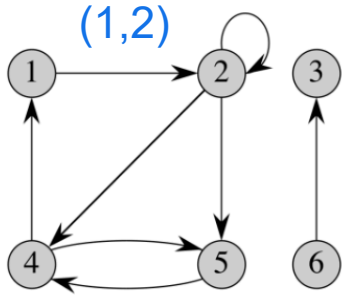
Graphs and Graph Traversals

Graphs

A set V of vertices (nodes).

A set $E \in V \times V$ of edges. That is, pairs of vertices.

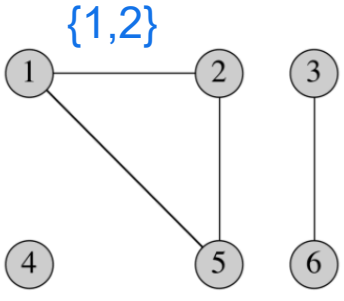
n (nodes) = 6
m (edges) = 8



(a)

n = 6

m = 4



(b)

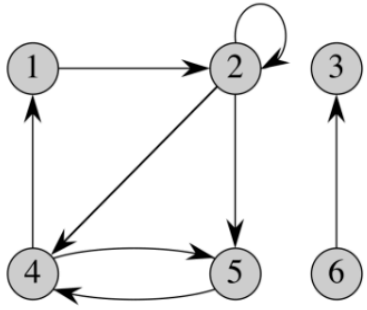
same in undirected graph

- Directed graphs: edges are ordered pairs. (2,5) have arrows
- Undirected graphs: edges are unordered pairs. {2,5} connect with line
- Weighted graphs: each edge has a number associated with it.
- Notation: n = |V|, m = |E|.
- Note that $0 \leq m \leq n^2$ for directed graphs and $0 \leq m \leq (n^2 + n)/2$ for undirected graphs.

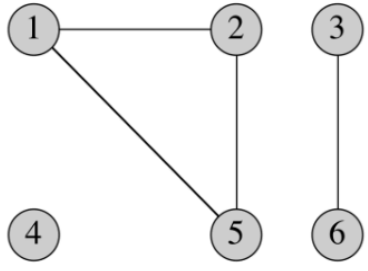
Graphs

Models for many things:

- Wiring network (telephone, electricity, oil, water,. . .).
- Road network (intersections are vertices, roads between intersections are edges)
- Friends on SoMe.
- Followers on SoMe..
- WWW pages
- Co-authorship.



(a)



(b)

Lots of algorithmic questions on graphs

- How to represent graphs on a computer (data structure)?
- Is there a path between two specified vertices?
- What is the shortest path between two specified vertices?
- What is the smallest subset of edges that still keeps all vertices connected?
- What is the largest collection of edges where no edges have any common vertices?
-

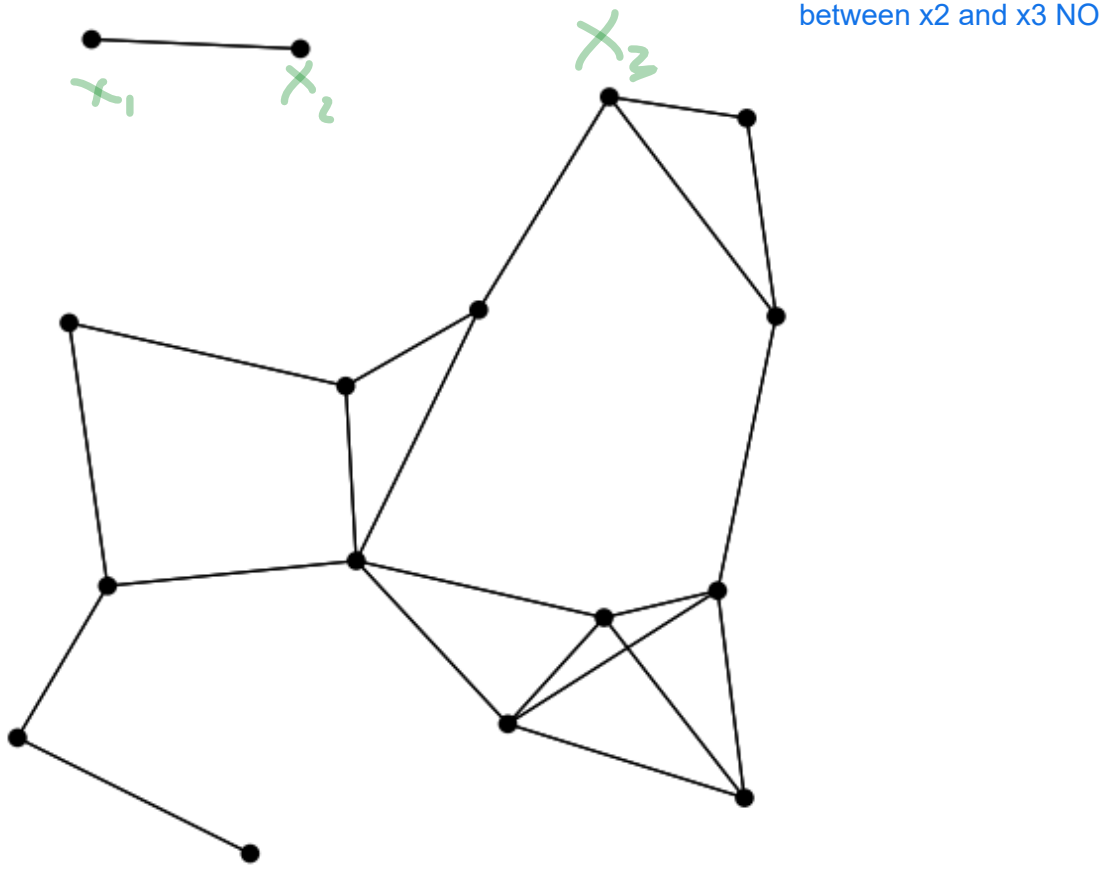
...

