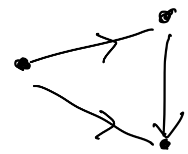
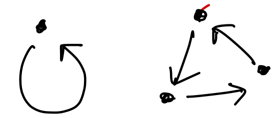
Directed Acyclic Graphs: DAG

No cycles!!



✓ okay.



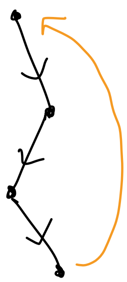
X X. no!!

Theorem: G is a DAG $\iff$ a DFS has no back edges

$\iff$ all DFS traversals have no back edges

Proof: ($\Longrightarrow$)

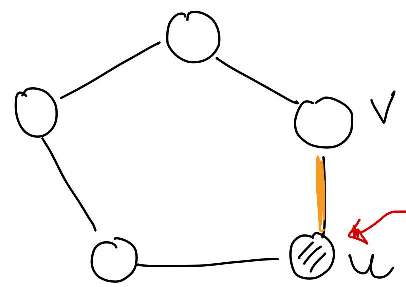
Assume DFS has back edge



← clearly there is a cycle!!

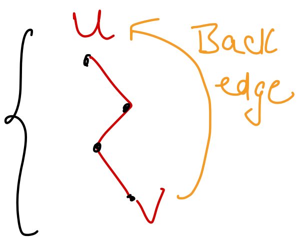
($\Longleftarrow$)

Assume G has a cycle.



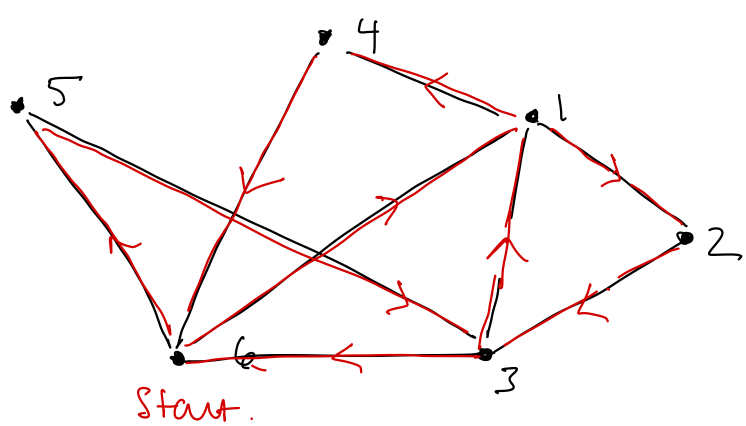
Assume DFS hits vertex u first in the cycle.

White path theorem!
v is a descendent of u.



Euler Tour:

A *tour* of the edges of G: you must visit each edge exactly once and come back to where you started.



Order of tour:

6, 1, 2, 3, 1, 4, 6, 5, 3, 6.

How do you know if you have an Euler tour?

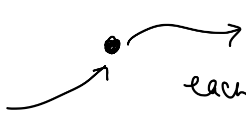
Undirected graph:

E.T. exists $\iff$ all degrees are <u>even</u>.

Directed graph:

E.T. exists $\iff$ in degree = outdegree for each vertex.

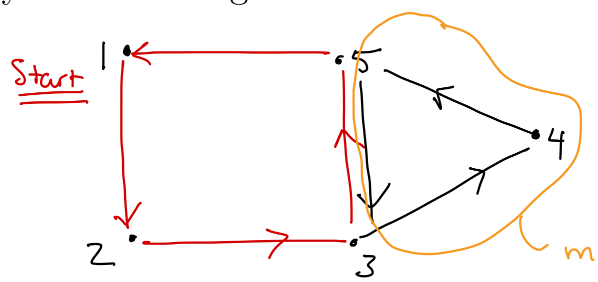
How does this work?
If there is an Euler tour then...



each time the tour comes in, it leaves... "using up" two vertices...

If the above is satisfied (indeg =outdeg or all degrees even), then how do we find a Euler Tour?

- Can we simply start walking and find the tour?

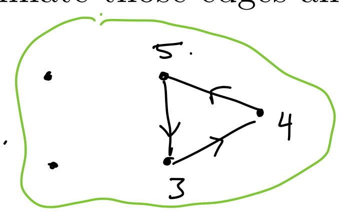


- Suppose we start "following" edges...
- We could follow 1-2-3-5-1
- Now we are 'stuck", even though we didn't complete the tour.

