Lecture 6

Graphs II – DFS Applications and Friends

Karl Bringmann, Dominic Zimmer

Saarland University

June, 2019

Midterm-Contest

- Date: 20.06.2020 at 10:00
  Exam duration: 2:30h.
  Plan for the contest to end at about 13:00
- Topics: Lecture 1 - Lecture 6 (this lecture)
  Lecture next week is not relevant for the Midterm contest
- Cheat-Sheet
  - 100.000 characters, `.txt` file
  - Submission on your personal status page
  - Submit until 18.06. 23:59
  - Contents: Algorithms from the lecture, your template, your previous submissions, ...
- Virtual Machine
  - Mandatory for taking part in the exams
  - Published this week
  - Install as soon as possible!

Recap

Past Topics

- Traversals
  - BFS
  - DFS
  - Connected Components
  - Topological Sorting
- Shortest Paths
  - Dijkstra
  - Bellman-Ford
    - Shortest Path Faster Algorithm
  - Floyd-Warshall

Recap

Shortest Paths

Our Options for solving SSSP are

	BFS	Dijkstra	Bellman-Ford	Floyd-Warshall
Running Time	$\mathcal{O}(V+E)$	$\mathcal{O}((V+E)\log V)$	$\mathcal{O}(VE)$	$\mathcal{O}(V^3)$
Max Size	$V, E \leq 10^7$	$V, E \leq 10^6$	$V \cdot E \leq 10^7$	$V \leq 500$
Unweighted	✓	✓	✓	✓
Weighted	✗	✓	✓	✓
Negative Weights	✗	✗	✓	✓
Sparse ($E \approx V$)	$\mathcal{O}(V)$	$\mathcal{O}(V \log V)$	$\mathcal{O}(V^2)$	$\mathcal{O}(V^3)$
Dense ($E \approx V^2$)	$\mathcal{O}(V^2)$	$\mathcal{O}(V^2 \log V)$ / $\mathcal{O}(V^2)^*$	$\mathcal{O}(V^3)$	$\mathcal{O}(V^3)$

Recap

Shortest Paths

Our Options for solving SSSP are

	BFS	Dijkstra	Bellman-Ford	Floyd-Warshall
Running Time	$\mathcal{O}(V+E)$	$\mathcal{O}((V+E)\log V)$	$\mathcal{O}(VE)$	$\mathcal{O}(V^3)$
Max Size	$V, E \leq 10^7$	$V, E \leq 10^6$	$V \cdot E \leq 10^7$	$V \leq 500$
Unweighted	✓	✓	✓	✓
Weighted	✗	✓	✓	✓
Negative Weights	✗	✗	✓	✓
Sparse ($E \approx V$)	$\mathcal{O}(V)$	$\mathcal{O}(V \log V)$	$\mathcal{O}(V^2)$	$\mathcal{O}(V^3)$
Dense ($E \approx V^2$)	$\mathcal{O}(V^2)$	$\mathcal{O}(V^2 \log V)$ / $\mathcal{O}(V^2)^*$	$\mathcal{O}(V^3)$	$\mathcal{O}(V^3)$

* : requires adaptation of our implementation to one of $\mathcal{O}(V^2)$

Recap

Shortest Paths

Our Options for solving APSP are

	BFS	Dijkstra	Bellman-Ford	Floyd-Warshall
Running Time	$\mathcal{O}(V \cdot (V+E))$	$\mathcal{O}(V \cdot ((V+E)\log V))$	$\mathcal{O}(V \cdot (VE))$	$\mathcal{O}(V^3)$
Unweighted	✓	✓	✓	✓
Weighted	✗	✓	✓	✓
Negative Weights	✗	✗	✓	✓
Sparse ($E \approx V$)	$\mathcal{O}(V^2)$	$\mathcal{O}(V^2 \log V)$	$\mathcal{O}(V^3)$	$\mathcal{O}(V^3)$
Dense ($E \approx V^2$)	$\mathcal{O}(V^3)$	$\mathcal{O}(V^3 \log V)$ / $\mathcal{O}(V^3)^*$	$\mathcal{O}(V^4)$	$\mathcal{O}(V^3)$

* : requires adaptation of our implementation to one of $\mathcal{O}(V^2)$

Recap

Upcoming Topics

- Traversals
  - BFS
  - DFS
  - Connected Components
  - Topological Sorting
- Shortest Paths
  - Dijkstra
  - Bellman-Ford
    - Shortest Path Faster Algorithm
  - Floyd-Warshall

Recap

Upcoming Topics

- Traversals
  - BFS
  - DFS
  - Connected Components
  - Topological Sorting
- Shortest Paths
  - Dijkstra
  - Bellman-Ford
    - Shortest Path Faster Algorithm
  - Floyd-Warshall

Recap

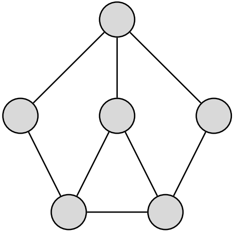
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
        if (visited[u] == UNVISITED)
            dfs(u);


   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- **Bold** arrows show the *DFS-Tree*

Recap

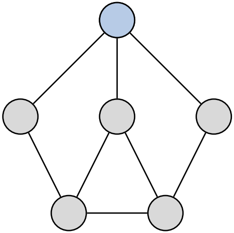
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
           dfs(u);


   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- **Bold** arrows show the *DFS-Tree*

Recap

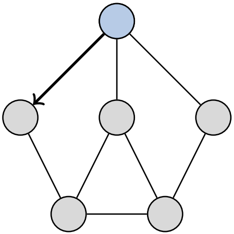
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
           dfs(u);


   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- **Bold** arrows show the *DFS-Tree*

Recap

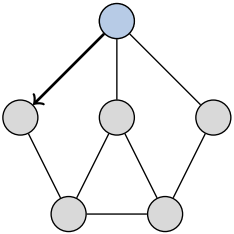
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
           dfs(u);


   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

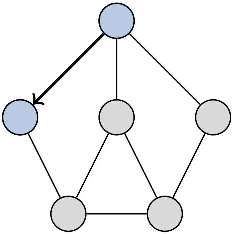
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
           dfs(u);


   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

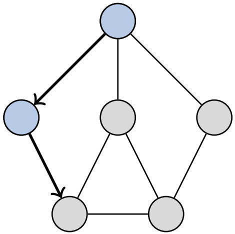
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
           dfs(u);


   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

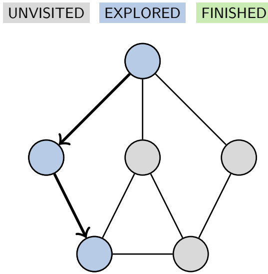
Recap

Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
           dfs(u);


   }
   visited[v] = FINISHED;
}
```



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

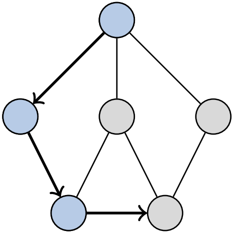
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
   visited[v] = EXPLORED;
    for (auto u: adj[v]) {
         if (visited[u] == UNVISITED)
              dfs(u);


    }
    visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

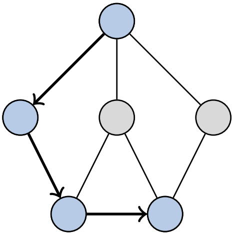
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
           dfs(u);


   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

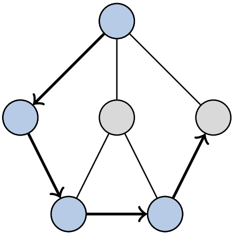
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
           dfs(u);


   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

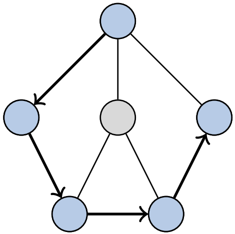
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
        if (visited[u] == UNVISITED)
            dfs(u);


   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

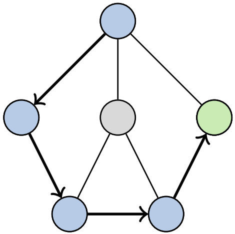
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
        if (visited[u] == UNVISITED)
             dfs(u);


   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

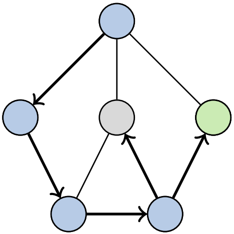
Example Code: DFS

```cpp
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
           dfs(u);


   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

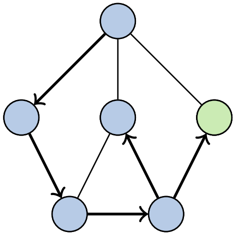
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
           dfs(u);


   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

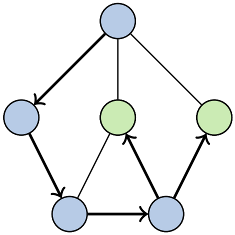
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
           dfs(u);


   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

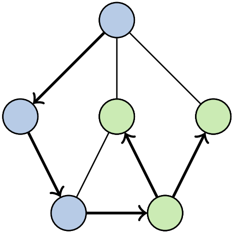
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
           dfs(u);


   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

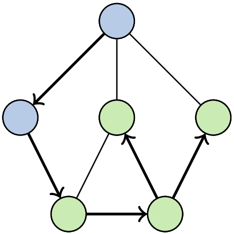
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
        if (visited[u] == UNVISITED)
            dfs(u);


   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

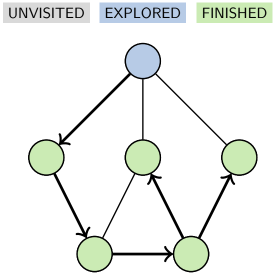
Recap

Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
           dfs(u);


   }
   visited[v] = FINISHED;
}
```



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

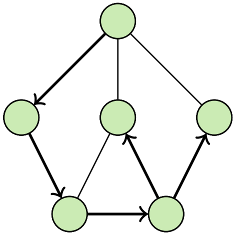
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
           dfs(u);


   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

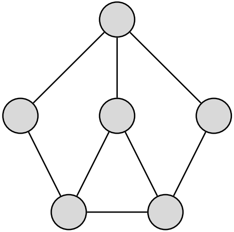
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);


     }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

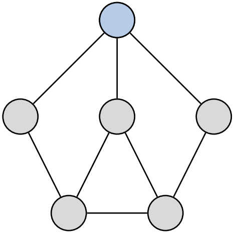
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);


     }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

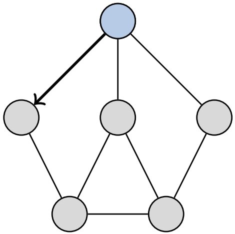
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);


     }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

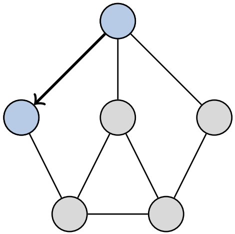
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);


     }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

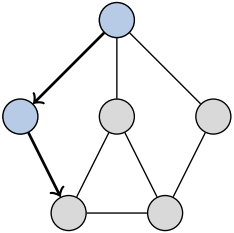
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);


     }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

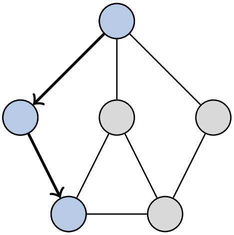
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);


     }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

max planck institut informatik

Recap

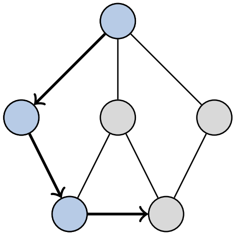
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);


     }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

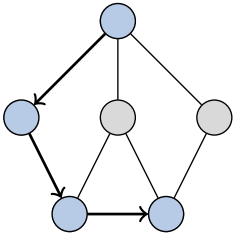
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);


     }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

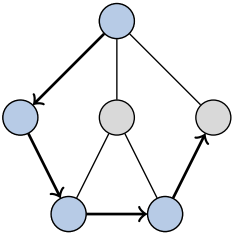
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);


     }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

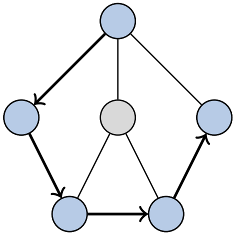
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);


     }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

Recap

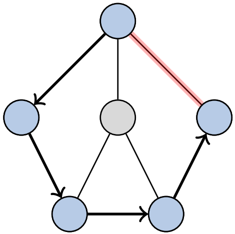
Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);


     }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**

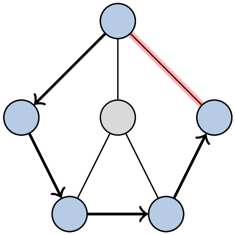
Recap

Example Code: DFS

```cpp
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
        if (visited[u] == UNVISITED)
          dfs(u);
        else if (visited[u] == EXPLORED)
          cout << "Back edge: " << u << "-" << v << endl;
   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**
- Edges $u \longrightarrow v$ are called Back Edges

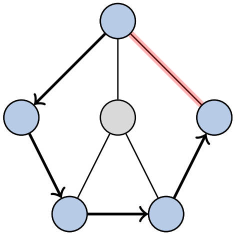
Recap

Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);
       else if (visited[u] == EXPLORED)
         cout << "Back edge: " << u << "-" << v << endl;
   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**
- Edges $u \longrightarrow v$ are called Back Edges
  - Back Edges indicate cycles in the original graph

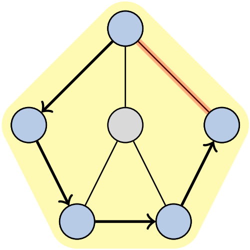
Recap

Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);
       else if (visited[u] == EXPLORED)
         cout << "Back edge: " << u << "-" << v << endl;
   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**
- Edges $u \longrightarrow v$ are called Back Edges
  - Back Edges indicate cycles in the original graph

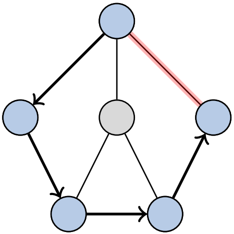
Recap

Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
        if (visited[u] == UNVISITED)
          dfs(u);
        else if (visited[u] == EXPLORED)
          cout << "Back edge: " << u << "-" << v << endl;
   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**
- Edges $u \longrightarrow v$ are called Back Edges
  - Back Edges indicate cycles in the original graph

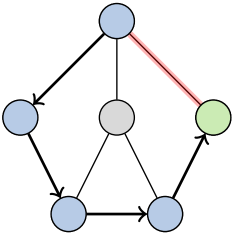
Recap

Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
        if (visited[u] == UNVISITED)
          dfs(u);
        else if (visited[u] == EXPLORED)
          cout << "Back edge: " << u << "-" << v << endl;
   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**
- Edges $u \longrightarrow v$ are called Back Edges
  - Back Edges indicate cycles in the original graph

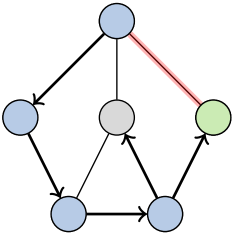
Recap

Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);
       else if (visited[u] == EXPLORED)
         cout << "Back edge: " << u << "-" << v << endl;
   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**
- Edges $u \longrightarrow v$ are called Back Edges
  - Back Edges indicate cycles in the original graph

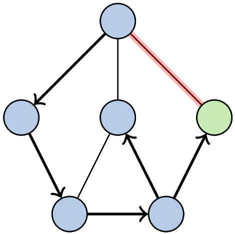
Recap

Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);
       else if (visited[u] == EXPLORED)
         cout << "Back edge: " << u << "-" << v << endl;
   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**
- Edges $u \longrightarrow v$ are called Back Edges
  - Back Edges indicate cycles in the original graph

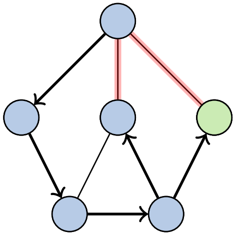
Recap

Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);
       else if (visited[u] == EXPLORED)
         cout << "Back edge: " << u << "-" << v << endl;
   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**
- Edges $u \longrightarrow v$ are called Back Edges
  - Back Edges indicate cycles in the original graph

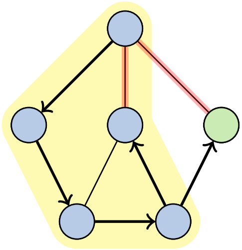
Recap

Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);
       else if (visited[u] == EXPLORED)
         cout << "Back edge: " << u << "-" << v << endl;
   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**
- Edges $u \longrightarrow v$ are called Back Edges
  - Back Edges indicate cycles in the original graph

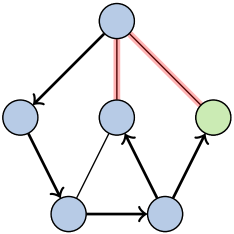
Recap

Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
        if (visited[u] == UNVISITED)
          dfs(u);
        else if (visited[u] == EXPLORED)
          cout << "Back edge: " << u << "-" << v << endl;
   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**
- Edges $u \longrightarrow v$ are called Back Edges
  - Back Edges indicate cycles in the original graph

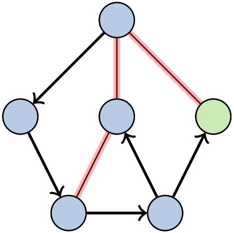
Recap

Example Code: DFS

```cpp
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
        if (visited[u] == UNVISITED)
          dfs(u);
        else if (visited[u] == EXPLORED)
          cout << "Back edge: " << u << "-" << v << endl;
   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**
- Edges $u \longrightarrow v$ are called Back Edges
  - Back Edges indicate cycles in the original graph

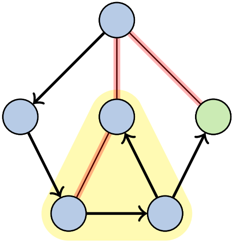
Recap

Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
        if (visited[u] == UNVISITED)
          dfs(u);
        else if (visited[u] == EXPLORED)
          cout << "Back edge: " << u << "-" << v << endl;
   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**
- Edges $u \longrightarrow v$ are called Back Edges
  - Back Edges indicate cycles in the original graph

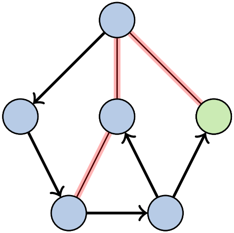
Recap

Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);
       else if (visited[u] == EXPLORED)
         cout << "Back edge: " << u << "-" << v << endl;
   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**
- Edges $u \longrightarrow v$ are called Back Edges
  - Back Edges indicate cycles in the original graph

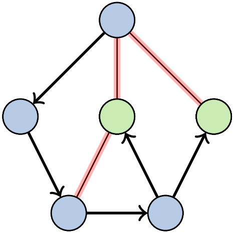
Recap

Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);
       else if (visited[u] == EXPLORED)
         cout << "Back edge: " << u << "-" << v << endl;
   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**
- Edges $u \longrightarrow v$ are called Back Edges
  - Back Edges indicate cycles in the original graph

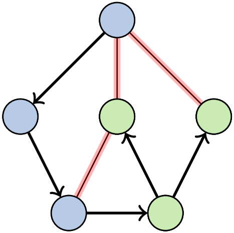
Recap

Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);
       else if (visited[u] == EXPLORED)
         cout << "Back edge: " << u << "-" << v << endl;
   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**
- Edges $u \longrightarrow v$ are called Back Edges
  - Back Edges indicate cycles in the original graph

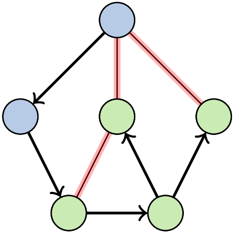
Recap

Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);
       else if (visited[u] == EXPLORED)
         cout << "Back edge: " << u << "-" << v << endl;
   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**
- Edges $u \longrightarrow v$ are called Back Edges
  - Back Edges indicate cycles in the original graph

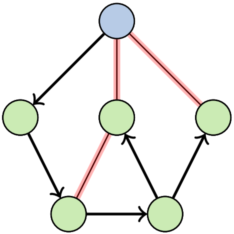
Recap

Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);
       else if (visited[u] == EXPLORED)
         cout << "Back edge: " << u << "-" << v << endl;
   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**
- Edges $u \longrightarrow v$ are called Back Edges
  - Back Edges indicate cycles in the original graph

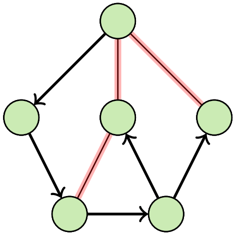
Recap

Example Code: DFS

