Analysis of Algorithms, I

CSOR W4231

Eleni Drinea
Computer Science Department

Columbia University

Depth-first search, topological sorting

Outline

Today

1. Recap
2. Applications of BFS
   - Testing bipartiteness
3. Depth-first search (DFS)
4. Applications of DFS
   - Cycle detection
   - Topological sorting

Review of the last lecture

- Graphs (directed, undirected, weighted, unweighted)
  - Notation: $G = (V, E)$, $|V| = n$, $|E| = m$
- Representing graphs
  1. Adjacency matrix
  2. Adjacency list
- Trees, bipartite graphs, the degree theorem
- Linear graph algorithms
- Breadth-first search (BFS)

Claim 1.

Let T be a BFS tree, let x and y be nodes in T belonging to layers L_i and L_j respectively, and let (x, y) be an edge in G. Then i and j differ by at most 1.

An algorithm for s-t connectivity: breadth-first search

Breadth-first search (`BFS`(G, s)): explore G starting from s **outward in all possible directions**, adding reachable nodes one **layer** at a time.

- First add all nodes that are joined by an edge to s: these nodes form the first layer.
  If G is unweighted, these are the nodes at distance 1 from s.
- Then add all nodes that are joined by an edge to a node in the first layer: these nodes form the second layer.
  If G is unweighted, these are the nodes at distance 2 from s.
- And so on and so forth.

Today

1. Recap
2. Applications of BFS
   - Testing bipartiteness
3. Depth-first search (DFS)
4. Applications of DFS
   - Cycle detection
   - Topological sorting

Testing bipartiteness & graph 2-colorability

Testing bipartiteness

- **Input:** a graph $G = (V, E)$
- **Output:** **yes** if G is bipartite, **no** otherwise

Equivalent problem (*why?*)

- **Input:** a graph $G = (V, E)$
- **Output:** **yes** if and only if we can color all the vertices in G using at most 2 colors –say red and white– so that no edge has two endpoints with the same color.

Why wouldn't we be able to 2-color a graph?

Fact: If a graph contains an odd-length cycle, then it is not 2-colorable.

So a **necessary** condition for a graph to be 2-colorable is that it does not contain odd-length cycles.

Is this condition also **sufficient**, *that is, if a graph does not contain odd-length cycles, then is it* 2*-colorable?*

In other words, are odd cycles the only obstacle to bipartiteness?

Algorithm for 2-colorability

BFS provides a natural way to 2-color a graph $G = (V, E)$:

- Start BFS from any vertex; color it red.
- Color white all nodes in the first layer L_1 of the BFS tree. If there is an edge between two nodes in L_1, output **no** and stop.
- Otherwise, continue from layer L_1, coloring red the vertices in even layers and white in odd layers.
- If BFS terminates and all nodes in V have been explored (hence 2-colored), output **yes**.

Analyzing the algorithm

Upon termination of the algorithm

- either we successfully 2-colored all vertices and output **yes**, that is, declared the graph bipartite;
- or we stopped at some level because there was an edge between two vertices of that level and output **no**; in this case, we declared the graph non-bipartite.

This algorithm is **efficient**. *Is it a* **correct** *algorithm for* 2*-colorability?*

Showing correctness

To prove correctness, we must show the following statement.

If our algorithm outputs

1. **yes**, then the 2-coloring it returns is a valid 2-coloring of G;
2. **no**, then indeed G cannot be 2-colored by **any** algorithm (e.g., because it contains an odd-length cycle).

The next claim proves that this is indeed the case by examining the possible outputs of our algorithm. Note that the output depends solely on whether *there is an edge in G between two nodes in the same BFS layer*.

Correctness of algorithm for 2-colorability

Claim 2.

Let G be a connected graph, and let L_1, L_2, ... be the layers produced by BFS starting at node s. Then exactly one of the following is true.

1. *There is no edge in G joining two nodes in the same BFS layer. Then G is bipartite and has no odd length cycles.*

2. *There is an edge in G joining two nodes in the same BFS layer. Then G contains an odd length cycle, hence is not bipartite.*

Corollary 1.

A graph is bipartite if and only if it contains no odd length cycle.

Proof of Claim 2, part 1

1. **Assume** that no edge in G joins two nodes of the same layer of the BFS tree.

By Claim 1, all edges in G not belonging to the BFS tree are

- either edges between nodes in the same layer;
- or edges between nodes in adjacent layers.

Our assumption implies that all edges of G not appearing in the BFS tree are between nodes in adjacent layers.

Since our coloring procedure gives such nodes different colors, the whole graph can be 2-colored, hence it is bipartite.

Proof of Claim 2, part 2

2. **Assume** that there is an edge $(u, v) \in E$ between two nodes u and v on the same layer.

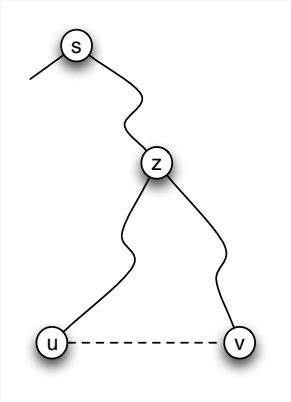
Obviously G is not 2-colorable by our algorithm: both endpoints of edge (u, v) are assigned the same color.

Our algorithm returns **no**, hence declares G non-bipartite.

Can we show existence of an odd-length cycle and prove that G indeed is not 2-colorable by **any** *algorithm?*

Proof of correctness, part 2

- Let u, v appear at layer L_j and edge $(u, v) \in E$.
- Let z be the common ancestor at max depth of u and v in the BFS tree (z might be s). Suppose z appears at layer L_i with $i < j$.
- Consider the following path in G: from z to u follow edges of the BFS tree, then to v via edge (u, v) and back to z following edges of the BFS tree. This is a cycle starting and ending at z, consisting of $(j - i) + 1 + (j - i) = 2(j - i) + 1$ edges, hence of odd length.



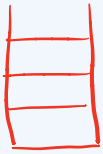
Today

1. Recap
2. Applications of BFS
   - Testing bipartiteness
3. Depth-first search (DFS)
4. Applications of DFS
   - Cycle detection
   - Topological sorting

Finding your way in a maze

• chalk → explored [ ]

• string → stack



Application for DFS ?
Use it to find your way in a maze of interconnected rooms.

Depth-first search (DFS): starting from a vertex s, explore the graph as deeply as possible, then backtrack

1. Try the first edge out of s, towards some node v.
2. Continue from v until you reach a dead end, that is a node whose neighbors have all been explored.
3. Backtrack to the first node with an unexplored neighbor and repeat 2.

Remark: DFS answers s-t connectivity — Because it explores all paths that start from s.

Use chalk to mark rooms that have already been explored, and use string to backtrack.

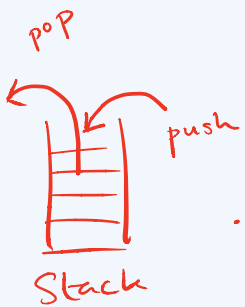
DFS vs BFS

Similarities (btwn DFS and BFS)

- Linear-time algorithms that essentially can be used to perform the same tasks

Differences (btwn DFS and BFS)

- DFS is more *impulsive*: when it *discovers* an *unexplored* node, it moves on to exploring it right away; BFS defers exploring until all nodes in the layer have been discovered.
- DFS is naturally recursive and implemented using a stack.
  - A stack is a LIFO (Last-In First-Out) data structure implemented as a doubly linked list: **insert** (push)/**extract** (pop) the top element requires $O(1)$ time.

