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

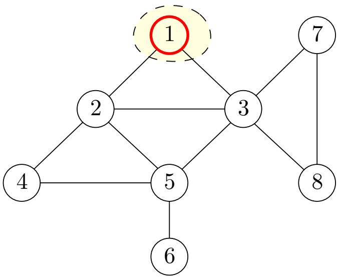
- Algorithm: starting from s, search for all vertices that are reachable from s and check if the set contains t
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)

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

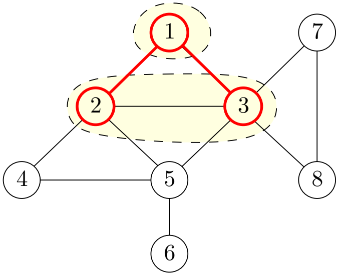
Breadth-First Search (BFS)

- 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



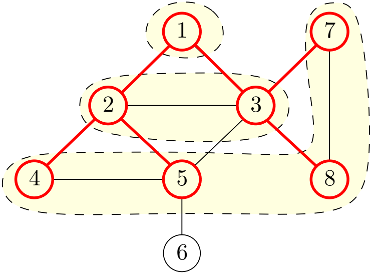
Breadth-First Search (BFS)

- 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



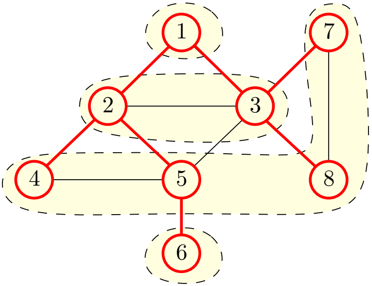
Breadth-First Search (BFS)

- 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



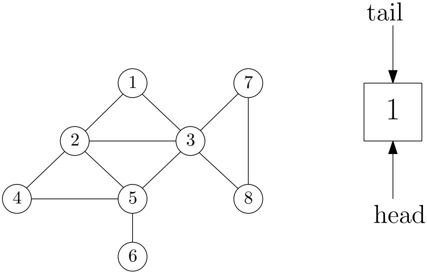
Implementing BFS using a Queue

BFS(s)

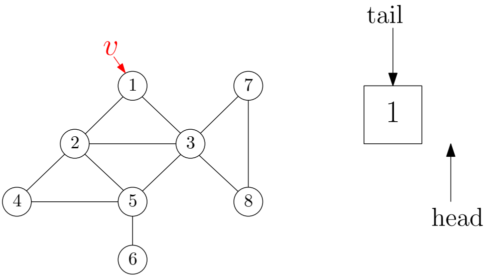
```
head ← 1, tail ← 1, queue[1] ← s
mark s as "visited" and all other vertices as "unvisited"
while head ≤ tail do
    v ← queue[head], head ← head + 1
    for all neighbors u of v do
        if u is "unvisited" then
            tail ← tail + 1, queue[tail] = u
            mark u as "visited"
```

