Data structure and algorithm in Python

Graph

Seongjin Lee

Dept of Aerospace and Software Engineering,
Gyeongsang National University

Table of contents

1. Graphs
2. Data Structures for Graphs
3. Graph Traversals

Graphs

Graphs

Example : Graphs

A graph G is simply a set V of vertices and a collection E of pairs of vertices from V, called edges. A graph is a way of representing connections or relationships between pairs of objects from some set V.

Graphs

Edges in a graph are either directed or undirected.

- An edge (u, v) is said to be directed from u to v if the pair (u, v) is ordered, with u preceding v.
- An edge (u, v) is said to be undirected if the pair (u, v) is not ordered.

Undirected edges are sometimes denoted with set notation, as $\{u, v\}$, but for simplicity we use the pair notation (u, v), noting that in the undirected case (u, v) is the same as (v, u).

Definition : undirected, directed and mixed graph

- If all the edges in a graph are undirected, then we say the graph is an undirected graph.
- Likewise, a directed graph, also called a digraph, is a graph whose edges are all directed.
- A graph that has both directed and undirected edges is often called a mixed graph.

An undirected or mixed graph can be converted into a directed graph by replacing every undirected edge (u, v) by the pair of directed edges (u, v) and (v, u).

Graphs

- The two vertices joined by an edge are called the end vertices (or endpoints) of the edge.
- If an edge is directed, its first endpoint is its origin and the other is the destination of the edge.
- Two vertices u and v are said to be adjacent if there is an edge whose end vertices are u and v.

Graphs

- An edge is said to be incident to a vertex if the vertex is one of the edge's endpoints.
- The outgoing edges of a vertex are the directed edges whose origin is that vertex.
- The incoming edges of a vertex are the directed edges whose destination is that vertex.

Graphs

- The degree of a vertex v, denoted $deg(v)$, is the number of incident edges of v.
- The in-degree and out-degree of a vertex v are the number of the incoming and outgoing edges of v, and are denoted $indeg(v)$ and $outdeg(v)$, respectively.

Graphs

- The definition of a graph refers to the group of edges as a collection, not a set, thus allowing two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.
- Another special type of edge is one that connects a vertex to itself. Namely, we say that an edge (undirected or directed) is a ring or a self-loop if its two endpoints coincide.

Graphs

- With few exceptions, graphs do not have parallel edges or self-loops. Such graphs are said to be simple.
- Thus, we can usually say that the edges of a simple graph are a set of vertex pairs (and not just a collection).

Throughout this chapter, we assume that a graph is simple unless otherwise specified.

Graphs

- A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.
- A cycle is a path that starts and ends at the same vertex, and that includes at least one edge.

1. A path is simple if each vertex in the path is distinct.
2. A cycle is simple if each vertex in the cycle is distinct, except for the first and last one.
3. A directed path is a path such that all edges are directed and are traversed along their direction.
4. A directed cycle is similarly defined.

Graphs

- A directed graph is acyclic if it has no directed cycles.
- If a graph is simple, we may omit the edges when describing path P or cycle C, as these are well defined, in which case P is a list of adjacent vertices and C is a cycle of adjacent vertices.

Graphs

- Given vertices u and v of a (directed) graph G, we say that u reaches v, and that v is reachable from u, if G has a (directed) path from u to v.
- In an undirected graph, the notion of reachability is symmetric, that is to say, u reaches v if an only if v reaches u.
- However, in a directed graph, it is possible that u reaches v but v does not reach u, because a directed path must be traversed according to the respective directions of the edges.

- A graph is connected if, for any two vertices, there is a path between them.
- A directed graph $\vec{G}$ is strongly connected if for any two vertices u and v of $\vec{G}$, u reaches v and v reaches u.

- A subgraph of a graph G is a graph H whose vertices and edges are subsets of the vertices and edges of G, respectively.
- A spanning subgraph of G is a subgraph of G that contains all the vertices of the graph G.
- If a graph G is not connected, its maximal connected subgraphs are called the connected components of G.
- A forest is a graph without cycles.
- A tree is a connected forest, that is, a connected graph without cycles.
- A spanning tree of a graph is a spanning subgraph that is a tree.

