CS 341: ALGORITHMS
Lecture 13: graph algorithms II – finishing BFS, depth first search
Readings: see website
Trevor Brown
https://student.cs.uwaterloo.ca/~cs341
trevor.brown@uwaterloo.ca

BFS APPLICATION:
TESTING WHETHER A GRAPH IS BIPARTITE

(UNDIRECTED) BIPARTITE GRAPHS AND BFS

A graph is bipartite if the nodes can be partitioned into sets $R$ and $B$ such that each edge has one endpoint in $R$ and one endpoint in $B$.

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

PROOF
PART 1: ODD CYCLE $\Rightarrow$ NOT BIPARTITE
- Suppose there is an odd length cycle $v_1, v_2, ..., v_{2k+1}, v_k$.

PROOF
PART 2: ALL CYCLES HAVE EVEN LENGTH $\Rightarrow$ BIPARTITE
- Let $v_i$ be any node, and $d(v)$ be the distance from $v_i$ to $v$.

CRUCIAL PROPERTY:
NO ODD CYCLES

Claim: a graph is bipartite if and only if it does not contain an odd length cycle.

What happens if I create an odd length cycle?

Edge with both endpoints in $B$!

WLOG let $v_i \in R$.

Then we must have $v_2 \in B$.

And finally $v_{2k+1} \in R \iff$ Contradiction!

Both endpoints in $R$!

$R = \text{odd } d(v)$ and $B = \text{even } d(v)$.

WTP: no edge between red nodes, no edge between blue nodes.
**BAD EDGES MEAN ODD CYCLES**

**Claim:** if there were an edge between red nodes, or between blue nodes, there would be an odd length cycle

- WLOG suppose for contradiction \((u, v) \in E\) where \(u, v \in R\)
- Since \(u, v \in R\), distances \(d(u)\) and \(d(v)\) from \(v_i\) are both odd

Recall \(d(u) = \text{length of shortest path } v_i \rightarrow \cdots \rightarrow u\)

...and \(d(v) = \text{odd}\), the shortest path \(v_i \rightarrow \cdots \rightarrow v\)

So there is no edge \((u, v)\) where \(u, v \in R\) [case \(B\) is similar]

**ALGORITHM FOR TESTING BIPARTITENESS**

```
1 Bipartition adj[1..n]
2 colour[1..n] = white, ..., white
3 dist[1..n] = infinity, ..., infinity
4 for start = 1..n
5   if colour[start] = white
6     BFS(adj, start, colour, dist)
7     for edge in adj
8       let u and v be endpoints of edge
9       if dist[u] % 2 = dist[v] % 2
10          return NotBiPartite
B = nodes u with even dist[u]
R = nodes u with odd dist[u]
13 return B, R
```

**DEPTH-FIRST SEARCH OF A DIRECTED GRAPH**

A depth-first search uses a stack (or recursion) instead of a queue.
We define predecessors and colour vertices as in BFS.
It is also useful to specify a discovery time \(d[v]\) and a finishing time \(f[v]\)
for every vertex \(v\).
We increment a time counter every time a value \(d[v]\) or \(f[v]\) is assigned.
We eventually visit all the vertices, and the algorithm constructs a depth-first forest.

**DEPTH FIRST SEARCH ALGORITHM**

```
1 global variables
2 pre[1..n] = white, ..., white
3 colour[1..n] = white, ..., white
4 dist[1..n] = infinity, ..., infinity
5 time = 0
6 depthFirstSearch(u) { u \in V }
7   time = time + 1
8   if colour[u] = white
9     colour[u] = grey
10    time = time + 1
11    dist[u] = time
12    for each v \in adj[u]
13      if colour[v] = white
14        DepthFirstSearch(v)
15      edge = (u, v)
16     return
17   time = time + 1
18   colour[u] = white
19   return
```

**DFS TREE / FOREST**

As in breadth first search, \(pred[]\) array induces a forest

Let's match the graph's edge directions (opposite from \(pred\))

Could draw BFS forest this way also...