Example of an algorithmic question

Determine if there is a path between two given vertices.

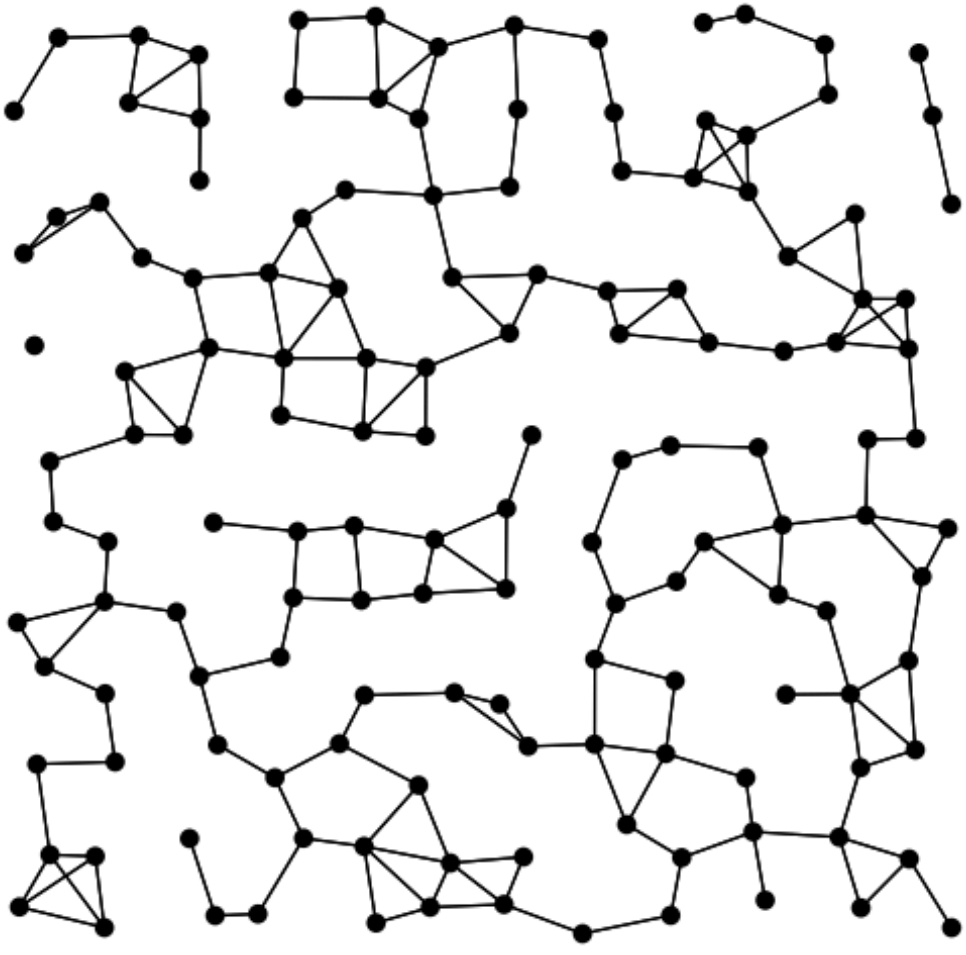
between these two yes

between x2 and x3 NO



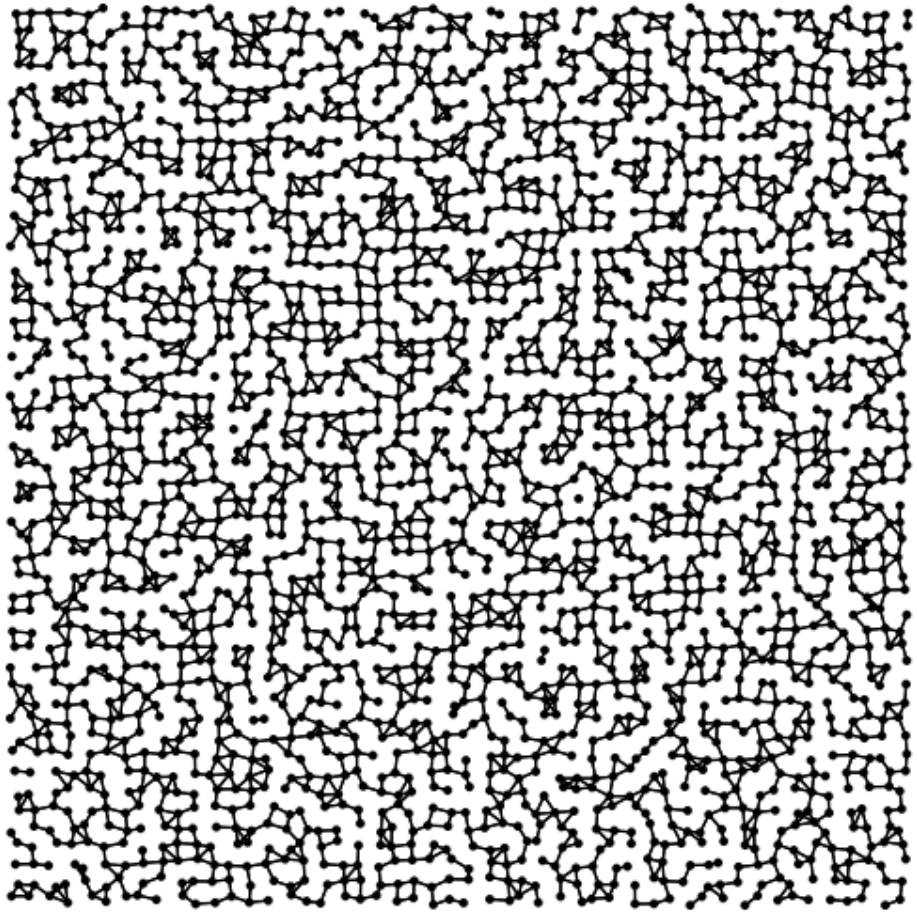
Example of an algorithmic question

Determine if there is a path between two given vertices.



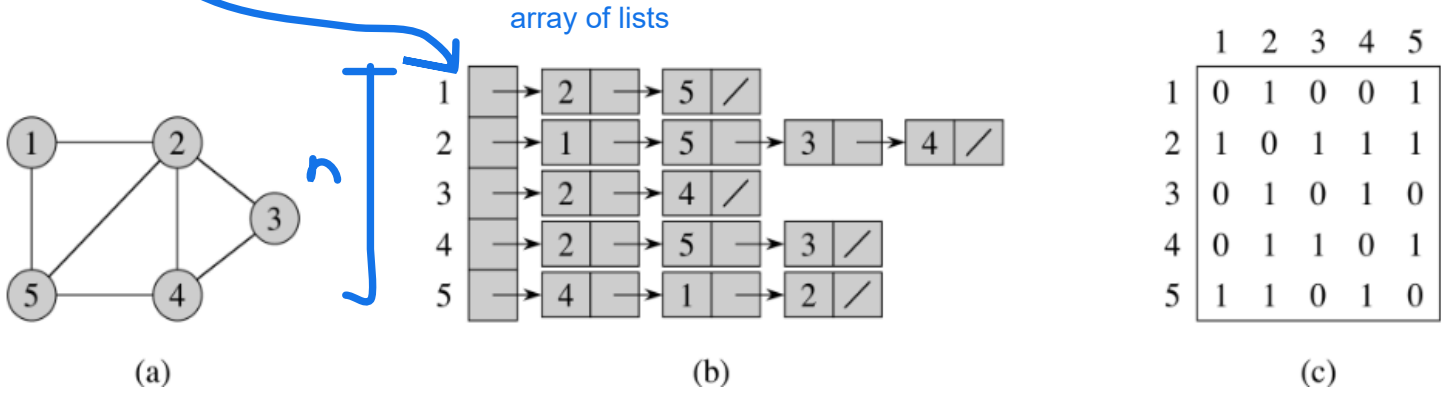
Example of an algorithmic question

Determine if there is a path between two given vertices.



Data structures for graphs

Adjacency lists and adjacency matrix



if 1, a line between two points, fx 5,2

for each edge contaion the edge it connects DIRECTLY to

Adjacency lists: the list for u contains v for all edges (u, v) ∈ E. Nodes are represented as integers between 1 and n (or 0 and n − 1). Space: O(n + m) for adjacency lists, $O(n^2)$ for adjacency matrix. Unless otherwise noted, the adjacency lists representation in algorithms is used in this course.

An edge in an undirected graph is represented as two directed edges (so for the purposes of implementation, undirected graphs are just a special case of directed graphs).

Graph Traversal

Task: Given a graph in adjacency list representation, visit all vertices and edges. The goal is to learn about various properties of the graph.
General Idea: Visit a starting vertex **s**. Use the edges in the neighbor lists of visited vertices to visit more vertices.

Mark vertices along the way to keep track of the process:

- **White vertices:** not yet visited
- **Gray vertices:** visited, but not all edges in the neighbor list have been used.
- **Black vertices:** visited, and all edges in the neighbor list have been used.