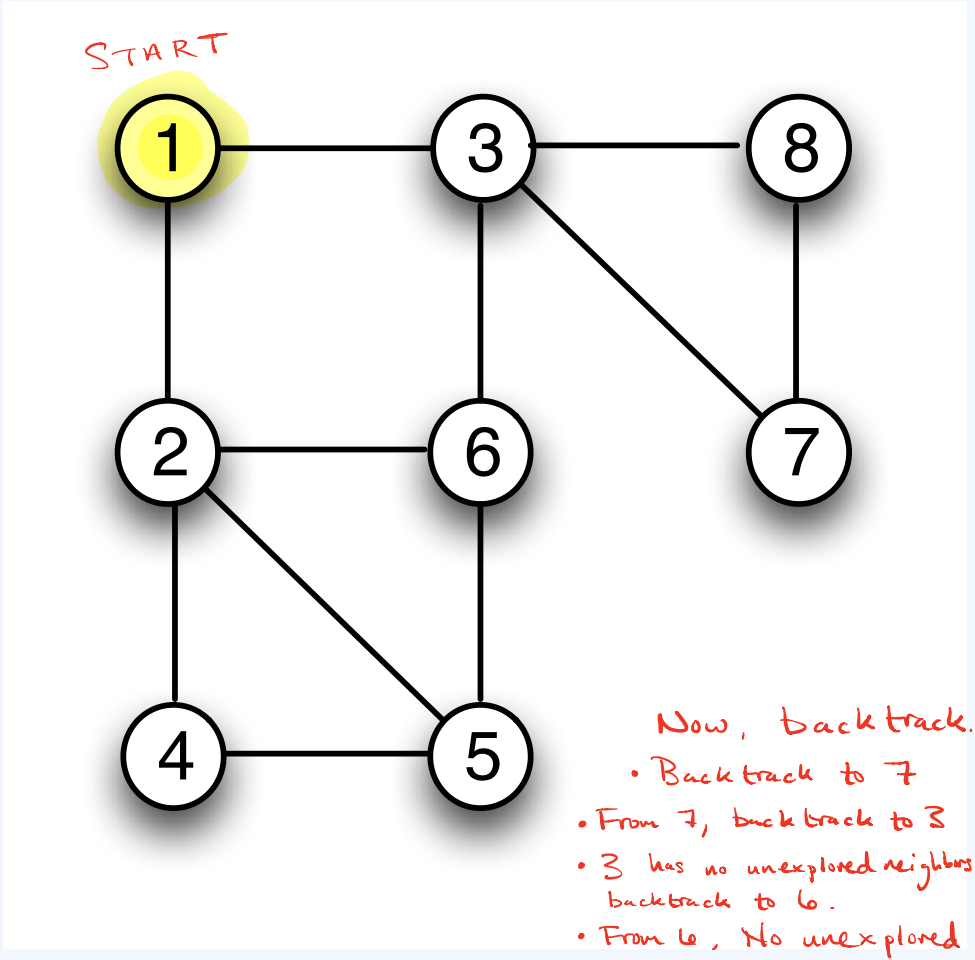


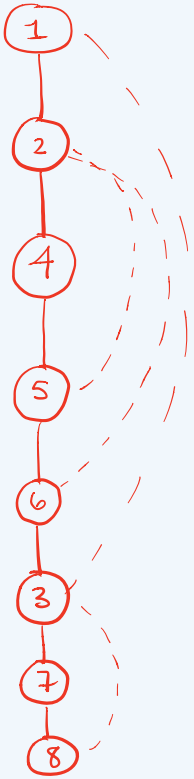
· Insert elements to the top of the stack (push)

· Extract the top element (pop).

(Always act on the top of stack.)

An undirected graph G_1



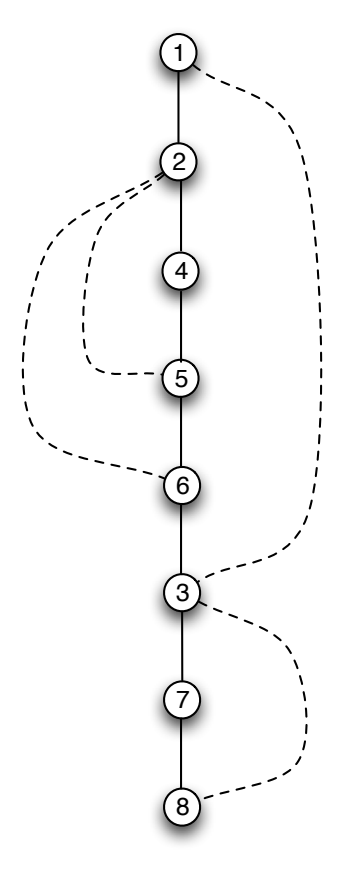


Output of DFS is a tree

Now, backtrack.

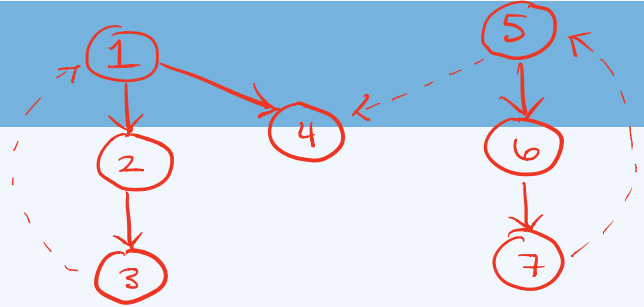
- Backtrack to 7
- From 7, backtrack to 3
- 3 has no unexplored neighbors backtrack to 6.
- From 6, No unexplored neighbors, backtrack to node 5.
- Node 5 has no unexplored neighbors. Backtrack to 4. ... to 2 ... to 1
- DFS terminates.

The DFS tree for the undirected graph G_1



Dashed edges belong to the graph but not to the DFS tree. Ties are broken by considering nodes by increasing index.

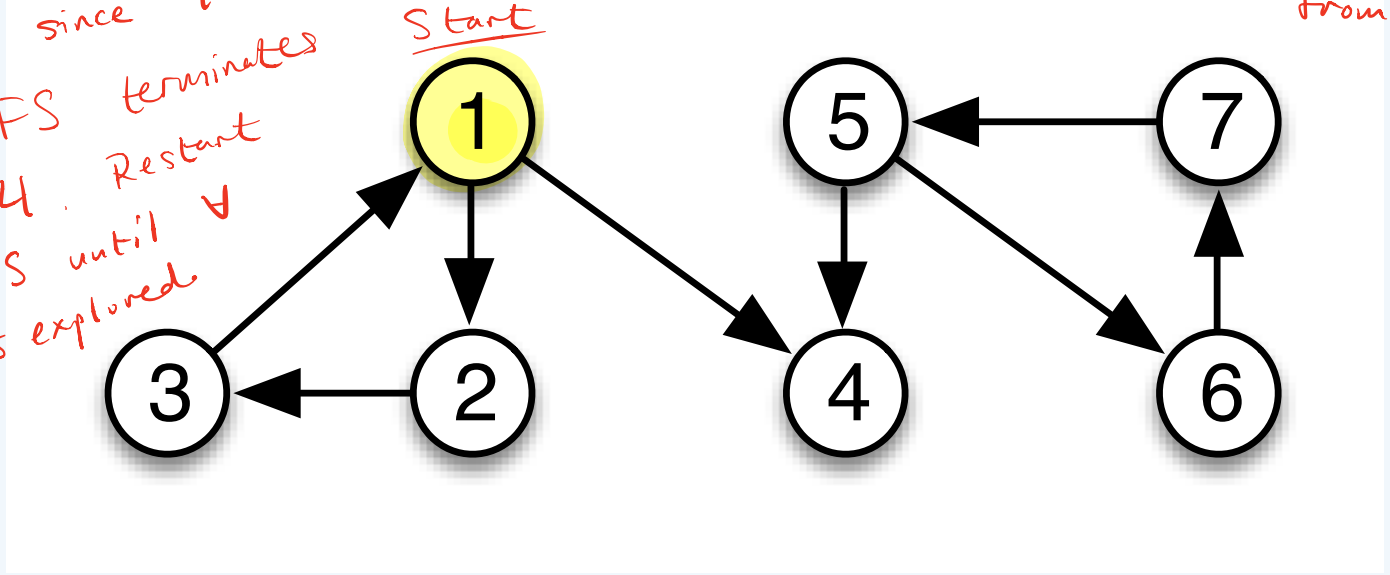
A directed graph G



Back-track from 3 to 2 to 1 then go to 4, since 4 is unexplored.

- Back-track to 6 – None
- Back-track to 5 ...
- Terminate from 5

DFS terminates @ 4. Restart DFS until all nodes explored.