Proposition

If G is a graph with m edges and vertex set $\mathcal{V}$, then

$$\sum_{v \in \mathcal{V}} deg(v) = 2m.$$

Proposition

If G is a directed graph with m edges and vertex set $\mathcal{V}$, then

$$\sum_{v \in \mathcal{V}} indeg(v) = \sum_{v \in \mathcal{V}} outdeg(v) = m.$$

Proposition

Let G be a simple graph with n vertices and m edges.

- If G is undirected, then

$$m \leq \frac{n(n-1)}{2}$$

- If G is directed, then

$$m \leq n(n-1).$$

Graphs

Proposition

Let G be a undirected graph with n vertices and m edges.

- If G is connected, then
$$m \geq n-1.$$
- If G is a tree, then
$$m = n-1.$$
- If G is a forest, then
$$m \leq n-1.$$

Graphs

The Graph ADT

The Graph ADT

Since a graph is a collection of vertices and edges, we model the abstraction as a com- bination of three data types:

- Vertex
- Edge
- Graph

The Graph ADT

A Veretex is lightweight object that stores an arbitrary element provided by the user, supporting the method:

- `element()`: Retrieve the stored element.

The Graph ADT

An Edge stores an associated object, supporting the following methods:

- `element()`: Retrieve the stored element.
- `endpoints()`: Return a tuple (u, v) such that vertex u is the origin of the edge and vertex v is the destination; for an undirected graph, the orientation is arbitrary.
- `opposite(v)`: Assuming vertex v is one endpoint of the edge (either origin or destination), return the other endpoint.

The Graph ADT

The primary abstraction for a graph is the Graph ADT. We presume that a graph can be either undirected or directed, with the designation declared upon construction; recall that a mixed graph can be represented as a directed graph, modeling edge $\{u, v\}$ as a pair of directed edges (u, v) and (v, u).

The Graph ADT

The Graph ADT includes the following methods

- `vertex_count()`: Return the number of vertices.
- `vertices()`: Return an iteration of all vertices.
- `edge_count()`: Return the number of edges.
- `edges()`: Return an iteration of all edges.
- `get_edge(u,v)`: Return the edge from vertex u to v, if one exists; otherwise return None.
- `degree(v, out=True)`:
  - For an undirected graph, return the number of edges incident to vertex v.
  - For a directed graph, return the number of outgoing (resp. incoming) edges incident to vertex v, as designated by the optional parameter.

The Graph ADT

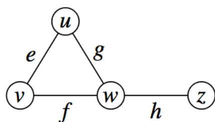
- `incident_edges(v, out=True)`:
  - Return an iteration of all edges incident to vertex v.
  - In the case of a directed graph,
    - report outgoing edges by default;
    - report incoming edges if the optional parameter is set to False.
- `insert_vertex(x=None)`: Create and return a new Vertex storing element x.
- `insert_edge(u, v, x=None)`: Create and return a new Edge from vertex u to vertex v, storing element x (None by default).
- `remove_vertex(v)`: Remove vertex v and all its incident edges from the graph.
- `remove_edge(e)`: Remove edge e from the graph.

Data Structures for Graphs

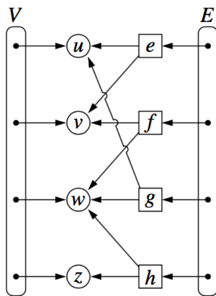
Data Structures for Graphs

- Edge List
- Adjacency List
- Adjacency Map
- Adjacency Matrix

Edge List



(a)



(b)

Figure 14.4: (a) A graph G; (b) schematic representation of the edge list structure for G. Notice that an edge object refers to the two vertex objects that correspond to its endpoints, but that vertices do not refer to incident edges.