```
vector<int> visited(V, UNVISITED);

void dfs(int v) {
  visited[v] = EXPLORED;
   for (auto u: adj[v]) {
       if (visited[u] == UNVISITED)
         dfs(u);
       else if (visited[u] == EXPLORED)
         cout << "Back edge: " << u << "-" << v << endl;
   }
   visited[v] = FINISHED;
}
```

UNVISITED EXPLORED FINISHED



- Edges $u \longrightarrow v$ are called **Tree Edges**
- Edges $u \longrightarrow v$ are called Back Edges
  - Back Edges indicate cycles in the original graph

Bipartite Graphs

Definition: Bipartite

An undirected graph is called bipartite, if its vertices can be partitioned into disjoint sets L and R, such that there are no edges between the vertices of either of those sets.

Bipartite Graphs

Definition: Bipartite

An undirected graph is called bipartite, if its vertices can be partitioned into disjoint sets L and R, such that there are no edges between the vertices of either of those sets.

Problem: Bipartite Check

Given an undirected graph G, is it bipartite?

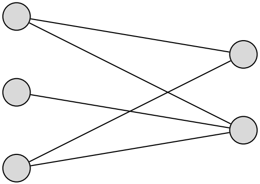
Bipartite Graphs

Definition: Bipartite

An undirected graph is called bipartite, if its vertices can be partitioned into disjoint sets L and R, such that there are no edges between the vertices of either of those sets.

Problem: Bipartite Check

Given an undirected graph G, is it bipartite?



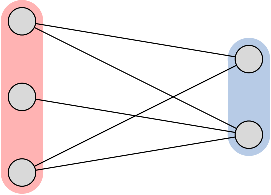
Bipartite Graphs

Definition: Bipartite

An undirected graph is called bipartite, if its vertices can be partitioned into disjoint sets L and R, such that there are no edges between the vertices of either of those sets.

Problem: Bipartite Check

Given an undirected graph G, is it bipartite?



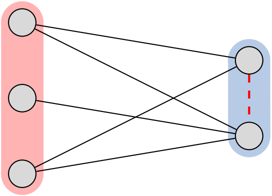
Bipartite Graphs

Definition: Bipartite

An undirected graph is called bipartite, if its vertices can be partitioned into disjoint sets L and R, such that there are no edges between the vertices of either of those sets.

Problem: Bipartite Check

Given an undirected graph G, is it bipartite?



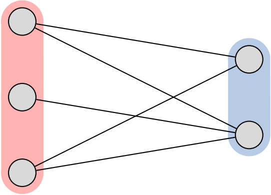
Bipartite Graphs

Definition: Bipartite

An undirected graph is called bipartite, if its vertices can be partitioned into disjoint sets L and R, such that there are no edges between the vertices of either of those sets.

Problem: Bipartite Check

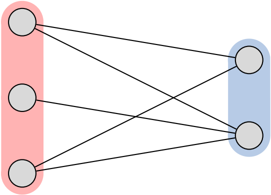
Given an undirected graph G, is it bipartite?



Bipartite Graphs

Problem: Bipartite Check

Given an undirected graph G, is it bipartite?



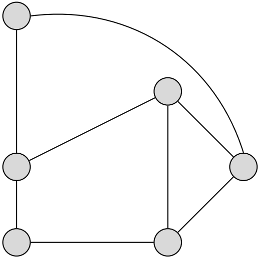
Theorem

An undirected graph is bipartite if and only if it can be 2-colored, i.e. no edge has the same color on both ends

Bipartite Graphs

Problem: Bipartite Check

Given a graph G, is it bipartite?



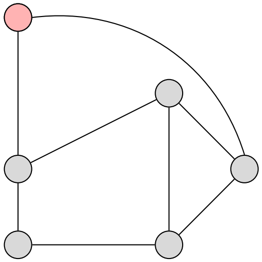
Algorithm Idea

Traverse the graph and try to 2-color the vertices

Bipartite Graphs

Problem: Bipartite Check

Given a graph G, is it bipartite?



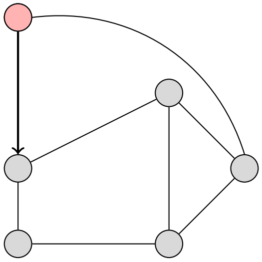
Algorithm Idea

Traverse the graph and try to 2-color the vertices

Bipartite Graphs

Problem: Bipartite Check

Given a graph G, is it bipartite?



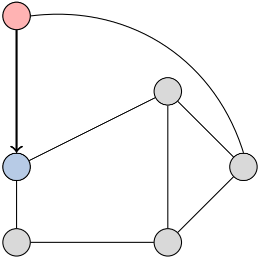
Algorithm Idea

Traverse the graph and try to 2-color the vertices

Bipartite Graphs

Problem: Bipartite Check

Given a graph G, is it bipartite?



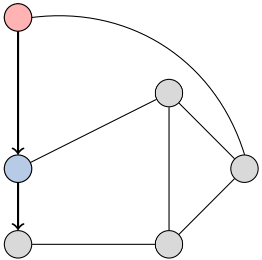
Algorithm Idea

Traverse the graph and try to 2-color the vertices

Bipartite Graphs

Problem: Bipartite Check

Given a graph G, is it bipartite?



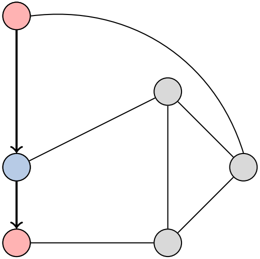
Algorithm Idea

Traverse the graph and try to 2-color the vertices

Bipartite Graphs

Problem: Bipartite Check

Given a graph G, is it bipartite?



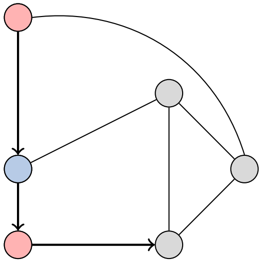
Algorithm Idea

Traverse the graph and try to 2-color the vertices

Bipartite Graphs

Problem: Bipartite Check

Given a graph G, is it bipartite?



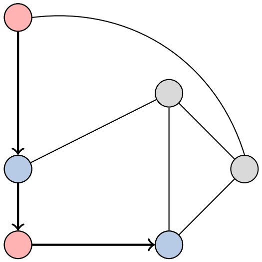
Algorithm Idea

Traverse the graph and try to 2-color the vertices

Bipartite Graphs

Problem: Bipartite Check

Given a graph G, is it bipartite?



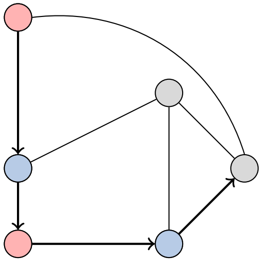
Algorithm Idea

Traverse the graph and try to 2-color the vertices

Bipartite Graphs

Problem: Bipartite Check

Given a graph G, is it bipartite?



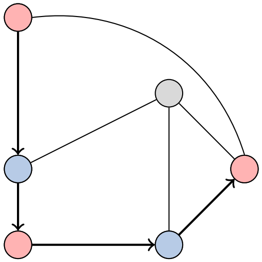
Algorithm Idea

Traverse the graph and try to 2-color the vertices

Bipartite Graphs

Problem: Bipartite Check

Given a graph G, is it bipartite?



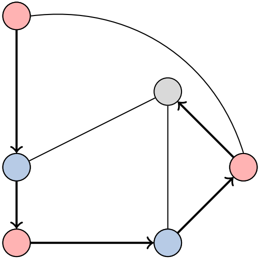
Algorithm Idea

Traverse the graph and try to 2-color the vertices

Bipartite Graphs

Problem: Bipartite Check

Given a graph G, is it bipartite?



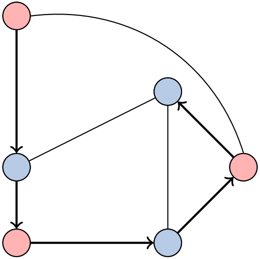
Algorithm Idea

Traverse the graph and try to 2-color the vertices

Bipartite Graphs

Problem: Bipartite Check

Given a graph G, is it bipartite?



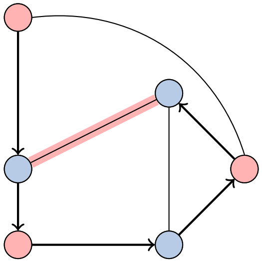
Algorithm Idea

Traverse the graph and try to 2-color the vertices

Bipartite Graphs

Problem: Bipartite Check

Given a graph G, is it bipartite?



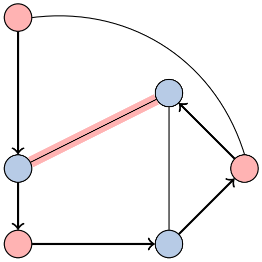
Algorithm Idea

Traverse the graph and try to 2-color the vertices

Bipartite Graphs

Problem: Bipartite Check

Given a graph G, is it bipartite?



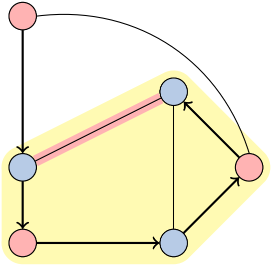
The following statements are equivalent

- G is bipartite
- G is 2-colorable

Bipartite Graphs

Problem: Bipartite Check

Given a graph G, is it bipartite?



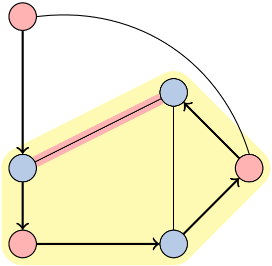
The following statements are equivalent

- G is bipartite
- G is 2-colorable

Bipartite Graphs

Problem: Bipartite Check

Given a graph G, is it bipartite?



The following statements are equivalent

- G is bipartite
- G is 2-colorable
- G has no cycle of odd length

Bipartite Graphs

Example Code: Bipartite-Check

```
vector<int> colors(V, -1); // -1 means unvisited

void dfs(int v) {
  for (auto u: adj[v])
    if (colors[u] == -1) {
      colors[u] = 1 - colors[v];
      dfs(u);
    } else if (colors[u] == colors[v]) {
      cout << "Impossible" << endl;
      exit(0);
    }
}

