CSE 431/531: Algorithm Analysis and Design (Fall 2024)

Connectivity, Graph Traversal and Bipartiteness

Lecturer: Kelin Luo

Department of Computer Science and Engineering
University at Buffalo

Announcements: Quiz 3

- Posted on Ublearns
- Should take $<$ 30 minutes, 2 attempts
- Due Tue 10 Sep @ 11:59PM

Outline

Connectivity Problem

Input: graph $G = (V, E)$, (using linked lists)

two vertices $s, t \in V$

Output: whether there is a path connecting s to t in G

Connectivity Problem

Input: graph $G = (V, E)$, (using linked lists)

two vertices $s, t \in V$

Output: whether there is a path connecting s to t in G

- Algorithm: starting from s, search for all vertices that are reachable from s and check if the set contains t

Connectivity Problem

Input: graph $G = (V, E)$, (using linked lists)
two vertices $s, t \in V$

Output: whether there is a path connecting s to t in G

- Algorithm: starting from s, search for all vertices that are reachable from s and check if the set contains t
  - Breadth-First Search (BFS)

Connectivity Problem

Input: graph $G = (V, E)$, (using linked lists)

two vertices $s, t \in V$

Output: whether there is a path connecting s to t in G

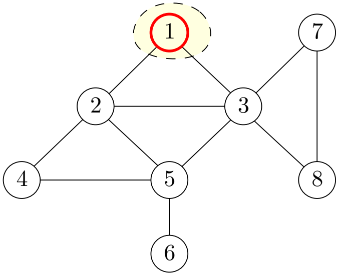
- Algorithm: starting from s, search for all vertices that are reachable from s and check if the set contains t
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)

Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- L_{j+1} contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in L_j

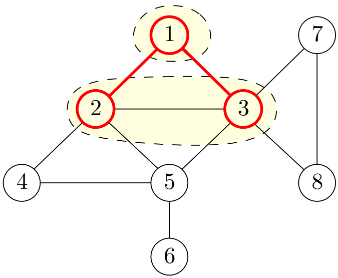
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- L_{j+1} contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in L_j



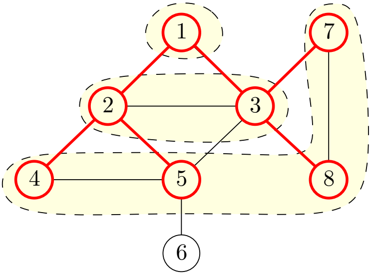
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- L_{j+1} contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in L_j



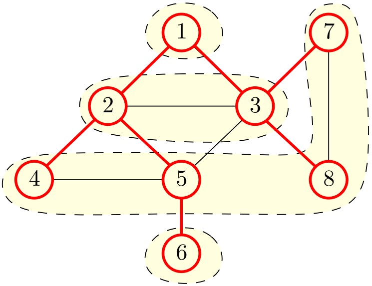
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- L_{j+1} contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in L_j



Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- L_{j+1} contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in L_j



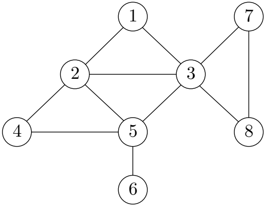
Implementing BFS using a Queue

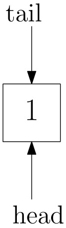
BFS(s)

$head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$
mark s as "visited" and all other vertices as "unvisited"
while $head \leq tail$ **do**
    $v \leftarrow queue[head], head \leftarrow head + 1$
    for all neighbors u of v **do**
        if u is "unvisited" **then**
            $tail \leftarrow tail + 1, queue[tail] = u$
            mark u as "visited"

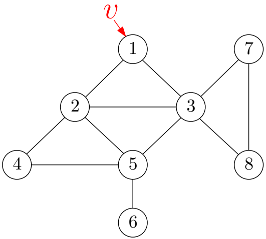
- Running time: $O(n + m)$.

