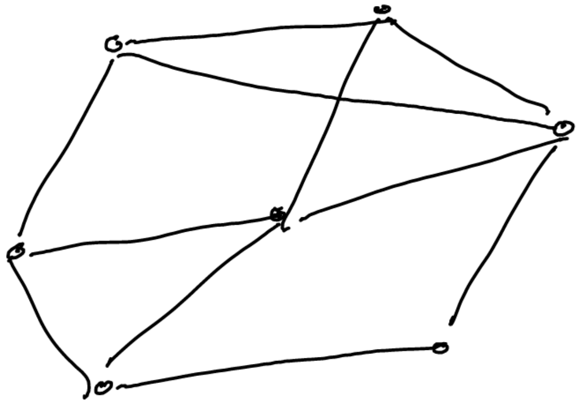
Graph Algorithms

Graphs

A graph is a pair $G = (V, E)$ of sets

- V is the set of vertices
- E is the set of edges
- Each element of E is a pair of vertices

Visually:



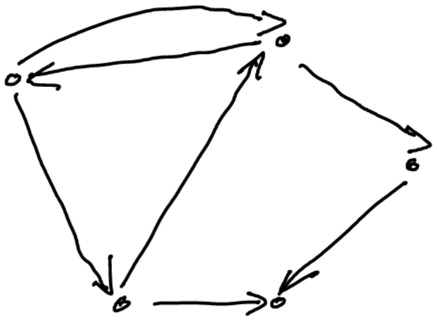
The endpoints v and w of an edge (v, w) are adjacent to each other and incident with the edge.

The degree of a vertex is the number of edges incident with it.

In an undirected graph, edges do not have directions. They are unordered pairs: $(v, w) = (w, v)$.

In a directed graph, $(v, w) \neq (w, v)$. Edge (v, w) is directed from v to w.

Visually:

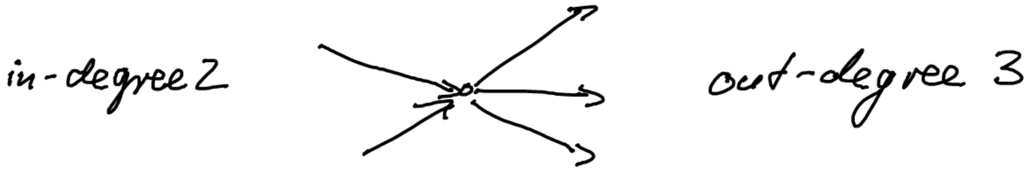


For a directed edge (v, w):

- (v, w) is an out-edge of v and an in-edge of w
- v is the tail of (v, w)
- w is the head of (v, w)

The out-degree of v is the number of edges that have v as their tails.

The in-degree of v is the number of edges that have v as their heads.



A path from v to w is a sequence of vertices $\langle u_1, u_2, \dots, u_k \rangle$ such that

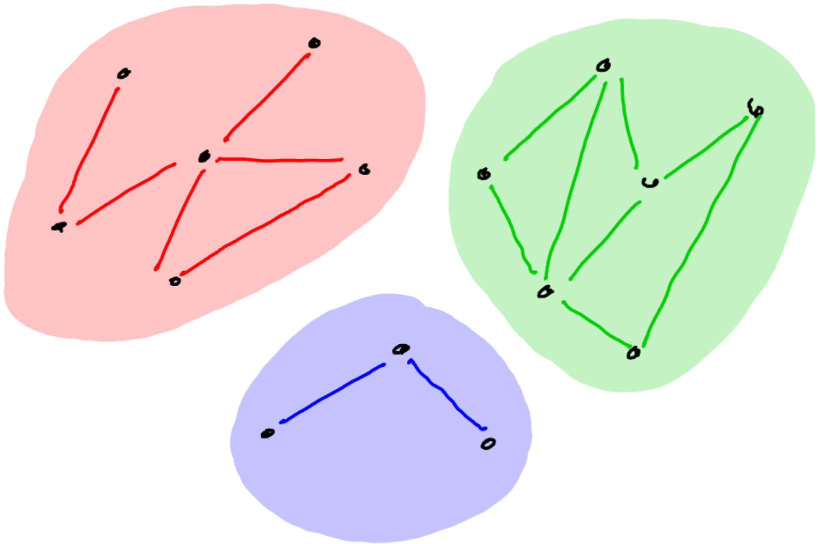
- $u_1 = v$
- $u_k = w$
- $\forall\, 1 \le i < k$: (u_i, u_{i+1}) is an edge of the graph

A cycle is a path $\langle u_1, u_2, \dots, u_k \rangle$ where $u_1 = u_k$.

A path or cycle is simple if it contains each vertex of G at most once.

A graph is connected if there exists a path between each pair of its vertices.

The connected components of a graph are its maximal connected subgraphs.



Graphs are everywhere

- Social networks
- Websites
- Road networks
- Object interactions in programs
- Phylogenetic trees
- ...

Graphs can be used to model relationships between entities:

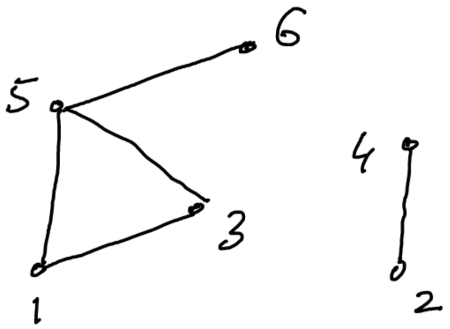
- Entities = vertices
- Related entities connected by an edge

Graph representations

We want:
- Little space
- Fast operations
  - Vertex insertion / deletion
  - Edge insertion / deletion
  - Degree queries
  - Adjacency queries
  - List all neighbours of a vertex

Adjacency matrix

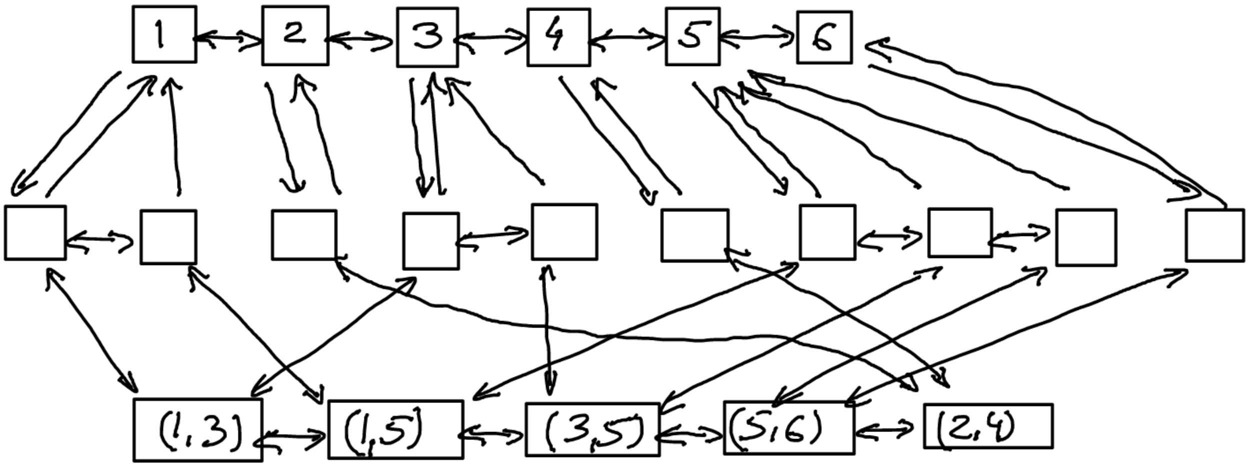
$$\begin{array}{c|cccccc|} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 2 & 0 & 0 & 0 & 1 & 0 & 0 \\ 3 & 1 & 0 & 0 & 0 & 1 & 0 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 \\ 5 & 1 & 0 & 1 & 0 & 0 & 1 \\ 6 & 0 & 0 & 0 & 0 & 1 & 0 \\ \hline \end{array}$$



