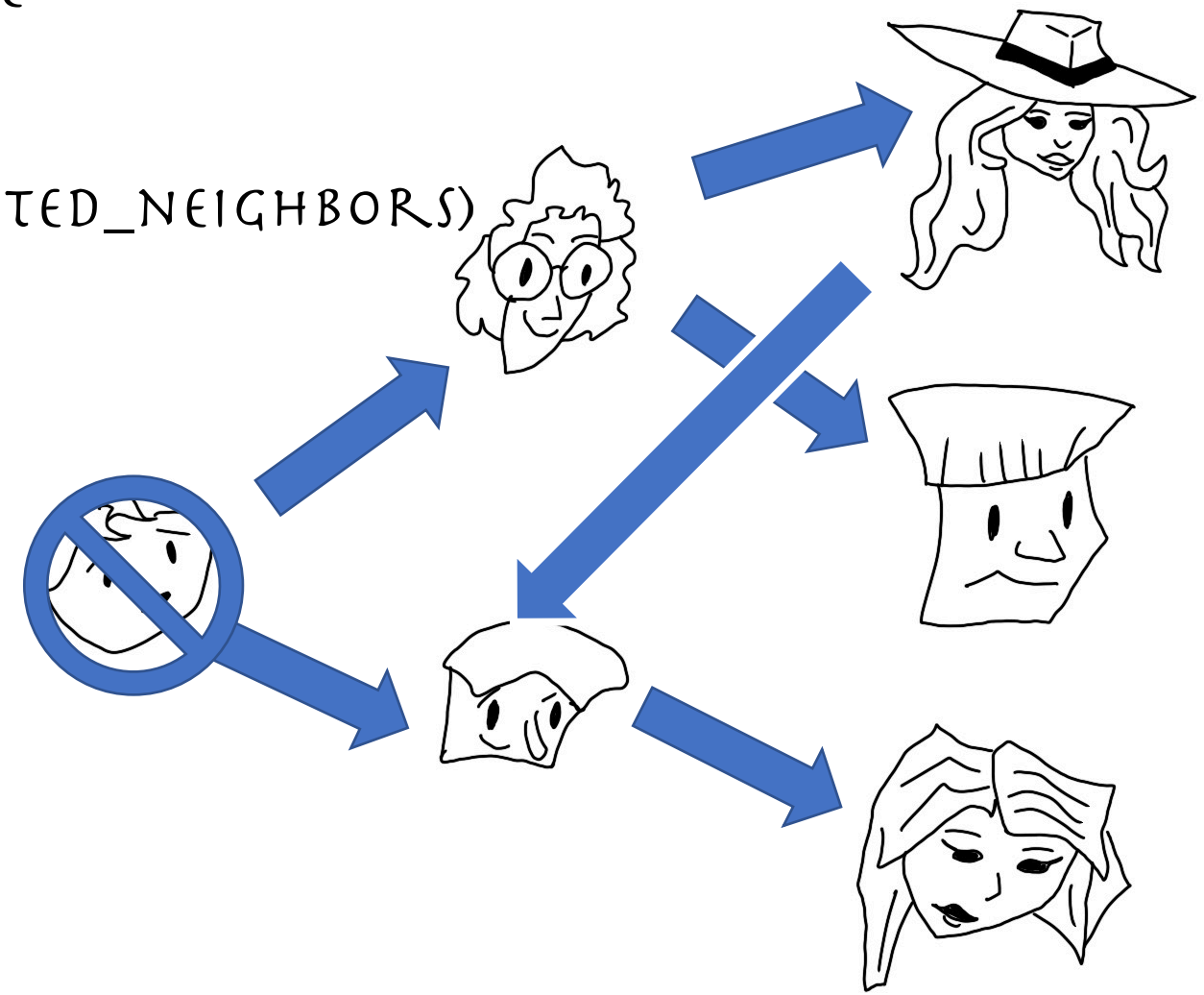


BREADTH FIRST SEARCH

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

CURRENT: KID

QUEUE:
<EMPTY>

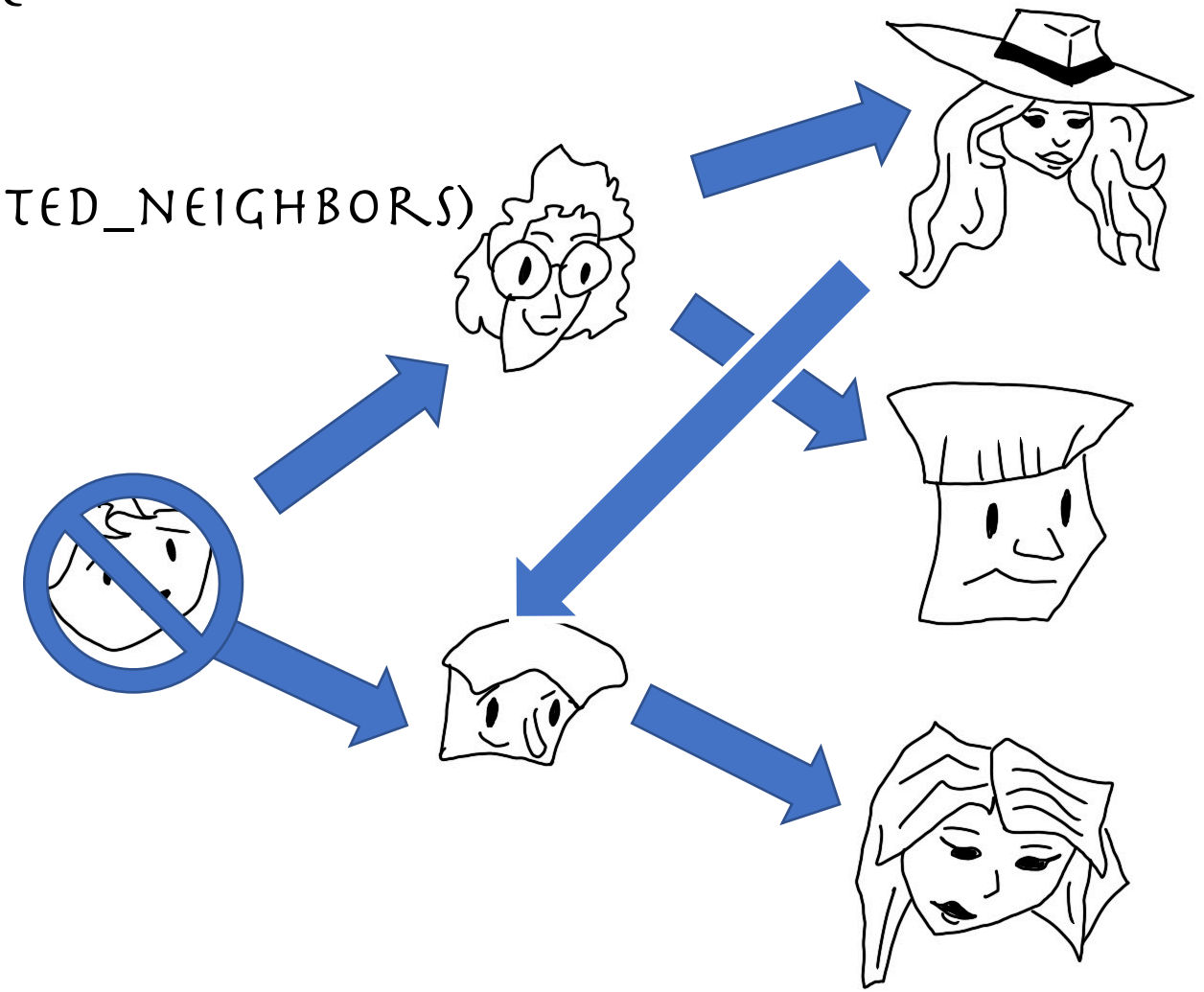


BREADTH FIRST SEARCH

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

CURRENT: KID

QUEUE:
ALICE
BOB

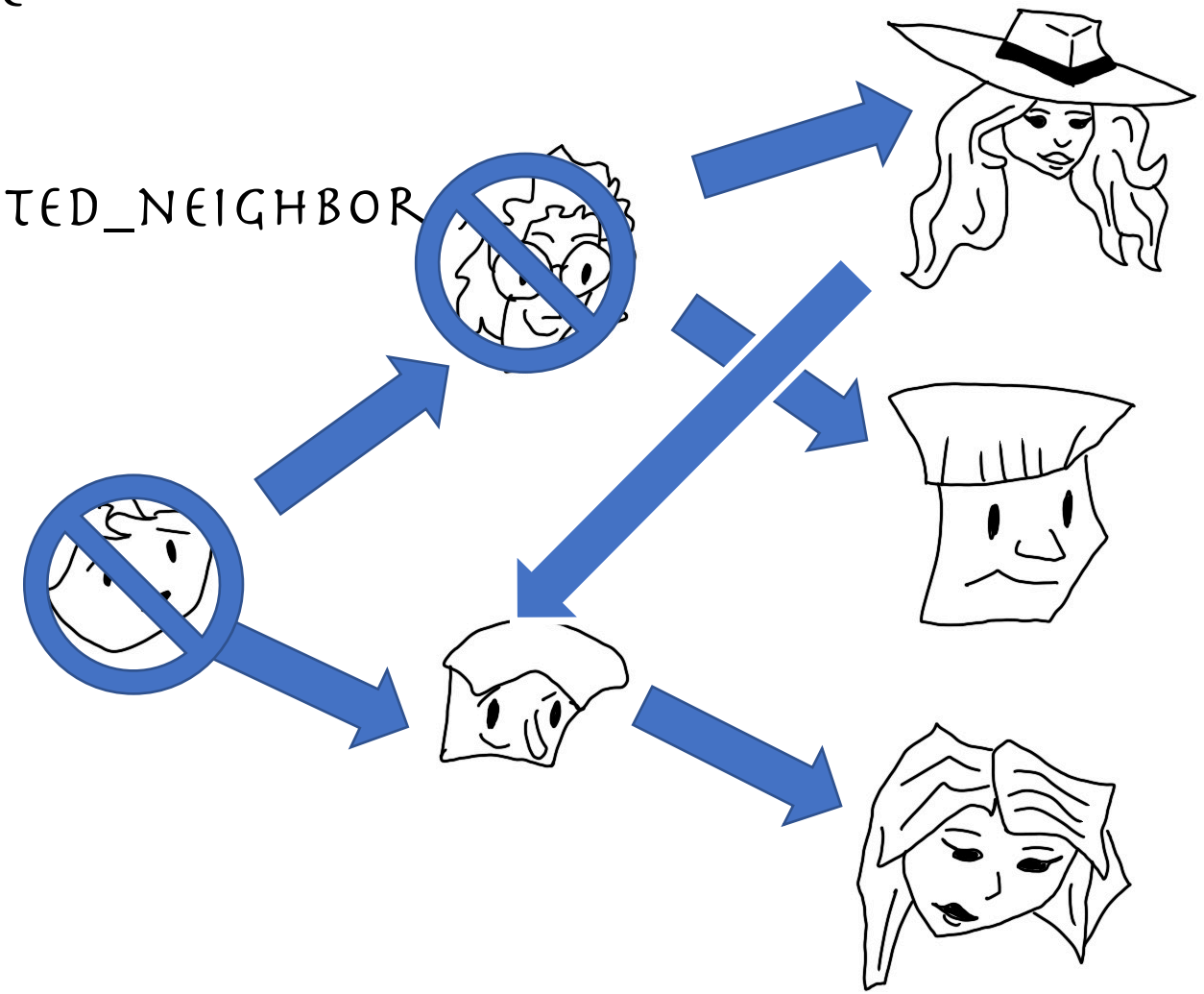


BREADTH FIRST SEARCH

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

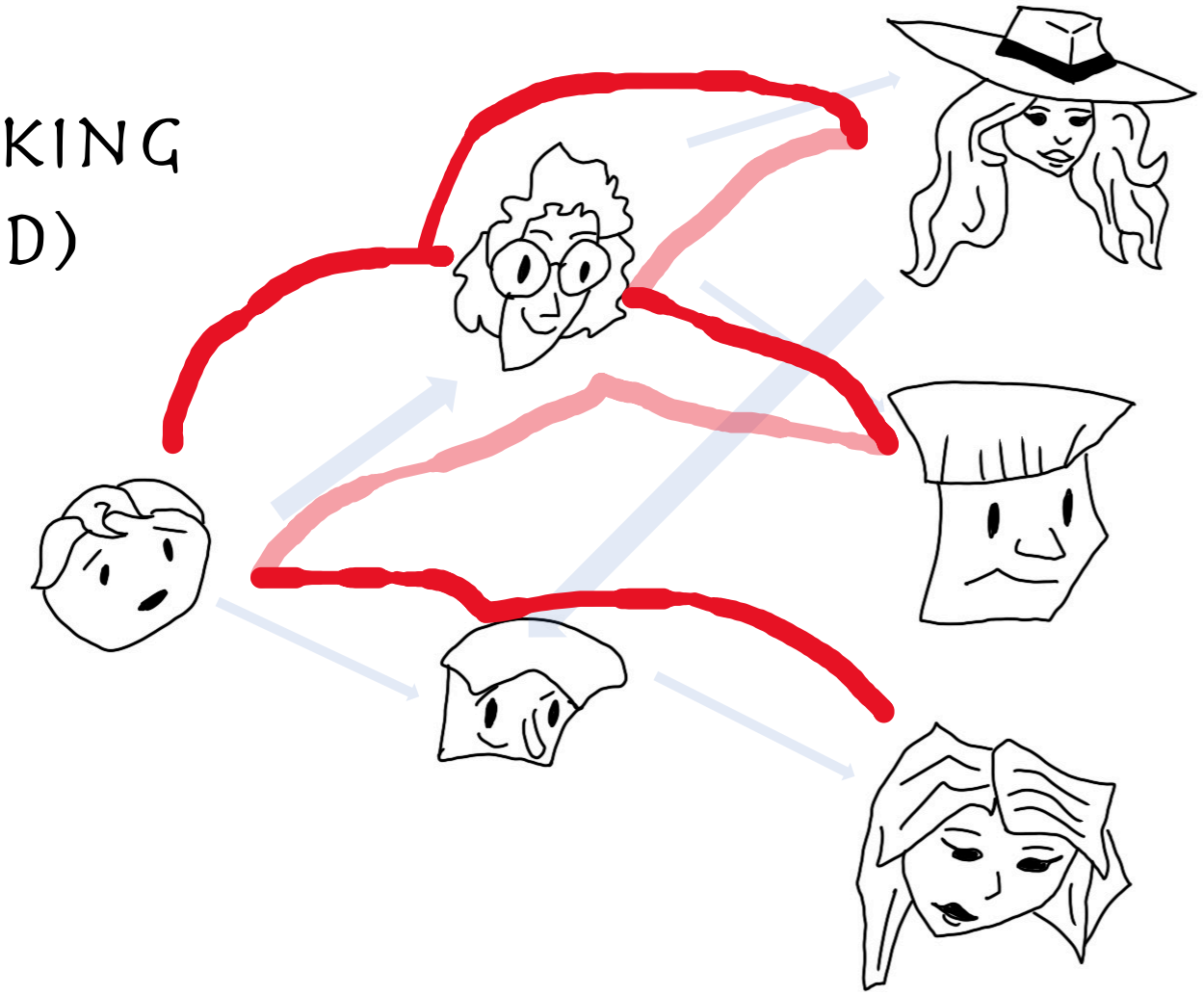
CURRENT: ALICE

QUEUE:
BOB



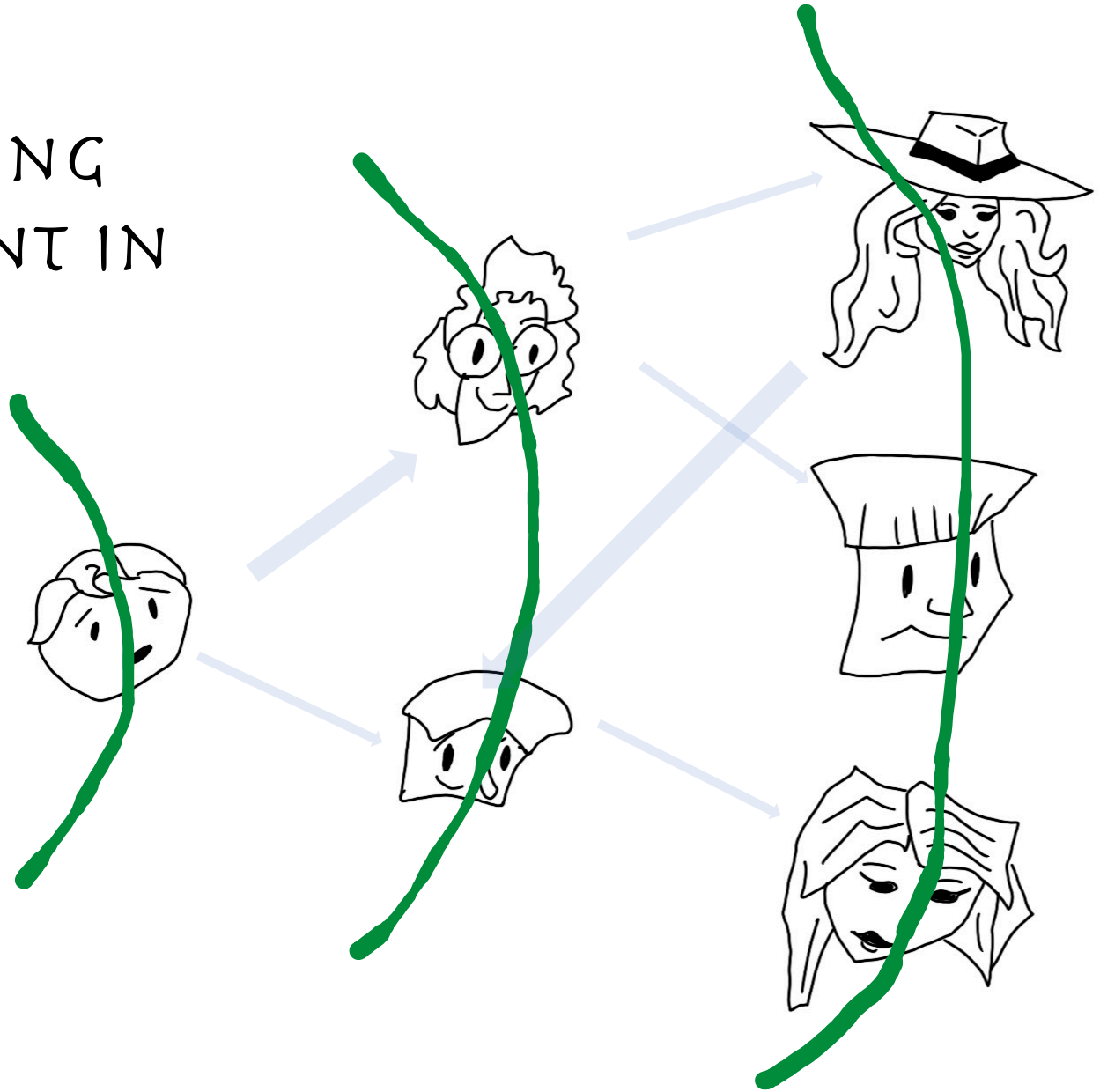
A QUICK NOTE ON BFS VS DFS

DFS: RED
(BACKTRACKING
IN LIGHT RED)



A QUICK NOTE ON BFS VS DFS

BFS: EXPANDING SEARCH FRONT IN GREEN

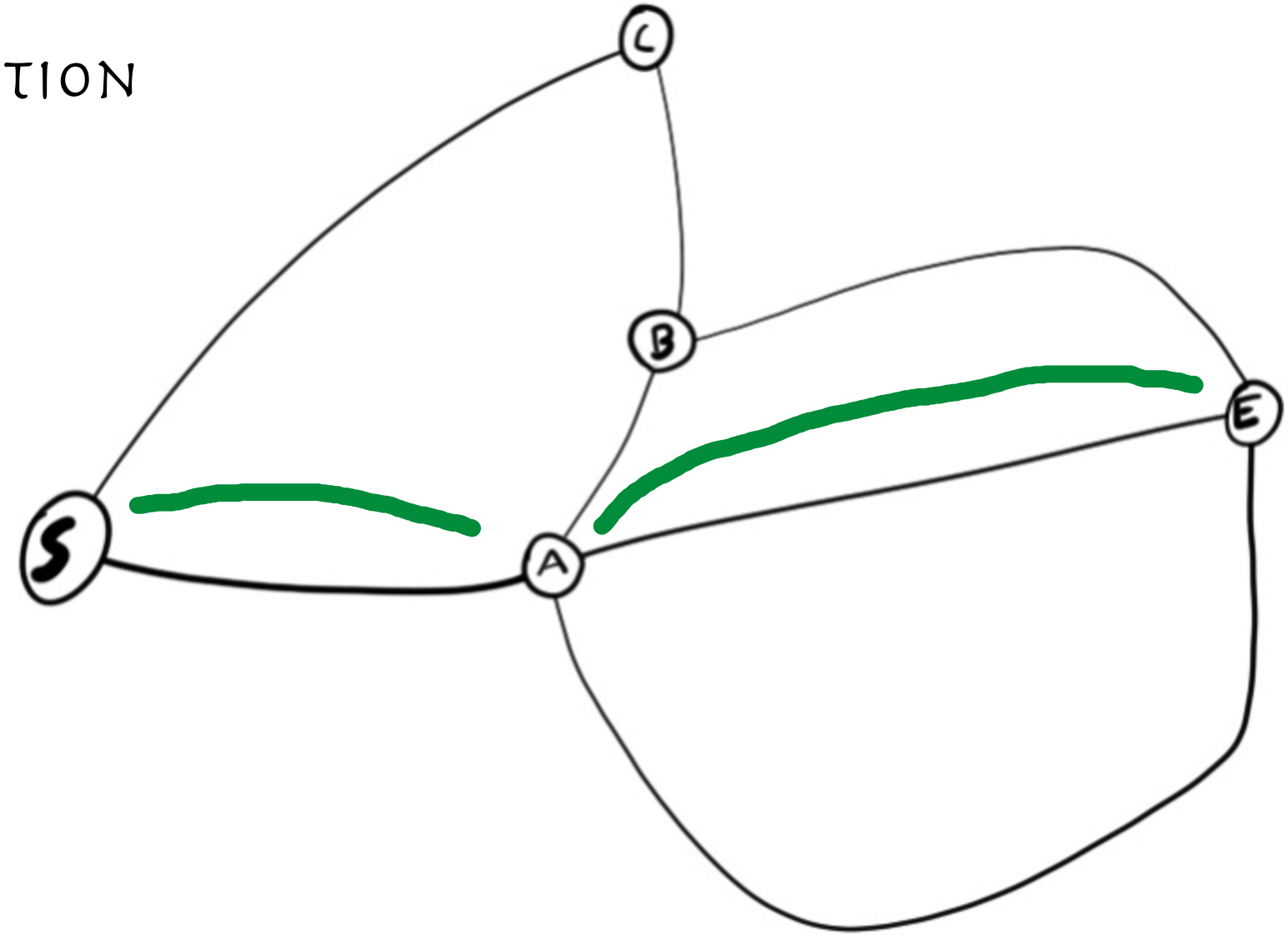


YOU FOUND THE KEY!

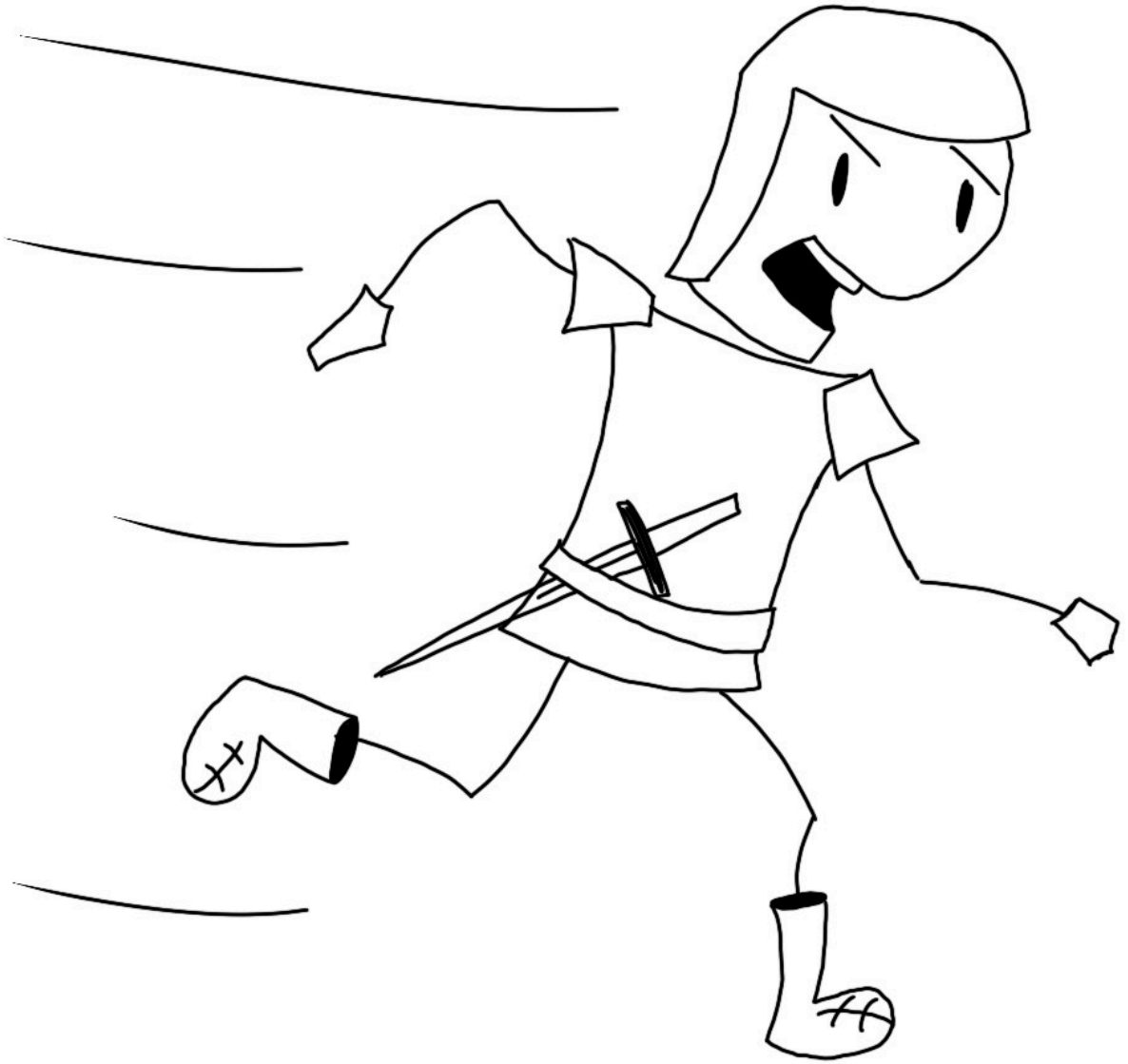
YOU FOUND THE KEY!