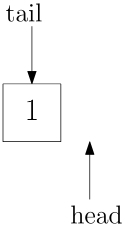
Example of BFS via Queue



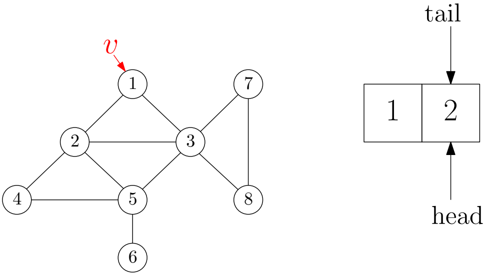


Example of BFS via Queue

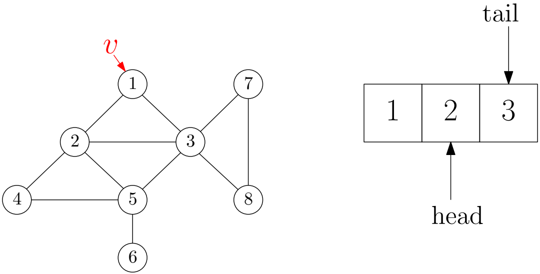




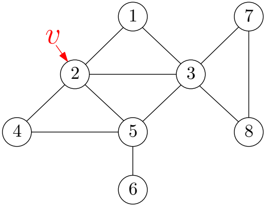
Example of BFS via Queue

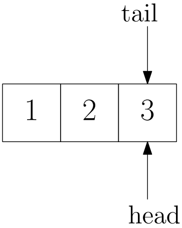


Example of BFS via Queue

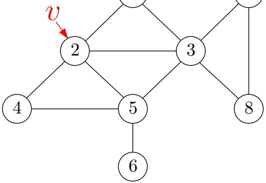


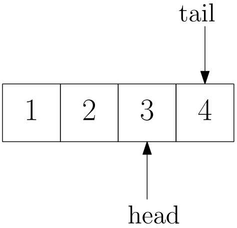
Example of BFS via Queue



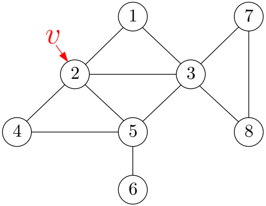


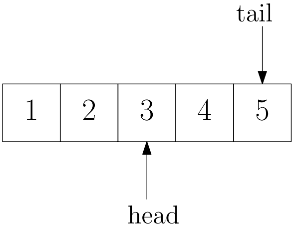
Example of BFS via Queue



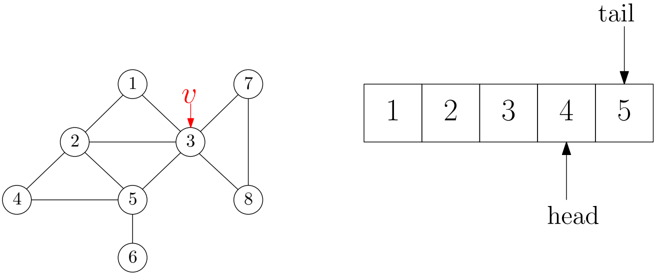


Example of BFS via Queue

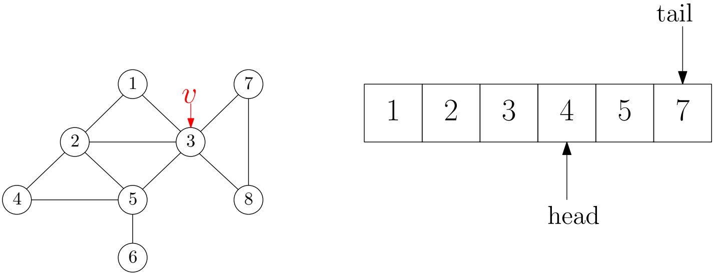




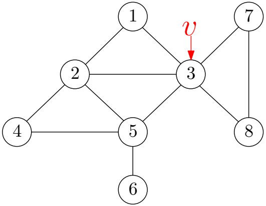
Example of BFS via Queue

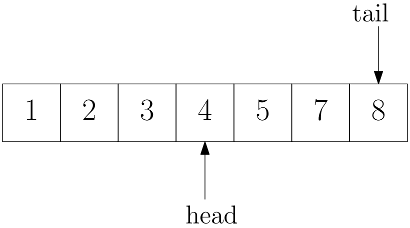


Example of BFS via Queue

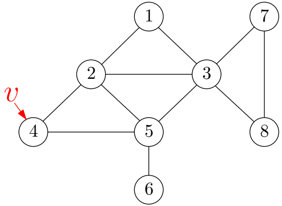


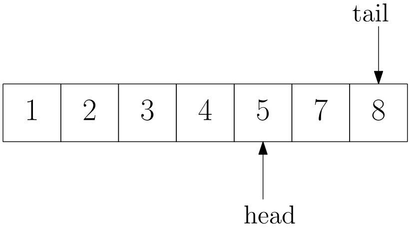
Example of BFS via Queue



