Advanced Algorithmic Techniques
(COMP523)
Testing for bipartiteness
Bipartite graphs

• A graph \( G=(V,E) \) is bipartite if any only if it can be partitioned into sets \( A \) and \( B \) such that each edge has one endpoint in \( A \) and one endpoint in \( B \).

• Often, we write \( G=(A \cup B,E) \).
Alternative definitions

• A graph $G=(V,E)$ is bipartite if any only if its nodes can be coloured with 2 colours (say red and green), such that every vertex has one red endpoint and one green endpoint.

• A graph $G=(V,E)$ is bipartite if any only if it does not contain any cycles of odd length.
No odd cycles

• A graph $G=(V,E)$ is bipartite if any only if it does not contain any cycles of odd length.

• $\Rightarrow$ Assume that $G$ is bipartite

• Suppose that $G$ does contain an odd cycle (proof by contradiction), $C = u_1 u_2 u_3 \ldots u_n u$ for some $u$ in $A$ (wlog), or alternatively, for some $u$ that is red.

• Because $G$ is bipartite, $u_2$ must be green, and then $u_3$ must be red, and so on.

• Generally, we observe that for all $k$ in $\{1,2, \ldots ,n\}$, $u_k$ is red if $k$ is odd and green if $k$ is even.

• By assumption, $n$ is odd, so it must be red. But then $u$ cannot be red, because $G$ is bipartite.
Alternative definitions

• A graph $G=(V,E)$ is bipartite if any only if its nodes can be coloured with 2 colours (say red and green), such that every vertex has one red endpoint and one green endpoint.

• A graph $G=(V,E)$ is bipartite if any only if it does not contain any cycles of odd length.

• Sometimes, these alternatives definitions are also called “characterisations”.

Testing bipartiteness

- Given a graph $G=(V,E)$, decide if it is bipartite or not.
- Given a graph $G=(V,E)$ decide if it is 2-colourable or not.
- Given a graph $G=(V,E)$ decide if it is contains cycles of odd length or not.
Colouring the nodes
Colouring the nodes

• Does this remind you of something?

• It is essentially BFS!

• We label the nodes of layer 1 red, the nodes of layer 2 green, and so on.

• Implementation:

• Add a check for odd/even and assign a colour accordingly.

• In the end, check all edges to see if they have endpoints of the same colour.
Algorithm BFS(G, s)

Initialise empty list \( L_0 \)
Initialise colour list \( C \)
Insert \( s \) into \( L_0 \)
Set \( C[s] = \text{red} \)

Set \( i=0 \)
While \( L_i \) is not empty
  Initialise empty list \( L_{i+1} \)
  for each node \( v \) in \( L_i \)
    for all edges \( e \) incident to \( v \)
      if edge \( e \) is unexplored
        let \( w \) be the other endpoint of \( e \)
        if node \( w \) is unexplored
          label \( e \) as discovery edge
          insert \( w \) into \( L_{i+1} \)
          \( \text{If } i+1 \text{ is odd, set } C[w] = \text{red}, \text{ else set } C[w] = \text{green} \)
        else
          label \( e \) as cross edge
  \( i = i+1 \)

For all edges \( e=(u,v) \) in \( G \)
  if \( C[u] = C[v] \) return “not bipartite”
Return “bipartite”
Running time

• What did we add?

  • A colour assignment for the starting node.
  
  • An odd/even check and a colour assignment for each node in the loop.
  
  • An extra loop for checking the edges of their graph for the colours of their endpoints.

• How much more do we “pay” (asymptotically)?

  • Nothing!

• Running time $O(m+n)$. 
Correctness

• We started at an arbitrary node $s$.

• Maybe we were lucky / unlucky?
Properties of BFS

• For simplicity, assume that the graph is connected.

• The traversal visits all vertices of the graph.

• The discovery edges form a spanning tree.

• The path of the spanning tree from s to a node v at level i has i edges, and this is the shortest path.

• If e=(u,v) is a cross edge, then the u and v differ by at most one level.
Properties of BFS

• If e=(u,v) is a cross edge, then the u and v differ by at most one level.

• If e=(u,v) is a discovery edge, then the u and v differ by at most one level.
Correctness

• Suppose that G is bipartite. Then, all cycles must be of even length.

• Suppose to the contrary that the algorithm returns “not bipartite”.

  • This means that it has found an edge $e=(x,y)$ with endpoints of the same colour.

  • Since the endpoints of any edge can not differ by more than one layer and layers have alternating colours, $x$ and $y$ must be in the same layer.
Correctness

• Consider the lowest common ancestor $z$ of $x$ and $y$ in the BFS tree.

• Let $L_i$ be the layer of $z$ and let $L_j$ be the layer of $x$ and $y$.

• Consider the cycle $(z \ldots x), (x,y), (y \ldots z)$.

• Length: $(j-i) + 1 + (j-i)$ (odd)

• Contradiction!
Correctness

• Suppose that G is not bipartite. Then, it must contain a cycle of odd length.

• Suppose to the contrary that the algorithm returns “bipartite”.

  • This means that it has not found any edge $e=(x,y)$ with endpoints of the same colour.

  • This also obviously means that there is no edge with endpoints in the same layer.

  • By the earlier discussion, all edges must have endpoints that lie in consecutive layers.