Each top level DFSVisit calls the root of a tree
Recall: DFSVisit(\(v\))
Basic DFS Properties to Remember

- Nodes start white
- A node \( v \) turns gray when it is discovered, which is when the first call to \( DFSVisit(v) \) happens
- After \( v \) is turned gray, we recurse on its neighbours
- After recursing on all neighbours, we turn \( v \) black
- Recursive calls on neighbours end
- Each call iterates over the neighbours.
- Effectively: “for each node, for each neighbour, do \( O(1) \) work + recurse.”
- \( d[v] \) is a discovery time
- \( f[v] \) is a finish time
- Each call returns with \( d[v] \) or \( f[v] \)

Classifying Edge in DFS

- If \( pred[v] = u \), then \((u,v)\) is a tree edge
- Else if \( v \) is a descendent of \( u \) in the DFS forest: forward edge
- Else if \( v \) is an ancestor of \( u \) in the DFS forest: back edge
- Else: \((u,v)\) is a cross edge

Explore \( D[] \), \( F[] \) and Colour]

Observe: every node \( v \) that is white-reachable from \( u \) when we first call \( DFSVisit(u) \) becomes gray after \( u \) and black before \( u \) (so \( f[u] \) is nested inside \( i_u \))

Runtime Complexity of DFS (for Adj. lists)

- Definition: we use \( i_u \) to denote \((d[u], f[u])\), which we call the interval of \( u \)
- Definition: \( v \) is white-reachable from \( u \) if there is a path from \( u \) to \( v \) containing only white nodes (excluding \( u \))

Definitions

- Theorem: Let \( u, v \) be any nodes.
  - The following statements are all equivalent.
  - \( v \) is white-reachable from \( u \) when we call \( DFSVisit(u) \)
  - \( v \) turns gray after \( u \) and black before \( u \)
  - \( (d[u], f[u]) \) interval of \( u \) is discovered during \( DFSVisit(u) \)
  - \( v \) is a descendant of \( u \) in the DFS forest
CLASSIFYING EDGE TYPES IN DFS
DFS inspects every edge in the graph.

When DFS inspects an edge \((u, v)\), the colour of \(v\) and relationship between the intervals of \(u\) and \(v\) determine the edge type.

<table>
<thead>
<tr>
<th>edge type</th>
<th>colour of (v)</th>
<th>discovery/finish times</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>white</td>
<td>(d(u) &lt; d(v) &lt; f(v) &lt; f(u))</td>
</tr>
<tr>
<td>back</td>
<td>black (or grey)</td>
<td>(d(u) &lt; d(v) &lt; f(u) &lt; f(v))</td>
</tr>
<tr>
<td>cross</td>
<td>grey</td>
<td>(d(u) &lt; f(u) &lt; d(v) &lt; f(v))</td>
</tr>
</tbody>
</table>

Recall: \((v\) is discovered during \(DFSVisit(u)\))
- \(v\) is white-reachable from \(u\) when we call \(DFSVisit(u)\)
- \(v\) is a descendant of \(u\) in the DFS tree
- \(v\) turns grey after \(u\) and block before \(u\)
- \(u\) nested inside \(T_u\)

It discovered during DFS visit
\(u\) is a child of \(v\) in the DFS tree
\(v\) is a descendant of \(u\)
\(v\) is already discovered
\(v\) is an ancestor of \(u\)
\(v\) is not a descendant and not an ancestor

\(u\) is not a descendent, and not an ancestor

\((v\) is discovered during \(DFSVisit(u)\))
- \(v\) is white-reachable from \(u\) when we call \(DFSVisit(u)\)
- \(v\) is a descendant of \(u\) in the DFS forest
- \(v\) turns grey after \(u\) and block before \(u\)
- \(u\) nested inside \(T_u\)

APPLICATION OF DFS: STRONG CONNECTEDNESS
Testing existence of all-to-all paths

CLASSIFYING EDGE TYPES IN DFS
DFS inspects every edge in the graph.

When DFS inspects an edge \((u, v)\), the colour of \(v\) and relationship between the intervals of \(u\) and \(v\) determine the edge type.

<table>
<thead>
<tr>
<th>edge type</th>
<th>colour of (v)</th>
<th>discovery/finish times</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree</td>
<td>white</td>
<td>(d(u) &lt; d(v) &lt; f(v) &lt; f(u))</td>
</tr>
<tr>
<td>back</td>
<td>black (or grey)</td>
<td>(d(u) &lt; d(v) &lt; f(u) &lt; f(v))</td>
</tr>
<tr>
<td>cross</td>
<td>grey</td>
<td>(d(u) &lt; f(u) &lt; d(v) &lt; f(v))</td>
</tr>
</tbody>
</table>

Recall: \((v\) is discovered during \(DFSVisit(u)\))
- \(v\) is white-reachable from \(u\) when we call \(DFSVisit(u)\)
- \(v\) is a descendant of \(u\) in the DFS tree
- \(v\) turns grey after \(u\) and block before \(u\)
- \(u\) nested inside \(T_u\)

If \(u\) were earlier, then \(v\) would be discovered before \(v\) finishes [because of edge \((u, v)\)], so intervals would not be disjoint!

