More Tricks with DFS

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

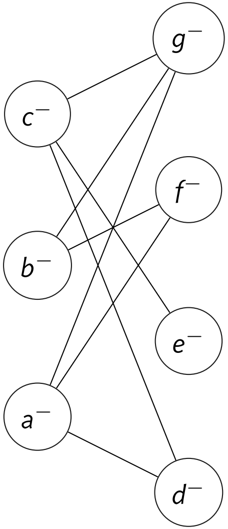
Objectives

Your Objectives: Use DFS to

- check if a graph is bipartite
- find articulation points
- find bridges (cut edges)
- see if a graph has cycles
- find strongly connected components

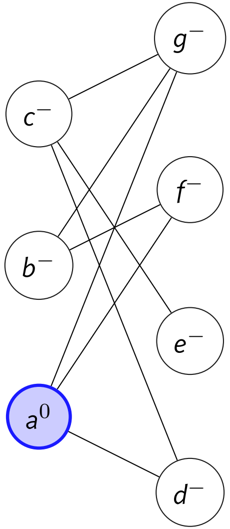
Check if a graph is bipartite

- Also called 2-coloring
- Use either BFS or DFS
- Start root with color 0
- Color each direct neighbor color 1
  For vertex `u` use `1 - color[u]` for neighbors.
- Recurse / Enqueue
- If you find an already visited neighbor with the same color as the parent, the graph is not bipartite.



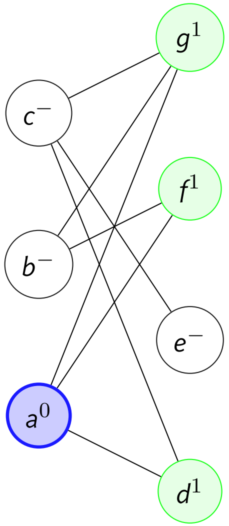
Check if a graph is bipartite

- Also called 2-coloring
- Use either BFS or DFS
- Start root with color 0
- Color each direct neighbor color 1
  For vertex `u` use `1 - color[u]` for neighbors.
- Recurse / Enqueue
- If you find an already visited neighbor with the same color as the parent, the graph is not bipartite.



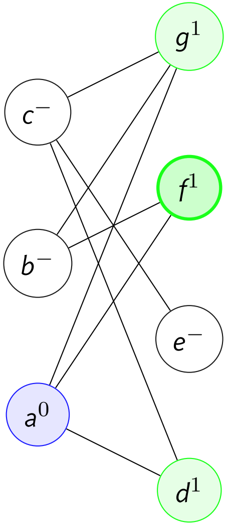
Check if a graph is bipartite

- Also called 2-coloring
- Use either BFS or DFS
- Start root with color 0
- Color each direct neighbor color 1
  For vertex `u` use `1 - color[u]` for neighbors.
- Recurse / Enqueue
- If you find an already visited neighbor with the same color as the parent, the graph is not bipartite.



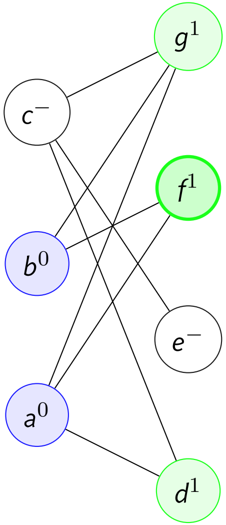
Check if a graph is bipartite

- Also called 2-coloring
- Use either BFS or DFS
- Start root with color 0
- Color each direct neighbor color 1
  For vertex `u` use `1 - color[u]` for neighbors.
- Recurse / Enqueue
- If you find an already visited neighbor with the same color as the parent, the graph is not bipartite.



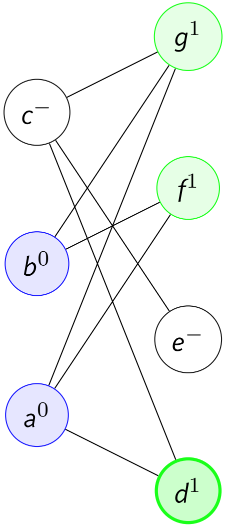
Check if a graph is bipartite

- Also called 2-coloring
- Use either BFS or DFS
- Start root with color 0
- Color each direct neighbor color 1
  For vertex `u` use `1 - color[u]` for neighbors.
- Recurse / Enqueue
- If you find an already visited neighbor with the same color as the parent, the graph is not bipartite.