- If we find a cycle (instead of the whole tour), we can eliminate those edges and continue:

- From above, we found 1-2-3-5-1

⟹ Update G to ....



⟹ From vertex 3, find cycle 3-4-5-3.

⟹ "Splice" this into first cycle.

⟹ Final tour: 1-2-3-4-5-3-5-1

The Euler WALK algorithm:

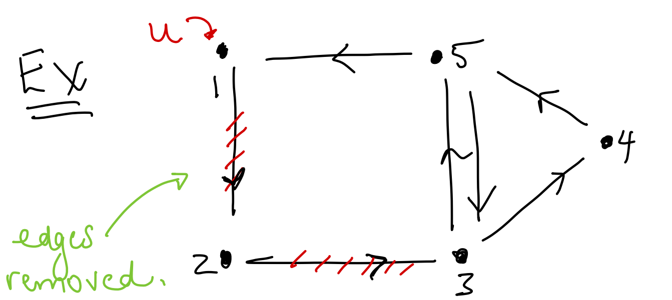
- Starts at a vertex u.
- Returns a *cycle* from u to u stored in queue Q. ← Stores the resulting cycle in Q.

EULERWALK(u):

```
makenull(Q)
Enqueue(u, Q)
v = u
Repeat:
    v = delete(NN(v))
    Enqueue(v, Q)
until v = u.
Return Q.
```

the edge is removed from G. Note this alters the graph.

Ex



- Q = {1}
- Q = {1, 2}
- Q = {1, 2, 3}
- Q = {1, 2, 3, 5} etc...

Snapshot at this step.

The Euler Tour algorithm:

- Will use a Stack to keep track of the cycles (found by Euler walk).
- Will output the vertices of the final tour

Algorithm EulerTour

```
makenull(S).
Push(1, S)
while (|S| > 0)
    v ← pop(S)
    if NN(v) = ∅ , output v.
    else
        push( Eulerwalk(v), S ).
```

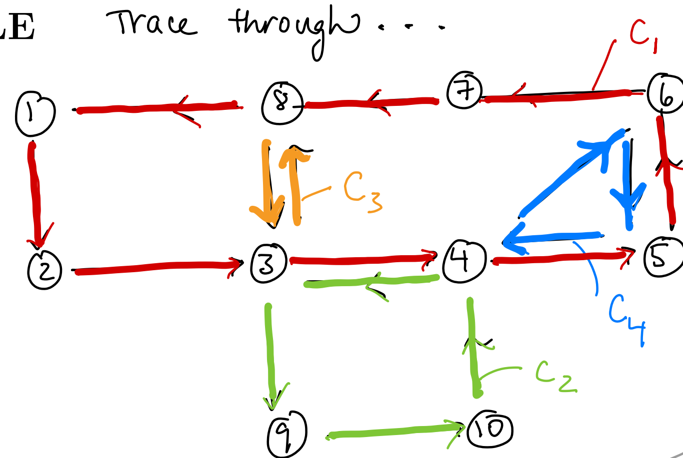
the entire Eulerwalk is pushed onto the Stack.

Complexity: $O(|V| + |E|)$

Each vertex is examined.

Each edge is examined.

EXAMPLE Trace through . . .



- Starts at Vertex 1
- assume it finds cycle C1 in the Eulerwalk.

  S = 1, 2, 3, 4, 5, 6, 7, 8, 1.
- Pops of 1, 2, 3
- Finds cycle 3, 9, 10, 4, 3 and pushes it on Stack.

  S = 3, 9, 10, 4, 3, 4, 5, 6, 7, 8, 1.
- Finds cycle 3, 8, 3 , added to stack.

  S = 3, 8, 3, 9, 10, 4, 3, 4, 5, 6, 7, 8, 1.
- Pops off 3, 8, 3, 9, 10, 4
- Finds cycle 4, 6, 5, 4.