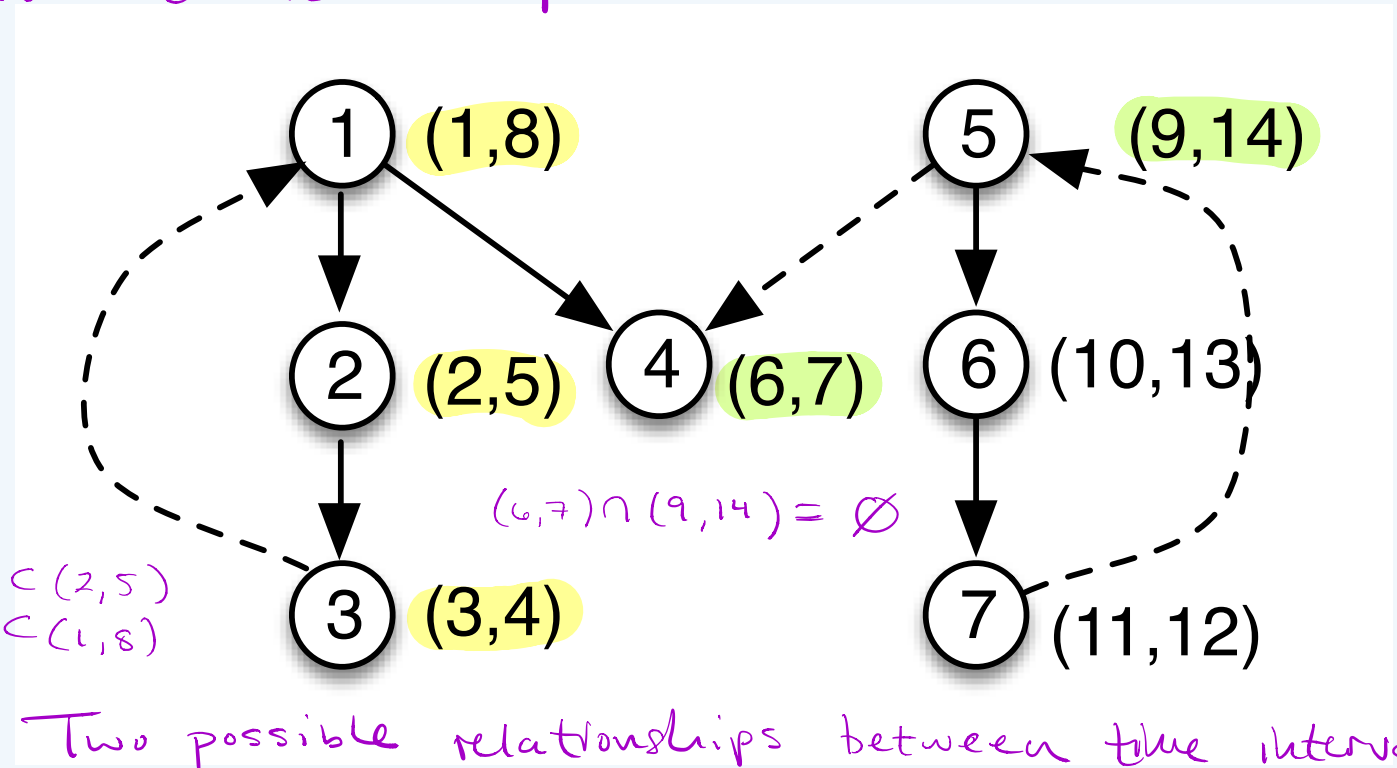


Restart DFS until all nodes in the graph have been explored.

- Output of DFS in this case is a forest of trees.

The DFS forest for the directed graph G

Notice the relationship of the time intervals



Two possible relationships between time intervals:

(i) Containment (ii) Disjoint

Dashed edges belong to G but not to the trees in the DFS forest.
$(start, finish)$ intervals appear to the right of every node.

Pseudocode for DFS exploration of the entire graph

TOTAL TIME = $O(n+m)$

```
DFS(G = (V, E))
    for u ∈ V do                                   O(n)
        explored[u] = 0
    end for
    for u ∈ V do                                   O(n)
        if explored[u] == 0 then Search(u)
        end if
    end for

Search(u)
    previsit(u)                                    → O(1)
    explored[u] = 1                                → O(1)
    for (u, v) ∈ E do                              O(deg(u))
        if explored[v] == 0 then Search(v)
        end if
    end for
    postvisit(u)                                   → O(1)
```

For fixed u, how many times do you call Search(u)? ONCE.

Let $T(u)$ = time for Search(u) EXCLUDING the time spent on recursive calls inside Search(u)

"Excluding" to avoid Double-counting

$* \quad T(1) + T(2) + \cdots + T(n)$

$= \sum_{u \in V} T(u) = c \cdot \sum_{u \in V} \deg(u)$

$= c \cdot 2m$

$= O(m)$

For fixed u: (time for comparison (explored[v] == 0))

$T(u) = c' + O(1) \cdot \deg(u) = c \cdot \deg(u)$

Running time for `DFS` if `previsit`, `postvisit` take $O(1)$ time?

Directed graphs: classification of edges

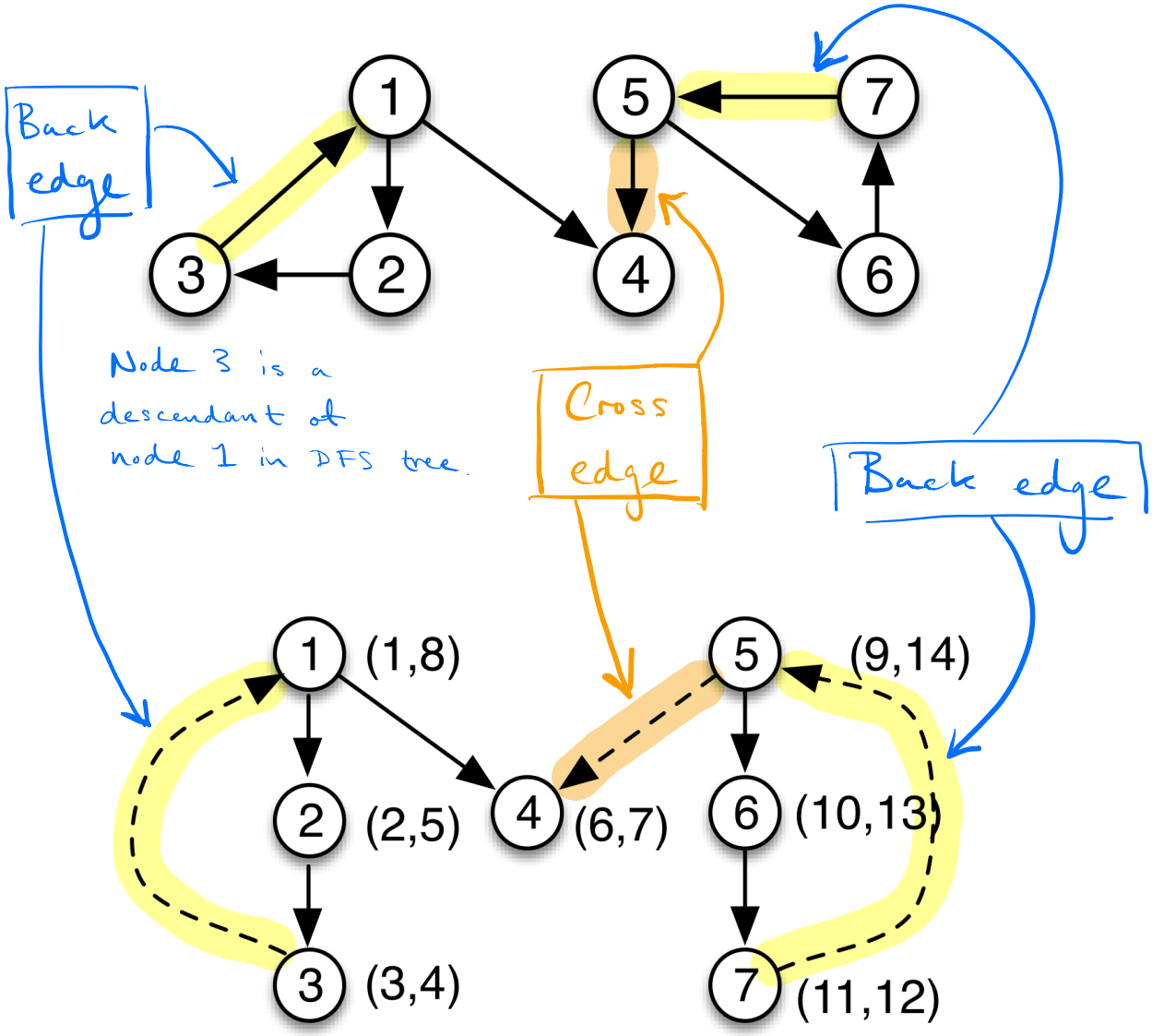
Remember these classifications. Important for structures of the DFS graph in our algorithms.

Graph edges that do not belong to the DFS tree(s) may be

1. forward: from a vertex to a *descendant* (other than a *child*)
2. back: from a vertex to an *ancestor*
   Examples: edges $(3, 1), (7, 5)$ in G
   ↩ must be in the same DFS tree.
3. cross: from right to left (no ancestral relation), that is
   - from tree to tree (example: edge $(5, 4)$ in G)
   - between nodes in the same tree but on different branches

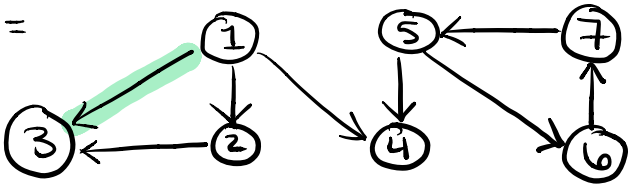
Ancestral relationships w.r.t. the DFS tree output.

(4) and (5) belong to different trees in the DFS output, but don't have to, to be a cross edge.