Graph Traversal

Generic algorithm for graph traversal:

$\mathrm{GenericGraphTraversal1(s)}$

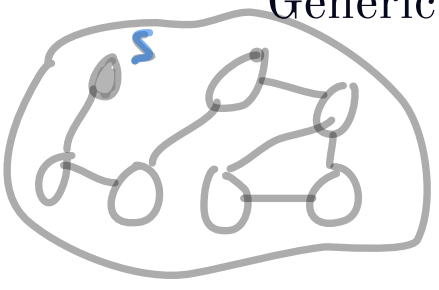
```
Make s gray and all other vertices white
while there are gray vertices:
    choose a gray vertex v
    if v's neighbor list is used up:
        make v black
    else
        choose an unused edge (v, u) from v's neighbor list
        if u is white:
            make u gray
```

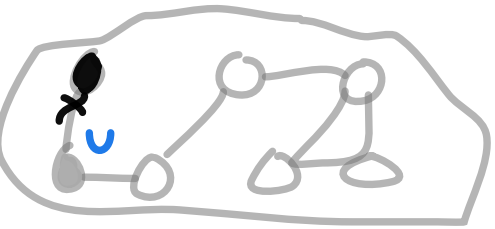
A vertex's life cycle: white → gray → black. When the algorithm stops, all vertices are either white or black.

Graph Traversal

Later in the course, we will encounter three variants,each with different strategies for choosing the next edge (v, u) to use, that is, for the choices (*).

GenericGraphTraversal1(s) s is the starting vertice/node



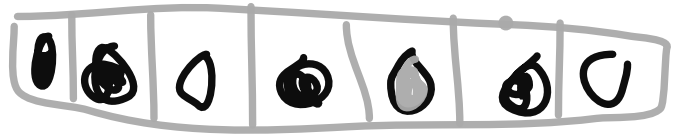



```
Make s gray and all other vertices white
while there are gray vertices:
    choose a gray vertex v (*)    grey = visited but not all edges are used yet
    if v's neighbor list is used up:
            make v black
    else
            choose an unused edge (v, u) from v's neighbor list (*)
            if u is white:
                make u gray
```

- Breadth-First-Search (BFS)
- Depth First-Search (DFS)
- Priority-Search (Dijkstra's algorithm, A*)

To look around the entire graph:

for loop, that loops over a whole array, will eventually find all whites, despite their ID



can jump over black ones, even if not connected to look at the whole graph.
so everything will be black when we are done

```
GenericGraphTraversalGlobal()
    Make all vertices white
    for all vertices s:
        If s is white:
            GenericGraphTraversal2(s)
```

just like the algortihm before, but divided into two algorithms. Now the top makes the vertices white, and the bottom makes them gray

```
GenericGraphTraversal2(s)
    Make s gray
    while there are gray vertices:
        choose a gray vertex v (*)
        If v's neighbor list is used up
            make v black
        else
            choose  an unused edge(v, u) from v's neighbor list (*)
            If u is white:
                make u gray
```

number of vertices + number of edges (every thing must be visited)

If (*) takes time O(1), the total running time is O(n + m). (A given edge can only be chosen once, so all the work done in the else part takes O(m) time in total.The rest takes O(n) time in total.)

when the algorithm is done, every vertrici will be black

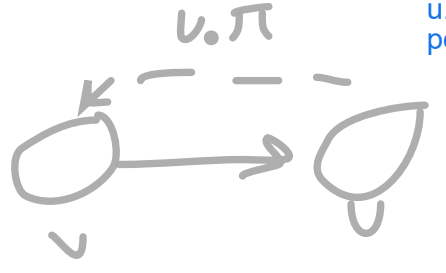
Remember who discovered whom:

u is not starting vertex

When a vertex u (≠ s) is visited for the first time, it stores in the variable u.π the vertex that discovered u (the predecessor).Note that u.π is set at most once (after being initialized to NIL), because u is made gray at the same time.

```
GenericGraphTraversalGlobalWithParents()
Make all vertices white and set their π to NIL
    for all vertices s:
        If s is white:
                GenericGraphTraversal3(s)

GenericGraphTraversal3(s)
    Make s gray
    while there are gray vertices:
        choose a gray vertex v (*)
        If v's neighbor list is used up:
                make v black
            else
                choose an unused border (v, u) from v's neighbor list (*)
            if u is white:
                make u gray
                set u.π equal to v
```



u.pi saves what vertrices pointed to u first.

Breadth-first search (BFS)

Strategy: Keep the gray vertices in a queue, and use up the neighbor lists immediately. Also add a variable v.d to every vertex v (d stands for distance).

The most commonly used version is the one without the Global part (for BFS we are often more interested in one specific s rather than traversing the entire graph).

Invariant: queue = all gray vertices.

Running time: O(n + m).