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

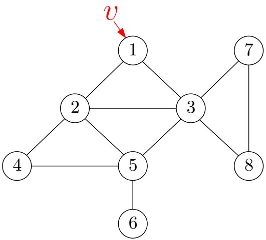
Example of BFS via Queue

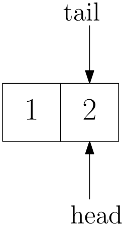


Example of BFS via Queue

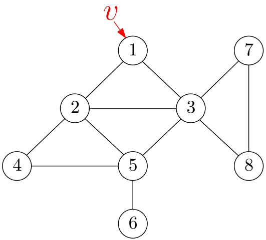


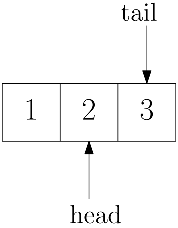
Example of BFS via Queue



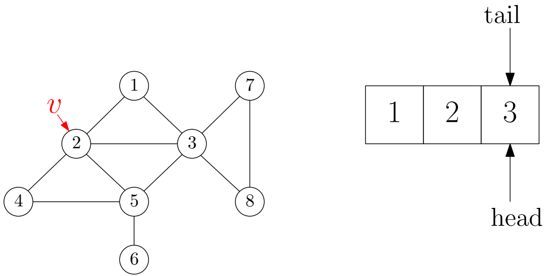


Example of BFS via Queue

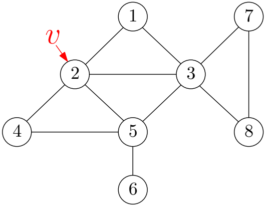


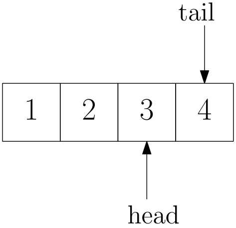


Example of BFS via Queue

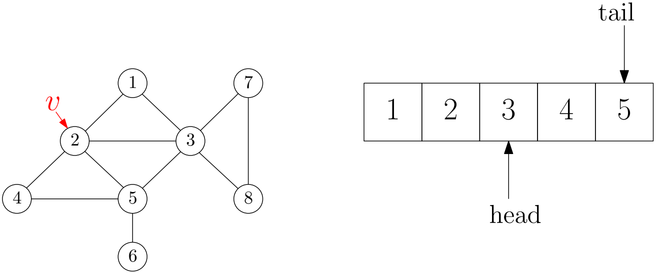


Example of BFS via Queue

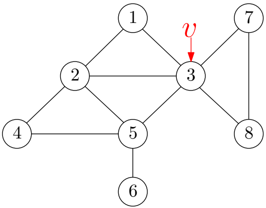


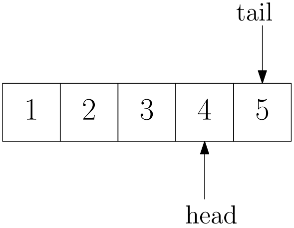


Example of BFS via Queue

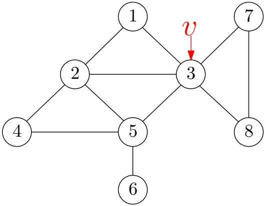


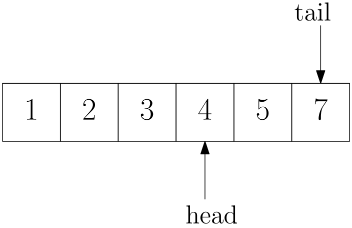
Example of BFS via Queue



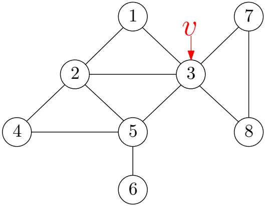


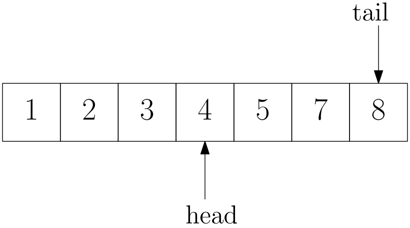
Example of BFS via Queue



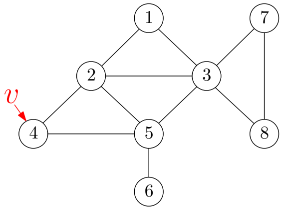


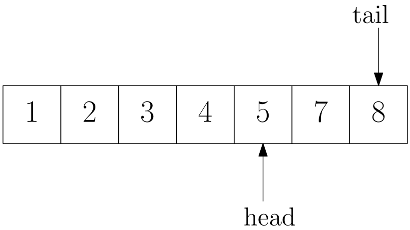
Example of BFS via Queue

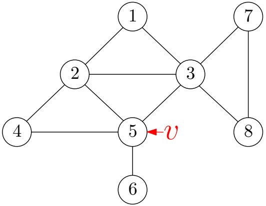


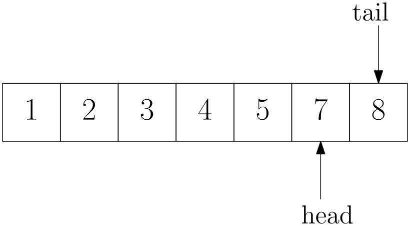


Example of BFS via Queue

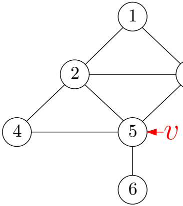


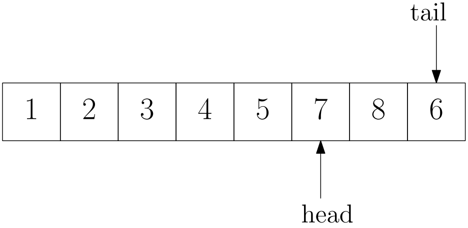



1
2
3
4
5
6
7
8