colors[start] = 0; // colors are 0, 1
// assume adj to be connected
dfs(start);
cout << "Possible" << endl;
```

Articulation Points & Bridges

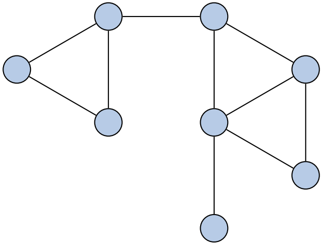
Sabotage

Problem: Sabotage

A terrorist organization is threatening to attack the city.

Last week we made sure that every citizen can reach everyone else. The Terrorists will try to disconnect the city by blocking either one intersection or one road.

Which intersections and roads do we need to protect?



Articulation Points & Bridges

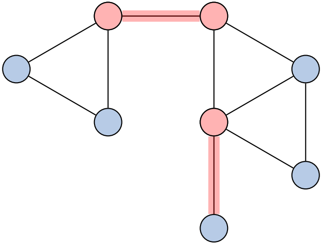
Sabotage

Problem: Sabotage

A terrorist organization is threatening to attack the city.

Last week we made sure that every citizen can reach everyone else. The Terrorists will try to disconnect the city by blocking either one intersection or one road.

Which intersections and roads do we need to protect?



Articulation Points & Bridges

Sabotage

Problem: Sabotage

A terrorist organization is threatening to attack the city.

Last week we made sure that every citizen can reach everyone else. The Terrorists will try to disconnect the city by blocking either one intersection or one road.

Which intersections and roads do we need to protect?

Definition: Articulation Point

A vertex whose removal (as well as its incident edges) increases the number of connected components is called an *Articulation Point*.

Definition: Bridge

An edge whose removal increases the number of connected components is called a *Bridge*.

Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

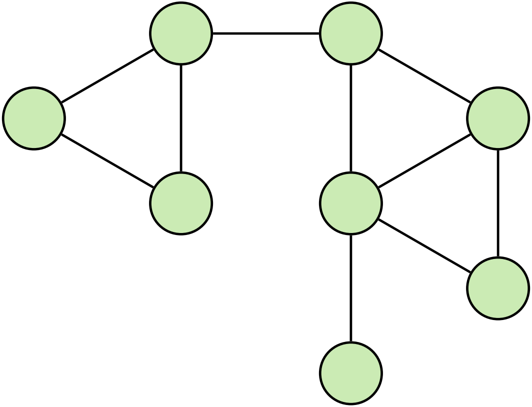
For a vertex v of a DFS tree, we denote by

- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge

Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

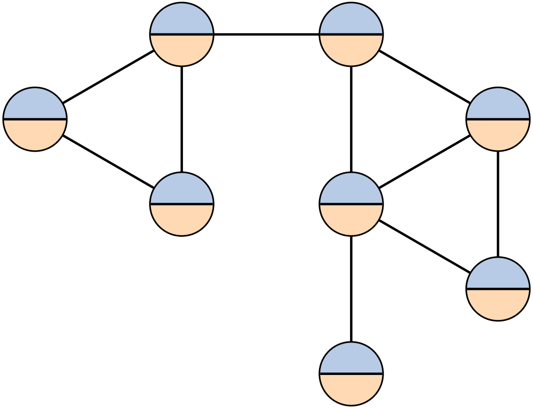
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

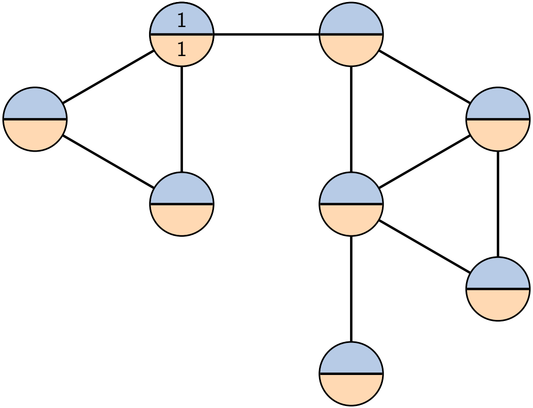
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

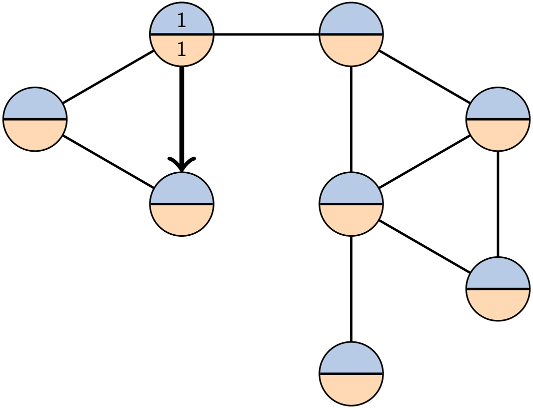
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

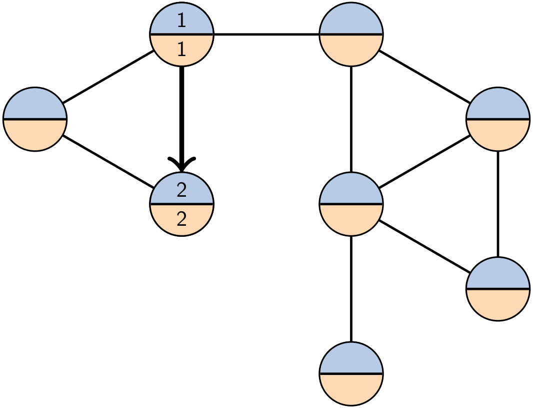
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

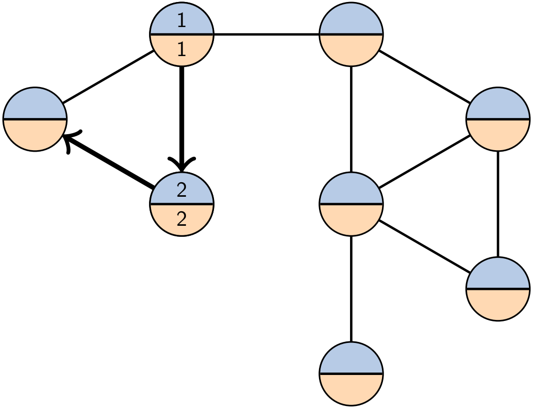
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

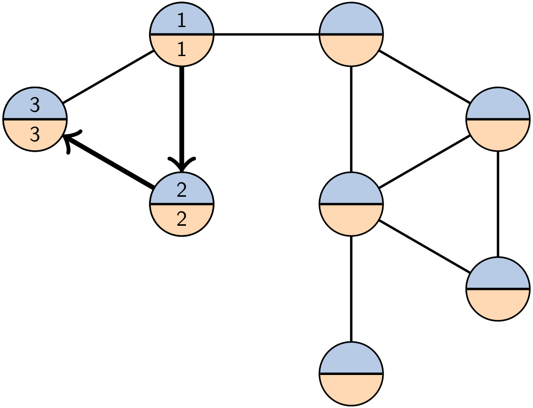
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

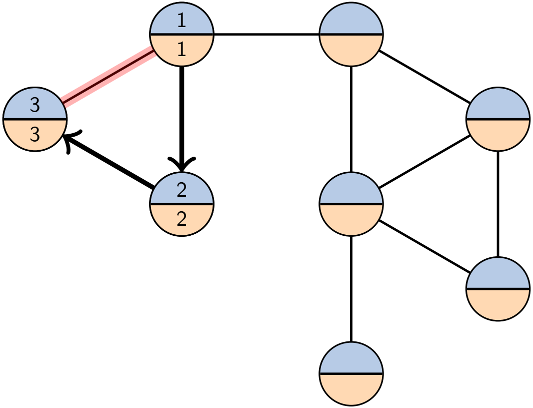
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

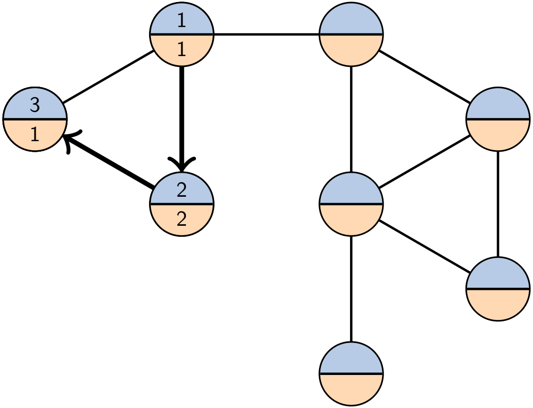
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

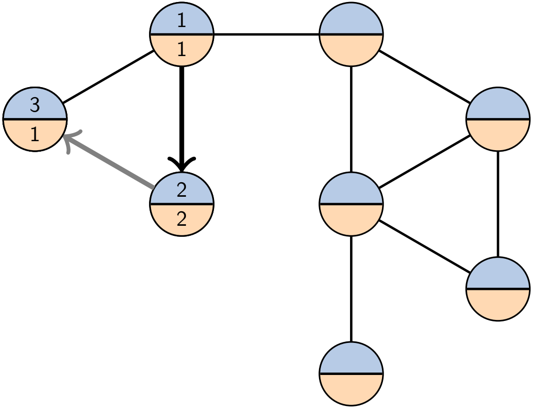
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

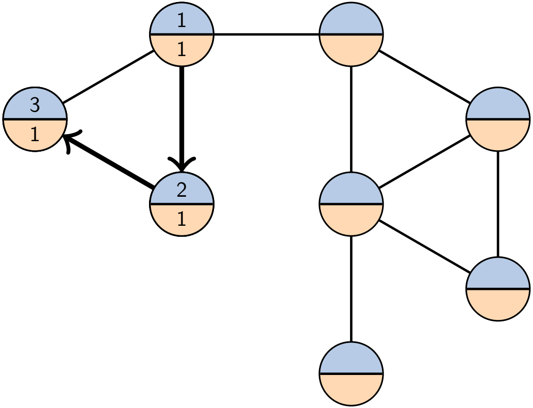
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

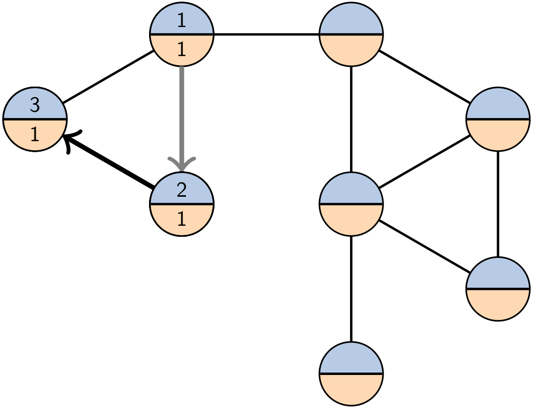
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: dfs_num and dfs_min

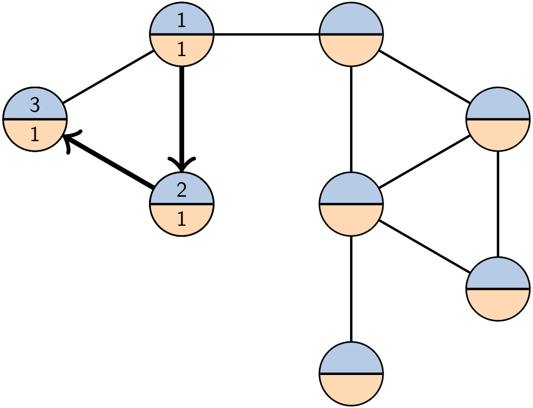
- dfs_num[v] the timestamp at which v was explored
- dfs_min[v] the minimum dfs_num reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

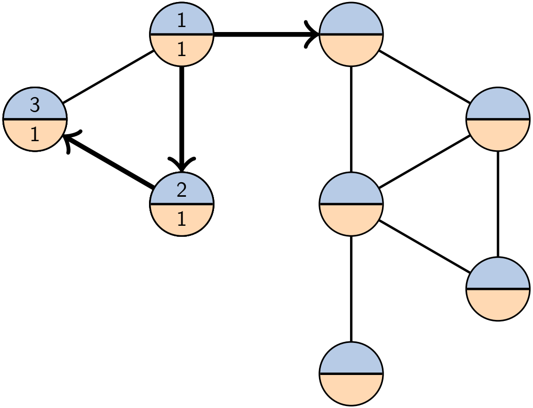
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

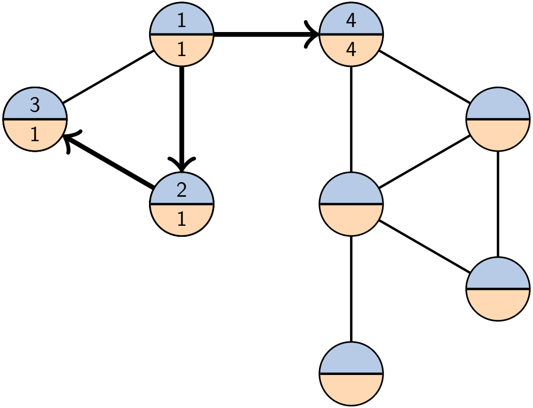
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

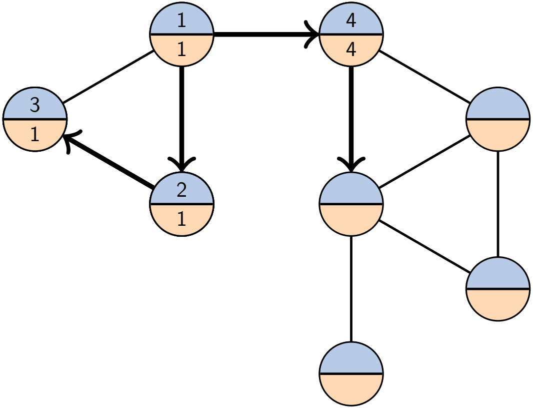
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

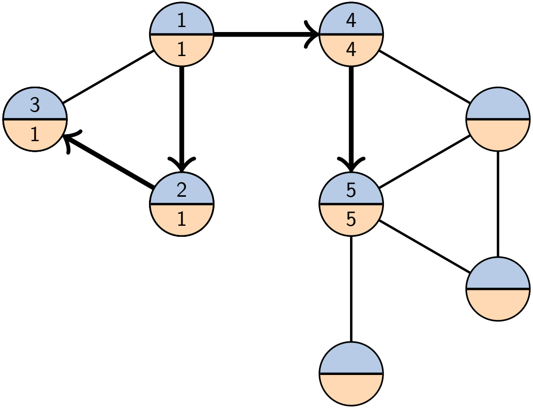
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

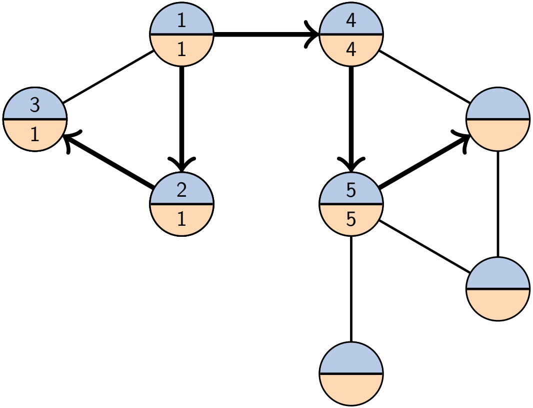
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

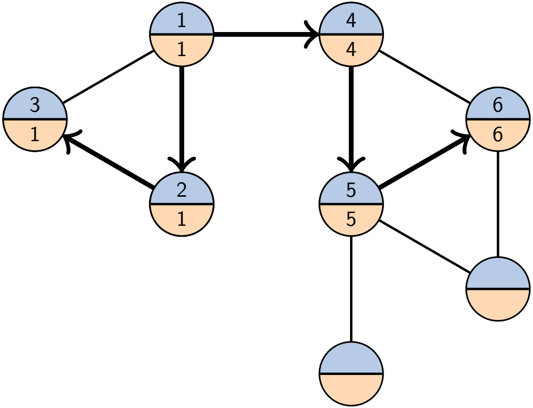
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

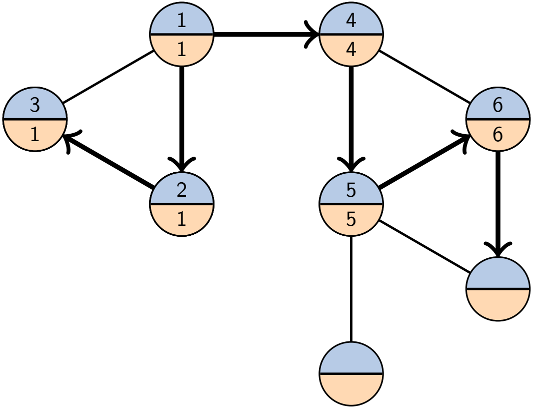
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

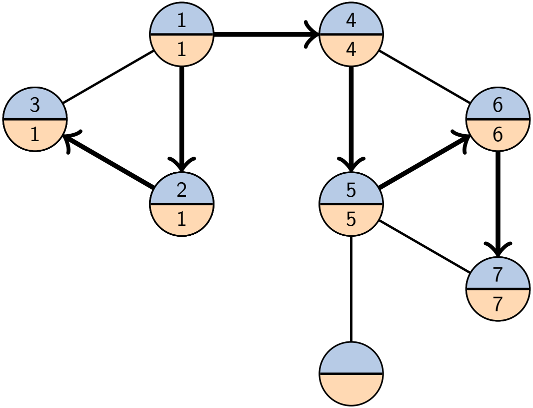
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

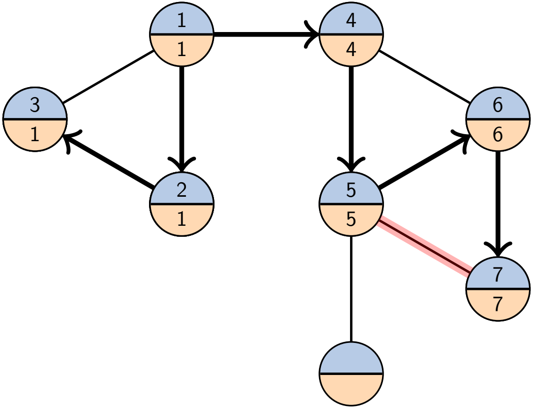
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

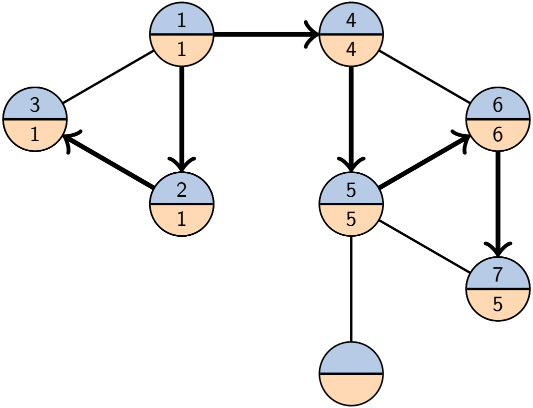
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

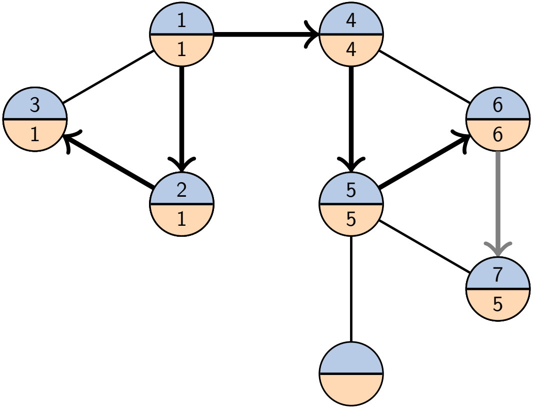
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

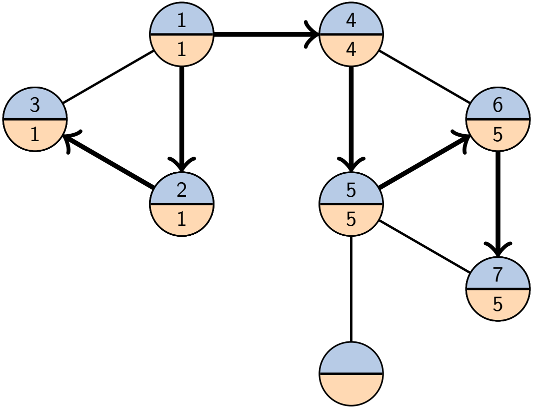
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

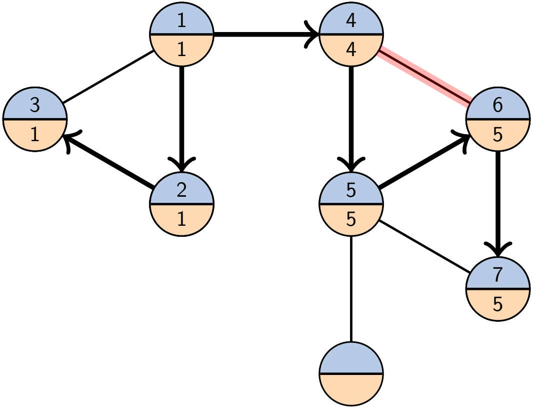
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

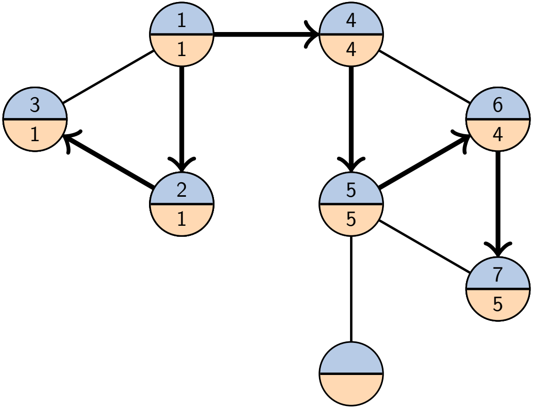
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

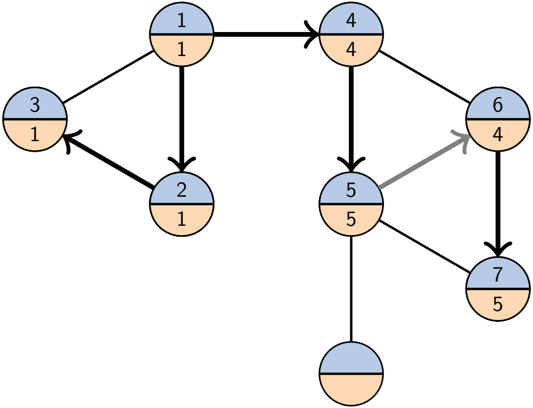
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

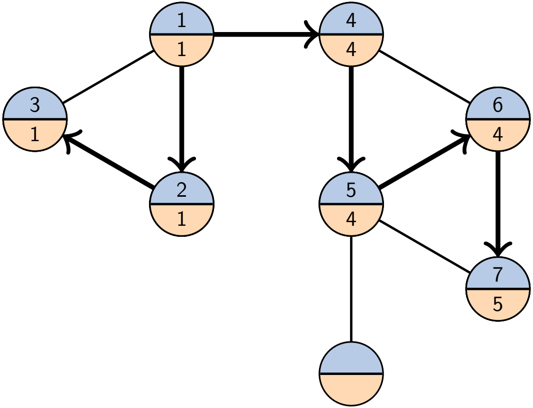
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

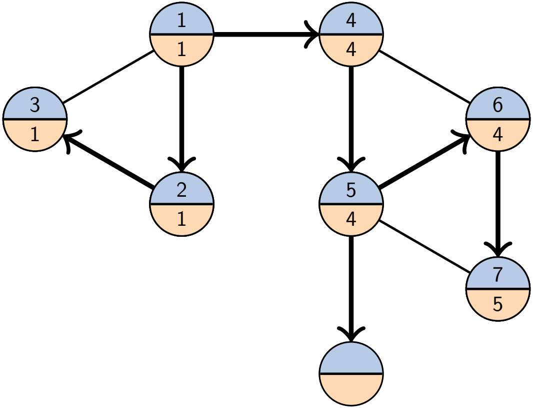
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

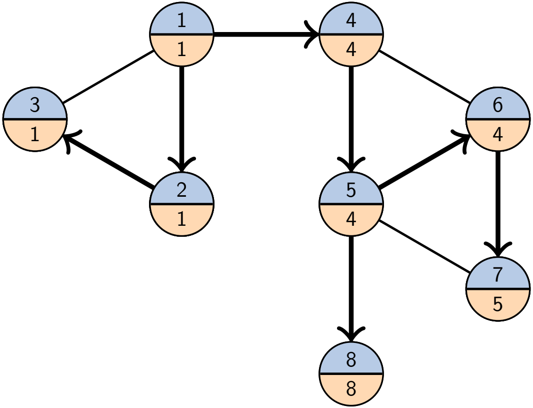
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

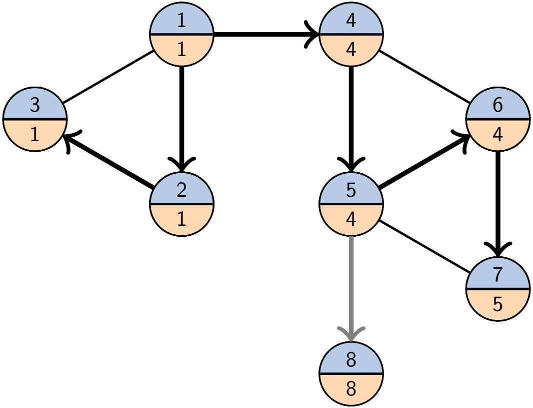
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

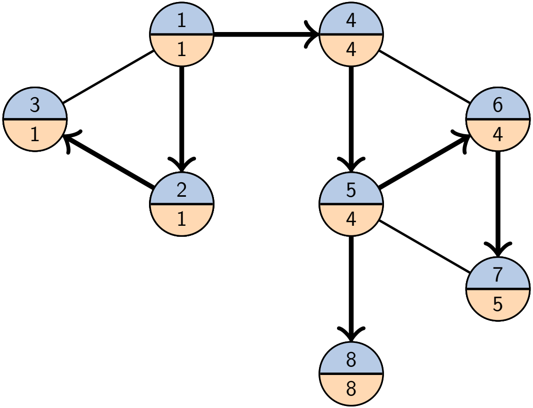
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

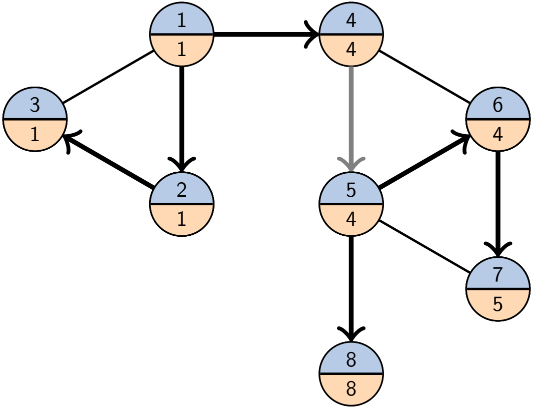
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

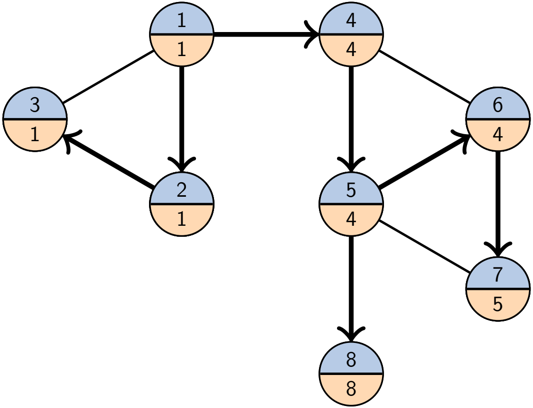
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

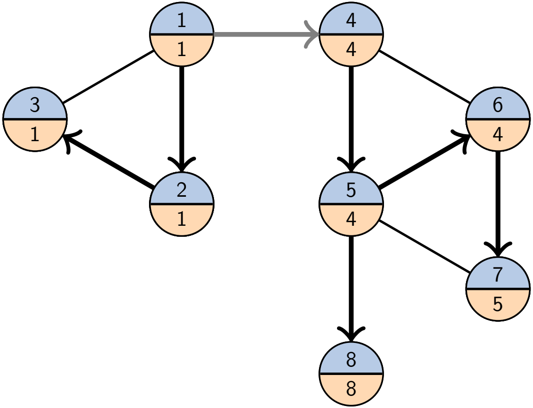
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

Definition: `dfs_num` and `dfs_min`

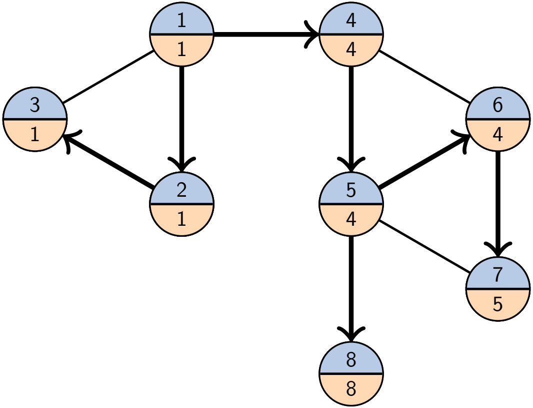
- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



Articulation Points & Bridges

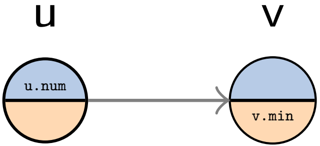
Definition: `dfs_num` and `dfs_min`

- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge



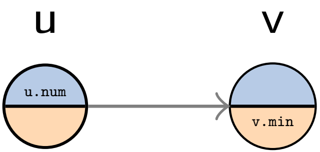
Articulation Points & Bridges

Consider backtracking along a Tree Edge $u \longrightarrow v$:



Articulation Points & Bridges

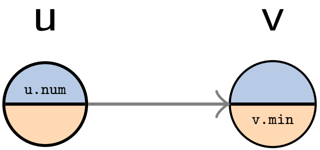
Consider backtracking along a Tree Edge $u \longrightarrow v$:



- `u.num` < `v.min`:
- `u.num` = `v.min`:
- `u.num` > `v.min`:

Articulation Points & Bridges

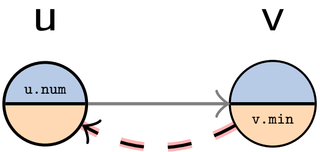
Consider backtracking along a Tree Edge $u \longrightarrow v$:



- `u.num < v.min`:
- `u.num = v.min`:
- `u.num > v.min`:

Articulation Points & Bridges

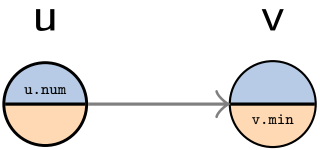
Consider backtracking along a Tree Edge $u \longrightarrow v$:



- `u.num < v.min`:
- `u.num = v.min`:
- `u.num > v.min`:

Articulation Points & Bridges

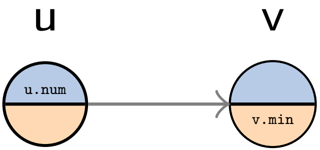
Consider backtracking along a Tree Edge $u \longrightarrow v$:



- `u.num` $<$ `v.min`:
- `u.num` $=$ `v.min`:
- `u.num` $>$ `v.min`:

Articulation Points & Bridges

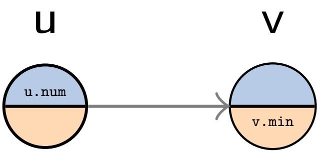
Consider backtracking along a Tree Edge $u \longrightarrow v$:



- `u.num` $<$ `v.min`: Removing `u` or `u` $\rightarrow$ `v` disconnects `v`
- `u.num` $=$ `v.min`:
- `u.num` $>$ `v.min`:

Articulation Points & Bridges

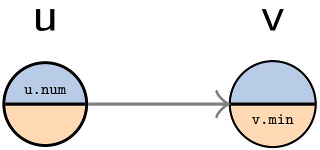
Consider backtracking along a Tree Edge $u \longrightarrow v$:



- `u.num` $<$ `v.min`: `u` is Articulation Point, `u` $\rightarrow$ `v` is Bridge
- `u.num` $=$ `v.min`:
- `u.num` $>$ `v.min`:

Articulation Points & Bridges

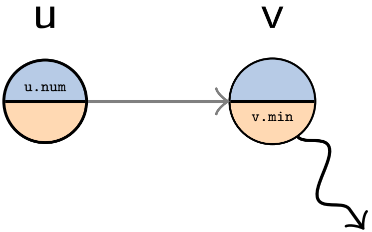
Consider backtracking along a Tree Edge $u \longrightarrow v$:



- `u.num` $<$ `v.min`: `u` is Articulation Point, `u` $\rightarrow$ `v` is Bridge
- `u.num` $=$ `v.min`:
- `u.num` $>$ `v.min`:

Articulation Points & Bridges

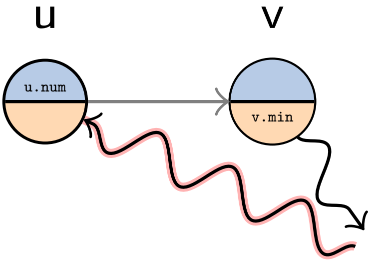
Consider backtracking along a Tree Edge $u \longrightarrow v$:



- `u.num` < `v.min`: `u` is Articulation Point, `u` → `v` is Bridge
- `u.num` = `v.min`:
- `u.num` > `v.min`:

Articulation Points & Bridges

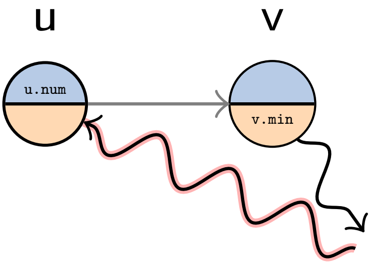
Consider backtracking along a Tree Edge $u \longrightarrow v$:



- `u.num` $<$ `v.min`: `u` is Articulation Point, `u` $\rightarrow$ `v` is Bridge
- `u.num` $=$ `v.min`:
- `u.num` $>$ `v.min`:

Articulation Points & Bridges

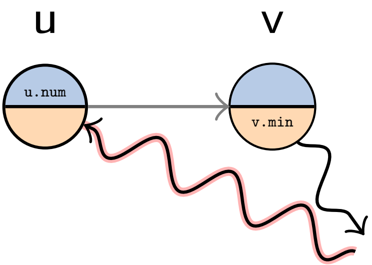
Consider backtracking along a Tree Edge $u \longrightarrow v$:



- `u.num` $<$ `v.min`: `u` is Articulation Point, `u` $\rightarrow$ `v` is Bridge
- `u.num` $=$ `v.min`: Removing `u` disconnects `v`
- `u.num` $>$ `v.min`:

Articulation Points & Bridges

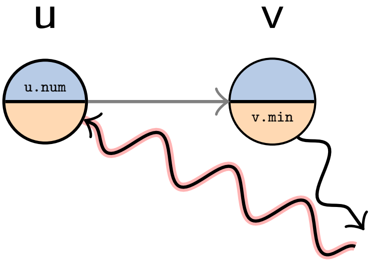
Consider backtracking along a Tree Edge $u \longrightarrow v$:



- `u.num` $<$ `v.min`: `u` is Articulation Point, `u` $\to$ `v` is Bridge
- `u.num` $=$ `v.min`: Removing `u` $\to$ `v` does not disconnect `v`
- `u.num` $>$ `v.min`:

Articulation Points & Bridges

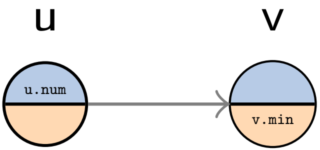
Consider backtracking along a Tree Edge $u \longrightarrow v$:



- `u.num` $<$ `v.min`: `u` is Articulation Point, `u` $\rightarrow$ `v` is Bridge
- `u.num` $=$ `v.min`: `u` is Articulation Point
- `u.num` $>$ `v.min`:

Articulation Points & Bridges

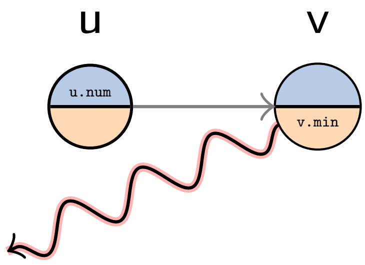
Consider backtracking along a Tree Edge $u \longrightarrow v$:



- u.num < v.min: u is Articulation Point, u → v is Bridge
- u.num = v.min: u is Articulation Point
- u.num > v.min:

Articulation Points & Bridges

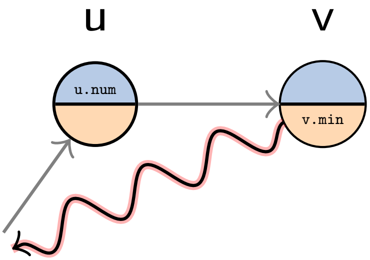
Consider backtracking along a Tree Edge $u \longrightarrow v$:



- `u.num` < `v.min`: `u` is Articulation Point, `u` $\rightarrow$ `v` is Bridge
- `u.num` = `v.min`: `u` is Articulation Point
- `u.num` > `v.min`:

Articulation Points & Bridges

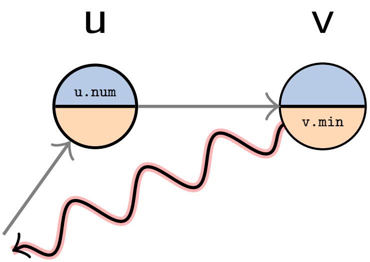
Consider backtracking along a Tree Edge $u \longrightarrow v$:



- `u.num` < `v.min`: `u` is Articulation Point, `u` $\rightarrow$ `v` is Bridge
- `u.num` = `v.min`: `u` is Articulation Point
- `u.num` > `v.min`:

Articulation Points & Bridges

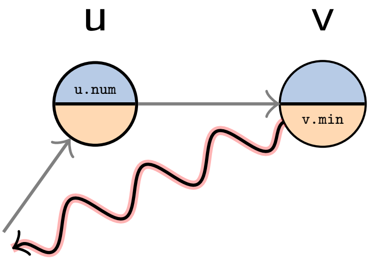
Consider backtracking along a Tree Edge $u \longrightarrow v$:



- `u.num` $<$ `v.min`: `u` is Articulation Point, `u` $\rightarrow$ `v` is Bridge
- `u.num` $=$ `v.min`: `u` is Articulation Point
- `u.num` $>$ `v.min`: `v` will stay connected

Articulation Points & Bridges

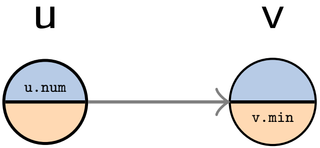
Consider backtracking along a Tree Edge $u \longrightarrow v$:



- `u.num` $<$ `v.min`: `u` is Articulation Point, `u` $\rightarrow$ `v` is Bridge
- `u.num` $=$ `v.min`: `u` is Articulation Point
- `u.num` $>$ `v.min`: –

Articulation Points & Bridges

Consider backtracking along a Tree Edge $u \longrightarrow v$:



- `u.num` $<$ `v.min`: `u` is Articulation Point, `u` $\to$ `v` is Bridge
- `u.num` $=$ `v.min`: `u` is Articulation Point
- `u.num` $>$ `v.min`: –

Articulation Points & Bridges

Finding Articulation Points and Bridges

- Start DFS, record `dfs_num` and `dfs_min`
- When backtracking from a Tree Edge $\texttt{u} \rightarrow \texttt{v}$, test if
  - $\texttt{dfs\_num[u]} \leq \texttt{dfs\_min[v]}$ for Articulation Points
  - $\texttt{dfs\_num[u]} < \texttt{dfs\_min[v]}$ for Bridges

Articulation Points & Bridges

Finding Articulation Points and Bridges

- Start DFS, record `dfs_num` and `dfs_min`
- When backtracking from a Tree Edge $\mathtt{u} \rightarrow \mathtt{v}$, test if
  - $\mathtt{dfs\_num[u]} \leq \mathtt{dfs\_min[v]}$ for Articulation Points
  - $\mathtt{dfs\_num[u]} < \mathtt{dfs\_min[v]}$ for Bridges

Attention

The root of the DFS needs extra care.

It is an Articulation Point if and only if it has at least two distinct children in the DFS tree.

Articulation Points & Bridges

Example Code

