Outline

Graphs

- Graphs are one of the unifying themes of computer science.

Graphs

- Graphs are one of the unifying themes of computer science.
- That so many different structures can be modeled using a single formalism is a source of great power to the educated programmer.

Graphs

- Graphs are one of the unifying themes of computer science.
- That so many different structures can be modeled using a single formalism is a source of great power to the educated programmer.

Formal Definition

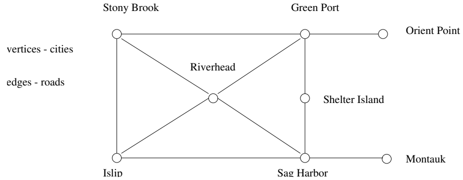
A graph $G = (V, E)$ is defined by a set of *vertices* V, and a set of *edges* E consisting of ordered or unordered pairs of vertices from V.

Graphs

- Graphs are one of the unifying themes of computer science.
- That so many different structures can be modeled using a single formalism is a source of great power to the educated programmer.

Example: Road Networks

In modeling a road network, the vertices may represent the cities or junctions, certain pairs of which are connected by roads/edges.

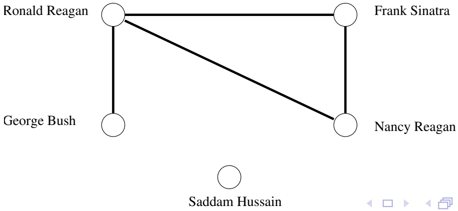


Graphs

- Graphs are one of the unifying themes of computer science.
- That so many different structures can be modeled using a single formalism is a source of great power to the educated programmer.

Example: Friendship Graph

A graph where the vertices are people, and there is an edge between two people if and only if they are friends.



Flavors of Graphs

- The first step in any graph problem is determining which flavor of graph you are dealing with.

Flavors of Graphs

- The first step in any graph problem is determining which flavor of graph you are dealing with.
- Learning to talk the talk is an important part of walking the walk.

Flavors of Graphs

- The first step in any graph problem is determining which flavor of graph you are dealing with.
- Learning to talk the talk is an important part of walking the walk.
- The flavor of graph has a big impact on which algorithms are appropriate and efficient.

Flavors of Graphs

Directed vs. Undirected Graphs

Flavors of Graphs

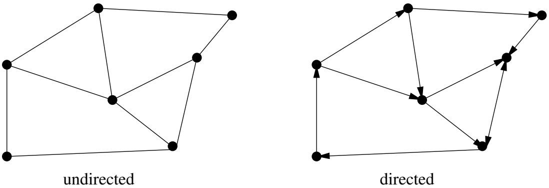
Directed vs. Undirected Graphs

A graph $G = (V, E)$ is *undirected* if edge $(x, y) \in E$ implies that (y, x) is also in E.

Flavors of Graphs

Directed vs. Undirected Graphs

A graph $G = (V, E)$ is *undirected* if edge $(x, y) \in E$ implies that (y, x) is also in E.



Flavors of Graphs

Directed vs. Undirected Graphs

A graph $G = (V, E)$ is *undirected* if edge $(x, y) \in E$ implies that (y, x) is also in E.

Applications/Examples:

Flavors of Graphs

Directed vs. Undirected Graphs

A graph $G = (V, E)$ is *undirected* if edge $(x, y) \in E$ implies that (y, x) is also in E.

Applications/Examples:

- Road networks

Flavors of Graphs

Directed vs. Undirected Graphs

A graph $G = (V, E)$ is *undirected* if edge $(x, y) \in E$ implies that (y, x) is also in E.

Applications/Examples:

- Road networks
  - Road networks *between* cities are undirected
  - Street networks *within* cities may be directed because of one-way streets.

Flavors of Graphs

Directed vs. Undirected Graphs

A graph $G = (V, E)$ is *undirected* if edge $(x, y) \in E$ implies that (y, x) is also in E.

Applications/Examples:

- Road networks
- Program Flow

Flavors of Graphs

Directed vs. Undirected Graphs

A graph $G = (V, E)$ is *undirected* if edge $(x, y) \in E$ implies that (y, x) is also in E.

Applications/Examples:

- Road networks
- Program Flow
  - Program-flow graphs are typically directed, because the execution flows from one line to the next and changes direction only at branches

Flavors of Graphs

Directed vs. Undirected Graphs

A graph $G = (V, E)$ is *undirected* if edge $(x, y) \in E$ implies that (y, x) is also in E.

Applications/Examples:

- Road networks
- Program Flow
- Friendship Graph
    - Is the friendship graph directed or undirected? If I am your friend, does that mean you are my friend?

Flavors of Graphs

Connected vs. Unconnected

Flavors of Graphs

Connected vs. Unconnected

An *undirected* graph is *connected* if there is a path between any two vertices.

Flavors of Graphs

Connected vs. Unconnected

An *undirected* graph is *connected* if there is a path between any two vertices.

A *directed* graph is *strongly connected* if there is a directed path between any two vertices.

Flavors of Graphs

Connected vs. Unconnected

An *undirected* graph is *connected* if there is a path between any two vertices.
A *directed* graph is *strongly connected* if there is a directed path between any two vertices.

Applications/Examples:

Flavors of Graphs

Connected vs. Unconnected

An *undirected* graph is *connected* if there is a path between any two vertices.
A *directed* graph is *strongly connected* if there is a directed path between any two vertices.

Applications/Examples:

- Friendship Graph

Flavors of Graphs

Connected vs. Unconnected

An *undirected* graph is *connected* if there is a path between any two vertices.
A *directed* graph is *strongly connected* if there is a directed path between any two vertices.

Applications/Examples:

- Friendship Graph
  - Is there a path of friends between any two people?

Flavors of Graphs

Weighted vs. Unweighted Graphs

Flavors of Graphs

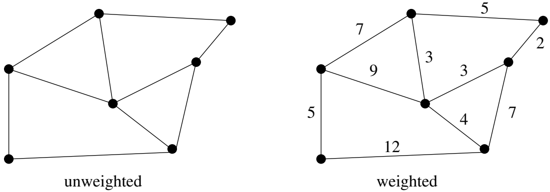
Weighted vs. Unweighted Graphs

In *weighted* graphs, each edge (or vertex) of G is assigned a numerical value, or weight.

Flavors of Graphs

Weighted vs. Unweighted Graphs

In *weighted* graphs, each edge (or vertex) of G is assigned a numerical value, or weight.



Flavors of Graphs

Weighted vs. Unweighted Graphs

In *weighted* graphs, each edge (or vertex) of G is assigned a numerical value, or weight.

Applications/Examples:

Flavors of Graphs

Weighted vs. Unweighted Graphs

In *weighted* graphs, each edge (or vertex) of G is assigned a numerical value, or weight.

Applications/Examples:

- Road Network

Flavors of Graphs

Weighted vs. Unweighted Graphs

In *weighted* graphs, each edge (or vertex) of G is assigned a numerical value, or weight.

Applications/Examples:

- Road Network
  - The edges of a road network graph might be weighted with their length, drive-time or speed limit.

Flavors of Graphs

Weighted vs. Unweighted Graphs

In *weighted* graphs, each edge (or vertex) of G is assigned a numerical value, or weight.

Applications/Examples:

- Road Network
- Friendship Graph

Flavors of Graphs

Weighted vs. Unweighted Graphs

In *weighted* graphs, each edge (or vertex) of G is assigned a numerical value, or weight.

Applications/Examples:

- Road Network
- Friendship Graph
  - We could model the strength of a friendship by associating each edge with an appropriate value

Flavors of Graphs

Simple vs. Non-simple Graphs

Flavors of Graphs

Simple vs. Non-simple Graphs

A *self-loop* is an edge (x, x).

Flavors of Graphs

Simple vs. Non-simple Graphs

A *self-loop* is an edge (x, x).

A *pseudograph* has self-loops.

Flavors of Graphs

Simple vs. Non-simple Graphs

A *self-loop* is an edge (x, x).

A *pseudograph* has self-loops.

An edge (x, y) is a *multi-edge* if it occurs more than once in the graph.

Flavors of Graphs

Simple vs. Non-simple Graphs

A *self-loop* is an edge (x, x).
A *pseudograph* has self-loops.
An edge (x, y) is a *multi-edge* if it occurs more than once in the graph.
A *multi-graph* has multi-edges or self-loops.

Flavors of Graphs

Simple vs. Non-simple Graphs

A *self-loop* is an edge (x, x).

A *pseudograph* has self-loops.

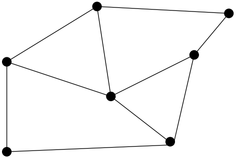
An edge (x, y) is a *multi-edge* if it occurs more than once in the graph.

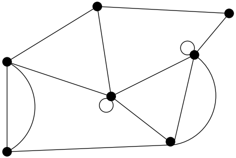
A *multi-graph* has multi-edges or self-loops.

A graph with no multi-edges or self-loops is called *simple*

Simple vs. Non-simple Graphs

A *self-loop* is an edge (x, x).
A *pseudograph* has self-loops.
An edge (x, y) is a *multi-edge* if it occurs more than once in the graph.
A *multi-graph* has multi-edges or self-loops.
A graph with no multi-edges or self-loops is called *simple*


simple


non−simple

Flavors of Graphs

Simple vs. Non-simple Graphs

A *self-loop* is an edge (x, x).
A *pseudograph* has self-loops.
An edge (x, y) is a *multi-edge* if it occurs more than once in the graph.
A *multi-graph* has multi-edges or self-loops.
A graph with no multi-edges or self-loops is called *simple*

Applications/Examples:

Flavors of Graphs

Simple vs. Non-simple Graphs

A *self-loop* is an edge (x, x).
A *pseudograph* has self-loops.
An edge (x, y) is a *multi-edge* if it occurs more than once in the graph.
A *multi-graph* has multi-edges or self-loops.
A graph with no multi-edges or self-loops is called *simple*

Applications/Examples:

- Road Network

Flavors of Graphs

Simple vs. Non-simple Graphs

A *self-loop* is an edge (x, x).
A *pseudograph* has self-loops.
An edge (x, y) is a *multi-edge* if it occurs more than once in the graph.
A *multi-graph* has multi-edges or self-loops.
A graph with no multi-edges or self-loops is called *simple*

Applications/Examples:

- Road Network
  - A loop road may result in a self-loop

Flavors of Graphs

Simple vs. Non-simple Graphs

A *self-loop* is an edge (x, x).
A *pseudograph* has self-loops.
An edge (x, y) is a *multi-edge* if it occurs more than once in the graph.
A *multi-graph* has multi-edges or self-loops.
A graph with no multi-edges or self-loops is called *simple*

Applications/Examples:

- Road Network
- Chemical Graphs

Flavors of Graphs

Simple vs. Non-simple Graphs

A *self-loop* is an edge (x, x).
A *pseudograph* has self-loops.
An edge (x, y) is a *multi-edge* if it occurs more than once in the graph.
A *multi-graph* has multi-edges or self-loops.
A graph with no multi-edges or self-loops is called *simple*

Applications/Examples:

- Road Network
- Chemical Graphs
  - A graph can be used to represent a chemical compound. There may be double bonds between atoms so the graphs may not be simple

Flavors of Graphs

Simple vs. Non-simple Graphs

A *self-loop* is an edge (x, x).
A *pseudograph* has self-loops.
An edge (x, y) is a *multi-edge* if it occurs more than once in the graph.
A *multi-graph* has multi-edges or self-loops.
A graph with no multi-edges or self-loops is called *simple*

Applications/Examples:

- Road Network
- Chemical Graphs
- Friendship Graph

Flavors of Graphs

Simple vs. Non-simple Graphs

A *self-loop* is an edge (x, x).
A *pseudograph* has self-loops.
An edge (x, y) is a *multi-edge* if it occurs more than once in the graph.
A *multi-graph* has multi-edges or self-loops.
A graph with no multi-edges or self-loops is called *simple*

Applications/Examples:

- Road Network
- Chemical Graphs
- Friendship Graph
  - Am I my own friend? *i.e., are there self-loops?*
  - Some people are friends through multiple connections which may be modeled by multi-edges

Flavors of Graphs

Sparse vs. Dense Graphs

Flavors of Graphs

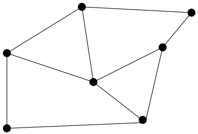
Sparse vs. Dense Graphs

Graphs are *sparse* when only a small fraction of the possible number of vertex pairs actually have edges defined between them.
Graph are *dense* when a large fraction of the possible number of vertex pairs actually have edges defined between them.

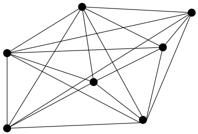
Flavors of Graphs

Sparse vs. Dense Graphs

Graphs are *sparse* when only a small fraction of the possible number of vertex pairs actually have edges defined between them.
Graph are *dense* when a large fraction of the possible number of vertex pairs actually have edges defined between them.



sparse

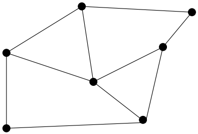


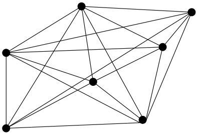
dense

Flavors of Graphs

Sparse vs. Dense Graphs

Graphs are *sparse* when only a small fraction of the possible number of vertex pairs actually have edges defined between them.
Graph are *dense* when a large fraction of the possible number of vertex pairs actually have edges defined between them.



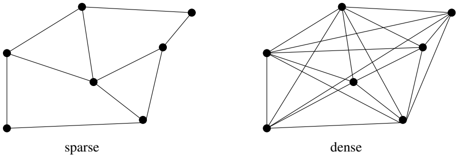


How many possible edges are there in a simple, undirected graph with n vertices?

Flavors of Graphs

Sparse vs. Dense Graphs

Graphs are *sparse* when only a small fraction of the possible number of vertex pairs actually have edges defined between them.
Graph are *dense* when a large fraction of the possible number of vertex pairs actually have edges defined between them.



How many possible edges are there in a simple, undirected graph with n vertices? $\binom{n}{2}$

Flavors of Graphs

Sparse vs. Dense Graphs

Graphs are *sparse* when only a small fraction of the possible number of vertex pairs actually have edges defined between them.
Graph are *dense* when a large fraction of the possible number of vertex pairs actually have edges defined between them.

The *degree* of a vertex is the number of edges adjacent to it.

Flavors of Graphs

Sparse vs. Dense Graphs

Graphs are *sparse* when only a small fraction of the possible number of vertex pairs actually have edges defined between them.
Graph are *dense* when a large fraction of the possible number of vertex pairs actually have edges defined between them.

The *degree* of a vertex is the number of edges adjacent to it.

- In *dense* graphs, most vertices have high degrees

Flavors of Graphs

Sparse vs. Dense Graphs

Graphs are *sparse* when only a small fraction of the possible number of vertex pairs actually have edges defined between them.
Graph are *dense* when a large fraction of the possible number of vertex pairs actually have edges defined between them.

The *degree* of a vertex is the number of edges adjacent to it.

- In *dense* graphs, most vertices have high degrees
- In *sparse* graphs, most vertices have low degrees

Flavors of Graphs

Sparse vs. Dense Graphs

Graphs are *sparse* when only a small fraction of the possible number of vertex pairs actually have edges defined between them.
Graph are *dense* when a large fraction of the possible number of vertex pairs actually have edges defined between them.

The *degree* of a vertex is the number of edges adjacent to it.