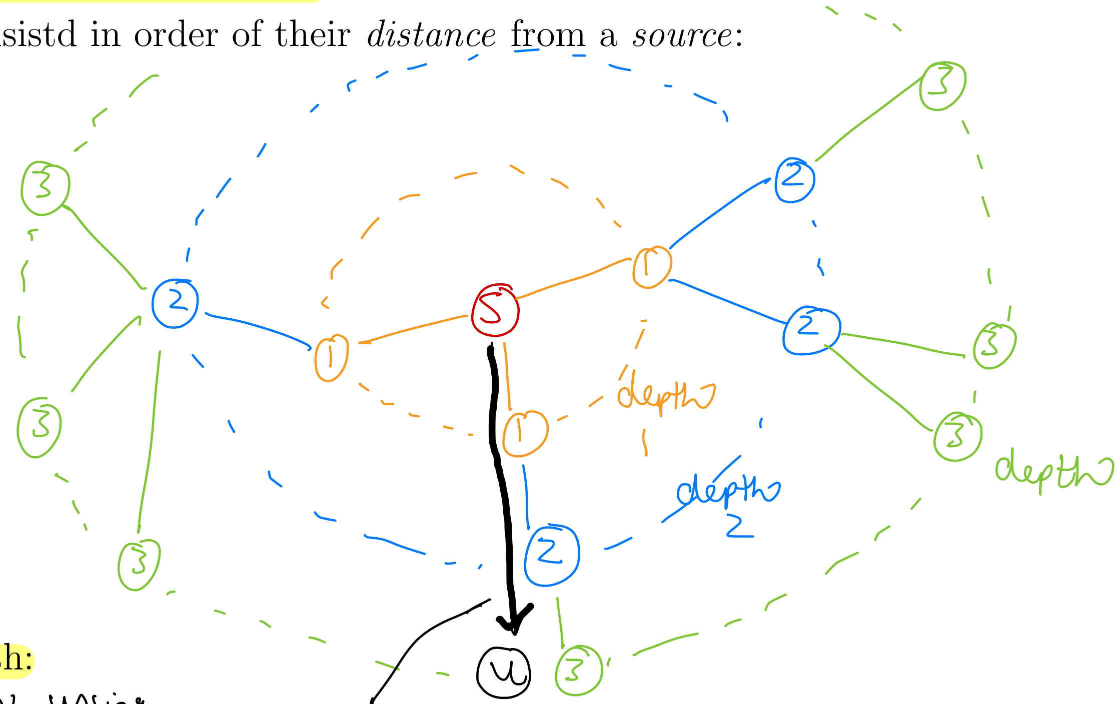
  S = 4, 6, 5, 4, 3, 4, 5, 6, 7, 8, 1
- Pops off all remaining.

OUTPUT:

1, 2, 3, 8, 3, 9, 10, 4, 6, 5, 4, 3, 4,
5, 6, 7, 8, 1

Breadth First Search: BFS

Nodes are visistd in order of their *distance* from a *source*:



Grows outward based on distance from s.

Shortest Path:

s → u path using min # edges

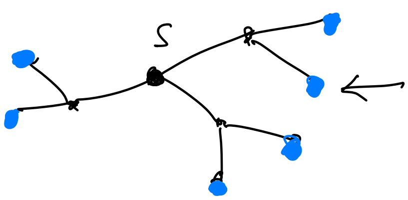
Behind the algorithm:

- As with DFS will use: d[v] : distance (# edges) from s → v.

  p[v] : parent pointer.

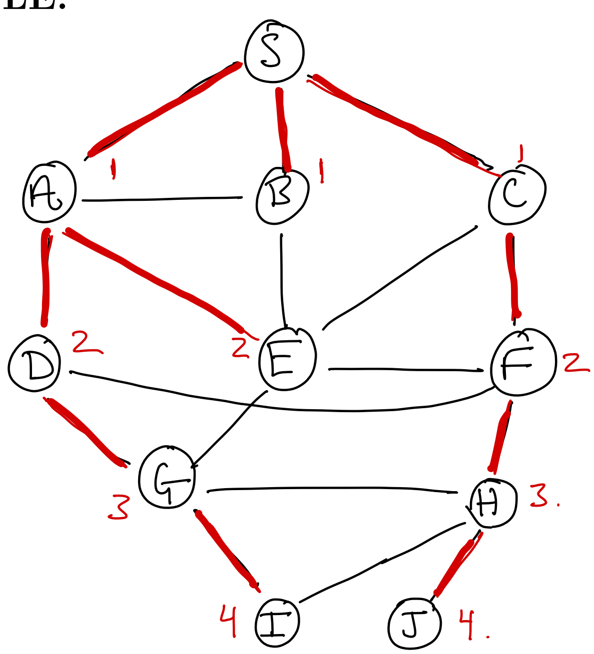
  colour[v] : white / grey / black.