\((v\) is discovered during \(DFSVisit(u)\))
- \(v\) is white-reachable from \(u\) when we call \(DFSVisit(u)\)
- \(v\) is a descendant of \(u\) in the DFS forest
- \(v\) turns grey after \(u\) and block before \(u\)
- \(u\) nested inside \(T_u\)

In a directed graph,
- \(v\) is reachable from \(w\) if there is a path from \(w\) to \(v\)
- we denote such a path \(w \rightarrow v\)

A graph \(G\) is strongly connected if
every node is reachable from every other node
- More formally: \(\forall w, v \in V \exists w \rightarrow v\)

STRONG CONNECTEDNESS
Is this graph strongly connected?
- Yes, One big cycle.

No path from \(c\) to other nodes.

How about this one?
STRONG CONNECTEDNESS

- How about this graph?

[Graph with multiple intersecting cycles]

- Yes, Multiple intersecting cycles.

- How about this one?

[Graph with two cycles and a one-directional path between them]

- No. Two cycles with only a one-directional path between them.

OTHER APPLICATIONS OF CHECKING STRONG CONNECTEDNESS

- You gain some symmetry from knowing a graph is strongly connected.

- For example, you can start a graph traversal at any node, and know the traversal will reach every node.

Without strong connectedness, if you want to run a graph traversal that reaches every node in a single pass, you would have to do additional processing to determine an appropriate starting node.

OTHER APPLICATIONS OF CHECKING STRONG CONNECTEDNESS

- Useful as a sanity check!

- Suppose you want to run an algorithm that requires strong connectedness, and you believe your input graph is strongly connected.

- Validate your input by testing whether this is true!

- Subtle, difficult-to-detect bugs often result if such an algorithm is run only on one component of a graph. (More concrete applications once we generalize and talk about strongly connected components...)

A USEFUL LEMMA

- Lemma: a graph is strongly connected if and only if for any node s, all nodes are reachable from s, and s is reachable from all nodes.

\[
\text{IsStronglyConnected}(G = (V, E)) \text{ where } V = v_1, v_2, ..., v_n
\]

\[
\text{(colour, d, f)} = \text{DFSVisit}(v_1, G)
\]

for \( i = 1..n \)

- if colour[v_i] \# black then return false

Construct graph H by reversing all edges in G

\[
\text{(colour, d, f)} = \text{DFSVisit}(v_1, H)
\]

for \( i = 1..n \)

- if colour[v_i] \# black then return false

return true

THE ALGORITHM

Create an algorithm to determine whether every node is reachable from a given node s.

How to use DFS to determine whether every node is reachable from a given node s?

How to use DFS to determine whether s is reachable from every node?

DFS from x and see if every node turns black.

DFS from x and see if every node turns black.

What if we first reverse the direction of every edge?

DFS from x in the original graph

DFS from x in the new graph iff s = x in the original graph.

PROVE BOTH DIRECTIONS:

(⇒) Suppose for all u, v we have u⇝v. Fix any s. Node s is reachable from all nodes, and vice versa.

For any u, v, we have s⇝u and s⇝v.

(⇐) Suppose s is reachable from all nodes and vice versa. Subtle, difficult-to-detect bugs often result if such an algorithm is run only on one component of a graph.
EXAMPLE EXECUTION 1

Every node is black. Next step!

\[ \text{DFSVisit}(a) \text{ in } G \]

(\(a\) is arbitrary)

Next step!

REVERSING EDGES:

Could the result change if we started at a different node?

EXAMPLE EXECUTION 2

REVERSING EDGES:

target

target
REVERSING EDGES: ADJACENCY MATRIX

reverse all edges

REVERSING EDGES: ADJACENCY MATRIX

reverse all edges

REVERSING EDGES: ADJACENCY MATRIX

reverse all edges

REVERSING EDGES: ADJACENCY MATRIX

reverse all edges

REVERSING EDGES: ADJACENCY MATRIX

reverse all edges

REVERSING EDGES: ADJACENCY MATRIX

reverse all edges
RUNTIME COMPLEXITY
FOR ADJACENCY LIST REPRESENTATION:

- IsStronglyConnected(G = (V, E)) where V = \{v₁, v₂, ..., vₙ\}
  - (colour, d, f) = DFSVisit(v₁, G)
  - for i = 1..n
    - if colour[vᵢ] ≠ black then return false
  - Construct graph H by reversing all edges in G
  - (colour, d, f) = DFSVisit(v₁, H)
  - for i = 1..n
    - if colour[vᵢ] ≠ black then return false
  - return true

**Complexity:** \(O(n + m)\)