```
int dfs_counter = 0;
const int UNVISITED = -1;
int dfsRoot, rootChildren;

vector<int> dfs_num(V, UNVISITED);
vector<int> dfs_min(V, UNVISITED);
vector<int> dfs_parent(V, -1);

for (int i = 0; i < V; i++)
  if (dfs_num[i] == UNVISITED) {
    dfsRoot = i; rootChildren = 0;
    dfs(i); // code on next slide
    if (rootChildren > 1)
      cout << i << " is AP" << endl;
  }
```

- Use `dfs_parent` to avoid walking undirected edges twice

Articulation Points & Bridges

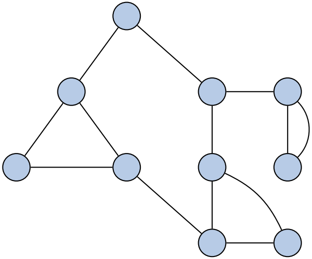
Example Code: Articulation Points & Bridges

```
void dfs(int u) {
  dfs_min[u] = dfs_num[u] = dfs_counter++;
  for (auto v: adj[u]) {
    if (dfs_num[v] == UNVISITED) { // Tree Edge
      dfs_parent[v] = u;
      if (u == dfsRoot) rootChildren++;

      dfs(v);

      if (dfs_num[u] <= dfs_min[v] && u != dfsRoot)
        cout << u << " is AP" << endl;
      if (dfs_num[u] < dfs_min[v])
        cout << u << "-" << v << " is Bridge" << endl;
      dfs_min[u] = min(dfs_min[u], dfs_min[v]);
    } else if (v != dfs_parent[u]) // Back Edge
      dfs_min[u] = min(dfs_min[u], dfs_num[v]);
  }
}
```

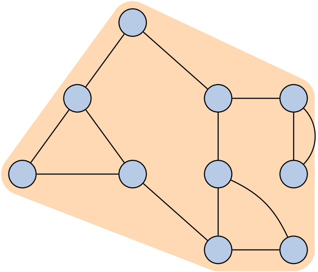
Strongly Connected Components



In undirected graphs, we call a (maximal) subset of V

- *connected component* if every of its vertices is reachable from every other vertex

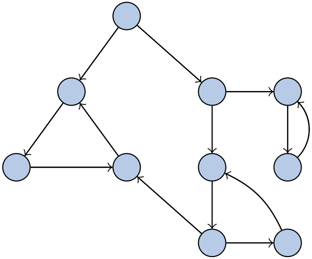
Strongly Connected Components



In undirected graphs, we call a (maximal) subset of V

- *connected component* if every of its vertices is reachable from every other vertex

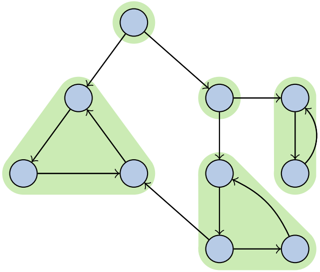
Strongly Connected Components



In directed graphs, we call a (maximal) subset of V

- *strongly connected component* if every of its vertices is reachable from every other vertex

Strongly Connected Components



In directed graphs, we call a (maximal) subset of V

- *strongly connected component* if every of its vertices is reachable from every other vertex

Strongly Connected Components

Tarjan's Algorithm

Algorithm Idea

Maintain a `stack` containing the vertices yet to be assigned to their SCCs.

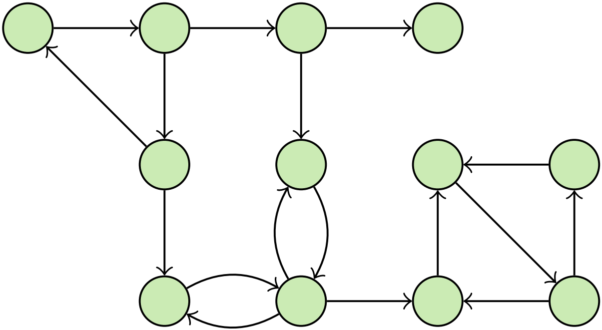
For each vertex v of a DFS tree, record

- `dfs_num[v]` the timestamp at which v was explored
- `dfs_min[v]` the minimum `dfs_num` reachable from v using the directed Tree Edges and at most one Back Edge **to a vertex on the** `stack`

When backtracking past a vertex where `dfs_num` $==$ `dfs_min` use the `stack` to compute the SCC starting at `dfs_num`.

Strongly Connected Components

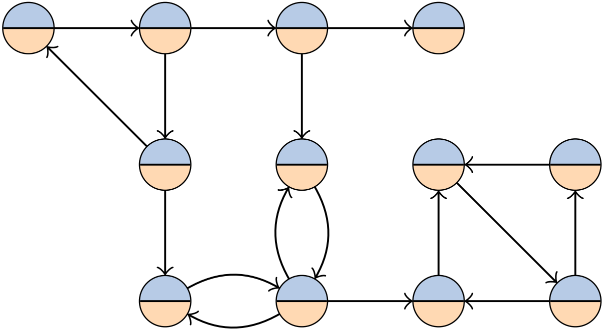
Tarjan's Algorithm



`stack:`

Strongly Connected Components

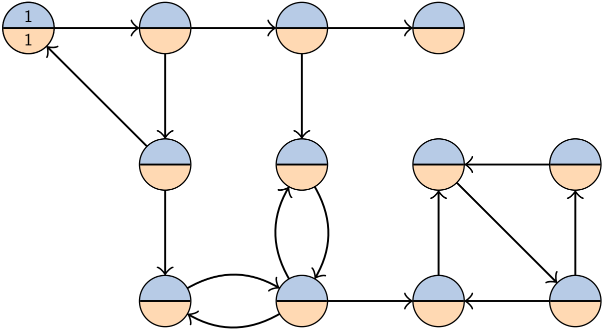
Tarjan's Algorithm



`stack:`

Strongly Connected Components

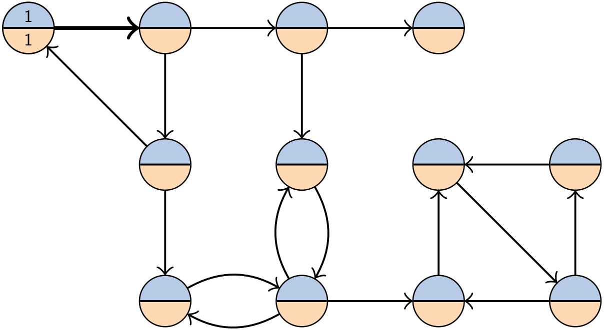
Tarjan's Algorithm



stack: 1

Strongly Connected Components

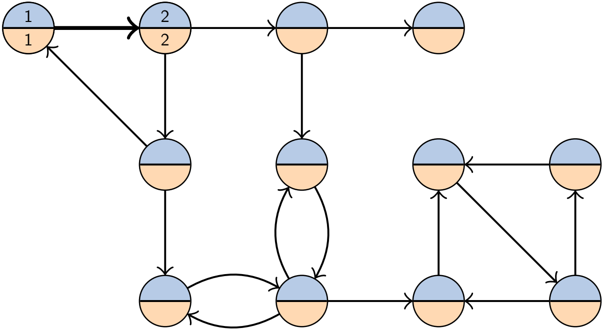
Tarjan's Algorithm



stack: 1

Strongly Connected Components

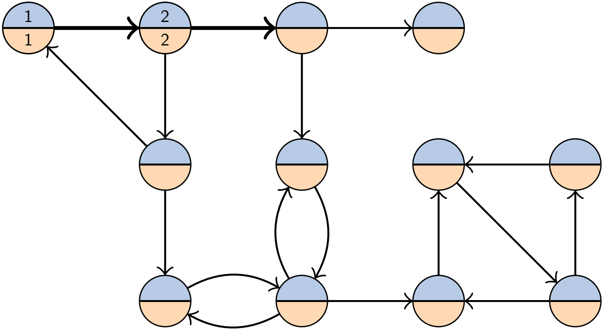
Tarjan's Algorithm



stack: 1 2

Strongly Connected Components

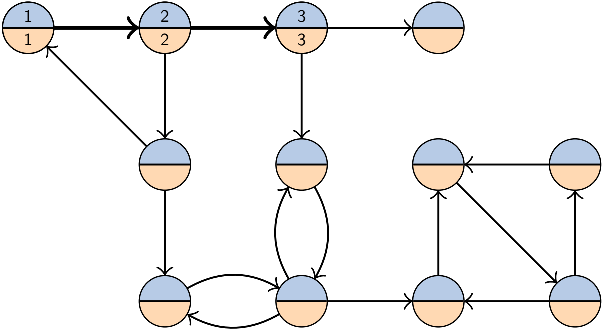
Tarjan's Algorithm



stack: 1 2

Strongly Connected Components

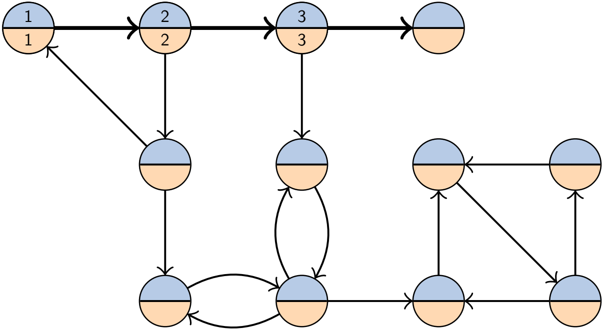
Tarjan's Algorithm



stack: 1 2 3

Strongly Connected Components

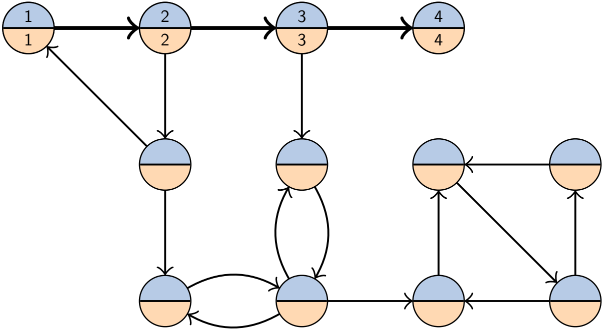
Tarjan's Algorithm



stack: 1 2 3

Strongly Connected Components

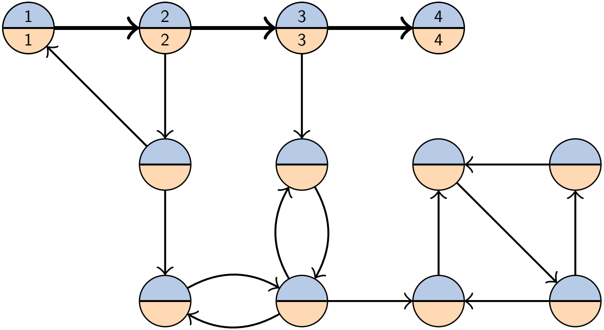
Tarjan's Algorithm



stack: 1 2 3 4

Strongly Connected Components

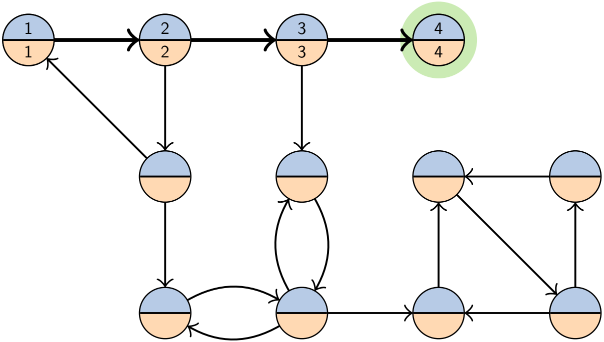
Tarjan's Algorithm



stack: 1 2 3 4

Strongly Connected Components

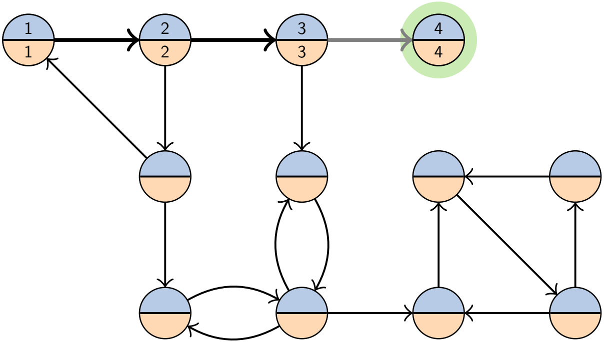
Tarjan's Algorithm



stack: 1 2 3

Strongly Connected Components

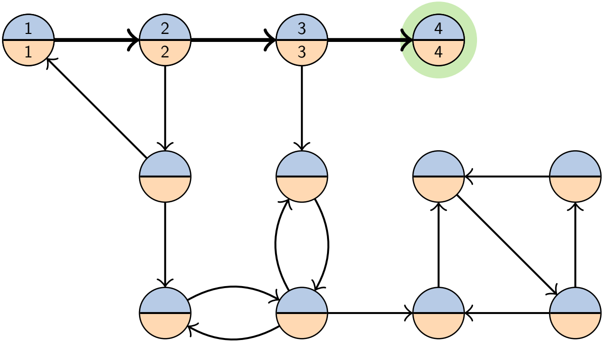
Tarjan's Algorithm



stack: 1 2 3

Strongly Connected Components

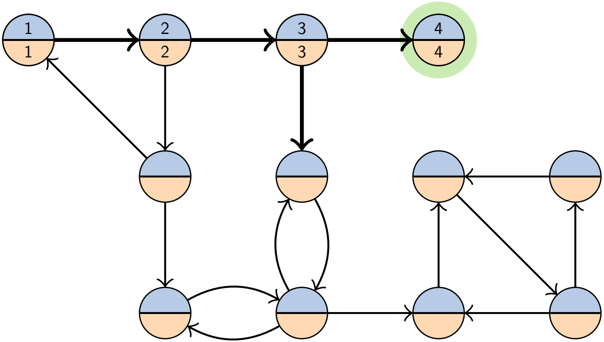
Tarjan's Algorithm



stack: 1 2 3

Strongly Connected Components

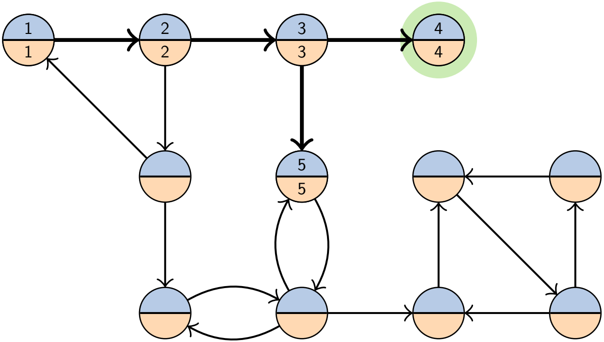
Tarjan's Algorithm



