CS 341: ALGORITHMS
Lecture 11: 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 $B$.

CRUCIAL PROPERTY:
NO ODD CYCLES
Claim: 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 $v_1, v_2, ..., v_{2k+1}, v_1$... until $v_{2k} \in B$

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 $v$.

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

![Diagram showing odd cycle](image)

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

**ALGORITHM FOR TESTING BIPARTITENESS**

1. Call BFS on each component to calculate distances for each node
2. Modified BFS that reuses the same colour array and same dist array
3. If both even or both odd
   - Return an actual bipartition
4. Runtime complexity?
   - Can be done in $O(n + m)$

**DEPTH-FIRST SEARCH**

- Breadth first search
- Depth first search

**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. Time = 0
2. for each node v:
   3. d[v] = Time
   4. f[v] = Time
   5. Call DFSVisit(v)

Example execution starting at node 1

DFS forest

DFS tree / forest

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

Let’s match the graph’s edge directions (opposite from pred)
```

Example execution starting at node 1

DFS forests
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 before $DFSVisit(v)$ does, so the neighbours of $v$ turn black before $v$

Also gets a discovery time $d[v]$ at this point

Also gets a finish time $f[v]$ at this point

CLASSIFYING EDGE $u \rightarrow v$ IN DFS

- If $pred(u) = u$, 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

Can we classify edges without inspecting the DFS forest?
Perhaps using $d[\cdot]$, $f[\cdot]$, $colour[\cdot]$

RUNTIME COMPLEXITY OF DFS (FOR ADJ. LISTS)

- Theorem: Let $u, v$ be any nodes.
  The following statements are all equivalent:
  - $(v$ is white-reachable from $u$ when we call $DFSVisit(u))$
  - $(v$ is discovered during $DFSVisit(u))$
  - $(v$ is discovered during $DFSVisit(u))$
  - $(v$ is a descendent of $u$ in the DFS forest)

DEFINITIONS

- 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 $u$)

EXPLORING 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 $I_u$ is nested inside $I_u$)

- **Consider the tree of recursive calls rooted at $DFSVisit(u)$:**
  - $v$ is discovered by a call in this tree
  - If $v$ is still discovered inside $I_u$
    - If $v$ is a descendent of $u$ in the DFS forest
      - $v$ turns gray after $u$ and black before $u$
    - If $v$ is white-reachable from $u$ when $DFSVisit(s)$ is called

EXPLORING D[], F[] AND COLOUR[]

- **Observe**: every node $v$ that is white-reachable from $u$ becomes gray after $u$ and black before $u$ (so $I_u$ is nested inside $I_u$)

- **Start $DFSVisit(u)$:**
  - Color $u$ grey, and set $u$’s discovery time

- **Perform $DFSVisit$ calls recursively:**
  - Color a black, set its finish time and return from $DFSVisit(u)$

SUMMARIZING IN A THEOREM

- **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 $u)$
  - $(v$ is discovered during $DFSVisit(u))$
  - $(v$ is discovered during $DFSVisit(u))$
  - $(v$ is a descendent 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>forward</td>
<td>white</td>
<td>([d(u),f(u)])</td>
</tr>
<tr>
<td>back</td>
<td>grey</td>
<td>([d(u),f(u)])</td>
</tr>
<tr>
<td>cross</td>
<td>black</td>
<td>([d(u),f(u)])</td>
</tr>
</tbody>
</table>

Recall: (\(v\) is discovered during dfsVisit\((u)\))

- \((u,v)\) is white-reachable from \(u\) when we call dfsVisit\((u)\)
- \((u,v)\) is a descendant of \(u\) in the DFS forest
- \((u,v)\) turns grey after \(u\) and block before \(v\)
- \([d(u),f(u)]\) nested inside \(I_u\)

Proof:
So, \(u\) must be earlier.
If \(I_u\) were earlier, then \(u\) would be discovered before \(v\) finishes (because of edge \((u,v)\)).
Intervals would not be disjoint.

If \(I_u\) were earlier, then \(v\) would be discovered before \(v\) finishes (because of edge \((u,v)\)).
So intervals would not be disjoint.

Intervals are nested.
QED

USEFUL FACT: PARENTHESES THEOREM

- Theorem: for each pair of nodes \(w, v\) in the intervals of \(w\) and \(v\) are either disjoint or nested
- Proof: Suppose the intervals are not disjoint.
  - Then either \(d(v) \in I_v\) or \(d[w] \in I_v\)