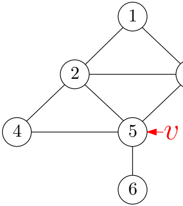


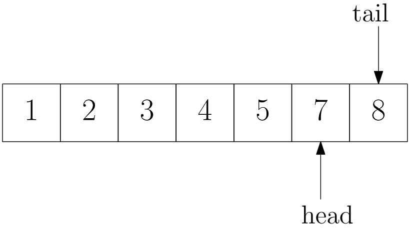
Example of BFS via Queue



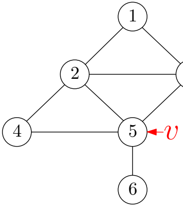


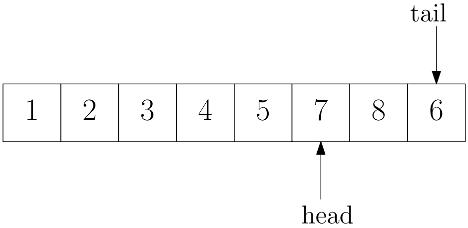
Example of BFS via Queue



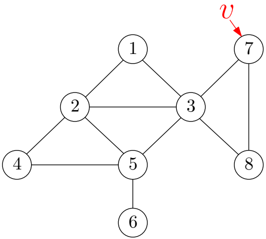


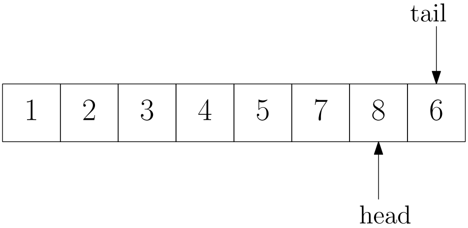
Example of BFS via Queue



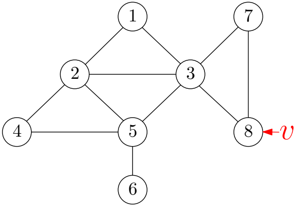


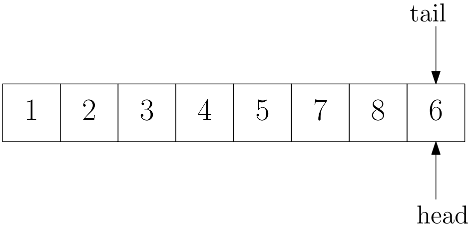
Example of BFS via Queue



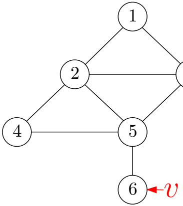


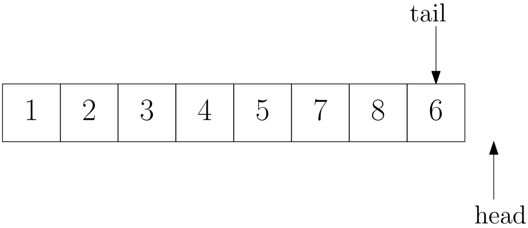
Example of BFS via Queue





Example of BFS via Queue





Example of BFS via Queue

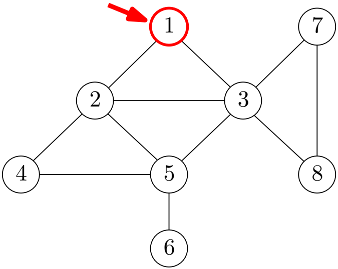
Edges included in BFS algorithm starting with vertex 1: {1, 2}, {1, 3}, {2, 4}, {2, 5}, {3, 7}, {3, 8}, {5, 6}

Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back

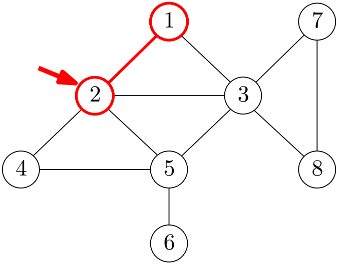
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