- Vertex insertion / deletion: $\Theta(n^2)$ time
- Edge insertion / deletion: $O(1)$ time
- Degree queries: $\Theta(n)$ time ($O(1)$ time)
- Adjacency queries: $O(1)$ time
- List all neighbours: $\Theta(n)$ time

Space: $\Theta(n^2)$

Adjacency list



- Vertex insertion / deletion: $O(1)$ time
- Edge insertion / deletion: $O(1)$ time
- Degree queries: $\Theta(\deg(v))$ time $(O(1)$ time$)$
- Adjacency queries $\Theta(\min(\deg(v), \deg(w)))$
- List all neighbours: $\Theta(\deg(v))$ time

Space: $O(n+m)$

The space usage of adjacency matrices is prohibitive and most operations are slower.

$\Rightarrow$ Use adjacency lists throughout this course

Learning the structure of a graph

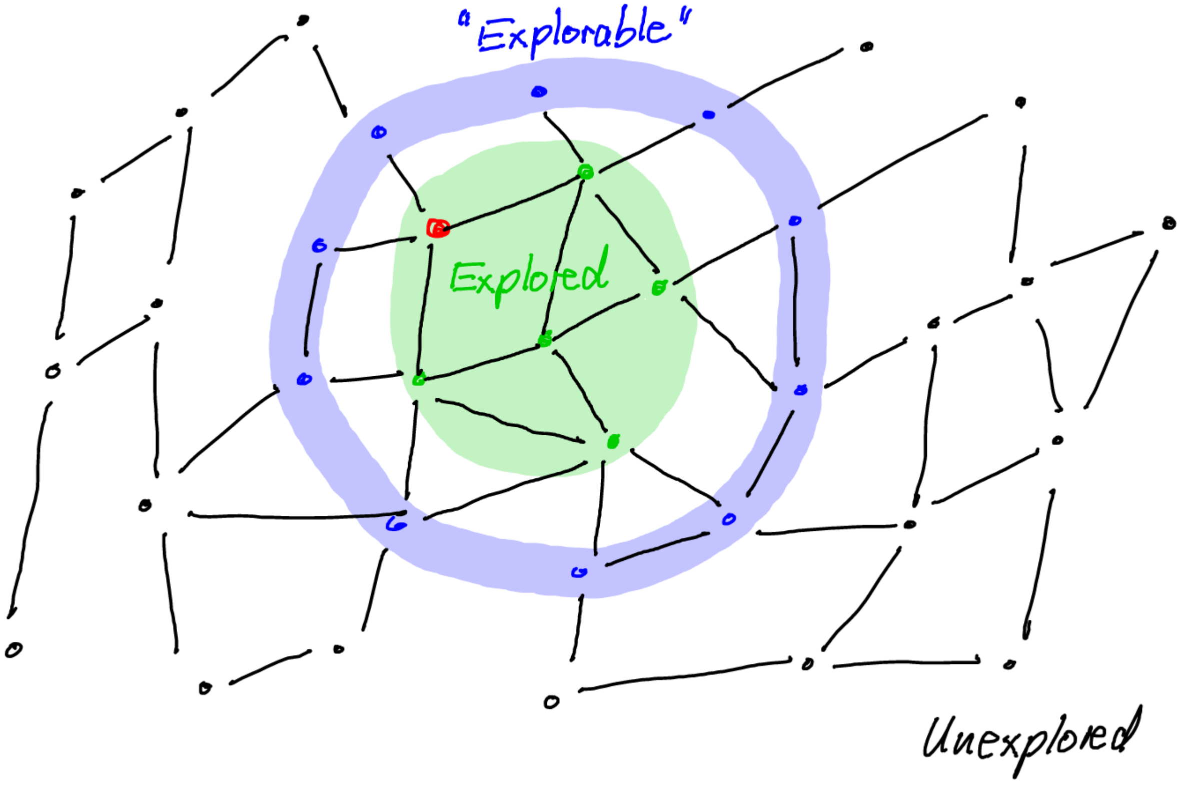
Solving graph problems = learning the graph's "structure", i.e., discovering connectivity properties (which vertex can reach which other vertex and how?)

Tool: Graph traversal

Start at some vertex and discover new vertices by following edges

```
TraverseGraph(G)
  Mark every vertex as unexplored                                 1
  for every vertex v of G do
      if v is unexplored then TraverseFromVertex(G, v)

TraverseFromVertex(G, v)                                          2
  Mark v as explored
  S = Adj(v)
  while S is not empty do
     Remove an edge (x, y) from S
     if y is not explored then                                    3
        Mark y as explored
        S = S ∪ Adj(y)
```



Variants of graph exploration

Depth-first search (DFS)

- S is a stack
- Add edges to S using Push operations
- Remove edges from S using Pop operations

Breadth-first search (BFS)

- S is a queue
- Add edges to S using Enqueue operations
- Remove edges from S using Dequeue operations

Dijkstra's algorithm / Prim's algorithm

- S is a priority queue
- Add edges to S using Insert operations
- Remove edges from S using DeleteMin operations
- Prim: priority = edge weight
- Dijkstra: priority = edge weight + dist(s, x)

By altering the representation of S, we change which structural properties we discover:

- Dijkstra / BFS: shortest paths
- Prim: minimum spanning tree
- DFS: harder to quantify but probably the most powerful strategy of them all

The cost of graph traversal

Lemma: If S supports insertions in $t_i(m)$ time and deletions in $t_d(m)$ time, then graph traversal takes $O(n + m(t_i(m) + t_d(m)))$ time.

Proof: Part 1 of the algorithm takes constant time per vertex, $O(n)$ in total.

The same holds for Part 2, excluding the addition of edges to S.

Every edge is added to S at most twice (once per endpoint) because we add $Adj(x)$ to S only if x is unexplored and we mark x as explored at the same time.

$\Rightarrow \leq 2m$ additions to S, cost: $2m\, t_i(m) = O(m\, t_i(m))$

We can only delete edges from S that were added. Each iteration of part 3 removes an edge from S.