- In *dense* graphs, most vertices have high degrees
- In *sparse* graphs, most vertices have low degrees
- In *regular* graphs, all vertices have the same degrees

Flavors of Graphs

Sparse vs. Dense Graphs

Graphs are *sparse* when only a small fraction of the possible number of vertex pairs actually have edges defined between them.
Graph are *dense* when a large fraction of the possible number of vertex pairs actually have edges defined between them.

Applications/Examples:

- Road Network

Flavors of Graphs

Sparse vs. Dense Graphs

Graphs are *sparse* when only a small fraction of the possible number of vertex pairs actually have edges defined between them.
Graph are *dense* when a large fraction of the possible number of vertex pairs actually have edges defined between them.

Applications/Examples:

- Road Network
  - Road networks are sparse because of road junctions. Hard to have too many roads emerging from one intersection.

Flavors of Graphs

Sparse vs. Dense Graphs

Graphs are *sparse* when only a small fraction of the possible number of vertex pairs actually have edges defined between them.
Graph are *dense* when a large fraction of the possible number of vertex pairs actually have edges defined between them.

Applications/Examples:

- Road Network
- Chemical Graphs

Flavors of Graphs

Sparse vs. Dense Graphs

Graphs are *sparse* when only a small fraction of the possible number of vertex pairs actually have edges defined between them.
Graph are *dense* when a large fraction of the possible number of vertex pairs actually have edges defined between them.

Applications/Examples:

- Road Network
- Chemical Graphs
  - Chemical graphs are sparse because each atom has a bounded valence (limited number of potential neighbors)

Flavors of Graphs

Sparse vs. Dense Graphs

Graphs are *sparse* when only a small fraction of the possible number of vertex pairs actually have edges defined between them.
Graph are *dense* when a large fraction of the possible number of vertex pairs actually have edges defined between them.

Applications/Examples:

- Road Network
- Chemical Graphs
- Friendship Graph

Flavors of Graphs

Sparse vs. Dense Graphs

Graphs are *sparse* when only a small fraction of the possible number of vertex pairs actually have edges defined between them.
Graph are *dense* when a large fraction of the possible number of vertex pairs actually have edges defined between them.

Applications/Examples:

- Road Network
- Chemical Graphs
- Friendship Graph
  - Even the most gregarious person only knows an insignificant fraction of everyone on earth.
  - Vertex Degrees: Who has the most friends?
  - Cliques: What is the largest clique?

Flavors of Graphs

Cyclic vs. Acyclic Graphs

Flavors of Graphs

Cyclic vs. Acyclic Graphs

An *acyclic* graph does not contain any simple cycles.

Cyclic vs. Acyclic Graphs

An *acyclic* graph does not contain any simple cycles.

A *cycle* is a path where the last vertex is adjacent to the first.

Flavors of Graphs

Cyclic vs. Acyclic Graphs

An *acyclic* graph does not contain any simple cycles.

A cycle in which no vertex repeats (such as 1-2-3-1 versus 1-2-3-2-1) is said to be *simple*.

Flavors of Graphs

Cyclic vs. Acyclic Graphs

An *acyclic* graph does not contain any simple cycles.

Connected undirected acyclic graphs are called *trees*

Flavors of Graphs

Cyclic vs. Acyclic Graphs

An *acyclic* graph does not contain any simple cycles.

Directed acyclic graphs are called *DAGs*

Flavors of Graphs

Cyclic vs. Acyclic Graphs

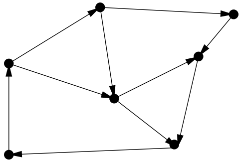
An *acyclic* graph does not contain any simple cycles.

The shortest cycle in the graph defines its *girth*, while a simple cycle which passes through each vertex is said to be a *Hamiltonian cycle*

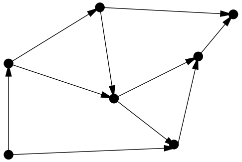
Flavors of Graphs

Cyclic vs. Acyclic Graphs

An *acyclic* graph does not contain any simple cycles.



cyclic



acyclic

Flavors of Graphs

Cyclic vs. Acyclic Graphs

An *acyclic* graph does not contain any simple cycles.

Applications/Examples:

- Street Network

Flavors of Graphs

Cyclic vs. Acyclic Graphs

An *acyclic* graph does not contain any simple cycles.

Applications/Examples:

- Street Network
    - A street network is likely a cyclic graph - you can eventually return to the place where you started

Flavors of Graphs

Cyclic vs. Acyclic Graphs

An *acyclic* graph does not contain any simple cycles.

Applications/Examples:

- Street Network
- Scheduling Problems

Flavors of Graphs

Cyclic vs. Acyclic Graphs

An *acyclic* graph does not contain any simple cycles.

Applications/Examples:

- Street Network
- Scheduling Problems
  - DAGs arise naturally in scheduling problems, where a directed edge (x, y) indicates that x must occur before y.

Cyclic vs. Acyclic Graphs

An *acyclic* graph does not contain any simple cycles.

Applications/Examples:

- Street Network
- Scheduling Problems
- Friendship Graph

Flavors of Graphs

Cyclic vs. Acyclic Graphs

An *acyclic* graph does not contain any simple cycles.

Applications/Examples:

- Street Network
- Scheduling Problems
- Friendship Graph
  - How long will it take for my gossip to get back to me?

Flavors of Graphs

Embedded vs. Topological Graphs

Flavors of Graphs

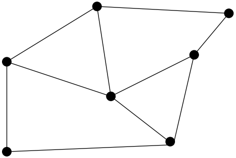
Embedded vs. Topological Graphs

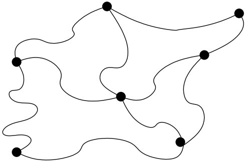
A graph is *embedded* if the vertices and edges have been assigned geometric positions.

Flavors of Graphs

Embedded vs. Topological Graphs

A graph is *embedded* if the vertices and edges have been assigned geometric positions.


embedded


topological

Flavors of Graphs

Embedded vs. Topological Graphs

A graph is *embedded* if the vertices and edges have been assigned geometric positions.

Applications/Examples:

- Traveling Salesman Problem
    - TSP or Shortest path on points in the plane.

Flavors of Graphs

Embedded vs. Topological Graphs

A graph is *embedded* if the vertices and edges have been assigned geometric positions.

Applications/Examples:

- Traveling Salesman Problem
- Friendship Graph

Flavors of Graphs

Embedded vs. Topological Graphs

A graph is *embedded* if the vertices and edges have been assigned geometric positions.

Applications/Examples:

- Traveling Salesman Problem
- Friendship Graph
    - A full understanding of social networks requires an *embedded graph* where each vertex is associated with the point on this world where they live.

Flavors of Graphs

Implicit vs. Explicit Graphs

Flavors of Graphs

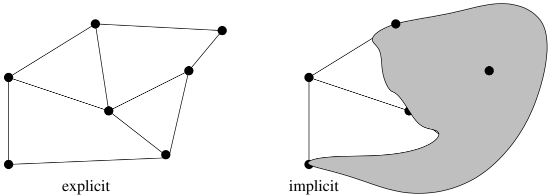
Implicit vs. Explicit Graphs

Many graphs are not explicitly constructed and then traversed, but built as we use them.

Flavors of Graphs

Implicit vs. Explicit Graphs

Many graphs are not explicitly constructed and then traversed, but built as we use them.



Flavors of Graphs

Implicit vs. Explicit Graphs

Many graphs are not explicitly constructed and then traversed, but built as we use them.

Applications/Examples:

- A good example arises in backtrack search.

Flavors of Graphs

Implicit vs. Explicit Graphs

Many graphs are not explicitly constructed and then traversed, but built as we use them.

Applications/Examples:

- A good example arises in backtrack search.
- Friendship Graph

Flavors of Graphs

Implicit vs. Explicit Graphs

Many graphs are not explicitly constructed and then traversed, but built as we use them.

Applications/Examples:

- A good example arises in backtrack search.
- Friendship Graph
  - Social networking services are built on the premise of *explicitly* defining links between their member-friends.
  - The complete (world-wide) friendship graph is represented *implicitly*.

Flavors of Graphs

Labeled vs. Unlabeled Graphs

Flavors of Graphs

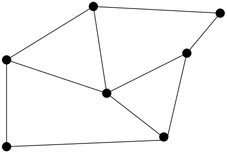
Labeled vs. Unlabeled Graphs

In *labeled* graphs, each vertex is assigned a unique identifier to distinguish it from all other vertices.
In *unlabeled* graphs, no such distinctions have been made.

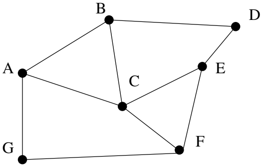
Flavors of Graphs

Labeled vs. Unlabeled Graphs

In *labeled* graphs, each vertex is assigned a unique identifier to distinguish it from all other vertices.
In *unlabeled* graphs, no such distinctions have been made.



unlabeled



labeled

Flavors of Graphs

Labeled vs. Unlabeled Graphs

In *labeled* graphs, each vertex is assigned a unique identifier to distinguish it from all other vertices.
In *unlabeled* graphs, no such distinctions have been made.

Applications/Examples:

- Isomorphism Testing

Flavors of Graphs

Labeled vs. Unlabeled Graphs

In *labeled* graphs, each vertex is assigned a unique identifier to distinguish it from all other vertices.
In *unlabeled* graphs, no such distinctions have been made.

Applications/Examples:

- Isomorphism Testing
    - An important graph problem is *isomorphism testing*, determining whether the topological structure of two graphs are in fact identical if we ignore any labels.

Flavors of Graphs

Labeled vs. Unlabeled Graphs

In *labeled* graphs, each vertex is assigned a unique identifier to distinguish it from all other vertices.
In *unlabeled* graphs, no such distinctions have been made.

Applications/Examples:

- Isomorphism Testing
- Road Networks

Flavors of Graphs

Labeled vs. Unlabeled Graphs

In *labeled* graphs, each vertex is assigned a unique identifier to distinguish it from all other vertices.
In *unlabeled* graphs, no such distinctions have been made.

Applications/Examples:

- Isomorphism Testing
- Road Networks
  - Road networks between cities are labeled. Each node is labeled by a city name.

Flavors of Graphs

Labeled vs. Unlabeled Graphs

In *labeled* graphs, each vertex is assigned a unique identifier to distinguish it from all other vertices.
In *unlabeled* graphs, no such distinctions have been made.

Applications/Examples:

- Isomorphism Testing
- Road Networks
- Chemical Graphs

Flavors of Graphs

Labeled vs. Unlabeled Graphs

In *labeled* graphs, each vertex is assigned a unique identifier to distinguish it from all other vertices.
In *unlabeled* graphs, no such distinctions have been made.

Applications/Examples:

- Isomorphism Testing
- Road Networks
- Chemical Graphs
  - Chemical graphs are labeled by their atom type. *Note that the labels in this case are not unique*

Flavors of Graphs

Labeled vs. Unlabeled Graphs

In *labeled* graphs, each vertex is assigned a unique identifier to distinguish it from all other vertices.
In *unlabeled* graphs, no such distinctions have been made.

Applications/Examples:

- Isomorphism Testing
- Road Networks
- Chemical Graphs
- Friendship Graph

Flavors of Graphs

Labeled vs. Unlabeled Graphs

In *labeled* graphs, each vertex is assigned a unique identifier to distinguish it from all other vertices.
In *unlabeled* graphs, no such distinctions have been made.

Applications/Examples:

- Isomorphism Testing
- Road Networks
- Chemical Graphs
- Friendship Graph
  - Does each vertex have a name/label which reflects its identity, and is this label important for our analysis?
  - Much of the study of social networks is unconcerned with labels on graphs.

Data Structures for Graphs

There are two main data structures used to represent graphs:

We assume the graph $G = (V, E)$ contains n vertices and m edges.

Data Structures for Graphs

There are two main data structures used to represent graphs:

1. adjacency matrices

We assume the graph $G = (V, E)$ contains n vertices and m edges.

Data Structures for Graphs

There are two main data structures used to represent graphs:

1. adjacency matrices
2. adjacency lists

We assume the graph $G = (V, E)$ contains n vertices and m edges.

Adjacency Matrices

Adjacency Matrices

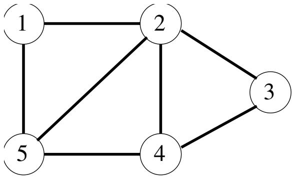
Definition

An *adjacency matrix* represents graph G using an $n \times n$ matrix, M, where element $M[i, j]$ is 1 if (i, j) is an edge of G, and 0 if it is not an edge of G.

Adjacency Matrices

Definition

An *adjacency matrix* represents graph G using an $n \times n$ matrix, M, where element $M[i, j]$ is 1 if (i, j) is an edge of G, and 0 if it is not an edge of G.



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Adjacency Matrices

Definition

An *adjacency matrix* represents graph G using an $n \times n$ matrix, M, where element $M[i, j]$ is 1 if (i, j) is an edge of G, and 0 if it is not an edge of G.

How much space does this data structure take if:

- There are 0 edges on n nodes?

Adjacency Matrices

Definition

An *adjacency matrix* represents graph G using an $n \times n$ matrix, M, where element $M[i,j]$ is 1 if (i,j) is an edge of G, and 0 if it is not an edge of G.

How much space does this data structure take if:

- There are 0 edges on n nodes? $O(n^2)$

Adjacency Matrices

Definition

An *adjacency matrix* represents graph G using an $n \times n$ matrix, M, where element $M[i,j]$ is 1 if (i,j) is an edge of G, and 0 if it is not an edge of G.

How much space does this data structure take if:

- There are 0 edges on n nodes? $O(n^2)$
- There are n edges on n nodes?

Adjacency Matrices

Definition

An *adjacency matrix* represents graph G using an $n \times n$ matrix, M, where element $M[i, j]$ is 1 if (i, j) is an edge of G, and 0 if it is not an edge of G.

How much space does this data structure take if:

- There are 0 edges on n nodes? $O(n^2)$
- There are n edges on n nodes? $O(n^2)$

Adjacency Matrices

Definition

An *adjacency matrix* represents graph G using an $n \times n$ matrix, M, where element $M[i,j]$ is 1 if (i,j) is an edge of G, and 0 if it is not an edge of G.

How much space does this data structure take if:

- There are 0 edges on n nodes? $O(n^2)$
- There are n edges on n nodes? $O(n^2)$
- There are $\frac{n(n-1)}{2}$ edges on n nodes?

Adjacency Matrices

Definition

An *adjacency matrix* represents graph G using an $n \times n$ matrix, M, where element $M[i,j]$ is 1 if (i,j) is an edge of G, and 0 if it is not an edge of G.

How much space does this data structure take if:

- There are 0 edges on n nodes? $O(n^2)$
- There are n edges on n nodes? $O(n^2)$
- There are $\frac{n(n-1)}{2}$ edges on n nodes? $O(n^2)$

Adjacency Matrices

Definition

An *adjacency matrix* represents graph G using an $n \times n$ matrix, M, where element $M[i,j]$ is 1 if (i,j) is an edge of G, and 0 if it is not an edge of G.

How much space does this data structure take if:

- There are 0 edges on n nodes? $O(n^2)$
- There are n edges on n nodes? $O(n^2)$
- There are $\frac{n(n-1)}{2}$ edges on n nodes? $O(n^2)$

It uses excessive space for graphs with many vertices and relatively few edges.

Adjacency Matrices

Definition

An *adjacency matrix* represents graph G using an $n \times n$ matrix, M, where element $M[i,j]$ is 1 if (i,j) is an edge of G, and 0 if it is not an edge of G.