- A Queue to keep track of the *fridge* elements as the tree grows...



← Queue stores "outer" vertices

EXAMPLE:



- Queue: Q = s.
- Reach A, B, C at depth 1.

  Q = ~~A~~, B, C.
- Reach D, E from A

  Q = B, C, D, E
- Reach F from C

  Q = D, E, F.
- Reach G from D.

  Q = E, F, G
- Reach H from F.

  Q = G, H. etc...

ALGORITHM: BFS(s) ← source.

Complexity $O(|V|)$.

$\forall v$ set :

colour[v] = white

d[v] = ∞

p[v] = nil.

colour[s] = grey

d[s] = 0

p[s] = nil

makenull(Q)

Enqueue(s, Q).

Initialize...

Q = s, d(s) = 0.

executed $O(|E|)$ times.

While |Q| > 0 do :

u = dequeue(Q)

$\forall$ v adjacent to u do :

if colour[v] = white,

colour[v] = grey

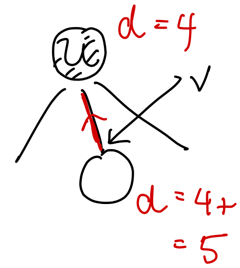
d[v] = d[u] + 1

p[v] = u

Enqueue(v, Q).

colour[u] = black.

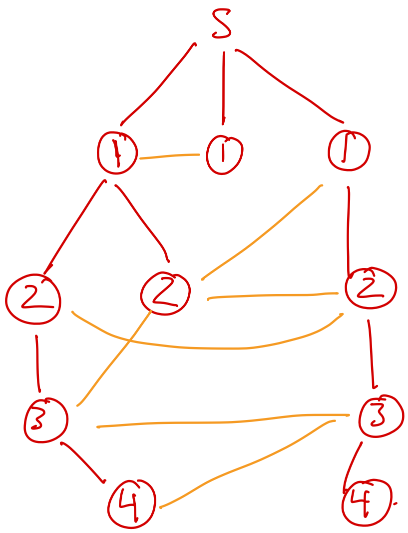
Constant time per vertex, executed $O(|V|)$ times. $\therefore$ total $O(|V|)$



Complexity: From above :

$O(|V| + |E|)$ like DFS...

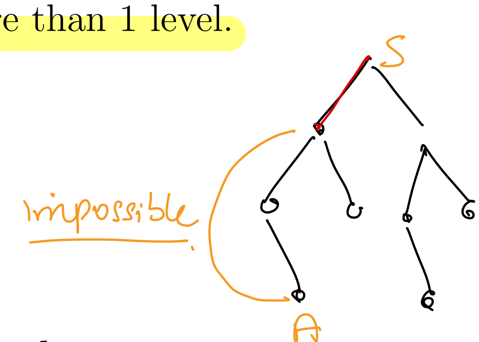
The result is a BFS tree consisting of tree edges:



red: tree edges of BFS tree.

yellow: unused graph edges.

The *other* edges in the graph that are not part of the tree are also shown above. Note that they cannot jump more than 1 level.

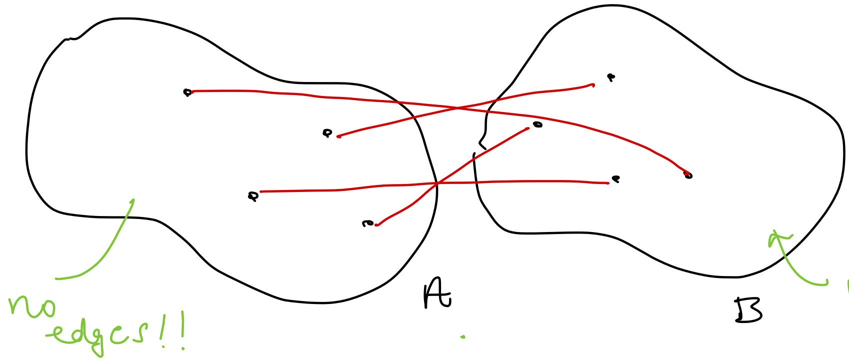


Why? Vertex A is at depth 3.

If the yellow edge existed, then vertex A should have been placed at depth 2.