Adjacency List

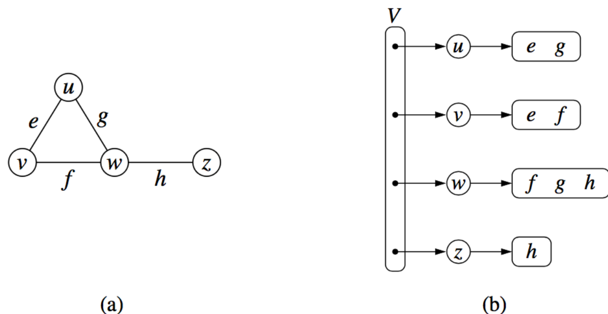


Figure 14.5: (a) An undirected graph G; (b) a schematic representation of the adjacency list structure for G. Collection V is the primary list of vertices, and each vertex has an associated list of incident edges. Although not diagrammed as such, we presume that each edge of the graph is represented with a unique Edge instance that maintains references to its endpoint vertices.

Adjacency Map

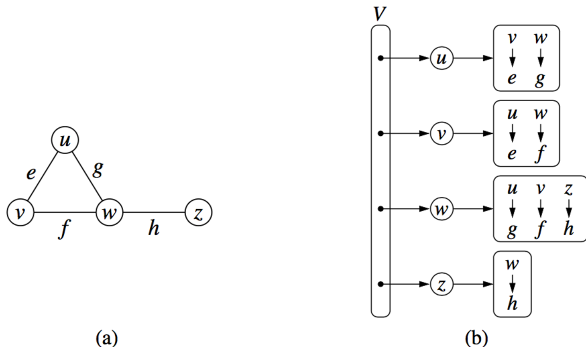
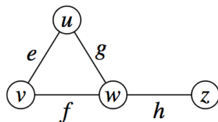


Figure 14.6: (a) An undirected graph G; (b) a schematic representation of the adjacency map structure for G. Each vertex maintains a secondary map in which neighboring vertices serve as keys, with the connecting edges as associated values. Although not diagrammed as such, we presume that there is a unique Edge instance for each edge of the graph, and that it maintains references to its endpoint vertices.

Adjacency Matrix



(a)

	0	1	2	3
u → 0		e	g	
v → 1	e		f	
w → 2	g	f		h
z → 3			h	

(b)

Figure 14.7: (a) An undirected graph G; (b) a schematic representation of the auxiliary adjacency matrix structure for G, in which n vertices are mapped to indices 0 to $n-1$. Although not diagrammed as such, we presume that there is a unique Edge instance for each edge, and that it maintains references to its endpoint vertices. We also assume that there is a secondary edge list (not pictured), to allow the edges( ) method to run in $O(m)$ time, for a graph with m edges.

Data Structures for Graphs

Python Implementation

Python Implementation

```
class Graph:
  class Vertex:
    __slots__ = '_element'

    def __init__(self, x):
      self._element = x

    def element(self):
      return self._element

    def __hash__(self):
      return hash(id(self))

    def __str__(self):
      return str(self._element)
```

Python Implementation

```
class Edge:
  __slots__ = '_origin', '_destination', '
  _element'

  def __init__(self, u, v, x):
    self._origin = u
    self._destination = v
    self._element = x

  def endpoints(self):
    return (self._origin, self._destination)
```

Python Implementation

```
def opposite(self, v):
  if not isinstance(v, Graph.Vertex):
    raise TypeError('v must be a Vertex')
  return self._destination if v is self.
  _origin else self._origin
  raise ValueError('v not incident to edge')
```

Python Implementation

```
def element(self):
  return self._element

def __hash__(self):
  return hash( (self._origin, self.
  _destination) )

def __str__(self):
  return '({0},{1},{2})'.format(self._origin
  ,self._destination,self._element)
```

Python Implementation

```
def __init__(self, directed=False):
  self._outgoing = {}
  self._incoming = {} if directed else self.
  _outgoing

def _validate_vertex(self, v):
  if not isinstance(v, self.Vertex):
    raise TypeError('Vertex expected')
  if v not in self._outgoing:
    raise ValueError('Vertex does not belong
    to this graph.')
```