v

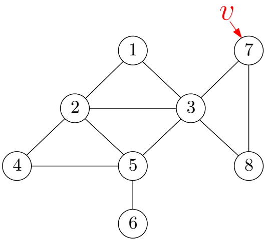

tail
1
2
3
4
5
7
8
head

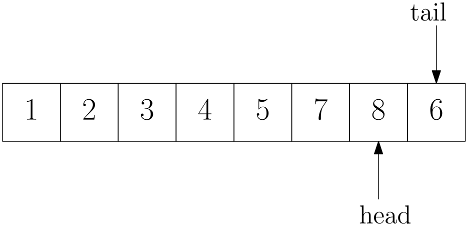
Example of BFS via Queue



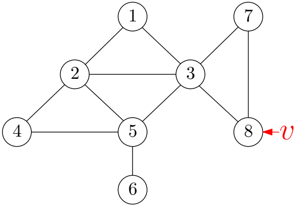


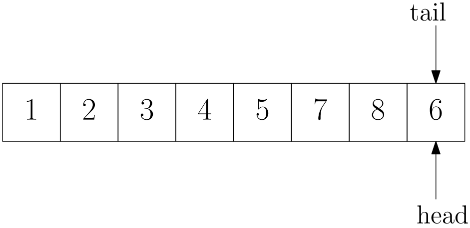
Example of BFS via Queue



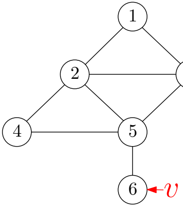


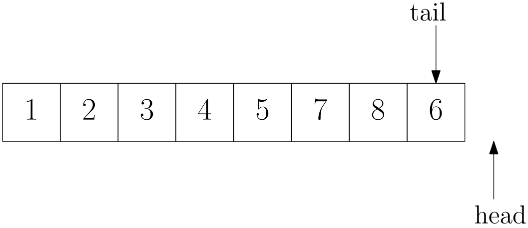
Example of BFS via Queue





Example of BFS via Queue



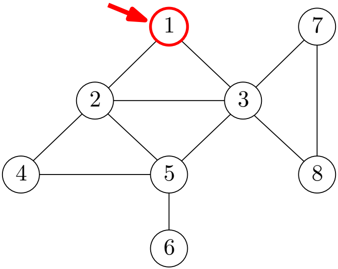


Depth-First Search (DFS)

- 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

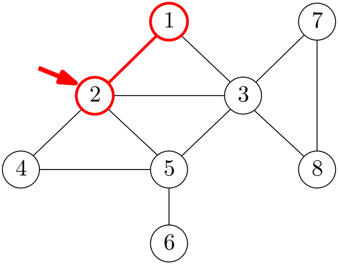
Depth-First Search (DFS)

- 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



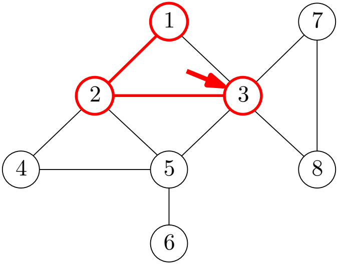
Depth-First Search (DFS)

- 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



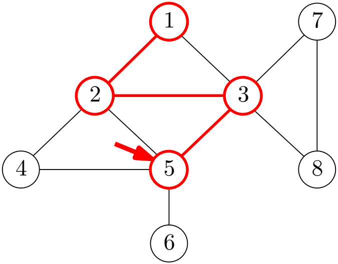
Depth-First Search (DFS)

- 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



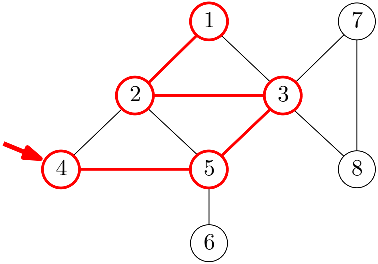
Depth-First Search (DFS)

- 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



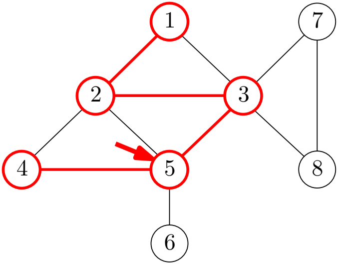
Depth-First Search (DFS)

- 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



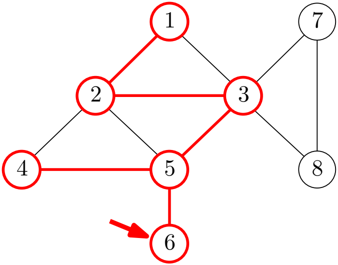
Depth-First Search (DFS)