stack: 1 2 3

Strongly Connected Components

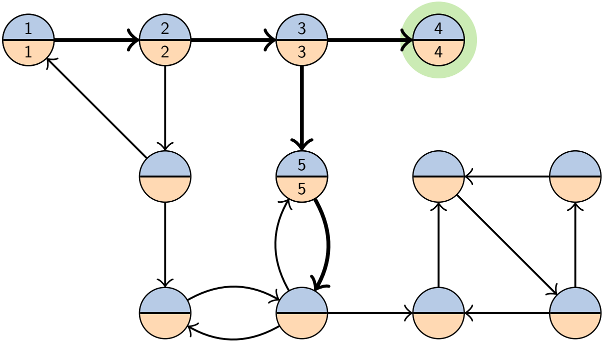
Tarjan's Algorithm



stack: 1 2 3 5

Strongly Connected Components

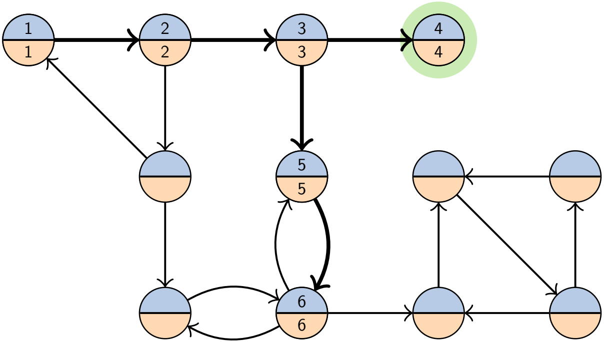
Tarjan's Algorithm



`stack: 1 2 3 5`

Strongly Connected Components

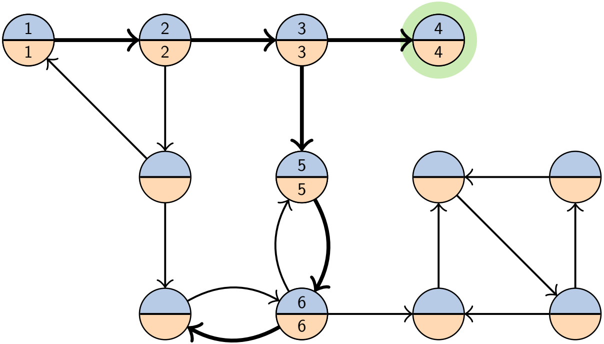
Tarjan's Algorithm



`stack: 1 2 3 5 6`

Strongly Connected Components

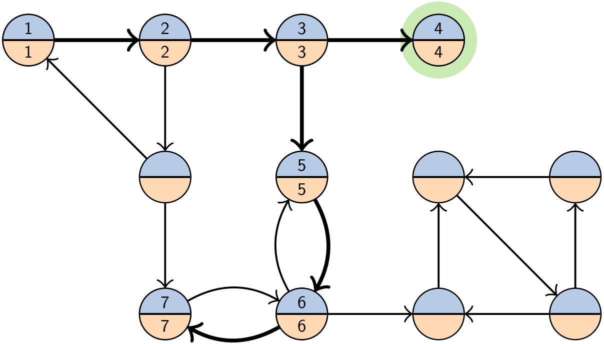
Tarjan's Algorithm



stack: 1 2 3 5 6

Strongly Connected Components

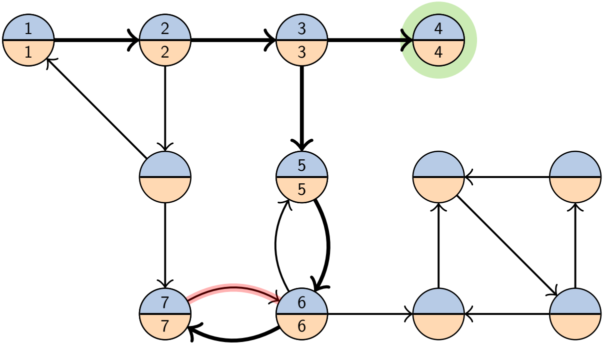
Tarjan's Algorithm



stack: 1 2 3 5 6 7

Strongly Connected Components

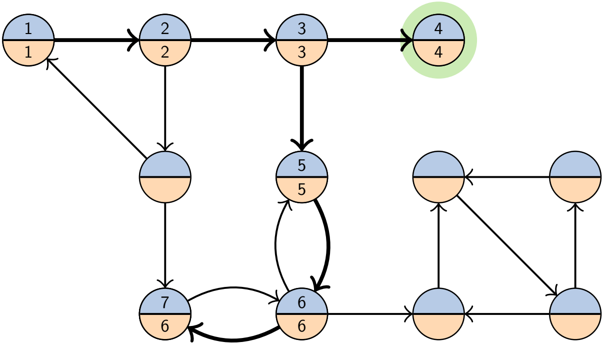
Tarjan's Algorithm



stack: 1 2 3 5 6 7

Strongly Connected Components

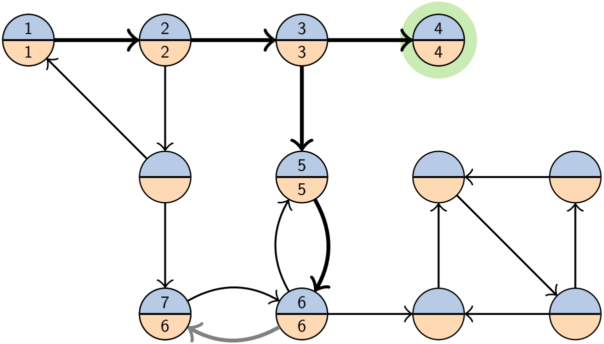
Tarjan's Algorithm



stack: 1 2 3 5 6 7

Strongly Connected Components

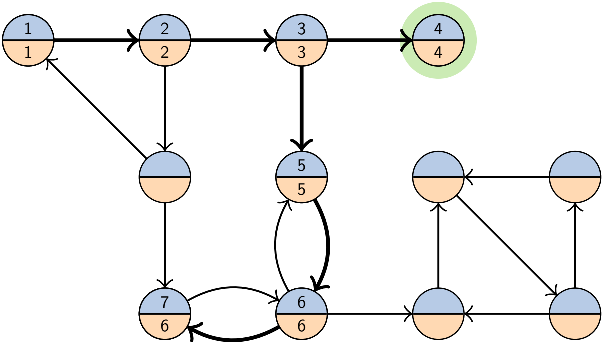
Tarjan's Algorithm



stack: 1 2 3 5 6 7

Strongly Connected Components

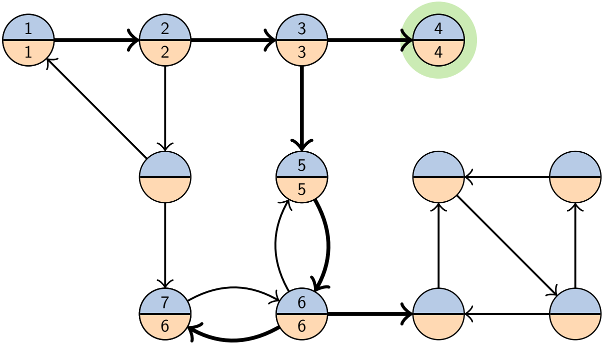
Tarjan's Algorithm



stack: 1 2 3 5 6 7

Strongly Connected Components

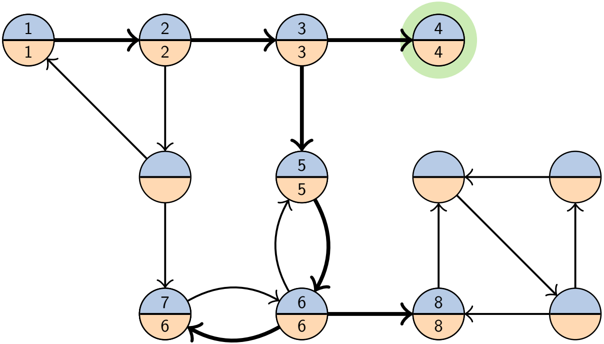
Tarjan's Algorithm



stack: 1 2 3 5 6 7

Strongly Connected Components

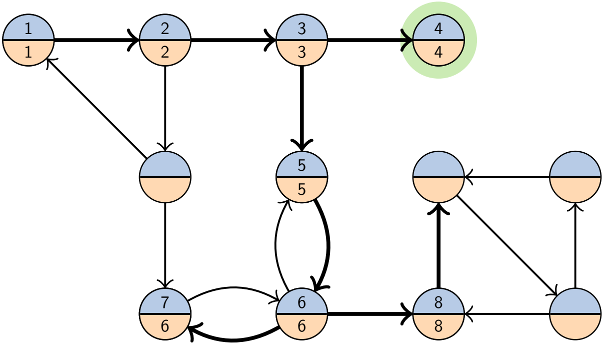
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8

Strongly Connected Components

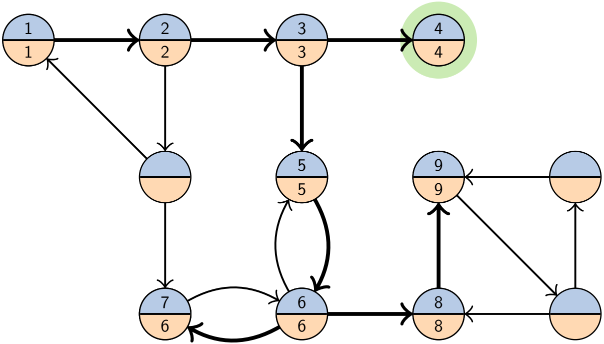
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8

Strongly Connected Components

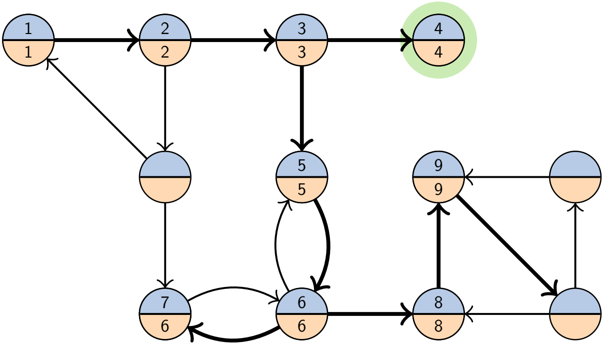
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9

Strongly Connected Components

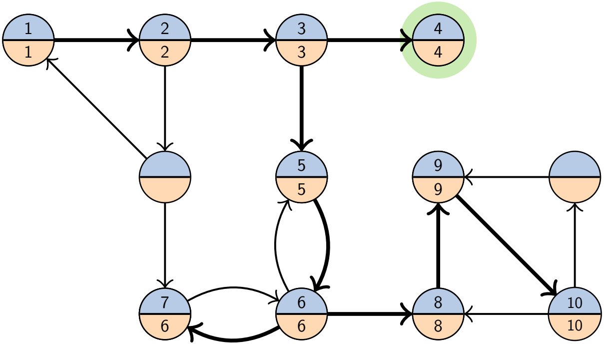
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9

Strongly Connected Components

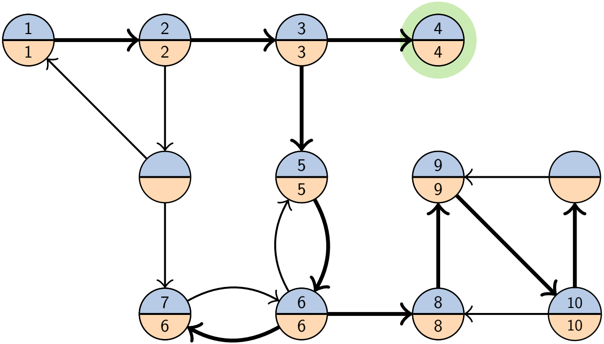
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9 10

Strongly Connected Components

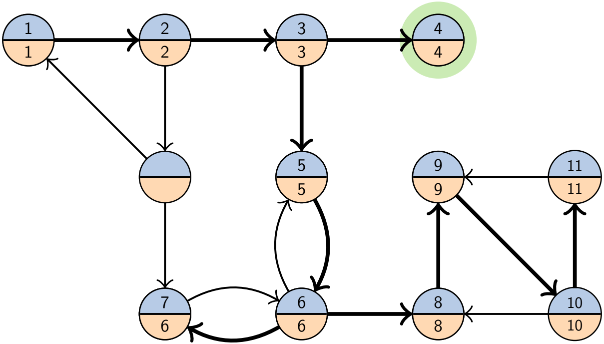
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9 10

Strongly Connected Components

Tarjan's Algorithm

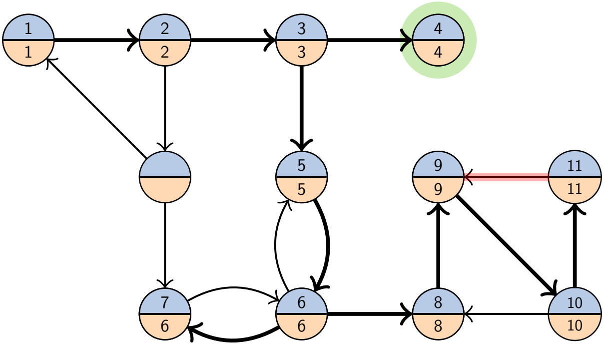


stack: 1 2 3 5 6 7 8 9 10 11

max planck institut informatik

Strongly Connected Components

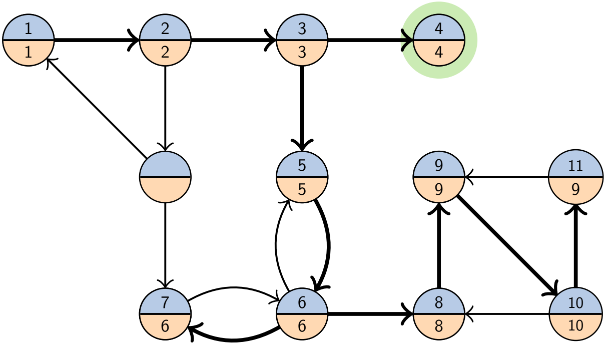
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9 10 11

Strongly Connected Components

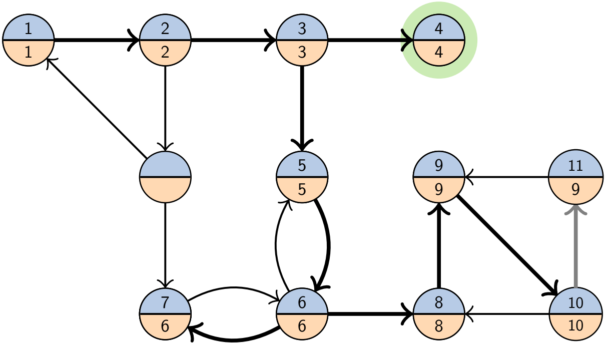
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9 10 11

Strongly Connected Components

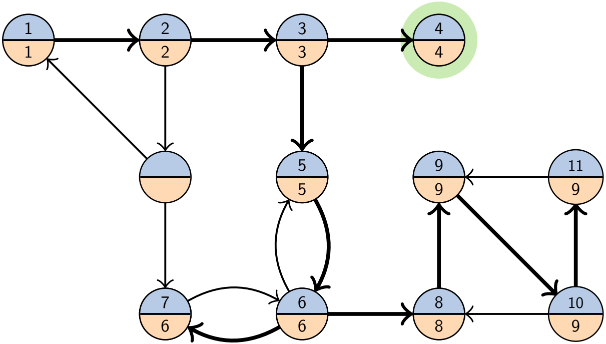
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9 10 11

Strongly Connected Components

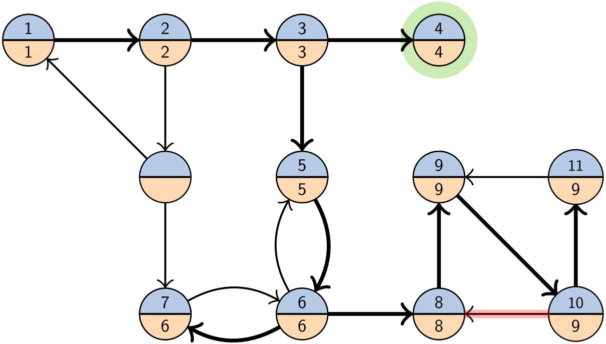
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9 10 11

Strongly Connected Components

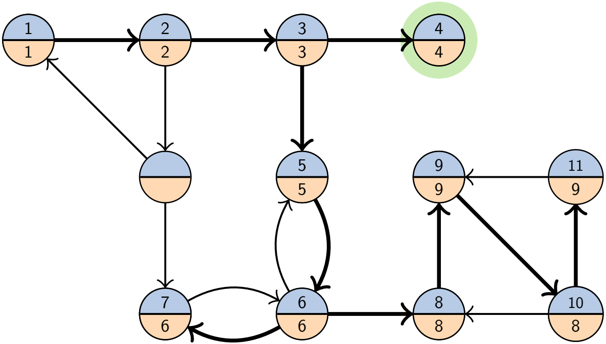
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9 10 11

Strongly Connected Components

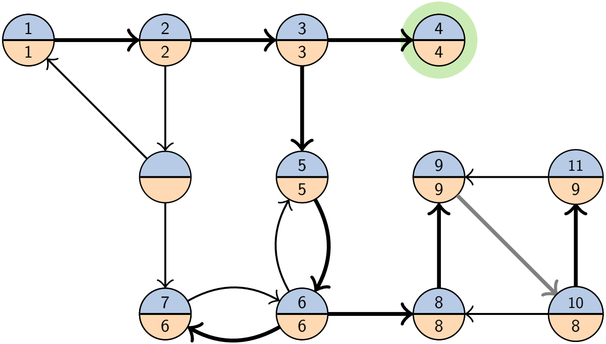
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9 10 11

Strongly Connected Components

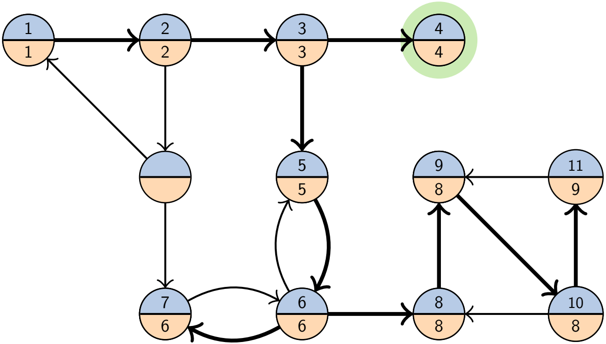
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9 10 11

Strongly Connected Components

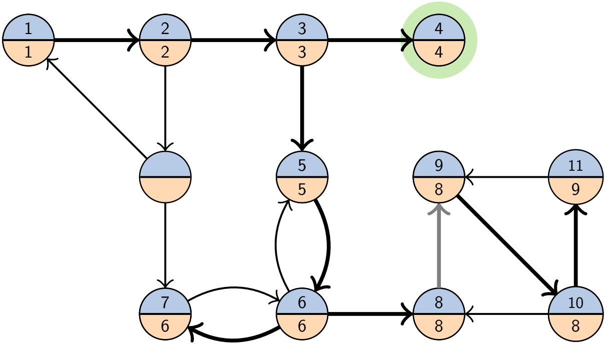
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9 10 11

Strongly Connected Components

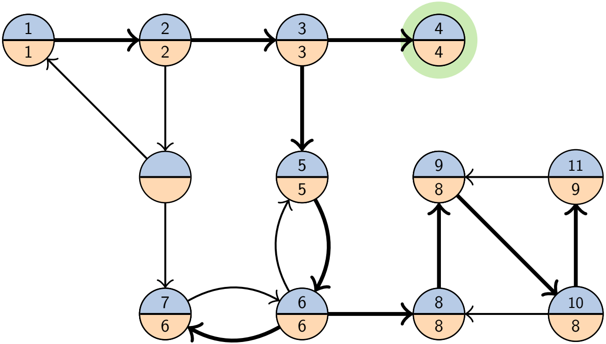
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9 10 11

Strongly Connected Components

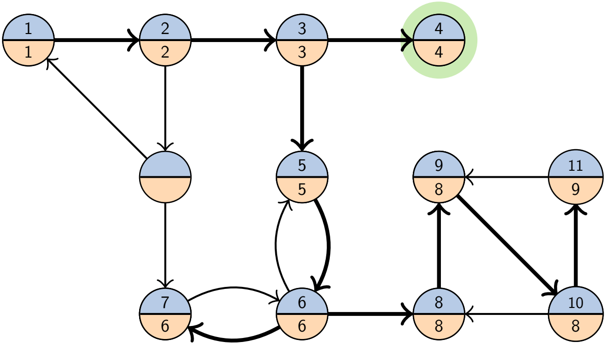
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9 10 11

Strongly Connected Components

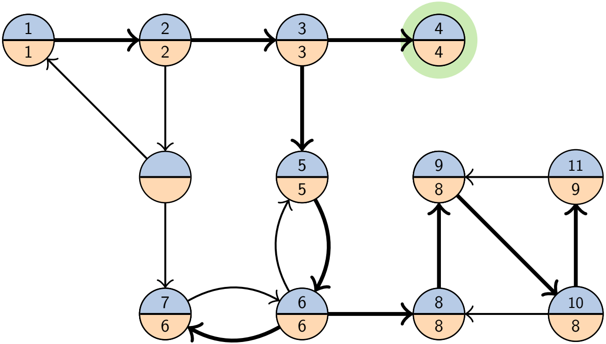
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9 10 11

Strongly Connected Components

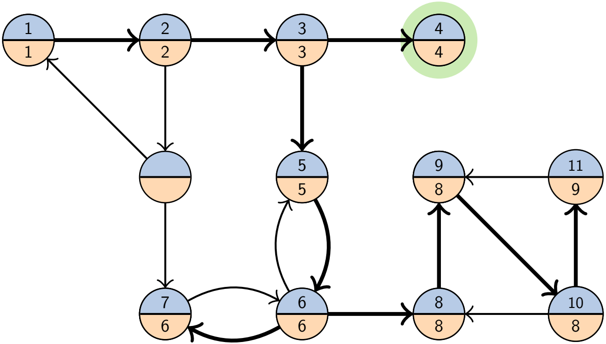
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9 10 11

Strongly Connected Components

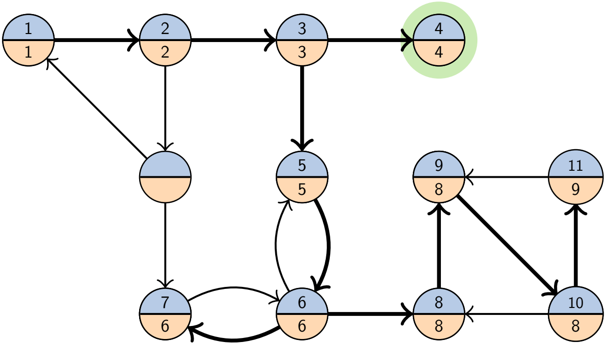
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9 10 11

Strongly Connected Components

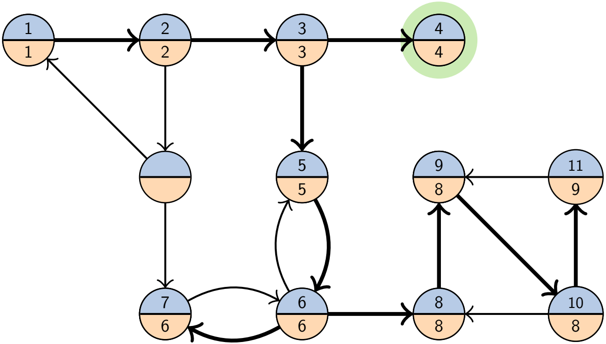
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9 10 11

Strongly Connected Components