Bipartite Graphs:

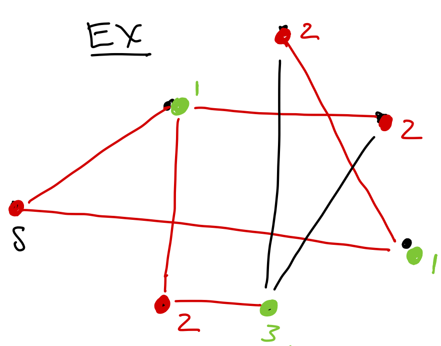
$G = (V, E)$ There are disjoint sets A,B such that: $V = A \cup B$, $E \subseteq A \times B$.

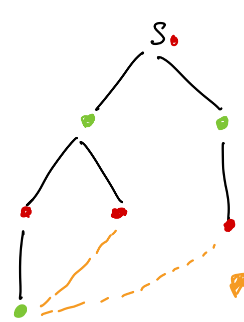


All edges in G go from A − B.

no edges from B − B.

Exercise: Determine if G is bipartite (in $O(|V| + |E|)$). USE BFS.



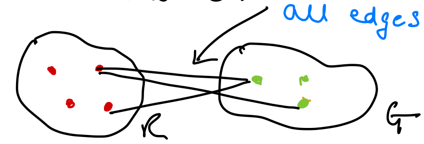


↳ then colour the vertices on different levels alternating colours!

↳ then check that you don't have any "other" edges from R→R or G→R

↳ Solution:

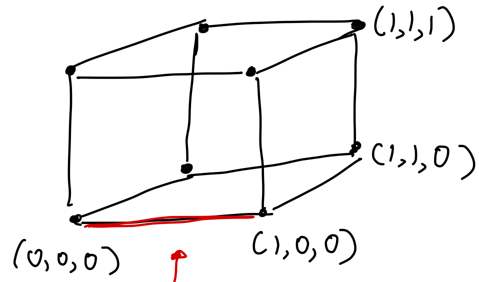
all edges R → G.



The Hypercube: H_d

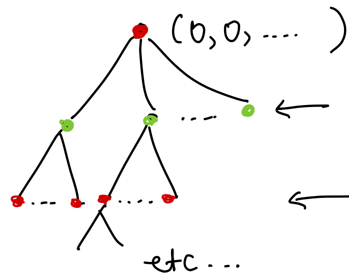
- $V = \{0,1\}^d$. Ex. if $d = 3$, $V = (0,0,0), (1,0,0), (0,1,0), (0,0,1)$ $(1,1,0), (1,0,1), (0,1,1), (1,1,1)$.

- E: Two edges are connected if their vertices differ by only 1 digit.



Exercise: Show that H_d is bipartite.

→ you can similarly run BFS on the hypercube:



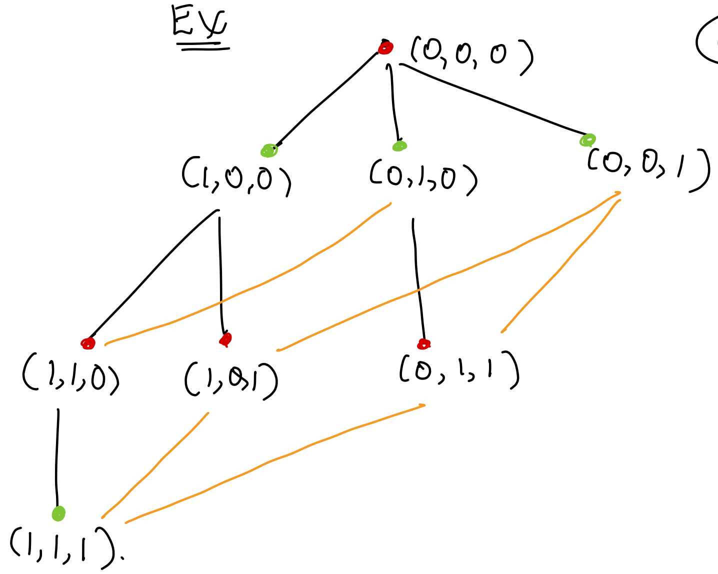
← d children. (all with one '1')

← all have two 1's

No edges in the hypercube from vertices with the same number of 1's.

Ex

$d = 3$.



Bipartite using Red/Green.