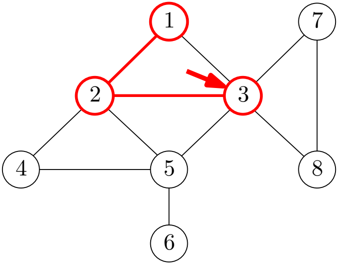
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



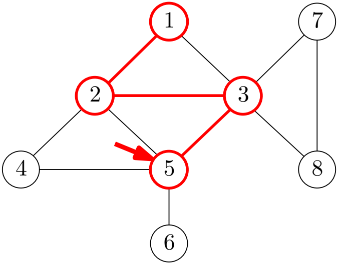
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



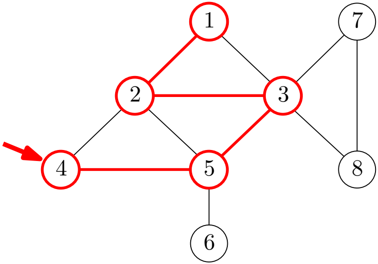
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



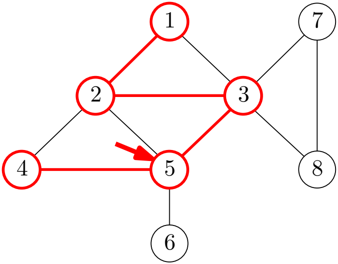
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



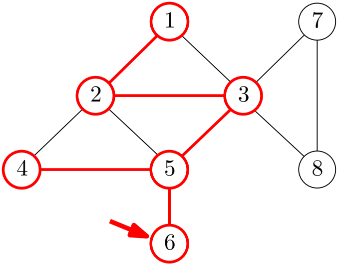
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



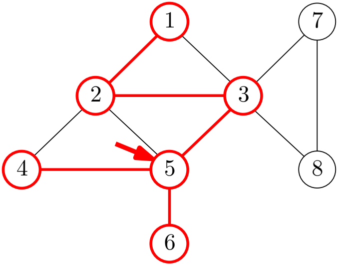
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



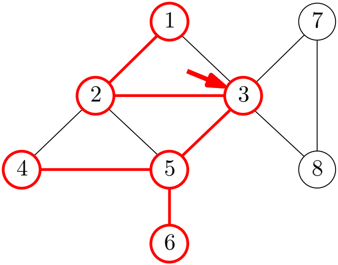
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



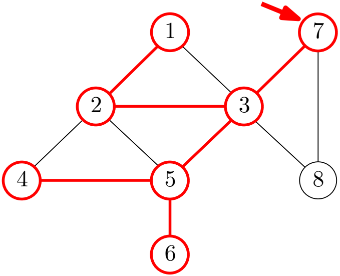
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



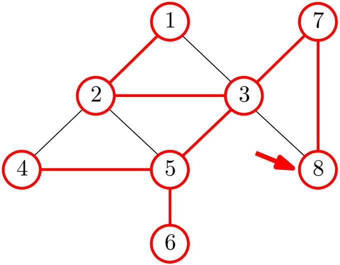
Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back



Depth-First Search (DFS)

- Starting from s
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back

Edges included in DFS algorithm starting with vertex 1: {1, 2}, {2, 3}, {3, 5}, {5, 4}, {5, 6}, {3, 7}, {7, 8}

Implementing DFS using Recurrsion

DFS(s)

```
mark all vertices as "unvisited"
recursive-DFS(s)
```

recursive-DFS(v)

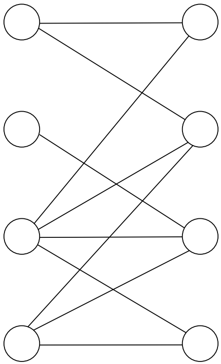
```
mark v as "visited"
for all neighbors u of v do
    if u is unvisited then recursive-DFS(u)
```

Outline

Outline

Testing Bipartiteness: Applications of BFS

Def. A graph $G = (V, E)$ is a bipartite graph if there is a partition of V into two sets L and R such that for every edge $(u, v) \in E$, either $u \in L, v \in R$ or $v \in L, u \in R$.



Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$

Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g

Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of s must be in R

Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of s must be in R
- Neighbors of neighbors of s must be in L

Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of s must be in R
- Neighbors of neighbors of s must be in L
- $\cdots$

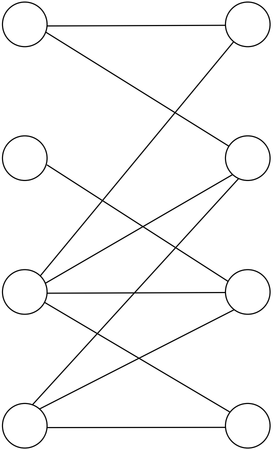
Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of s must be in R
- Neighbors of neighbors of s must be in L
- $\cdots$
- Report "not a bipartite graph" if contradiction was found

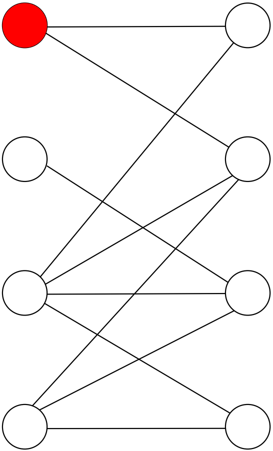
Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of s must be in R
- Neighbors of neighbors of s must be in L
- $\cdots$
- Report "not a bipartite graph" if contradiction was found
- If G contains multiple connected components, repeat above algorithm for each component