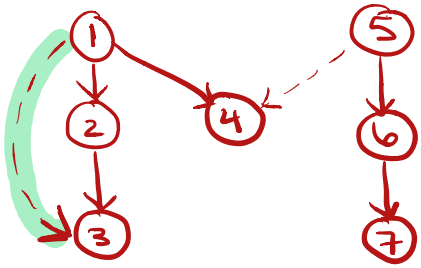


Example of forward edge:

G =



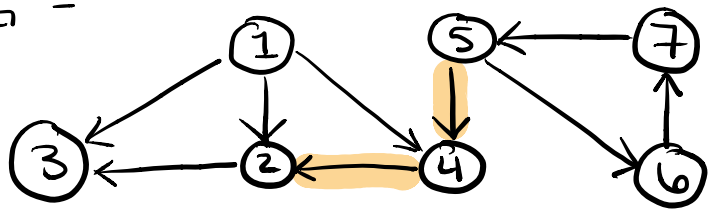
DFS output:



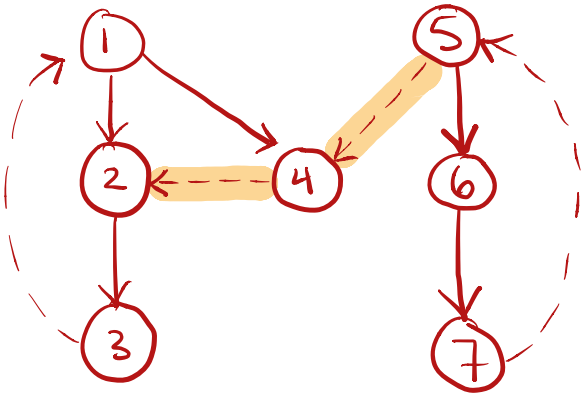
(1, 3) is a forward edge here.

Cross edge example

G =



DFS output:



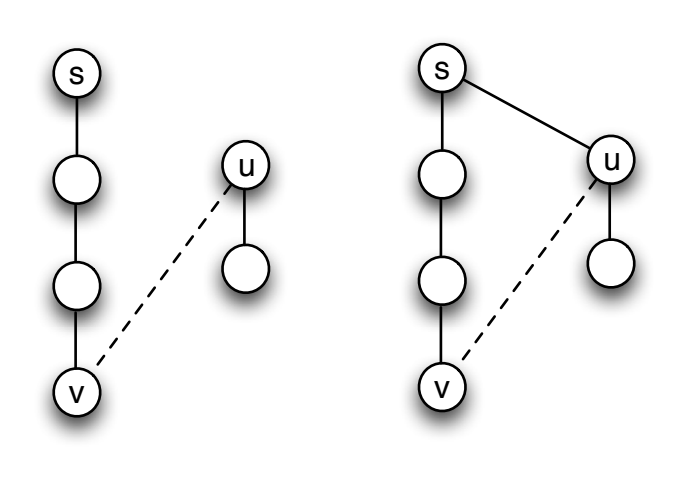
Cross edges:

(4,2) and (5,4)

There is no ancestral relationship between nodes 4 and 2. That is, they are not on the same path in the output of DFS. Likewise for nodes 5 and 4.

Undirected graphs: classification of edges

Cross and *forward* edges do not exist in undirected graphs.

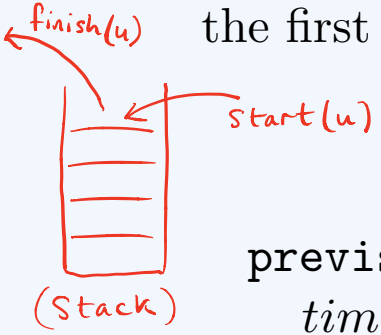


In undirected graphs, DFS only yields *back* and *tree* edges.

Time intervals for vertices

Subroutines `previsit`(u), `postvisit`(u) may be used to maintain a notion of time:

- In `DFS`(G), initialize a counter *time* to 0.
- Increment the counter by 1 every time `previsit`(u), `postvisit`(u) are accessed.
- Store the times $start(u)$ and $finish(u)$ corresponding to the first and last time u was visited during `DFS`(G).

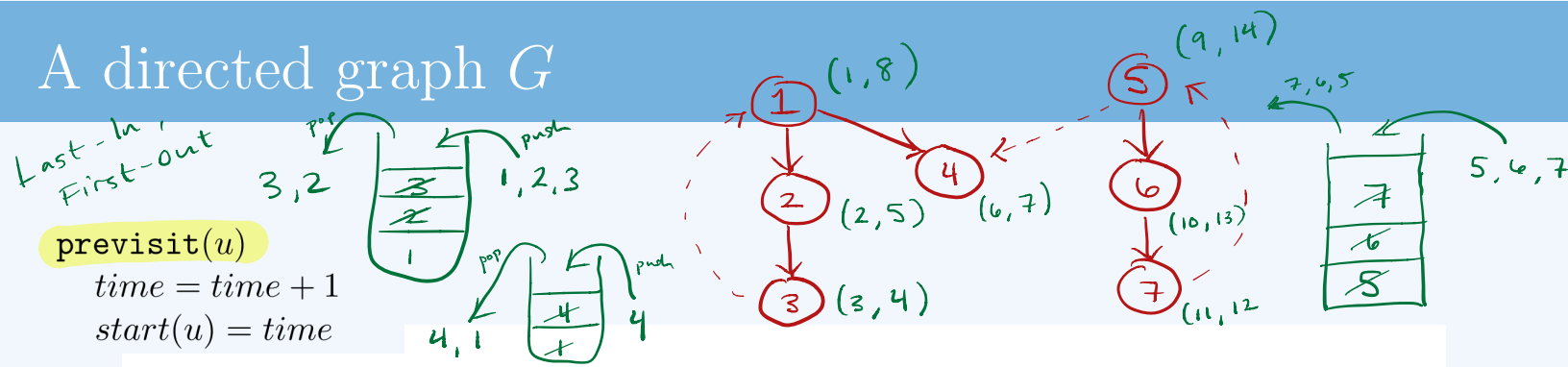


`previsit`(u)
$time = time + 1$
$start(u) = time$

`postvisit`(u)
$time = time + 1$
$finish(u) = time$

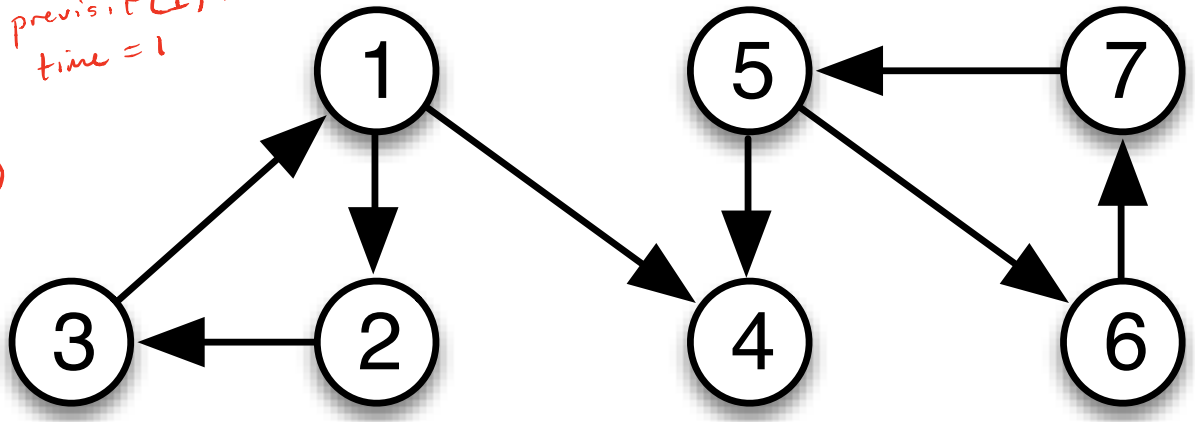
The interval $(start(u), finish(u))$ tells you the interval of time during which vertex u was active in the algorithm. (See Below for example)

A directed graph G



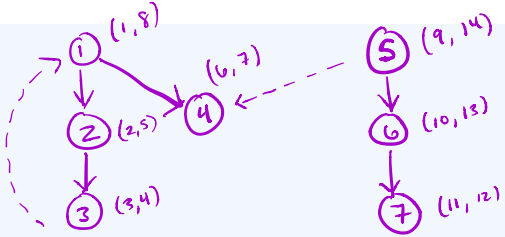
previsit(u)
$time = time + 1$
$start(u) = time$

previsit(1):
time = 1



previsit(u) recorded when you push u.

postvisit(u)
$time = time + 1$
$finish(u) = time$



postvisit(u) recorded when you pop u.