  - WLOG suppose \(d[w] \in I_v\)
  - Then \(v\) is discovered during dfsVisit\((u)\)
  - So \(v\) must turn grey after \(w\) and block before \(u\)
  - So \(f[w] < f[u]\)
  - So the intervals are nested. QED

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>forward</td>
<td>white</td>
<td>([d(u),f(u)])</td>
</tr>
<tr>
<td>back</td>
<td>grey</td>
<td>([d(u),f(u)])</td>
</tr>
<tr>
<td>cross</td>
<td>black</td>
<td>([d(u),f(u)])</td>
</tr>
</tbody>
</table>

Recall: (\(v\) is discovered during dfsVisit\((u)\))

- \((u,v)\) is white-reachable from \(u\) when we call dfsVisit\((u)\)
- \((u,v)\) is a descendant of \(u\) in the DFS forest
- \((u,v)\) turns grey after \(u\) and block before \(v\)
- \([d(u),f(u)]\) nested inside \(I_u\)

Proof:
So, \(u\) must be earlier.
If \(I_u\) were earlier, then \(v\) would be discovered before \(v\) finishes (because of edge \((u,v)\)).
Intervals would not be disjoint.

Intervals are nested.
QED

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>forward</td>
<td>white</td>
<td>([d(u),f(u)])</td>
</tr>
<tr>
<td>back</td>
<td>grey</td>
<td>([d(u),f(u)])</td>
</tr>
<tr>
<td>cross</td>
<td>black</td>
<td>([d(u),f(u)])</td>
</tr>
</tbody>
</table>

Recall: (\(v\) is discovered during dfsVisit\((u)\))

- \((u,v)\) is white-reachable from \(u\) when we call dfsVisit\((u)\)
- \((u,v)\) is a descendant of \(u\) in the DFS forest
- \((u,v)\) turns grey after \(u\) and block before \(v\)
- \([d(u),f(u)]\) nested inside \(I_u\)

Proof:
So, \(u\) must be earlier.
If \(I_u\) were earlier, then \(v\) would be discovered before \(v\) finishes (because of edge \((u,v)\)).
Intervals would not be disjoint.

Intervals are nested.
QED

STRONG CONNECTEDNESS

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

A graph \(G\) is strongly connected if
- every node is reachable from every other node.
- More formally: \(\forall u,v \in G \exists w \in G \text{ such that } w \to u \land w \to v\)

APPLICATION OF DFS (OR BFS): STRONG CONNECTEDNESS

Testing existence of all-to-all paths
**STRONG CONNECTEDNESS**

- Is this graph **strongly connected**?

  ![Graph 1](image1)

  - No path from c to other nodes.

  How about this one?

  ![Graph 2](image2)

  - Yes. One big cycle.

**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.

**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.

Proof: $(\Rightarrow)$ Suppose G is strongly connected. Then for all $u, v$ in G, we have $u \rightarrow v$. Fix any $s$. Node $s$ is reachable from all nodes, and vice versa.

$(\Leftarrow)$ Suppose some $s$ is reachable from all nodes and vice versa. For any $u, v$, we have $u \rightarrow v$ and $v \rightarrow u$. So G is strongly connected.

**CREATING AN ALGORITHM**

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

  DFS from $x$ and see if every node turns black.

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

  Then $x \rightarrow u$ in this new graph IFF $u \rightarrow x$ in the original graph.
THE ALGORITHM

- IsStronglyConnected($G = (V, E)$) 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

EXAMPLE EXECUTION 1

```
DFTVisit(a) in G
(a is arbitrary)
```

```
Every node is black. Next step!
```

```
DFTVisit(a) in H
```

```
Every node is black. Next step!
```

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

```
No path from those nodes to a
so G is not strongly connected!
```

EXAMPLE EXECUTION 2

```
DFTVisit(a) in G
(a is arbitrary)
```

```
Every node is black. Next step!
```

```
DFTVisit(a) in H
```

```
Some nodes are not black
```

```
No path from those nodes to a
so G is not strongly connected!
```

REVERSING EDGES:
ADJACENCY MATRIX

```
Source
```

```
Target
```

```
reverse all edges
```

```
reverse all edges
```

```
reverse all edges
```

```
reverse all edges
```

```
reverse all edges
```
REVERSING EDGES: ADJACENCY MATRIX

source target

reverse all edges

REVERSING EDGES: ADJACENCY MATRIX

source target

reverse all edges
REVERSING EDGES: ADJACENCY MATRIX

```
REVERSING EDGES:
ADJACENCY MATRIX

Can do matrix transpose, or can
just hold rows as columns and vice
versa in your code.

REVERSING EDGES:
ADJACENCY MATRIX

Complexity?

```