COMP36111: Advanced Algorithms I

Lecture 1a:

Some Basic Graph Algorithms

Ian Pratt-Hartmann

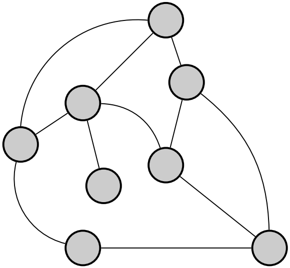
Room KB2.38: email: ipratt@cs.man.ac.uk

2017–18

- In this lecture, we consider algorithms for determining very simple properties of (directed and undirected) graphs.
- The lecture is divided into three parts. The first establishes notation and terminology; the second introduces some very basic algorithms based on depth-first search; the third presents a generalization—Tarjan's algorithm for strongly connected components.

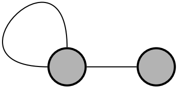
Outline

- A graph is a pair $G = (V, E)$, where V is a finite set and E a set of subsets of V of cardinality 2.
- We call the elements of V vertices, and the elements of E edges.
- If $\{u, v\} \in E$, we say that u and v are neighbours.
- If $v \in V$, $e \in E$ and $v \in e$, we say v and e are adjacent.
- Graphs are typically displayed pictorially:

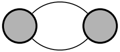


- The following are not pictures of graphs:

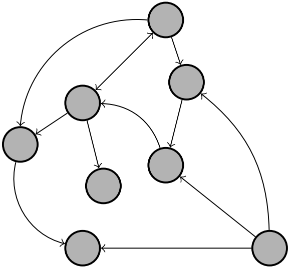
  - Self-loops:



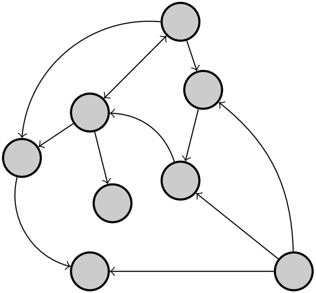
  - Multiple edges



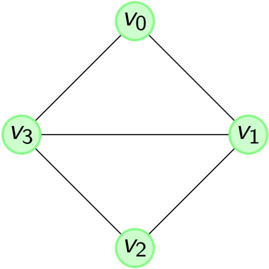
  - Directions on edges



- A directed graph is a pair $G = (V, E)$, where V is a set and E a set of ordered pairs of distinct elements of V.
- Vertices, edges neighbours and adjacency are defined as with graphs.
- Directed graphs are again often depicted pictorially (notice the arrows on the edges):



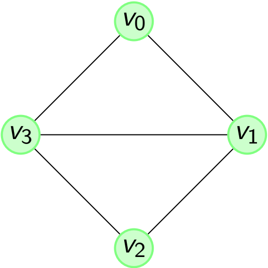
- (Directed) graphs may be stored using adjacency lists, interpreted in the obvious way. Here is an example of an undirected graph:




```
0: 1, 3
1: 0, 2, 3
2: 1, 3
3: 0, 1, 2
```

- From any vertex, the adjacent edges can be accessed efficiently.
- From any edge, the adjacent vertices can be accessed efficiently.

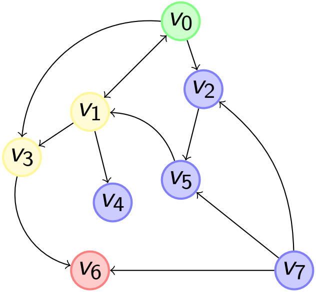
- Alternatively, graphs can be stored using using (symmetric) matrices.



$$\begin{pmatrix} * & 1 & 0 & 1 \\ 1 & * & 1 & 1 \\ 0 & 1 & * & 1 \\ 1 & 1 & 1 & * \end{pmatrix}$$

- Note that we do not care about the diagonal elements.
- This method is wasteful in terms of memory, but often more convenient than adjacency lists.
- In these lectures, we will employ adjacency lists by default.