Advantages

Disadvantages

Adjacency Matrices

Definition

An *adjacency matrix* represents graph G using an $n \times n$ matrix, M, where element $M[i, j]$ is 1 if (i, j) is an edge of G, and 0 if it is not an edge of G.

Advantages

- Quickly determine if edge (i, j) is in G.

Disadvantages

Adjacency Matrices

Definition

An *adjacency matrix* represents graph G using an $n \times n$ matrix, M, where element $M[i,j]$ is 1 if (i,j) is an edge of G, and 0 if it is not an edge of G.

Advantages

- Quickly determine if edge (i,j) is in G.
- Rapid updates for edge insertion and deletion

Disadvantages

Adjacency Matrices

Definition

An *adjacency matrix* represents graph G using an $n \times n$ matrix, M, where element $M[i,j]$ is 1 if (i,j) is an edge of G, and 0 if it is not an edge of G.

Advantages

- Quickly determine if edge (i,j) is in G.
- Rapid updates for edge insertion and deletion
- Efficiently store a dense graph. Max edges there are $O(n^2)$ storage to store $\frac{n(n-1)}{2}$ edges

Disadvantages

Adjacency Matrices

Definition

An *adjacency matrix* represents graph G using an $n \times n$ matrix, M, where element $M[i,j]$ is 1 if (i,j) is an edge of G, and 0 if it is not an edge of G.

Advantages

- Quickly determine if edge (i,j) is in G.
- Rapid updates for edge insertion and deletion
- Efficiently store a dense graph. Max edges there are $O(n^2)$ storage to store $\frac{n(n-1)}{2}$ edges

Disadvantages

- May use excessive space for graphs with many vertices and relatively few edges

Adjacency Lists

Adjacency Lists

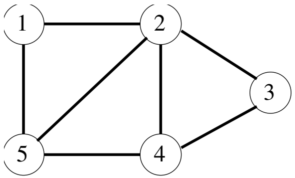
Definition

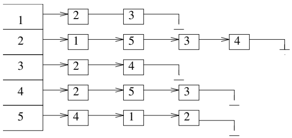
An *adjacency list* consists of an array of n pointers, where the ith element points to a linked list of the edges incident on vertex i.

Adjacency Lists

Definition

An *adjacency list* consists of an array of n pointers, where the ith element points to a linked list of the edges incident on vertex i.





Adjacency Lists

Definition

An *adjacency list* consists of an array of n pointers, where the ith element points to a linked list of the edges incident on vertex i.

Advantages

Disadvantages

Adjacency Lists

Definition

An *adjacency list* consists of an array of n pointers, where the ith element points to a linked list of the edges incident on vertex i.

Advantages

- More efficient for storing sparse graphs.

Disadvantages

Adjacency Lists

Definition

An *adjacency list* consists of an array of n pointers, where the ith element points to a linked list of the edges incident on vertex i.

Advantages

- More efficient for storing sparse graphs.

Disadvantages

- Requires pointers

Adjacency Lists

Definition

An *adjacency list* consists of an array of n pointers, where the ith element points to a linked list of the edges incident on vertex i.

Advantages

- More efficient for storing sparse graphs.

Disadvantages

- Requires pointers
- Harder to verify whether a given edge (i, j) is in G, since we must search through the appropriate list to find the edge. It takes $O(d_i)$ time where d_i is the degree of the i^{th} vertex. Note that d_i is much less than n when the graph is sparse.

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
Faster to find vertex degree of vertex i	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
Faster to find vertex degree of vertex i	Matrices $= O(n)$ Lists $= O(d_i)$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
Faster to find vertex degree of vertex i	Matrices $= O(n)$ Lists $= O(d_i)$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
Faster to find vertex degree of vertex i	Matrices $= O(n)$ Lists $= O(d_i)$
Less memory on sparse graphs	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$

Comparison of Data Structures

Comparison	Winner
Faster to test if (i,j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
Faster to find vertex degree of vertex i	Matrices $= O(n)$ Lists $= O(d_i)$
Less memory on sparse graphs	Matrices $= O(n^2)$ Lists $= O(m+n)$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
Faster to find vertex degree of vertex i	Matrices $= O(n)$ Lists $= O(d_i)$
Less memory on sparse graphs	Matrices $= O(n^2)$ Lists $= O(m+n)$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$

Example

Consider a road network that is 20 avenues wide $\times$ 200 avenues long.

Example

Consider a road network that is 20 avenues wide $\times$ 200 avenues long.

There are $\sim 20 \times 200 = 4000$ vertices in the resulting graph.

Example

Consider a road network that is 20 avenues wide $\times$ 200 avenues long.

There are $\sim 20 \times 200 = 4000$ vertices in the resulting graph.

How much space does it take to represent this network as a graph using an Adjacency Matrix?

Example

Consider a road network that is 20 avenues wide $\times$ 200 avenues long.

There are $\sim 20 \times 200 = 4000$ vertices in the resulting graph.

How much space does it take to represent this network as a graph using an Adjacency Matrix?

4000×4000 matrix

Assume each entry is 1 byte

Example

Consider a road network that is 20 avenues wide $\times$ 200 avenues long.

There are $\sim 20 \times 200 = 4000$ vertices in the resulting graph.

How much space does it take to represent this network as a graph using an Adjacency Matrix?

4000×4000 matrix
$\Rightarrow$ 16,000,000 bytes $=$ 16MB

Assume each entry is 1 byte

Example

Consider a road network that is 20 avenues wide $\times$ 200 avenues long.

There are $\sim 20 \times 200 = 4000$ vertices in the resulting graph.

How much space does it take to represent this network as a graph using an Adjacency Matrix?

4000×4000 matrix

$\Rightarrow$ 16,000,000 bytes = 16MB

How much space does it take to represent this network as a graph using an Adjacency List?

Assume each entry is 1 byte

Example

Consider a road network that is 20 avenues wide $\times$ 200 avenues long.

There are $\sim 20 \times 200 = 4000$ vertices in the resulting graph.

How much space does it take to represent this network as a graph using an Adjacency Matrix?

4000×4000 matrix

$\Rightarrow$ 16,000,000 bytes = 16MB

How much space does it take to represent this network as a graph using an Adjacency List?

4000 vertices with an average degree of 4.

Assume each entry is 1 byte

Example

Consider a road network that is 20 avenues wide $\times$ 200 avenues long.

There are $\sim 20 \times 200 = 4000$ vertices in the resulting graph.

How much space does it take to represent this network as a graph using an Adjacency Matrix?

4000×4000 matrix
$\Rightarrow$ 16,000,000 bytes = 16MB

How much space does it take to represent this network as a graph using an Adjacency List?

4000 vertices with an average degree of 4.
$\Rightarrow 4000 \times 4 =$16,000 bytes = 16KB

Assume each entry is 1 byte

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
Faster to find vertex degree of vertex i	Matrices $= O(n)$ Lists $= O(d_i)$
Less memory on sparse graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
Faster to find vertex degree of vertex i	Matrices $= O(n)$ Lists $= O(d_i)$
Less memory on sparse graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
Less memory on dense graphs	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
Faster to find vertex degree of vertex i	Matrices $= O(n)$ Lists $= O(d_i)$
Less memory on sparse graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
Less memory on dense graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
Faster to find vertex degree of vertex i	Matrices $= O(n)$ Lists $= O(d_i)$
Less memory on sparse graphs	Matrices $= O(n^2)$ Lists $= O(m+n)$
Less memory on dense graphs	Matrices $= O(n^2)$ Lists $= O(m+n)$
	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
Faster to find vertex degree of vertex i	Matrices $= O(n)$ Lists $= O(d_i)$
Less memory on sparse graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
Less memory on dense graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
Edge (i, j) insertion or deletion	Matrices $=$ Lists $=$
	Matrices $=$ Lists $=$

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
Faster to find vertex degree of vertex i	Matrices $= O(n)$ Lists $= O(d_i)$
Less memory on sparse graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
Less memory on dense graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
Edge (i, j) insertion or deletion	Matrices $= O(1)$ Lists $= O(d_i + d_j)$
	Matrices $=$ Lists $=$

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
Faster to find vertex degree of vertex i	Matrices $= O(n)$ Lists $= O(d_i)$
Less memory on sparse graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
Less memory on dense graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
Edge (i, j) insertion or deletion	Matrices $= O(1)$ Lists $= O(d_i + d_j)$
	Matrices $=$ Lists $=$

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
Faster to find vertex degree of vertex i	Matrices $= O(n)$ Lists $= O(d_i)$
Less memory on sparse graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
Less memory on dense graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
Edge (i, j) insertion or deletion	Matrices $= O(1)$ Lists $= O(d_i + d_j)$
Faster graph traversal	Matrices $=$ Lists $=$

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
Faster to find vertex degree of vertex i	Matrices $= O(n)$ Lists $= O(d_i)$
Less memory on sparse graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
Less memory on dense graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
Edge (i, j) insertion or deletion	Matrices $= O(1)$ Lists $= O(d_i + d_j)$
Faster graph traversal	Matrices $= O(n^2)$ Lists $= O(m + n)$

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
Faster to find vertex degree of vertex i	Matrices $= O(n)$ Lists $= O(d_i)$
Less memory on sparse graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
Less memory on dense graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
Edge (i, j) insertion or deletion	Matrices $= O(1)$ Lists $= O(d_i + d_j)$
Faster graph traversal	Matrices $= O(n^2)$ Lists $= O(m + n)$

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
Faster to find vertex degree of vertex i	Matrices $= O(n)$ Lists $= O(d_i)$
Less memory on sparse graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
Less memory on dense graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
Edge (i, j) insertion or deletion	Matrices $= O(1)$ Lists $= O(d_i + d_j)$
Faster graph traversal	Matrices $= O(n^2)$ Lists $= O(m + n)$
Better for most problems?	

Comparison of Data Structures

Comparison	Winner
Faster to test if (i, j) exists	Matrices $= O(1)$ Lists $= O(d_i)$
Faster to find vertex degree of vertex i	Matrices $= O(n)$ Lists $= O(d_i)$
Less memory on sparse graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
Less memory on dense graphs	Matrices $= O(n^2)$ Lists $= O(m + n)$
Edge (i, j) insertion or deletion	Matrices $= O(1)$ Lists $= O(d_i + d_j)$
Faster graph traversal	Matrices $= O(n^2)$ Lists $= O(m + n)$
Better for most problems?	Lists

Traversing a Graph

Definition

A *graph traversal* is a walk-through of a graph.

Applications

Traversing a Graph

Definition

A *graph traversal* is a walk-through of a graph.

Applications

- Printing a graph
- Copying a graph
- Converting between graph representations
- Finding a path in a maze

Traversing a Graph

Definition

A *graph traversal* is a walk-through of a graph.

Properties of a good graph traversal algorithm

Definition

A *graph traversal* is a walk-through of a graph.

Properties of a good graph traversal algorithm

1. Efficient
2. Correct

Traversing a Graph

Definition

A *graph traversal* is a walk-through of a graph.

Properties of a good graph traversal algorithm

1. Efficient
   Visit each edge at most twice (once coming and once going)
2. Correct

Traversing a Graph

Definition

A *graph traversal* is a walk-through of a graph.

Properties of a good graph traversal algorithm

1. Efficient
   Visit each edge at most twice (once coming and once going)
2. Correct
   Perform the traversal in a systematic way so that we don't miss anything (i.e., visit every edge and vertex).

Key Idea

The key idea is that we must mark each vertex when we first visit it, and keep track of what have not yet completely explored.

Key Idea

The key idea is that we must mark each vertex when we first visit it, and keep track of what have not yet completely explored.

Three States of a Vertex

Key Idea

The key idea is that we must mark each vertex when we first visit it, and keep track of what have not yet completely explored.

Three States of a Vertex

1. Undiscovered

Key Idea

The key idea is that we must mark each vertex when we first visit it, and keep track of what have not yet completely explored.

Three States of a Vertex

1. Undiscovered
   The vertex in its initial state.

Key Idea

The key idea is that we must mark each vertex when we first visit it, and keep track of what have not yet completely explored.

Three States of a Vertex

1. Undiscovered

   The vertex in its initial state.

2. Discovered

Key Idea

The key idea is that we must mark each vertex when we first visit it, and keep track of what have not yet completely explored.

Three States of a Vertex

1. Undiscovered
   The vertex in its initial state.
2. Discovered
   The vertex after we have encountered it, but before we have checked out all its incident edges.

Key Idea

The key idea is that we must mark each vertex when we first visit it, and keep track of what have not yet completely explored.

Three States of a Vertex

1. Undiscovered
   The vertex in its initial state.
2. Discovered
   The vertex after we have encountered it, but before we have checked out all its incident edges.
3. Processed

Key Idea

The key idea is that we must mark each vertex when we first visit it, and keep track of what have not yet completely explored.

Three States of a Vertex

1. Undiscovered
   The vertex in its initial state.
2. Discovered
   The vertex after we have encountered it, but before we have checked out all its incident edges.
3. Processed
   The vertex after we have visited all its incident edges.

To Do List

- Need to maintain a structure containing all the vertices we have discovered but not yet completely explored (i.e., processed).

To Do List

- Need to maintain a structure containing all the vertices we have discovered but not yet completely explored (i.e., processed).
- Initially, only a single start vertex is considered to be discovered.

To Do List

- Need to maintain a structure containing all the vertices we have discovered but not yet completely explored (i.e., processed).
- Initially, only a single start vertex is considered to be discovered.
- To completely explore a vertex, look at each edge going out of it. For each edge to an undiscovered vertex, mark it discovered and add it to the structure.

To Do List

- Need to maintain a structure containing all the vertices we have discovered but not yet completely explored (i.e., processed).
- Initially, only a single start vertex is considered to be discovered.
- To completely explore a vertex, look at each edge going out of it. For each edge to an undiscovered vertex, mark it discovered and add it to the structure.
- Each edge is considered exactly twice, when each of its endpoints are explored.

Correctness of Graph Traversal

Every edge and vertex in the connected component is eventually visited.

Correctness of Graph Traversal

Every edge and vertex in the connected component is eventually visited.

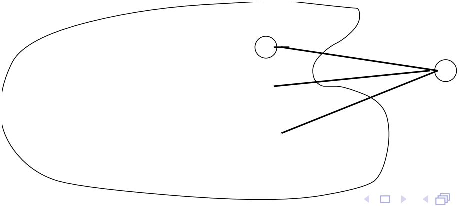
Proof by Contradiction

Correctness of Graph Traversal

Every edge and vertex in the connected component is eventually visited.

Proof by Contradiction

Assume that there exists a vertex v which remains undiscovered whose neighbor u was discovered.



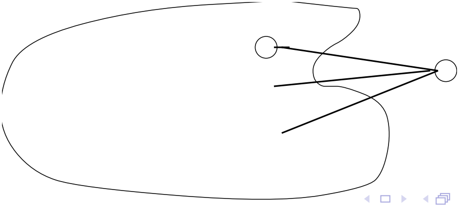
Correctness of Graph Traversal

Every edge and vertex in the connected component is eventually visited.

Proof by Contradiction

Assume that there exists a vertex v which remains undiscovered whose neighbor u was discovered.

- If the vertex u was discovered then it was added to the To-Do list.



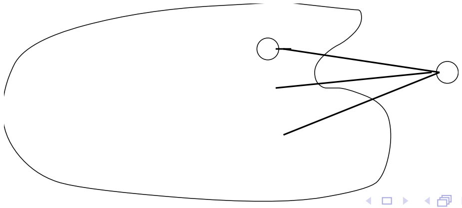
Correctness of Graph Traversal

Every edge and vertex in the connected component is eventually visited.