$\Rightarrow \leq 2m$ deletions from S, cost: $2m\, t_d(m) = O(m\, t_d(m))$
$\leq 2m$ iterations of part 3, cost: $O(m)$

Total cost: $O(n) + O(m\, t_i(m)) + O(m\, t_d(m)) + O(m)$
$= O(n + m(t_i(m) + t_d(m)))$ $\square$

Corollary: DFS and BFS take $O(n+m)$ time. Dijkstra's and Prim's algorithm (as described here) take $O(n + m \log n)$ time.

Proof:

- Push / Pop on a stack take $O(1)$ time each $\Rightarrow t_i(m) = t_d(m) = O(1)$ for DFS
- Enqueue / Dequeue on a queue take $O(1)$ time each $\Rightarrow t_i(m) = t_d(m) = O(1)$ for BFS
- A binary heap supports Insert / DeleteMin operations in $O(\log m)$ time $\Rightarrow t_i(m) = t_d(m) = O(\log m)$ for Prim and Dijkstra. However, $m \le n^2$.
  $\Rightarrow \log m \le \log n^2 = 2 \log n$, so
  $t_i(m) = t_d(m) = O(\log n)$. □

We'll discuss Prim and Dijkstra in more detail as examples of greedy algorithms. For the remainder of this topic, we'll focus on BFS and DFS, that is, on what types of structural properties of a graph we can compute in linear time.

Computing connected components

Lemma: Let $C_1, C_2, \dots, C_k$ be the connected components of G. For $1 \le i \le k$, let v_i be the first vertex in C_i. Vertex v_i is unexplored when TraverseGraph inspects v_i and the resulting call TraverseFromVertex(G, v_i) explores exactly the vertices in C_i.

Proof: By induction on i.

For $i=1$, v_1 is the first vertex of G and all vertices of G are unexplored when TraverseGraph inspects v_1. In particular, all vertices of C_1 are unexplored. We show below that this implies that the call TraverseFromVertex(G, v_1) explores exactly the vertices of C_1.

For $i > 1$, the calls TraverseFromVertex(G, v_1), …, TraverseFromVertex(G, v_{i-1}) explore exactly the vertices of $C_1, \dots, C_{i-1}$, by the induction hypothesis. Moreover, there is no vertex $u \notin \{v_1, \dots, v_{i-1}\}$ such that a call TraverseFromVertex(G, u) is made before v_i is inspected. Indeed, if $u \in C_j$, for some j, and u precedes v_j, this contradicts the choice of v_j. If u succeeds v_j, then $j < i$ because u precedes v_i. Since the call TraverseFromVertex(G, v_j) explores exactly the vertices of C_j, u is explored by the time TraverseGraph inspects it. Thus, the call TraverseFromVertex(G, u) is not made.

Now, since TraverseFromVertex(G, v_1), …, TraverseFromVertex(G, v_{i-1}) explore exactly the vertices of $C_1, \dots, C_{i-1}$ and these are the only calls made before v_i is inspected, the vertices of C_i are unexplored when v_i is inspected. As for the case when $i=1$, we show that this implies that TraverseGraph(G, v_i) explores exactly the vertices of C_i.

So assume the vertices of C_i are unexplored when the call TraverseFromVertex(G, v_i) is made. We show first that it explores only vertices in C_i. Assume the contrary and let u be the first vertex explored by TraverseFromVertex(G, v_i) that is not in C_i. Then $u \neq v_i$ and u is explored as the result of removing an edge (w, u) from S. This edge is inserted into S when w is explored, so w is explored before u and thus belongs to C_i. Since u is adjacent to w, this shows that $u \in C_i$ $\lightning$.

Now assume $\exists$ vertex $u \in C_i$ that is not explored by TraverseFromVertex(G, v_i). Since $u \in C_i$, there exists a path $P = \langle x_1, x_2, \dots, x_t \rangle$ from v_i to u, that is, $x_1 = v_i$ and $x_t = u$. Let x_j be the first vertex in P that is not explored by TraverseFromVertex(G, v_i). Such a vertex must exist because u is not explored. Moreover, $j > 1$ because $x_1 = v_i$ is explored right at the beginning of TraverseFromVertex(G, v_i). When TraverseFromVertex(G, v_i) explores x_{j-1}, it inserts edge (x_{j-1}, x_j) into S. When it removes this edge from S, x_j is unexplored because $x_j \in C_i$ and thus is unexplored when the call TraverseFromVertex(G, v_i) is made and this call does not explore x_j. This, however, implies that TraverseFromVertex(G, v_i) does explore x_j when it removes the edge (x_{j-1}, x_j) from S $\lightning$

$\square$

Corollary: Graph traversal can be used to compute the connected components of G.

Proof: More precisely, we want to assign a label $C(v)$ to every vertex v such that $C(v) = C(w)$ iff v and w belong to the same component. By the previous lemma, the following augmented graph traversal procedure computes such a labelling.

```
TraverseGraph(G)
  Mark every vertex as unexplored
  $c = 1$
  for every vertex v of G do
      if v is unexplored then
          TraverseFromVertex(G, v, c)
          $c = c + 1$

TraverseFromVertex(G, v, c)
  Mark v as explored
  $C(v) = c$
  $S = Adj(v)$
  while S is not empty do
      Remove an edge (x, y) from S
      if y is not explored then
          Mark y as explored
          $C(y) = c$
          $S = S \cup Adj(y)$
```

□

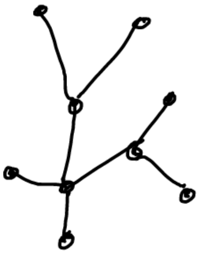
Corollary: The connected components of a graph can be computed in $O(n+m)$ time.

Proof: The modifications we made to the graph traversal procedure add only $O(n)$ to its running time. Since BFS and DFS both take $O(n+m)$ time, we can use either to compute connected components in $O(n+m)$ time. □

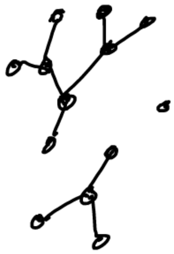
Computing a spanning tree (forest)

A **tree** is a connected graph without cycles.
A **forest** is a graph whose connected components are trees.

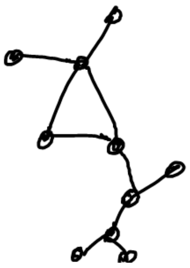
A tree:



A forest but not a tree:

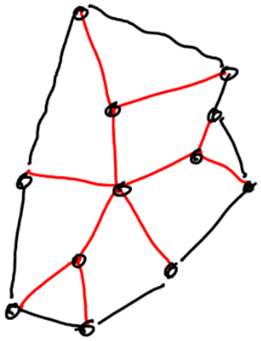


Not a tree nor a forest:

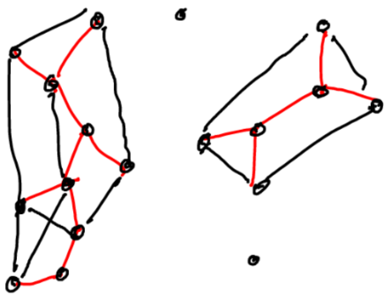


A **spanning tree** of a connected graph G is a tree $T \subseteq G$ that contains all vertices of G.