Python Implementation

```python
def is_directed(self):
  return self._incoming is not self._outgoing

def vertex_count(self):
  return len(self._outgoing)

def vertices(self):
  return self._outgoing.keys()
```

Python Implementation

```
def edge_count(self):
  total = sum(len(self._outgoing[v]) for v in
  self._outgoing)
  return total if self.is_directed() else
  total // 2

def edges(self):
  result = set()
  for secondary_map in self._outgoing.values()
  :
    result.update(secondary_map.values())
  return result
```

Python Implementation

```
def get_edge(self, u, v):
  self._validate_vertex(u)
  self._validate_vertex(v)
  return self._outgoing[u].get(v)

def degree(self, v, outgoing=True):
  self._validate_vertex(v)
  adj = self._outgoing if outgoing else self.
  _incoming
  return len(adj[v])
```

Python Implementation

```
def incident_edges(self, v, outgoing=True):
  self._validate_vertex(v)
  adj = self._outgoing if outgoing else self.
  _incoming
  for edge in adj[v].values():
    yield edge

def insert_vertex(self, x=None):
  v = self.Vertex(x)
  self._outgoing[v] = {}
  if self.is_directed():
    self._incoming[v] = {}
  return v
```

Python Implementation

```
def insert_edge(self, u, v, x=None):
  if self.get_edge(u, v) is not None:
    raise ValueError('u and v are already
    adjacent')
  e = self.Edge(u, v, x)
  self._outgoing[u][v] = e
  self._incoming[v][u] = e
```

Graph Traversals

Definition : Graph Traversals

A traversal is a systematic procedure for exploring a graph by examining all of its vertices and edges.

Graph Traversals

Definition : Graph Traversals

A traversal is a systematic procedure for exploring a graph by examining all of its vertices and edges.

A traversal is efficient if it visits all the vertices and edges in time proportional to their number, that is, in linear time.

Graph Traversals

Graph traversal algorithms are key to answering many fundamental questions about graphs involving the notion of reachability, that is, in determining how to travel from one vertex to another while following paths of a graph.

Graph Traversals

Interesting problems that deal with reachability in an undirected graph G include the following:

- Computing a path from vertex u to vertex v, or reporting that no such path exists.
- Given a start vertex s of G, computing, for every vertex v of G, a path with the minimum number of edges between s and v, or reporting that no such path exists.
- Testing whether G is connected.
- Computing a spanning tree of G, if G is connected.
- Computing the connected components of G.
- Computing a cycle in G, or reporting that G has no cycles.

Graph Traversals

Interesting problems that deal with reachability in a directed graph $\vec{G}$ include the following:

- Computing a directed path from vertex u to vertex v, or reporting that no such path exists.
- Finding all the vertices of $\vec{G}$ that are reachable from a given vertex s.
- Determine whether $\vec{G}$ is acyclic.
- Determine whether $\vec{G}$ is strongly connected.

Graph Traversals

Depth-First Search

Depth-First Search

Depth-first search (DFS) is useful for testing a number of properties of graphs, including whether there is a path from one vertex to another and whether or not a graph is connected.

Depth-First Search

Procedure of DFS

1. Begin at a specific starting vertex s in G, and set s as "visited". The vertex s is now our "current" vertex - call our current vertex u.
2. Traverse G by considering an edge (u, v) incident to the current vertex u.
   - If (u, v) leads to a visited vertex v, ignore that edge.
   - If (u, v) leads to an unvisited vertex v, go to v.
   - Set v as "visited", and make it the current vertex, repeated the computation above.
   - Eventually, we will get to a "dead end", that is, a current vertex v such that all the edges incident to v lead to visited vertices.
   - To get out of this impasse, we backtracking along the edge that brought to v, going back to a previously visited vertex u.
   - We then make u our current vertex and repeat the computation above for any edges incident to u that we have not yet considered.