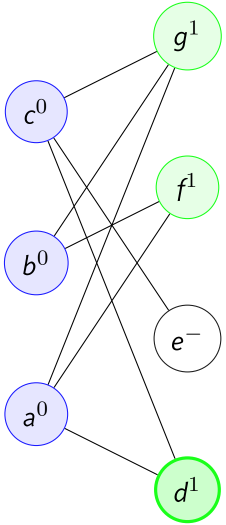
Check if a graph is bipartite

- Also called 2-coloring
- Use either BFS or DFS
- Start root with color 0
- Color each direct neighbor color 1
  For vertex `u` use `1 - color[u]` for neighbors.
- Recurse / Enqueue
- If you find an already visited neighbor with the same color as the parent, the graph is not bipartite.



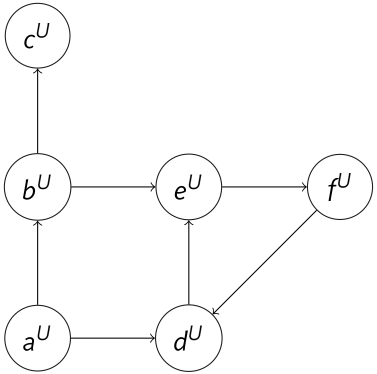
Check if a graph is bipartite

- Also called 2-coloring
- Use either BFS or DFS
- Start root with color 0
- Color each direct neighbor color 1
  For vertex `u` use `1 - color[u]` for neighbors.
- Recurse / Enqueue
- If you find an already visited neighbor with the same color as the parent, the graph is not bipartite.



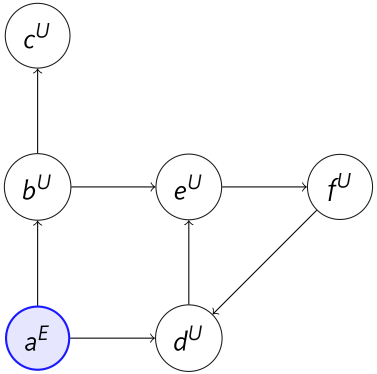
Detecting Cycles

- Use 3 states:
  - Unvisited
  - Explored — we entered the node but haven't finished it yet
  - Visited — mark when we are done with the node.
- Edge types:
  - Explored $\rightarrow$ Unvisited : Parent discovers new child
  - Explored $\rightarrow$ Visited: A forward or cross edge
  - Explored $\rightarrow$ Explored: A back edge / cycle



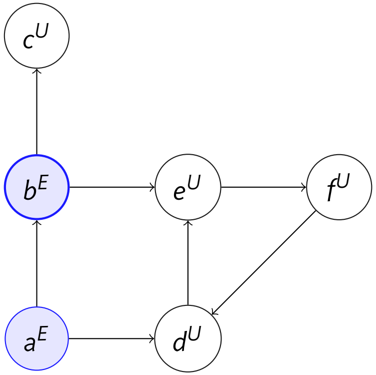
Detecting Cycles

- Use 3 states:
  - Unvisited
  - Explored — we entered the node but haven't finished it yet
  - Visited — mark when we are done with the node.
- Edge types:
  - Explored $\to$ Unvisited : Parent discovers new child
  - Explored $\to$ Visited: A forward or cross edge
  - Explored $\to$ Explored: A back edge / cycle



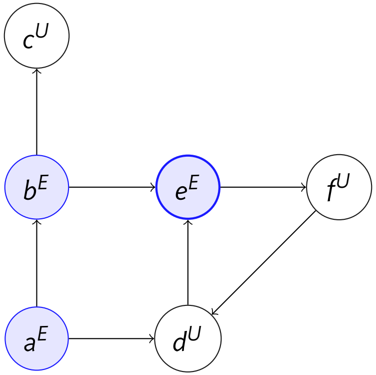
Detecting Cycles

- Use 3 states:
  - Unvisited
  - Explored — we entered the node but haven't finished it yet
  - Visited — mark when we are done with the node.
- Edge types:
  - Explored $\rightarrow$ Unvisited : Parent discovers new child
  - Explored $\rightarrow$ Visited: A forward or cross edge
  - Explored $\rightarrow$ Explored: A back edge / cycle