A **spanning forest** of a graph G is a forest $F \subseteq G$ that contains all vertices of G and has the same number of connected components of G.



A spanning tree



A spanning forest

Lemma: A spanning forest (tree) of a (connected) graph can be computed using graph traversal.

Proof: We augment the procedure as follows:

```
TraverseGraph(G)
  Mark every vertex as unexplored
  Mark every edge as a non-tree edge
  for every vertex v of G do
      if v is unexplored then TraverseFromVertex(G, v)

TraverseFromVertex(G, v)
  Mark v as explored
  S = Adj(v)
  while S is not empty do
      Remove an edge (x, y) from S
      if y is not explored then
          Mark y as explored
          S = S ∪ Adj(y)
          Mark edge (x, y) as a tree edge
```

The set of tree edges form a spanning forest T if and only if

- Two vertices x and y can reach each other in T iff they can reach each other in G (G and T have the same number of connected components).
- T has no cycle.

We start with an observation:

Observation: Every edge in S has an explored endpoint, namely the vertex that added the edge to S.

Corollary: When an edge is labelled as a tree edge, both its endpoints are explored.

Proof: An edge (x,y) is labelled as a tree edge after removing it from S and only if y is unexplored. By the above observation, x must be explored at this time and immediately before marking (x,y) as a tree edge, we mark y as explored. $\square$

T has no cycle: Assume it does and consider the last edge e added to the cycle C. Both endpoints of e already have an incident edge in C at the time e is added.

Since both endpoints of a tree edge are explored, both endpoints of e are therefore explored when we mark e as a tree edge. However, we mark an edge as a tree edge only if it has an unexplored endpoint at the time we remove it from S. ↯

Two vertices are connected in T iff they are in G.

Since $T \subseteq G$, two disconnected vertices in G must be disconnected in T. Thus, it suffices to prove that any two vertices in the same component of G can reach each other in T.

Let $C_1, \dots, C_k$ once again be the connected components of G, and let v_i be the first inspected vertex of each component C_i. It suffices to prove that every vertex in C_i can reach v_i.

We have shown that the call TraverseFromVertex(G, v_i) explores all vertices in C_i. Now assume that there exists a vertex in C_i that cannot reach v_i in T and choose u to be the vertex explored first among all such vertices. Then $u \neq v_i$ and u is explored after removing an edge (w, u) from S. This edge is inserted into S when w is explored. Thus, w is explored before u and thus can reach v_i in T, by the choice of u. Since we make (w,u) a tree edge when exploring u, u can thus also reach v_i in T. $\lightning$

$\square$

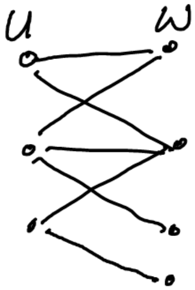
Corollary: A spanning forest (tree) of a (connected) graph G can be computed in $O(n+m)$ time.

Proof: Use BFS or DFS. $\square$

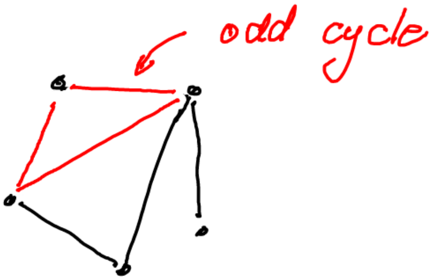
Testing bipartiteness

A graph $G = (V, E)$ is **bipartite** if $V = U \cup W$, $U \cap W = \emptyset$ and every edge e in E has an endpoint in U and one in W.

Bipartite:



Not bipartite:



Lemma: A graph is bipartite if and only if it does not contain an odd cycle.

Proof: Assume the graph contains an odd cycle $\langle w_1, w_2, \ldots, w_k \rangle$ s.t. w.l.o.g. $w_1 \in U$. Then $w_2 \in W$, $w_3 \in U, \ldots$ In general $w_i \in U$ for odd i and $w_i \in W$ for even i. Since k is odd, we have $w_1, w_k \in U$, a contradiction because $(w_1, w_k) \in E$.

Assume all cycles are even. Let T be a spanning tree of G. Choose an arbitrary root r, add all vertices at even depth in T to U and all vertices at odd depth to W. Clearly every edge of T has one edge in U and one in W. If every edge in $G \setminus T$ has one endpoint in U and on in W, then G is bipartite, so assume $\exists (x,y) \in E$ s.t. w.l.o.g. $x, y \in U$. Then x and y are at even depths in T, so the path from x to y has an even number of edges. Adding (x,y) to this path, we obtain an odd cycle. ↯ $\square$

Lemma: In a BFS tree T (spanning tree computed using BFS) of a connected undirected graph G, the endpoints of every edge (v,w) of G satisfy $|\text{dist}_T(r,v) - \text{dist}_T(r,w)| \leq 1$, where r is the root of T (the vertex for which we call BFSFromVertex(G,r)) and $\text{dist}_T(r,x)$

is the number of edges on the unique shortest path from r to x in T.

Can you prove this path is unique?

Proof: We claim that the vertices of G are explored by increasing distance in T. Assume the contrary. Then there exists a vertex pair (x,y) such that $\mathrm{dist}_T(r,x) < \mathrm{dist}_T(r,y)$ and y is explored before x. We choose this pair such that $\mathrm{dist}_T(r,y)$ is minimized. Let p_x and p_y be the parents of x and y, respectively, in T. Then $\mathrm{dist}_T(r,p_x) < \mathrm{dist}_T(r,p_y)$. By the choice of the pair (x,y), p_x is explored before p_y, so the edge (p_x,x) is inserted into S before (p_y,y). Since S is a queue (we're using BFS), this implies we remove (p_x,x) from S before (p_y,y), so x is visited before y. ↯ A special case we haven't considered yet is when $x=r$ and thus x has no parent. In this case, we immediately obtain a contradiction because r is the very first vertex we explore.

Now we are ready to prove the lemma: Assume there are two vertices x and y such that $\mathrm{dist}_T(r,y) > \mathrm{dist}_T(r,x)+1$ and (x,y) is an edge of G. Then $\mathrm{dist}_T(r,p_y) > \mathrm{dist}_T(r,x)$, so p_y is explored after x and the edge (p_y,y) is inserted into S after (x,y). Since p_y is y's parent, y is unexplored when we remove (p_y,y) from S. Since (x,y) is removed from S before (p_y,y), y is also unexplored when we remove (x,y) from S. Thus, we would have made x y's parent. ↯ □

The above lemma holds for undirected graphs but not for directed ones. Can you see why?

Corollary: Let G be an undirected graph and T a BFS tree of G with root r. Then G is bipartite iff there is no edge (x,y) in G such that $dist_T(r,x) = dist_T(r,y)$.

Proof: If there exists such an edge (x,y), then let $P_x = \langle u_0, u_1, \dots, u_d \rangle$ and $P_y = \langle v_0, v_1, \dots, v_d \rangle$ be the paths from r to x and y in T. Since both paths start at r, we have $u_0 = v_0$, so there exists an index h such that $u_i = v_i \ \forall\, 0 \le i \le h$ and $u_i \ne v_i \ \forall\, h < i \le d$. The latter follows because T is a tree. Now $\langle u_i, \dots, u_d, v_d, v_{d-1}, \dots, v_i \rangle$ is an odd cycle in G, so G is not bipartite.

