COMP36111: Advanced Algorithms

Lecture 1:

Getting Started

Ian Pratt-Hartmann

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

2019–20
• In this lecture, we consider some basic algorithms operating on directed graphs.

• The lecture is divided into four parts:
  • review of basic concepts and notation;
  • application of the familiar depth-first-search algorithm;
  • detecting cycles and ‘topological sorting’;
  • a classic algorithm that every Computer Scientist should know.
Outline

Directed graphs (revision)

Testing strong connectivity (revision)

Topological sort

Tarjan’s algorithm for strongly connected components

Summary
• 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 $V$.

• The elements of $V$ are vertices, and the elements of $E$ are edges. If $e = (u, v) \in E$ is an edge, then $u$ and $v$ are neighbours (of each other), and are incident on $e$.

• Directed graphs are often depicted pictorially (notice the arrows on the edges):
• The following are not pictures of directed graphs:

  • Self-loops:

  • Multiple edges

  • Directionless edges
Directed graphs may be stored using **adjacency lists**, interpreted in the obvious way. Here is an example of an undirected graph:

![Graph Diagram]

- From any vertex, the adjacent edges can be accessed in a single operation.
- From any edge, the adjacent vertices can be accessed in a single operation.
• Alternatively, directed graphs can be stored using (symmetric) matrices.

\[
\begin{pmatrix}
\ast & 1 & 0 & 1 \\
0 & \ast & 1 & 1 \\
0 & 0 & \ast & 0 \\
0 & 1 & 1 & \ast
\end{pmatrix}
\]

• 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 \textit{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 \( v_0 \rightarrow v_1 \rightarrow v_3 \rightarrow v_6 \).

• A directed graph is \textit{strongly connected} if every vertex is reachable from every other.
• 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 . . .

![Diagram](image)

• Note that we do not insist that the vertices in a cycle are all distinct. Warning: texts vary as to exact definitions.
- 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.

- Note that we do not insist that the vertices in a cycle are all distinct. Warning: texts vary as to exact definitions.
These notions give rise to the following problems:

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

**STRONG CONNECTIVITY**
Given: A directed graph \( G = (V, E) \).
Return: Yes if \( G \) is strongly connected, No otherwise.
• 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 strongly connected components of a graph \( G = (V, E) \) form a partition of \( V \).

• Evidently, a directed graph is strongly connected just in case it has exactly one strongly connected component.

• This notion gives rise to the following task:

**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 strongly connected components of a directed graph.
The following example illustrates the problem of finding the strongly connected components of a directed graph.
• Note the difference between
   (i) the problems CYCLICITY and STRONG CONNECTIVITY
   (ii) the task STRONGLY CONNECTED COMPONENTS.
• In this course (and in Complexity Theory generally),
  ‘problems’ have YES/NO answers.
• We will always put problems in \textcolor{blue}{blue} boxes and other tasks in
  \textcolor{green}{green} boxes.
• Often, the difference is less dramatic than might be expected.
Outline

Directed graphs (revision)

Testing strong connectivity (revision)

Topological sort

Tarjan’s algorithm for strongly connected components

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

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

• 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 reverse($G$)
    $G'$.vertices = $G$.vertices
    for each $u \in G'$.vertices do
        $G'$.edges($u$) = ∅
    for each $u \in G$.vertices do
        for each $v \in G$.edges($u$) do
            add $u$ to $G'$.edges($v$)
    return $G'$
end reverse