  • Take any cycle $(z, \ldots, z)$. Since for every edge in this cycle there is a change of layer (from $j$ to $j+1$ or from $j+1$ to $j$), the cycle must have even length.

  • Contradiction!
Directed graphs

- Nodes are arranged as a list, each node points to the neighbours.

- For directed graphs, the node points in two directions, for in-degree and for out-degree.
DFS and BFS on directed graphs

• Very similar to their version on undirected graphs.

• When we are at a node and we examine its neighbours, a neighbour is now only a node that we can reach with a directed edge.

• The running time is still $O(n+m)$. 
Breadth-First Search
Directed Acyclic Graphs

• A directed acyclic graph (DAG) $G$ is a graph that does not have any cycles.

![Not a DAG](image1)

![A DAG](image2)
Properties of DAGs

• They appear quite often in many applications.

• Example - prerequisite modules: To take module A you need to have taken module B and module C.

• If the module prerequisite relation has a cycle, then it is impossible to get a degree!
Topological Ordering

• Given a directed graph $G$, a topological ordering of $G$ is an ordering of the nodes $u_1, u_2, \ldots, u_n$, such that for every edge $e=(u_i, u_j)$, it holds that $i < j$.

• Intuitively, a topological ordering orders the nodes in a way such that all edges point "forward".
Topological Ordering implies DAG

• If graph $G$ has a topological ordering, then $G$ is a DAG.

• Suppose by contradiction that $G$ has a topological ordering $(u_1, u_2, \ldots, u_n)$ but it also has a cycle $C$.

• Let $u_j$ be the smallest element of $C$ according to the topological ordering.

• Let $u_i$ be its predecessor in the cycle (i.e., there is an edge $e=(u_i, u_j)$).

• $u_i$ must appear before $u_j$ in the topological order, by the presence of this edge.

• This contradicts the fact that $u_j$ was the smallest element of $C$ according to the topological ordering.
Does DAG imply topological ordering?

- **TO** => **DAG** was proved via proof-by-contradiction.
- **DAG** => **TO** will be proved via “proof-by-algorithm”.
- We will design an *efficient* algorithm that, given a DAG $G$, finds a topological ordering of $G$. 
How do we start?

- Could we have started with anything other than node 1?
- The starting node must have no incoming edges!
- Can we always find such a node?
Source node

• A source node is a node with no incoming edges.

• **Lemma**: Every DAG has at least one source node.

• Proof by contradiction:

  • Assume that every node has at least one incoming edge.

  • Start from any node \( u \) and follow edges from \( u \) backwards.

    • Equivalently, we move to a neighbour of \( u \) in \( G^{\text{rev}} \).

  • We can do that **for every node**, since by assumption there is no source.

  • After at least \( n+1 \) steps, we will have visited the same node twice.

  • The graph has a cycle, therefore it can’t be a DAG. **Contradiction!**
Pictorially
Another simple fact

• If we remove a node \( u \) and all its incident edges from a DAG \( G \), the resulting graph \( G' \) is still a DAG.

• If \( G' \) had a cycle, the same cycle would be present in \( G \).
DAG implies topological ordering

• Proof-by-induction:

  • Base Case: If the DAG has one or two nodes, it clearly has a topological ordering.

  • Inductive step: Assume that a DAG with up to $k$ nodes has a topological ordering (Inductive Hypothesis). We will prove that a DAG with $k+1$ nodes has a topological ordering.

    • By our lemma, there is at least one source node in $G$, and let $u$ be such a node.

    • Put $u$ first in the topological ordering (safe, since $u$ is a source).

    • Consider the graph $G'$, obtained by $G$ if we remove $u$ and its incident edges.

    • $G'$ is a DAG (by the simple fact) with $k$ nodes.

      • It has a topological ordering by the induction hypothesis.

    • Append this ordering to $u$. 
Where is the “proof-by-algorithm”? 

• We can turn that induction proof into an algorithm.

Algorithm \texttt{TopologicalSort}(G) 
Find a source vertex \( u \) and put it first in the order. 
Let \( G’=G-\{u\} \) 
\texttt{TopologicalSort}(G’) 
Append this order after \( u \)
Example
Running time

• We need to find a source $u$.

• We could check each node of the graph.

• We check $n$ nodes in the first iteration, $n-1$ nodes in the second, and so on...

• What is the running time of this?
  
  • $O(n^2)$

• Can we do better?
A faster algorithm

• We will be more efficient in the choice of sources.

• We will say that a node is active, if it has not been selected (and therefore removed) as a source by the algorithm.

• We maintain two things:
  • (a) For each node $w$, the number of incoming edges from active nodes.
  • (b) The set $S$ of all active nodes that have no incoming edges from other active nodes.
A faster algorithm

• In the beginning, all nodes are active and we can initialise \((a)\) and \((b)\) via a pass through the graph (time \(O(m+n)\))

• In each iteration:
  
  • We select a node \(u\) from the set \(S\).
  
  • We delete \(u\).
  
  • We go through all the neighbours \(w\) of \(u\) and we reduce their value in \((a)\) (i.e., number of incoming edges from active nodes) by 1.
  
  • When the value of \((a)\) for some node \(w\) goes to 0, \(w\) is added to the set \(S\).
Reading

Kleinberg and Tardos 3.4, 3.6 (for bipartiteness and topological sort)

Roughgarden 8.5 (for topological sort)

CLRS 20.4 (for topological sort)

See you next year!