Now assume that $dist_T(r,x) \ne dist_T(r,y)$ for all (x,y) in T. By the previous lemma, this implies that $dist_T(r,y) - dist_T(r,x) \in \{-1, 1\}$. Now consider any cycle $C = \langle x_1, x_2, \dots, x_k \rangle$ in G. Let $e_1, e_2, \dots, e_{k-1}$ be its edges, that is, $e_i = (x_i, x_{i+1}) \ \forall\, 1 \le i \le k-1$. We assign a weight $w(e_i) = dist_T(r, x_{i+1}) - dist_T(r, x_i)$ to each edge e_i. Note that $w(e_i) \in \{-1, 1\} \ \forall\, 1 \le i \le k-1$. Also, note that $dist_T(r, x_k) = dist(r, x_1) + \sum_{i=1}^{k-1} w(e_i)$. But $x_1 = x_k$, so $dist_T(r, x_k) = dist(r, x_1)$ and $\sum_{i=1}^{k-1} w(e_i) = 0$. Since $w(e_i) \in \{1, -1\} \ \forall i$, this implies that there are as many edges with weight 1 as there are edges with weight -1

in C. Thus, C has an even number of edges. Since we chose C arbitrarily, this shows that every cycle of G is even, that is, G is bipartite. □

Corollary: It takes $O(m+n)$ time to test whether a graph G is bipartite.

Proof: We augment BFS as follows, which adds only $O(n+m)$ to its running time.

```
BFS(G)
Mark every vertex as unexplored
for every vertex v of G do
    if v is unexplored then TraverseFromVertex(G, v)
for every edge (v, w) of G do
    if $d(v) = d(w)$ then return false
return true

BFSFromVertex(G, v)
Mark v as explored
$d(v) = 0$
S = Adj(v)    // S is a queue here
while S is not empty do
    Remove an edge (x, y) from S
    if y is not explored then
        Mark y as explored
        $d(y) = d(x) + 1$
        $S = S \cup Adj(y)$
```

□

We could in fact have used any graph exploration strategy and the spanning tree T it produces because it is not hard to extend the above argument to show that G is bipartite iff $\forall (x,y) \in E$, $\mathrm{dist}_T(r,x) - \mathrm{dist}_T(r,y)$ is odd.

Topological sorting

A topological ordering of a graph G is a total order $\prec$ defined over the vertices of G such that $v \prec w$ for every edge (v,w).

Why do we care?

→ Vertices are activities, tasks, ... Edges are ordering constraints. We want a sequential order of these activities that satisfies all ordering constraints.

Does such an ordering always exist?

Lemma: A directed graph G has a topological ordering if and only if it has no directed cycles.

Proof: Assume G has a directed cycle C and admits a topological ordering $\prec$. Since G is finite, so is C and there exists a maximal vertex $v \in C$, that is, $u \prec v$ $\forall u \in C, u \neq v$. In particular $w \prec v$, for v's successor w in C, a contradiction.

Now assume G has no directed cycle. If G has only one vertex, the only possible ordering of its vertex set is trivially a topological ordering.

If G has $n>1$ vertices, it must have a source, that is, a vertex of in-degree 0. Assume the contrary. We construct a cycle in G, which contradicts our assumption that G has no cycles. Pick an arbitrary vertex x_0 and define $P_0 = \langle x_0 \rangle$. Given a path $P_i = \langle x_i, x_{i-1}, \dots, x_0 \rangle$, we try to extend it by adding an in-neighbour x_{i+1} of x_i to it to obtain a new path $P_{i+1} = \langle x_{i+1}, x_i, \dots, x_0 \rangle$. If $x_{i+1} \in P_i$, that is, $x_{i+1} = x_j$, for some $0 \le j \le i$, then $\langle x_{i+1}, x_i, \dots, x_j \rangle$ is a cycle in G. Since G is finite, this will happen eventually because we run out of vertices not in P_i yet. Thus, G must contain a source.

Let $G' := G - s$, for some source s of G. Since $G' \subset G$, G' is acyclic. Thus, by the inductive hypothesis, there exists a topological ordering $\prec'$ of G'. Since s has no in-neighbours of G, we can extend $\prec'$ to a topological ordering $\prec$ of G by defining

$$u \prec v \text{ iff } \begin{array}{l} s \notin \{u,v\} \text{ and } u \prec' v \text{ or} \\ u = s \text{ and } v \neq s. \end{array}$$

□

We discuss 2 ways to topologically sort a DAG.

Solution 1: Use DFS

First thing we need is a recursive implementation of DFS because it makes it explicit when we're done exploring a subtree:

```
DFS(G)
Mark every vertex of G as unexplored
for every vertex v of G do
    if v is unexplored then DFSFromVertex(G,v)

DFSFromVertex(G,v)
Mark v as explored
for every edge $(v,w) \in Adj(v)$ do
    if w is unexplored then DFSFromVertex(G,w)
```

Observation: When DFSFromVertex(G,v) returns, all out-neighbours of v are explored.

This suggests a strategy for topological sorting: Number vertices in decreasing order and number each vertex v when DFSFromVertex(G,v) is about to return because then – not completely obvious – all its out-neighbours should have been numbered already and thus have a higher number. This gives the following code:

```
DFS(G)
Mark every vertex as unexplored
c = n
for every vertex v of G do
    if v is unexplored then c = DFSFromVertex(G, v, c)

DFSFromVertex(G, v, c)
Mark v as explored
for every edge (v, w) ∈ Adj(v) do
    if w is unexplored then c = DFSFromVertex(G, w, c)
number(v) = c
return c - 1
```

As already said, it is not entirely obvious that this does indeed produce a topological ordering. While all out-neighbours of v are explored when DFSFromVertex(G, v) returns, they may be numbered much later than they are marked as explored, so they may not all be numbered when DFSFromVertex(G, v) returns. This should not come as a surprise. The above observation holds for every directed graph G, while we already proved that we can find a topological ordering only if G is acyclic. So the correctness proof better use this property of G.

Lemma: If G is acyclic, then the above modification of DFS computes a topological ordering of G.

Proof: It suffices to show that, for each vertex v of G, all out-neighbours of v are numbered before

v is numbered, that is, before, DFSFromVertex(G,v) returns. Indeed, since we number vertices backwards and it is easy to see that we do not use the same number twice, this implies that all out-neighbours of v receive a higher number than v.

Consider the recursion tree of the algorithm. The root is DFS(G). Its children are all invocations DFSFromVertex(G,v) made by DFS(G). The children of an invocation DFSFromVertex(G,v) are the invocations DFSFromVertex(G,w) it makes. Now observe that, before an invocation DFSFromVertex(G,v) is made, v is unexplored; after the invocation returns v is explored and numbered. Thus, a vertex v can be explored but unnumbered only during descendant invocations DFSFromVertex(G,w) of DFSFromVertex(G,v) because the invocation DFSFromVertex(G,v) must have been made but must not have returned yet.