MAP -> GRAPH

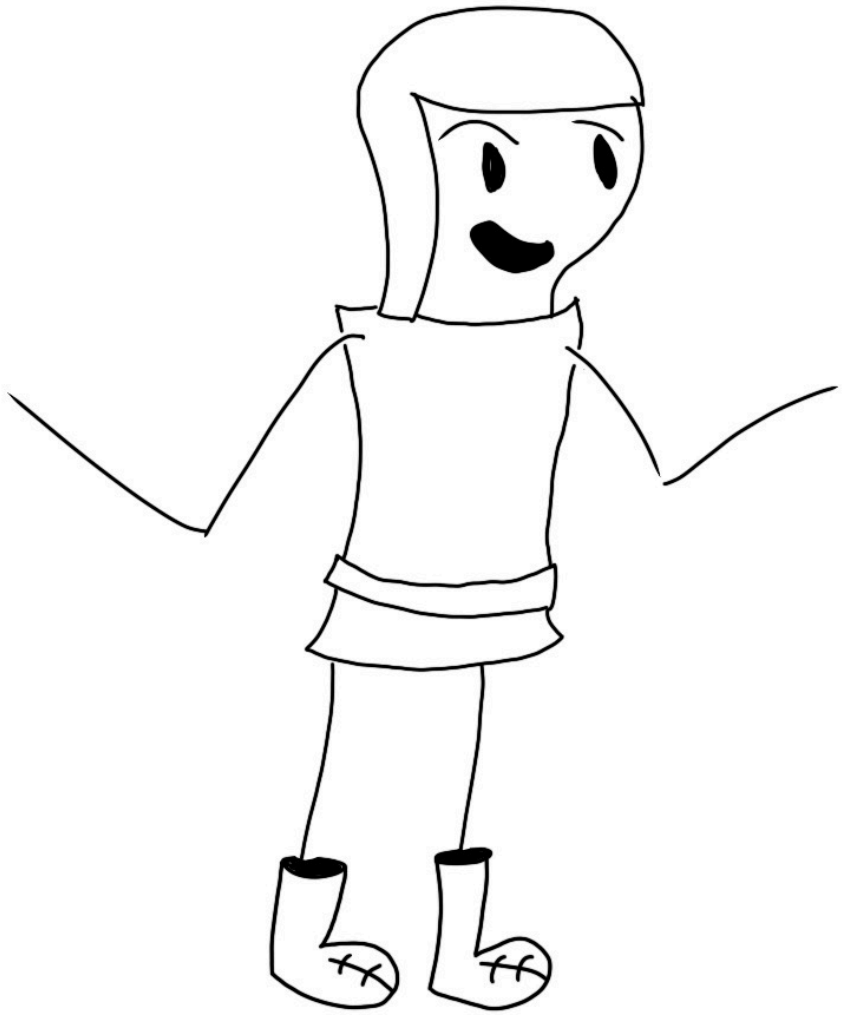
BFS SOLUTION



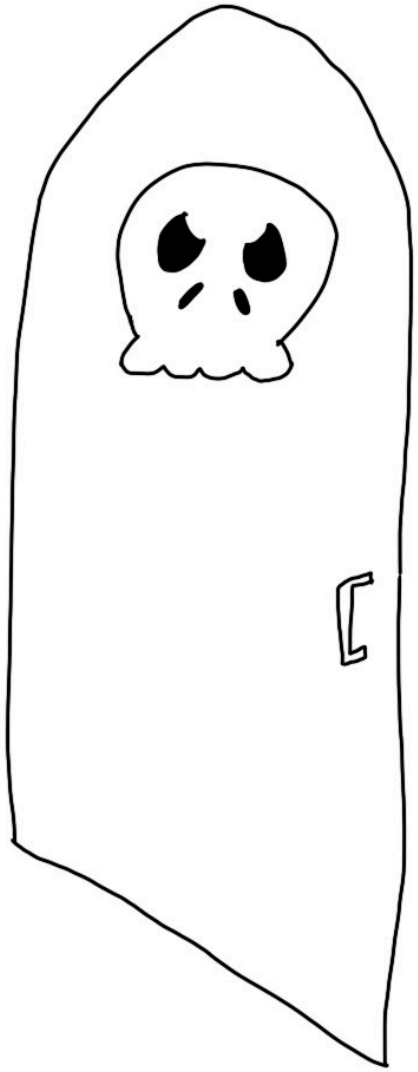
YOU FOUND THE KEY!



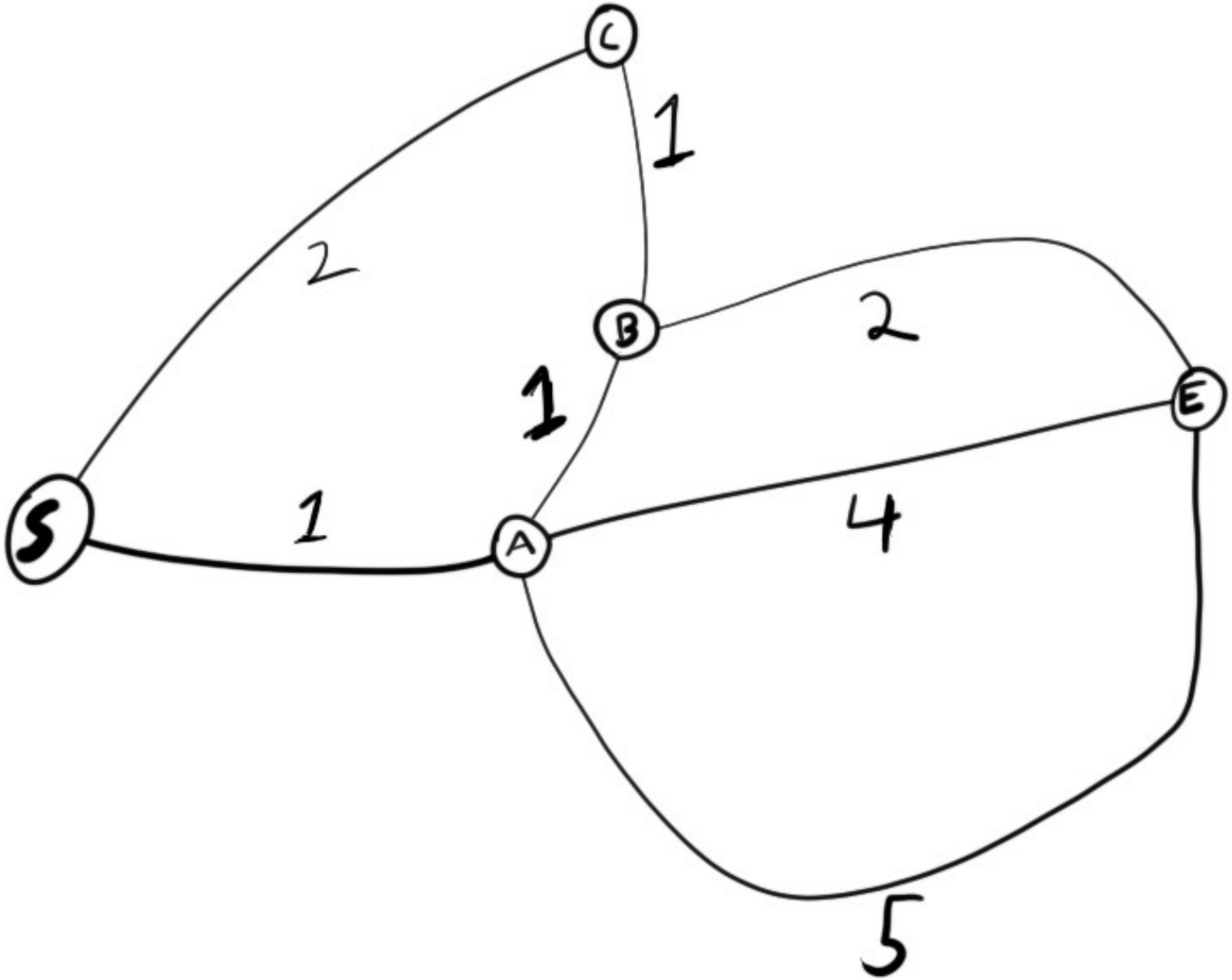
YOU FOUND THE KEY!



A WIZARD NAMED DIJKSTRA



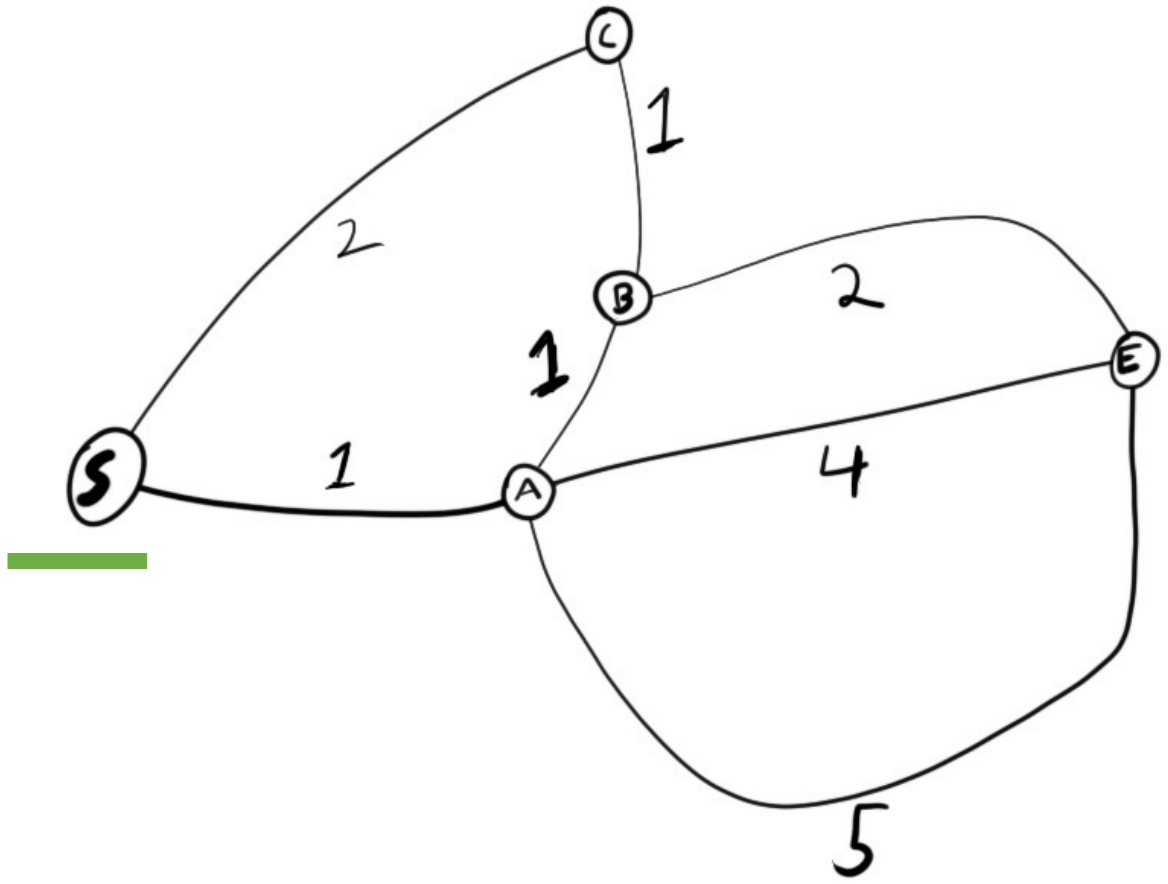
YOU FOUND THE KEY!



DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM PREVIOUS NODES

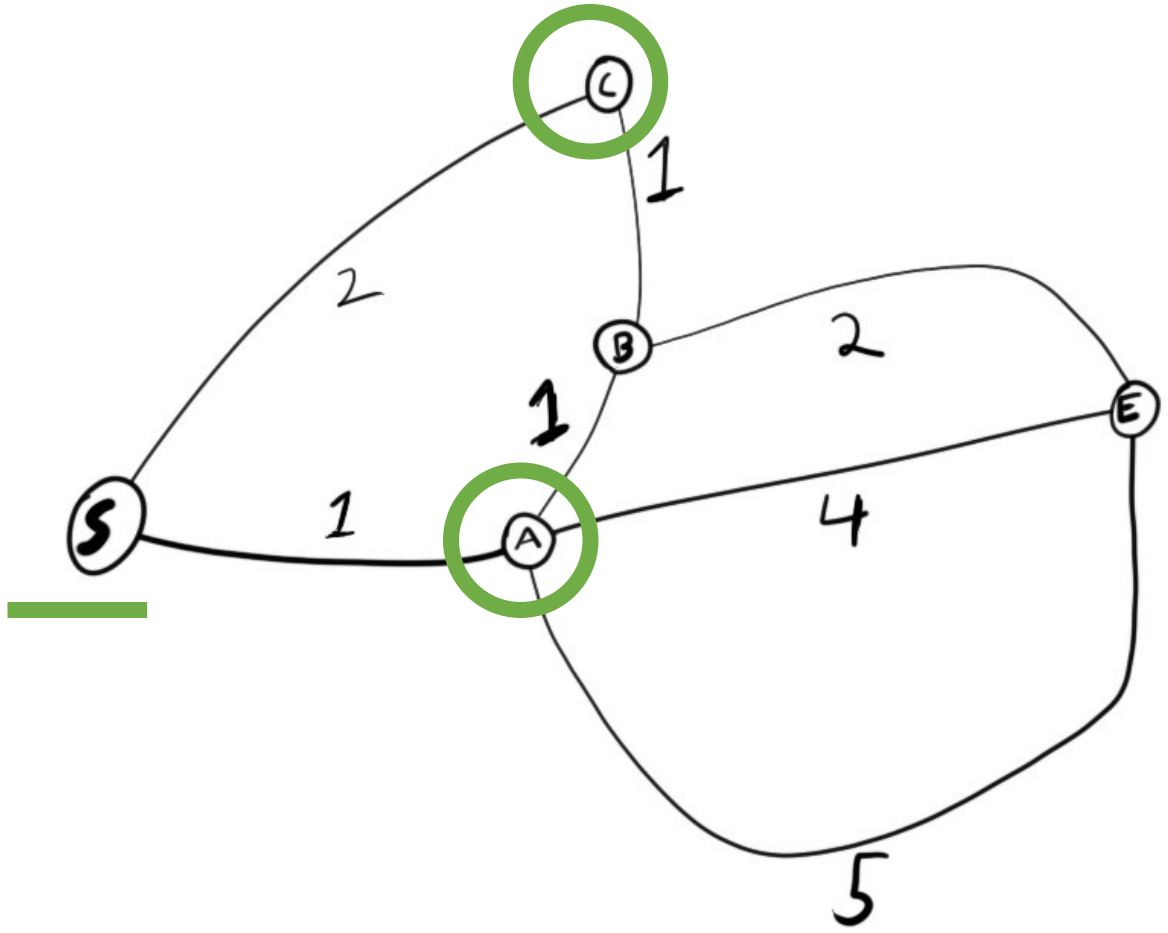
NODES	DISTANCE FROM START	PREVIOUS
START	0	Ø
A	∞	Ø
B	∞	Ø
C	∞	Ø
END	∞	Ø



DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM PREVIOUS NODES

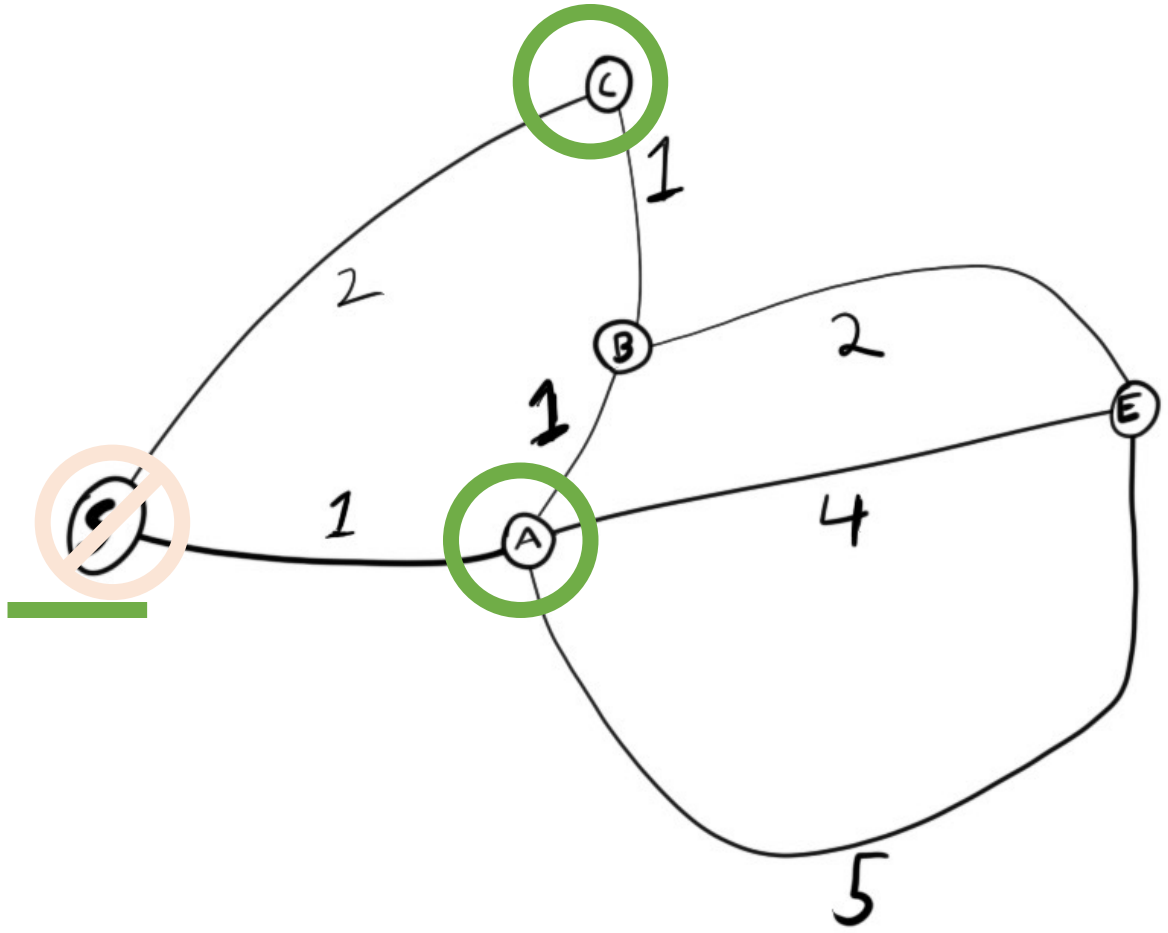
NODES	DISTANCE FROM START	PREVIOUS
START	0	Ø
A	∞	Ø
B	∞	Ø
C	∞	Ø
END	∞	Ø



DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM PREVIOUS NODES

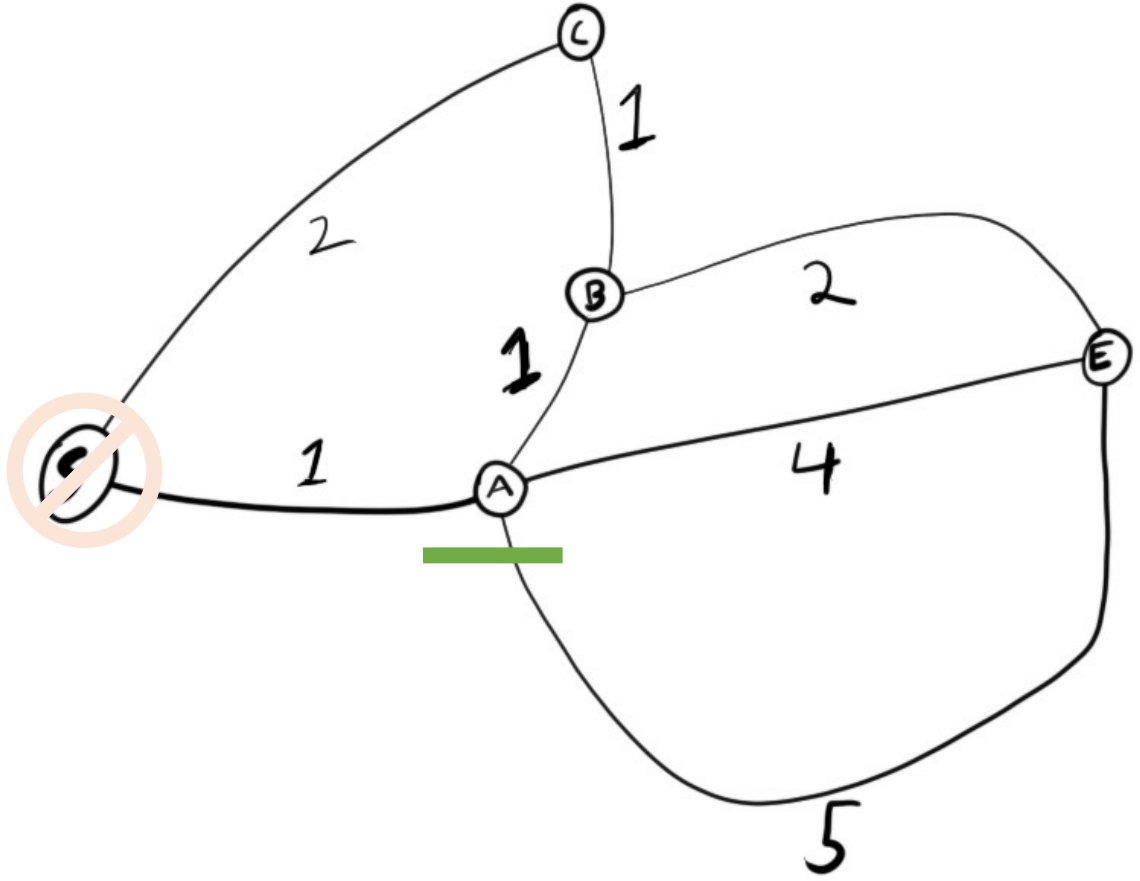
NODES	DISTANCE FROM START	PREVIOUS
START	0	Ø
A	1	START
B	∞	Ø
C	2	START
END	∞	Ø



DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM PREVIOUS NODES

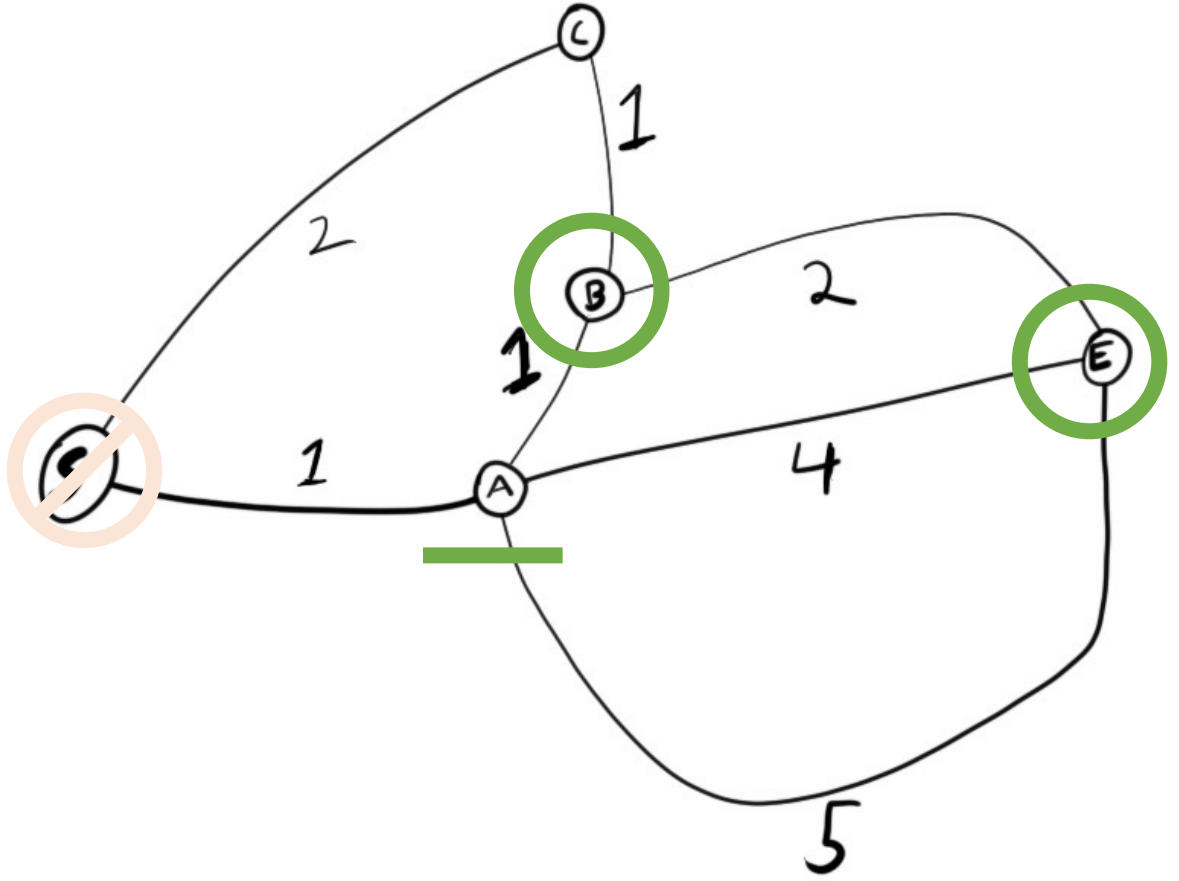
NODES	DISTANCE FROM START	PREVIOUS
START	0	Ø
A	1	START
B	∞	Ø
C	2	START
END	∞	Ø



DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM PREVIOUS NODES

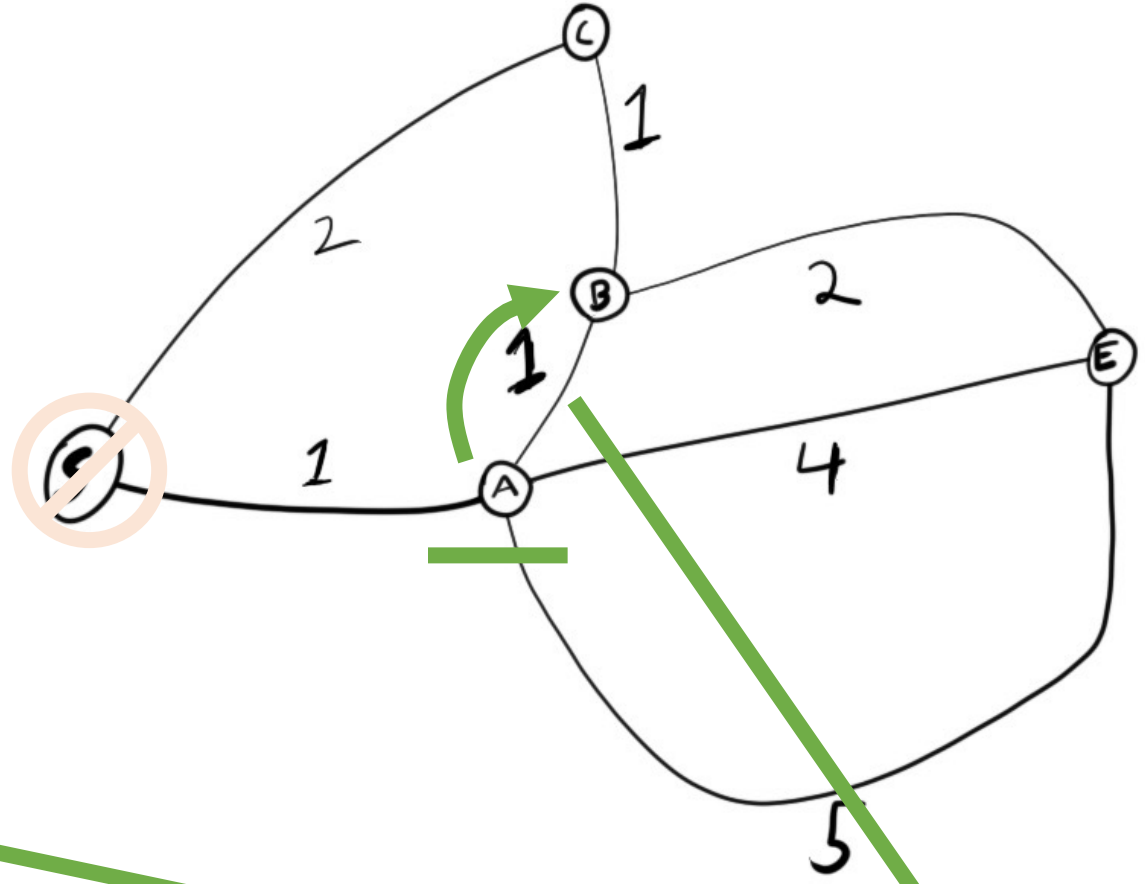
NODES	DISTANCE FROM START	PREVIOUS
START	0	Ø
A	1	START
B	∞	Ø
C	2	START
END	∞	Ø



DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM PREVIOUS NODES

NODES	DISTANCE FROM START	PREVIOUS
START	0	Ø
A	1	START
B	∞	Ø
C	2	START
END	∞	Ø



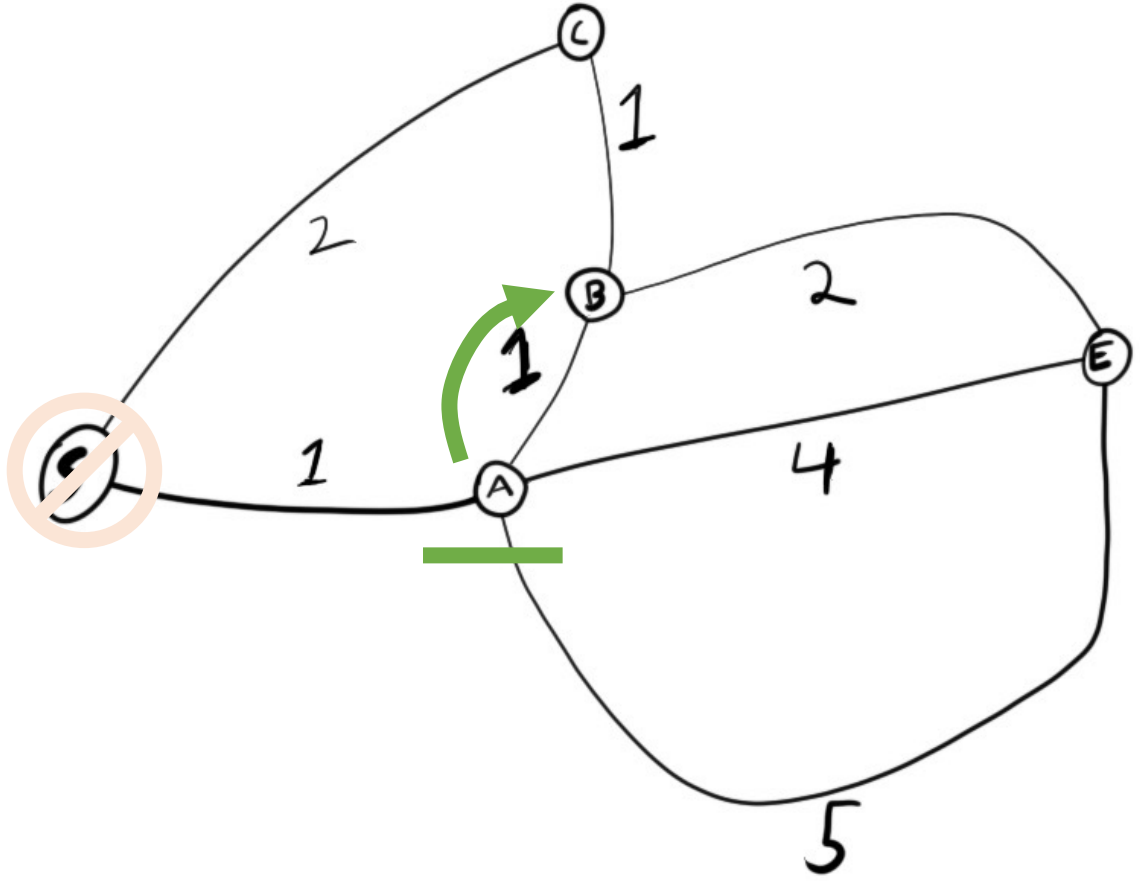
DISTANCE FROM START TO B

D_SB = D_SA + D_AB = 1 + 1

DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM PREVIOUS NODES

NODES	DISTANCE FROM START	PREVIOUS
START	0	Ø
A	1	START
B	∞	Ø
C	2	START
END	∞	Ø



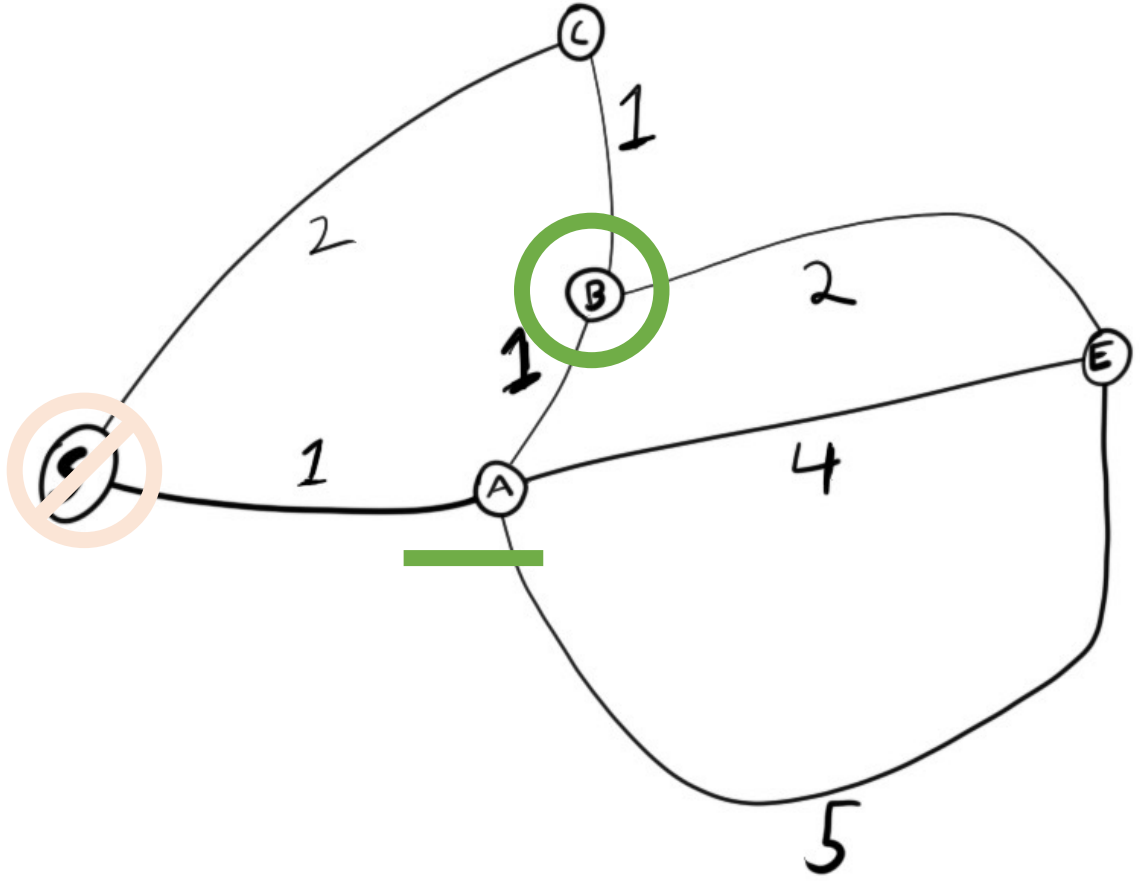
DISTANCE FROM START TO B

D_SB = D_SA + D_AB = 1 + 1

DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM PREVIOUS NODES

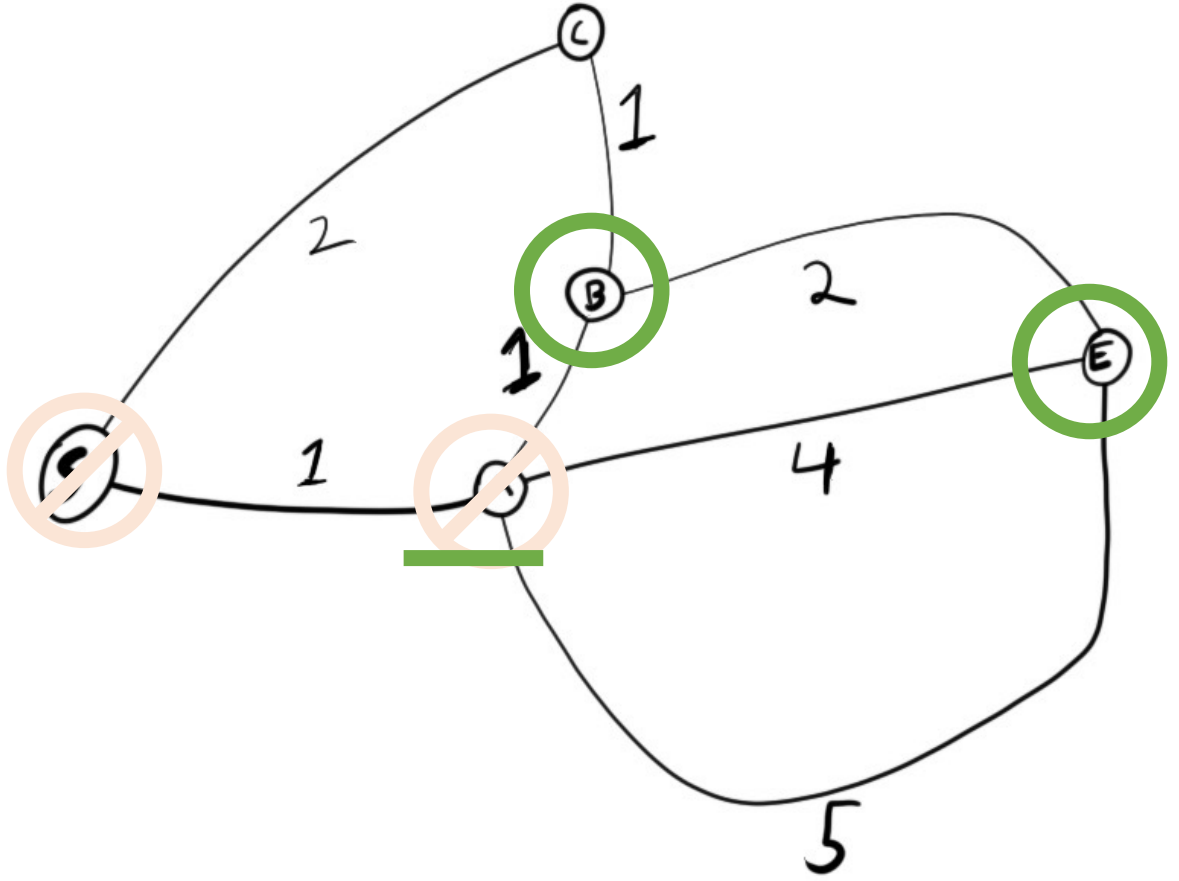
NODES	DISTANCE FROM START	PREVIOUS
START	0	Ø
A	1	START
B	2	A
C	2	START
END	5	A



DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM PREVIOUS NODES

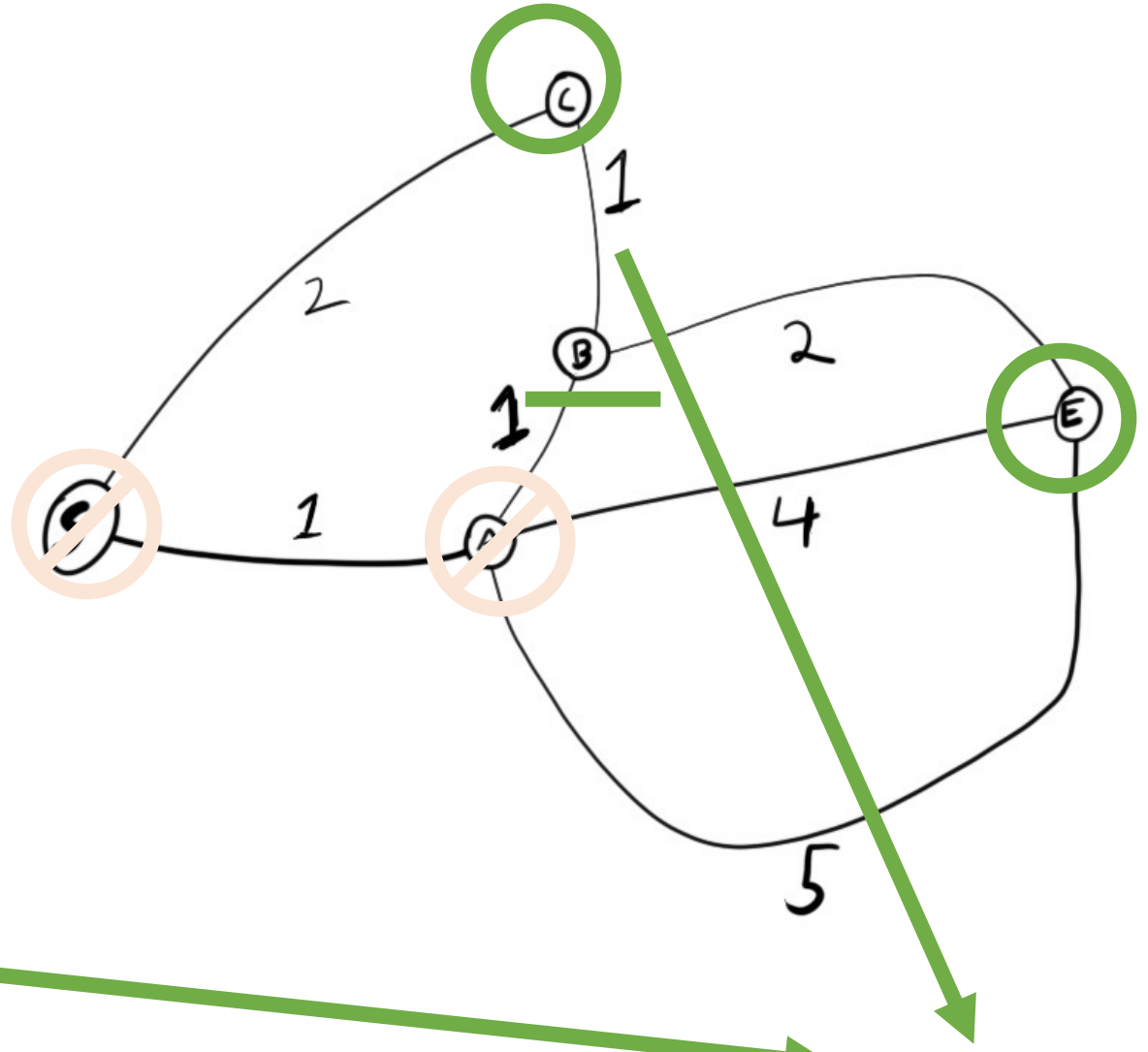
NODES	DISTANCE FROM START	PREVIOUS
START	0	Ø
A	1	START
B	2	A
C	2	START
END	5	A



DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM PREVIOUS NODES