(5,4) is a cross edge.
⇒ start(v) < start(u) since v already explored and finish(v) < finish(u)

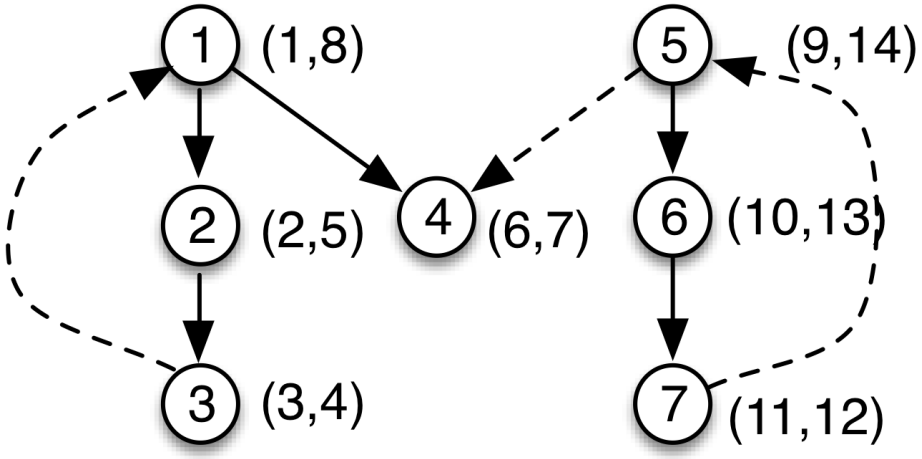
On the time intervals of vertices u, v

If we use an explicit stack, then

- $start(u)$ is the time when u is pushed in the stack
- $finish(u)$ is the time when u is popped from the stack (that is, all of its neighbors have been explored).

1. *How do intervals* $[start(u), finish(u)]$, $[start(v), finish(v)]$ *relate?* Two possible relationships between the intervals: (i) Containment (ii.) Disjoint
2. *What do the contents of the stack correspond to in the DFS tree (and the graph), if* s *was the first vertex pushed in the stack and* v *the last?* • corresponds to a path

• Interval containment corresponds to ancestral relationships
→ The interval of the descendant is fully contained in the interval of the ancestor.



- The time interval of descendants is fully contained within the interval of the parent node.
- If there is no ancestral relation, the time intervals are disjoint

Identifying back edges using time

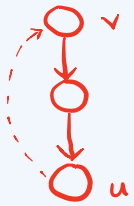
1. Intervals $[start(u), finish(u)]$ and $[start(v), finish(v)]$
   - either contain each other (u is an ancestor of v or vice versa)
   - or they are disjoint.
2. If s was the first vertex pushed in the stack and v is the last, the vertices currently in the stack form an s-v path.

Claim 3 (Back edges).

Let $(u, v) \in E$. Edge (u, v) is a back edge in a DFS tree if and only if

$$start(v) < start(u) < finish(u) < finish(v).$$

(eg. interval of u is fully contained within the interval of v)

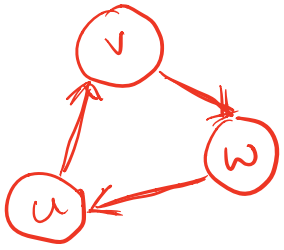


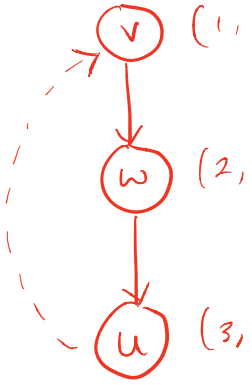
Use this fact to identify back edges in the DFS output.

Pf: (u,v) back edge in DFS tree

($\Rightarrow$)

$\Rightarrow$ start(v) < start(u) < finish(u) < finish(v)





If (u,v) a back-edge

then v must be an ancestor of u $\Rightarrow$ v enters the stack before u

$\Rightarrow$ start(v) < start(u)

and clearly start(u) < finish(u).

Since v in first, then by construction of DFS alg.,

v leaves last,

so finish(u) < finish(v)

$\Rightarrow$ start(v) < start(u) < finish(u) < finish(v)

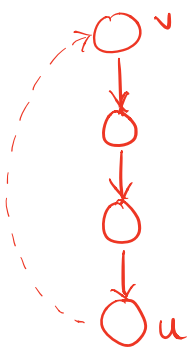
(⇐) Given the inequality

$start(v) < start(u) < finish(u) < finish(v)$

⇒ Since $start(v) < start(u)$

then v must be an ancestor of u.

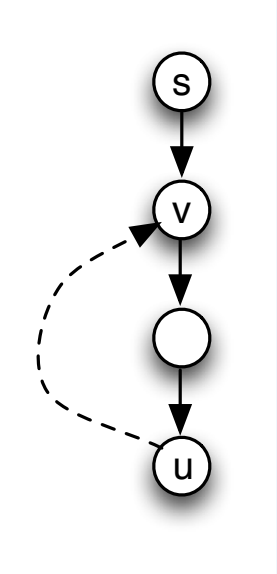
This implies there is a path from v to u



if there is an edge (u,v) in the graph then the edge (u,v) in the DFS tree must be a back edge.

Proof of Claim 3 (identifying back edges)

Proof.



If (u, v) is a back edge, the claim follows.

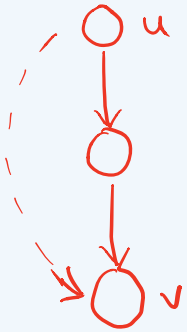
Otherwise, v was pushed in the stack before u and is still in the stack when u is pushed into it. Then there is a $v - u$ path in the DFS tree, so v is an ancestor of u and (u, v) is a back edge. $\square$

Identifying forward and cross edges

What conditions must the start and finish numbers satisfy if

1. $(u, v) \in E$ *is a forward edge in the DFS tree?* Not iff
2. $(u, v) \in E$ *is a cross edge in the DFS tree?* iff

(u,v) forward edge



interval of v fully contained in the interval of u.

start(u) < start(v) < finish(v) < finish(u)

(u,v) a cross edge

iff start(v) < finish(v) < start(u) < finish(u)

Identifying forward and cross edges

$s(v) <$

What conditions must the start and finish numbers satisfy if

1. $(u, v) \in E$ *is a forward edge in the DFS tree?*
2. $(u, v) \in E$ *is a cross edge in the DFS tree?*

1. If edge $(u, v) \in E$ is a forward edge, then

$$start(u) < start(v) < finish(v) < finish(u).$$

[the interval of v is fully contained within the interval of u]

(NOT iff)

2. If edge $(u, v) \in E$ is a cross edge, then

iff

$$start(v) < finish(v) < start(u) < finish(u).$$

Not iff : Inequality implies two possible cases : (i) edge is part of the DFS tree ; (ii) edge is a forward edge.

Output of DFS:

- For undirected graphs, the output of DFS is all the connected components of the input graph. If the graph is not connected, this will take the form as a forest of trees. If the graph is connected, the output will be a single, undirected tree.

- For directed graphs, the output of DFS is a forest of directed trees.

Today

$G = (V, E)$ has a cycle

$\Longleftrightarrow$

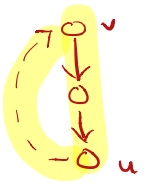
DFS finds a back edge

Claim 4.

$G = (V, E)$ has a cycle if and only if $\texttt{DFS}(G)$ *yields a back edge.*

Pf:

($\Leftarrow$) Trivial;



gives cycle in the graph G.

($\Rightarrow$) Spz $G = (V, E)$ has a cycle

W.l.o.g. let v be the first node of the cycle discovered by DFS, and let u be the node preceeding v in the cycle.

All nodes in the cycle will appear in the DFS sub-tree of v since there is path from any node in the cycle to all other nodes in the cycle (all nodes are reachable from v).

That is, since v is the first node in the cycle, all nodes in the cycle appear below v in the DFS output tree

Since there is a path from v to every node in the cycle. Every node will appear, including u. More, since u precedes v, there is an edge from u to v. Since v is the first node in the DFS tree. and has already been explored, (u,v) is a back edge. since it is an edge from a descendant to an ancestor.

Application I: Cycle detection

Claim 4.

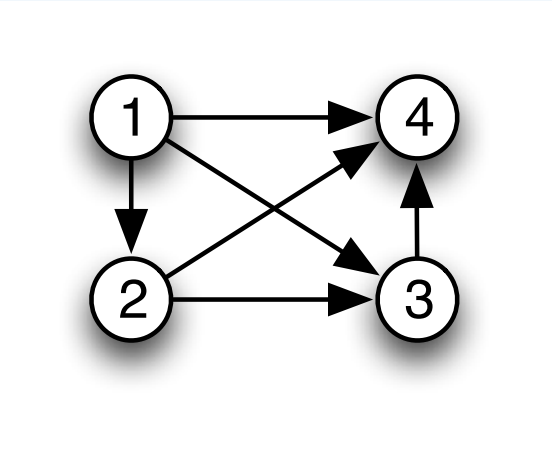
$G = (V, E)$ has a cycle if and only if `DFS`(G) *yields a back edge.*

Proof.