Breadth-first search (BFS)

1-4 is inizilation

O(n+m)

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY          s = starting node/vertex
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅                   Que is empty
 9  ENQUEUE(Q, s)           Enqueue = put back in the que
                            Here put Starting point in que
10  while Q ≠ ∅
11      u = DEQUEUE(Q)      Dequeue = take one out of the que in front
12      for each v ∈ G.Adj[u]   for each neighbors
13          if v.color == WHITE
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)
18      u.color = BLACK
```

.d = distance from starting node
.pi = the predecessor/parent
(who discovered me)

Breadth-first search (BFS)

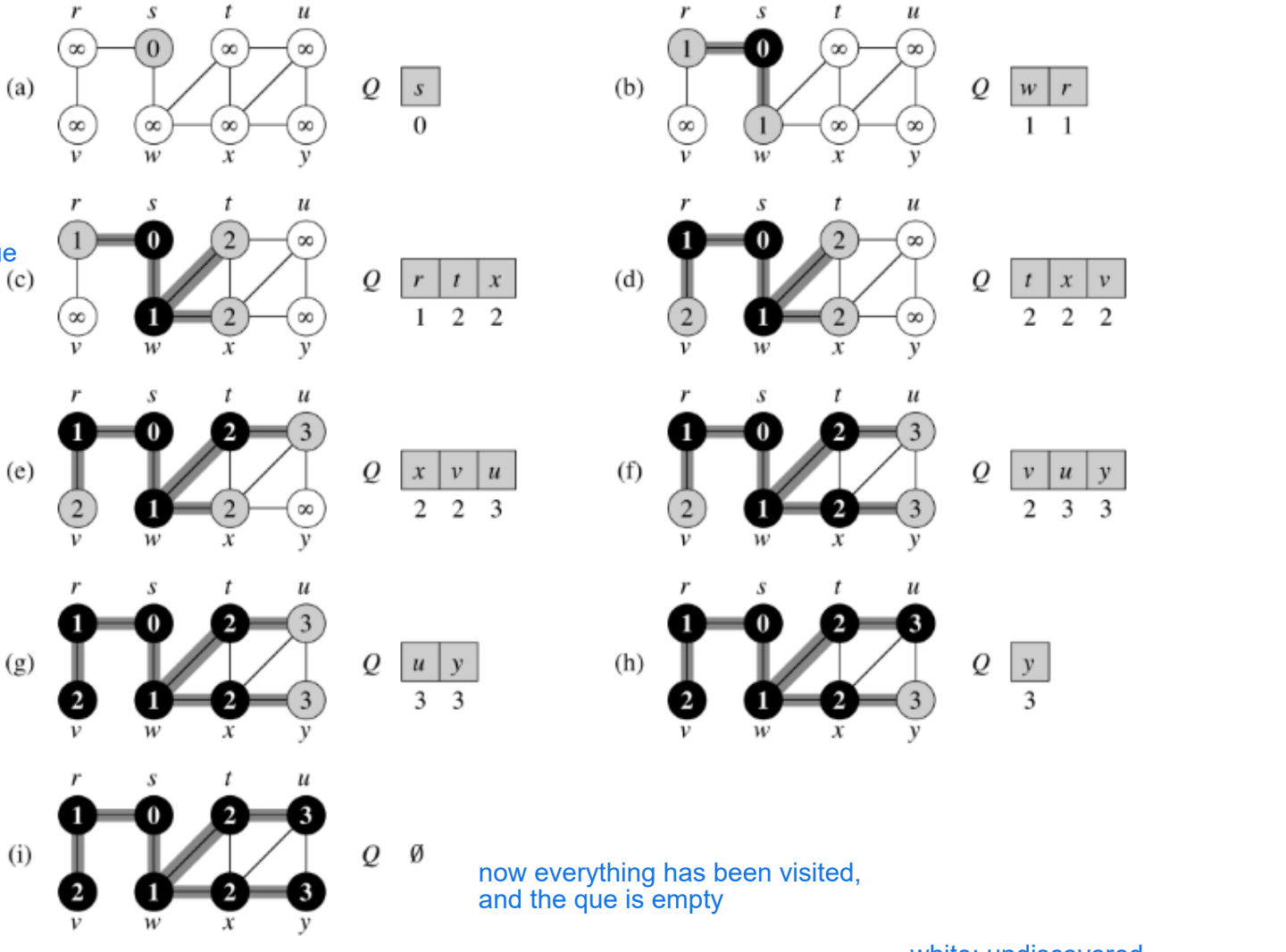
unorientated graph

Example:

start at s, look at is neighbors
are you white? yes, than put in que

becuase we looked at all edges it is black

look at first in que w,
if neighbors white,add to que



now everything has been visited,
and the que is empty

white: undiscovered
grey: disovered
black: exhausted the neighbors

Breadth-first search (BFS)

For BFS, the theorem about $\mathrm{GenericGraphTraversal3(s)}$ can be extended to also say something about the values of v.d:

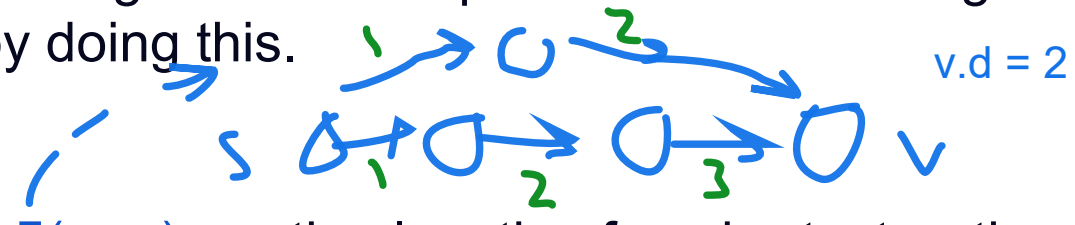
Theorem: The vertices that are discovered (made non-white) during a call to $\mathrm{GenericGraphTraversal3(s)}$ form a tree with s as the root and π in the discovered vertices as parent pointers.
For each path from a vertex v to the root in the tree, there exists the same path in the graph, but in the opposite direction (from s to v), and v.d equals the number of edges on this path.

Proof: It is easy to see that this statement is an invariant that is maintained during the execution of BFS(G, s).

Note: v.d is set at most once (after initialization to $-\infty$):
v.d is only set when v is white, and v is made non-white at the same time as v.d is set.

Characteristics of BFS

Running time: O(n + m).
The proof is the same as under GenericGraphTraversalGlobal, because the choices (*) in BFS takes O(1) time.
As mentioned, in BFS you often only call it once on a starting node s, without using the Global part. But the running time can only decrease by doing this.



Definition:
We define $\delta(s, v)$ as the length of a shortest path, measured in number of edges, from the starting vertex s to vertex v.
If no path exists, $\delta(s, v) = \infty$.

Theorem: when BFS stops, it holds that $v.d = \delta(s,v)$ for all vertices v.

In other words, BFS finds the shortest paths (measured by number of edges) from s to all v.

Depth-first search (DFS)

Strategy: Keep the gray nodes in a STACK, and advance minimally in their neighbor lists at each step.

The stack is implicit in the recursive formulation below (i.e., it is the recursion stack), but it can also be implemented explicitly. More precisely: The elements on the stack are the gray nodes, each with a partially traversed neighbor list — specifically, traversed in the for-loop in DFS-Visit.

(Note: the code on the left corresponds to the Global part in the terminology used earlier.) DFS also adds timestamps u.d for “discovery” (white → gray) and u.f for “finish” (gray → black) to every node u. (u.d is not "distance" in DFS.)

Running time: O(n + m).

Depth-first search (DFS)

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1  time = time + 1          // white vertex u has just been discovered
2  u.d = time
3  u.color = GRAY
4  for each v ∈ G.Adj[u]    // explore edge (u, v)
5      if v.color == WHITE
6          v.π = u
7          DFS-VISIT(G, v)  recursive call respondsible for the stack
8  u.color = BLACK          // blacken u; it is finished
9  time = time + 1
10 u.f = time               save it as finish time
```