- 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



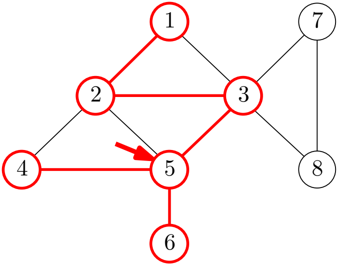
Depth-First Search (DFS)

- 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



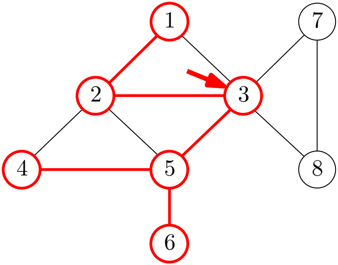
Depth-First Search (DFS)

- 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



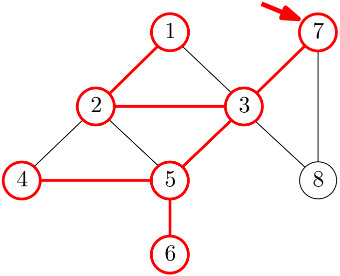
Depth-First Search (DFS)

- 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



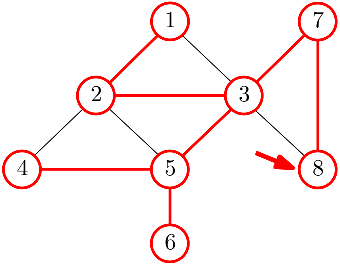
Depth-First Search (DFS)

- 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



Implementing DFS using Recurrsion

DFS(s)

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

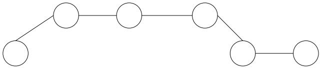
recursive-DFS(v)

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

Outline

Path Graph (or Linear Graph)

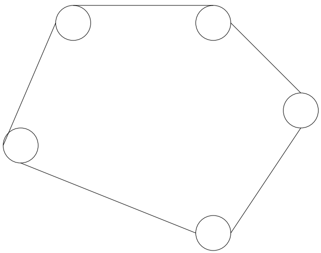
Def. An undirected graph $G = (V, E)$ is a path if the vertices can be listed in an order $\{v_1, v_2, ..., v_n\}$ such that the edges are the $\{v_i, v_{i+1}\}$ where $i = 1, 2, ..., n - 1$.



- Path graphs are connected graphs.

Cycle Graph (or Circular Graph)

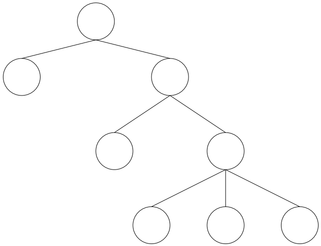
Def. An undirected graph $G = (V, E)$ is a cycle if its vertices can be listed in an order $v_1, v_2, ..., v_n$ such that the edges are the $\{v_i, v_{i+1}\}$ where $i = 1, 2, ..., n-1$, plus the edge $\{v_n, v_1\}$.



- The degree of all vertices is 2.

Tree

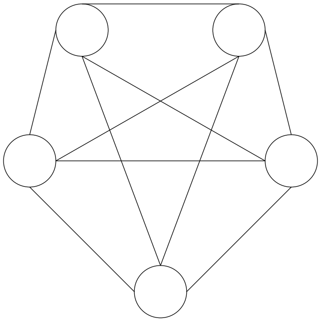
Def. An undirected graph $G = (V, E)$ is a tree if any two vertices are connected by exactly one path. Or the graph is a connected acyclic graph.



- Most important type of special graphs: most computational problems are easier to solve on trees or lines.

Complete Graph

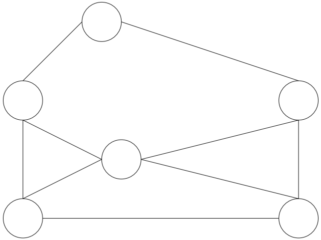
Def. An undirected graph $G = (V, E)$ is a complete graph if each pair of vertices is joined by an edge.



- A complete graph contains all possible edges.

Planar Graph

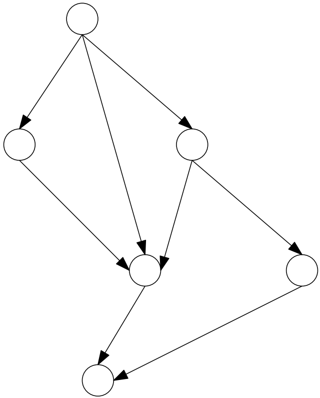
Def. An undirected graph $G = (V, E)$ is a planar graph if its vertices and edges can be drawn in a plane such that no two of the edges intersect.