If (u, v) is a back edge, together with the path on the DFS tree from v to u, it forms a cycle.
Conversely, suppose G has a cycle. Let v be the first vertex from the cycle discovered by `DFS`(G). Let (u, v) be the preceding edge in the cycle. Since there is a path from v to *every* vertex in the cycle, all vertices in the cycle are now discovered **and** fully explored **before** v is popped from the stack. Hence the interval of u is contained in the interval of v. By Claim 1, (u, v) is a back edge. □

Application II: Topological sorting in DAGs

- An undirected acyclic graph has an extremely simple structure: it is a tree, hence a sparse graph ($O(n)$ edges).
- A directed acyclic graph (DAG) may be dense ($\Omega(n^2)$ edges): e.g., $V = \{1, \ldots, n\}$, $E = \{(i, j) \text{ if } i < j\}$.



Topological sorting: motivation

Input:

- a set of tasks $\{1, 2, \ldots, n\}$ that need to be performed
- a set of dependencies, each of the form (i, j), indicating that task i must be performed before task j.

Output: a valid order in which the tasks may be performed, so that all dependencies are respected.

Example: tasks are courses and certain courses must be taken before others.

How can we model this problem using a graph? What kind of graph must arise and why?

Topological ordering: definition

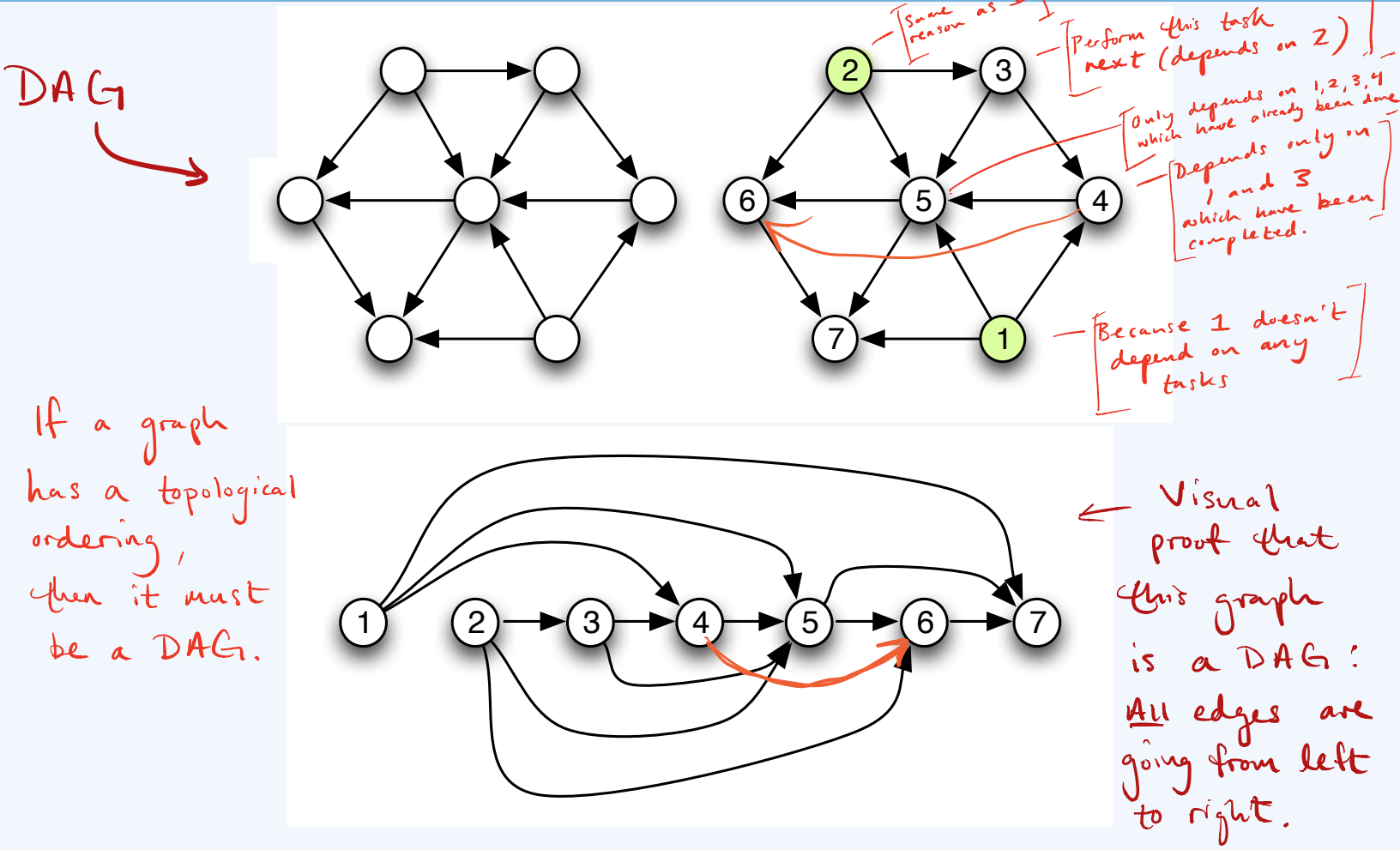
Definition 2.

A topological ordering of G is an ordering of its nodes as $1, 2, \ldots, n$ such that for every edge (i, j), we have $i < j$.

originating end pt has an index smaller than the destination end pt.

- All edges point **forward** in the topological ordering.
- It provides an order in which all tasks can be safely performed: when we try to perform task j, all tasks required to precede it have already been done.

Example of DAG and its topological sorting



A DAG (top left), its topological sort (top right) and a drawing emphasizing the topological sort (bottom).

Topological sorting in DAGs

Claim 5.

If G has a topological ordering, then G is a DAG.

Proof: By contradiction (*exercise*).

A visual proof is provided by the linearized graph of the previous slide: vertices appear in increasing order, edges go from left to right, hence no cycles.

Is the converse true: does every DAG have a topological ordering? And how can we find it?

* Yes, every DAG has a topological ordering.

G has a topological ordering $\Rightarrow$ G is a DAG

Is the converse true? That is, is it true that:

G is a DAG $\Rightarrow$ G has a topological ordering?

Yes; Consider the following:

Suppose $|V| = n$, If n had an outgoing edge here, the graph would have a cycle.

- Does every node in a DAG have an outgoing edge?

No. If every node had an outgoing edge, then the graph has a cycle which is not possible for a DAG.

This property implies:

No $\Rightarrow$ Every DAG has ≥ 1 sink

Ex: 

Every DAG has at least one sink.

- Does every node in a DAG have an incoming edge?

No; Same argument

It would otherwise have a cycle.

This property implies:

No $\Rightarrow$ Every DAG has ≥ 1 source

Ex:

Every DAG has at least one source.

To find a topological ordering for a DAG, start with a source node. We know at least one source node exists for all DAGs.

Structural properties of DAGs

In a DAG, can **every** vertex have

- *an outgoing edge?*
- *an incoming edge?*

Definition 3 (source and sink).

A source is a node with no incoming edges.
A sink is a node with no outgoing edges.

Fact 4.

Every DAG has at least one source and at least one sink.

To find a topological ordering for a DAG, start with a source node (We know at least one source node exists for all DAGs.):

(I.) Label the source node first (label = 1) and then remove the node from the graph and its adjacent edges.

(II.) Next, find another source node. We know there be another source in the remaining graph since the graph is still a DAG — removing nodes and edges does not create cycles. Label the next source as 2.

(III.) Repeat this procedure recursively.

How can we use Fact 4 to find a topological order?

The node that we label *first* in the topological sorting must have no incoming edges. Fact 4 guarantees that such a node exists.

Fact 5.

Let G' be the graph after a source node and its adjacent edges have been removed. Then G' is a DAG.

Proof: removing edges from G cannot yield a cycle!

This gives rise to a recursive algorithm for finding the topological order of a DAG. Its correctness can be shown by induction (use Facts 4, 5 to show induction step).

Algorithm for topological sorting

Data structure to use

stores in-degrees of every node.

`TopologicalOrder`(G)

1. Find a source vertex s and order it first.
2. Delete s and its adjacent edges from G; let G' be the new graph.
3. `TopologicalOrder`(G')
4. Append the order found after s.

Running time: $O(n^2)$. Can be improved to $O(n+m)$.

→ n calls to TopologicalOrder(.)