Proof by Contradiction

Assume that there exists a vertex v which remains undiscovered whose neighbor u was discovered.

- If the vertex u was discovered then it was added to the To-Do list.
- This means that vertex u will eventually be explored/processed.



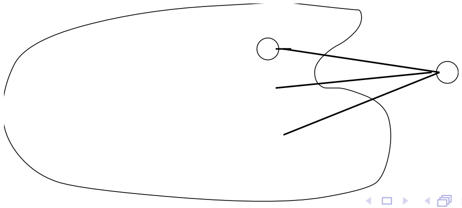
Correctness of Graph Traversal

Every edge and vertex in the connected component is eventually visited.

Proof by Contradiction

Assume that there exists a vertex v which remains undiscovered whose neighbor u was discovered.

- If the vertex u was discovered then it was added to the To-Do list.
- This means that vertex u will eventually be explored/processed.
- When the vertex u is explored/processed then it will visit (discover) vertex v and add vertex v to the to-do list which means that vertex v will eventually be explored/processed.



To Do List

- Need to maintain a structure containing all the vertices we have discovered but not yet completely explored (i.e., processed).
- Initially, only a single start vertex is considered to be discovered.
- To completely explore a vertex, look at each edge going out of it. For each edge to an undiscovered vertex, mark it discovered and add it to the structure.
- Each edge is considered exactly twice, when each of its endpoints are explored.

Primary Graph Traversal Algorithms

To Do List

- Need to maintain a structure containing all the vertices we have discovered but not yet completely explored (i.e., processed).
- Initially, only a single start vertex is considered to be discovered.
- To completely explore a vertex, look at each edge going out of it. For each edge to an undiscovered vertex, mark it discovered and add it to the structure.
- Each edge is considered exactly twice, when each of its endpoints are explored.

Primary Graph Traversal Algorithms

1. Breadth-First Traversal
2. Depth-First Traversal

To Do List

- Need to maintain a structure containing all the vertices we have discovered but not yet completely explored (i.e., processed).
- Initially, only a single start vertex is considered to be discovered.
- To completely explore a vertex, look at each edge going out of it. For each edge to an undiscovered vertex, mark it discovered and add it to the structure.
- Each edge is considered exactly twice, when each of its endpoints are explored.

Primary Graph Traversal Algorithms

1. Breadth-First Traversal (traversal using a queue)
2. Depth-First Traversal

To Do List

- Need to maintain a structure containing all the vertices we have discovered but not yet completely explored (i.e., processed).
- Initially, only a single start vertex is considered to be discovered.
- To completely explore a vertex, look at each edge going out of it. For each edge to an undiscovered vertex, mark it discovered and add it to the structure.
- Each edge is considered exactly twice, when each of its endpoints are explored.

Primary Graph Traversal Algorithms

1. Breadth-First Traversal (traversal using a queue)
2. Depth-First Traversal (traversal using a stack)

To Do List

- Need to maintain a structure containing all the vertices we have discovered but not yet completely explored (i.e., processed).
- Initially, only a single start vertex is considered to be discovered.
- To completely explore a vertex, look at each edge going out of it. For each edge to an undiscovered vertex, mark it discovered and add it to the structure.
- Each edge is considered exactly twice, when each of its endpoints are explored.

Primary Graph Traversal Algorithms

1. Breadth-First Traversal (traversal using a queue)
2. Depth-First Traversal (traversal using a stack)

For certain problems, it makes absolutely no difference which one you use, but in other cases the distinction is crucial.

Breadth-First Traversal

Main Idea

- First explore adjacent vertices. Then all vertices adjacent to just explored. Then all vertices adjacent to those.
- We explore based on the distance from our starting point. The search systematically radiates out.
- Useful in shortest path in unweighted graphs

Breadth-First Traversal

Main Idea

- First explore adjacent vertices. Then all vertices adjacent to just explored. Then all vertices adjacent to those.
- We explore based on the distance from our starting point. The search systematically radiates out.
- Useful in shortest path in unweighted graphs

By-Products of BFS

1. Breadth First Tree
2. Shortest path from start vertex s to each vertex x in G.

Information associated with each node u

- color$[u]$
- $d[u]$
- $parent[u]$

Information associated with each node u

- color[u]
  WHITE $\Rightarrow$ u is undiscovered.
  GRAY $\Rightarrow$ u is discovered.
  BLACK $\Rightarrow$ u has been explored.
- $d[u]$
- $parent[u]$

Information associated with each node u

- color[u]
  WHITE $\Rightarrow$ u is undiscovered.
  GRAY $\Rightarrow$ u is discovered.
  BLACK $\Rightarrow$ u has been explored.
- $d[u]$
  distance from s to u.
- $parent[u]$

Information associated with each node u

- color[u]
  WHITE $\Rightarrow$ u is undiscovered.
  GRAY $\Rightarrow$ u is discovered.
  BLACK $\Rightarrow$ u has been explored.
- $d[u]$
  distance from s to u.
- $parent[u]$
  u's parent in Breadth First tree.

BFS Algorithm

Algorithm:
`breadthFirstSearch`(V, E, s)

BFS Algorithm

Initialize the Vertices

Algorithm:
`breadthFirstSearch`(V, E, s)

1 **for** *each vertex* $u \in (V - \{S\})$ **do**

BFS Algorithm

Initialize the Vertices

- Vertex is undiscovered

Algorithm:
`breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE

BFS Algorithm

Initialize the Vertices

- Vertex is undiscovered
- There parent in the BFS tree is unknown

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
```

BFS Algorithm

Initialize the Vertices

- Vertex is undiscovered
- There parent in the BFS tree is unknown
- We do not know the shortest path to the vertex yet

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$

BFS Algorithm

The Start Vertex

- Is discovered

Algorithm:
`breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
   $color[u] =$WHITE
   $parent[u] = nil$
   $d[u] = \infty$
$color[s] =$GRAY

BFS Algorithm

The Start Vertex

- Is discovered
- Has no parent in the BFS tree

Algorithm: breadthFirstSearch(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$

BFS Algorithm

The Start Vertex

- Is discovered
- Has no parent in the BFS tree
- The shortest path from s to s is 0

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$

BFS Algorithm

The To-Do List

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u] =$WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s] =$GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q

BFS Algorithm

The To-Do List

- Add the start vertex to the To-Do List

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
`enqueue`(Q, s)

BFS Algorithm

Process vertices on the To-Do List

- Look at each vertex on the list

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u] =$WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s] =$GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
`enqueue`(Q, s)
while $Q \neq \emptyset$ **do**

BFS Algorithm

Process vertices on the To-Do List

- Look at each vertex on the list
- Remove the next vertex, u, from the queue

Algorithm: breadthFirstSearch(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    $u =$ dequeue(Q)

BFS Algorithm

Process vertices on the To-Do List

- Look at each vertex on the list
- Remove the next vertex, u, from the queue
- Look at each neighbor, v

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    u = dequeue(Q)
    for *each v adjacent to u* **do**

BFS Algorithm

Process vertices on the To-Do List

- Look at each vertex on the list
- Remove the next vertex, u, from the queue
- Look at each neighbor, v
  If v is undiscovered

Algorithm: breadthFirstSearch(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    $u =$ dequeue(Q)
    for *each* v *adjacent to* u **do**
        if $color[v] == WHITE$ **then**

BFS Algorithm

Process vertices on the To-Do List

- Look at each vertex on the list
- Remove the next vertex, u, from the queue
- Look at each neighbor, v
  If v is undiscovered
  - v is now discovered

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    u = dequeue(Q)
    for *each* v *adjacent to* u **do**
        if $color[v] == WHITE$ **then**
            $color[v]$ =GRAY

BFS Algorithm

Process vertices on the To-Do List

- Look at each vertex on the list
- Remove the next vertex, u, from the queue
- Look at each neighbor, v
  If v is undiscovered
  - v is now discovered
  - The distance to v is the distance to u plus 1

Algorithm: breadthFirstSearch(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u] =$WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s] =$GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  $u =$ dequeue(Q)
  for *each* v *adjacent to* u **do**
    if $color[v] == WHITE$ **then**
      $color[v] =$GRAY
      $d[v] = d[u] + 1$

BFS Algorithm

Process vertices on the To-Do List

- Look at each vertex on the list
- Remove the next vertex, u, from the queue
- Look at each neighbor, v
  If v is undiscovered
  - v is now discovered
  - The distance to v is the distance to u plus 1
  - The parent of v is u

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    u = dequeue(Q)
    for *each* v *adjacent to* u **do**
        if $color[v] == WHITE$ **then**
            $color[v]$ =GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$

BFS Algorithm

Process vertices on the To-Do List

- Look at each vertex on the list
- Remove the next vertex, u, from the queue
- Look at each neighbor, v
  If v is undiscovered
  - v is now discovered
  - The distance to v is the distance to u plus 1
  - The parent of v is u
  - Add v to the To-Do List

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
```

BFS Algorithm

Process vertices on the To-Do List

- Look at each vertex on the list
- Remove the next vertex, u, from the queue
- Look at each neighbor, v
  If v is undiscovered
  - v is now discovered
  - The distance to v is the distance to u plus 1
  - The parent of v is u
  - Add v to the To-Do List
- u has been fully explored.

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```

BFS Algorithm

Comments

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```

BFS Algorithm

Comments

- $d[u]$ is the shortest path from s to u

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```

BFS Algorithm

Comments

- $d[u]$ is the shortest path from s to u
- We can follow parent pointers back to s to actually retrieve the shortest path.

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
`enqueue`(Q, s)
while $Q \neq \emptyset$ **do**
    $u =$ `dequeue`(Q)
    for *each* v *adjacent to* u **do**
        if $color[v] == WHITE$ **then**
            $color[v]$ =GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$
            `enqueue`(Q, v)
    $color[u]$ =BLACK

BFS Algorithm

Comments

- $d[u]$ is the shortest path from s to u
- We can follow parent pointers back to s to actually retrieve the shortest path.
- Obtain Breadth First Tree by only considering edges form $(u, parent[u])$

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```

BFS Algorithm

Comments

- $d[u]$ is the shortest path from s to u
- We can follow parent pointers back to s to actually retrieve the shortest path.
- Obtain Breadth First Tree by only considering edges form $(u, parent[u])$
- Each path in the Breadth First Tree must be the shortest path in the graph

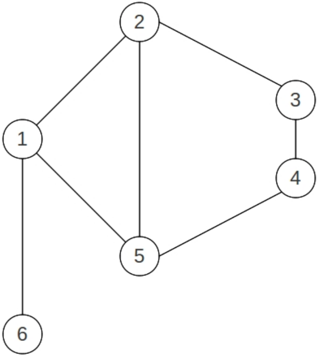
Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u] =$WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s] =$GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    $u =$ dequeue(Q)
    for *each* v *adjacent to* u **do**
        if $color[v] == WHITE$ **then**
            $color[v] =$GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$
            enqueue(Q, v)
    $color[u] =$BLACK

Example

Algorithm: `breadthFirstSearch`(V, E, s)

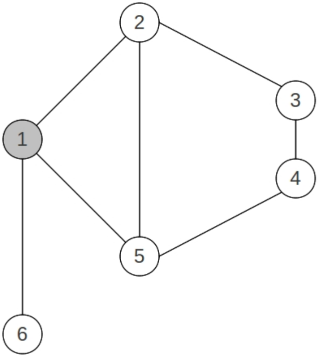
```
for each vertex u ∈ (V − {S}) do
    color[u] =WHITE
    parent[u] = nil
    d[u] = ∞
color[s] =GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] ==WHITE then
            color[v] =GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] =BLACK
```