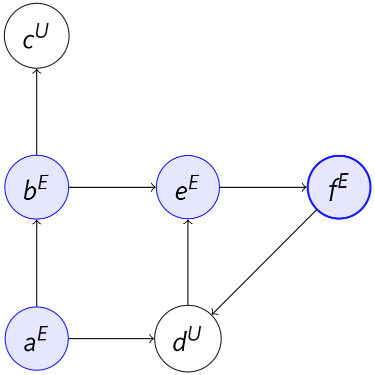
Detecting Cycles

- Use 3 states:
  - Unvisited
  - Explored — we entered the node but haven't finished it yet
  - Visited — mark when we are done with the node.
- Edge types:
  - Explored → Unvisited : Parent discovers new child
  - Explored → Visited: A forward or cross edge
  - Explored → Explored: A back edge / cycle



Detecting Cycles

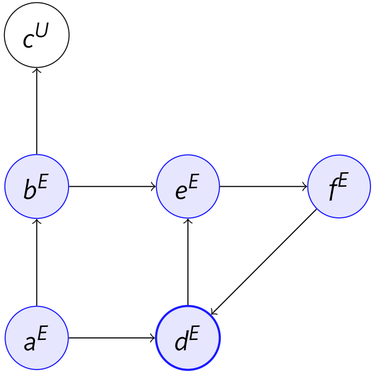
- Use 3 states:
  - Unvisited
  - Explored — we entered the node but haven't finished it yet
  - Visited — mark when we are done with the node.
- Edge types:
  - Explored $\to$ Unvisited : Parent discovers new child
  - Explored $\to$ Visited: A forward or cross edge
  - Explored $\to$ Explored: A back edge / cycle



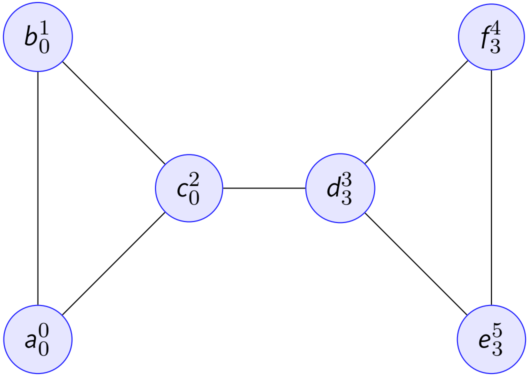
$$\begin{tikzcd}
c^U & & \\
b^E \arrow[u] \arrow[r] & e^E \arrow[r] & f^E \arrow[dl] \\
a^E \arrow[u] \arrow[r] & d^U \arrow[u] &
\end{tikzcd}$$

Detecting Cycles

- Use 3 states:
  - Unvisited
  - Explored — we entered the node but haven't finished it yet
  - Visited — mark when we are done with the node.
- Edge types:
  - Explored $\to$ Unvisited : Parent discovers new child
  - Explored $\to$ Visited: A forward or cross edge
  - Explored $\to$ Explored: A back edge / cycle

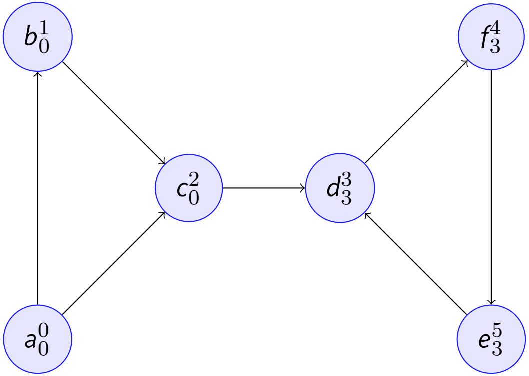

$$\begin{tikzcd} c^U & & \\ b^E \arrow[u] \arrow[r] & e^E \arrow[r] & f^E \arrow[dl] \\ a^E \arrow[u] \arrow[r] & d^E \arrow[u] & \end{tikzcd}$$

Finding Cut Nodes and Edges



- Superscript = `dfs_num`
- Subscript = `dfs_low`
- If `dfs_low[u] < dfs_num[u]`, then `u` belongs to a cycle.
- If `dfs_low[v] >= dfs_num[u]`, then `u` is a cut node.
- If `dfs_low[v] > dfs_num[u]`, then `u-v` is a cut edge.

Strongly Connected Components



- Superscript = `dfs_num`
- Subscript = `dfs_low`
- If `dfs_low[u] = dfs_num[u]`, then we have the root of a SCC.