NODES	DISTANCE FROM START	PREVIOUS
START	0	Ø
A	1	START
B	2	A
C	2	START
END	5	A

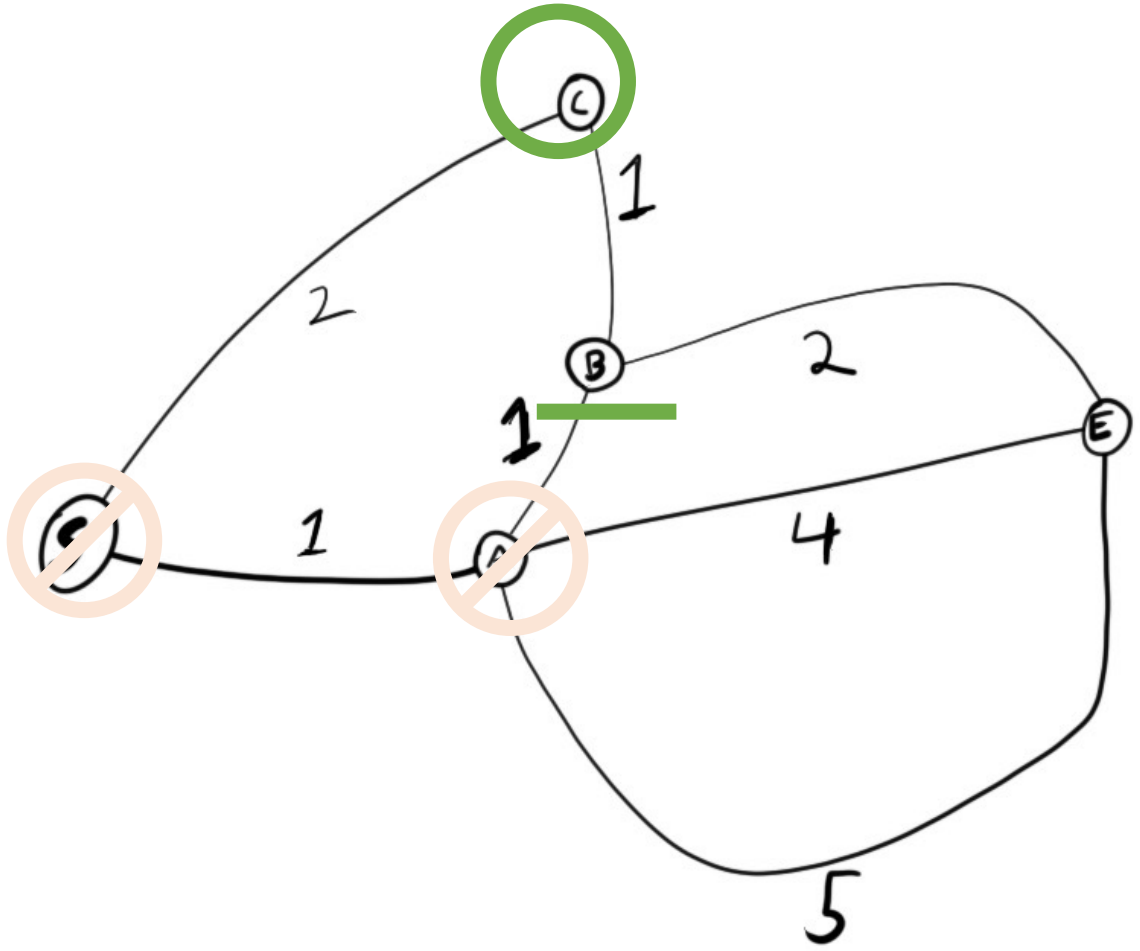


$D\_SC = D\_SB + D\_BC = 2 + 1$

DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM PREVIOUS NODES

NODES	DISTANCE FROM START	PREVIOUS
START	0	Ø
A	1	START
B	2	A
C	2	START
END	5	A

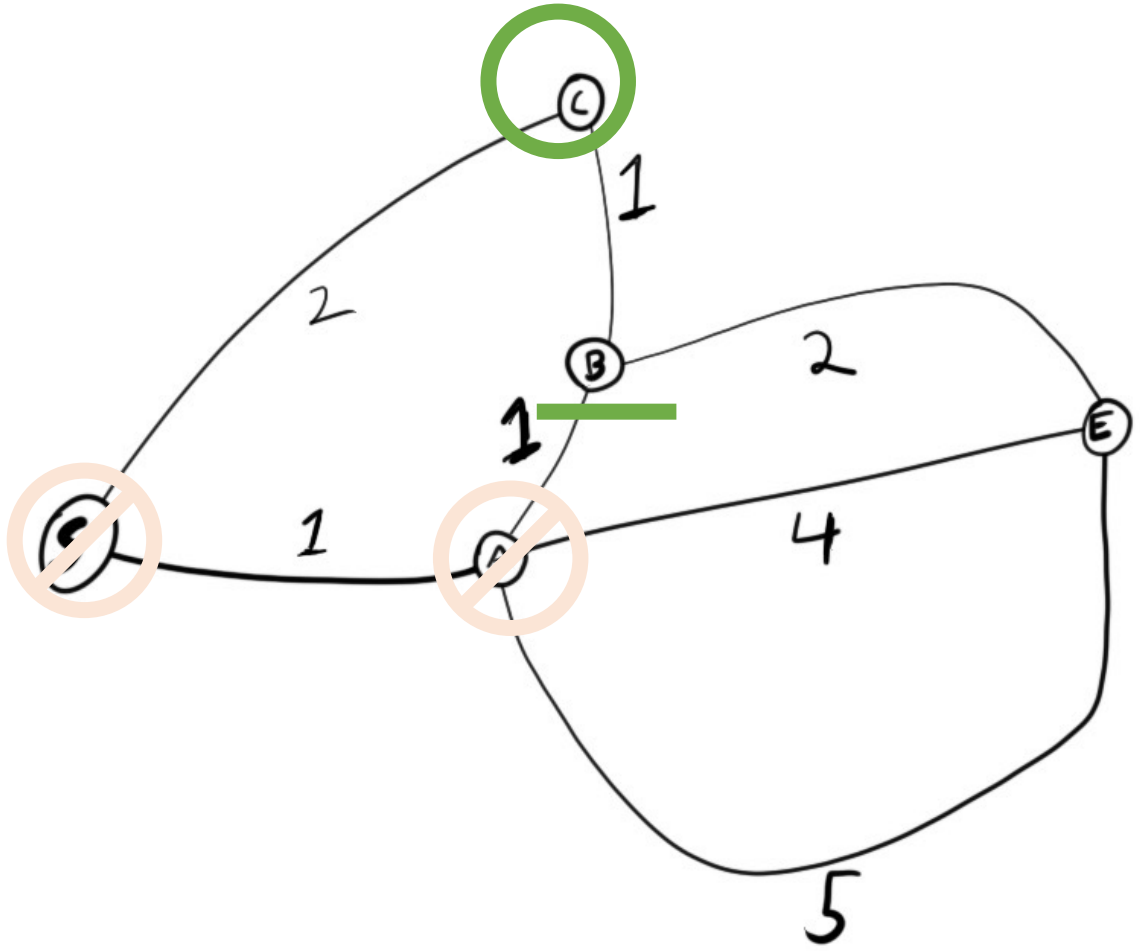


D_SC = D_SB + D_BC = 2 + 1

DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM PREVIOUS NODES

NODES	DISTANCE FROM START	PREVIOUS
START	0	Ø
A	1	START
B	2	A
C	2	START
END	5	A

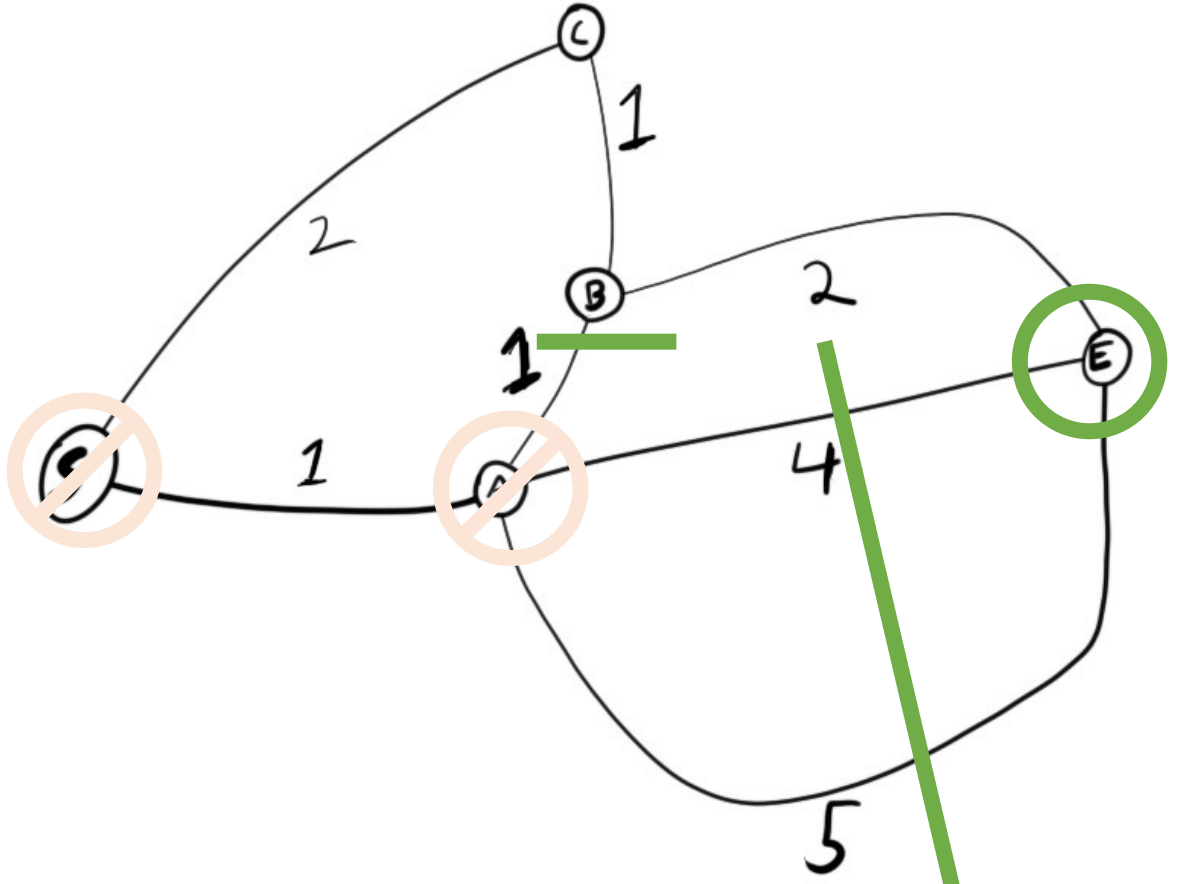


$D\_SC = D\_SB + D\_BC = 2 + 1$

DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM PREVIOUS NODES

NODES	DISTANCE FROM START	PREVIOUS
START	0	Ø
A	1	START
B	2	A
C	2	START
END	5	A

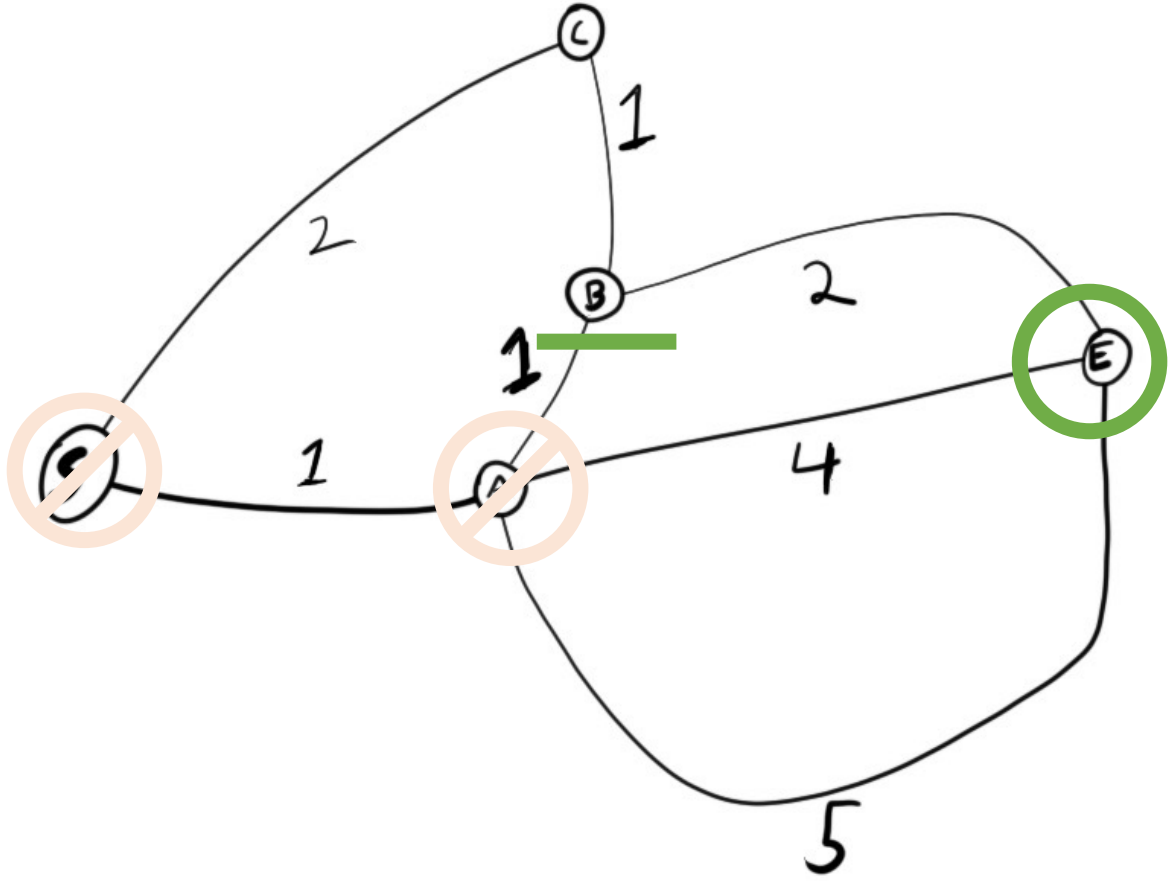


$D\_SE = D\_SB + D\_BE = 2 + 2$

DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM PREVIOUS NODES

NODES	DISTANCE FROM START	PREVIOUS
START	0	Ø
A	1	START
B	2	A
C	2	START
END	5	A

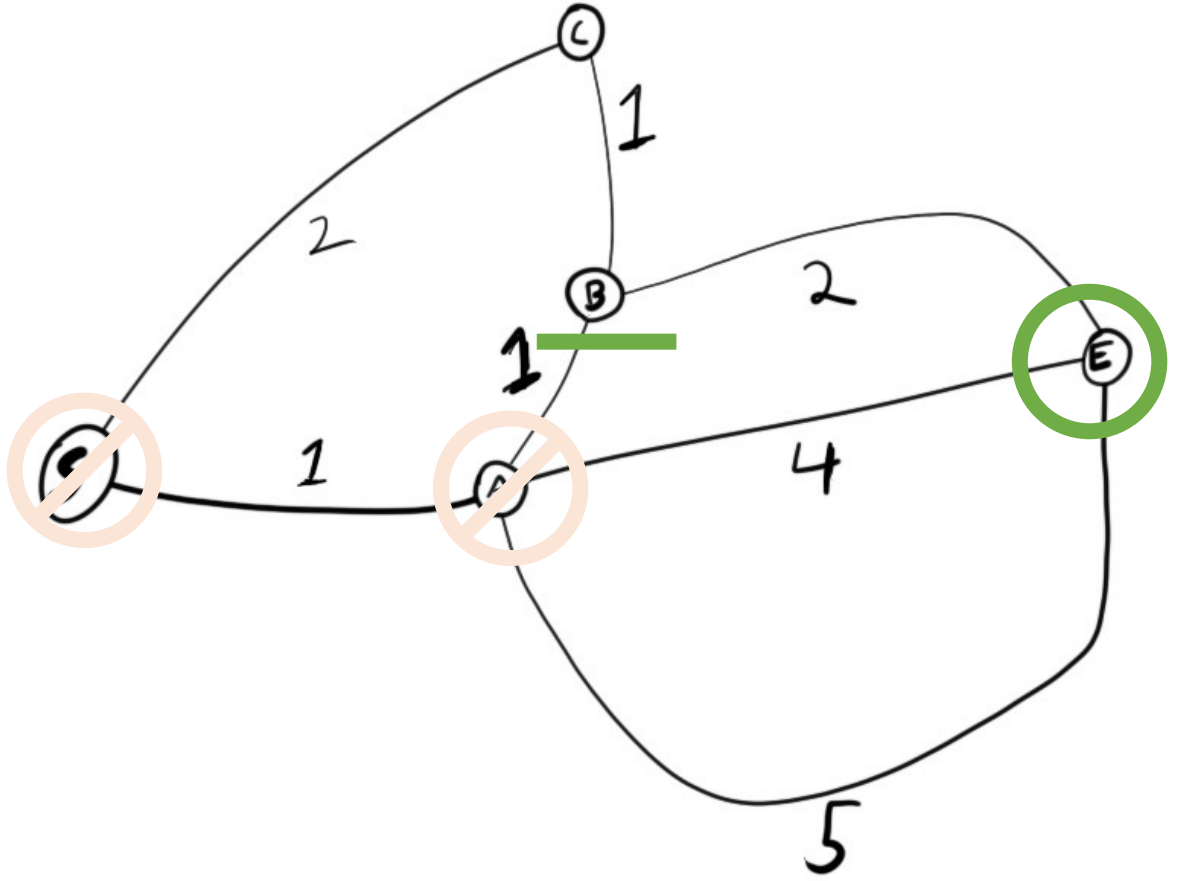


$D\_SE = D\_SB + D\_BE = 2 + 2$

DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM PREVIOUS NODES

NODES	DISTANCE FROM START	PREVIOUS
START	0	Ø
A	1	START
B	2	A
C	2	START
END	4	B



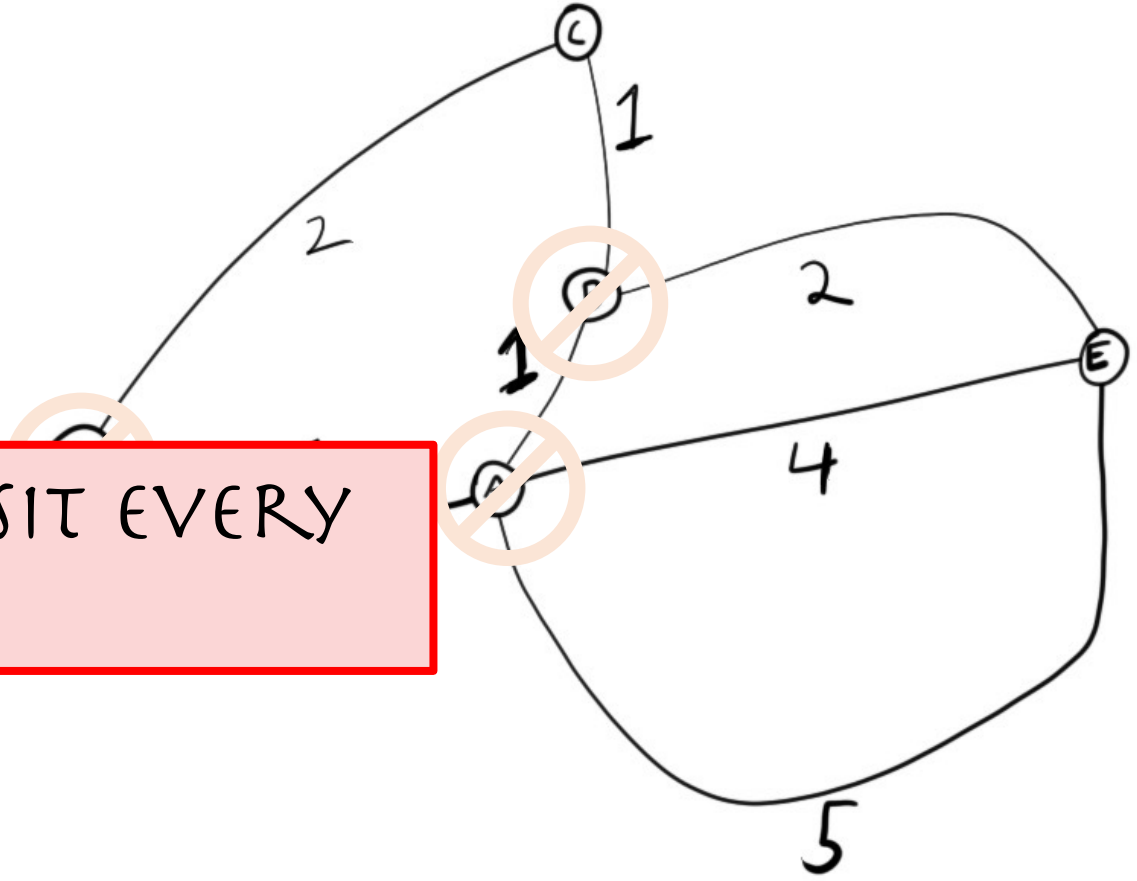
$D\_SE = D\_SB + D\_BE = 2 + 2$

DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM PREVIOUS NODES

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

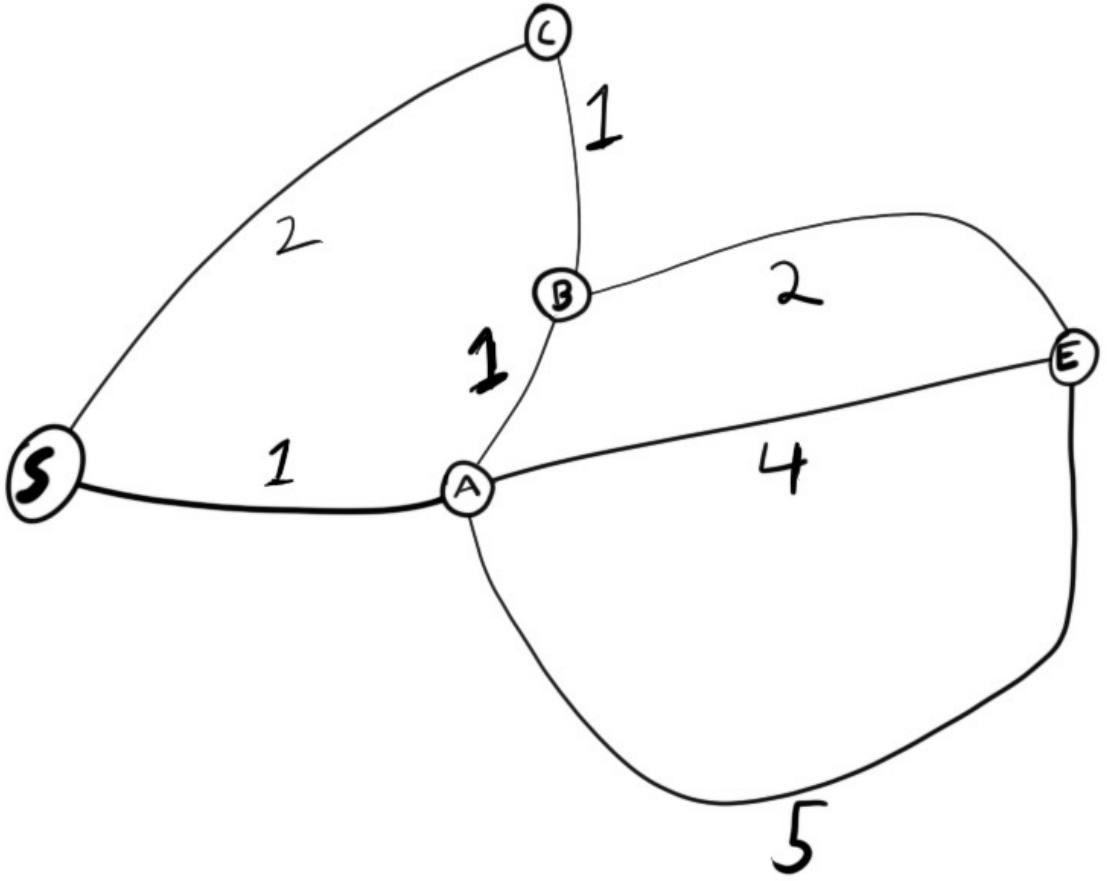
YOU HAVE TO VISIT EVERY NODE

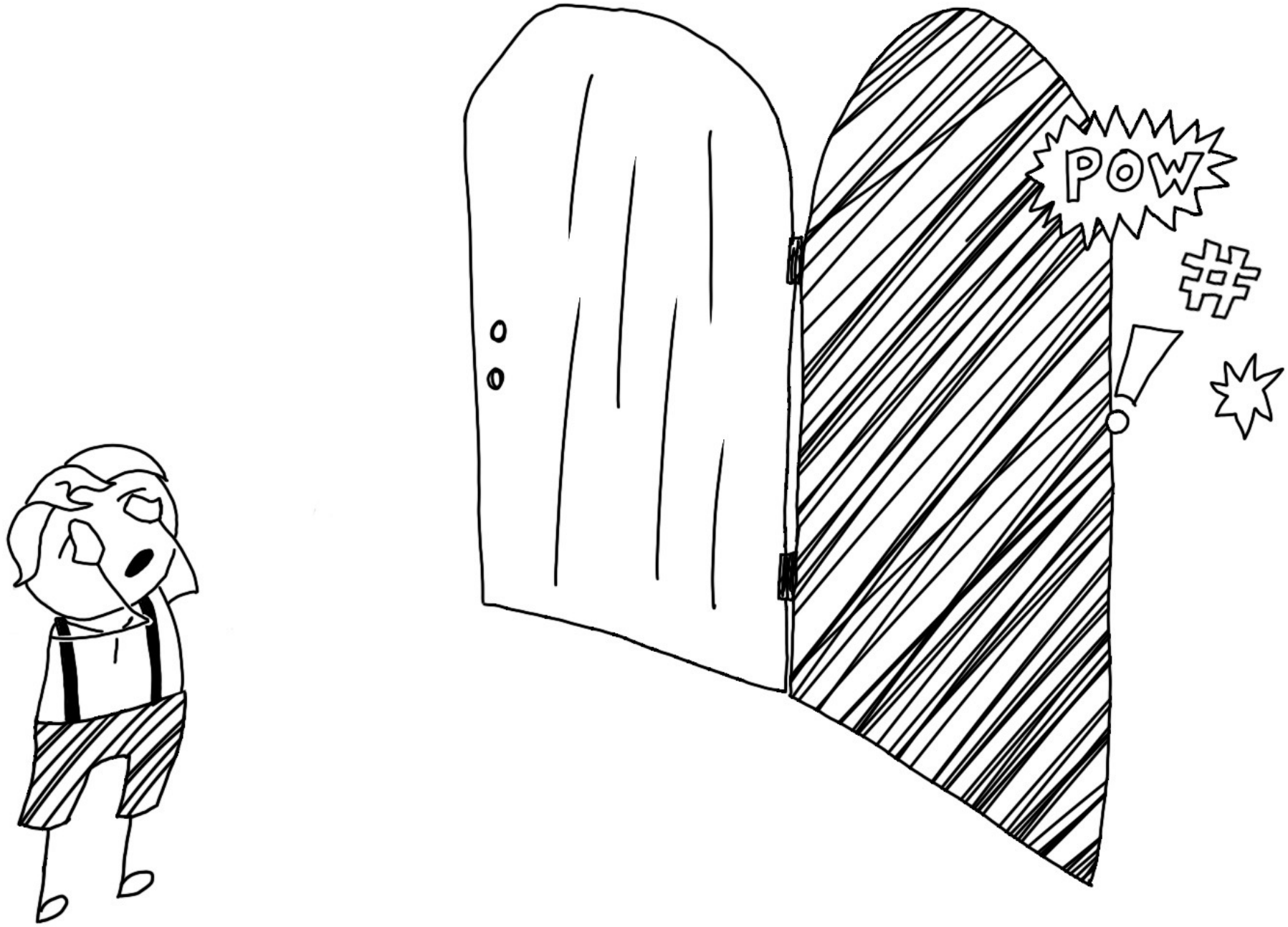


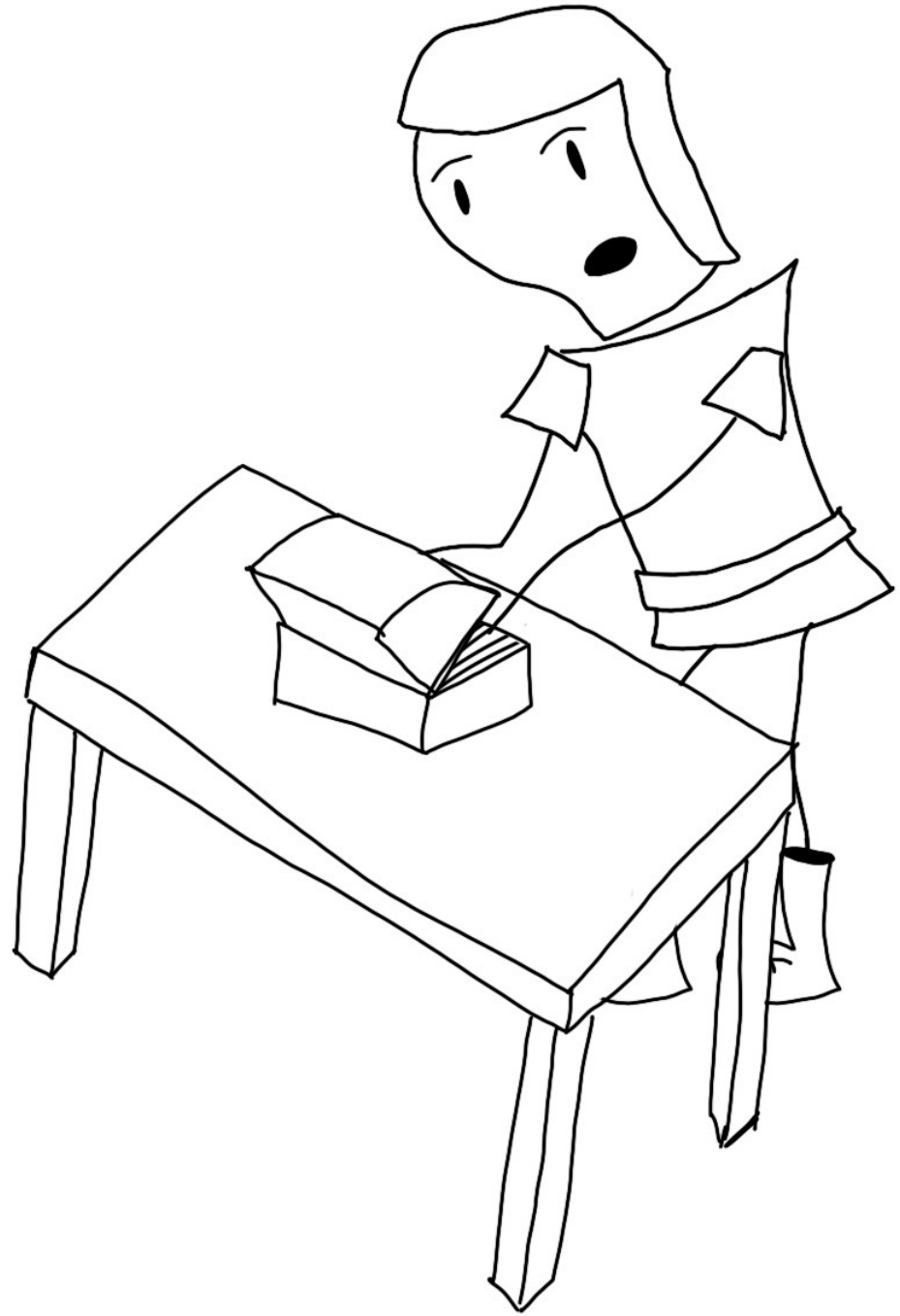
DIJKSTRA'S ALGORITHM

TRACKING DISTANCE FORM PREVIOUS NODES

NODES	DISTANCE FROM START	PREVIOUS
START	0	Ø
A	1	START
B	2	A
C	2	START
END	4	B

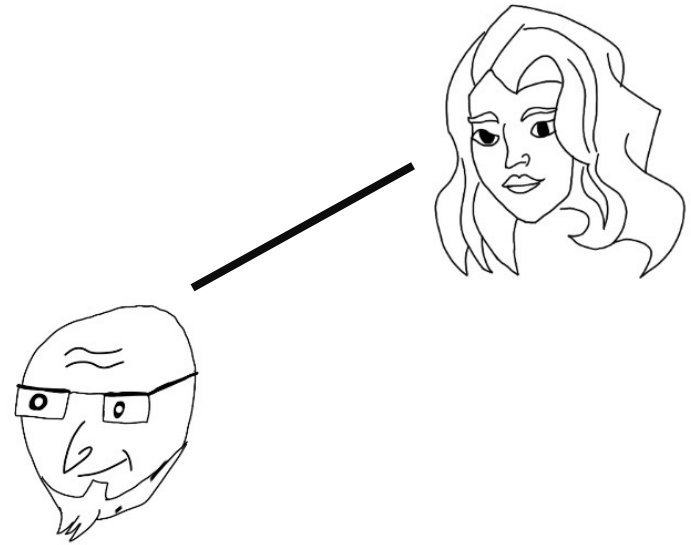


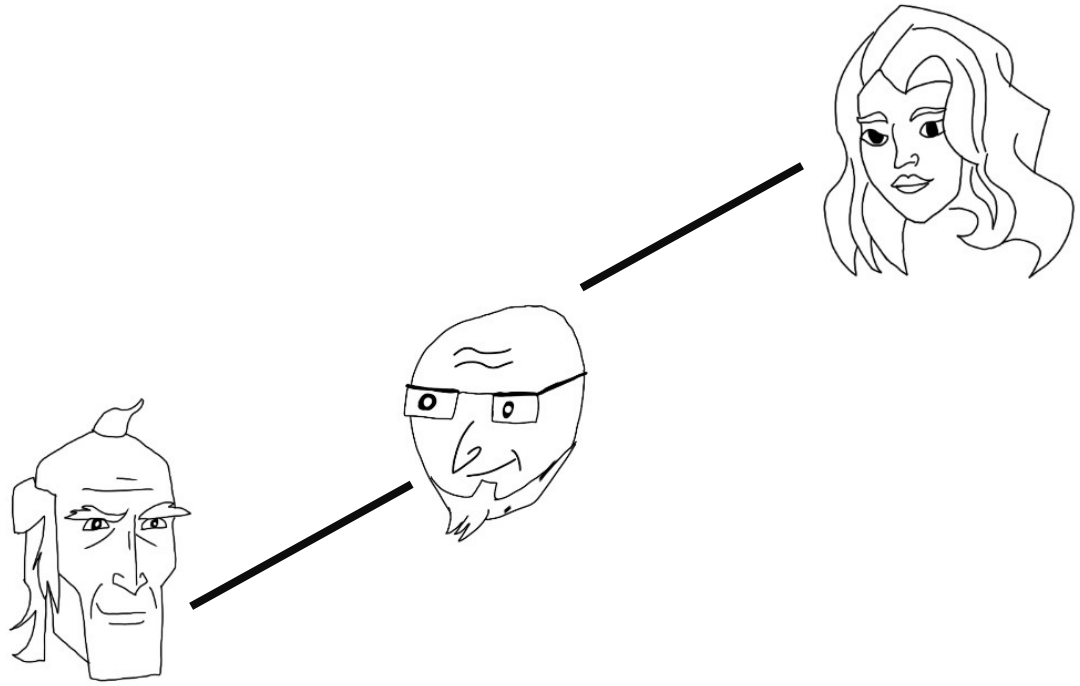

POW
#

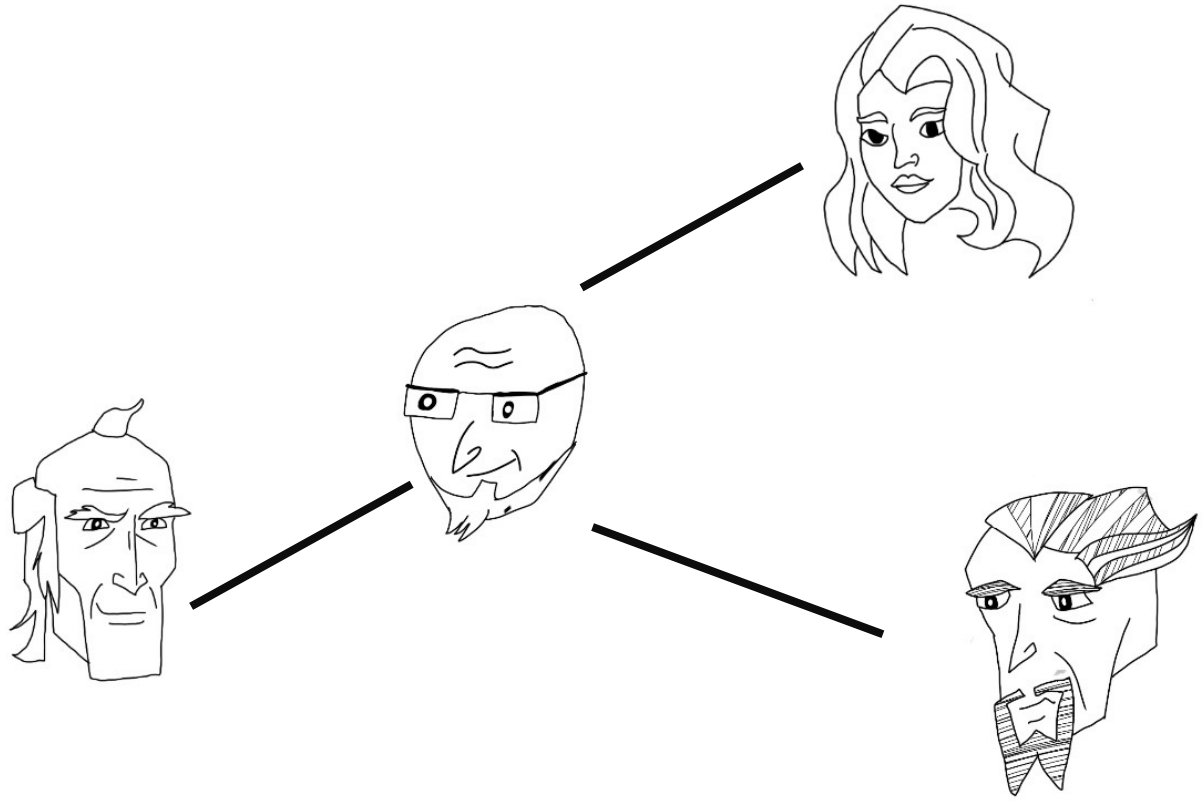


WHO'S REALLY IN CHARGE?!