at every color change we increase time

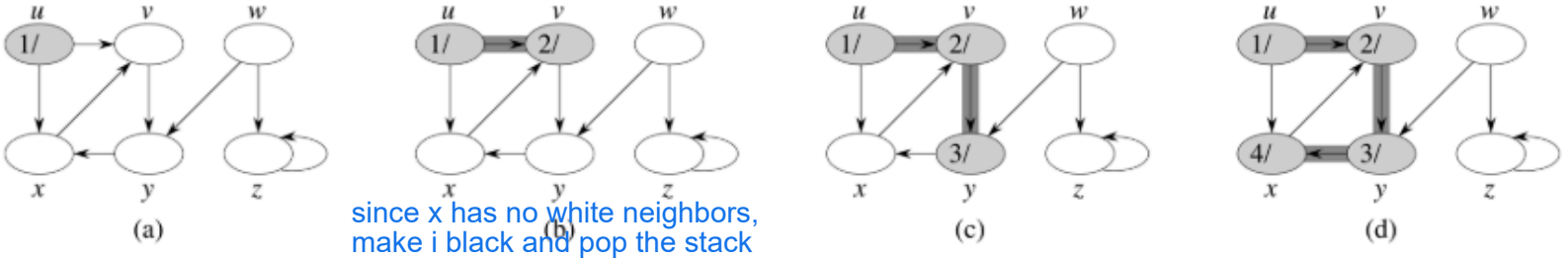
Depth-first search (DFS)

Example:

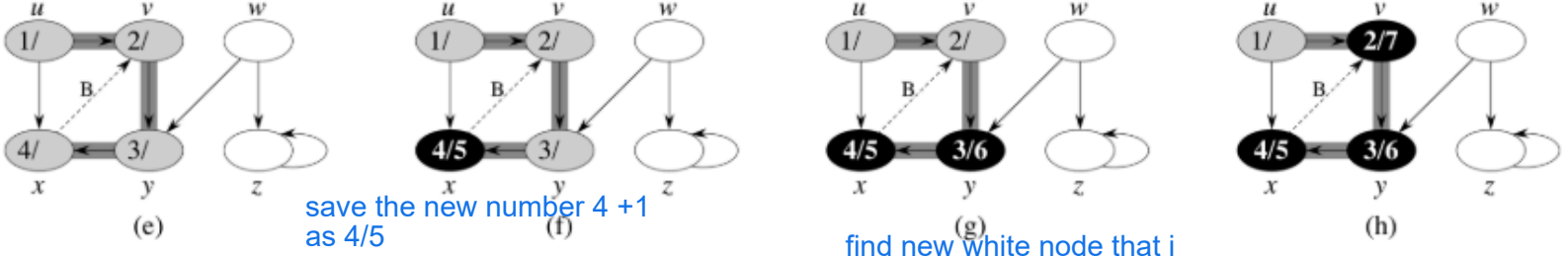
first number is discovery time n/

go to the first white neighbor
discovery time increases for v

v only has one edge,
that node is white, so
go to that one



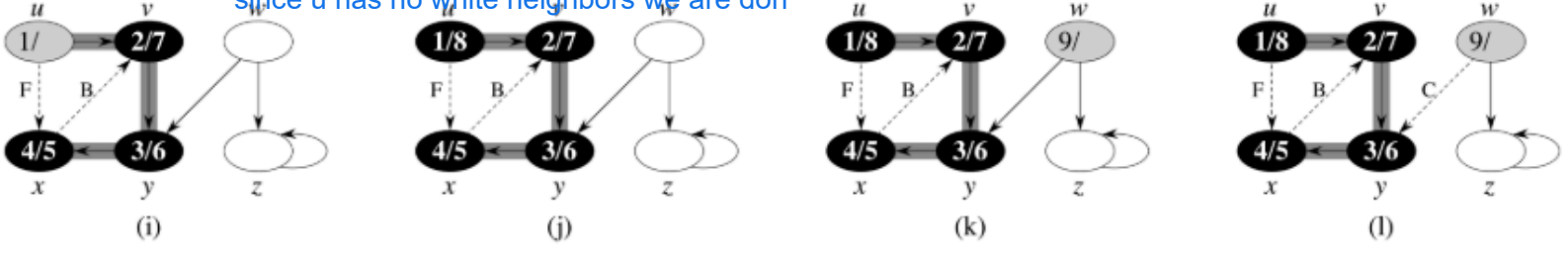
since x has no white neighbors,
make i black and pop the stack

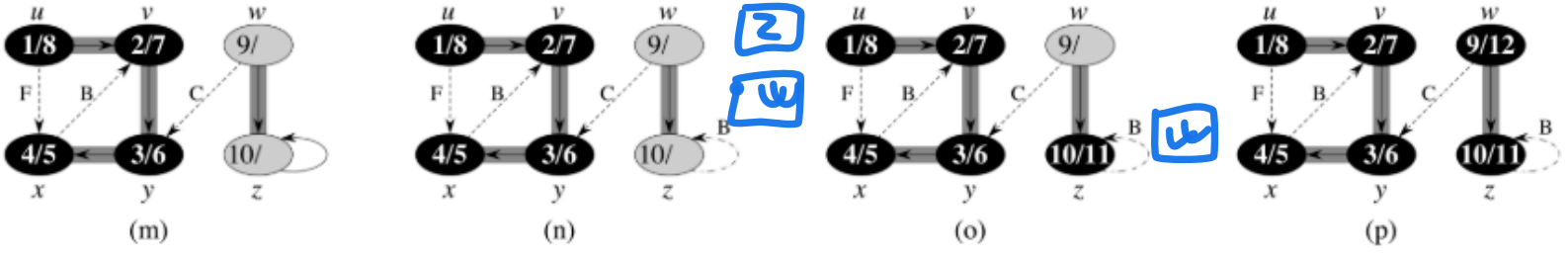


save the new number 4 +1
as 4/5

find new white node that i
s not connected

since u has no white neighbors we are don





Properties

Running time: O(n + m).
The proof is the same as for GenericGraphTraversalGlobal, since the choices (*) in DFS takes O(1) time.

Observe:

- Discovery (white → gray) of v = set v.d = call of DFS-Visit on v = PUSH of v onto the stack.
- Finish (gray → black) of v = set v.f = return from the call of DFS-Visit on v = POP of v from the stack.
- The value v.π is set when calling DFS-Visit on v.