- If $G = (V, E)$ is a (directed) graph, and $u, v \in V$, we say that v is reachable from u if there exists a sequence $u = u_0, \ldots, u_m = v$ from V with $m \geq 0$ such that, for each i $(0 \leq i < m)$ $(u_i, u_{i+1}) \in E$.
- In the following directed graph, v_6 is reachable from v_0



since we have the sequence $v_0 \to v_1 \to v_3 \to v_6$.

- However, v_7 is not reachable from v_0.

- A graph is connected if every node is reachable from every other.
- A directed graph is strongly connected if every vertex is reachable from every other.
- These notions give rise to the following two problems:

CONNECTIVITY
Given: A graph $G = (V, E)$.
Return: Yes if G is connected, No otherwise.

STRONG CONNECTIVITY
Given: A directed graph $G = (V, E)$.
Return: Yes if G is strongly connected, No otherwise.

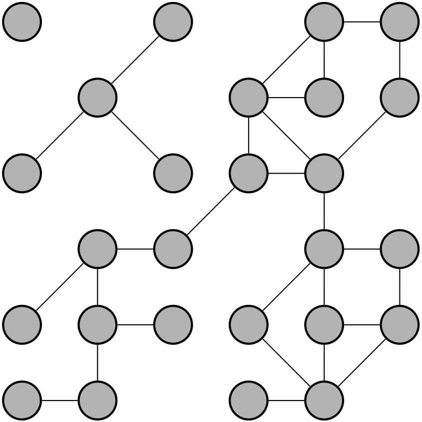
- The following are natural generalizations of the notions of connectedness and strong connectedness.
- A connected component of a graph is a maximal set of vertices each of which is reachable from any other.
- A strongly connected component of a directed graph is a maximal set of vertices each of which is reachable (in the directed graph sense) from any other.
- It is easy to see that the connected components of a graph $G = (V, E)$ form a partition of V. Similarly for the strongly connected components of a directed graph.

- A graph is connected just in case it has exactly one connected component.
- A directed graph is strongly connected just in case it has exactly one strongly connected component.
- These notions give rise to the following two computational tasks:

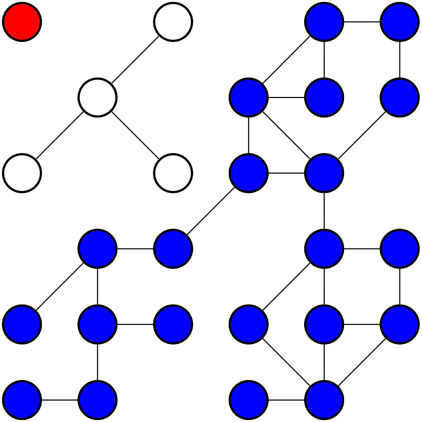
> CONNECTED COMPONENTS
> Given: A graph $G = (V, E)$.
> Return: The connected components of G.

> STRONGLY CONNECTED COMPONENTS
> Given: A directed graph $G = (V, E)$.
> Return: The strongly connected components of G.

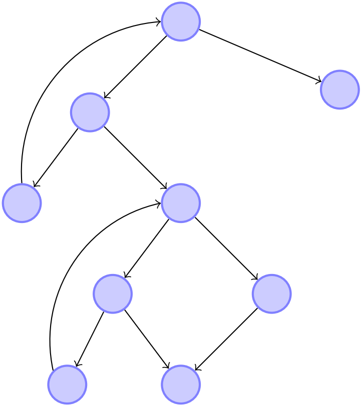
- The following example illustrates the problem of finding the connected components of a graph.



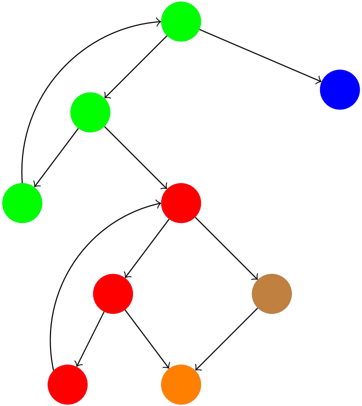
- The following example illustrates the problem of finding the connected components of a graph.