Now assume v has an unnumbered out-neighbour w at the time DFSFromVertex(G,v) returns. As we just argued, DFSFromVertex(G,w) must be an ancestor invocation of DFSFromVertex(G,v). Thus, there exist vertices $w = x_0, x_1, \ldots, x_k = v$ such that DFSFromVertex(G,x_i) makes the recursive call DFSFromVertex(G,x_{i+1}) and hence (x_i, x_{i+1}) is an edge of G, for all $0 \le i < k$. Since w is an out-neighbour of v, (x_k, x_0) is also an edge of G, so $\langle x_0, x_1, \ldots, x_k, x_0 \rangle$ is a directed cycle of G.

↯ □

Corollary: A topological ordering of a DAG G can be computed in $O(n+m)$ time.

Exercise: Augment the algorithm so that it tests whether the graph is acyclic. If so, output a valid topological ordering as proof. If not, output a cycle as proof.

Solution 2: Incrementally number sources

In our proof that a graph has a topological ordering iff it is acyclic, we proved two useful facts:

- Every acyclic graph has a source (in-degree-0 vertex).
- Every subgraph of an acyclic graph is acyclic.

We can use this to design a really simple algorithm for topological sorting.

```
TopSort(G)
Set in-deg(v) = 0 for every vertex v of G
for every edge (v,w) of G do
    in-deg(w) = in-deg(w) + 1
Q = an empty queue
for every vertex v of G do
    if in-deg(v) = 0 then Enqueue(Q,v)
c = 1
while Q is not empty do
    v = Dequeue(Q)
```

```
    numbers(v) = c
    c = c + 1
    for every edge (v, w) ∈ Adj(v) do
        in-deg(w) = in-deg(w) - 1
        if in-deg(w) = 0 then Enqueue(Q, w)
```

Lemma: The above TopSort procedure computes a topological ordering of G in $O(n+m)$ time.

Proof: To prove the correctness, it suffices to prove that Q contains exactly the vertices that are unnumbered and have only numbered in-neighbours. Indeed, this implies that picking any vertex in Q to give the next number to, as we do, ensures this vertex has a higher number than all its in-neighbours. Since the subgraph induced by unnumbered vertices is acyclic, it contains at least one source. Thus, Q is non-empty until all vertices are numbered. Thus, the algorithm assigns a number to each vertex and, as we just argued, the numbering satisfies all edges of G. Therefore, the final numbering represents a valid topological ordering. It remains to prove our claim.

Initially, all vertices are unnumbered and we initialize Q to contain all sources of G, so the invariant holds. Each iteration removes a vertex v from Q, numbers it, and decrements the in-degree of each of its out-neighbours, which ensures that the in-degree of each vertex w really represents the number of its un-numbered in-neighbours. If this

numbes drops to 0, w is added to Q.
Now we make three simple observations to finish the proof:

(i) No numbeed vetex is added to Q again. Indeed, we add only out-neighbours of v to Q. Since v was unnumbeed before, no such vetex can have been in Q before, that is, every out-neighbow of v is unnumbeed.

(ii) An out-neighbow of v is added to Q iff it has only numbeed in-neighbours now. This is obvious because we add an out-neighbow of v to Q iff its in-degree is 0.

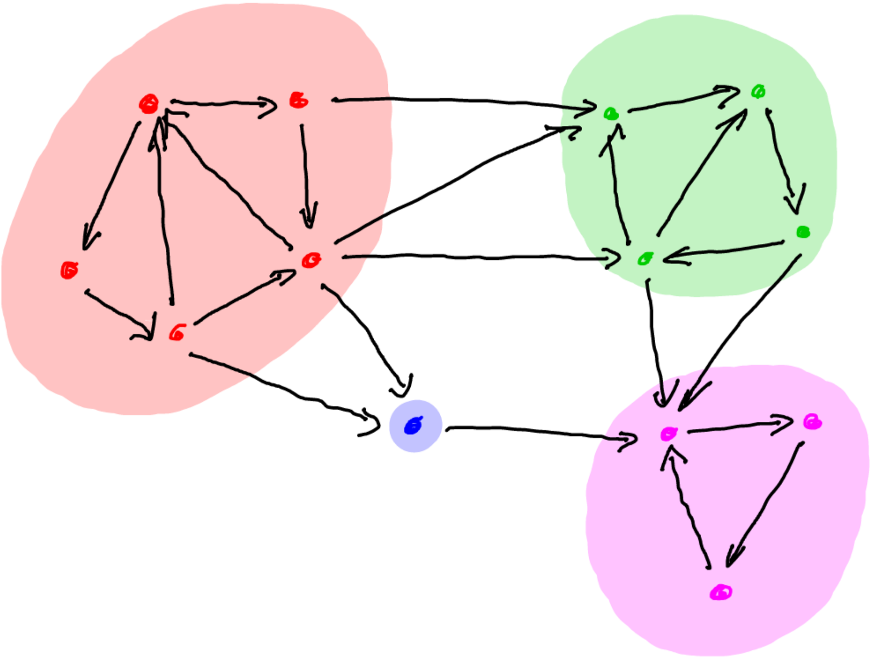
(iii) Any othes vetex with only numbeed in-neighbours is already in Q because v is the only new numbeed vetex and is not an in-neighbow of such a vetex.

Having proven the correctness of the algorithm, it remains to analyze its running time. The initialization before the main while-loop is easily seen to take $O(n+m)$ time because it consists of two loops over the vetices and one loop over the edges of G, with constant work pes iteration. Each iteration of the while-loop, including the nested for-loop spends $O(1+\text{out-deg}(v))$ time, where v is the dequeued vestex. We already argued that every vetex gets enqueued exactly once, so it also gets dequeued exactly once. Thus, the cost of all iterations of the while-loop is

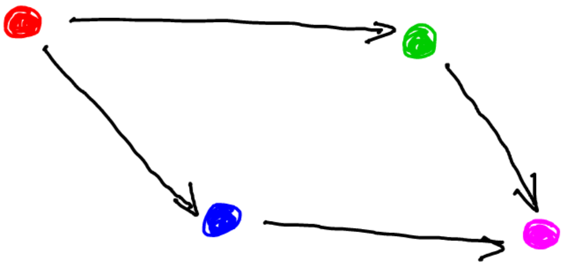
$$\sum_{v \in G} O(1+\text{out-deg}(v)) = O(n+m). \qquad \square$$

Strongly connected components

A directed graph is strongly connected if for every ordered pair of vertices (v,w), there exists a directed path from v to w. The strongly connected components of a directed graph are its maximal strongly connected subgraphs.



Strong connectivity and acyclicity are opposite concepts. In particular, if we contract each strongly connected component into a single vertex, we obtain a DAG.



We can compute the strongly connected components using a simple DFS-like algorithm that can in fact be implemented using DFS.

The algorithm maintains a partition of the vertices into three groups (represented by labelling them accordingly):

Finished vertices are vertices whose strongly connected components have already been identified. In particular, their out-neighbours are all finished.