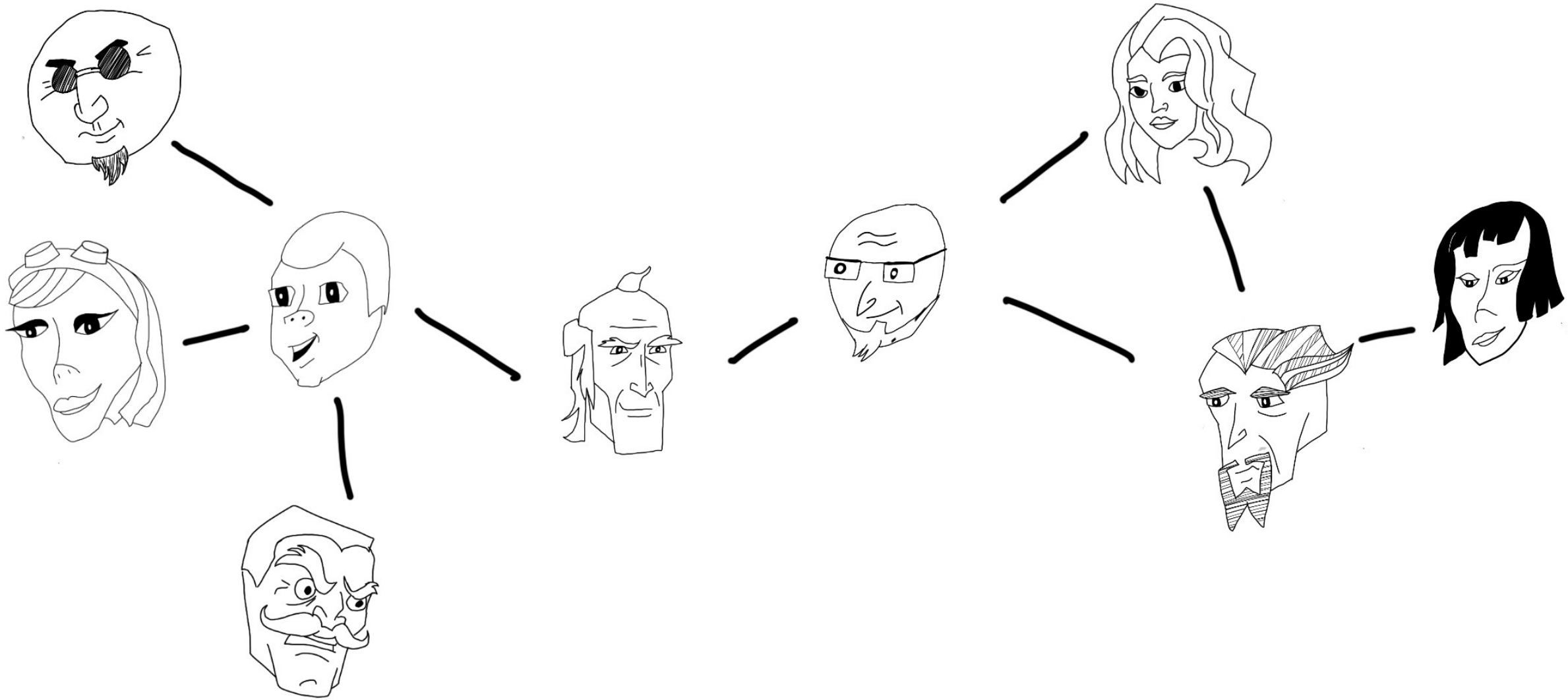
- YOU HAVE A LIST OF MESSAGES
- FRANK -> GINA
- GINA -> HECTOR
- HECTOR -> FRANK
- ?!?!?!?!








PEPE
SILVA



Who's really in charge?!

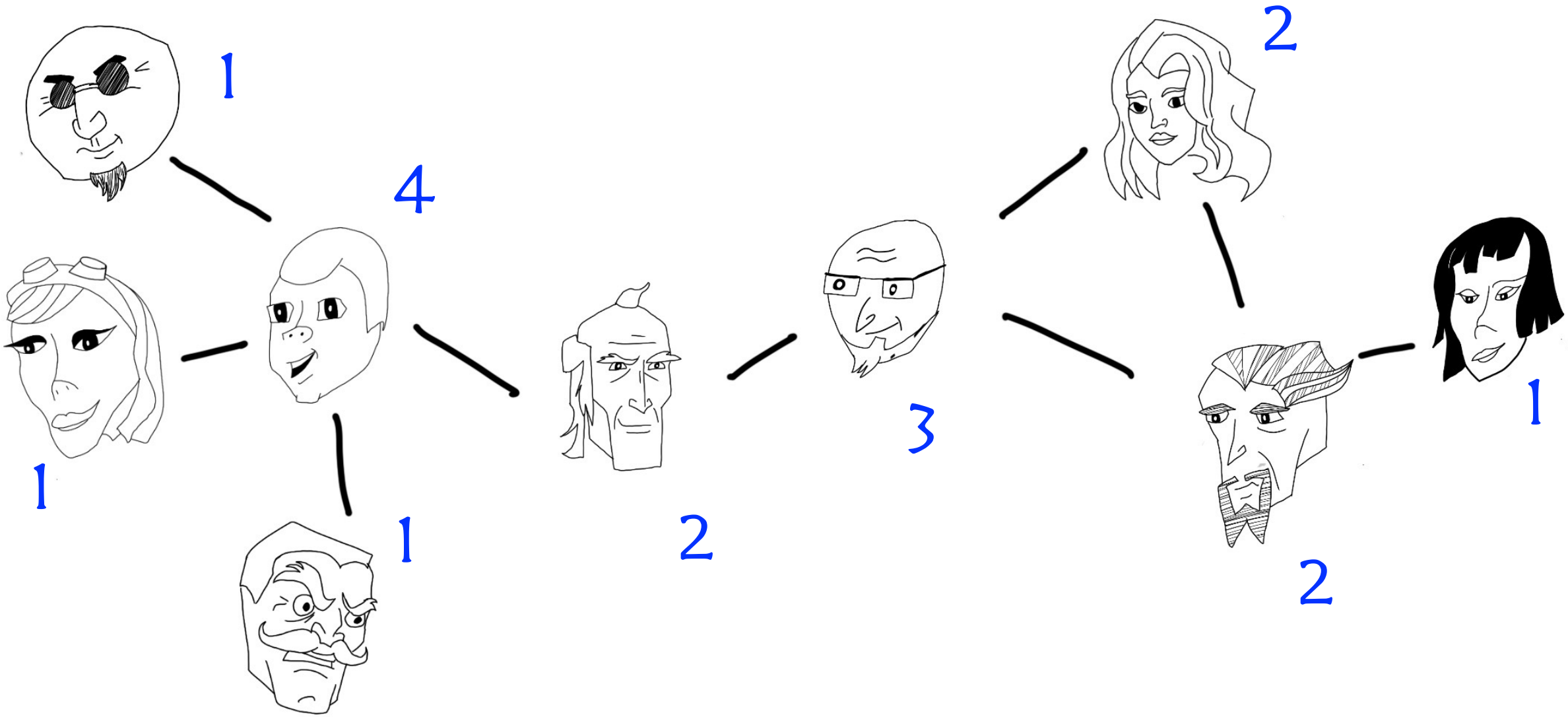
- Given a graph of people sending messages, how do you determine important people?
  - Articulation points
    - Nodes in a graph that reduce connectivity

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

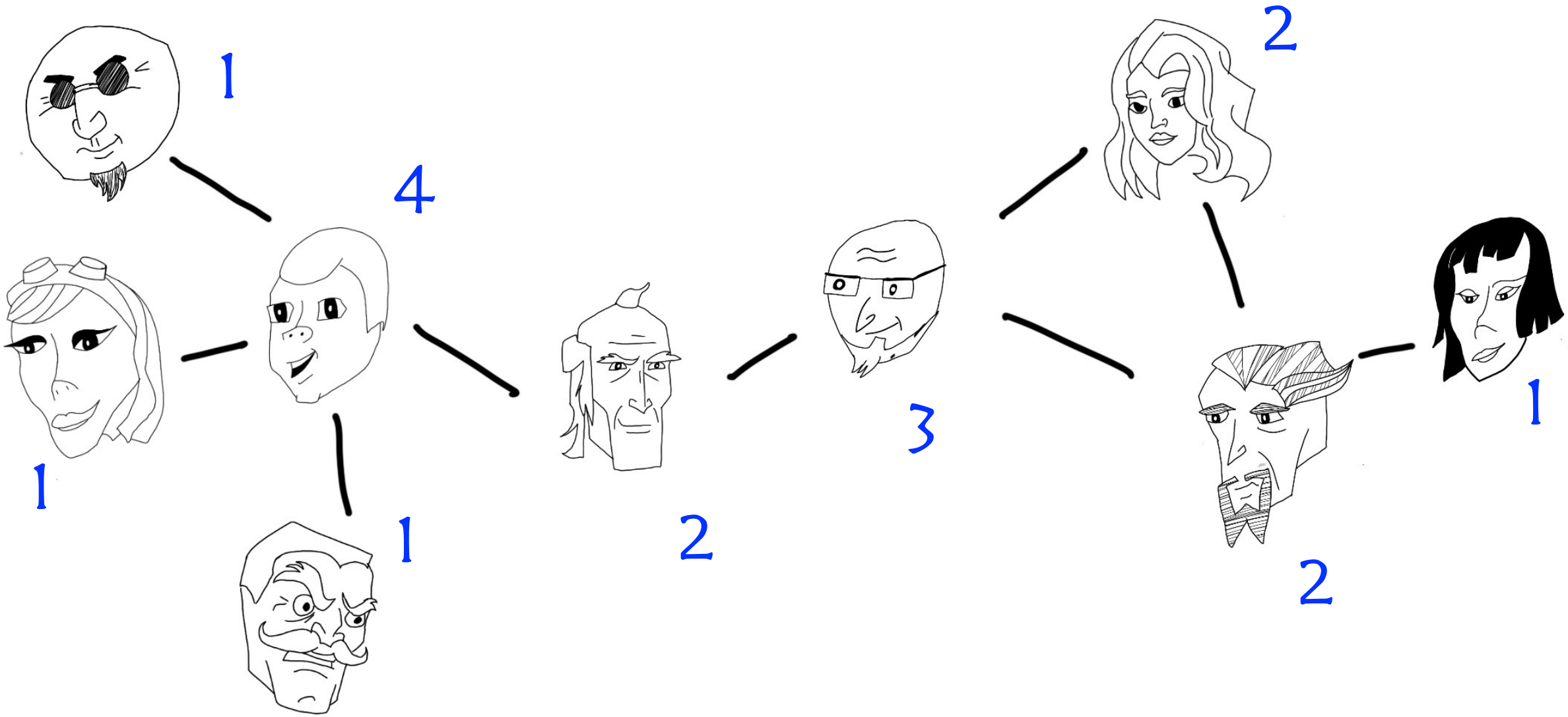
DEGREE CENTRALITY

- JUST COUNT THE DEGREES PER NODE



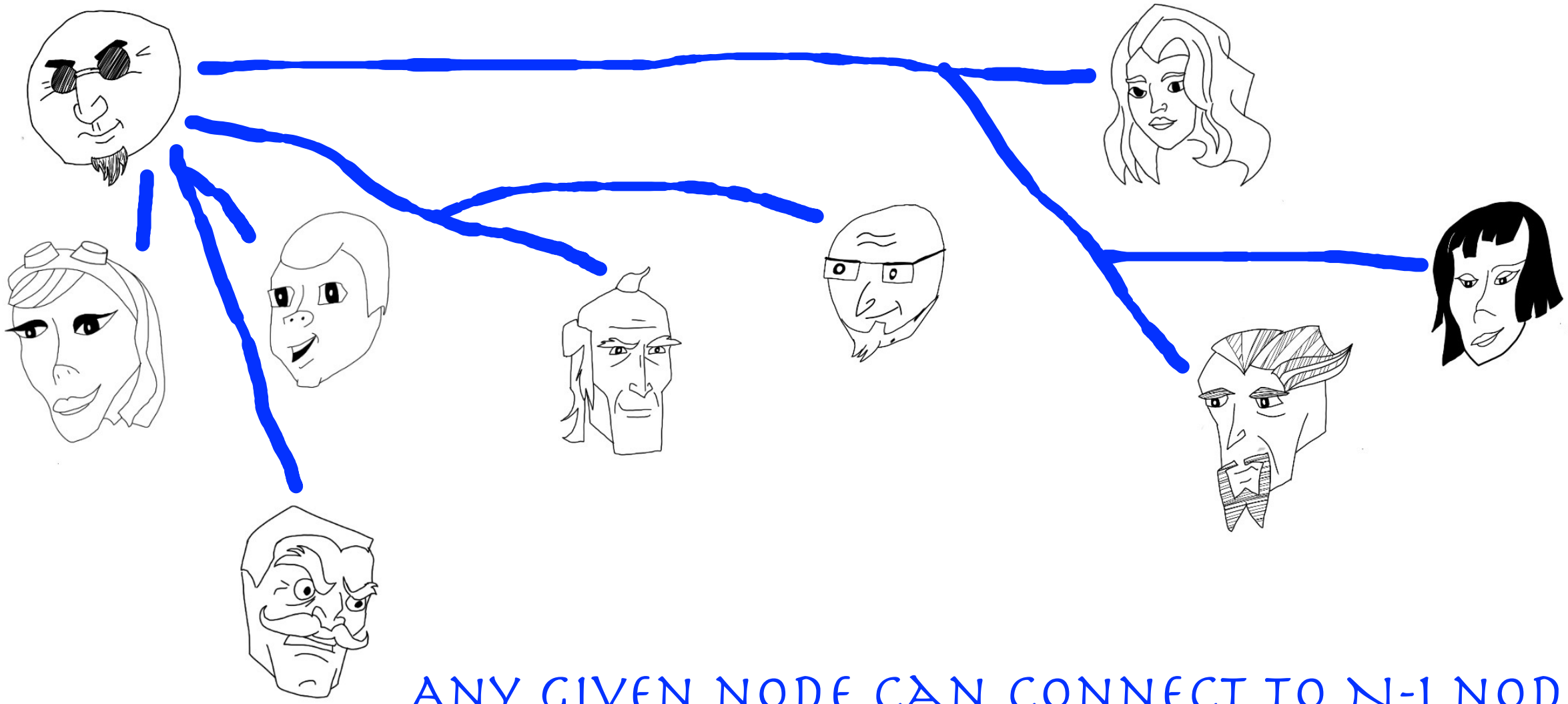
DEGREE CENTRALITY

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



DEGREE CENTRALITY

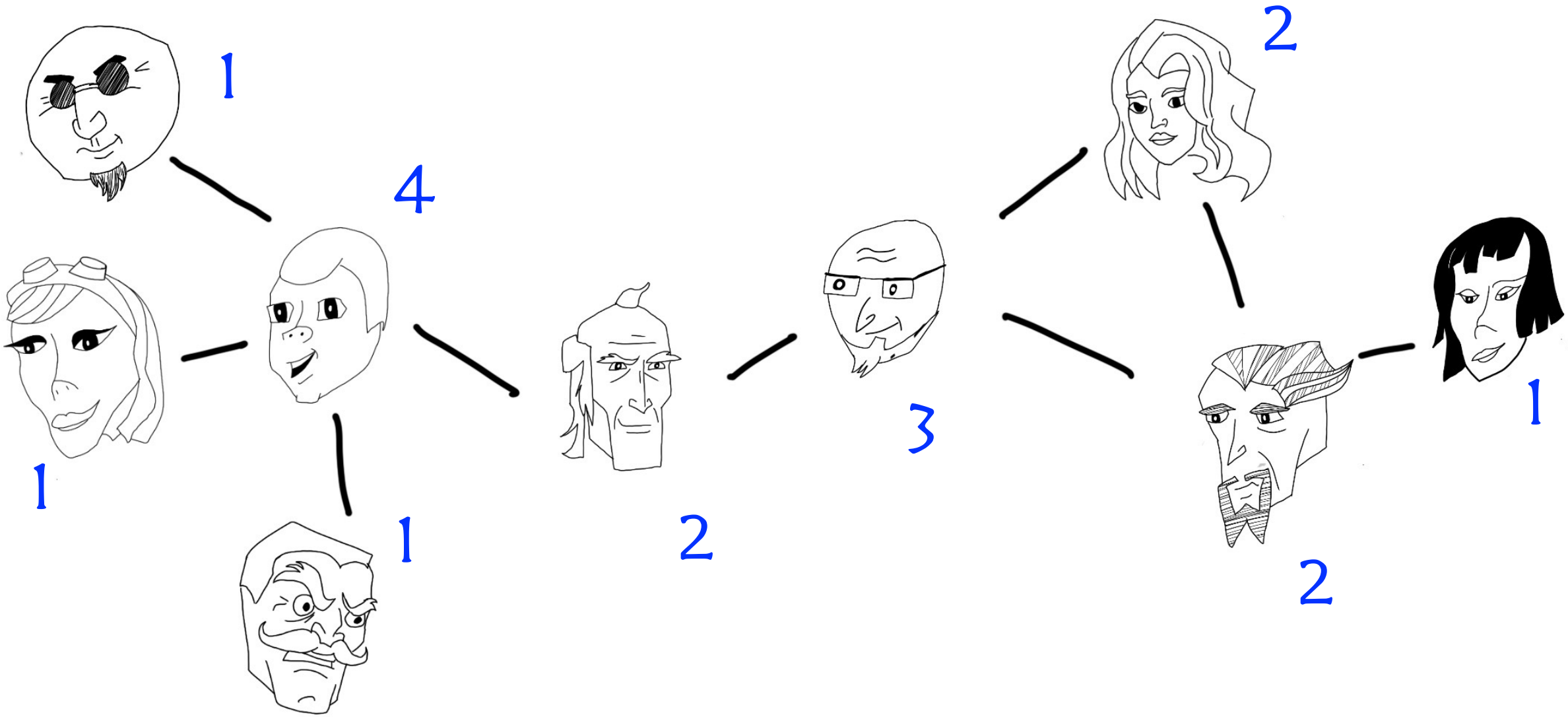
- ...AND NORMALIZE THEM BY MAX POSSIBLE DEGREE



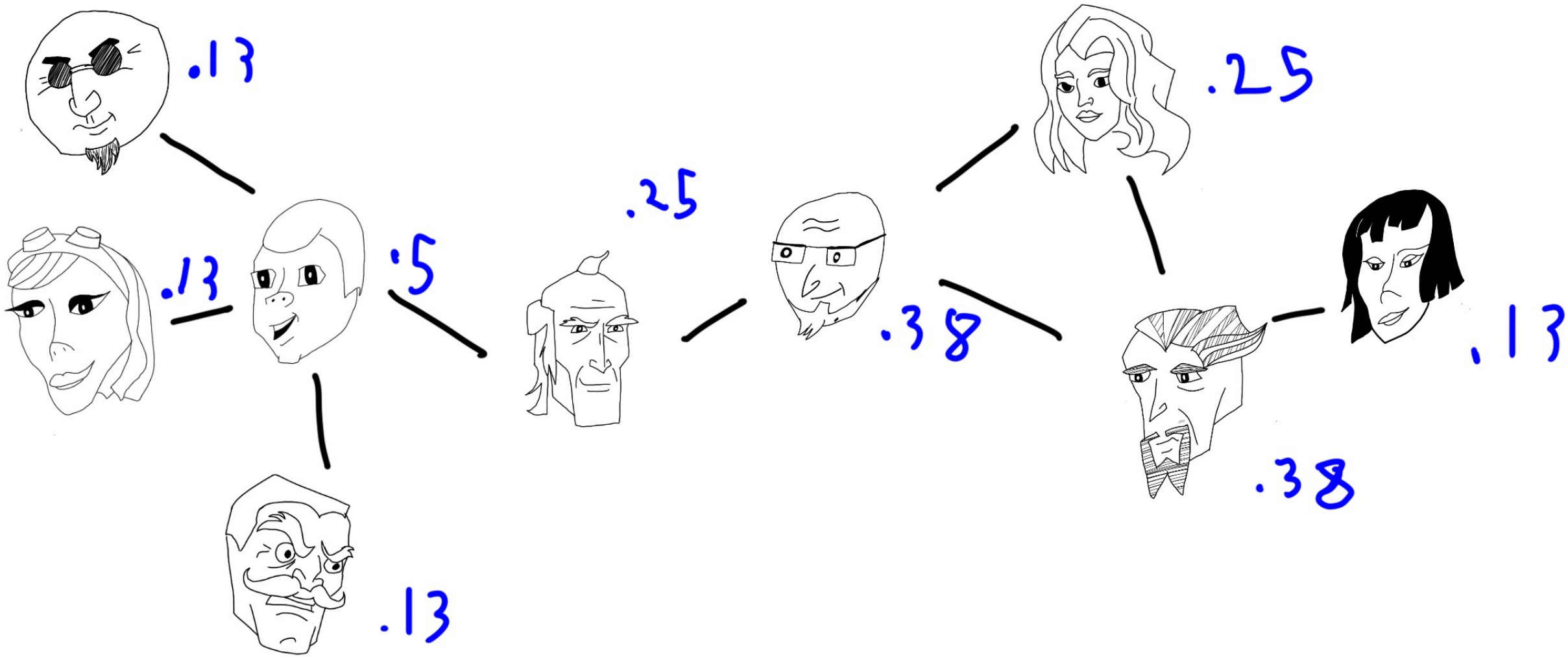
ANY GIVEN NODE CAN CONNECT TO N-1 NODES

DEGREE CENTRALITY

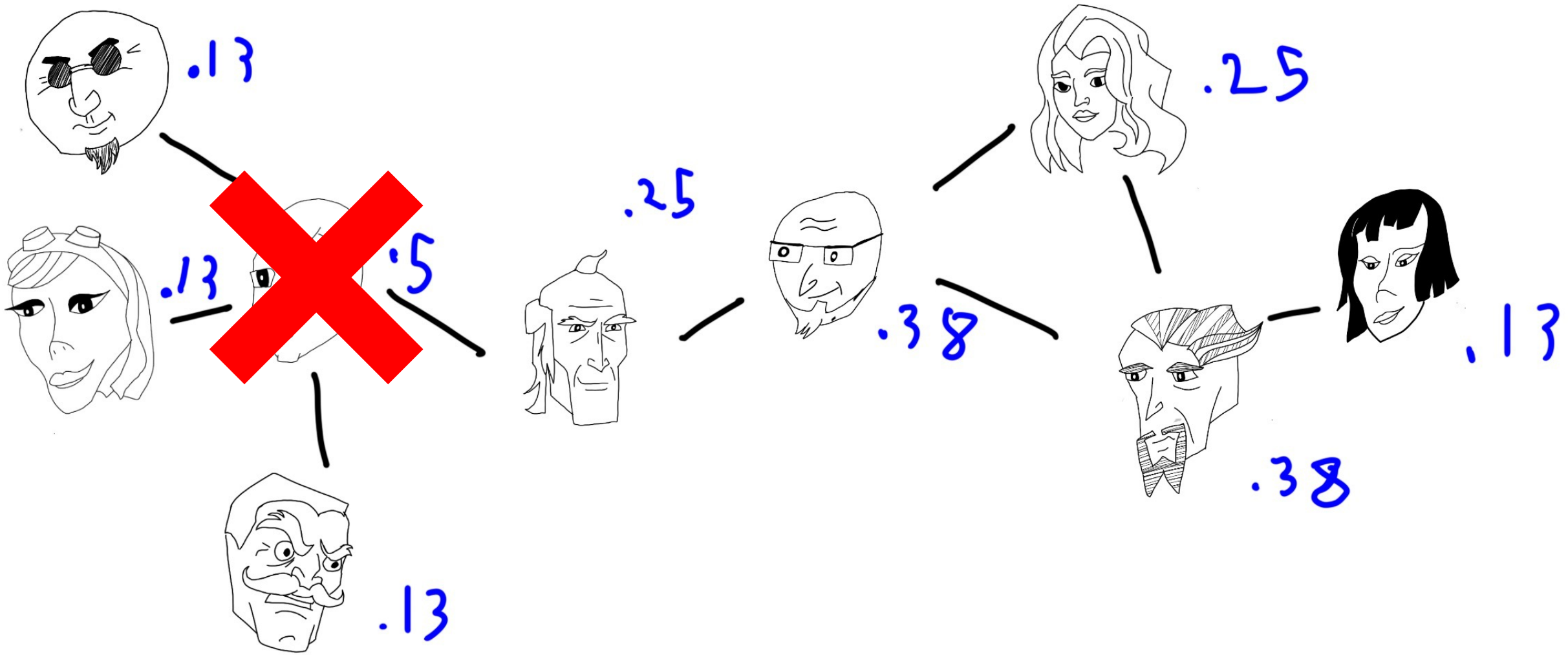
- ...AND NORMALIZE THEM BY MAX POSSIBLE DEGREE