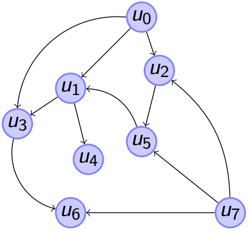
- The following example illustrates the problem of finding the strongly connected components of a directed graph.



- The following example illustrates the problem of finding the strongly connected components of a directed graph.



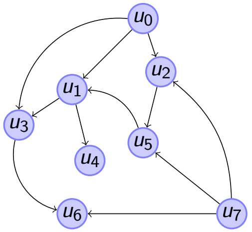
- A cycle in a directed graph G is a sequence of vertices $v_0, \ldots, v_k = v_0$ $(k \geq 2)$ such that, for all i $(0 \leq i < k)$, (v_i, v_{i+1}) is an edge. We call G cyclic if it has a cycle, otherwise acyclic.
- The following directed graph is ...



- This notion gives rise to the following problem:

CYCLICITY
Given: A directed graph $G = (V, E)$.
Return: Yes if G is cyclic, No otherwise.

- A cycle in a directed graph G is a sequence of vertices $v_0, \ldots, v_k = v_0$ $(k \geq 2)$ such that, for all i $(0 \leq i < k)$, (v_i, v_{i+1}) is an edge. We call G cyclic if it has a cycle, otherwise acyclic.
- The following directed graph is acyclic.



- This notion gives rise to the following problem:

CYCLICITY
Given: A directed graph $G = (V, E)$.
Return: Yes if G is cyclic, No otherwise.

Outline

Graphs and directed graphs

Depth-first search and other simple algorithms

Tarjan's algorithm for strongly connected components

- Here is a simple algorithm to reverse all the links in a directed graph, G.

```
begin reverse(G)
    G'.vertices = G.vertices
    for each u ∈ G'.vertices do
        G'.edges(u) = ∅
    for each u ∈ G.vertices do
        for each v ∈ G.edges(u) do
            add u to G'.edges(v)
    return G'
end reverse
```

- If G has n vertices and m edges, running time is:

- Here is a simple algorithm to reverse all the links in a directed graph, G.

begin $\texttt{reverse}(G)$
    G'.vertices $= G$.vertices
    for each $u \in G'$.vertices do
        G'.edges$(u) = \emptyset$
    for each $u \in G$.vertices do
        for each $v \in G$.edges(u) do
            add u to G'.edges(v)
    return G'
end $\texttt{reverse}$

- If G has n vertices and m edges, running time is: $O(m + n)$.

- Here is a simple algorithm to compute the in-degree of all vertices in a directed graph

```
begin inDegCompute(G)
    for each u ∈ G.vertices do
        G.inDeg(u) = 0
    for each u ∈ G.vertices do
        for each v ∈ G.edges(u) do
            increment G.inDeg(v)
end inDegCompute
```

- If G has n vertices and m edges, running time is: .

- Here is a simple algorithm to compute the in-degree of all vertices in a directed graph

```
begin inDegCompute(G)
    for each u ∈ G.vertices do
        G.inDeg(u) = 0
    for each u ∈ G.vertices do
        for each v ∈ G.edges(u) do
            increment G.inDeg(v)
end inDegCompute
```

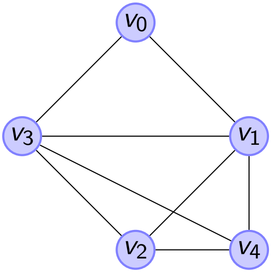
- If G has n vertices and m edges, running time is: $O(m + n)$.

- Here is a simple algorithm, depth-first search, that computes the vertices of a (directed or undirected) graph G reachable from a given vertex v.

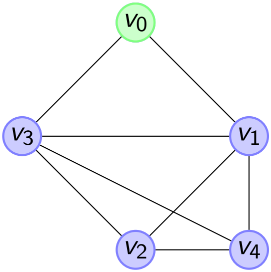
```
begin DFS(G, v)
      mark v
for each w ∈ G.edges(v) do
    if w unmarked do
      DFS(G, w)
end DFS
```

- This algorithm marks all vertices reachable from v.
- It works for with directed and undirected graphs.
- $\texttt{DFS}((V, E), v)$ runs in time $O(m + n)$ where $n = |V|$ and $m = |E|$.