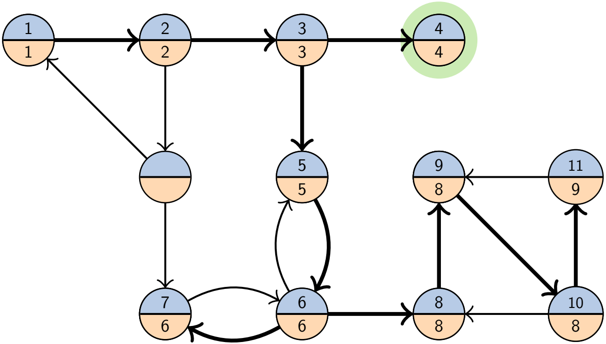
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9 10

Strongly Connected Components

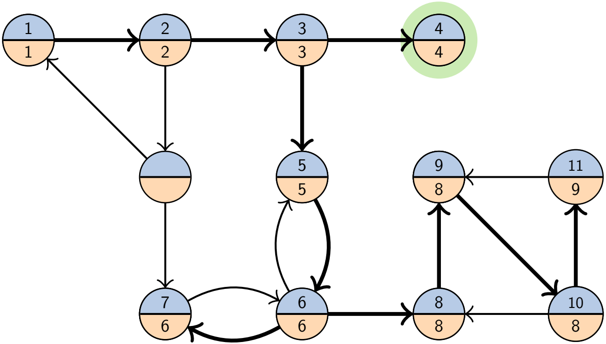
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8 9

Strongly Connected Components

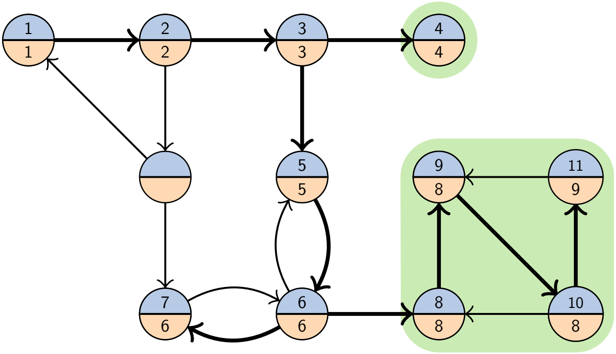
Tarjan's Algorithm



stack: 1 2 3 5 6 7 8

Strongly Connected Components

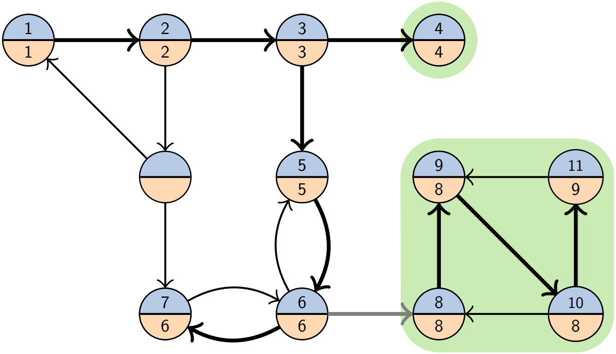
Tarjan's Algorithm



stack: 1 2 3 5 6 7

Strongly Connected Components

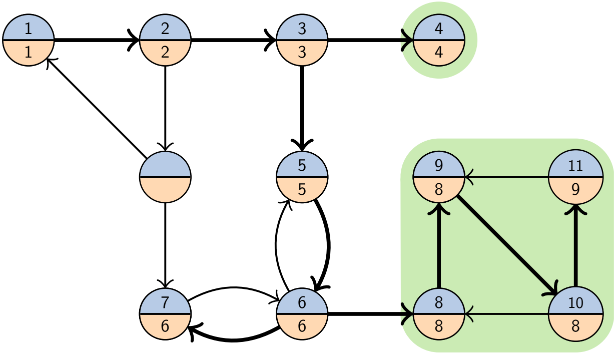
Tarjan's Algorithm



stack: 1 2 3 5 6 7

Strongly Connected Components

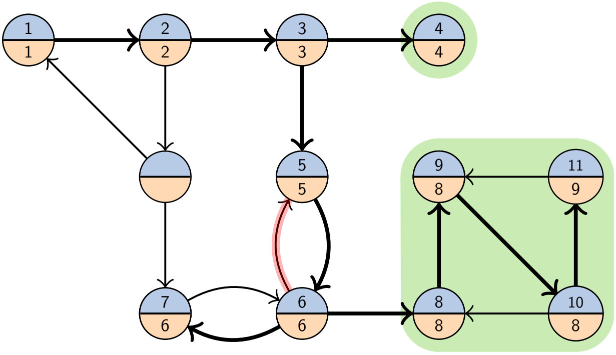
Tarjan's Algorithm



stack: 1 2 3 5 6 7

Strongly Connected Components

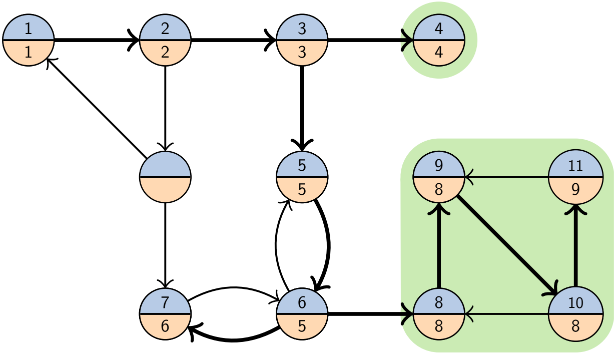
Tarjan's Algorithm



stack: 1 2 3 5 6 7

Strongly Connected Components

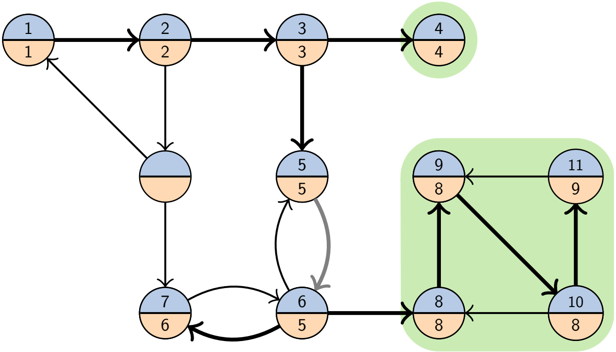
Tarjan's Algorithm



stack: 1 2 3 5 6 7

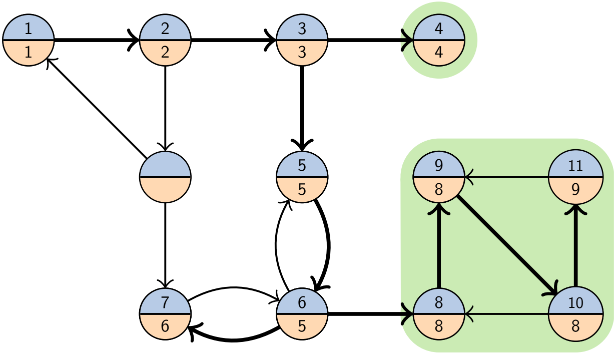
Strongly Connected Components

Tarjan's Algorithm



stack: 1 2 3 5 6 7

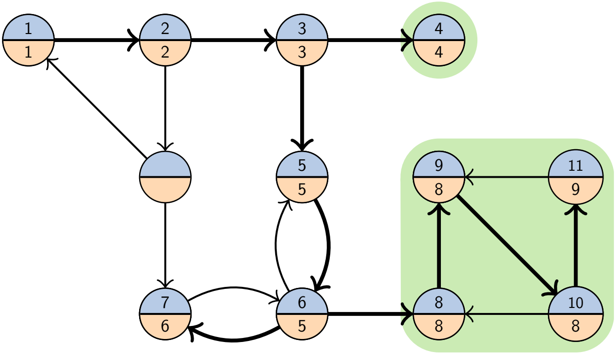
Strongly Connected Components
Tarjan's Algorithm



stack: 1 2 3 5 6 7

Strongly Connected Components

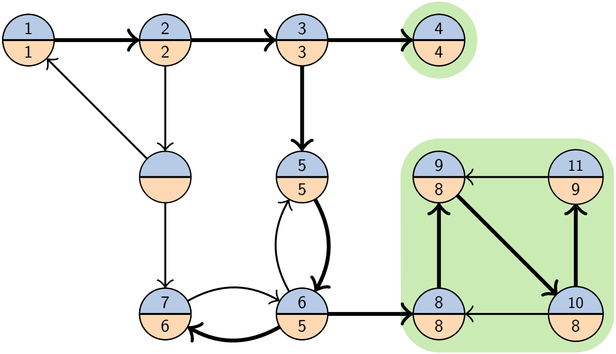
Tarjan's Algorithm



stack: 1 2 3 5 6 7

Strongly Connected Components

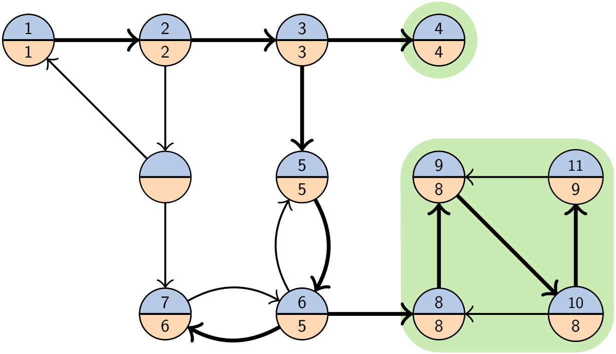
Tarjan's Algorithm



stack: 1 2 3 5 6 7

Strongly Connected Components

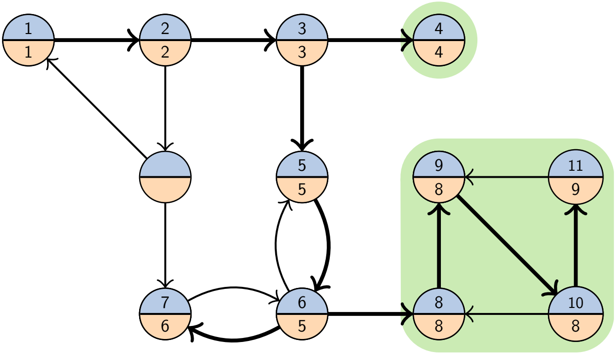
Tarjan's Algorithm



stack: 1 2 3 5 6 7

Strongly Connected Components

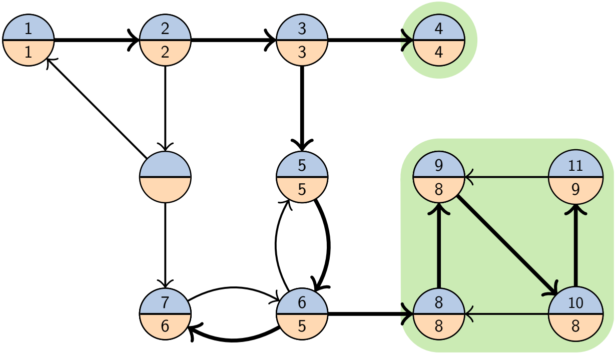
Tarjan's Algorithm



stack: 1 2 3 5 6

Strongly Connected Components

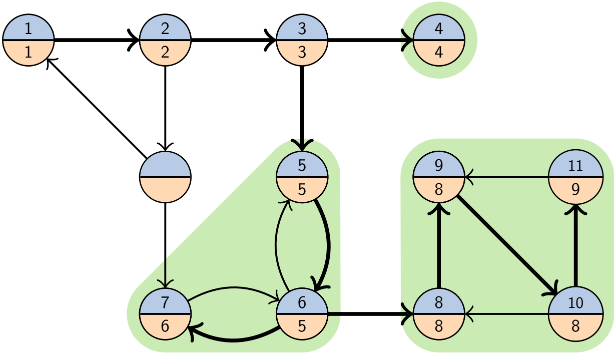
Tarjan's Algorithm



stack: 1 2 3 5

Strongly Connected Components

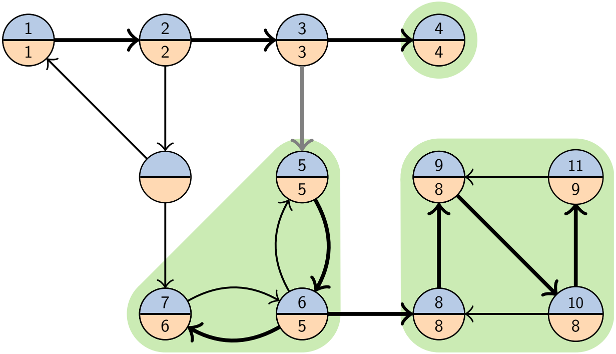
Tarjan's Algorithm



stack: 1 2 3

Strongly Connected Components

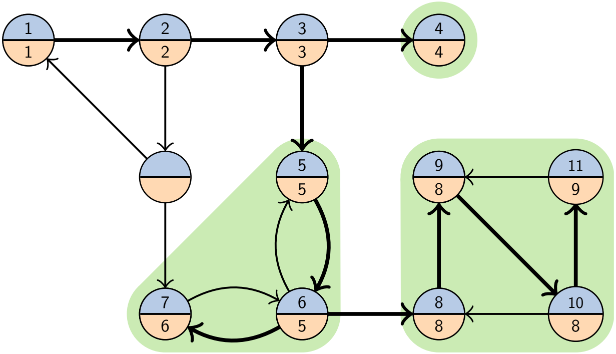
Tarjan's Algorithm



stack: 1 2 3

Strongly Connected Components

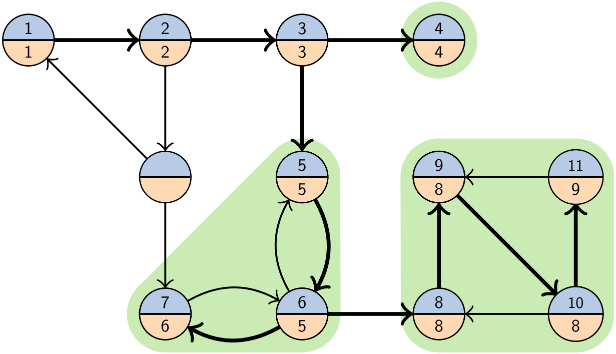
Tarjan's Algorithm



stack: 1 2 3

Strongly Connected Components

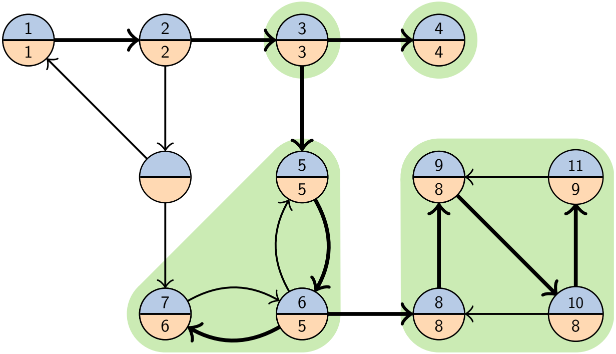
Tarjan's Algorithm



stack: 1 2 3

Strongly Connected Components

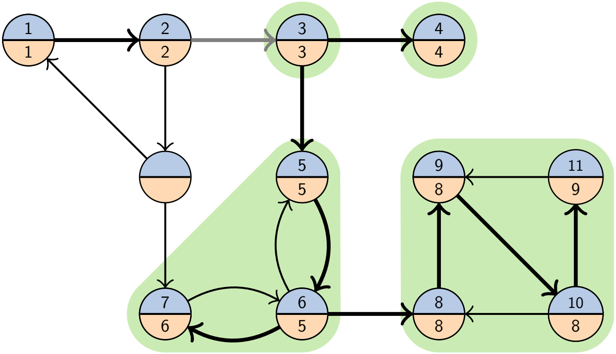
Tarjan's Algorithm



stack: 1 2

Strongly Connected Components

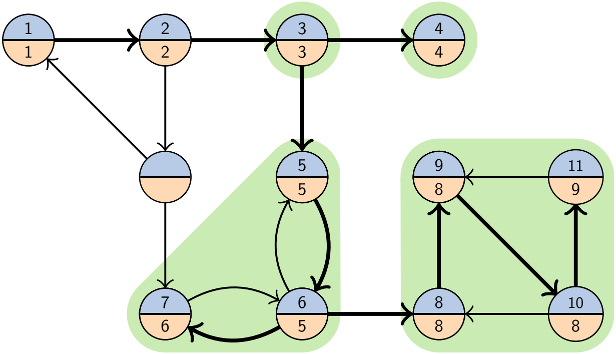
Tarjan's Algorithm



stack: 1 2

Strongly Connected Components

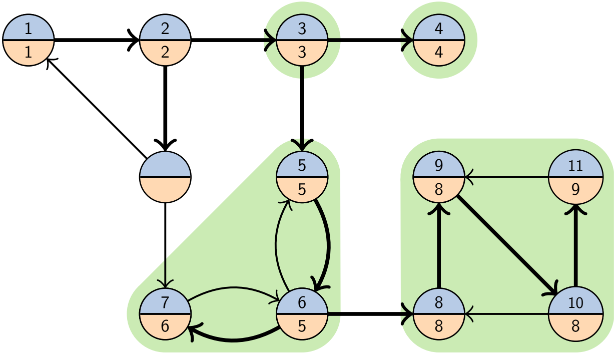
Tarjan's Algorithm



stack: 1 2

Strongly Connected Components

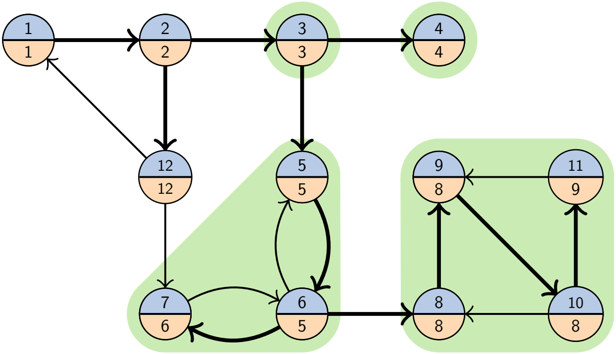
Tarjan's Algorithm



stack: 1 2

Strongly Connected Components

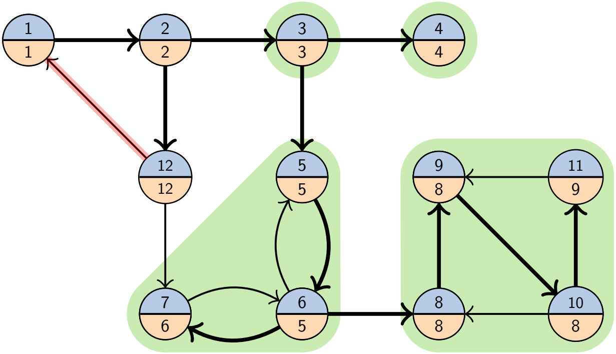
Tarjan's Algorithm



stack: 1 2 12

Strongly Connected Components

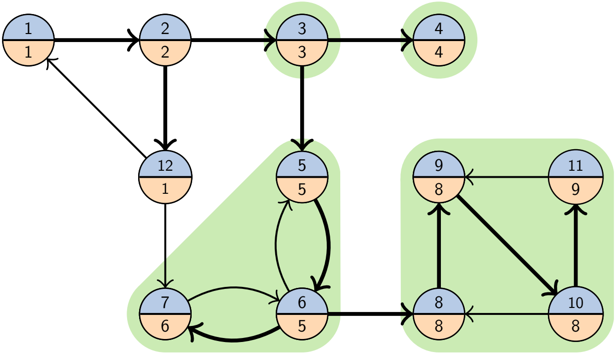
Tarjan's Algorithm



stack: 1 2 12

Strongly Connected Components

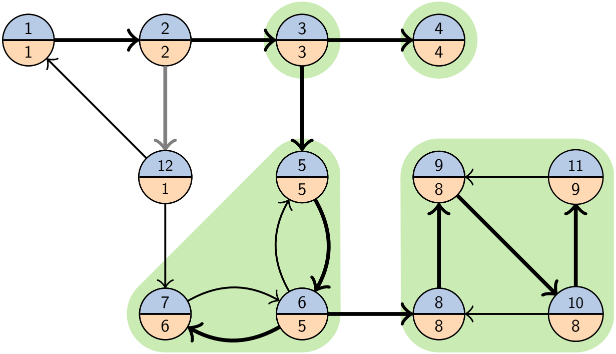
Tarjan's Algorithm



stack: 1 2 12

Strongly Connected Components

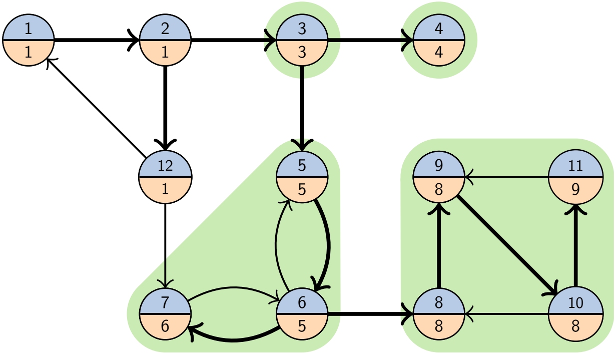
Tarjan's Algorithm



stack: 1 2 12

Strongly Connected Components

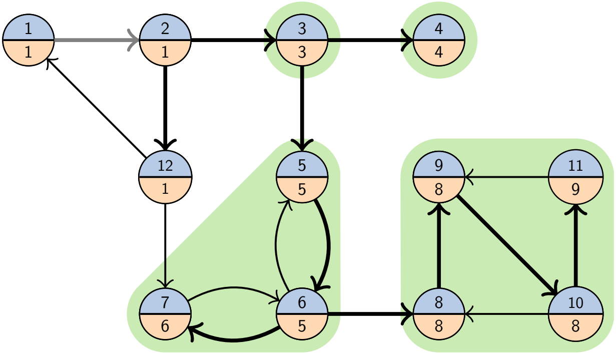
Tarjan's Algorithm



stack: 1 2 12

Strongly Connected Components

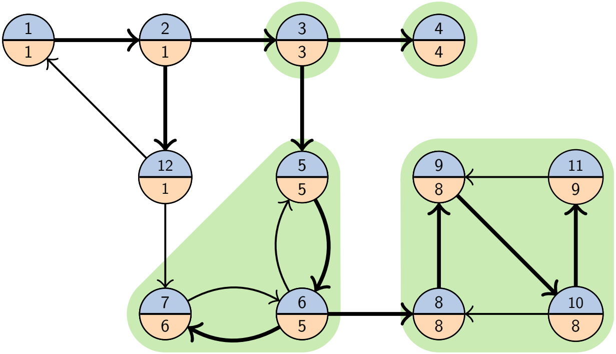
Tarjan's Algorithm



stack: 1 2 12

Strongly Connected Components

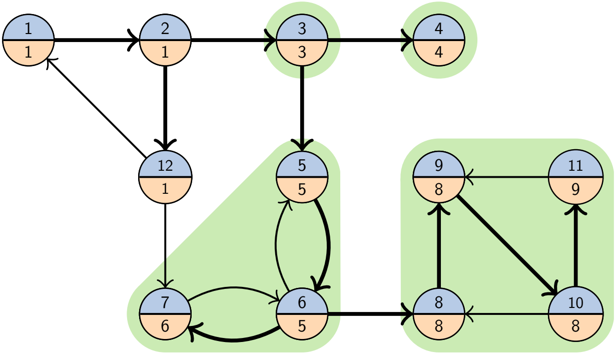
Tarjan's Algorithm



stack: 1 2 12

Strongly Connected Components

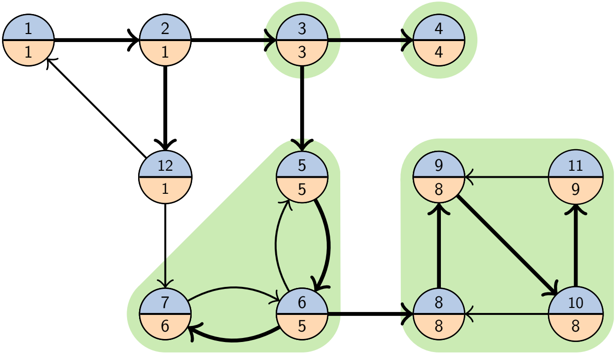
Tarjan's Algorithm



stack: 1 2 12

Strongly Connected Components

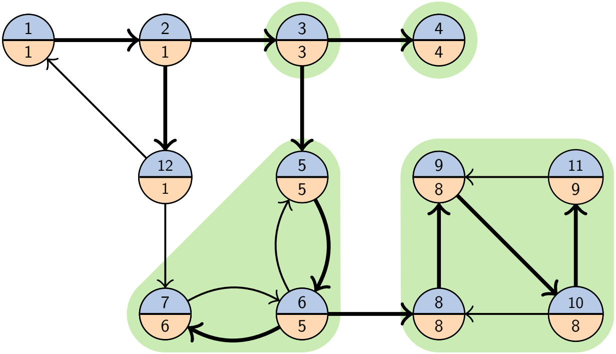
Tarjan's Algorithm



stack: 1 2 12

Strongly Connected Components

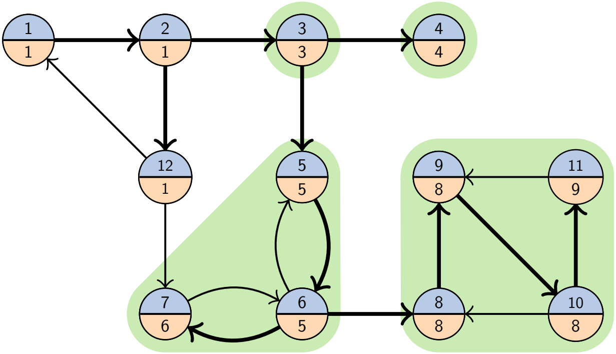
Tarjan's Algorithm



stack: 1 2 12

Strongly Connected Components

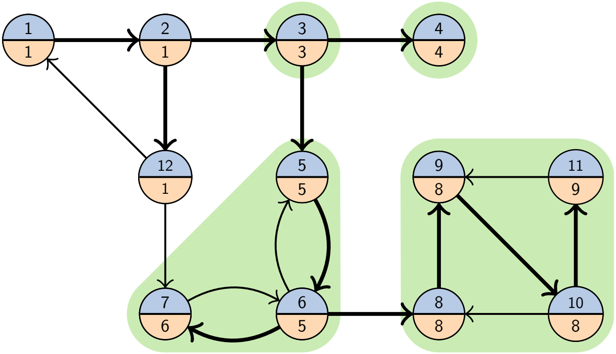
Tarjan's Algorithm



`stack:` 1 2

Strongly Connected Components

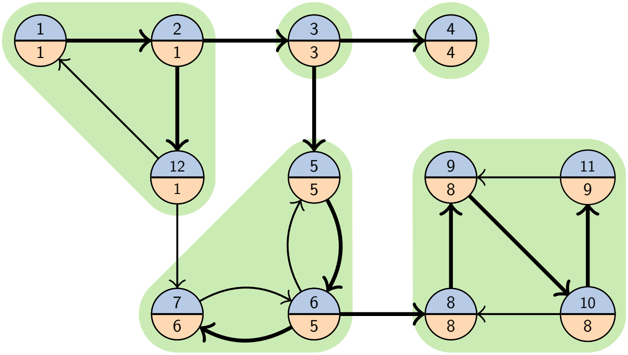
Tarjan's Algorithm



stack: 1

Strongly Connected Components

Tarjan's Algorithm



stack:

Strongly Connected Components

Example Code: Finding SCCs