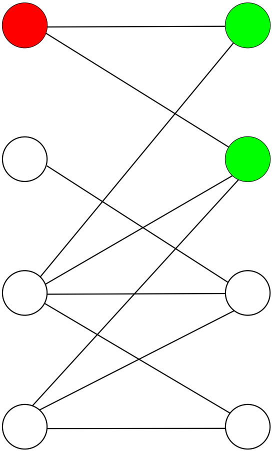
Test Bipartiteness



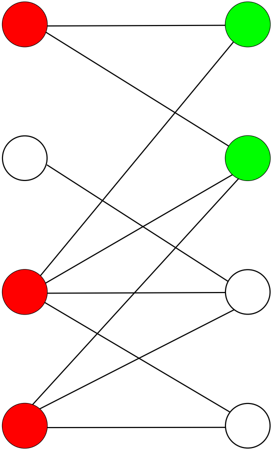
Test Bipartiteness



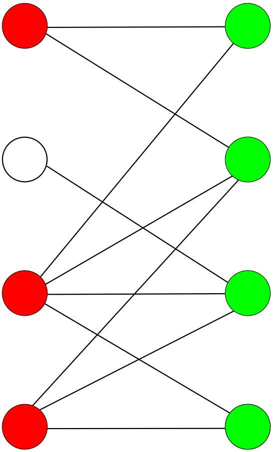
Test Bipartiteness



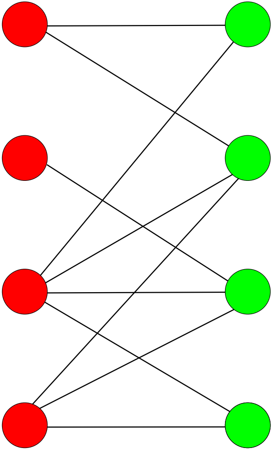
Test Bipartiteness



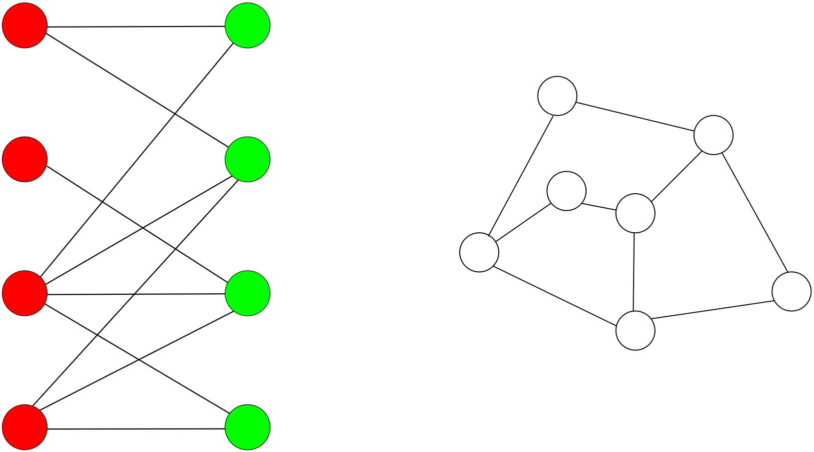
Test Bipartiteness



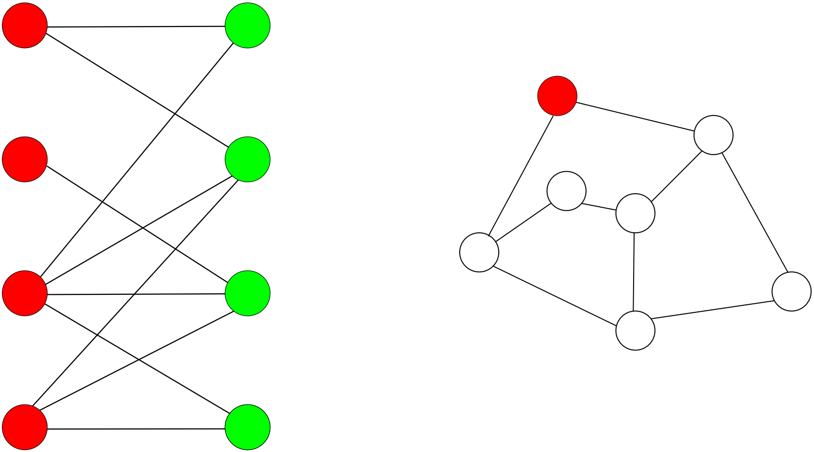
Test Bipartiteness



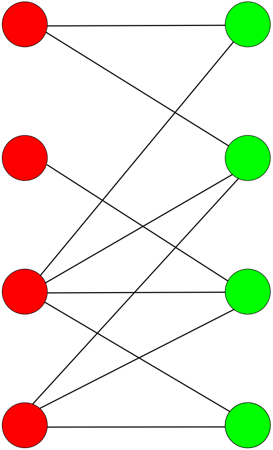
Test Bipartiteness

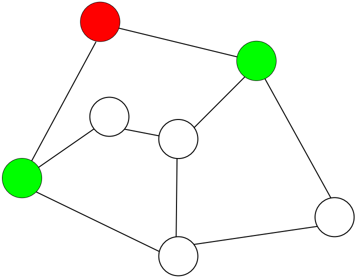


Test Bipartiteness

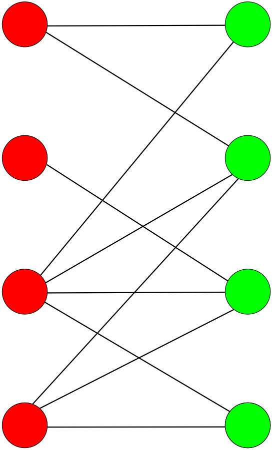


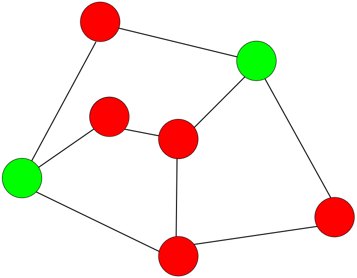
Test Bipartiteness



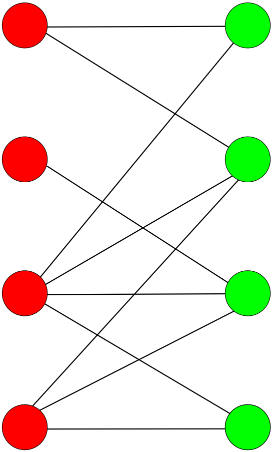


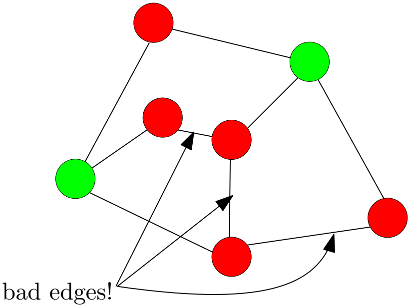
Test Bipartiteness





Test Bipartiteness





Testing Bipartiteness using BFS

BFS(s)

$head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$
mark s as "visited" and all other vertices as "unvisited"
while $head \leq tail$ **do**
    $v \leftarrow queue[head], head \leftarrow head + 1$
    for all neighbors u of v **do**
        if u is "unvisited" **then**
            $tail \leftarrow tail + 1, queue[tail] = u$
            mark u as "visited"

Testing Bipartiteness using BFS

test-bipartiteness(s)