Depth-First Search

- If all of u's incident edges lead to visited vertices, then we backtrack to the vertex we came from to get to u, and repeat the procedure at that vertex.
- Thus, we continue to backtrack along the path that we have traced so far until we find a vertex that has yet unexplored edges, take one such edge, and continue the traversal.
- The process terminates when our backtracking leads us back to the start vertex u, and there are no more unexplored edges incident to s.

Depth-First Search

Algorithm DFS(G,u): {We assume u has already been marked as visited}

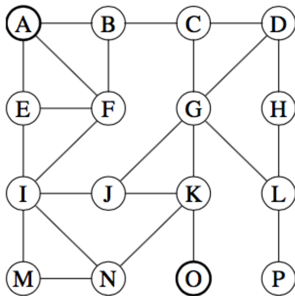
Input: A graph G and a vertex u of G

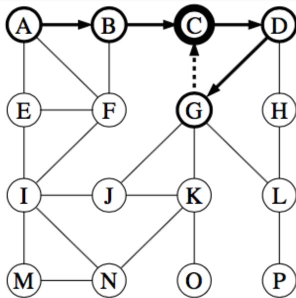
Output: A collection of vertices reachable from u, with their discovery edges

for each outgoing edge $e = (u, v)$ of u **do**
  if vertex v has not been visited **then**
    Mark vertex v as visited (via edge e).
    Recursively call DFS(G,v).

Code Fragment 14.4: The DFS algorithm.

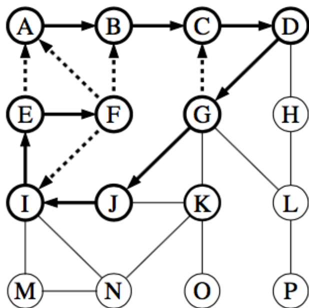
Depth-First Search

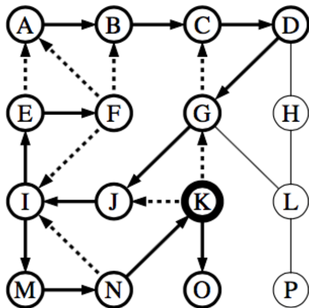




1: Example of depth-first search traversal on an undirected graph starting at vertex A. Assume that a vertex's adjacencies are considered in alphabetical order.

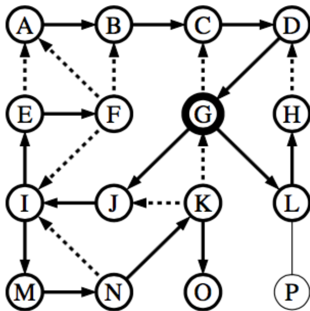
Depth-First Search



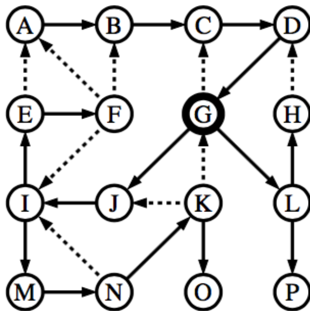


2: Example of depth-first search traversal on an undirected graph starting at vertex *A*. Assume that a vertex's adjacencies are considered in alphabetical order.

Depth-First Search



(e)



(f)

3: Example of depth-first search traversal on an undirected graph starting at vertex A. Assume that a vertex's adjacencies are considered in alphabetical order.

Properties of DFS

Proposition

Let G be an undirected graph on which a DFS traversal starting at a vertex s has been performed. Then the traversal visits all vertices in the connected component of s, and the discovery edges form a spanning tree of connected component of s.

Properties of DFS

Proposition

Let $\vec{G}$ be an directed graph. DFS on $\vec{G}$ starting at a vertex s visits all the vertices of $\vec{G}$ that are reachable from s. Also, the DFS tree contains directed paths from s to every vertex reachable from s.