Based on this, and the points above, it follows that:

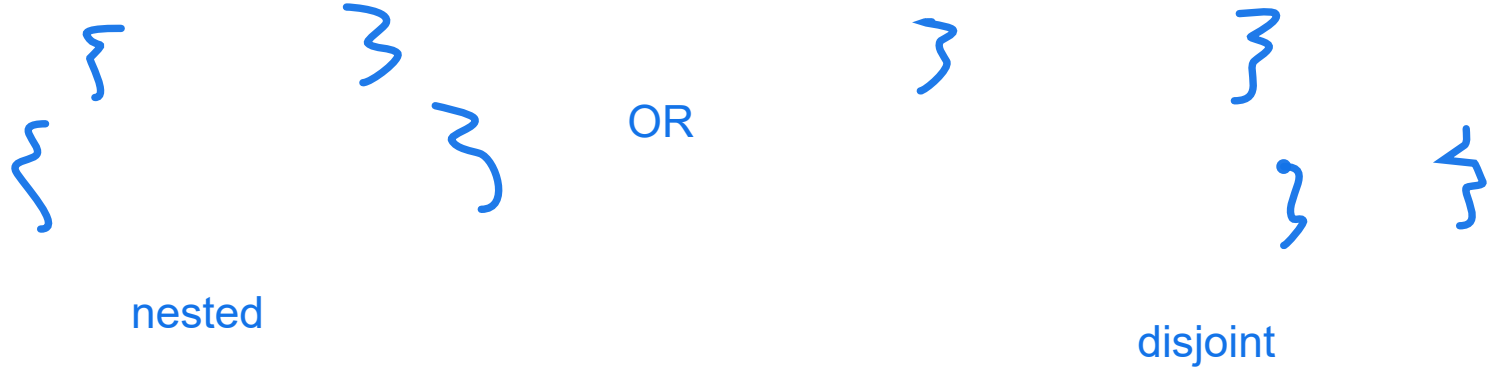
- The edges (v.π, v) exactly form the recursion trees for DFS-Visit (one tree for each call from DFS).
- The interval [v.d, v.f] is the period when v is on the stack.
- A node v is gray if and only if it is on the stack.

Properties

Because of the way a stack works:If two nodes u and v are on the stack at the same time, and v is on top, then v must be popped before u can be popped.

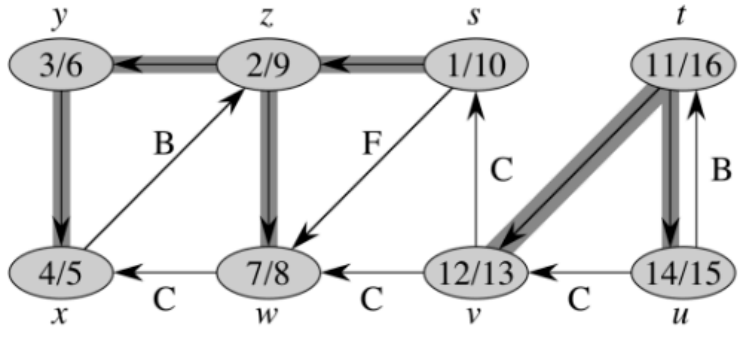
The interval [v.d, v.f] represents the period that v is on the stack. Therefore, for all pairs of nodes u and v, the intervals [u.d, u.f] and [v.d, v.f] must either: be disjoint (meaning u and v were never on the stack at the same time), or one interval must be completely contained within the other (meaning u and v were on the stack at the same time, and the node with the larger interval was placed on the stack first).

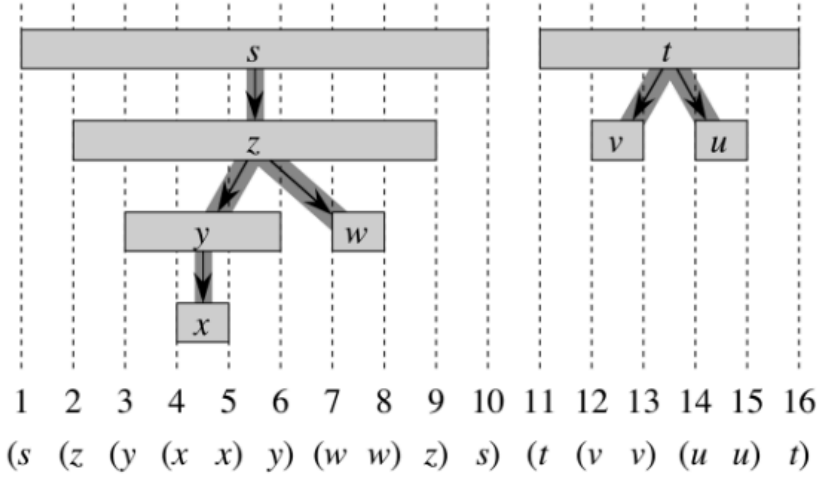
Discovery and finish times are therefore nested just like parentheses are.



Properties

Discovery and finish times are therefore nested just like parentheses are.





Properties

When an edge (u, v) is examined from u, we have the following cases:

1. Tree edges: v is white. (the fatter lines on the graph when we disover a white node
2. Back edges: v is gray (currently on the stack). B on the graph. when we have loop/cycle, that goes back in the tree
3. Forward edges: v is black (no longer on the stack, but was on the stack together with u). F on the graph
4. Cross edges: v is black (no longer on the stack and was not on the stack together with u). C on the graph