Active vertices are explored vertices that may have unexplored out-neighbours in the same strongly connected component.

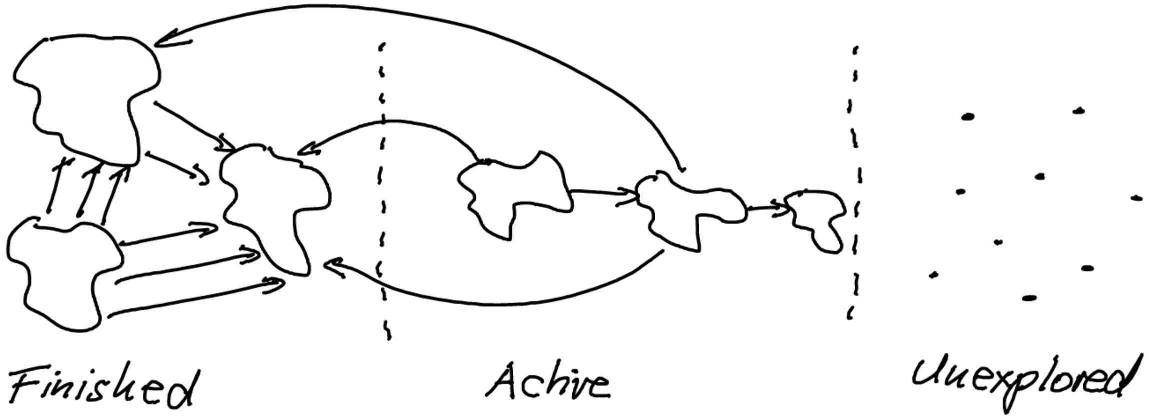
Unexplored vertices are, well, unexplored.

Even though the algorithm doesn't maintain this classification explicitly, it is helpful to distinguish explored edges (which we have followed already) from unexplored ones (which we have not followed yet).

The key invariant the algorithm maintains is:

(i) All out-edges of finished vertices are explored. Their heads are finished.

(ii) The subgraph defined by active vertices and explored edges between them is a "path of strongly connected components". More precisely, if $C_1, C_2, \dots, C_k$ are the strongly connected components of this subgraph, then the only edges not in these components are edges $(u_1, v_1), (u_2, v_2), \dots, (u_{k-1}, v_{k-1})$, where $u_i \in C_i$ and $v_i \in C_{i+1}$ $\forall\, 1 \le i \le k-1$.



Now the algorithm does the following until all vertices are finished:

- If there is no active vertex, choose an arbitrary unexplored vertex and make it active.
- If $C_1, C_2, \dots C_k$ are the current active components, choose an out-edge (x, y) of C_k that is unexplored.
    - If no such edge exists, mark all vertices in C_k as finished and mark C_k as a strongly connected component of G.
    - If y is unexplored, create a new component C_{k+1} with only y in it.
    - If y is active and $y \in C_i$, then merge $C_{i+1}, C_{i+2}, \dots, C_k$ into C_i.
    - If y is finished, do nothing.

Lemma: The algorithm maintains the stated invariant.

Proof: If the component C_k has no unexplored out-edges, then all out-neighbours of C_k are finished, by the invariant. Thus, marking all vertices in C_k as finished does not violate the invariant that finished vertices have only finished out-neighbours and only explored out-edges. Since the vertices in $C_1, C_2, \dots, C_k$ and the explored edges between them define a path of strongly connected components, so do the vertices in $C_1, C_2, \dots, C_{k-1}$ and the explored edges between them. This shows that the invariant is maintained when C_k has no unexplored out-edges.

If w is unexplored, making it active and adding the component C_{k+1} with only w in it maintains the invariant that $C_1, C_2, \dots, C_{k+1}$ form a path of strongly connected components. No new finished vertices are created, nor do any revert to an unfinished state. So the finished vertices continue to have only explored out-edges and only finished out-neighbours.

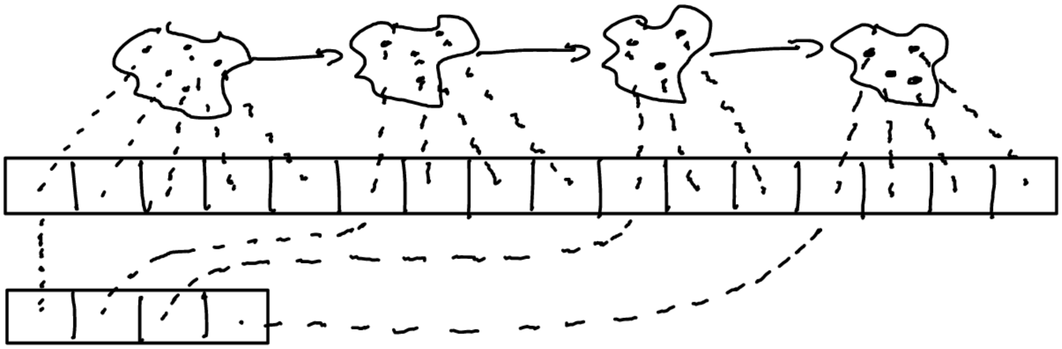
If w is active and belongs to C_i, the explored edges between vertices in $C_i, C_{i+1}, \dots, C_k$ define a strongly connected graph C_i'. $C_1, C_2, \dots, C_{i-1}$ remain strongly connected and C_j has exactly one explored out-edge, with endpoint in C_{j+1}, if $j < i-1$, or in C_i', if $j = i-1$. Thus, $C_1, C_2, \dots, C_{i-1}, C_i'$ form a path of strongly connected components. By the same argument as in the previous paragraph, finished vertices continue to have only explored out-edges and finished out-neighbours.

Finally, if w is finished, it is easy to verify that doing nothing maintains the invariant. □

Lemma: The algorithm correctly labels the strongly connected components of G.

Proof: We mark an active component C_k as a strongly connected component when all its out-edges are explored and, hence, all its out-neighbours are finished. C_k is a strongly connected subgraph of G, by the algorithm's invariant. If it's not a strongly connected component, then one of its out-neighbours must belong to the same component as the vertices in C_k. Since each such out-neighbour is already finished and finished vertices have only finished out-neighbours, no out-neighbours of C_k can reach any vertex in C_k. Thus, C_k is a strongly connected component. □

This algorithm can be implemented using DFS. We maintain a global stack of active vertices, sorted in the order we discovered them. We also maintain a stack of active components, each represented as the number of the first vertex in the component.



```
SCC(G)
Mark every vertex as unexplored
c = 1
AV = empty stack
AC = empty stack
for every vertex v ∈ G do
    if v is unexplored then SCCFromVertex(G, v)

SCCFromVertex(G, v)
Mark v as active
label(v) = c
c = c+1
Push(AV, v)
Push(AC, label(v))
for every out-edge (v,w) of v do
    if w is unexplored then SCCFromVertex(G, w)
    else if w is active then
        while label(w) < Top(AC) do Pop(AC)
if label(v) = Top(AC) then
    Pop(AC)
    do  w = Pop(AV)
        label(w) = label(v)
        Mark w as finished
    while w ≠ v
```

Lemma: SCC(G) takes $O(n+m)$ time.