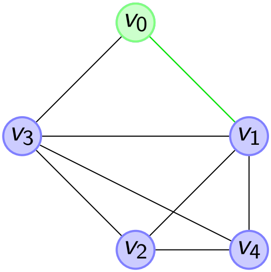
- Here is an animation:



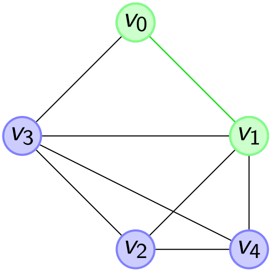
- Here is an animation:



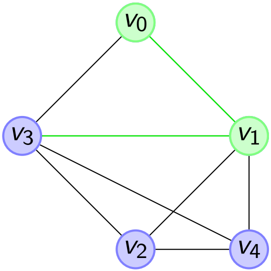
- Here is an animation:



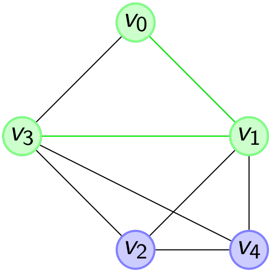
- Here is an animation:



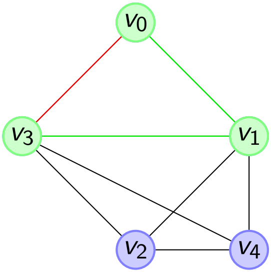
- Here is an animation:



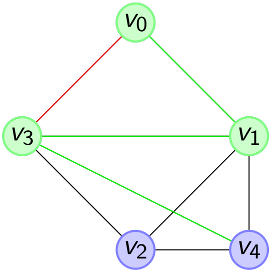
- Here is an animation:



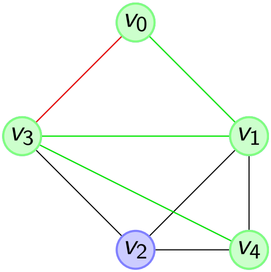
- Here is an animation:



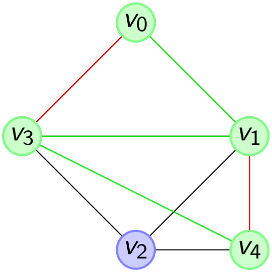
- Here is an animation:



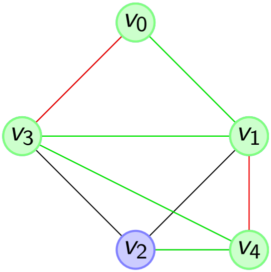
- Here is an animation:



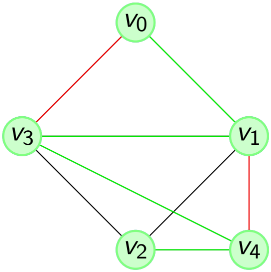
- Here is an animation:



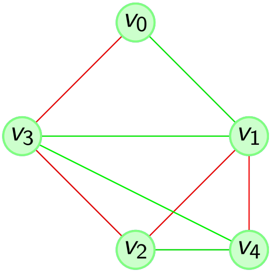
- Here is an animation:



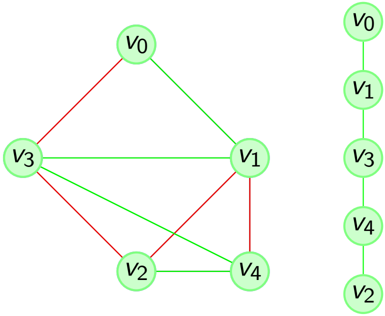
- Here is an animation:



- Here is an animation:



- Here is an animation:



Theorem

CONNECTIVITY of a graph (V, E) can be determined in time $O(|V| + |E|)$.

Proof.