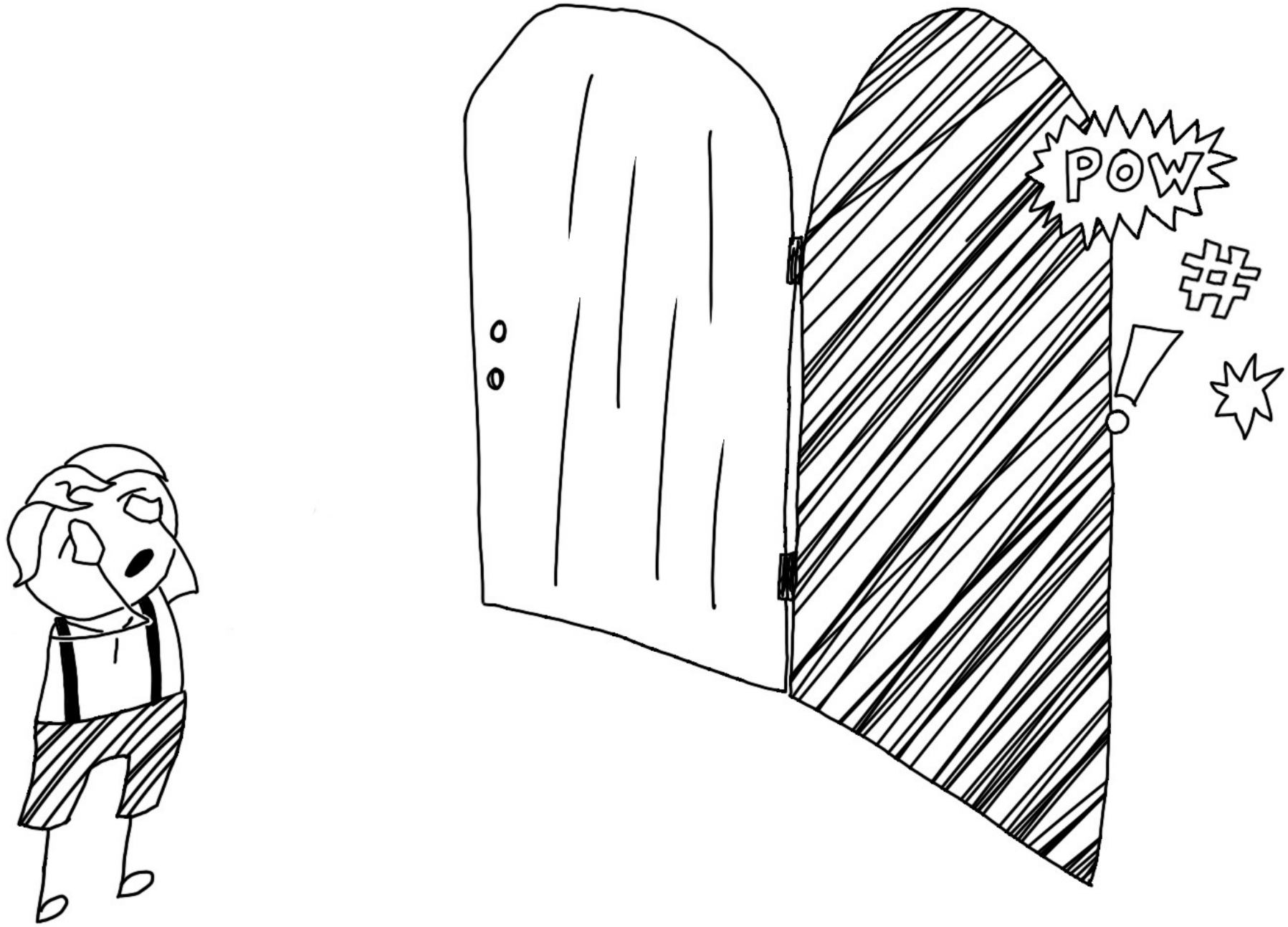


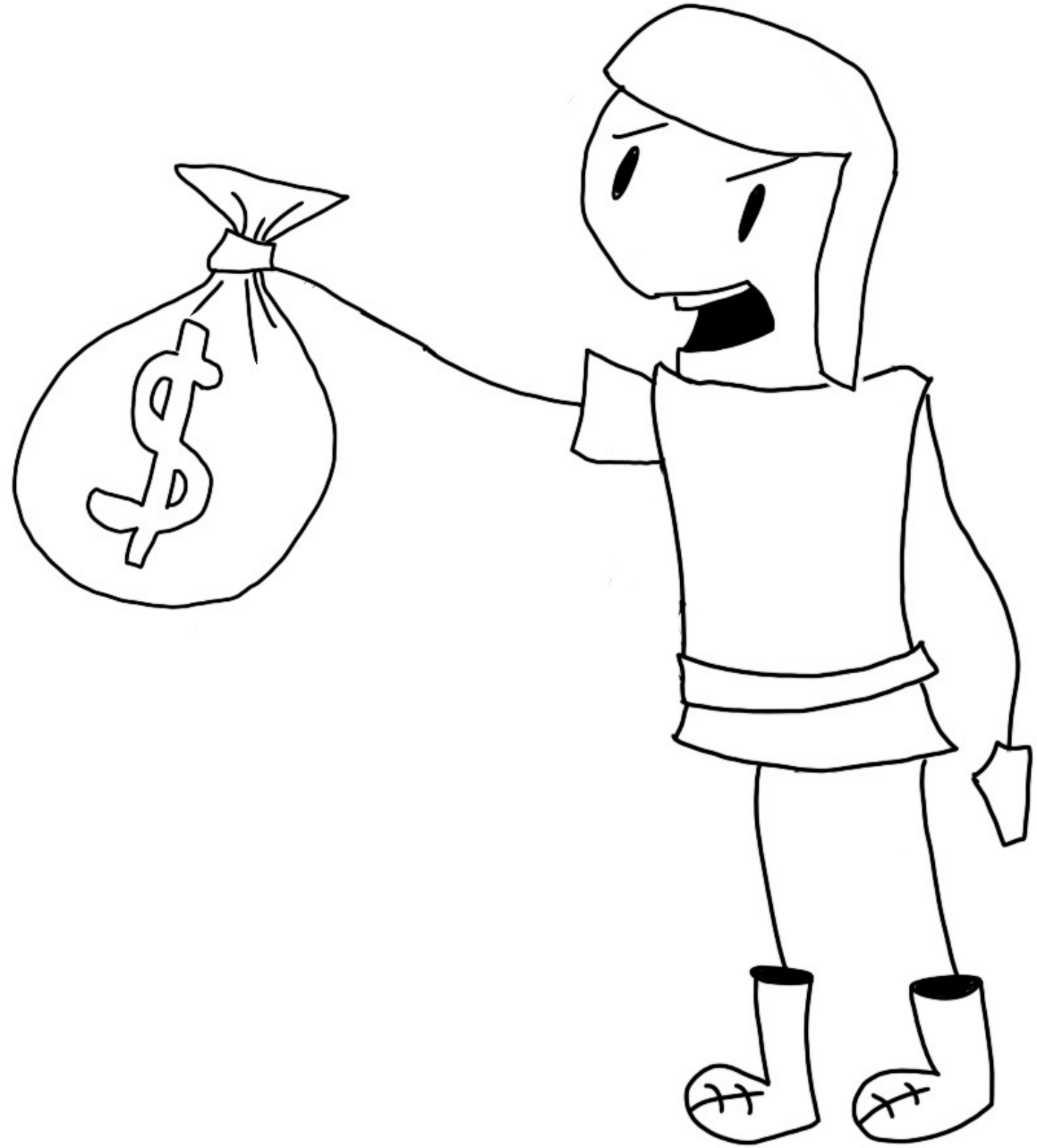
DEGREE CENTRALITY

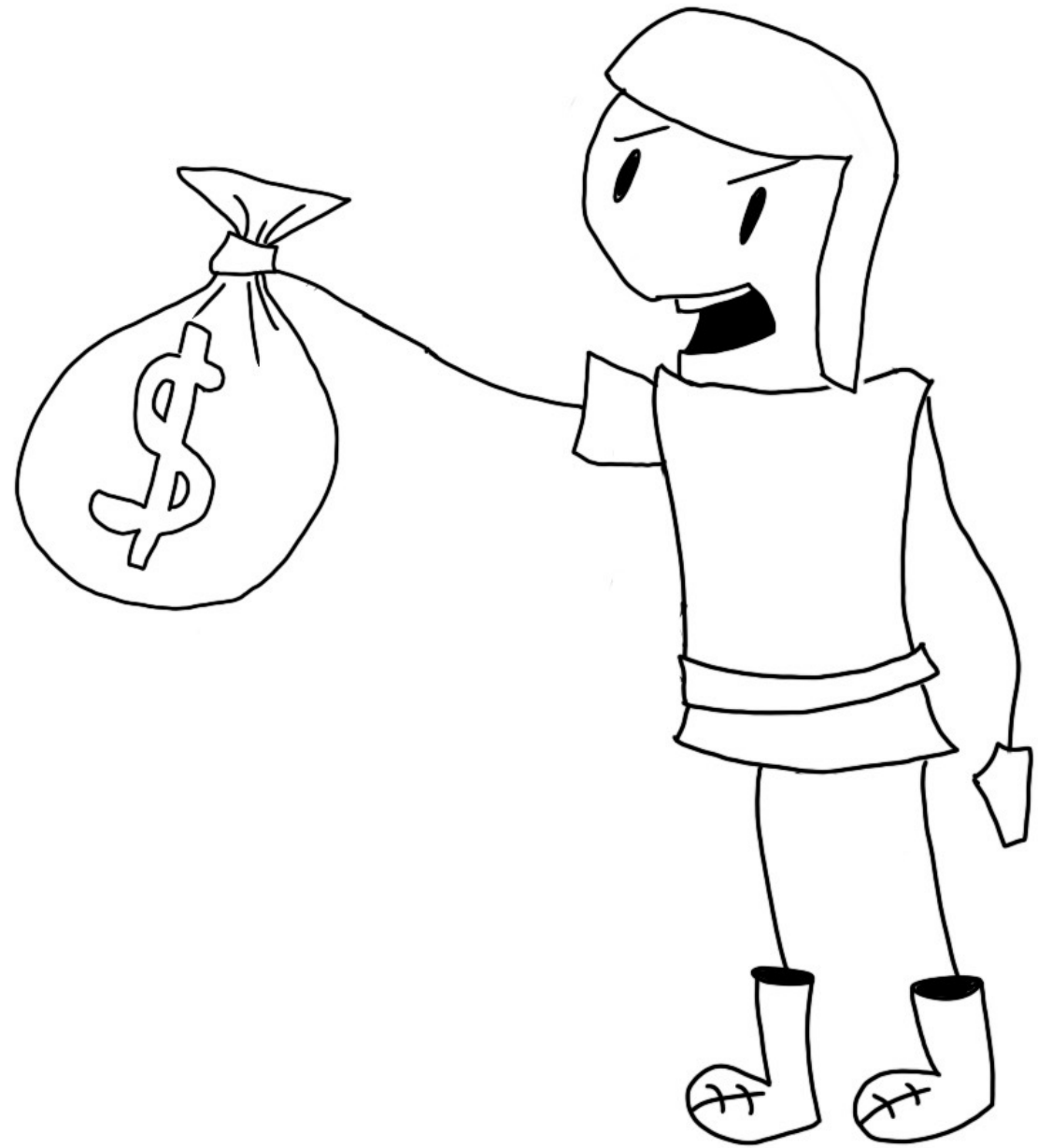


DEGREE CENTRALITY




POW
#





TOWN CHARTER

I. LEGISLATIVE DEPARTMENT
    1. MAYOR

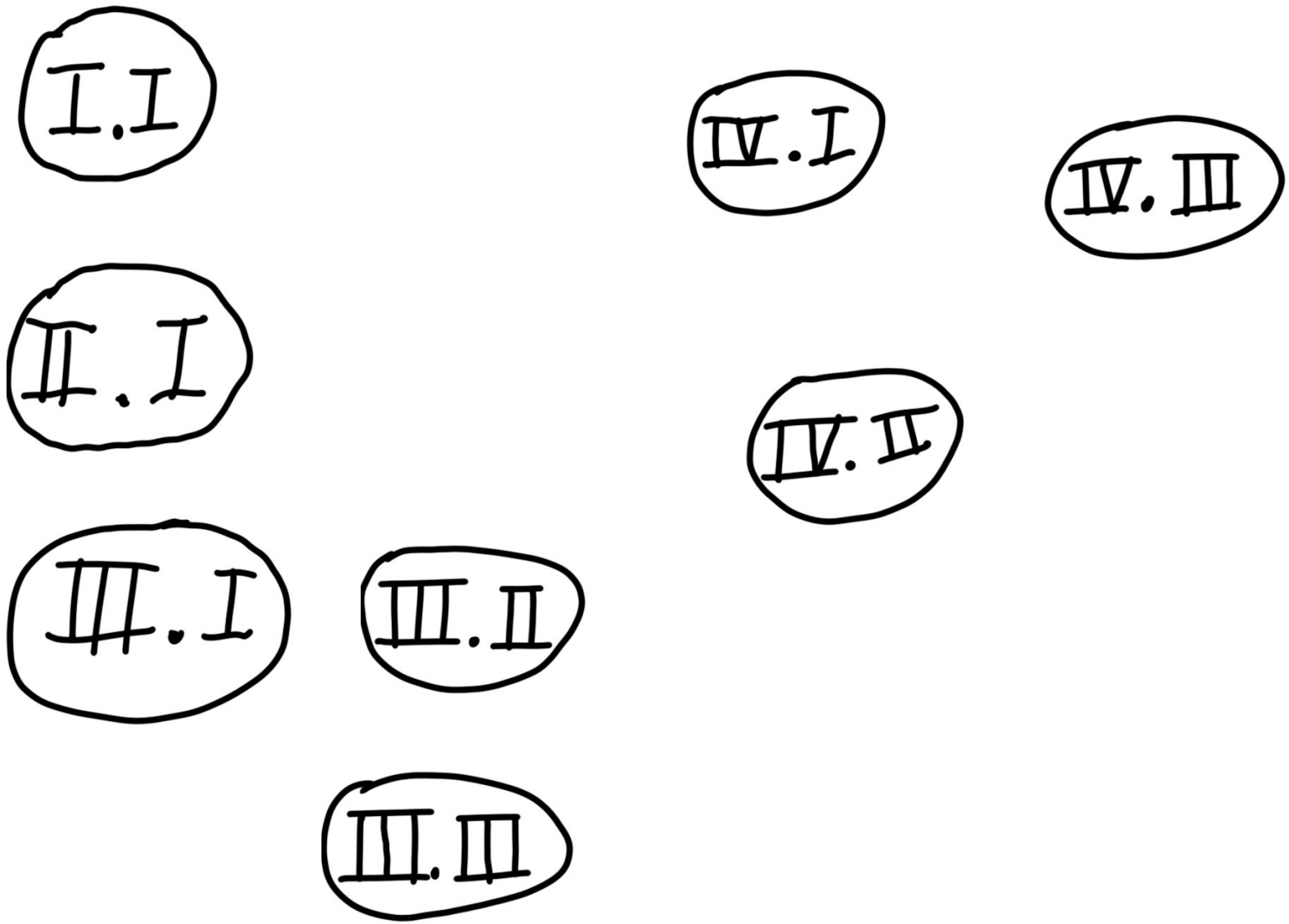
II. ADMINISTRATIVE DEPARTMENT
    1. MAYOR'S AIDE

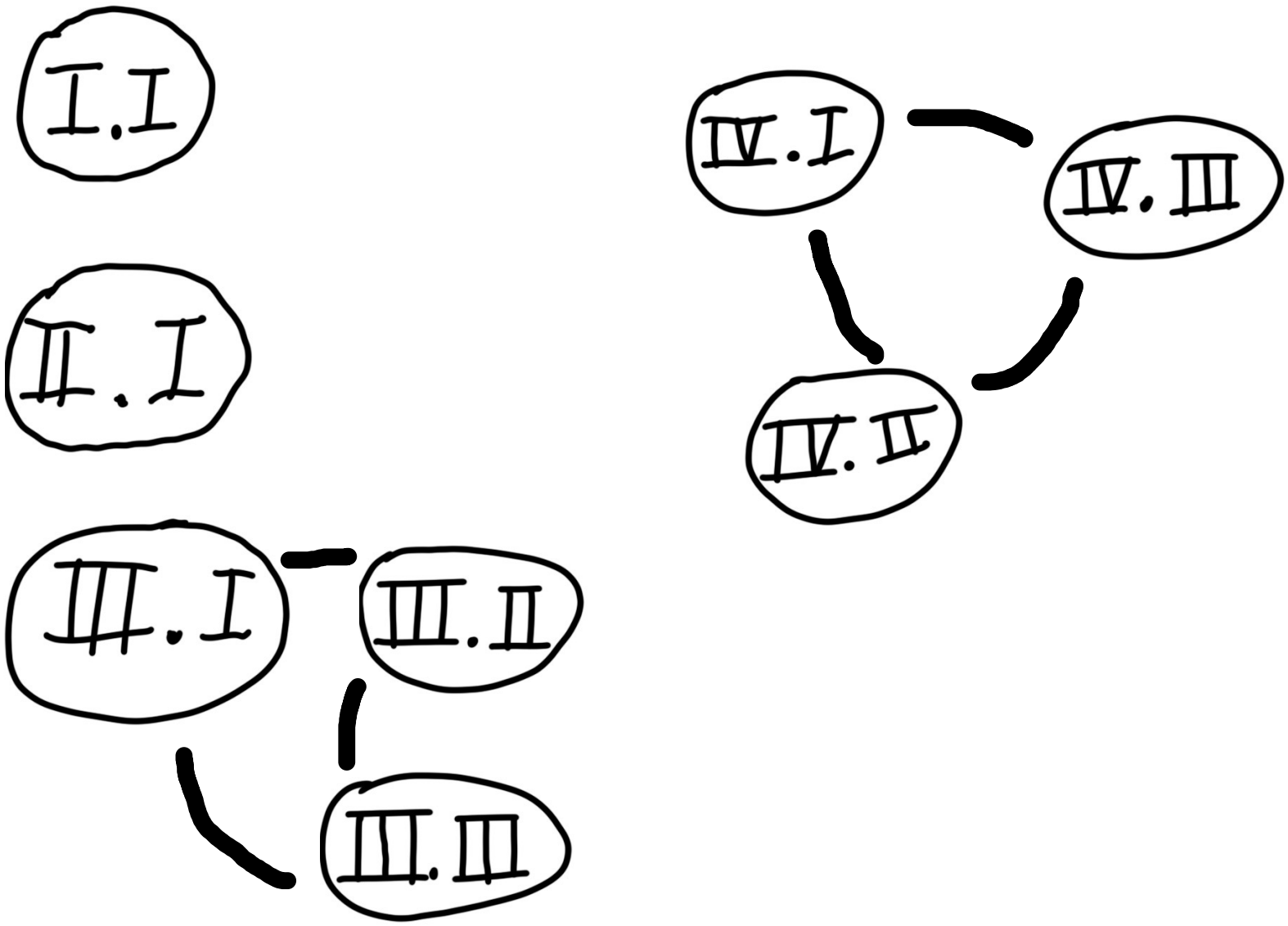
III. PUBLIC WORKS
    1. WATER WELL MAINTENANCE
    II. ROADS
    III. ANIMAL CONTROL