Example

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```

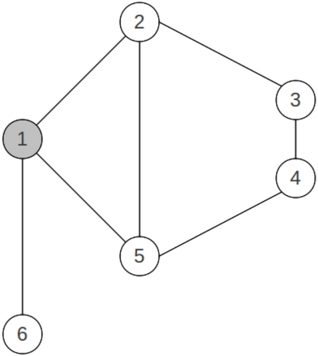


	1	2	3	4	5	6
Color[ ]	GRAY	WHITE	WHITE	WHITE	WHITE	WHITE
parent[ ]	NULL	NULL	NULL	NULL	NULL	NULL
d[ ]	0	∞	∞	∞	∞	∞

Q	1

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] =WHITE
    parent[u] = nil
    d[u] = ∞
color[s] =GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] ==WHITE then
            color[v] =GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] =BLACK
```

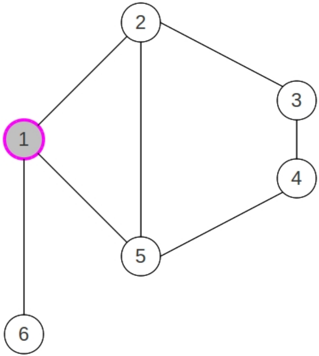


	1	2	3	4	5	6
Color[ ]	GRAY	WHITE	WHITE	WHITE	WHITE	WHITE
parent[ ]	NULL	NULL	NULL	NULL	NULL	NULL
d[ ]	0	∞	∞	∞	∞	∞
Q	1					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  u = dequeue(Q)
  for *each* v *adjacent to* u **do**
    if $color[v]$ ==*WHITE* **then**
      $color[v]$ =GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      enqueue(Q, v)
  $color[u]$ =BLACK



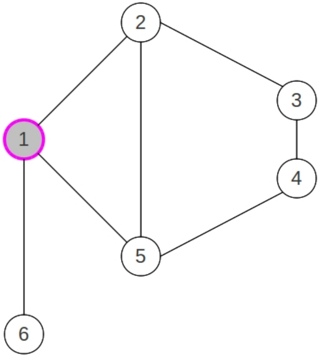
$u = 1$

	1	2	3	4	5	6
Color[ ]	GRAY	WHITE	WHITE	WHITE	WHITE	WHITE
parent[ ]	NULL	NULL	NULL	NULL	NULL	NULL
d[ ]	0	∞	∞	∞	∞	∞
Q						

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



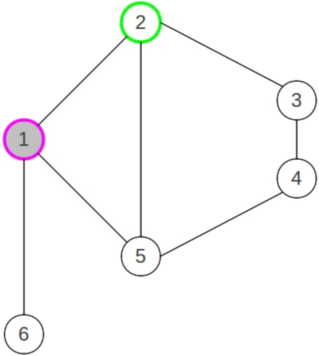
$u = 1$

	1	2	3	4	5	6
Color[ ]	GRAY	WHITE	WHITE	WHITE	WHITE	WHITE
parent[ ]	NULL	NULL	NULL	NULL	NULL	NULL
d[ ]	0	∞	∞	∞	∞	∞
Q						

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



$u = 1$

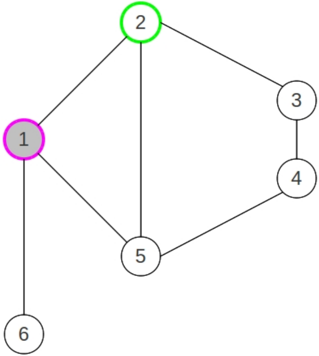
$v = 2$

	1	2	3	4	5	6
Color[ ]	GRAY	WHITE	WHITE	WHITE	WHITE	WHITE
parent[ ]	NULL	NULL	NULL	NULL	NULL	NULL
d[ ]	0	∞	∞	∞	∞	∞
Q						

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
`enqueue`(Q, s)
while $Q \neq \emptyset$ **do**
  $u =$ `dequeue`(Q)
  for *each* v *adjacent to* u **do**
    if $color[v] == WHITE$ **then**
      $color[v]$ =GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      `enqueue`(Q, v)
  $color[u]$ =BLACK



$u = 1$

$v = 2$

	1	2	3	4	5	6
Color[ ]	GRAY	WHITE	WHITE	WHITE	WHITE	WHITE
parent[ ]	NULL	NULL	NULL	NULL	NULL	NULL
d[ ]	0	∞	∞	∞	∞	∞
Q						

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
- $color[u]$ =WHITE
- $parent[u] = nil$
- $d[u] = \infty$

$color[s]$ =GRAY

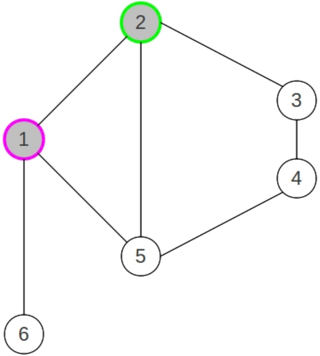
$parent[s] = nil$

$d[s] = 0$

create a queue, Q

`enqueue`(Q, s)

while $Q \neq \emptyset$ **do**
- $u =$ `dequeue`(Q)
- **for** *each* v *adjacent to* u **do**
  - **if** $color[v] == WHITE$ **then**
    - $color[v]$ =GRAY
    - $d[v] = d[u] + 1$
    - $parent[v] = u$
    - `enqueue`(Q, v)
- $color[u]$ =BLACK



$u = 1$

$v = 2$

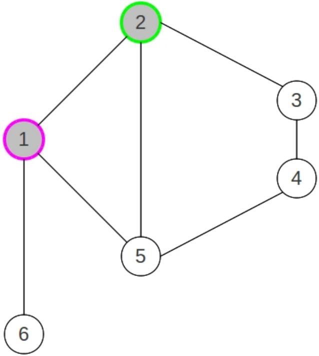
	1	2	3	4	5	6
Color[ ]	GRAY	GRAY	WHITE	WHITE	WHITE	WHITE
parent[ ]	NULL	NULL	NULL	NULL	NULL	NULL
d[ ]	0	∞	∞	∞	∞	∞

Q	

Example

Algorithm: breadthFirstSearch(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    $u =$ dequeue(Q)
    for *each* v *adjacent to* u **do**
        if $color[v] == WHITE$ **then**
            $color[v]$ =GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$
            enqueue(Q, v)
    $color[u]$ =BLACK



$u = 1$

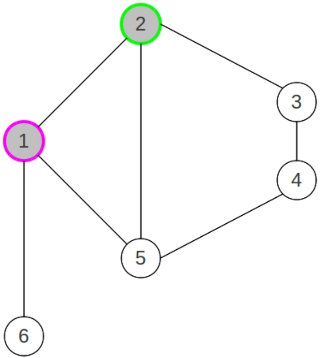
$v = 2$

	1	2	3	4	5	6
Color[ ]	GRAY	GRAY	WHITE	WHITE	WHITE	WHITE
parent[ ]	NULL	NULL	NULL	NULL	NULL	NULL
d[ ]	0	1	∞	∞	∞	∞
Q						

Example

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



$u = 1$

$v = 2$

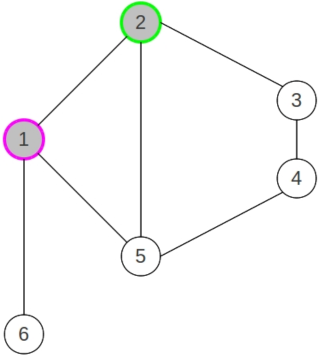
	1	2	3	4	5	6
Color[ ]	GRAY	GRAY	WHITE	WHITE	WHITE	WHITE
parent[ ]	NULL	1	NULL	NULL	NULL	NULL
d[ ]	0	1	∞	∞	∞	∞

Q	

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  $u =$ dequeue(Q)
  for *each* v *adjacent to* u **do**
    if $color[v] == WHITE$ **then**
      $color[v]$ =GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      enqueue(Q, v)
  $color[u]$ =BLACK



$u = 1$

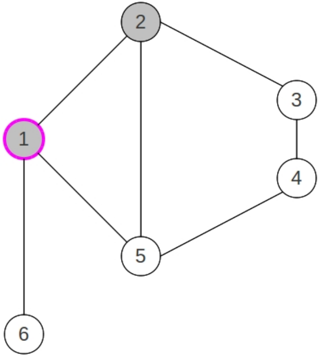
$v = 2$

	1	2	3	4	5	6
Color[ ]	GRAY	GRAY	WHITE	WHITE	WHITE	WHITE
parent[ ]	NULL	1	NULL	NULL	NULL	NULL
d[ ]	0	1	∞	∞	∞	∞
Q	2					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
`enqueue`(Q, s)
while $Q \neq \emptyset$ **do**
  $u =$ `dequeue`(Q)
  for *each v adjacent to u* **do**
    if $color[v]$ ==*WHITE* **then**
      $color[v]$ =GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      `enqueue`(Q, v)
  $color[u]$ =BLACK



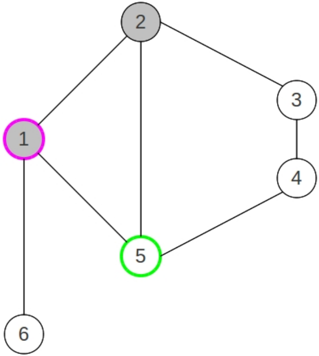
$u = 1$

	1	2	3	4	5	6
Color[ ]	GRAY	GRAY	WHITE	WHITE	WHITE	WHITE
parent[ ]	NULL	1	NULL	NULL	NULL	NULL
d[ ]	0	1	∞	∞	∞	∞
Q	2					

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] =WHITE
    parent[u] = nil
    d[u] = ∞
color[s] =GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] ==WHITE then
            color[v] =GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] =BLACK
```