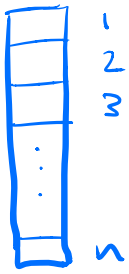
→ $O(n)$ per call

Running Time of TopologicalOrder procedure

`TopologicalOrder`(G)

1. Find a source vertex s and order it first. → $O(n)$
2. Delete s and its adjacent edges from G; let G' be the new graph. → $O(\deg(s))$
3. `TopologicalOrder`(G') →
4. Append the order found after s. →

We will introduce an array indegrees =



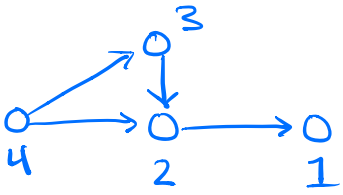
of size n, where:

indegrees[i] = indegree of node i

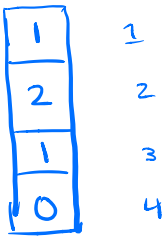
The array stores the indegree of each node.

This is useful for this since source nodes will have an indegree equal to 0.

Ex:



indegrees =



If a node v is such that indegrees[v] = 0, then that node is a source node. We use this array to identify a source node.

- How much time needed to initialize the array indegrees?

$$O(n+m)$$

Indeed, we need to scan the graph to determine how many in-neighbors every node has, and the graph has an adjacency list representation. Therefore, we will need to scan the adjacency list representation, which will take $O(n+m)$ time. (Scanning list and updating indegrees).

- Step 1 running time:

How much time do you need to find a source vertex given the array indegrees (Step 1)?

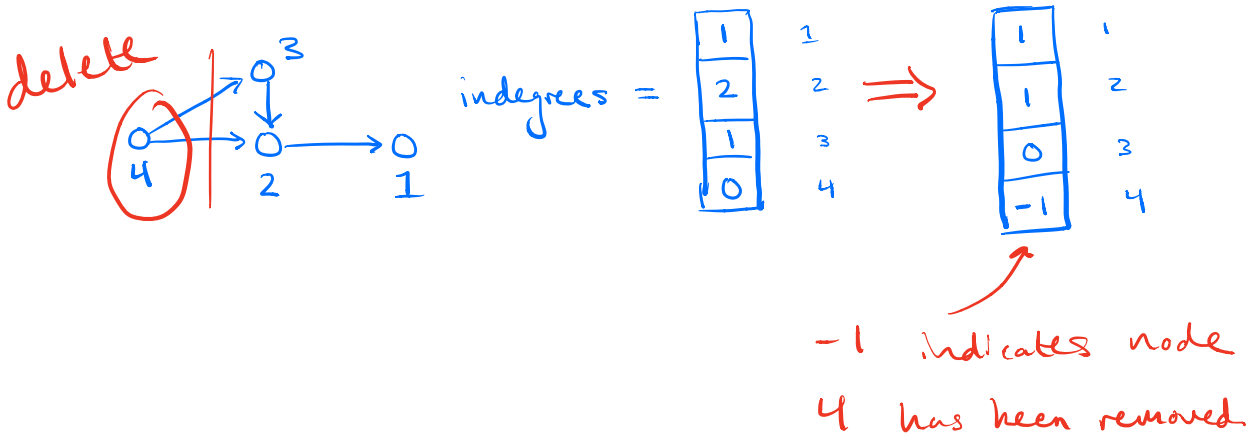
$$O(n)$$

We must scan the array indegrees to find a node with indegree = 0.

We are guaranteed such a node will exist since we are working with a DAG.

- Step 2 running time:

To remove a source node s and edges, all you need to do is update the array indegrees to delete the node. For example:



Then Step 2 takes $O(\deg(s))$ time for each source node.

Total running time:

$$n \cdot \left(O(n) + O(\deg(s)) \right)$$

$$= O(n^2) + O\left(\sum_{s \in V} \deg(s) \right) =$$

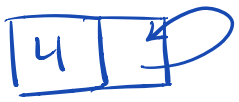
$= O(n^2) + O(m) = O(n^2)$

The bottleneck of this algorithm is step 1, identifying the source vertex. Therefore, there is room for improvement here.

To improve the algorithm, we will modify

- Step 2

Step 2: maintain a linked list with source nodes. Every node v with indegree[v] = 0 will be added to the linked list.

Sources 

Initialize sources: $O(n)$ time

- Step 1 now takes $O(1)$ time because just need to extract head element.

* Maintain a linked list of sources.

Modified Algorithm:

Step 1: $O(1)$

Step 2: $O(\deg(s))$

Total Running time:

$$O(n+m) + n \cdot \left(O(1) + O(\deg(s))\right) + + O(n)$$

$$= O(n+m) + O(n) + O(m) + O(n)$$

$$= O(n+m)$$

We will now cover yet another way to do topological ordering/sorting: 2

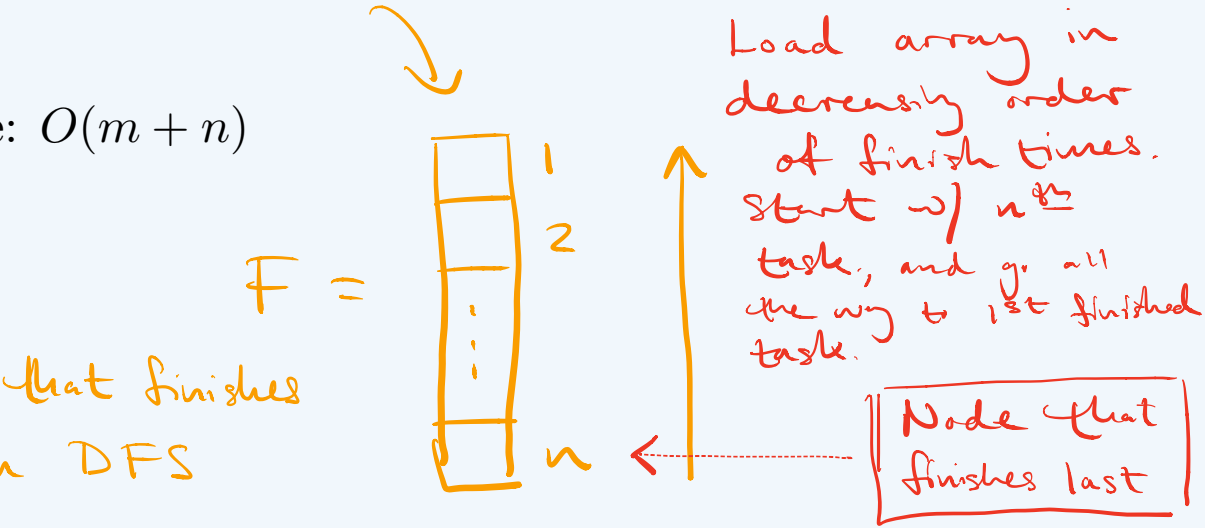
Topological sorting via DFS

Computing finish times requires $O(n+m)$ time.

Let $G = (V, E)$ be a DAG.

- Run `DFS`(G); compute *finish* times.
- Process the tasks in decreasing order of *finish* times.

Running time: $O(m+n)$



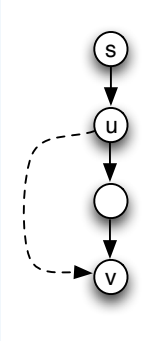
Intuition behind this algorithm

- The task v with the largest $finish$ has no incoming edges (if it had an incoming edge from some other task u, then u would have the largest $finish$). Hence v does not depend on any other task and it is safe to perform it first.
- The same reasoning shows that the task w with the second largest $finish$ has no incoming edges from any other task except (maybe) task v. Hence it is safe to perform w second.
- And so on and so forth.

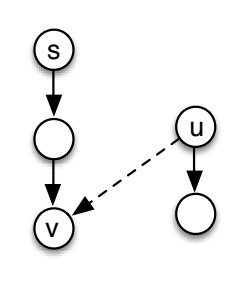
Formal proof of correctness

By Claim 4 there are no back edges in the DFS forest of a DAG. Thus every edge $(u, v) \in E$ is either

1. forward/tree: $start(u) < start(v) < finish(v) < finish(u)$



2. or cross edge: $finish(v) < start(u) < finish(u)$



Proof of correctness (cont'd)

Hence for every $(u, v) \in E$, $finish(v) < finish(u)$.

Consider a task v. All tasks u upon which v depends, that is, all tasks u such that there is an edge $(u, v) \in E$, satisfy $finish(v) < finish(u)$.

Since we are processing tasks in decreasing order of finish times, all tasks u upon which v depends have already been processed before we start processing v.

Let $G = (V, E)$ be a DAG.

- Run DFS(G); compute *finish* times.
- Process the tasks in decreasing order of *finish* times.

Correctness: Let w be the task with largest finish time. We want to show that it is safe to perform w first:

Since w has the largest finish time it does not depend on any other task. The task we perform first should be a source task. w should not have any incoming edges:

Suppose to the contrary that $\exists (x, w) \in E$.