- Most computational problems have good solutions in a planar graph.

Directed Acyclic Graph (DAG)

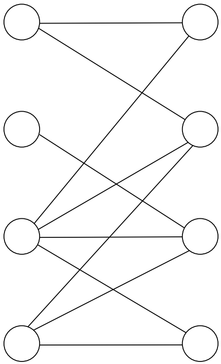
Def. A directed graph $G = (V, E)$ is a directed acyclic graph if it is a directed graph with no directed cycles



- DAG is equivalent to a partial ordering of nodes.

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 $v \in L, u \in R$.

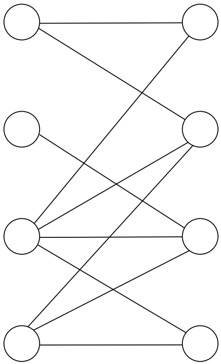


Outline

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
- $\cdots$

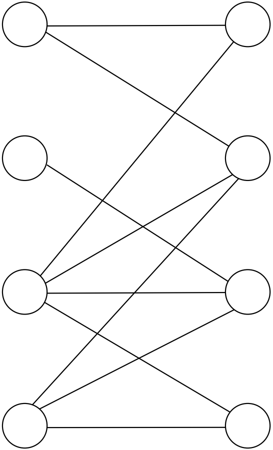
Testing Bipartiteness

- 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

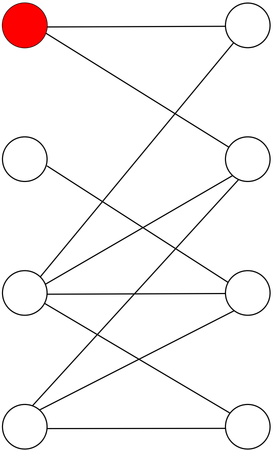
Testing Bipartiteness

- 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