```
stack<int> S; // stack
int dfs_counter = 0;
const int UNVISITED = -1;

vector<int> dfs_num(V, UNVISITED);
vector<int> dfs_min(V, UNVISITED);
vector<bool> on_stack(V, false);

for (int i = 0; i < V; i++) {
  if (dfs_num[i] == UNVISITED)
     dfs(i); // on next slide
}
```

Strongly Connected Components

Example Code: Finding SCCs

```
void dfs(int u) {
  dfs_min[u] = dfs_num[u] = dfs_counter++;
  S.push(u);
  on_stack[u] = true;
  for (auto v: adj[u]) {
    if (dfs_num[v] == UNVISITED)
      dfs(v);
    if (on_stack[v]) // only on_stack can update dfs_min
      dfs_min[u] = min(dfs_min[u], dfs_min[v]);
  }
  if (dfs_min[u] == dfs_num[u]) { // output result
    cout << "SCC: ";
    int v = -1;
    while (v != u) { // output SCC starting in u
      v = S.top() S.pop(); on_stack[v] = false;
      cout << v << " ";
    }
    cout << endl;
  }
}
```

Fun with DAGs

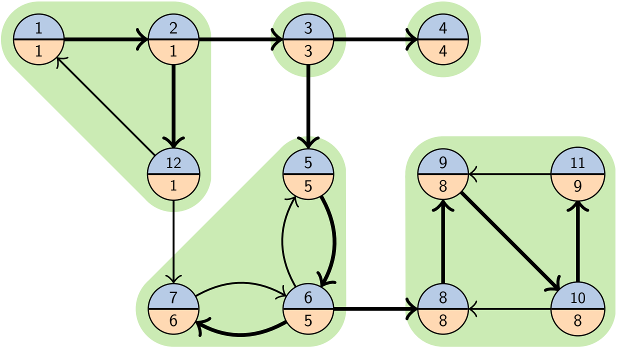
Definition

A *DAG* is a directed, acyclic graph

- No cycles by definition
- Is guaranteed to have at least one valid topological ordering

Fun with DAGs

Tarjan's Algorithm Insights

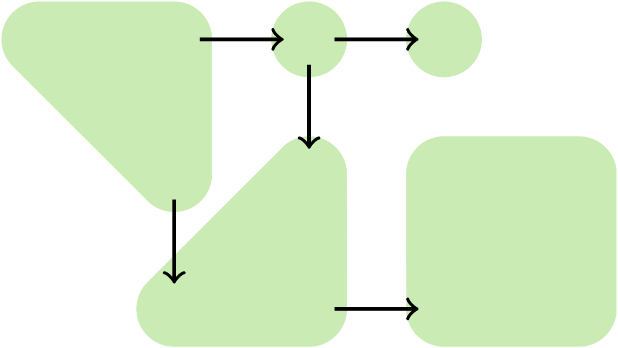


Features of Tarjan's Algorithm

- Compressing the SCCs into Super Vertices yields a DAG

Fun with DAGs

Tarjan's Algorithm Insights

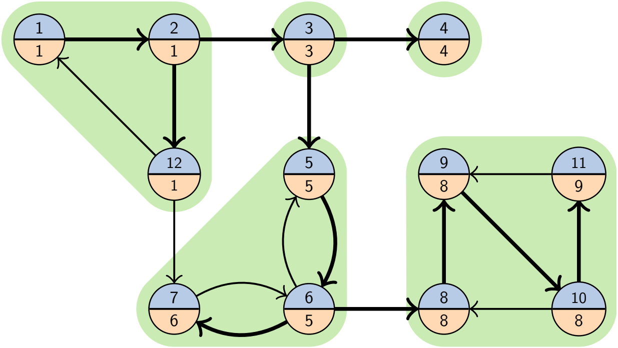


Features of Tarjan's Algorithm

- Compressing the SCCs into Super Vertices yields a DAG

Fun with DAGs

Tarjan's Algorithm Insights

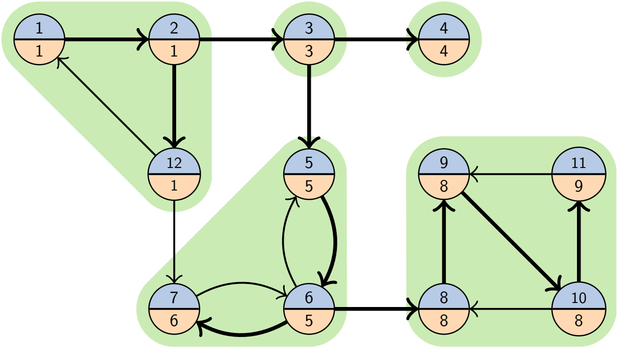


Features of Tarjan's Algorithm

- Compressing the SCCs into Super Vertices yields a DAG

Fun with DAGs

Tarjan's Algorithm Insights

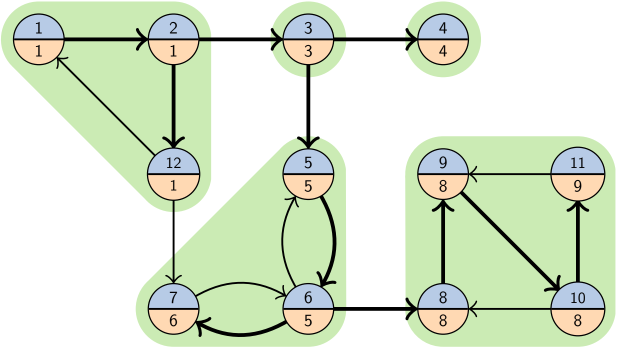


Features of Tarjan's Algorithm

- Compressing the SCCs into Super Vertices yields a DAG
- Tarjan's Algorithm outputs the SCCs in reverse top. order

Fun with DAGs

Tarjan's Algorithm Insights



Features of Tarjan's Algorithm

- Compressing the SCCs into Super Vertices yields a DAG
- Tarjan's Algorithm outputs the SCCs in reverse top. order
  - No need to compute a TS on the resulting DAG

Fun with DAGs

Definition

A *DAG* is a directed, acyclic graph

- No cycles by definition
- Is guaranteed to have at least one valid topological ordering

Fun with DAGs

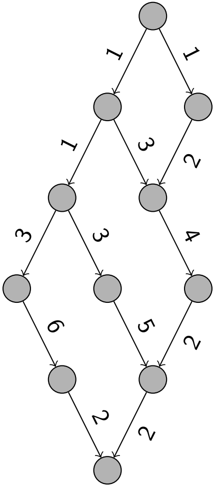
Definition

A *DAG* is a directed, acyclic graph

- No cycles by definition
- Is guaranteed to have at least one valid topological ordering
- Allows solving DP problems bottom-up using a TS

Fun with DAGs

Shortest Paths



Shortest Paths on DAGs

SSSP can be solved in $\mathcal{O}(V + E)$ on DAGs

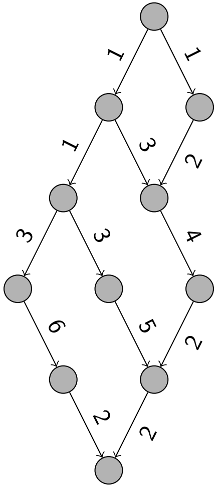
Fun with DAGs

Shortest Paths

Shortest Paths on DAGs

SSSP can be solved in $\mathcal{O}(V + E)$ on DAGs

- Compute topological ordering



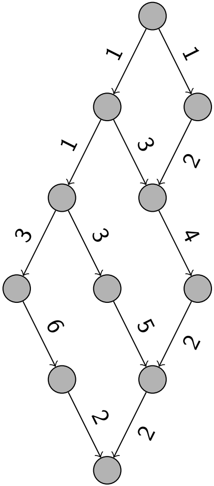
Fun with DAGs

Shortest Paths

Shortest Paths on DAGs

SSSP can be solved in $\mathcal{O}(V+E)$ on DAGs

- Compute topological ordering
- Relax SSSP values in topological order



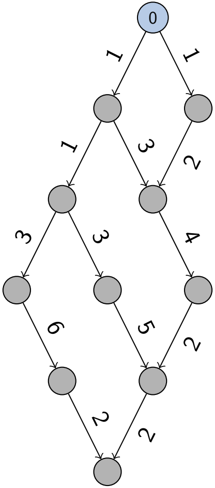
Fun with DAGs

Shortest Paths

Shortest Paths on DAGs

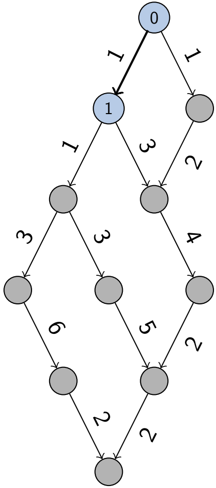
SSSP can be solved in $\mathcal{O}(V + E)$ on DAGs

- Compute topological ordering
- Relax SSSP values in topological order



Fun with DAGs

Shortest Paths



Shortest Paths on DAGs

SSSP can be solved in $\mathcal{O}(V + E)$ on DAGs

- Compute topological ordering
- Relax SSSP values in topological order

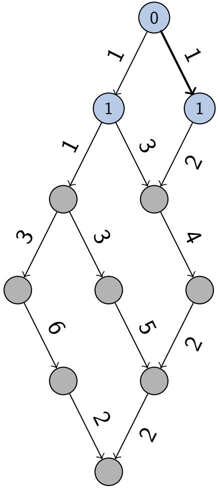
Fun with DAGs

Shortest Paths

Shortest Paths on DAGs

SSSP can be solved in $\mathcal{O}(V + E)$ on DAGs

- Compute topological ordering
- Relax SSSP values in topological order



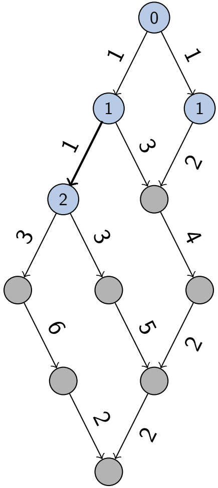
Fun with DAGs

Shortest Paths

Shortest Paths on DAGs

SSSP can be solved in $\mathcal{O}(V + E)$ on DAGs

- Compute topological ordering
- Relax SSSP values in topological order



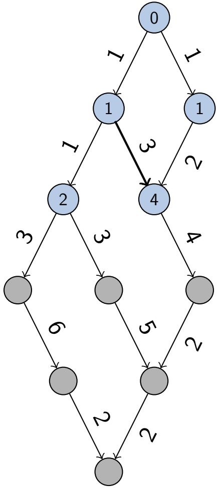
Fun with DAGs

Shortest Paths

Shortest Paths on DAGs

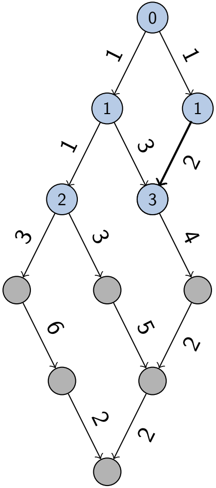
SSSP can be solved in $\mathcal{O}(V + E)$ on DAGs

- Compute topological ordering
- Relax SSSP values in topological order



Fun with DAGs

Shortest Paths



Shortest Paths on DAGs

SSSP can be solved in $\mathcal{O}(V + E)$ on DAGs

- Compute topological ordering
- Relax SSSP values in topological order

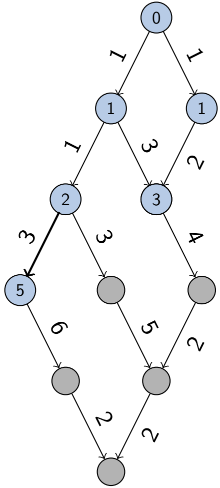
Fun with DAGs

Shortest Paths

Shortest Paths on DAGs

SSSP can be solved in $\mathcal{O}(V + E)$ on DAGs

- Compute topological ordering
- Relax SSSP values in topological order



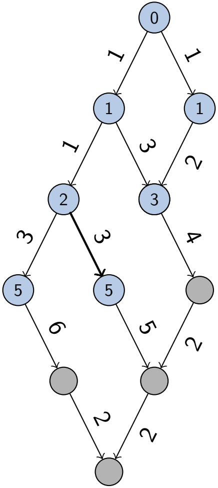
Fun with DAGs

Shortest Paths

Shortest Paths on DAGs

SSSP can be solved in $\mathcal{O}(V + E)$ on DAGs

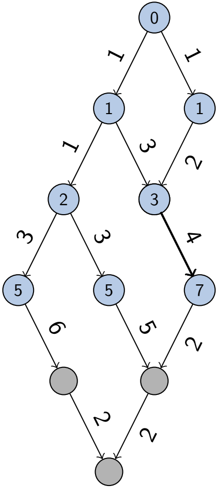
- Compute topological ordering
- Relax SSSP values in topological order



Shortest Paths on DAGs

SSSP can be solved in $\mathcal{O}(V + E)$ on DAGs

- Compute topological ordering
- Relax SSSP values in topological order



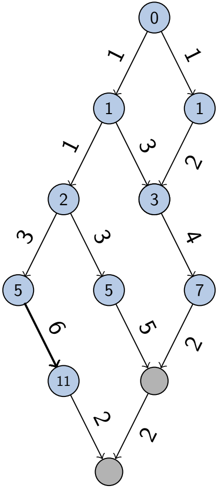
Fun with DAGs

Shortest Paths

Shortest Paths on DAGs

SSSP can be solved in $\mathcal{O}(V + E)$ on DAGs

- Compute topological ordering
- Relax SSSP values in topological order



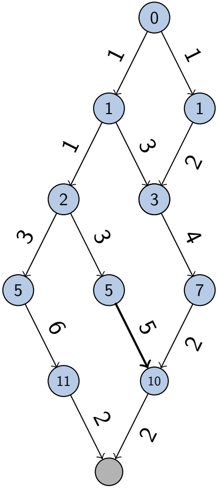
Fun with DAGs

Shortest Paths

Shortest Paths on DAGs

SSSP can be solved in $\mathcal{O}(V+E)$ on DAGs

- Compute topological ordering
- Relax SSSP values in topological order



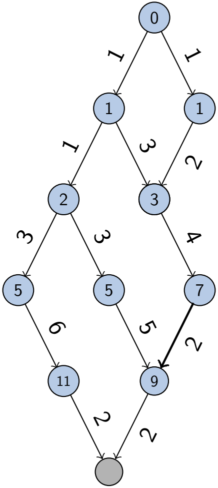
Fun with DAGs

Shortest Paths

Shortest Paths on DAGs

SSSP can be solved in $\mathcal{O}(V + E)$ on DAGs

- Compute topological ordering
- Relax SSSP values in topological order



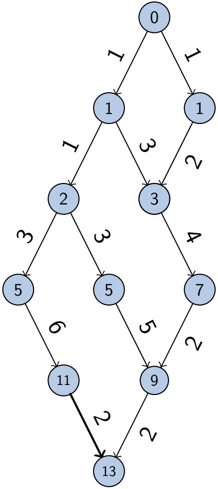
Fun with DAGs

Shortest Paths

Shortest Paths on DAGs

SSSP can be solved in $\mathcal{O}(V + E)$ on DAGs

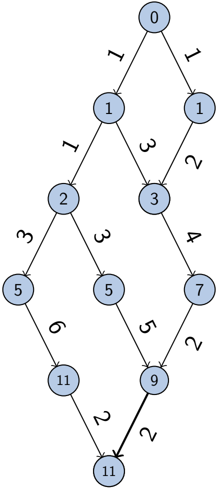
- Compute topological ordering
- Relax SSSP values in topological order



Shortest Paths on DAGs

SSSP can be solved in $\mathcal{O}(V + E)$ on DAGs

- Compute topological ordering
- Relax SSSP values in topological order



Fun with DAGs

Example Code: Shortest Paths

```
vector<int> dist(V, INF);
dist[start] = 0; // initialize all source vertices

// assume given toposort
vector<int> ts = compute_toposort();

for (auto u: ts)
    for (auto p: adj[u]) {
        int v = p.first;
        int w = p.second;
        // relax v
        dist[v] = min(dist[v], dist[u] + w);
    }
```

Fun with DAGs

Example Code: Shortest Paths

```
vector<int> dist(V, INF);
dist[start] = 0; // initialize all source vertices

// assume given toposort
vector<int> ts = compute_toposort();

for (auto u: ts)
    for (auto p: adj[u]) {
        int v = p.first;
        int w = p.second;
        // relax v
        dist[v] = min(dist[v], dist[u] + w);
    }
```

It's flexible!

Changing the relax operation can yield solutions to

- Longest Path
- Counting Paths
- ...

max planck institut informatik

Fun with DAGs

Baking Cake

Problem: Baking Cake

You have an infinite number of helping hands willing to bake cake with you. From the recipe you extract all subtasks to bake a cake as well as their durations. You note which tasks need to be done before which ones. How long does it take to bake the entire cake?

Fun with DAGs

Baking Cake

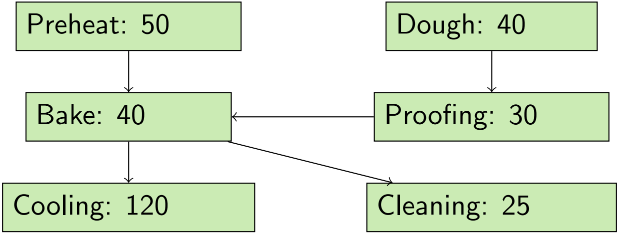
Problem: Baking Cake

You have an infinite number of helping hands willing to bake cake with you. From the recipe you extract all subtasks to bake a cake as well as their durations. You note which tasks need to be done before which ones. How long does it take to bake the entire cake?