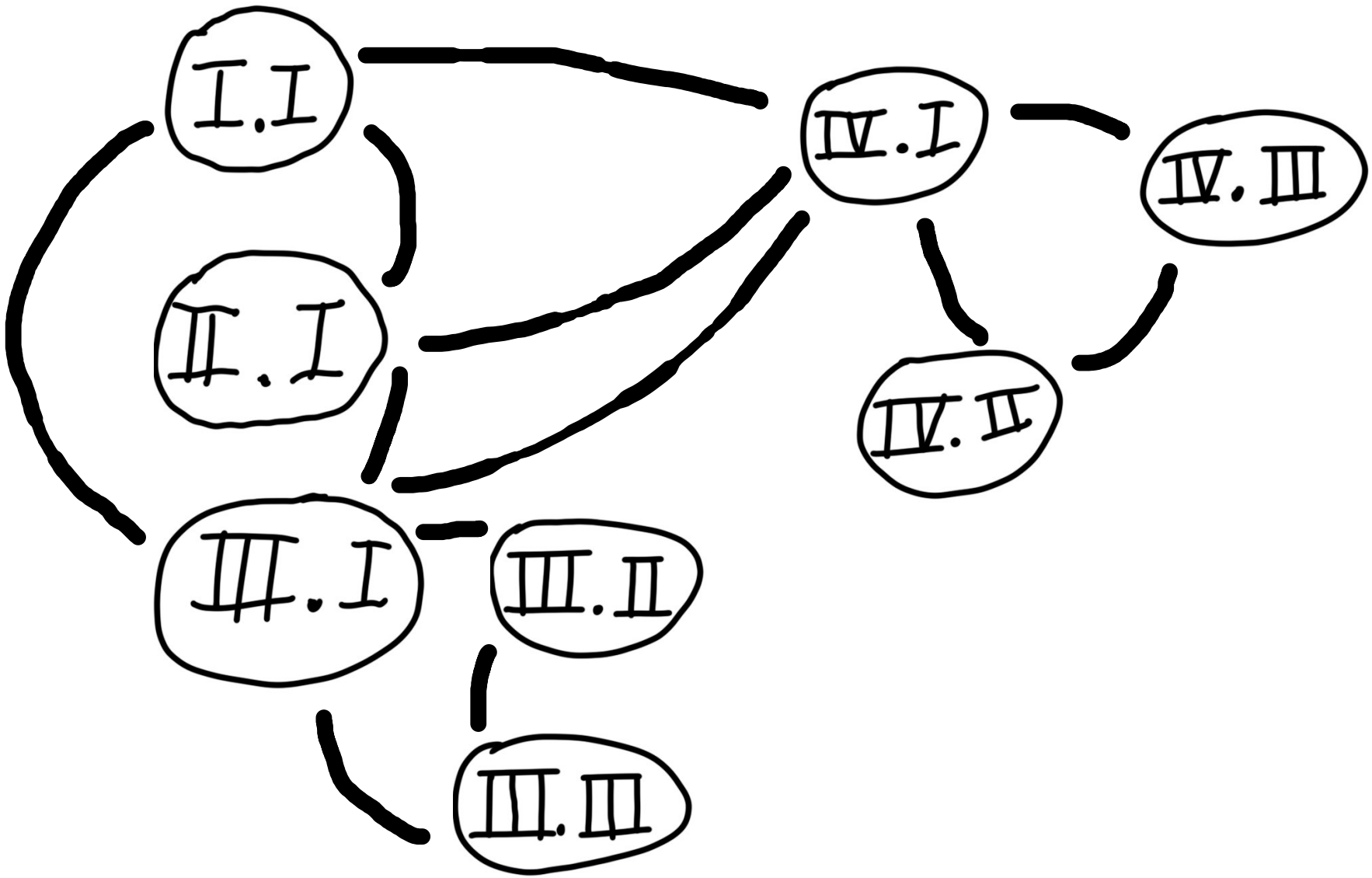
IV. EMERGENCY SERVICES
    1. FIRE
    II. MEDICAL
    III. HAZARDOUS MAGIC

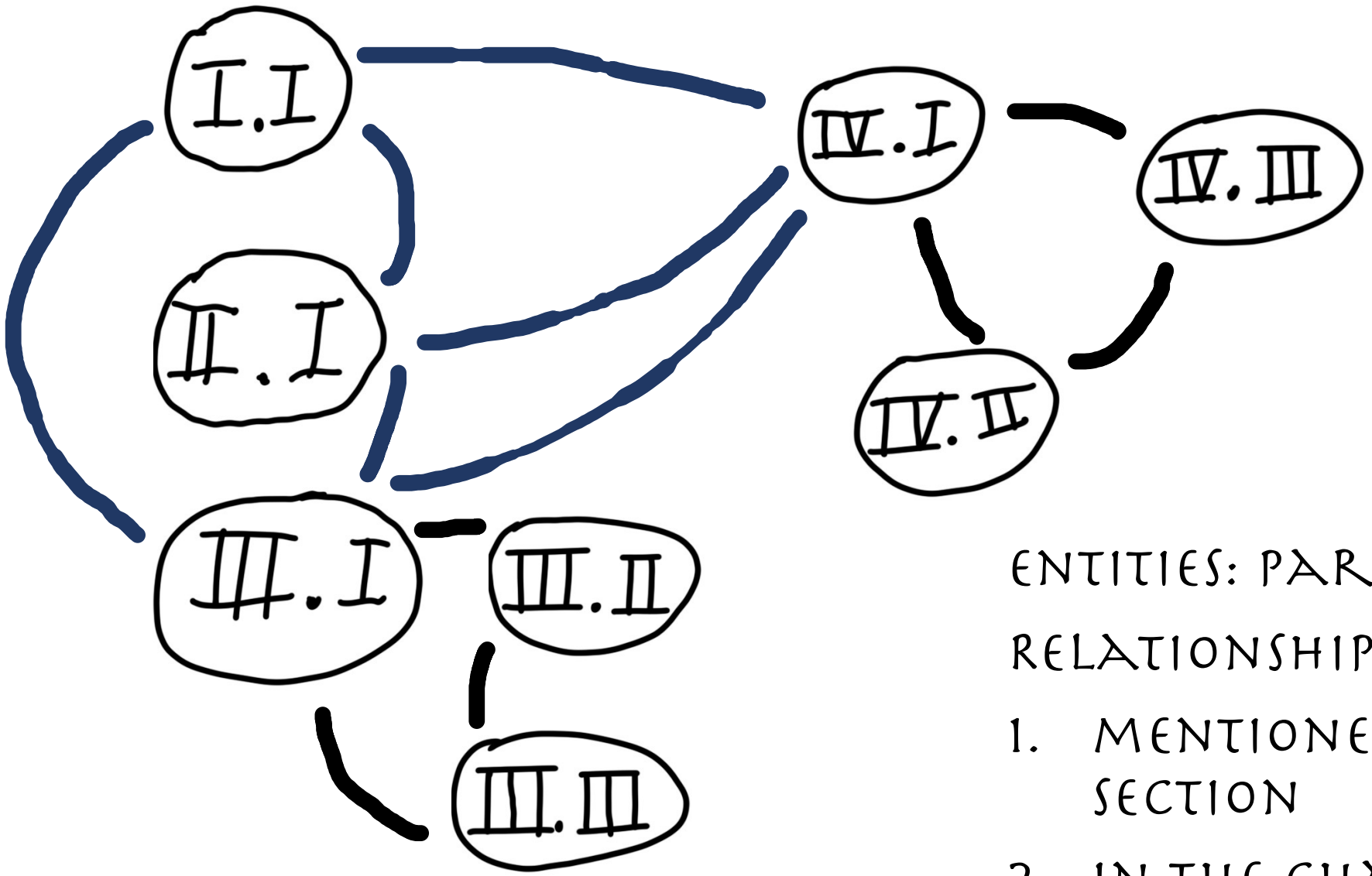
COMMUNITY DETECTION

- 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.III
IV.II

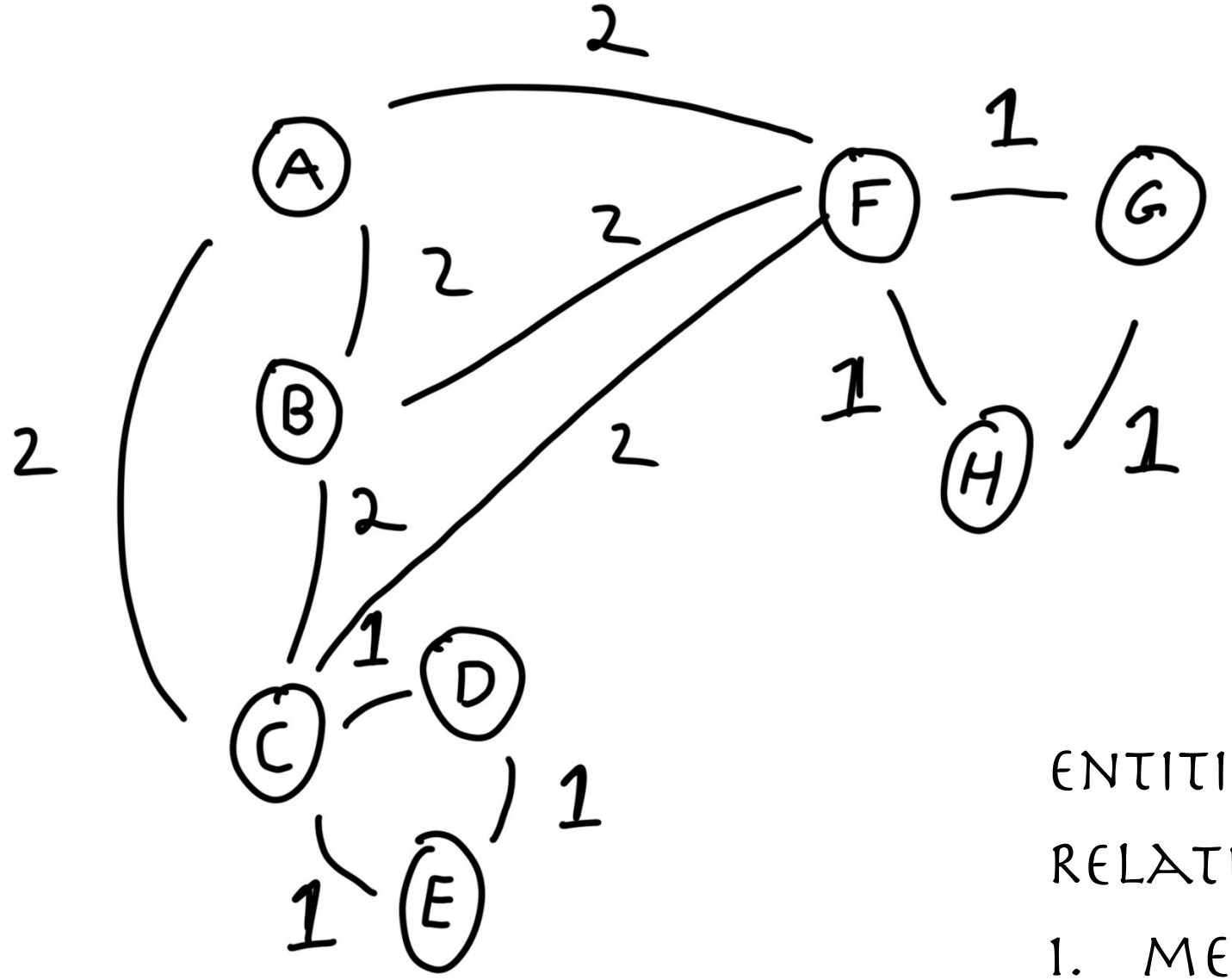

I.I
II.I
III.I
III.II
III.III
IV.I
IV.II
IV.III



ENTITIES: PARAGRAPH

RELATIONSHIPS:

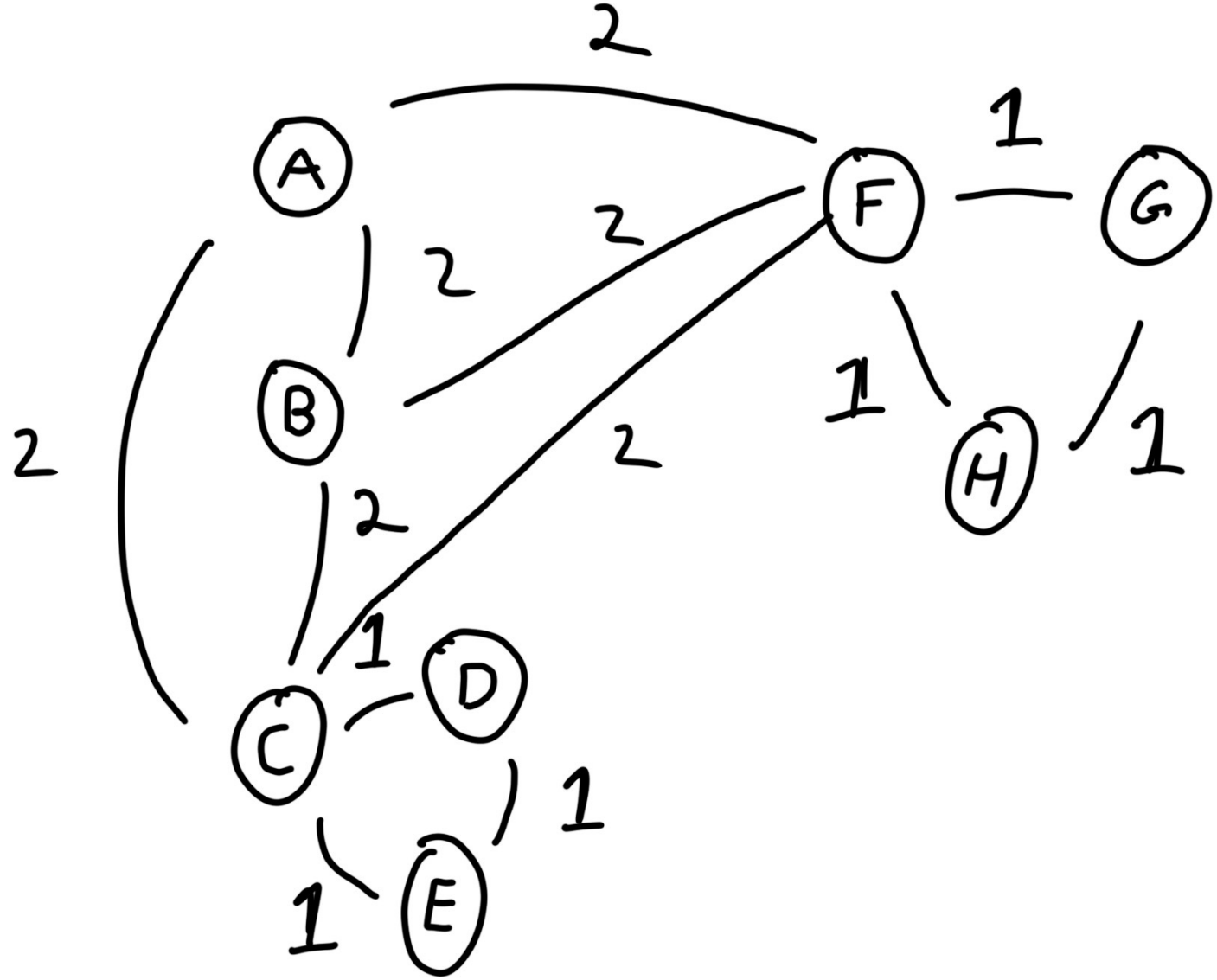
1. MENTIONED IN THE SAME SECTION
2. IN THE CHARTER



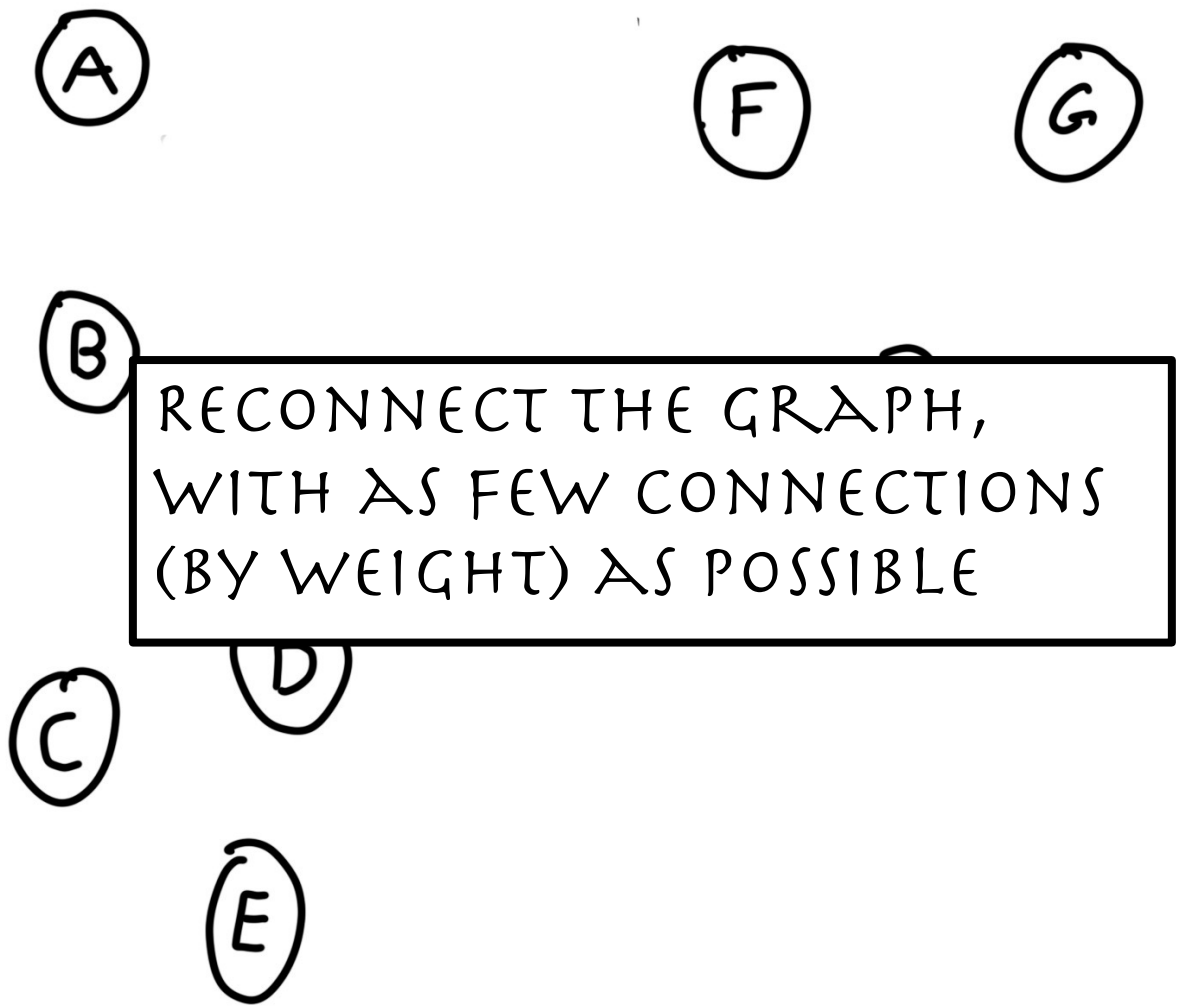
ENTITIES: PARAGRAPH

RELATIONSHIPS:

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

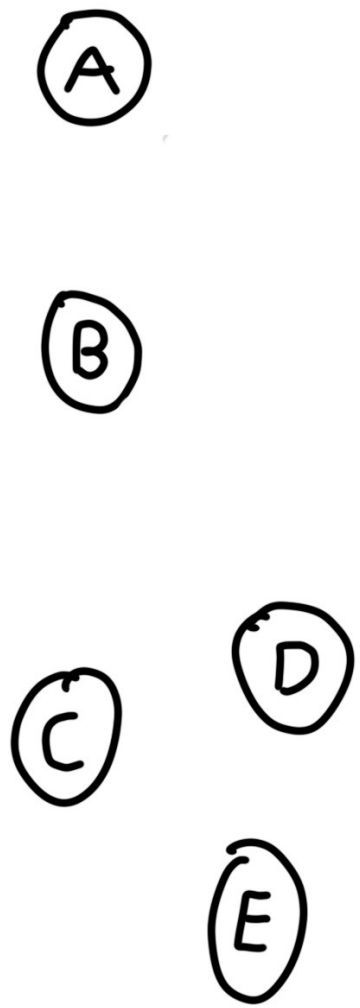


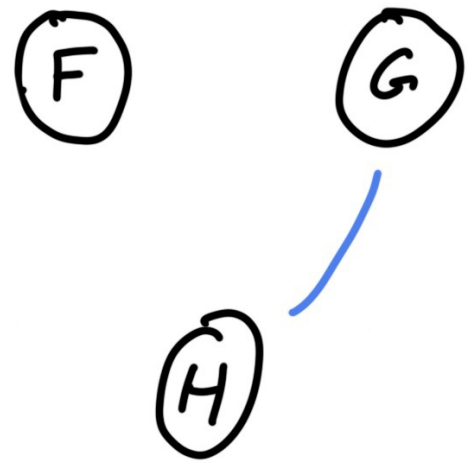
A	B	2
A	C	2
A	F	2
B	C	2
B	F	2
C	F	2
C	D	1
C	E	1
D	E	1
F	G	1
F	H	1
G	H	1



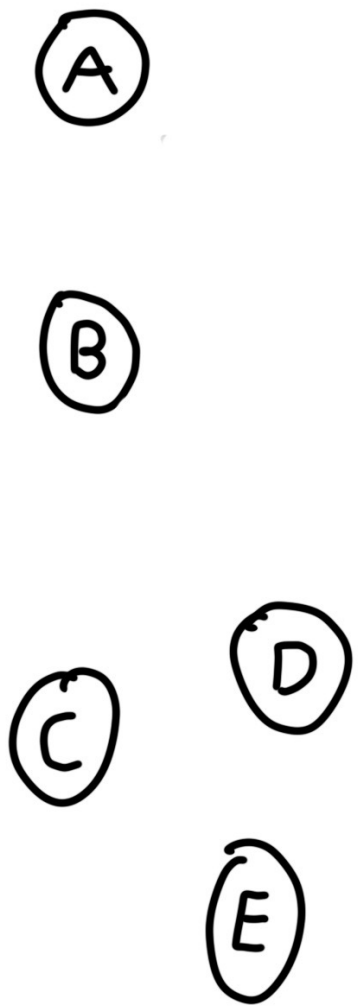
RECONNECT THE GRAPH,
WITH AS FEW CONNECTIONS
(BY WEIGHT) AS POSSIBLE

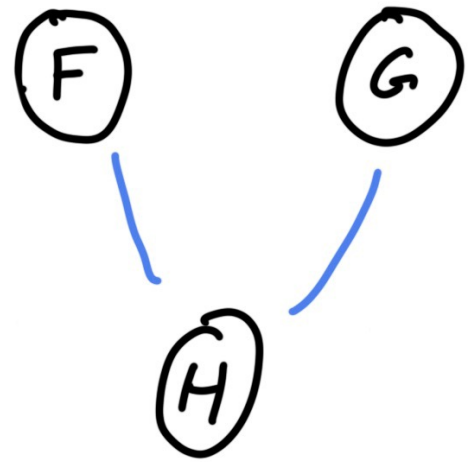
AB	2
AC	2
AF	2
BC	2
BF	2
CF	2
CD	1
CE	1
DE	1
FG	1
FH	1
GH	1



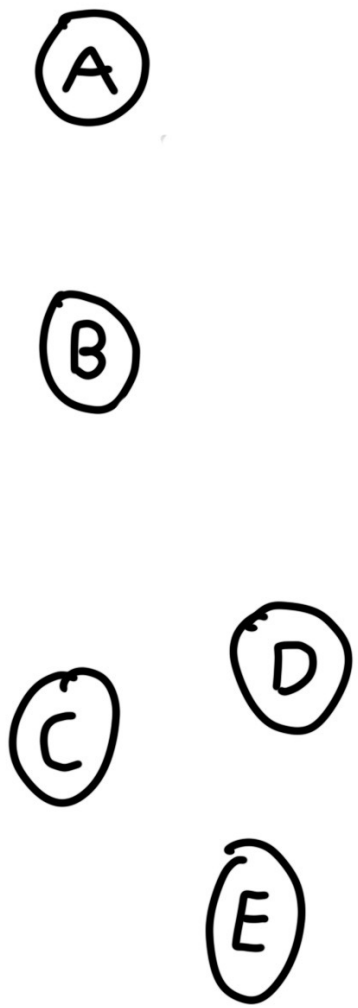


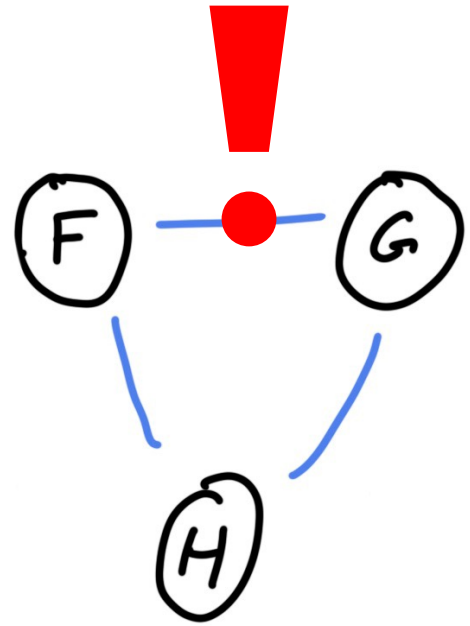
AB	2
AC	2
AF	2
BC	2
BF	2
CF	2
CD	1
CE	1
DE	1
FG	1
FH	1
~~GH~~	1



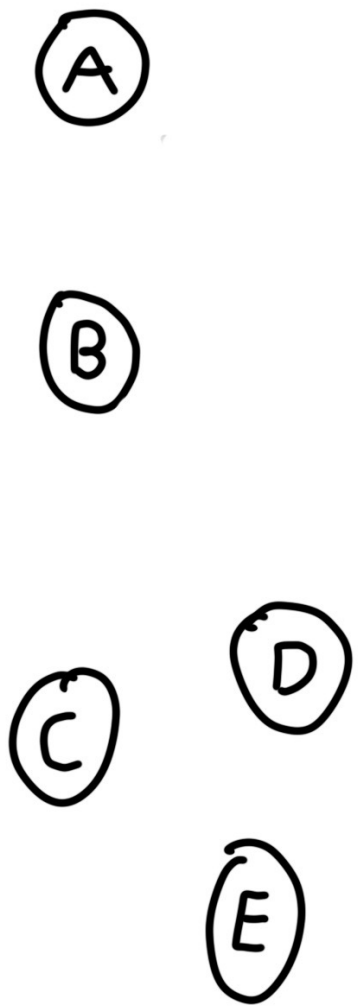


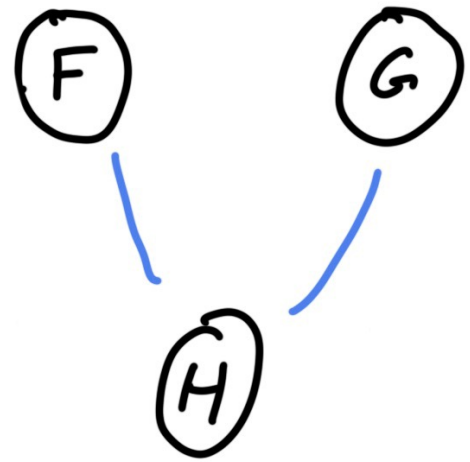
AB	2
AC	2
AF	2
BC	2
BF	2
CF	2
CD	1
CE	1
DE	1
FG	1
~~FH~~	~~1~~
~~GH~~	~~1~~



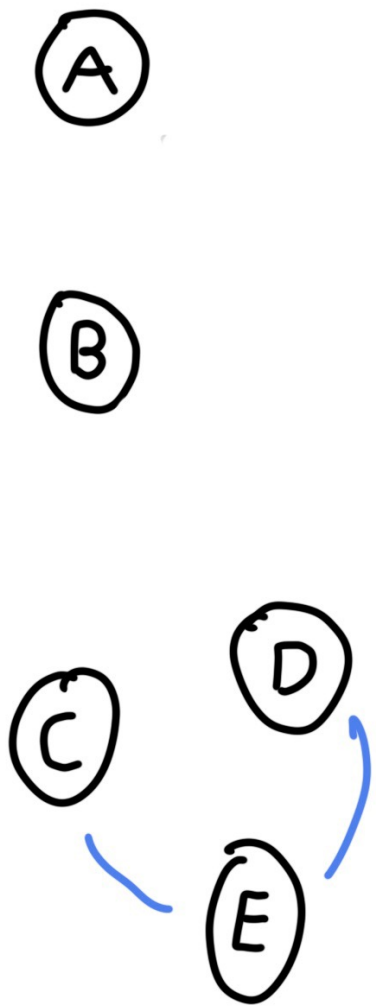


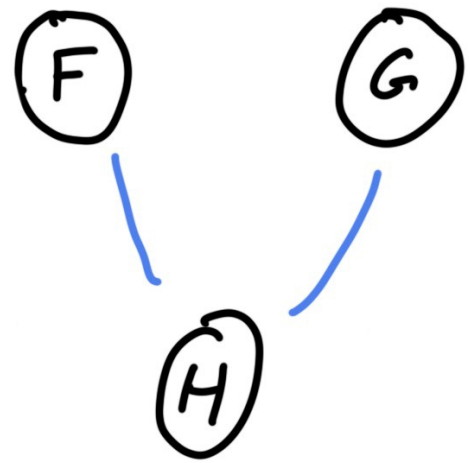
AB	2
AC	2
AF	2
BC	2
BF	2
CF	2
CD	1
CE	1
DE	1
~~FG~~	~~1~~
~~FH~~	~~1~~
~~GH~~	~~1~~