$u = 1$

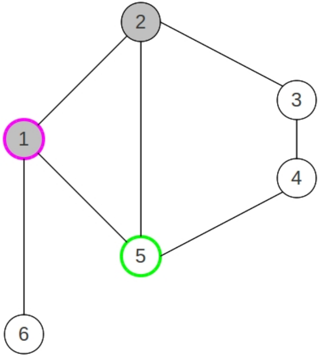
$v = 5$

	1	2	3	4	5	6
Color[ ]	GRAY	GRAY	WHITE	WHITE	WHITE	WHITE
parent[ ]	NULL	1	NULL	NULL	NULL	NULL
d[ ]	0	1	∞	∞	∞	∞
Q	2					

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] =WHITE
    parent[u] = nil
    d[u] = ∞
color[s] =GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] ==WHITE then
            color[v] =GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] =BLACK
```



$u = 1$

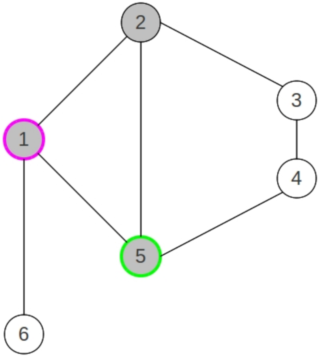
$v = 5$

	1	2	3	4	5	6
Color[ ]	GRAY	GRAY	WHITE	WHITE	WHITE	WHITE
parent[ ]	NULL	1	NULL	NULL	NULL	NULL
d[ ]	0	1	∞	∞	∞	∞
Q	2					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  $u =$ dequeue(Q)
  for *each v adjacent to u* **do**
    if $color[v]$ ==*WHITE* **then**
      $color[v]$ =GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      enqueue(Q, v)
  $color[u]$ =BLACK



$u = 1$

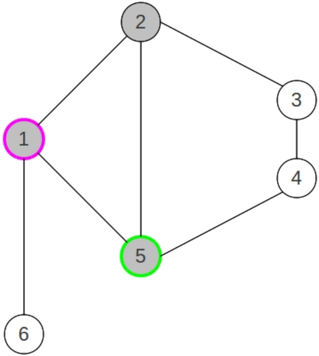
$v = 5$

	1	2	3	4	5	6
Color[ ]	GRAY	GRAY	WHITE	WHITE	GRAY	WHITE
parent[ ]	NULL	1	NULL	NULL	NULL	NULL
d[ ]	0	1	∞	∞	∞	∞
Q	2					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    $u =$ dequeue(Q)
    for *each v adjacent to u* **do**
        if $color[v] == WHITE$ **then**
            $color[v]$ =GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$
            enqueue(Q, v)
    $color[u]$ =BLACK



$u = 1$

$v = 5$

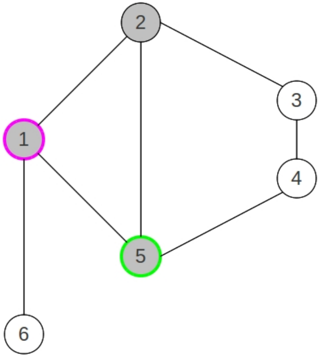
	1	2	3	4	5	6
Color[ ]	GRAY	GRAY	WHITE	WHITE	GRAY	WHITE
parent[ ]	NULL	1	NULL	NULL	NULL	NULL
d[ ]	0	1	∞	∞	1	∞
Q	2					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
- $color[u]$ =WHITE
- $parent[u] = nil$
- $d[u] = \infty$

$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
- $u =$ dequeue(Q)
- **for** *each* v *adjacent to* u **do**
  - **if** $color[v] == WHITE$ **then**
    - $color[v]$ =GRAY
    - $d[v] = d[u] + 1$
    - $parent[v] = u$
    - enqueue(Q, v)
- $color[u]$ =BLACK



$u = 1$

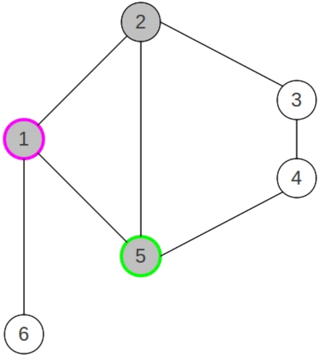
$v = 5$

	1	2	3	4	5	6
Color[ ]	GRAY	GRAY	WHITE	WHITE	GRAY	WHITE
parent[ ]	NULL	1	NULL	NULL	1	NULL
d[ ]	0	1	∞	∞	1	∞
Q	2					

Example

Algorithm: breadthFirstSearch(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  u = dequeue(Q)
  for *each v adjacent to u* **do**
    if $color[v]$ ==*WHITE* **then**
      $color[v]$ =GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      enqueue(Q, v)
  $color[u]$ =BLACK



$u = 1$

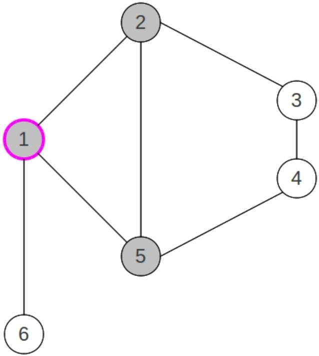
$v = 5$

	1	2	3	4	5	6
Color[ ]	GRAY	GRAY	WHITE	WHITE	GRAY	WHITE
parent[ ]	NULL	1	NULL	NULL	1	NULL
d[ ]	0	1	∞	∞	1	∞
Q	2, 5					

Example

Algorithm: breadthFirstSearch(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u] =$WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s] =$GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  $u =$ dequeue(Q)
  for *each v adjacent to u* **do**
    if $color[v] == WHITE$ **then**
      $color[v] =$GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      enqueue(Q, v)
  $color[u] =$BLACK



$u = 1$

	1	2	3	4	5	6
Color[ ]	GRAY	GRAY	WHITE	WHITE	GRAY	WHITE
parent[ ]	NULL	1	NULL	NULL	1	NULL
d[ ]	0	1	∞	∞	1	∞
Q	2, 5					

Example

Algorithm: breadthFirstSearch(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
- $color[u]$ =WHITE
- $parent[u] = nil$
- $d[u] = \infty$

$color[s]$ =GRAY

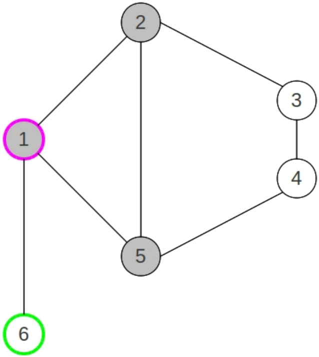
$parent[s] = nil$

$d[s] = 0$

create a queue, Q

enqueue(Q, s)

while $Q \neq \emptyset$ **do**
- $u =$ dequeue(Q)
- **for** *each v adjacent to u* **do**
  - **if** $color[v] == WHITE$ **then**
    - $color[v]$ =GRAY
    - $d[v] = d[u] + 1$
    - $parent[v] = u$
    - enqueue(Q, v)
- $color[u]$ =BLACK



$u = 1$

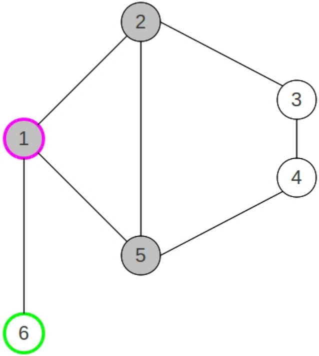
$v = 6$

	1	2	3	4	5	6
Color[ ]	GRAY	GRAY	WHITE	WHITE	GRAY	WHITE
parent[ ]	NULL	1	NULL	NULL	1	NULL
d[ ]	0	1	∞	∞	1	∞
Q	2, 5					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



$u = 1$

$v = 6$

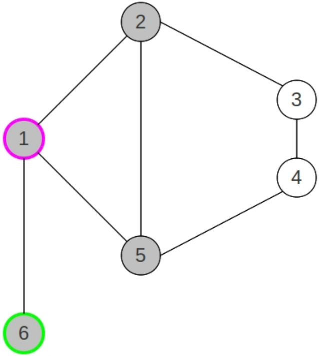
	1	2	3	4	5	6
Color[ ]	GRAY	GRAY	WHITE	WHITE	GRAY	WHITE
parent[ ]	NULL	1	NULL	NULL	1	NULL
d[ ]	0	1	∞	∞	1	∞

Q	2, 5

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u] =$WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s] =$GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  $u =$ dequeue(Q)
  for *each* v *adjacent to* u **do**
    if $color[v] ==$ *WHITE* **then**
      $color[v] =$GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      enqueue(Q, v)
  $color[u] =$BLACK



$u = 1$

$v = 6$

	1	2	3	4	5	6
Color[ ]	GRAY	GRAY	WHITE	WHITE	GRAY	GRAY
parent[ ]	NULL	1	NULL	NULL	1	NULL
d[ ]	0	1	∞	∞	1	∞
Q	2, 5					

Example

Algorithm: breadthFirstSearch(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
- $color[u]$ =WHITE
- $parent[u] = nil$
- $d[u] = \infty$

$color[s]$ =GRAY

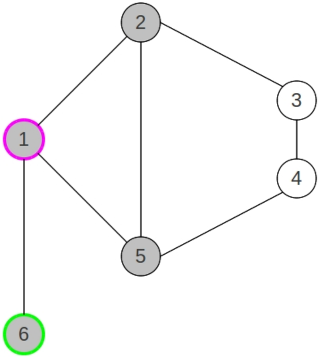
$parent[s] = nil$

$d[s] = 0$

create a queue, Q

enqueue(Q, s)

while $Q \neq \emptyset$ **do**
- $u =$ dequeue(Q)
- **for** *each* v *adjacent to* u **do**
  - **if** $color[v]$ ==*WHITE* **then**
    - $color[v]$ =GRAY
    - $d[v] = d[u] + 1$
    - $parent[v] = u$
    - enqueue(Q, v)
- $color[u]$ =BLACK



$u = 1$

$v = 6$

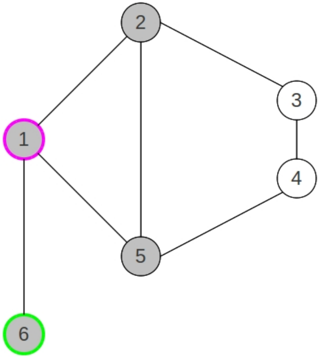
	1	2	3	4	5	6
Color[ ]	GRAY	GRAY	WHITE	WHITE	GRAY	GRAY
parent[ ]	NULL	1	NULL	NULL	1	NULL
d[ ]	0	1	∞	∞	1	1

Q	2, 5

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    $u =$ dequeue(Q)
    for *each* v *adjacent to* u **do**
        if $color[v] == WHITE$ **then**
            $color[v]$ =GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$
            enqueue(Q, v)
    $color[u]$ =BLACK



$u = 1$

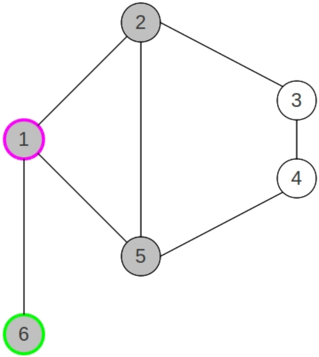
$v = 6$

	1	2	3	4	5	6
Color[ ]	GRAY	GRAY	WHITE	WHITE	GRAY	GRAY
parent[ ]	NULL	1	NULL	NULL	1	1
d[ ]	0	1	∞	∞	1	1
Q	2, 5					

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



$u = 1$

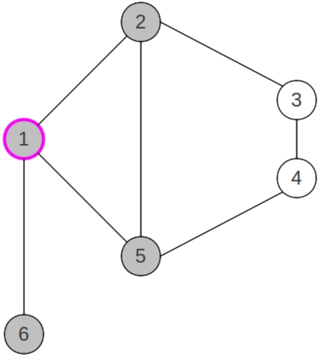
$v = 6$

	1	2	3	4	5	6
Color[ ]	GRAY	GRAY	WHITE	WHITE	GRAY	GRAY
parent[ ]	NULL	1	NULL	NULL	1	1
d[ ]	0	1	∞	∞	1	1
Q	2, 5, 6					

Example

Algorithm: breadthFirstSearch(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  u = dequeue(Q)
  for *each v adjacent to u* **do**
    if $color[v] == WHITE$ **then**
      $color[v]$ =GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      enqueue(Q, v)
  $color[u]$ =BLACK



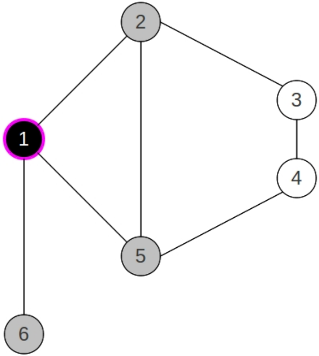
$u = 1$

	1	2	3	4	5	6
Color[ ]	GRAY	GRAY	WHITE	WHITE	GRAY	GRAY
parent[ ]	NULL	1	NULL	NULL	1	1
d[ ]	0	1	∞	∞	1	1
Q	2, 5, 6					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  $u =$ dequeue(Q)
  for *each* v *adjacent to* u **do**
    if $color[v] ==$ *WHITE* **then**
      $color[v]$ =GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      enqueue(Q, v)
  $color[u]$ =BLACK



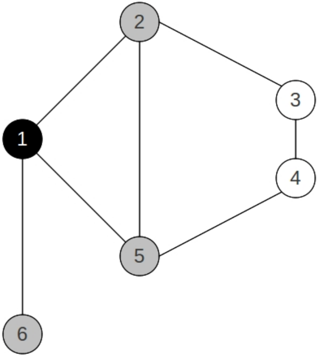
$u = 1$

	1	2	3	4	5	6
Color[ ]	BLACK	GRAY	WHITE	WHITE	GRAY	GRAY
parent[ ]	NULL	1	NULL	NULL	1	1
d[ ]	0	1	∞	∞	1	1
Q	2, 5, 6					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```

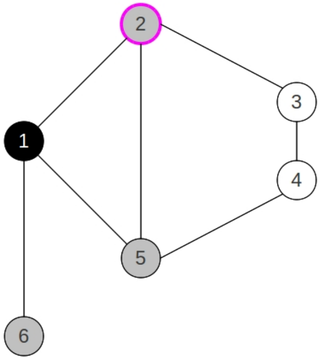


	1	2	3	4	5	6
Color[ ]	BLACK	GRAY	WHITE	WHITE	GRAY	GRAY
parent[ ]	NULL	1	NULL	NULL	1	1
d[ ]	0	1	∞	∞	1	1
Q	2, 5, 6					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    $u =$ dequeue(Q)
    for *each* v *adjacent to* u **do**
        if $color[v] == WHITE$ **then**
            $color[v]$ =GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$
            enqueue(Q, v)
    $color[u]$ =BLACK



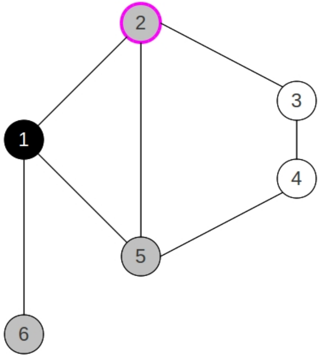
$u = 2$

	1	2	3	4	5	6
Color[ ]	BLACK	GRAY	WHITE	WHITE	GRAY	GRAY
parent[ ]	NULL	1	NULL	NULL	1	1
d[ ]	0	1	∞	∞	1	1
Q	5, 6					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



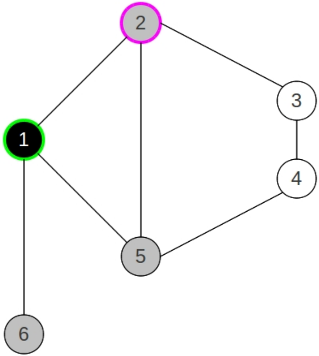
$u = 2$

	1	2	3	4	5	6
Color[ ]	BLACK	GRAY	WHITE	WHITE	GRAY	GRAY
parent[ ]	NULL	1	NULL	NULL	1	1
d[ ]	0	1	∞	∞	1	1
Q	5, 6					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



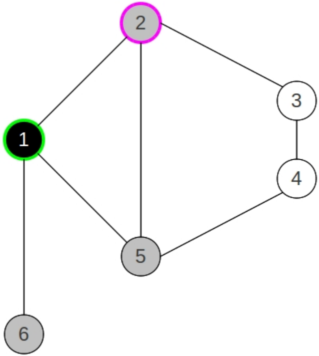
$u = 2$

$v = 1$

	1	2	3	4	5	6
Color[ ]	BLACK	GRAY	WHITE	WHITE	GRAY	GRAY
parent[ ]	NULL	1	NULL	NULL	1	1
d[ ]	0	1	∞	∞	1	1
Q	5, 6					

Algorithm: breadthFirstSearch(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  u = dequeue(Q)
  for *each* v *adjacent to* u **do**
    if $color[v] == WHITE$ **then**
      $color[v]$ =GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      enqueue(Q, v)
  $color[u]$ =BLACK



$u = 2$

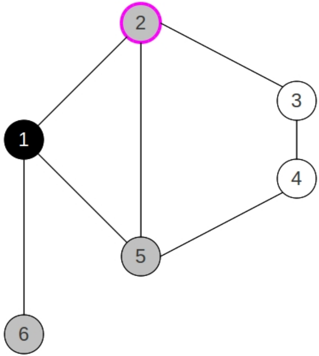
$v = 1$

	1	2	3	4	5	6
Color[ ]	BLACK	GRAY	WHITE	WHITE	GRAY	GRAY
parent[ ]	NULL	1	NULL	NULL	1	1
d[ ]	0	1	∞	∞	1	1
Q	5, 6					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



$u = 2$

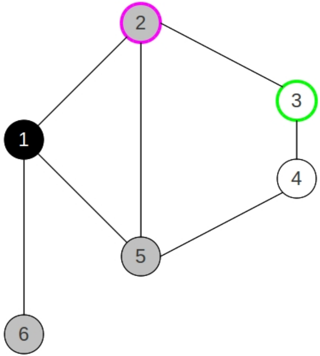
	1	2	3	4	5	6
Color[ ]	BLACK	GRAY	WHITE	WHITE	GRAY	GRAY
parent[ ]	NULL	1	NULL	NULL	1	1
d[ ]	0	1	∞	∞	1	1

Q	5, 6

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
`enqueue`(Q, s)
while $Q \neq \emptyset$ **do**
  $u =$ `dequeue`(Q)
  for *each v adjacent to u* **do**
    if $color[v] ==$ *WHITE* **then**
      $color[v]$ =GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      `enqueue`(Q, v)
  $color[u]$ =BLACK



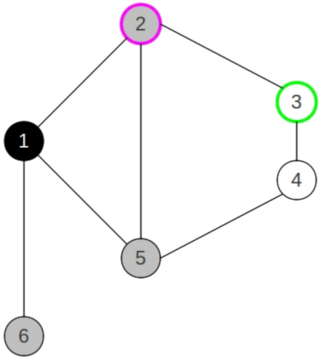
$u = 2$

$v = 3$

	1	2	3	4	5	6
Color[ ]	BLACK	GRAY	WHITE	WHITE	GRAY	GRAY
parent[ ]	NULL	1	NULL	NULL	1	1
d[ ]	0	1	∞	∞	1	1
Q	5, 6					

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



$u = 2$

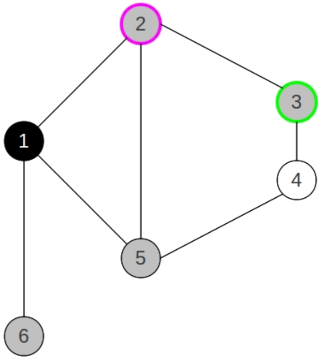
$v = 3$

	1	2	3	4	5	6
Color[ ]	BLACK	GRAY	WHITE	WHITE	GRAY	GRAY
parent[ ]	NULL	1	NULL	NULL	1	1
d[ ]	0	1	∞	∞	1	1
Q	5, 6					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    $u =$ dequeue(Q)
    for *each* v *adjacent to* u **do**
        if $color[v]$ ==*WHITE* **then**
            $color[v]$ =GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$
            enqueue(Q, v)
    $color[u]$ =BLACK



$u = 2$

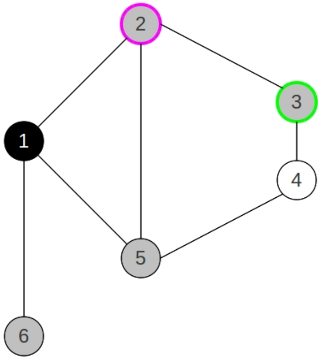
$v = 3$

	1	2	3	4	5	6
Color[ ]	BLACK	GRAY	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	NULL	NULL	1	1
d[ ]	0	1	∞	∞	1	1
Q	5, 6					

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



$u = 2$

$v = 3$

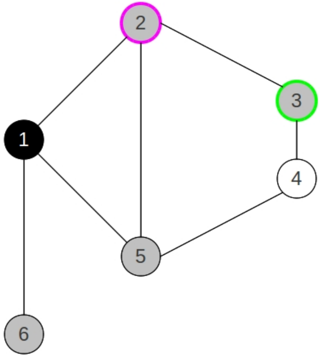
	1	2	3	4	5	6
Color[ ]	BLACK	GRAY	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	NULL	NULL	1	1
d[ ]	0	1	2	∞	1	1

Q	5, 6

Example

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



$u = 2$

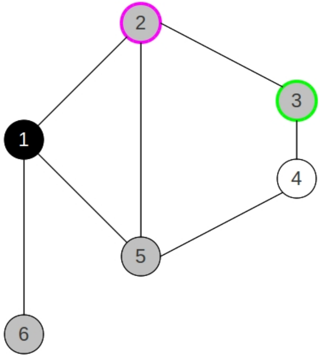
$v = 3$

	1	2	3	4	5	6
Color[ ]	BLACK	GRAY	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	∞	1	1
Q	5, 6					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u] =$WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s] =$GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  $u =$ dequeue(Q)
  for *each v adjacent to u* **do**
    if $color[v] == WHITE$ **then**
      $color[v] =$GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      enqueue(Q, v)
  $color[u] =$BLACK



$u = 2$

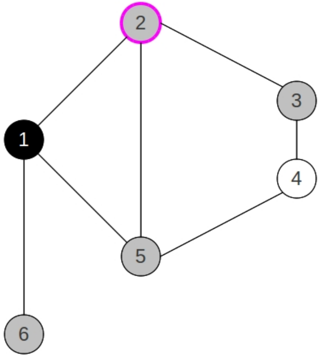
$v = 3$

	1	2	3	4	5	6
Color[ ]	BLACK	GRAY	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	∞	1	1
Q	5, 6, 3					

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



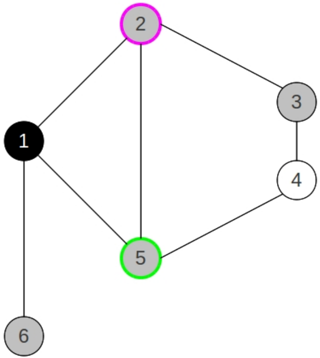
$u = 2$

	1	2	3	4	5	6
Color[ ]	BLACK	GRAY	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	∞	1	1
Q	5, 6, 3					

Example

Algorithm: breadthFirstSearch(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  u = dequeue(Q)
  for *each v adjacent to u* **do**
    if $color[v]$ ==*WHITE* **then**
      $color[v]$ =GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      enqueue(Q, v)
  $color[u]$ =BLACK



$u = 2$

$v = 5$

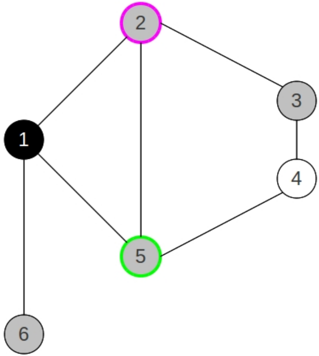
	1	2	3	4	5	6
Color[ ]	BLACK	GRAY	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	∞	1	1
Q	5, 6, 3					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
- $color[u]$ =WHITE
- $parent[u] = nil$
- $d[u] = \infty$

$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
- $u =$ dequeue(Q)
- **for** *each* v *adjacent to* u **do**
  - **if** $color[v] == WHITE$ **then**
    - $color[v]$ =GRAY
    - $d[v] = d[u] + 1$
    - $parent[v] = u$
    - enqueue(Q, v)
- $color[u]$ =BLACK



$u = 2$

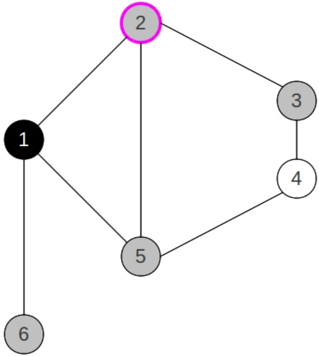
$v = 5$

	1	2	3	4	5	6
Color[ ]	BLACK	GRAY	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	∞	1	1
Q	5, 6, 3					

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



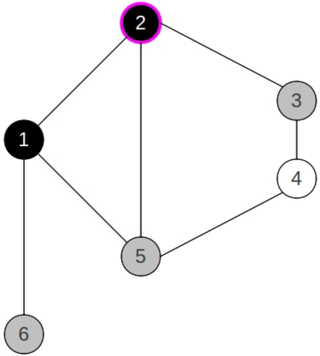
$u = 2$

	1	2	3	4	5	6
Color[ ]	BLACK	GRAY	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	∞	1	1
Q	5, 6, 3					

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



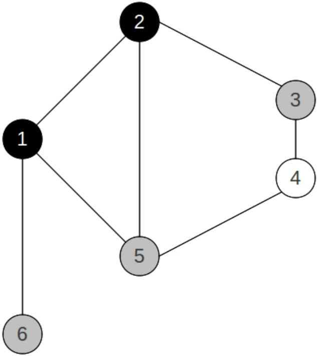
$u = 2$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	∞	1	1
Q	5, 6, 3					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

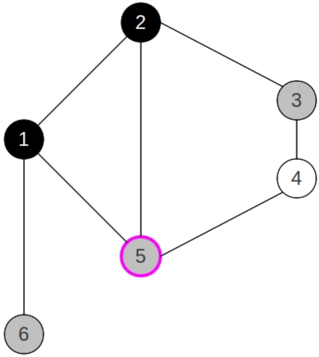
for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  $u =$ dequeue(Q)
  for *each* v *adjacent to* u **do**
    if $color[v] == WHITE$ **then**
      $color[v]$ =GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      enqueue(Q, v)
  $color[u]$ =BLACK



	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	∞	1	1
Q	5, 6, 3					

Example