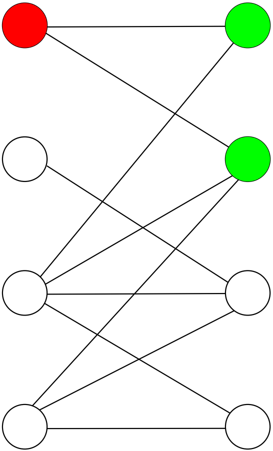
Test Bipartiteness



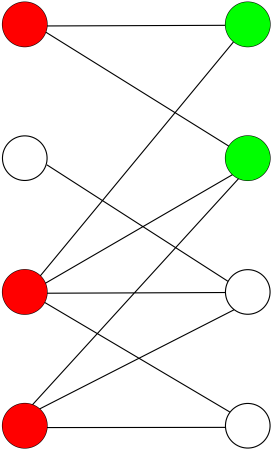
Test Bipartiteness



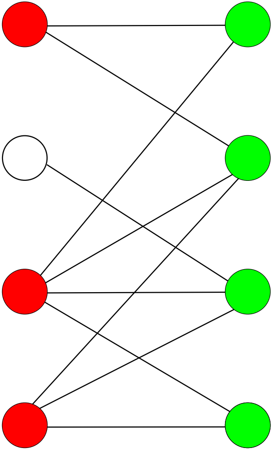
Test Bipartiteness



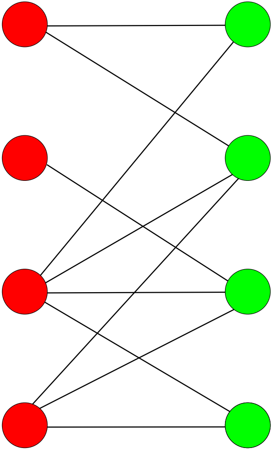
Test Bipartiteness



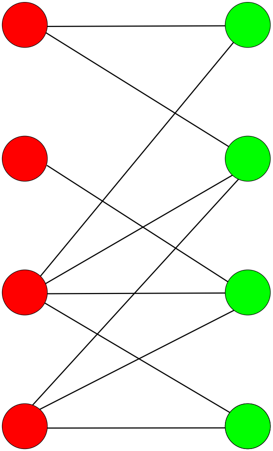
Test Bipartiteness

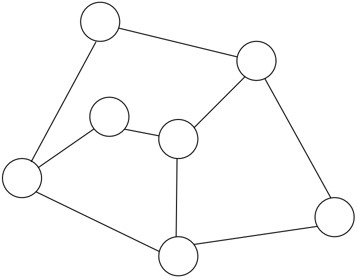


Test Bipartiteness

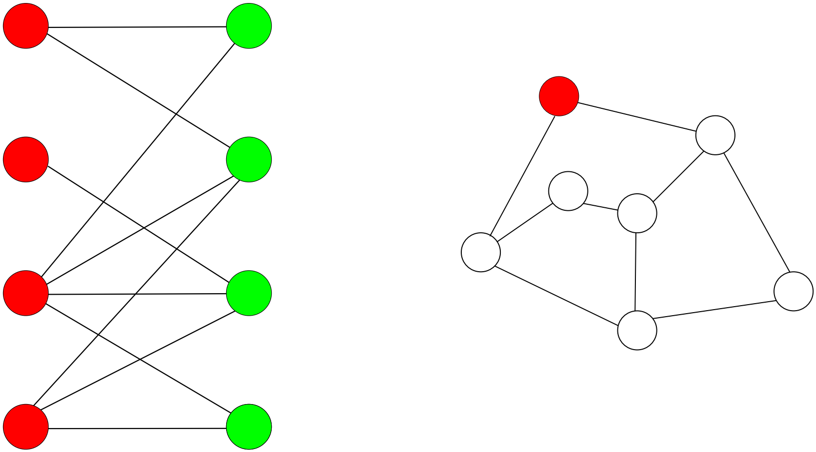


Test Bipartiteness

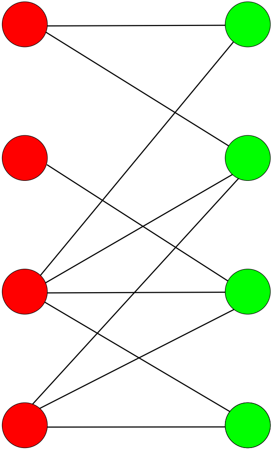


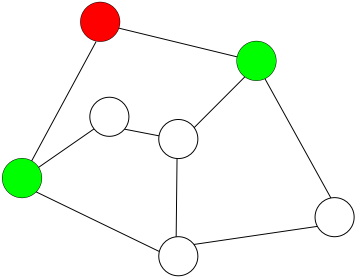


Test Bipartiteness

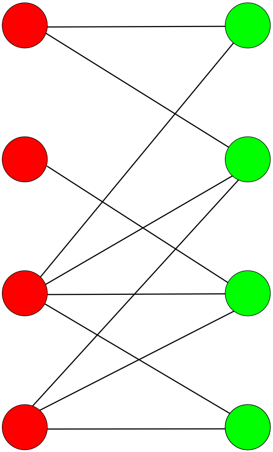


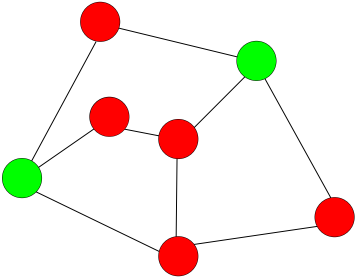
Test Bipartiteness



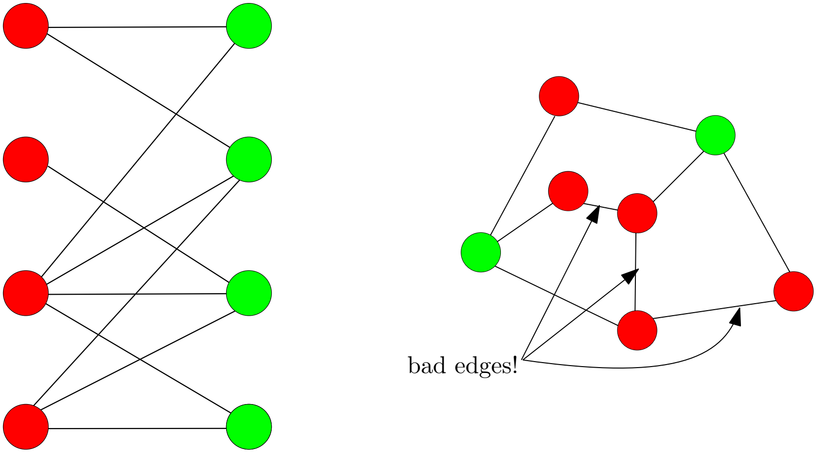


Test Bipartiteness





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