• If $G$ has $n$ vertices and $m$ edges, running time is: $O(m + n)$.
Here is a simple algorithm to compute the in-degree of all vertices in a directed graph:

```plaintext
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(u)
  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(u)
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 graph \( G \) reachable from a given vertex \( v \).

\[
\begin{align*}
\text{begin } & \text{DFS}(G, v) \\
& \text{mark } v \\
& \text{for each } w \in G.\text{edges}(v) \text{ do} \\
& \quad \text{if } w \text{ unmarked do} \\
& \quad \quad \text{DFS}(G, w) \\
\text{end DFS}
\end{align*}
\]

This algorithm marks all vertices reachable from \( v \).

\( \text{DFS}((V, E), v) \) 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

*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 $G^\leftarrow$. Now use dfs (twice) to check this.

□
Outline

Directed graphs (revision)

Testing strong connectivity (revision)

Topological sort

Tarjan’s algorithm for strongly connected components

Summary
• 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 $u_0, \ldots, u_{n-1}$ such that, for all edges $(u_i, u_j)$ 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”.

\[
\begin{array}{c}
\text{Input: a directed graph } G \\
\text{Output: a topological sorting } u_0, u_1, \ldots, u_{n-1} \\
\text{or outputs } \text{"cyclic" if G is not acyclic.}
\end{array}
\]
• 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 $u_0, \ldots, u_{n-1}$ such that, for all edges $(u_i, u_j)$ 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”.

[Diagram of a directed graph with vertices $u_0$, $u_1$, $u_2$, $u_3$, $u_4$, $u_5$, $u_6$, with edges directed as shown.]
• 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 $u_0, \ldots, u_{n-1}$ such that, for all edges $(u_i, u_j)$ 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”.

![Diagram showing a directed cycle](image)
• 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 $u_0, \ldots, u_{n-1}$ such that, for all edges $(u_i, u_j)$ 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 $u_0, \ldots, u_{n-1}$ such that, for all edges $(u_i, u_j)$ 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”.

\begin{center}
\begin{tikzpicture}[->, scale=0.8, every node/.style={scale=0.8}]
  \node (u3) at (0,0) {$u_3$};
  \node (u4) at (0.5,0.5) {$u_4$};
  \node (u6) at (-0.5,0.5) {$u_6$};
  \path (u3) edge (u6);
\end{tikzpicture}
\end{center}
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 $u_0, \ldots, u_{n-1}$ such that, for all edges $(u_i, u_j)$ 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 $S = \emptyset$ be a stack and let $i = 0$
    for each $v \in G$.vertices
        if $G$.inDeg($v$) = 0 then push $v$ on $S$
    while $S$ is non-empty
        $u = \text{pop}(S)$
        let sort($i$) = $u$
        increment $i$
        for each $v \in G$.edges($u$) do
            decrement $G$.inDeg($v$)
            if $G$.inDeg($v$) = 0 then push $v$ on $S$
        if $i = n$ then return sort(0), \ldots, sort($n - 1$)
    return "cyclic"
end DFS
• Running time is $O(m + n)$ where $n = |V|$ and $m = |E|$.

• Why is the algorithm correct? Critical observation:
  • When a vertex is popped off the stack, all of its predecessors (transitively) must already have been popped off the stack. So these vertices are never on cycles.
  • If any vertices remain after the stack is empty, there must be a (directed) subgraph in which no vertices have in-degree 0.

• Hence, every non-cyclic graph has a topological ordering!
Outline

Directed graphs (revision)

Testing strong connectivity (revision)

Topological sort

Tarjan’s algorithm for strongly connected components

Summary
• 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 at 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.edges(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\}, \{v_6\}, \{v_7\}, \{v_8\}. 
• Notice that the strongly connected components naturally form an acyclic directed graph. If called on a node \( v \), strongConnect computes a topological ordering for the sub-graph reachable from \( v \).

• If that subgraph is acyclic, strongConnect is equivalent to the above algorithm for topological sorting.
Outline

Directed graphs (revision)

Testing strong connectivity (revision)

Topological sort

Tarjan’s algorithm for strongly connected components

Summary
Summary

• The concepts strongly connected and strongly connected component for directed graphs
• Determining strong connectedness with depth-first search
• The concepts of cyclicity and topological sorting
• The topological sorting algorithm
• Tarjan’s algorithm for strongly connected components