```
Algorithm: breadthFirstSearch(V, E, s)
for each vertex u ∈ (V − {S}) do
    color[u] =WHITE
    parent[u] = nil
    d[u] = ∞
color[s] =GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] ==WHITE then
            color[v] =GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] =BLACK
```



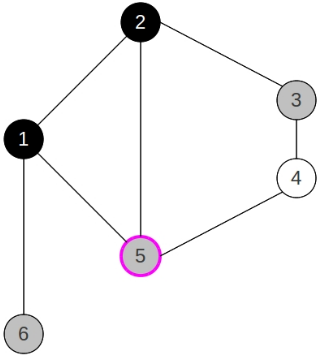
$u = 5$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	∞	1	1
Q	6, 3					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  $u =$ dequeue(Q)
  for *each v adjacent to u* **do**
    if $color[v] == WHITE$ **then**
      $color[v]$ =GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      enqueue(Q, v)
  $color[u]$ =BLACK



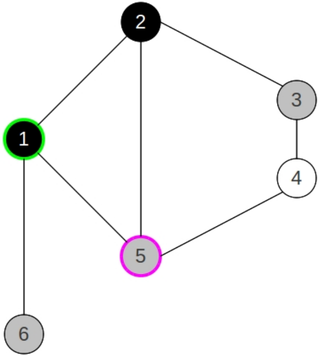
$u = 5$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	∞	1	1
Q	6, 3					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    $u =$ dequeue(Q)
    for *each v adjacent to u* **do**
        if $color[v] == WHITE$ **then**
            $color[v]$ =GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$
            enqueue(Q, v)
    $color[u]$ =BLACK



$u = 5$

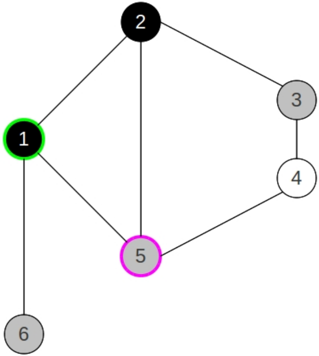
$v = 1$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	∞	1	1
Q	6, 3					

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



$u = 5$

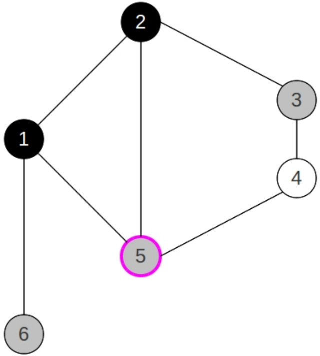
$v = 1$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	∞	1	1

Q	6, 3

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



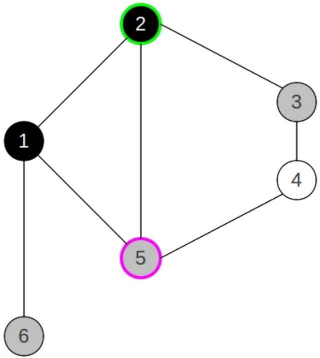
$u = 5$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	∞	1	1
Q	6, 3					

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



$u = 5$

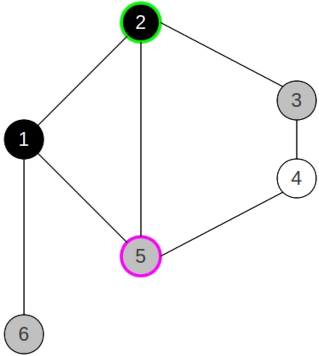
$v = 2$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	∞	1	1
Q	6, 3					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    $u =$ dequeue(Q)
    for *each* v *adjacent to* u **do**
        if $color[v] == WHITE$ **then**
            $color[v]$ =GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$
            enqueue(Q, v)
    $color[u]$ =BLACK



$u = 5$

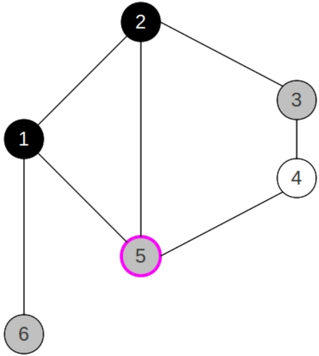
$v = 2$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	∞	1	1
Q	6, 3					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  $u =$ dequeue(Q)
  for *each v adjacent to u* **do**
    if $color[v]$ ==*WHITE* **then**
      $color[v]$ =GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      enqueue(Q, v)
  $color[u]$ =BLACK



$u = 5$

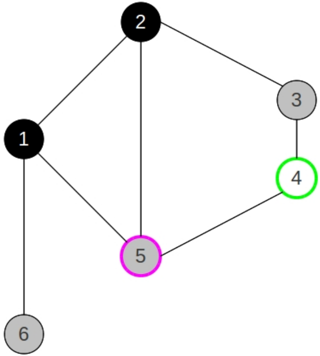
	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	∞	1	1
Q	6, 3					

Example

Algorithm: $\texttt{breadthFirstSearch}(V, E, s)$

for *each vertex* $u \in (V - \{S\})$ **do**
- $color[u]$ =WHITE
- $parent[u] = nil$
- $d[u] = \infty$

$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
$\texttt{enqueue}(Q, s)$
while $Q \neq \emptyset$ **do**
- $u = \texttt{dequeue}(Q)$
- **for** *each v adjacent to u* **do**
  - **if** $color[v] == WHITE$ **then**
    - $color[v]$ =GRAY
    - $d[v] = d[u] + 1$
    - $parent[v] = u$
    - $\texttt{enqueue}(Q, v)$
- $color[u]$ =BLACK



$u = 5$

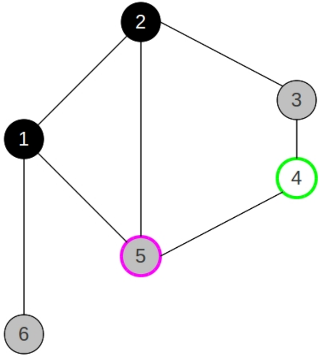
$v = 4$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	∞	1	1
Q	6, 3					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    color$[u]$ =WHITE
    parent$[u] = nil$
    $d[u] = \infty$
color$[s]$ =GRAY
parent$[s] = nil$
$d[s] = 0$
create a queue, Q
`enqueue`(Q, s)
while $Q \neq \emptyset$ **do**
    $u =$ `dequeue`(Q)
    for *each* v *adjacent to* u **do**
        if *color*$[v]$ ==*WHITE* **then**
            color$[v]$ =GRAY
            $d[v] = d[u] + 1$
            parent$[v] = u$
            `enqueue`(Q, v)
    color$[u]$ =BLACK



$u = 5$

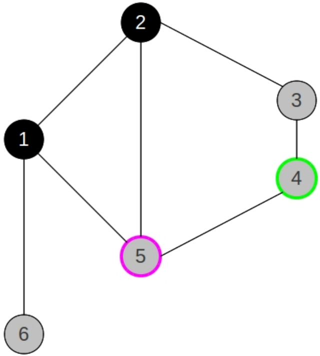
$v = 4$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	WHITE	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	∞	1	1
Q	6, 3					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    u = dequeue(Q)
    for *each* v *adjacent to* u **do**
        if $color[v]$ ==*WHITE* **then**
            $color[v]$ =GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$
            enqueue(Q, v)
    $color[u]$ =BLACK



$u = 5$

$v = 4$

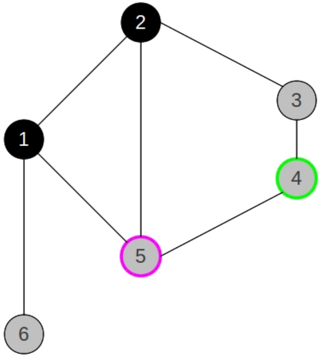
	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	∞	1	1

Q	6, 3

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



$u = 5$

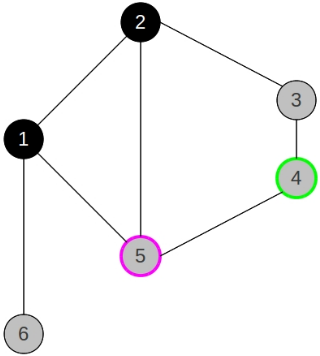
$v = 4$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	GRAY	GRAY
parent[ ]	NULL	1	2	NULL	1	1
d[ ]	0	1	2	2	1	1
Q	6, 3					

Example

Algorithm: breadthFirstSearch(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    u = dequeue(Q)
    for *each* v *adjacent to* u **do**
        if $color[v]$ ==*WHITE* **then**
            $color[v]$ =GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$
            enqueue(Q, v)
    $color[u]$ =BLACK



$u = 5$

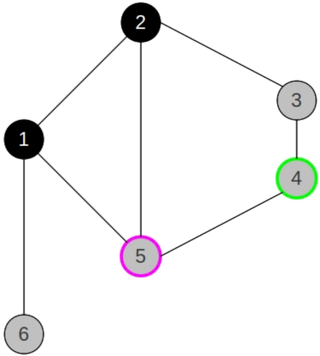
$v = 4$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	GRAY	GRAY
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q	6, 3, 4					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] =WHITE
    parent[u] = nil
    d[u] = ∞
color[s] =GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] ==WHITE then
            color[v] =GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] =BLACK
```



$u = 5$

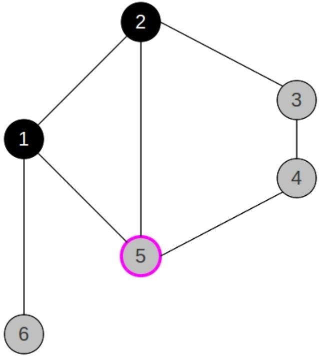
$v = 4$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	GRAY	GRAY
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q	6, 3, 4					

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] =WHITE
    parent[u] = nil
    d[u] = ∞
color[s] =GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] ==WHITE then
            color[v] =GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] =BLACK
```



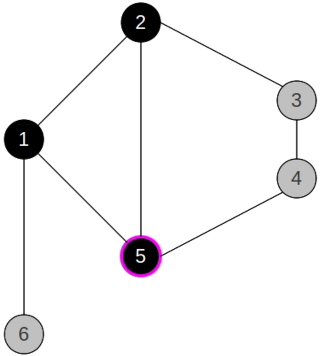
$u = 5$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	GRAY	GRAY
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q	6, 3, 4					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



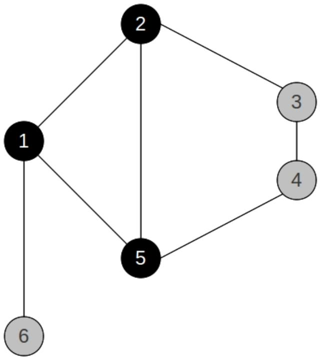
$u = 5$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	BLACK	GRAY
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q	6, 3, 4					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```

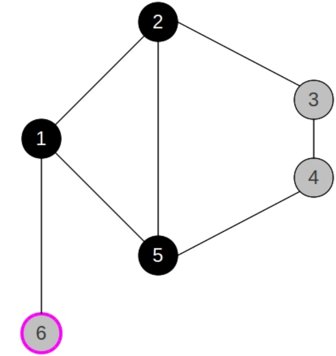


	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	BLACK	GRAY
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q	6, 3, 4					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    $u =$ dequeue(Q)
    for *each* v *adjacent to* u **do**
        if $color[v] ==$ *WHITE* **then**
            $color[v]$ =GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$
            enqueue(Q, v)
    $color[u]$ =BLACK



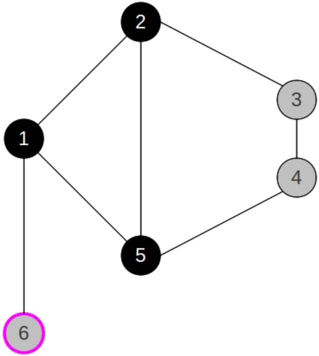
$u = 6$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	BLACK	GRAY
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q	3, 4					

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] =WHITE
    parent[u] = nil
    d[u] = ∞
color[s] =GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] ==WHITE then
            color[v] =GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] =BLACK
```



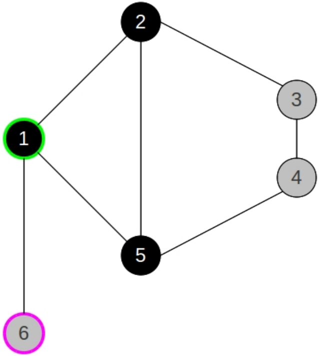
$u = 6$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	BLACK	GRAY
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q	3, 4					

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] =WHITE
    parent[u] = nil
    d[u] = ∞
color[s] =GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] ==WHITE then
            color[v] =GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] =BLACK
```



$u = 6$

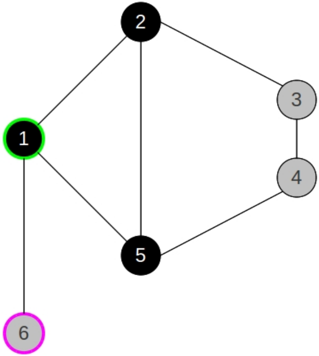
$v = 1$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	BLACK	GRAY
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q	3, 4					

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] =WHITE
    parent[u] = nil
    d[u] = ∞
color[s] =GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] ==WHITE then
            color[v] =GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] =BLACK
```



$u = 6$

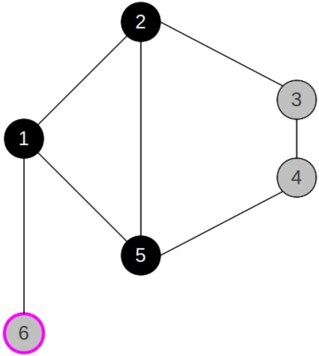
$v = 1$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	BLACK	GRAY
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q	3, 4					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



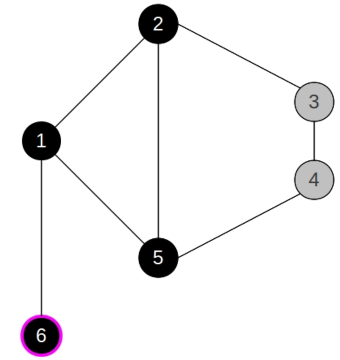
$u = 6$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	BLACK	GRAY
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q	3, 4					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  $u =$ dequeue(Q)
  for *each v adjacent to u* **do**
    if $color[v] == WHITE$ **then**
      $color[v]$ =GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      enqueue(Q, v)
  $color[u]$ =BLACK