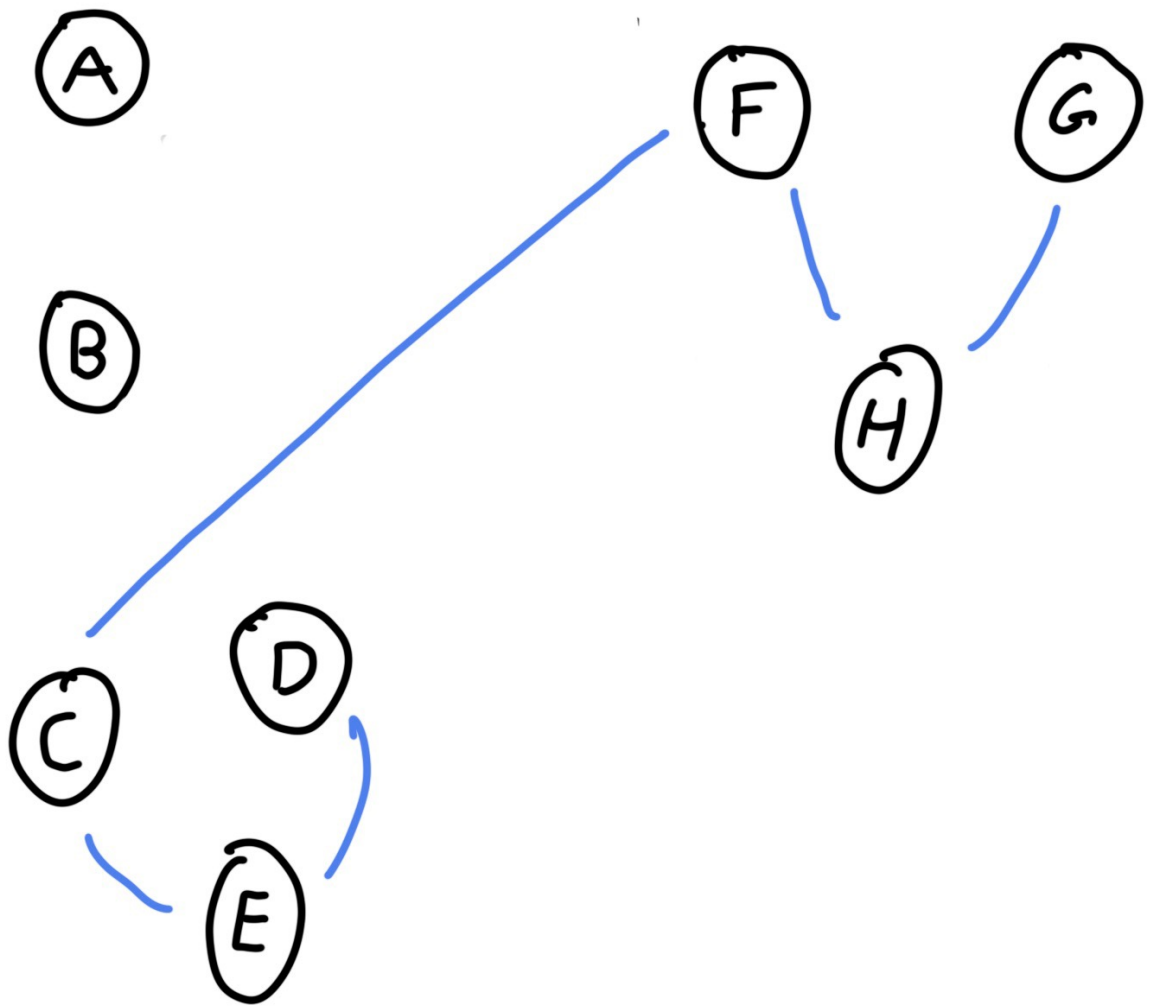


A	B	2
A	C	2
A	F	2
B	C	2
B	F	2
C	F	2
C	D	1
C	E	1
D	E	1
F	G	1
~~F~~	~~H~~	~~1~~
~~G~~	~~H~~	~~1~~

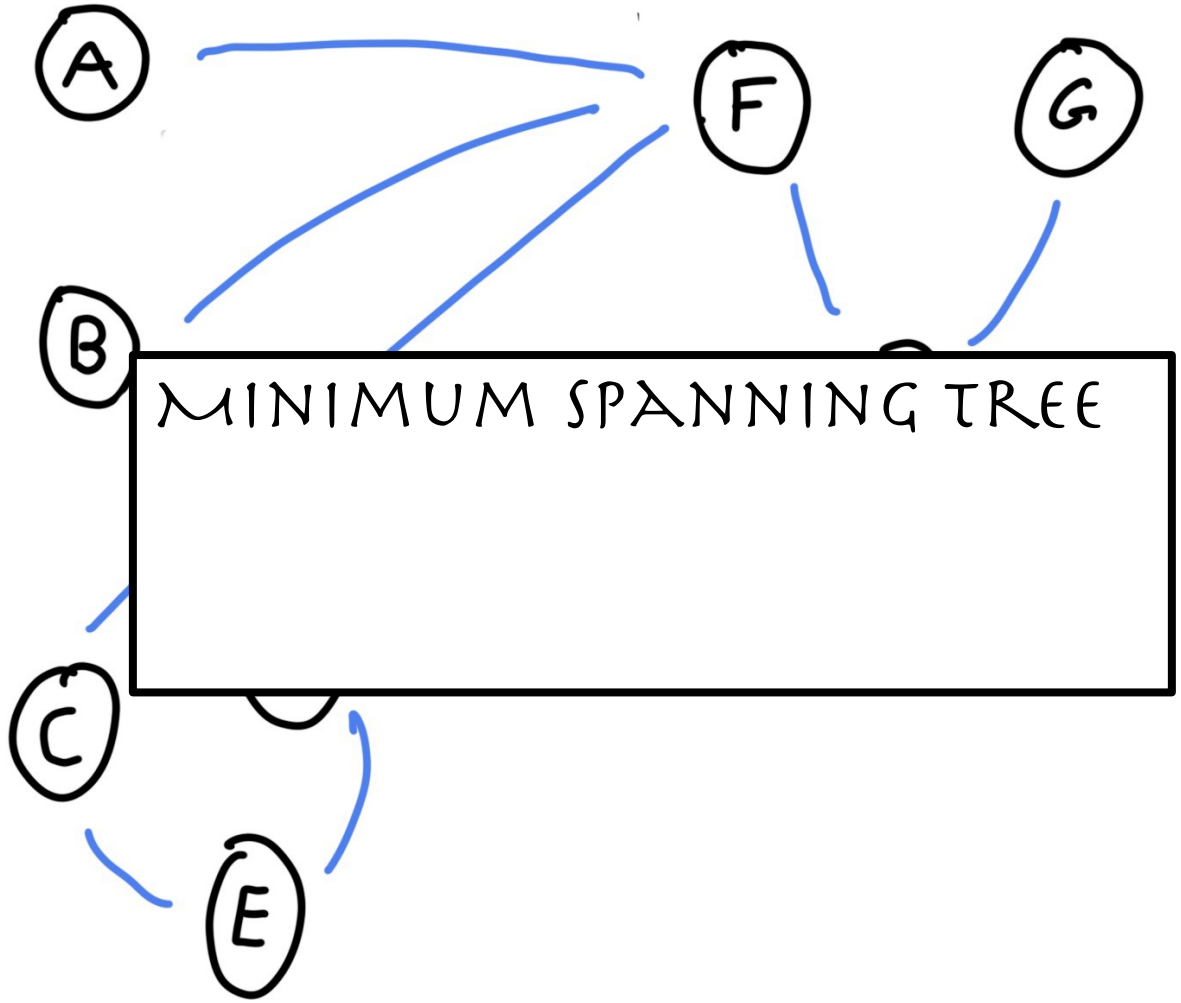




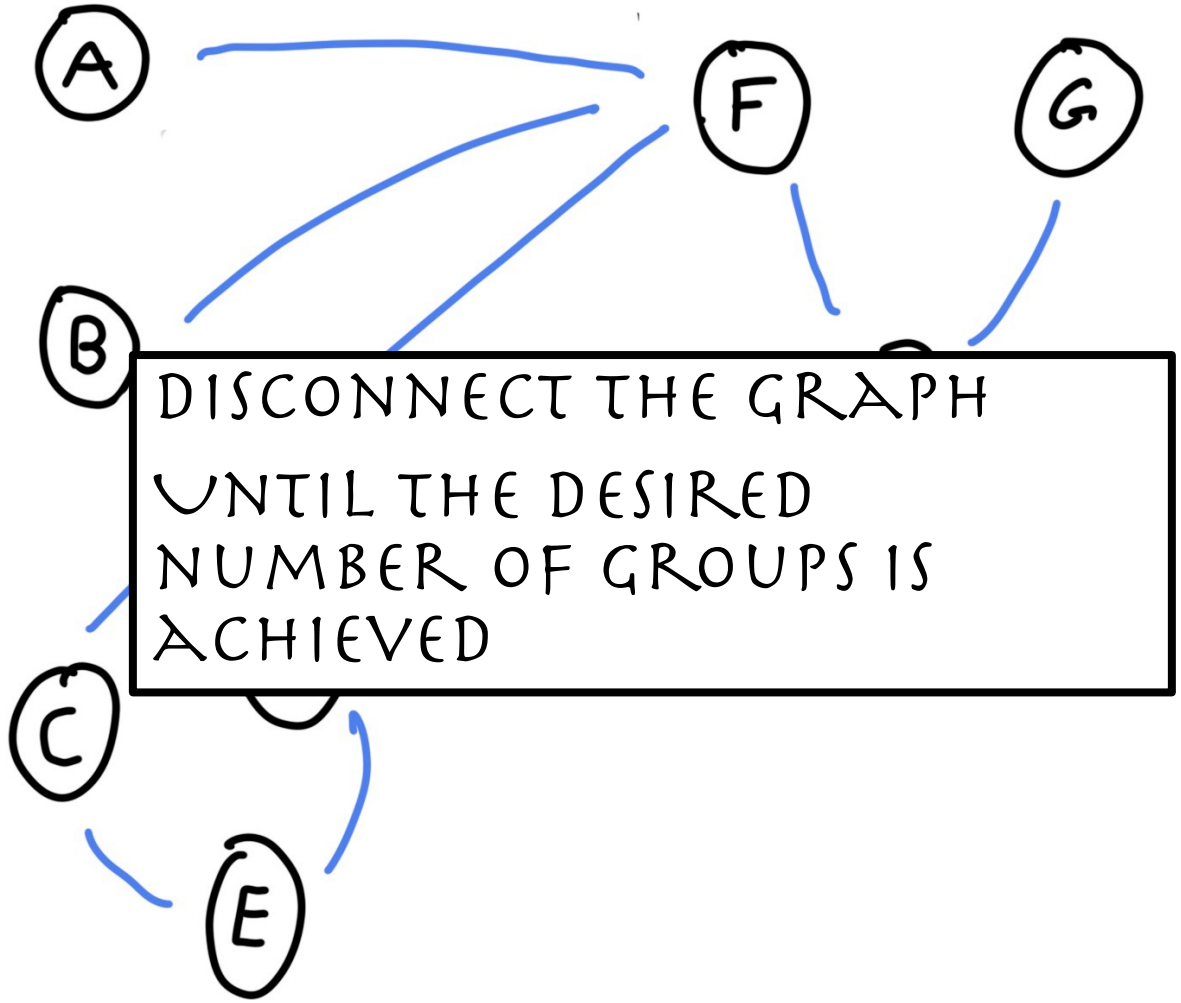
AB 2
AC 2
AF 2
BC 2
BF 2
CF 2
CD 1
CE 1
DE 1
FG 1
FH 1
GH 1



A B	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~~
~~G H~~	~~1~~



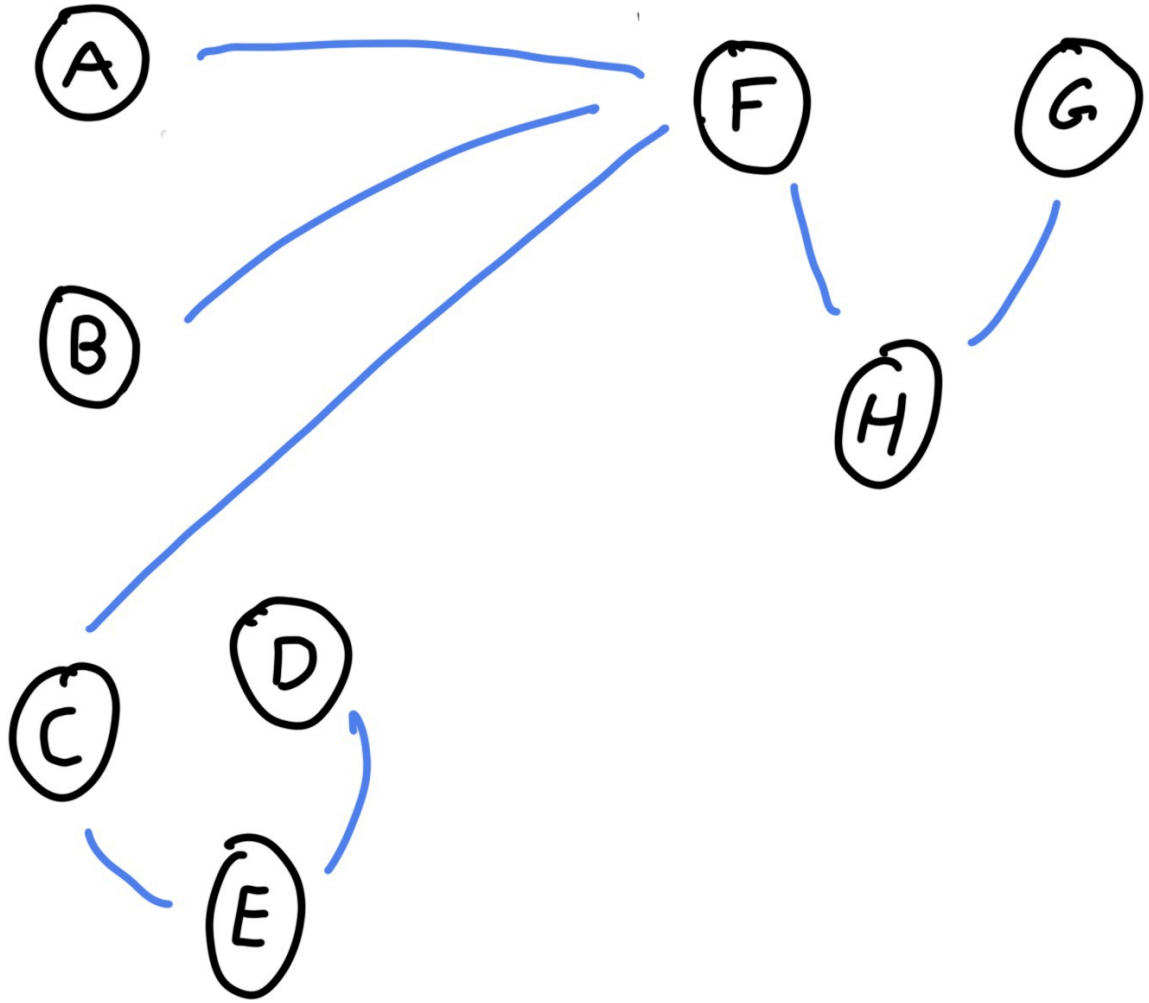
AB 2
AC 2
AF 2
BC 2
BF 2
CF 2
CD 1
CE 1
DE 1
FG 1
FH 1
GH 1



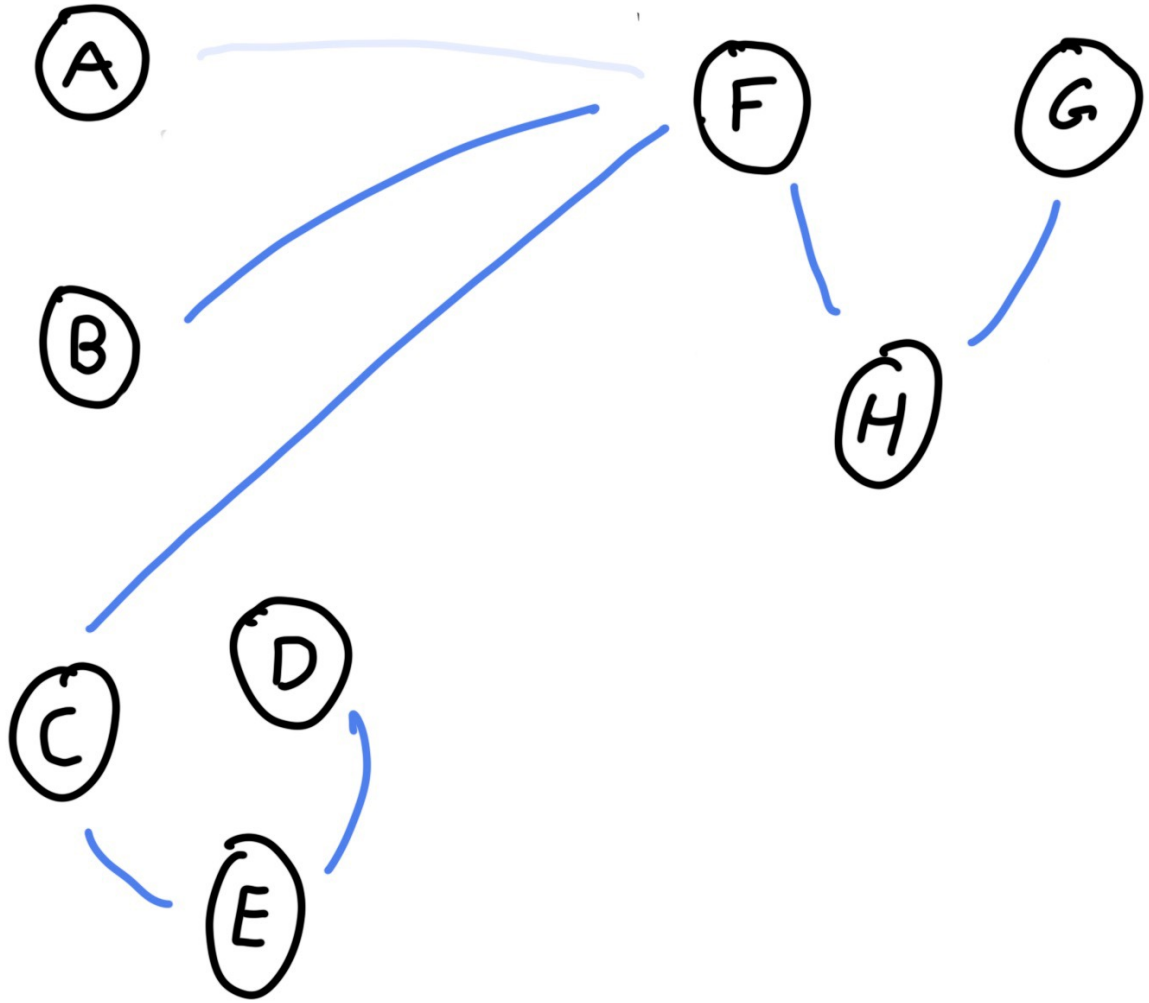
DISCONNECT THE GRAPH

UNTIL THE DESIRED NUMBER OF GROUPS IS ACHIEVED

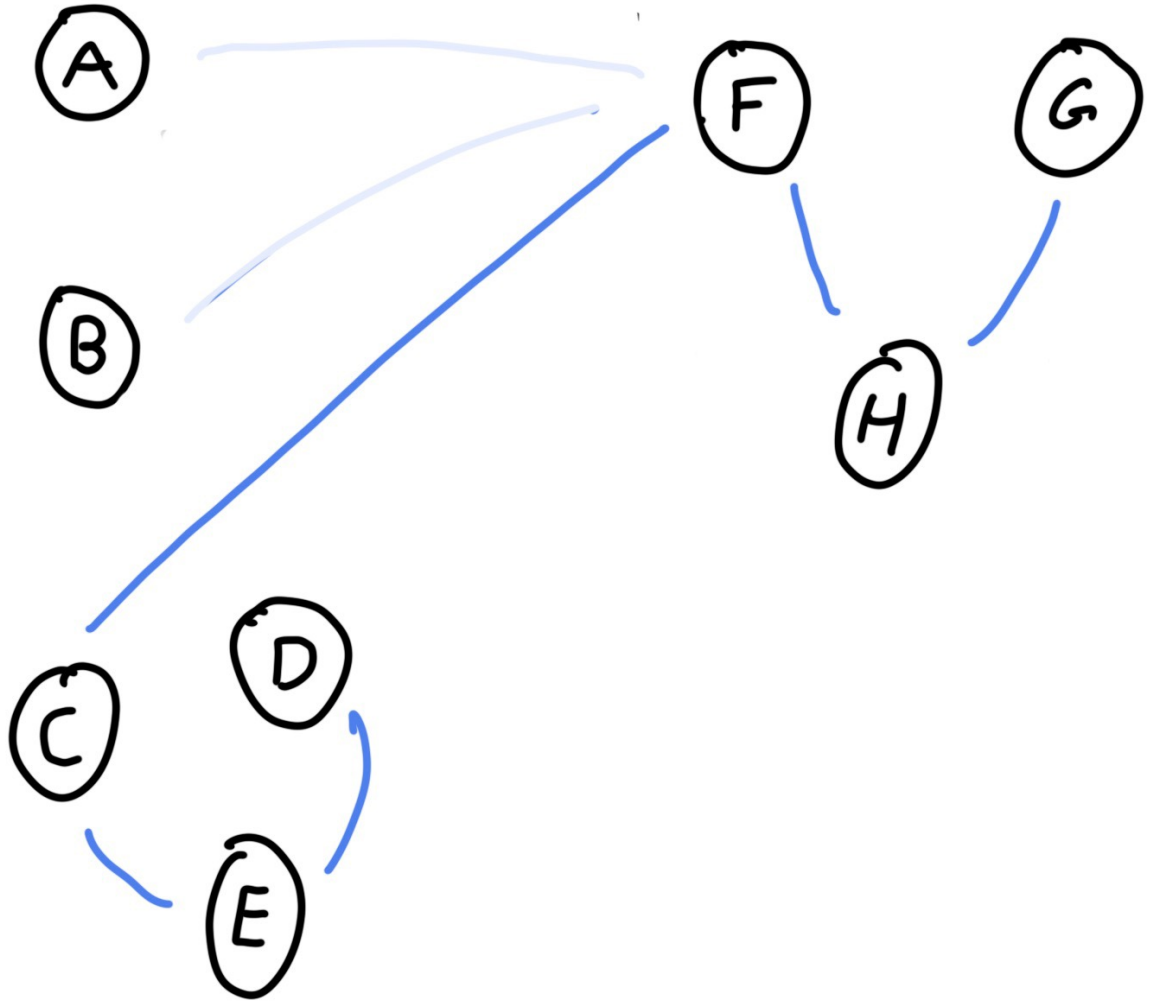
✗	AB	2
✗	AC	2
	AF	2
✗	BC	2
	BF	2
	CF	2
✗	CD	1
	CE	1
	DE	1
✗	FG	1
	FH	1
	GH	1



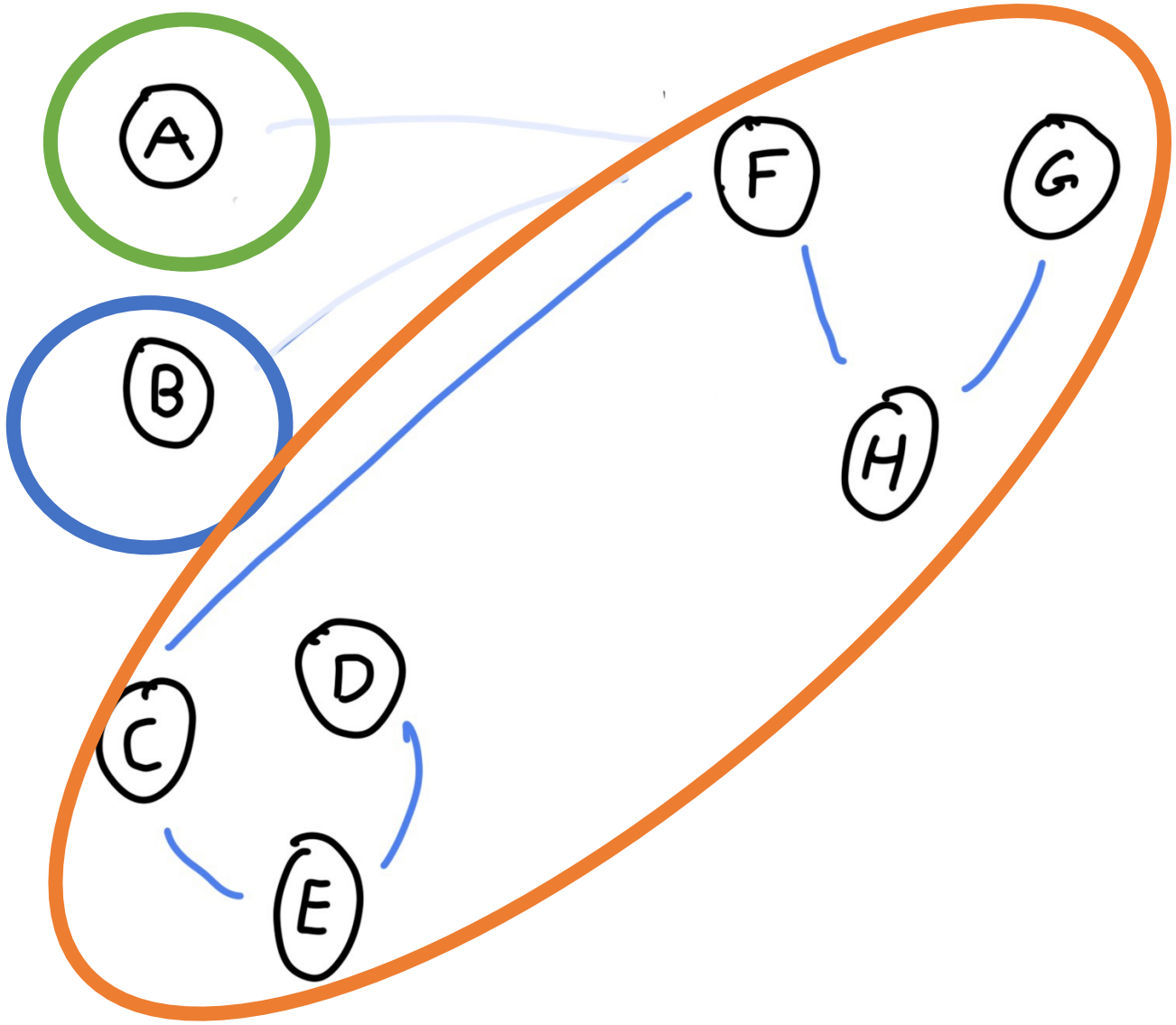
✗	AB	2
✗	AC	2
	~~AF~~	~~2~~
✗	BC	2
	~~BF~~	~~2~~
	~~CF~~	~~2~~
✗	CD	1
	~~CE~~	~~1~~
	~~DE~~	~~1~~
✗	FG	1
	~~FH~~	~~1~~
	~~GH~~	~~1~~



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



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



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

Town Charter

1. Legislative Department
    1. Mayor
2. Administrative Department
    1. Mayor's Aide
3. Public Works
    1. Water Well Maintenance
    2. Roads
    3. Animal Control
4. Emergency Services
    1. Fire
    2. Medical
    3. 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/Djikstra
- Connections
  - Centrality (degree, eigenvector, closeness, betweenness)
- Communities
  - Prim's
  - Louvain, K-L partition,

Mohammad Athar https://linktr.ee/mohammadAthar