Graph Traversals

DFS Implementation and Extensions

DFS Implementation and Extensions

```
def DFS(g, u, discovered):
  for e in g.incident_edges(u):
    v = e.opposite(u)
    if v not in discovered:
      discovered[v] = e
      DFS(g, v, discovered)
```

DFS Implementation and Extensions

```
def DFS(g, u, discovered):
  for e in g.incident_edges(u):
    v = e.opposite(u)
    if v not in discovered:
      discovered[v] = e
      DFS(g, v, discovered)
```

- In order to track which vertices have been visited, and to build a representation of the resulting DFS tree, the implementation introduces a third parameter, named `discovered`.
- This parameter should be a Python dictionary that maps a vertex to the tree edge that was used to discover that vertex.

DFS Implementation and Extensions

As a technicality, assume that the source vertex u occurs as a key of the dictionary, with `None` as its value.

DFS Implementation and Extensions

As a technicality, assume that the source vertex u occurs as a key of the dictionary, with `None` as its value.

A caller might start the traversal as follows:

```
result = {u: None}
DFS(g, u, result)
```

DFS Implementation and Extensions

The dictionary `result` serves two purposes.

- Internally, the dictionary provides a mechanism for recognizing visited vertices, as they will appear as keys in the dictionary.
- Externally, the DFS function arguments this dictionary as it proceeds, and thus the values within the dictionary are the DFS tree edges at the conclusion of the process.

Reconstructing a Path from u to v

We can use the basic DFS function as a tool to identify the (directed) path leading from vertex u to v, if v is reachable from u. This path can easily be reconstructed from the information that was recorded in the `discovery` dictionary during the traversal.

Reconstructing a Path from u to v

To reconstruct the path, excecute the following steps:

1. Begin at the end of the path, examining the discovery dictionary to determine what edge was used to reach a vertex v, and then what the other endpoint of that edge is.
2. Add that vertex to a list, and the repeat the process to determine what edge was used to discover it.
3. Once we have traced the path all the way back to the starting vertex u, we can reverse the list so that its is properly oriented from u to v, and return it to the caller.

Reconstructing a Path from u to v

```
def construct_path(u, v, discovered):
  path = []
  if v in discovered:
    path.append(v)
    walk = v
    while walk is not u:
      e = discovered[walk]
      parent = e.opposite(walk)
      path.append(parent)
      walk = parent
    path.reverse()
  return path
```

Computing all Connected Components

When a graph is not connected, the next goal we may have is to identify all of connected components of an undirected graph, or the strongly connected components of a directed graph.

Computing all Connected Components

When a graph is not connected, the next goal we may have is to identify all of connected components of an undirected graph, or the strongly connected components of a directed graph.

If an initial call to DFS fails to reach all vertices of a graph, we can restart a new call to DFS at one of those unvisited vertices.

Computing all Connected Components

```
def DFS_complete(g):
  forest = {}
  for u in g.vertices():
    if u not in forest:
      forest[u] = None
      DFS(g, u, forest)
  return forest
```

Graph Traversals

Breadth-First Search

Breadth-First Search

The advancing and backtracking of a depth-first search defines a traversal that could be physically traced by a single person exploring a graph.

Breadth-First Search

The advancing and backtracking of a depth-first search defines a traversal that could be physically traced by a single person exploring a graph.

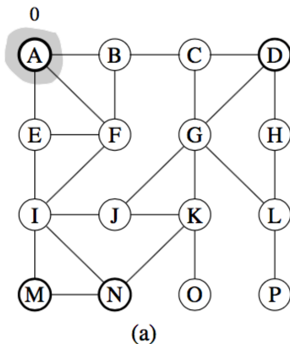
Now, we consider another algorithm for traversing a connected component of a graph, known as a breadth-first search (BFS).