$head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$
mark s as "visited" and all other vertices as "unvisited"
$color[s] \leftarrow 0$
while $head \leq tail$ **do**
    $v \leftarrow queue[head], head \leftarrow head + 1$
    for all neighbors u of v **do**
        if u is "unvisited" **then**
            $tail \leftarrow tail + 1, queue[tail] = u$
            mark u as "visited"
            $color[u] \leftarrow 1 - color[v]$
        else if $color[u] = color[v]$ **then**
            print("G is not bipartite") and exit

Testing Bipartiteness using BFS

```
mark all vertices as "unvisited"
for each vertex v ∈ V do
    if v is "unvisited" then
        test-bipartiteness(v)
print("G is bipartite")
```

Testing Bipartiteness using BFS

```
mark all vertices as "unvisited"
for each vertex v ∈ V do
    if v is "unvisited" then
        test-bipartiteness(v)
print("G is bipartite")
```

Obs. Running time of algorithm $= O(n+m)$

Testing Bipartiteness using DFS

test-bipartiteness-DFS(s)

1: mark all vertices as "unvisited"
2: recursive-test-DFS(s)

recursive-test-DFS(v)

1: mark v as "visited"
2: **for** all neighbors u of v **do**
3:     **if** u is unvisited **then** , recursive-test-DFS(u)

Testing Bipartiteness using DFS

test-bipartiteness-DFS(s)

mark all vertices as "unvisited"
$color[s] \leftarrow 0$
recursive-test-DFS(s)

recursive-test-DFS(v)

mark v as "visited"
for all neighbors u of v **do**
    if u is unvisited **then**
        $color[u] \leftarrow 1 - color[v]$, recursive-test-DFS($u$)
    else if $color[u] = color[v]$ **then**
        print("G is not bipartite") and exit

Testing Bipartiteness using DFS

mark all vertices as "unvisited"
for each vertex $v \in V$ **do**
    if v is "unvisited" **then**
        test-bipartiteness-DFS(v)
print("G is bipartite")

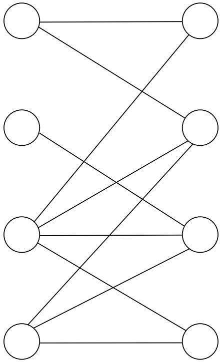
Testing Bipartiteness using DFS

mark all vertices as "unvisited"
for each vertex $v \in V$ **do**
    if v is "unvisited" **then**
        test-bipartiteness-DFS(v)
print("G is bipartite")

Obs. Running time of algorithm $= O(n+m)$

Bipartite Graph

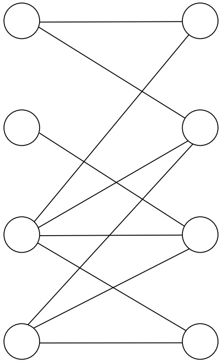
Def. An undirected graph $G = (V, E)$ is a bipartite graph if there is a partition of V into two sets L and R such that for every edge $(u, v) \in E$, either $u \in L, v \in R$ or $v \in L, u \in R$.



Bipartite Graph

Def. An undirected graph $G = (V, E)$ is a bipartite graph if there is a partition of V into two sets L and R such that for every edge $(u, v) \in E$, either $u \in L, v \in R$ or $v \in L, u \in R$.

Obs. Bipartite graph may contain cycles.

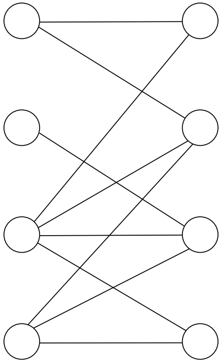


Bipartite Graph

Def. An undirected graph $G = (V, E)$ is a bipartite graph if there is a partition of V into two sets L and R such that for every edge $(u, v) \in E$, either $u \in L, v \in R$ or $v \in L, u \in R$.

Obs. Bipartite graph may contain cycles.

Obs. If a graph is a tree, then it is also a bipartite graph.



BFS and DFS

Obs. BFS and DFS naturally induce a tree.

BFS and DFS

Obs. BFS and DFS naturally induce a tree.

Obs. If G is a tree, then BFS tree $=$ DFS tree.

BFS and DFS

Obs. BFS and DFS naturally induce a tree.

Obs. If G is a tree, then BFS tree $=$ DFS tree.

Obs. If BFS tree $=$DFS tree, then G is a tree.