Pick any vertex v, run DFS on v, and check that all vertices have been marked. □

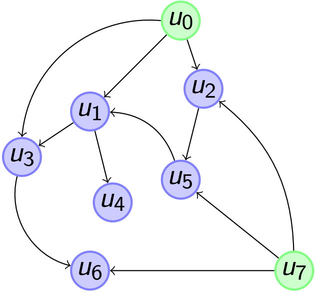
Theorem

STRONG CONNECTIVITY of a directed graph $G = (V, E)$ can be determined in time $O(|V| + |E|)$.

Proof.

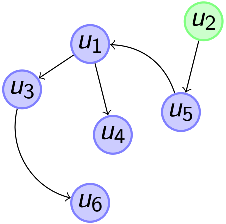
If V is empty, G is strongly connected. Otherwise, pick any $v_0 \in V$. Let $G^{\leftarrow}$ be the reversal of G. Then G is strongly connected if and only if every vertex $v \in V$ is reachable from v_0 in both G and $G^{\leftarrow}$. □

- Recall the definition of cycle and cyclicity for directed graphs, given above.
- A topological sort(ing) of a directed graph G is an ordering of its vertices as $v_0, \ldots, v_{n-1}$ such that, for all edges (v_i, v_j) we have $i < j$.



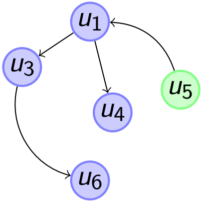
- It is simple to show that a graph is acyclic if and only if it admits a topological sorting.
- The following algorithm takes a directed graph and finds a topological sorting, or outputs "cyclic".

- Recall the definition of cycle and cyclicity for directed graphs, given above.
- A topological sort(ing) of a directed graph G is an ordering of its vertices as $v_0, \ldots, v_{n-1}$ such that, for all edges (v_i, v_j) we have $i < j$.



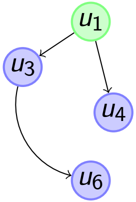
- It is simple to show that a graph is acyclic if and only if it admits a topological sorting.
- The following algorithm takes a directed graph and finds a topological sorting, or outputs "cyclic".

- Recall the definition of cycle and cyclicity for directed graphs, given above.
- A topological sort(ing) of a directed graph G is an ordering of its vertices as $v_0, \ldots, v_{n-1}$ such that, for all edges (v_i, v_j) we have $i < j$.



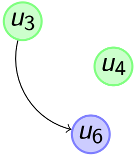
- It is simple to show that a graph is acyclic if and only if it admits a topological sorting.
- The following algorithm takes a directed graph and finds a topological sorting, or outputs "cyclic".

- Recall the definition of cycle and cyclicity for directed graphs, given above.
- A topological sort(ing) of a directed graph G is an ordering of its vertices as $v_0, \ldots, v_{n-1}$ such that, for all edges (v_i, v_j) we have $i < j$.



- It is simple to show that a graph is acyclic if and only if it admits a topological sorting.
- The following algorithm takes a directed graph and finds a topological sorting, or outputs “cyclic”.

- Recall the definition of cycle and cyclicity for directed graphs, given above.
- A topological sort(ing) of a directed graph G is an ordering of its vertices as $v_0, \ldots, v_{n-1}$ such that, for all edges (v_i, v_j) we have $i < j$.



- It is simple to show that a graph is acyclic if and only if it admits a topological sorting.
- The following algorithm takes a directed graph and finds a topological sorting, or outputs "cyclic".

- Recall the definition of cycle and cyclicity for directed graphs, given above.
- A topological sort(ing) of a directed graph G is an ordering of its vertices as $v_0, \dots, v_{n-1}$ such that, for all edges (v_i, v_j) we have $i < j$.

- It is simple to show that a graph is acyclic if and only if it admits a topological sorting.
- The following algorithm takes a directed graph and finds a topological sorting, or outputs "cyclic".

- Here is the pseudocode for topological sorting $G = (V, E)$

```
begin topSort(G)
      compute all in-degrees and store in G.inDeg
      let S = ∅ be a stack and let i = 0
      for each v ∈ G.vertices
          if G.inDeg(v) = 0 then push v on S
      while S is non-empty
          u = pop(S)
          let sort(i) = u
          increment i
          for each v ∈ G.edges(u) do
            decrement G.inDeg
            if G.inDeg(v) = 0
              push v on S
      if i = n then output sort(0), . . . , sort(n − 1)
      output "cyclic"
end DFS
```

- Running time is $O(m + n)$ where $n = |V|$ and $m = |E|$.