```
6 5 # number of steps+cost, ordering constraints
preheat 50
dough 10
proofing 30
bake 40
cooling 120
cleaning 25
dough proofing
preheat bake
proofing bake
bake cooling
bake cleaning
```

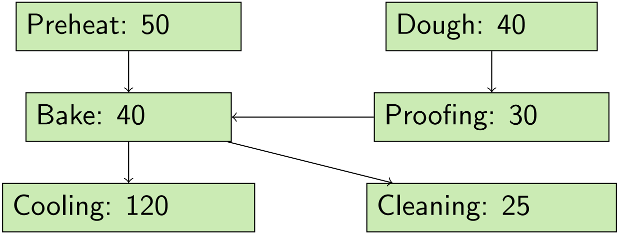
Fun with DAGs

Baking Cake



Fun with DAGs

Baking Cake



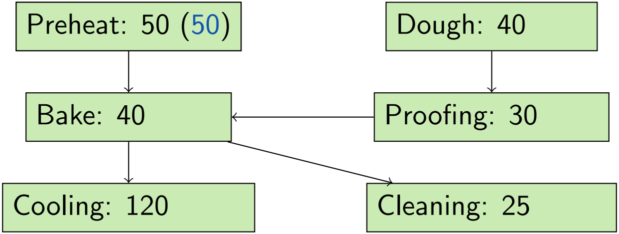
Solution Idea

The jobs must be acyclic for the recipe to terminate. On the resulting DAG

- Traverse the DAG in (any) topological order
- Propagate Maximum Duration to successors
- Find maximum among all sink vertices

Fun with DAGs

Baking Cake



Solution Idea

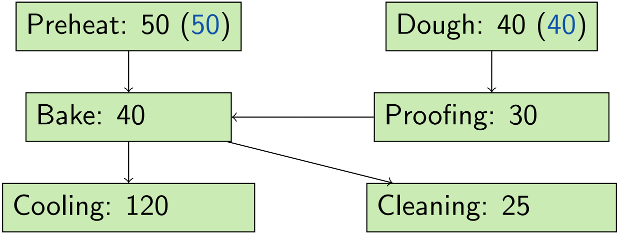
The jobs must be acyclic for the recipe to terminate. On the resulting DAG

- Traverse the DAG in (any) topological order
- Propagate Maximum Duration to successors
- Find maximum among all sink vertices

max planck institut informatik

Fun with DAGs

Baking Cake



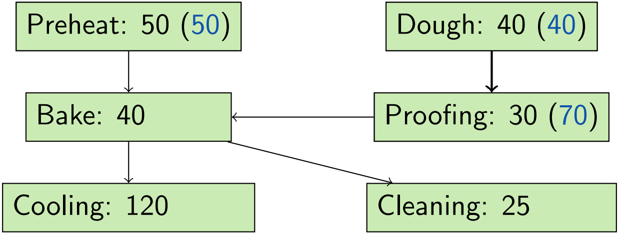
Solution Idea

The jobs must be acyclic for the recipe to terminate. On the resulting DAG

- Traverse the DAG in (any) topological order
- Propagate Maximum Duration to successors
- Find maximum among all sink vertices

Fun with DAGs

Baking Cake



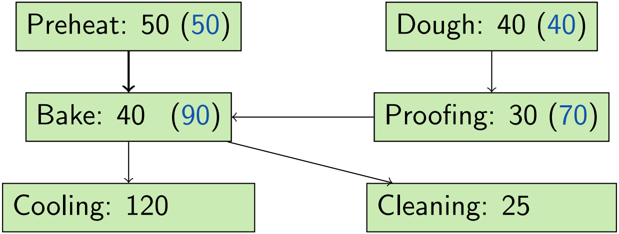
Solution Idea

The jobs must be acyclic for the recipe to terminate. On the resulting DAG

- Traverse the DAG in (any) topological order
- Propagate Maximum Duration to successors
- Find maximum among all sink vertices

Fun with DAGs

Baking Cake



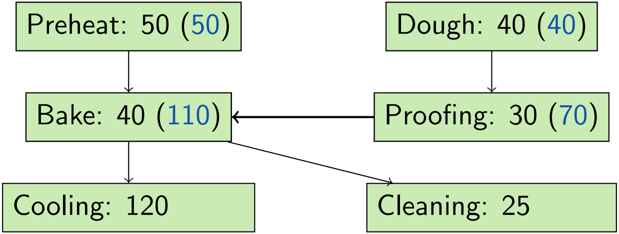
Solution Idea

The jobs must be acyclic for the recipe to terminate. On the resulting DAG

- Traverse the DAG in (any) topological order
- Propagate Maximum Duration to successors
- Find maximum among all sink vertices

Fun with DAGs

Baking Cake



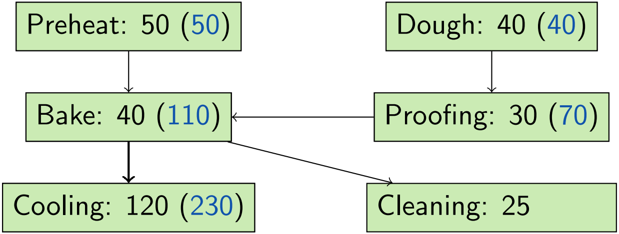
Solution Idea

The jobs must be acyclic for the recipe to terminate. On the resulting DAG

- Traverse the DAG in (any) topological order
- Propagate Maximum Duration to successors
- Find maximum among all sink vertices

Fun with DAGs

Baking Cake



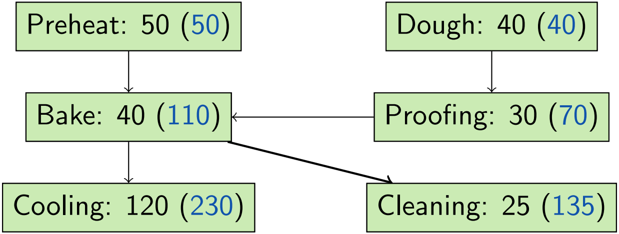
Solution Idea

The jobs must be acyclic for the recipe to terminate. On the resulting DAG

- Traverse the DAG in (any) topological order
- Propagate Maximum Duration to successors
- Find maximum among all sink vertices

Fun with DAGs

Baking Cake



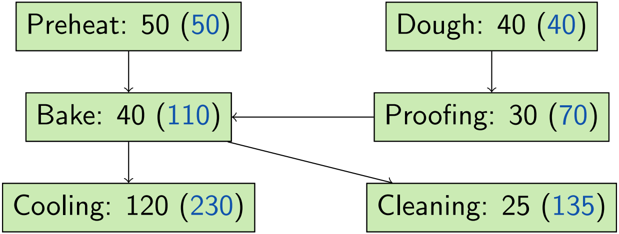
Solution Idea

The jobs must be acyclic for the recipe to terminate. On the resulting DAG

- Traverse the DAG in (any) topological order
- Propagate Maximum Duration to successors
- Find maximum among all sink vertices

Fun with DAGs

Baking Cake



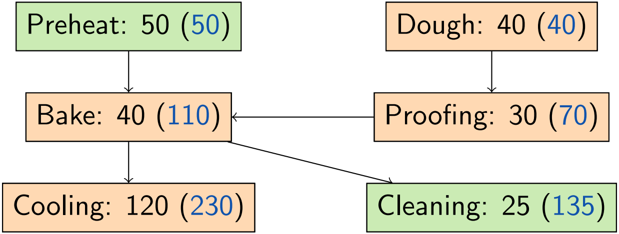
Solution Idea

The jobs must be acyclic for the recipe to terminate. On the resulting DAG

- Traverse the DAG in (any) topological order
- Propagate Maximum Duration to successors
- Find maximum among all sink vertices

Fun with DAGs

Baking Cake



Solution Idea

The jobs must be acyclic for the recipe to terminate. On the resulting DAG

- Traverse the DAG in (any) topological order
- Propagate Maximum Duration to successors
- Find maximum among all sink vertices

Special Graphs

We discussed so far

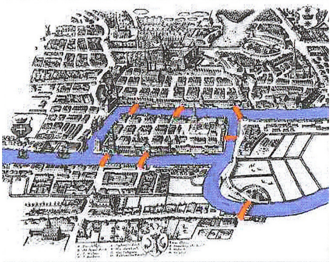
- Bipartite Graphs
- (Strongly) Connected Components
- Directed Acyclic Graphs

Up next

- Eulerian Graphs
- Planar Graphs

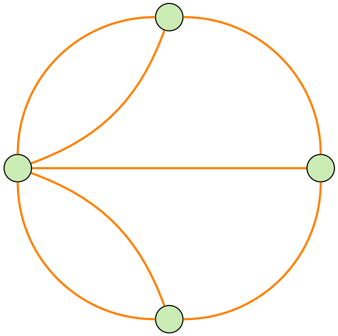
Problem: Königsberger Bridges

Find a tour (a cycle) through Königsberg that crosses each bridge exactly once



Problem: Königsberger Bridges

Find a tour (a cycle) through Königsberg that crosses each bridge exactly once

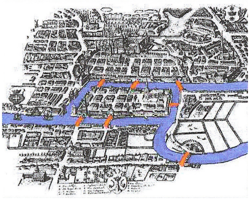


Special Graphs

Eulerian Graphs

Definition: Eulerian Cycle

A tour that visits every edge of a graph exactly once is called *Eulerian Cycle*. A graph that has at least one Eulerian Cycle is called *Eulerian Graph*.



Insight: Classifying Eulerian Cycles

In order to be able to leave every vertex that the cycle enters, every vertex along the path must have an even number of adjacent edges (the degree of a vertex).

Special Graphs

Eulerian Graphs

Euler's Theorem (undirected)

A connected, undirected graph has a Eulerian Cycle if and only if every vertex has even degree.

Euler's Theorem (directed)

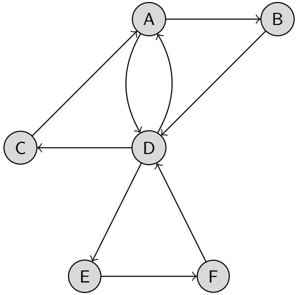
A connected, directed graph has a Eulerian Cycle if and only if the outdegree of every vertex is equal to its indegree.

Problem: Finding Eulerian Cycles

Given a graph G, determine if the graph is Eulerian and if so, output a Eulerian Cycle.

Special Graphs

Eulerian Graphs

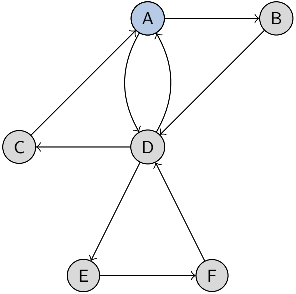


Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

Special Graphs

Eulerian Graphs

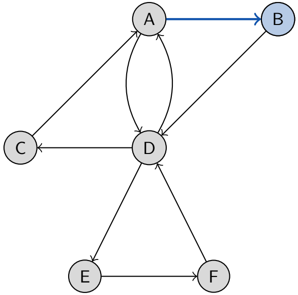


Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

Special Graphs

Eulerian Graphs

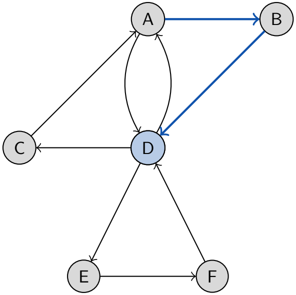


Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

Special Graphs

Eulerian Graphs

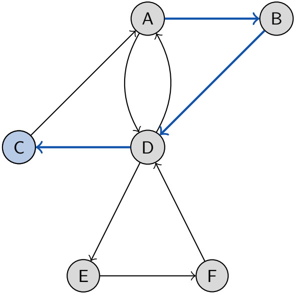


Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

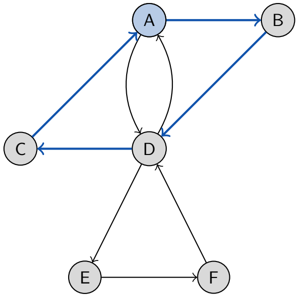
Special Graphs

Eulerian Graphs



Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

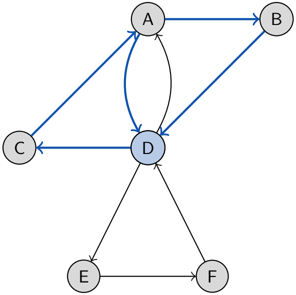


Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

Special Graphs

Eulerian Graphs

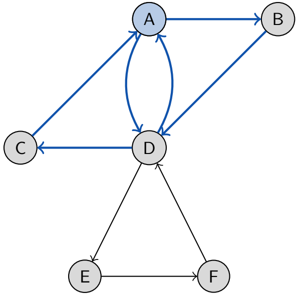


Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

Special Graphs

Eulerian Graphs



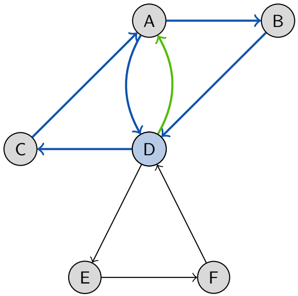
Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

Special Graphs

Eulerian Graphs

A



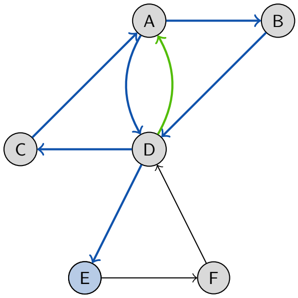
Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

Special Graphs

Eulerian Graphs

A



Eulerian Cycle: Algorithm Idea

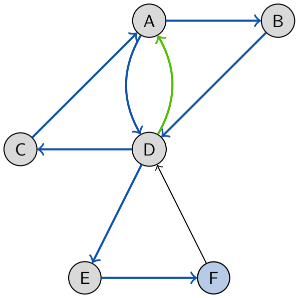
Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

max planck institut informatik

Special Graphs

Eulerian Graphs

A



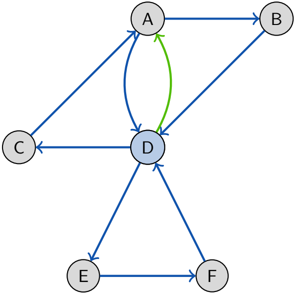
Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

Special Graphs

Eulerian Graphs

A



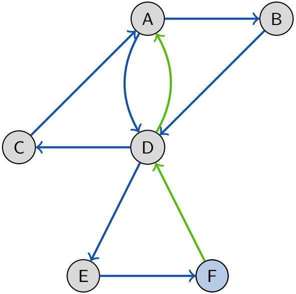
Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

Special Graphs

Eulerian Graphs

D - A



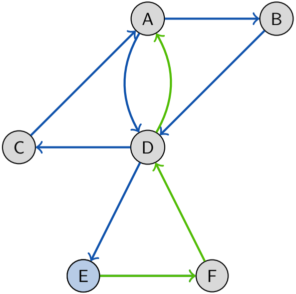
Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

Special Graphs

Eulerian Graphs

F - D - A



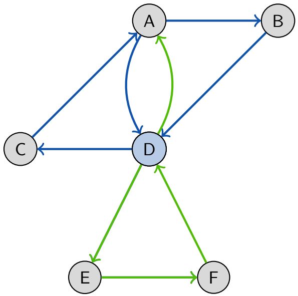
Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

Special Graphs

Eulerian Graphs

E - F - D - A



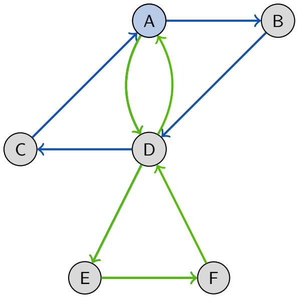
Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

Special Graphs

Eulerian Graphs

D - E - F - D - A



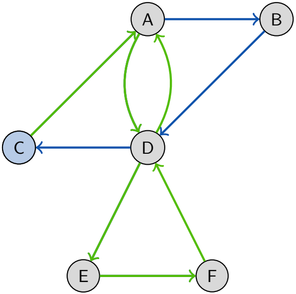
Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

Special Graphs

Eulerian Graphs

A - D - E - F - D - A



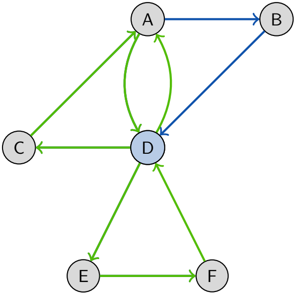
Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

Special Graphs

Eulerian Graphs

C - A - D - E - F - D - A



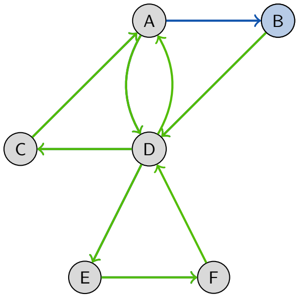
Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

Special Graphs

Eulerian Graphs

D - C - A - D - E - F - D - A



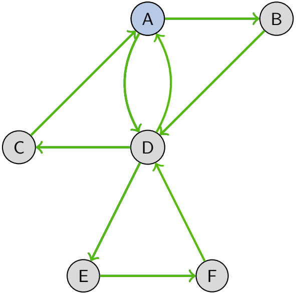
Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

Special Graphs

Eulerian Graphs

B - D - C - A - D - E - F - D - A



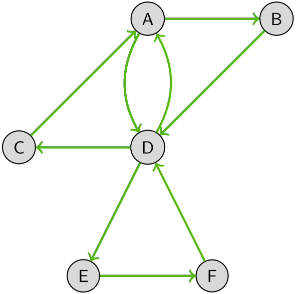
Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

Special Graphs

Eulerian Graphs

A - B - D - C - A - D - E - F - D - A



Eulerian Cycle: Algorithm Idea

Traverse the graph until you have completed a cycle. Backtrack once all outgoing edges are already taken. Output vertices whilst backtracking.

Special Graphs

Example Code: Eulerian Cycles

```
vector<int> indegree; // store indegree of each vertex
deque<int> cycle;

// test if solution can exist
for (int i = 0; i < V; i++)
  if (indegree[i] != adj[i].size()) {
    cout << "IMPOSSIBLE" << endl;
    exit(0);
  }

// start anywhere
find_cycle(0); // populate cycle
// test against disconnected graphs
if (cycle.size() != E + 1) {
  cout << "IMPOSSIBLE" << endl;
  exit(0);
}
for (auto v: cycle)
  cout << v << " ";
cout << endl;
```

Special Graphs

Example Code: Eulerian Cycles

```
void find_cycle(int u) {
  while (adj[u].size()) {
      int v = adj[u].back();
      adj[u].pop_back();
      find_cycle(v);
  }
  cycle.push_front(i);
}
```

max planck institut informatik

Special Graphs

Planar Graphs

Planar Graph

A Graph G is called planar, if it can be drawn in 2D space without any two edges crossing each other.

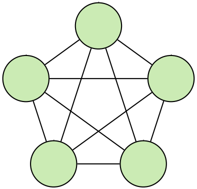


Figure: A non-planar graph

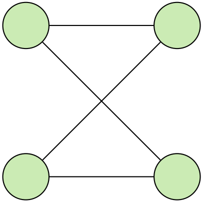


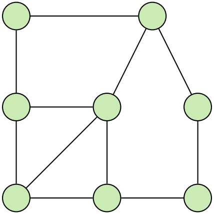
Figure: A planar graph

Special Graphs

Planar Graphs

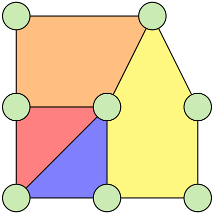
Problem: Counting Neighborhoods

Given a planar street map of a city. How many neigbourhood blocks are there?



Problem: Counting Neighborhoods

Given a planar street map of a city. How many neigbourhood blocks are there?



Problem: Counting Neighborhoods

Given a planar street map of a city. How many neigbourhood blocks are there?

Euler's Formula

Let G be a planar graph. The identity

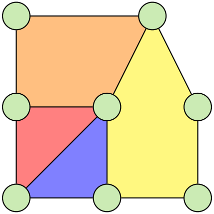
$$V - E + F = 1$$

holds, where F denotes the number of (enclosed) faces of the graph

Special Graphs

Problem: Counting Neighborhoods

Given a planar street map of a city. How many neigbourhood blocks are there?



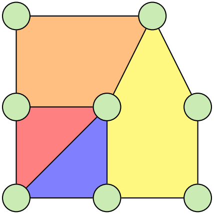
Euler's Formula

$$V - E + F = 1$$

Special Graphs

Problem: Counting Neighborhoods

Given a planar street map of a city. How many neigbourhood blocks are there?



Euler's Formula

$$V - E + F = 1$$

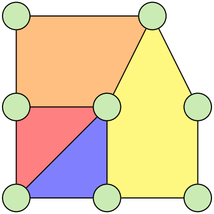
Thus, for the above graph, we get

$$8 - 11 + F = 1$$

Special Graphs

Problem: Counting Neighborhoods

Given a planar street map of a city. How many neigbourhood blocks are there?



Euler's Formula

$$V - E + F = 1$$

Thus, for the above graph, we get

$$F = 4$$