Proof: Apart from the two while-loops, SCC(G) is standard DFS, which takes $O(n+m)$ time. The cost of the while-loops is proportional to the number of elements pushed onto AV and AC. However, we perform one Push operation on each of these two stacks per invocation SCCFromVertex(G, v) and, since, SCC(G) implements DFS, there is exactly one such invocation per vertex v. Thus, the cost of the while-loops is $O(n)$, and the total cost of the algorithm is $O(n+m)$. ☐

Lemma: When SCC(G) terminates, we have label(v) = label(w) iff v and w belong to the same strongly connected component of G, for any two vertices $v, w \in G$.

Proof: Let $v_1, v_2, \dots, v_h$ be the vertices on AV and let $\ell_1, \ell_2, \dots, \ell_k$ be the entries on AC. We prove a number of invariants that prove that SCC(G) implements the high-level algorithm described earlier and that two vertices receive the same label iff they belong to the same strongly connected components.

(i) label$(v_1) <$ label$(v_2) < \dots <$ label(v_h) and $\ell_1 < \ell_2 < \dots < \ell_k$:
We push a vertex v onto AV, and its label onto AC immediately after giving v a label greater than all previously assigned labels.

(ii) A vertex is active iff it is in AV:
We mark a vertex as active immediately before pushing it onto AV and as finished immediately after popping it from AV.

(iii) $\forall 1 \le i \le k$, some vertex in AV has label ℓ_i:
This is true immediately after pushing ℓ_i onto AC because we push the corresponding vertex onto AV.

An invocation SCCFromVertex (G,v) that pops vertices from AV satisfies $label(v) = \ell_k$. Thus, v is currently on AV. We pop v and all its successors from AV. By (i), these successors have label greater than $\ell_k > \ell_i \; \forall i<k$. Thus, after popping ℓ_k, the invariant continues to hold.

We prove the remaining invariants and the correctness of the algorithm together.

(iv) For every $\ell_i \in AC$, there exists an invocation TraverseFromVertex (G,v) on the call stack such that $label(v) = \ell_i$. AC and AV are empty iff the call stack is empty.

(v) Every vertex $v \in AV$ satisfies $label(v) \geq \ell_1$.

(vi) Define subgraphs $C_1, C_2, \dots, C_k$ such that a vertex v belongs to C_i iff $v \in AV$ and $\ell_i \leq v < \ell_{i+1}$. Then every invocation SCCFromVertex (G,v) satisfies $v \in C_k$.

These invariants hold at the beginning of the algorithm because AV and AC are empty and no invocation SCCFromVertex (G,v) has been made yet. Now assume the invariants hold before an invocation SCCFromVertex (G,v) is made by SCC(G). Then SCCFromVertex (G,v) pushes v onto AV and $label(v)$ onto AC. This is equivalent to picking an arbitrary vertex v and creating a new active component containing it. It also maintains invariants (iv)-(vi) above.

Similarly, immediately before this invocation returns, it is the only invocation on the call stack. Thus, by (iv), AC has a single entry $\ell_1 = label(v)$ and, by (i), (iii) and (v), v is the bottom-most vertex on AV. Thus, before SCCFromVertex(G, v) returns, it removes ℓ_1 from AC and all vertices from AV, so AC and AV are empty again when SCCFromVertex(G, v) returns and invariants (iv)-(vi) hold.

Now consider an invocation SCCFromVertex(G, v) and assume the invariants hold after its first 5 lines. By (vi), $v \in C_k$, so every out-edge (v, w) we explore is an out-edge of C_k. If w is finished, we do nothing, just as the high-level algorithm. This clearly maintains the invariants. If w is unexplored, we invoke SCCFromVertex(G, w), which marks w as active, gives it a new label, and pushes w and $label(w)$ onto AV and AC, respectively. This maintains (iv) and, since $label(w) > label(v) \geq \ell_1$, (v). It also creates a new subgraph C_{k+1} containing only w. Since SCCFromVertex(G, w) is the new active invocation, this satisfies (vi). The high-level algorithm also creates a new component C_{k+1} containing only w when w is unexplored, so once again, the two algorithms behave the same. Finally, if w is active and $w \in C_i$, then the high-level algorithm merges components $C_{i+1}, \dots, C_k$ into C_i. Here, we pop all entries $\ell_j > label(w) \geq \ell_i$ from AC, so every vertex $x \in C_j$ now becomes a member of C_i. As for the invariants, removing entries from AC cannot violate (iv) except that AC may become empty. Since, by (ii), $w \in AV$, and thus, by (v), $label(w) \geq \ell_1$, we do not remove ℓ_i, so AC does not become empty and (iv) is maintained. (v) is maintained since ℓ_i and the contents of AV do not change. (vi) is maintained because $v \in C_k$ before merging $C_{i+1}, \dots, C_k$ into C_i, so $v \in C_i$ after and C_i is the new topmost component on AC.

It remains to argue that the invariants are maintained after SCCFromVertex(G, v) returns. We already did so for the case when SCCFromVertex(G, v) is called by SCC(G). So consider the case when SCCFromVertex(G, v) was called by another invocation SCCFromVertex(G, u), which becomes active again once SCCFromVertex(G, v) returns.

If label$(v) > \ell_k$, we do not change AC or AV before SCCFromVertex(G, v) returns. Thus, (iv) and (v) remain true. As for (vi), observe that there exists a vertex $x \in AV$ such that label$(x) = \ell_k$, by (iii). If label$(u) \geq \ell_k$, (vi) is maintained. Otherwise, label$(u) <$ label$(x) <$ label(v), which implies that SCCFromVertex(G, x) is called after SCCFromVertex(G, u) and before SCCFromVertex(G, v), which implies that it must have returned before SCCFromVertex(G, u) returns. This, however, violates invariant (iv), a contradiction. Thus, label$(u) \geq \ell_k$, and (vi) is maintained.

If label$(v) = \ell_k$, we remove ℓ_k from AC and we remove all vertices succeeding v from AV and label them with label(v) as their component label. By (ii), $v \in AV$ at this time. By (i), the vertices succeeding v in AV have labels no less than $\text{label}(v) = \ell_k$, that is, they belong to C_k. Any other vertex in AV has label less than ℓ_k (by (i) again) and thus does not belong to C_k. Thus, what we are doing is labelling all vertices in C_k as finished, and we assign them the same component label. By the correctness of the high-level algorithm, this correctly labels C_k as a strongly connected component of G, provided C_k has no unexplored out-edges at this time. To see that this is the case, observe that v has no unexplored out-edges left (because we backtrack from v). Every vertex $w \neq v$ in C_k is active and, by the definition of C_k, has a label greater than $\text{label}(v)$. Thus, the invocation SCCFromVertex(G, w) must have been made after SCCFromVertex(G, v). Since SCCFromVertex(G, v) is about to return, SCCFromVertex(G, w) must have returned first. At this point, all out-edges of w are explored. Thus, all out-edges of C_k are explored when we mark its vertices as finished. $\square$