Outline

Graphs and directed graphs

Depth-first search and other simple algorithms

Tarjan's algorithm for strongly connected components

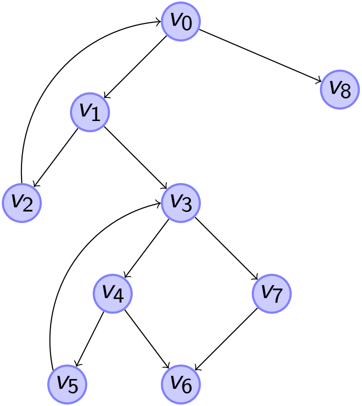
- Recall the definition of strongly connected component (SCC) for a directed graph, given above.
- The following algorithm, known as Tarjan's algorithm, allows us to determine the strongly connected components of a directed graph.
- There is a very good presentation on

  https://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm
- We reproduce the core of this algorithm (more or less verbatim from Wikipedia), and illustrate with an example.

- The algorithm has the following features:
  - It can be seen as a version of depth-first search.
  - It maintains a stack of vertices in contention to be in an SCC.
  - Each vertex is given an index and a lowlink value, which is the earliest node encountered so far and known to be in the same SCC as that vertex.
- The core of Tarjan's algorithm is the function `strongConnect`(v), which we call repeatedly on some vertex v until all vertices have been assigned to an SCC.
- This function uses a global variable `index`, initially set to zero, and a global stack of vertices, initially set to empty.

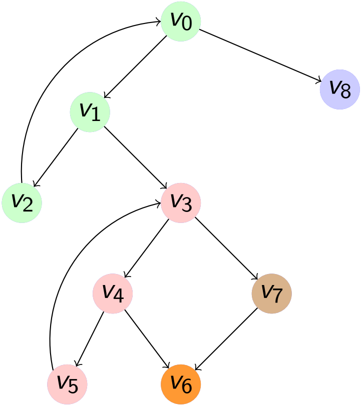
```
strongConnect(v)
    v.index := index
    v.lowlink := index
    increment index
    push v on stack
    for each w in G.successors(v)
        if w.index undefined
            strongConnect(w)
            v.lowlink := min(v.lowlink, w.lowlink)
        if w is on stack
            v.lowlink := min(v.lowlink, w.index)
    if v.lowlink = v.index
        repeat
            pop w off stack
            add w to current strongly connected component
        while w! = v
        output the current strongly connected component
end strongConnect
```

- The graph



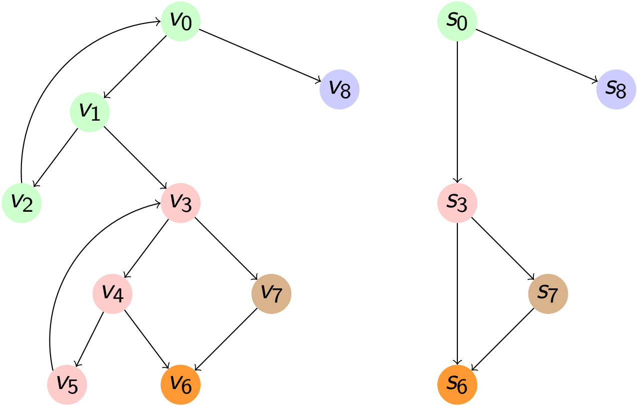
has strongly connected components:

- The graph



has strongly connected components:
$\{v_0, v_1, v_2\}, \ \{v_3, v_4, v_5\}, \ \{v6\}, \ \{v_7\}, \ \{v_8\}$.

- Notice that the strongly connected components naturally form an acyclic directed graph. Indeed, Tarjan's algorithm computes a topological ordering for this graph.



- In particular, if given an acyclic graph as input, this algorithm will compute a topological ordering—in fact, it is just the algorithm we encountered above for topological sorting.