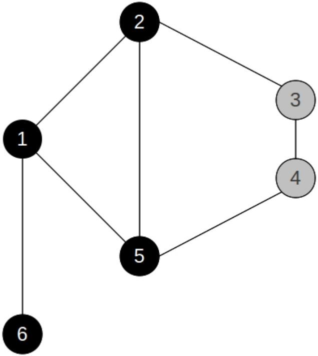
$u = 6$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q	3, 4					

Example

Algorithm: breadthFirstSearch(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u] =$WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s] =$GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    $u =$ dequeue(Q)
    for *each* v *adjacent to* u **do**
        if $color[v] == WHITE$ **then**
            $color[v] =$GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$
            enqueue(Q, v)
    $color[u] =$BLACK



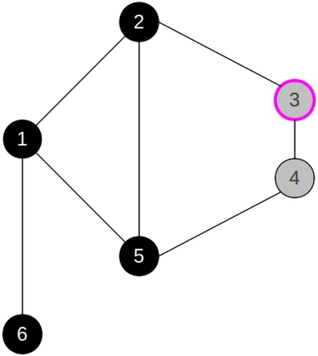
	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q	3, 4					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
- $color[u]$ =WHITE
- $parent[u] = nil$
- $d[u] = \infty$

$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
- u = dequeue(Q)
- **for** *each v adjacent to u* **do**
  - **if** $color[v]$ ==*WHITE* **then**
    - $color[v]$ =GRAY
    - $d[v] = d[u] + 1$
    - $parent[v] = u$
    - enqueue(Q, v)
- $color[u]$ =BLACK



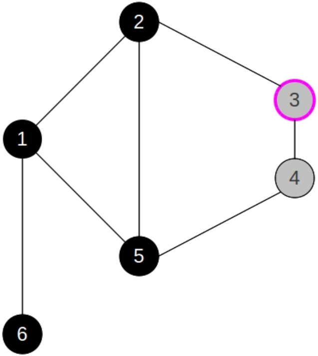
$u = 3$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q	4					

Example

Algorithm: breadthFirstSearch(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    u = dequeue(Q)
    for *each v adjacent to u* **do**
        if $color[v] == WHITE$ **then**
            $color[v]$ =GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$
            enqueue(Q, v)
    $color[u]$ =BLACK



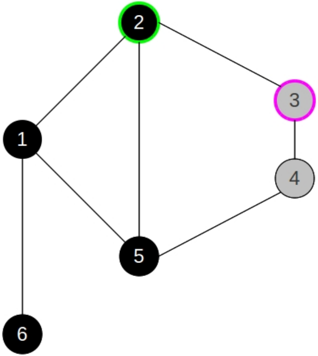
$u = 3$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q	4					

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] =WHITE
    parent[u] = nil
    d[u] = ∞
color[s] =GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] ==WHITE then
            color[v] =GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] =BLACK
```



$u = 3$

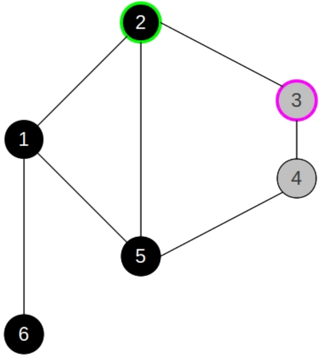
$v = 2$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q	4					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  $u =$ dequeue(Q)
  for *each* v *adjacent to* u **do**
    if $color[v] == WHITE$ **then**
      $color[v]$ =GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      enqueue(Q, v)
  $color[u]$ =BLACK



$u = 3$

$v = 2$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1

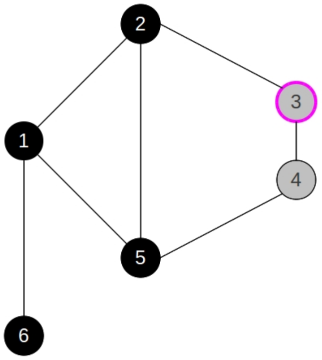
Q	4

Example

Algorithm: $\texttt{breadthFirstSearch}(V, E, s)$

for *each vertex* $u \in (V - \{S\})$ **do**
- $color[u] =$WHITE
- $parent[u] = nil$
- $d[u] = \infty$

$color[s] =$GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
$\texttt{enqueue}(Q, s)$
while $Q \neq \emptyset$ **do**
- $u = \texttt{dequeue}(Q)$
- **for** *each v adjacent to u* **do**
  - **if** $color[v] == WHITE$ **then**
    - $color[v] =$GRAY
    - $d[v] = d[u] + 1$
    - $parent[v] = u$
    - $\texttt{enqueue}(Q, v)$
- $color[u] =$BLACK



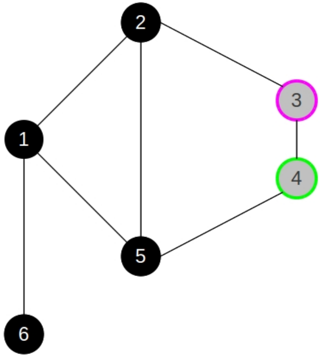
$u = 3$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q	4					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    $u =$ dequeue(Q)
    for *each v adjacent to u* **do**
        if $color[v]$ ==*WHITE* **then**
            $color[v]$ =GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$
            enqueue(Q, v)
    $color[u]$ =BLACK



$u = 3$

$v = 4$

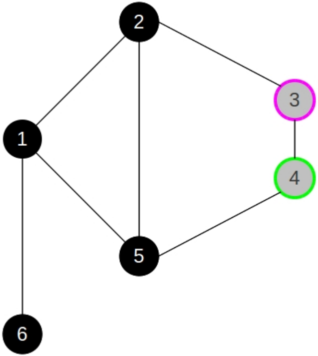
	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1

Q	4

Example

Algorithm: breadthFirstSearch(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    u = dequeue(Q)
    for *each v adjacent to u* **do**
        if $color[v] == WHITE$ **then**
            $color[v]$ =GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$
            enqueue(Q, v)
    $color[u]$ =BLACK



$u = 3$

$v = 4$

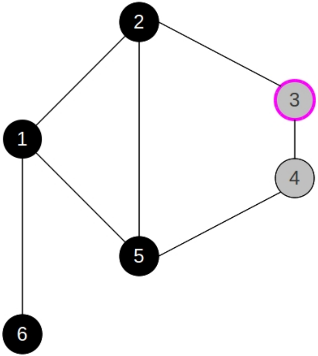
	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1

Q	4

Example

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```

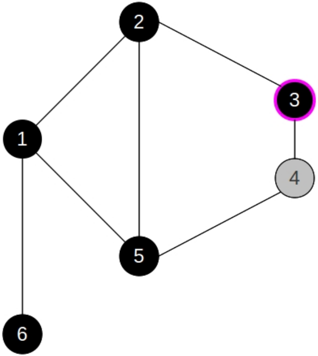


$u = 3$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	GRAY	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q	4					

Algorithm: `breadthFirstSearch`(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] =WHITE
    parent[u] = nil
    d[u] = ∞
color[s] =GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] ==WHITE then
            color[v] =GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] =BLACK
```

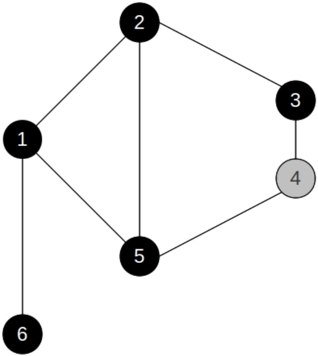


$u = 3$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	BLACK	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q	4					

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```

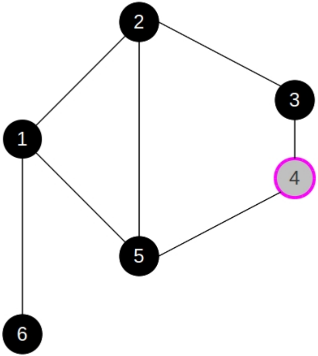


	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	BLACK	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q	4					

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  $u =$ dequeue(Q)
  for *each* v *adjacent to* u **do**
    if $color[v] == WHITE$ **then**
      $color[v]$ =GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      enqueue(Q, v)
  $color[u]$ =BLACK



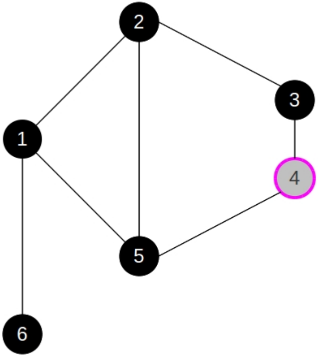
$u = 4$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	BLACK	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q						

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] = WHITE
    parent[u] = nil
    d[u] = ∞
color[s] = GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] == WHITE then
            color[v] = GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] = BLACK
```



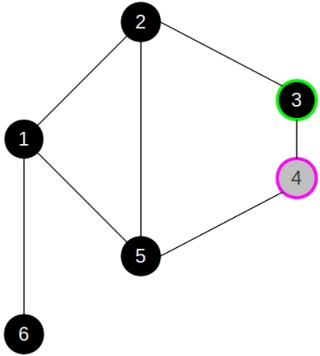
$u = 4$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	BLACK	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q						

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u] =$WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s] =$GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    $u =$ dequeue(Q)
    for *each v adjacent to u* **do**
        if $color[v] == WHITE$ **then**
            $color[v] =$GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$
            enqueue(Q, v)
    $color[u] =$BLACK



$u = 4$

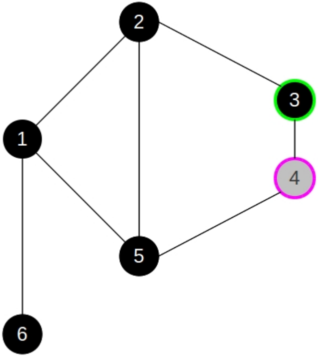
$v = 3$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	BLACK	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q						

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] =WHITE
    parent[u] = nil
    d[u] = ∞
color[s] =GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] ==WHITE then
            color[v] =GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] =BLACK
```



$u = 4$

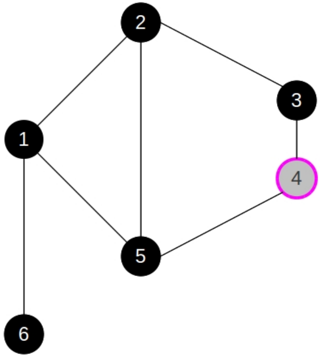
$v = 3$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	BLACK	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q						

Example

Algorithm: breadthFirstSearch(V, E, s)

```
for each vertex u ∈ (V − {S}) do
    color[u] =WHITE
    parent[u] = nil
    d[u] = ∞
color[s] =GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] ==WHITE then
            color[v] =GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] =BLACK
```



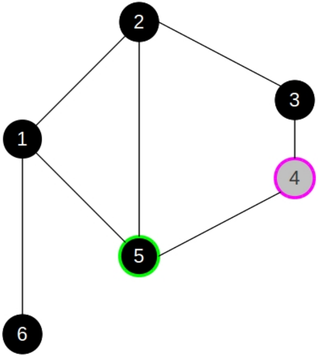
$u = 4$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	BLACK	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q						

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u]$ =WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    $u =$ dequeue(Q)
    for *each v adjacent to u* **do**
        if $color[v]$ ==*WHITE* **then**
            $color[v]$ =GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$
            enqueue(Q, v)
    $color[u]$ =BLACK



$u = 4$

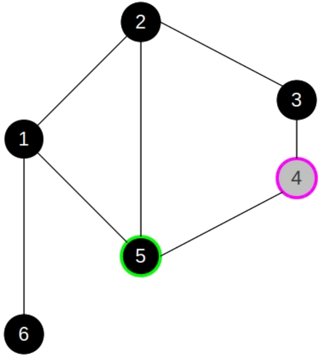
$v = 5$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	BLACK	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q						

Example

Algorithm: `breadthFirstSearch`(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
    $color[u] =$WHITE
    $parent[u] = nil$
    $d[u] = \infty$
$color[s] =$GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
    $u =$ dequeue(Q)
    for *each* v *adjacent to* u **do**
        if $color[v] == WHITE$ **then**
            $color[v] =$GRAY
            $d[v] = d[u] + 1$
            $parent[v] = u$
            enqueue(Q, v)
    $color[u] =$BLACK



$u = 4$

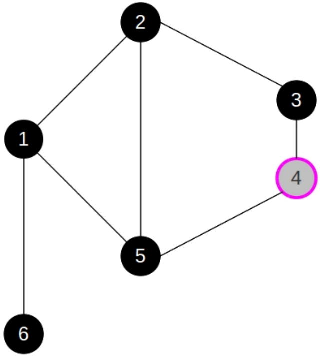
$v = 5$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	BLACK	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q						

Example

Algorithm: breadthFirstSearch(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  $u =$ dequeue(Q)
  for *each v adjacent to u* **do**
    if $color[v] ==$ *WHITE* **then**
      $color[v]$ =GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      enqueue(Q, v)
  $color[u]$ =BLACK

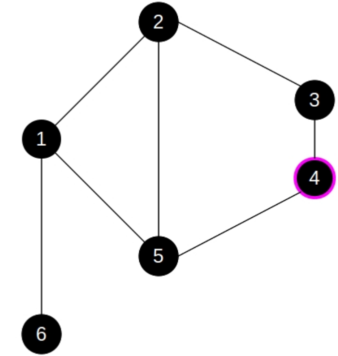


$u = 4$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	BLACK	GRAY	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q						

Example

```
Algorithm: breadthFirstSearch(V, E, s)
for each vertex u ∈ (V − {S}) do
    color[u] =WHITE
    parent[u] = nil
    d[u] = ∞
color[s] =GRAY
parent[s] = nil
d[s] = 0
create a queue, Q
enqueue(Q, s)
while Q ≠ ∅ do
    u = dequeue(Q)
    for each v adjacent to u do
        if color[v] ==WHITE then
            color[v] =GRAY
            d[v] = d[u] + 1
            parent[v] = u
            enqueue(Q, v)
    color[u] =BLACK
```



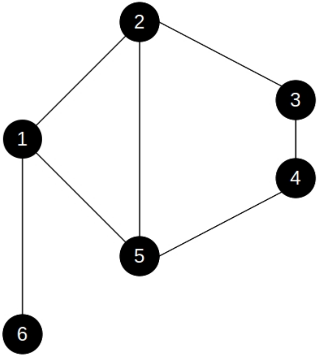
$u = 4$

	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	BLACK	BLACK	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1
Q						

Example

Algorithm: breadthFirstSearch(V, E, s)

for *each vertex* $u \in (V - \{S\})$ **do**
  $color[u]$ =WHITE
  $parent[u] = nil$
  $d[u] = \infty$
$color[s]$ =GRAY
$parent[s] = nil$
$d[s] = 0$
create a queue, Q
enqueue(Q, s)
while $Q \neq \emptyset$ **do**
  u = dequeue(Q)
  for *each v adjacent to u* **do**
    if $color[v] == WHITE$ **then**
      $color[v]$ =GRAY
      $d[v] = d[u] + 1$
      $parent[v] = u$
      enqueue(Q, v)
  $color[u]$ =BLACK



	1	2	3	4	5	6
Color[ ]	BLACK	BLACK	BLACK	BLACK	BLACK	BLACK
parent[ ]	NULL	1	2	5	1	1
d[ ]	0	1	2	2	1	1

Q	