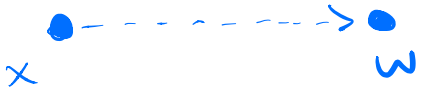
This means that x would also be discovered by DFS. There are 2 cases:

(1.) x is discovered by DFS before w.

(2.) x is discovered by DFS after w.

In Case (1), if x is discovered before

w, then x is an ancestor of w.



This implies that the finish time of x is larger than the finish time of w, a contradiction.

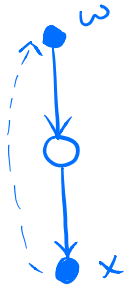
In Case (2), there are two sub-cases:

(I.) x discovered after w has finished $\Rightarrow$ (x, w) a cross edge.

This also implies that the finish time of x is greater than the finish time of w since for cross edges:

start(w) < finish(w) < start(x) < finish(x).

A contradiction.



(II.) x discovered after w while w is still in the stack. Then, (x, w) is a back edge $\Rightarrow$ the graph has a cycle, which contradicts G being a DAG. //

□.

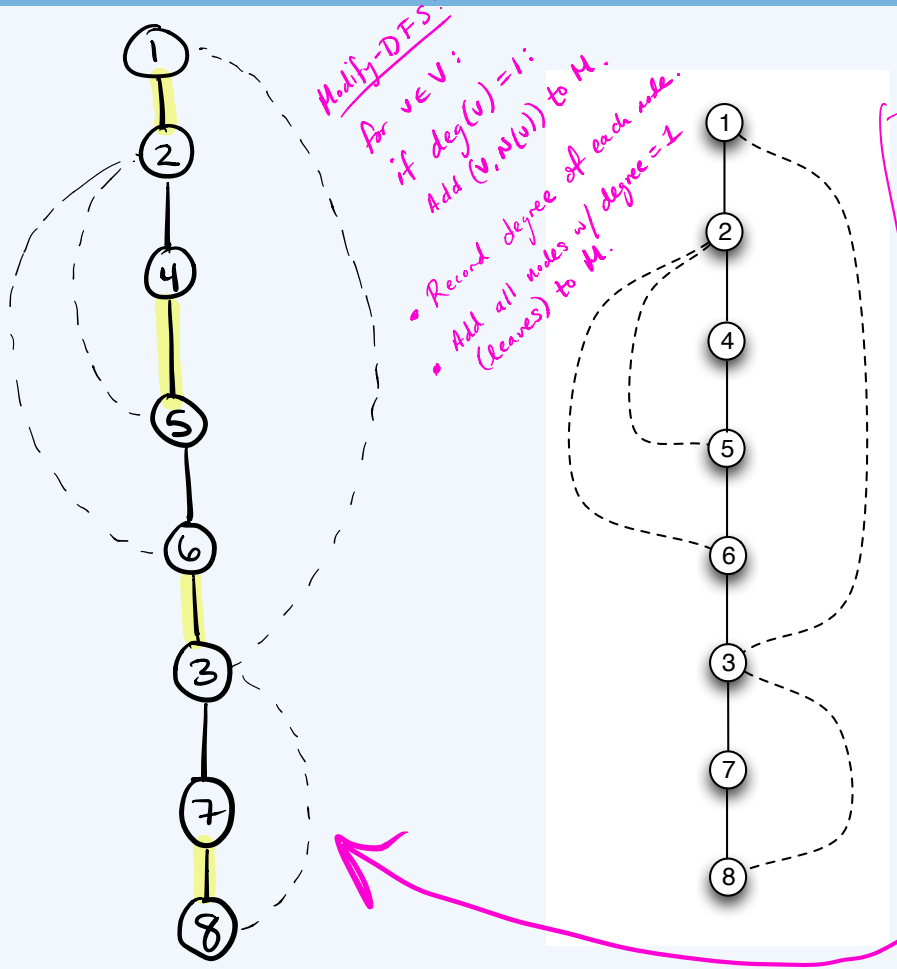
The following pages contain

SCRATCH WORK

for Homework problem

on Perfect Matchings

The DFS tree for the undirected graph G_1



Modify-DFS:
For $v \in V$:
if $deg(v) = 1$:
Add $(v, N(v))$ to M.

- Record degree of each node.
- Add all nodes w/ degree = 1 (leaves) to M.

if matched[v] == True:

* In the DFS tree add every other edge to M.

DELETE WHEN DONE

Dashed edges belong to the graph but not to the DFS tree. Ties are broken by considering nodes by increasing index.

```
DFS(G = (V, E))
    for u ∈ V do
        explored[u] = 0
    end for
    for u ∈ V do
        if explored[u] == 0 then Search(u)
        end if
    end for

Search(u)
    previsit(u)
    explored[u] = 1
    for (u, v) ∈ E do
        if explored[v] == 0 then Search(v)
        end if
    end for
    postvisit(u)
```

Running time for DFS if previsit, postv

Total time

O(n)

O(n)

O(m)

For fix

How many t

you call Se

Let T(u) =

EX

Sp

in

* T(1)

=

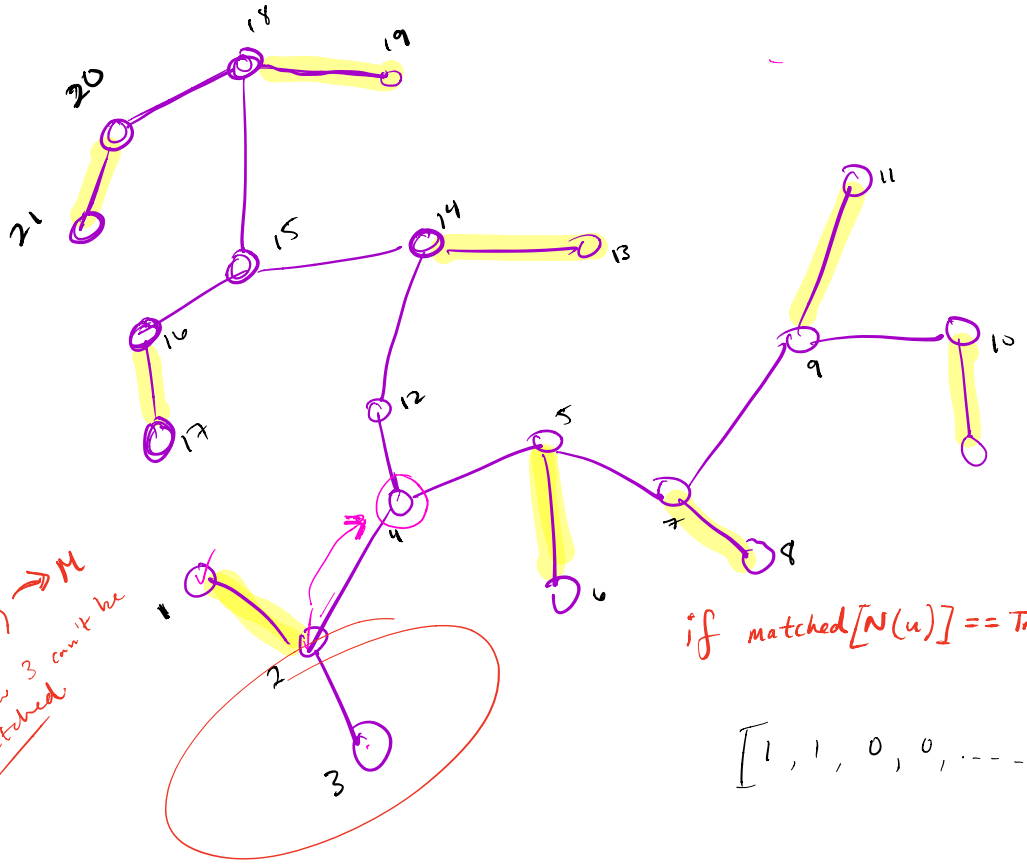
For fixed

T(u) =

```
for u ∈ V :
    if matched[u] == False :
        if deg(u) == 1 :
            Add (u, N(u)) to M
        if
```

Neighbor[u]

```
for u ∈ V :
    if matched[u] == False :
        if deg(u) == 1 :
```



(1, 2) → M

But now 3 can't be matched

```
if matched[N(u)] == True :
```

[1, 1, 0, 0, ----, 0]

if deg(u) == 1 and matched[u] == 0 and matched[N(u)] == 0 :

matched[u] = 1

matched[N(u)] = 1

else if deg(u) == 1 and matched[u] == 0 and matched[N(u)] == 1 :

matched[u] = 0

for $(u, v) \in E$:

if matched[u] == 0 AND matched[v] = 0

Matched[u] = 1

matched[v] = 1

if deg(u) == 1 and matched[u] == 0 :

if matched[N(u)] == 0 :

matched[u] = 1

matched[N(u)] = 1

else :

return 'False'

```
if sum(matched) == size(V) :
    return 'True'
```