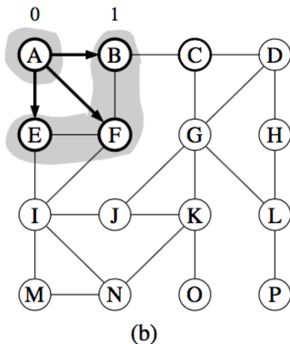
Breadth-First Search

A BFS proceeds in rounds and subdivides the vertices into levels. BFS starts at vertex s, which is at level 0.

- In the first round, we set all vertices adjacent to the start vertex s as "visited". These vertices are one step away from the beginning and are placed into level 1.
- In the second round, we allow all explorers to go two steps away from the starting vertex. These new vertices, which are adjacent to level 1 vertices and not previously assigned to a level, are placed into level 2 and marked as "visited".
- This process continues in similar fashion, terminating when no new vertices are found in a level.

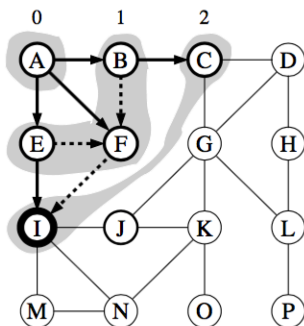
Breadth-First Search

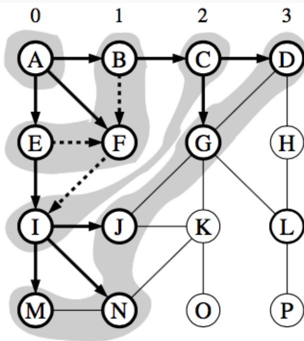




4: Example of breadth-first search traversal, where the edges incident to a vertex are considered in alphabetical order of the adjacent vertices.

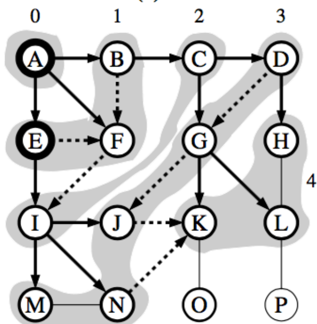
Breadth-First Search

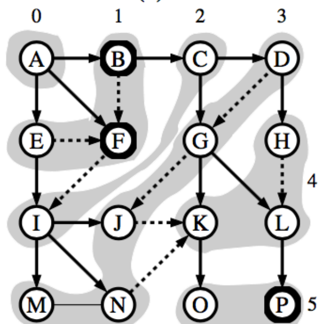




5: Example of breadth-first search traversal, where the edges incident to a vertex are considered in alphabetical order of the adjacent vertices.

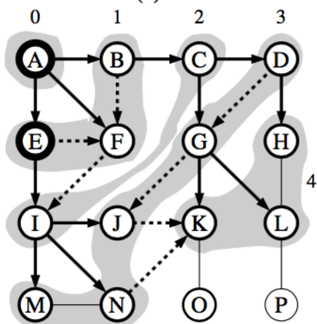
Breadth-First Search

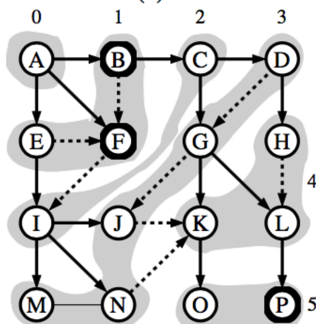




6: Example of breadth-first search traversal, where the edges incident to a vertex are considered in alphabetical order of the adjacent vertices.

Breadth-First Search





7: Example of breadth-first search traversal, where the edges incident to a vertex are considered in alphabetical order of the adjacent vertices.

Breadth-First Search

```
def BFS(g, s, discovered):
  level = [s]
  while len(level) > 0:
    next_level = []
    for u in level:
      for e in g.incident_edges(u):
        v = e.opposite(u)
        if v not in discovered:
          discovered[v] = e
          next_level.append(v)
    level = next_level
```

Breadth-First Search

```
def BFS_complete(g):
  forest = {}
  for u in g.vertices():
    if u not in forest:
      forest[u] = None
      BFS(g, u, forest)
  return forest
```