Data Structures in Python

12. Graphs

Prof. Moheb Ramzy Girgis
Department of Computer Science
Faculty of Science
Minia University

Graphs

- A ***graph*** is a way of representing relationships that exist between pairs of objects.
- That is, a ***graph*** is a set of objects, called ***vertices***, together with a collection of pairwise connections between them, called ***edges***.
- Graphs have applications in modeling many domains, including mapping, transportation, computer networks, and electrical engineering.
- Viewed abstractly, a ***graph G*** is simply a set ***V*** of ***vertices*** and a collection ***E*** of *pairs of vertices* from ***V***, called ***edges***.
- Thus, a graph is a way of representing connections or relationships between pairs of objects from some set *V*.
- ***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*.

Graphs

- An edge *(u,v)* is said to be ***undirected*** if the pair *(u,v)* is not ordered.
- If all the edges in a graph are ***undirected***, then we say the graph is an ***undirected graph***.
- In undirected graphs, edge *(u,v)* is the same as edge *(v,u)*.
- Likewise, a ***directed graph***, also called a ***digraph***, is a graph whose edges are all ***directed***.
- Graphs are typically visualized by drawing the vertices as ovals or rectangles and the edges as segments or curves connecting pairs of ovals and rectangles.
- **Example:** We can visualize collaborations among the researchers of a certain discipline by constructing a graph whose vertices are associated with the researchers themselves, and whose edges connect pairs of vertices associated with researchers who have coauthored a paper or book. (See the following figure)

3

Graphs

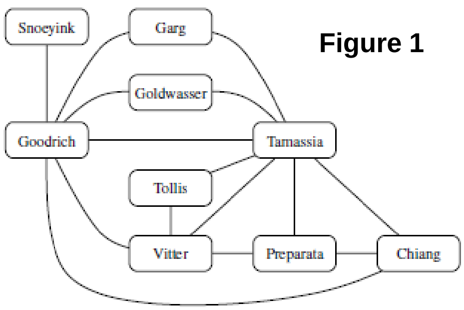


Figure 1

- Such edges are ***undirected*** because coauthorship is a ***symmetric*** relation; that is, if *A* has coauthored something with *B*, then *B* necessarily has *A*.
- 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*.
- 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.

4

Graphs

- The ***incoming edges*** of a vertex are the directed edges whose destination is that vertex.
- 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.
- Example: A directed graph representing a flight network
- A ***flight network***, is a graph *G* whose vertices are associated with airports, and whose edges are associated with flights, as shown in Figure 2.
- In graph *G*, the edges are ***directed*** because a given flight has a specific travel direction.

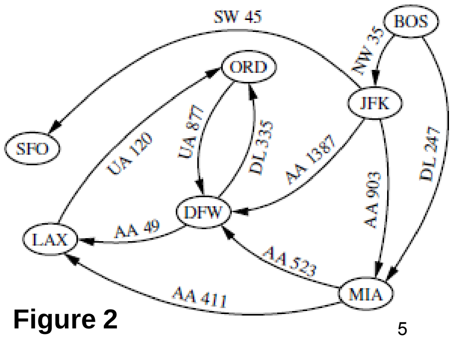


Figure 2

Graphs

- The *endpoints* of an *edge e* in *G* correspond respectively to the *origin* and *destination* of the flight corresponding to *e*.
- Two airports are ***adjacent*** in *G* if there is a flight that flies between them, and an edge *e* is ***incident*** to a vertex *v* in *G* if the flight for *e* flies to or from the airport for *v*.
- The ***outgoing edges*** of a vertex *v* correspond to the outbound flights from *v*'s airport, and the ***incoming edges*** correspond to the inbound flights to *v*'s airport.
- Finally, the ***in-degree*** of a vertex *v* of *G* corresponds to the number of inbound flights to *v*'s airport, and the ***out-degree*** of a vertex *v* in *G* corresponds to the number of outbound flights.
- For example, in Figure 2, the ***endpoints*** of edge UA 120 are LAX and ORD; hence, LAX and ORD are ***adjacent***. The ***in-degree*** of DFW is 3, and the ***out-degree*** of DFW is 2.

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***.
- A flight network can contain ***parallel edges***, such that ***multiple edges*** between the same pair of vertices could indicate different flights operating on the same route at different times of the day.
- Another special type of edge is one that connects a vertex to itself. Namely, we say that an edge (***undirected*** or ***directed***) is a ***self-loop*** if its two endpoints coincide.
- 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***).

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.
- A ***path*** is ***simple*** if each vertex in the path is distinct.
- A ***cycle*** is ***simple*** if each vertex in the cycle is distinct, except for the first and last one.
- A ***directed path*** is a path such that all edges are directed and are traversed along their direction.
- A ***directed cycle*** is similarly defined.
- For example, in Figure 2, (BOS, NW35, JFK, AA 1387, DFW) is a ***directed simple path***, and (LAX, UA 120, ORD, UA 877, DFW, AA 49, LAX) is a ***directed simple cycle***.
- Note that a directed graph may have a ***cycle*** consisting of two edges with opposite direction between the same pair of vertices, for example (ORD, UA 877, DFW, DL 335, ORD) in Figure 2.

Graphs

- A ***directed graph*** is ***acyclic*** if it has no directed cycles. For example, if we were to remove the edge UA 877 from the graph in Figure 2, the remaining graph is acyclic.
- 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*.
- 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*.
- A ***tree*** is a connected graph without cycles.
- A ***spanning tree*** of a graph is a ***spanning subgraph*** that is a ***tree***.

The Graph ADT

- A ***graph*** is a collection of ***vertices*** and ***edges***. We model the abstraction as a combination of three data types: ***Vertex***, ***Edge***, and ***Graph***.
- A ***Vertex*** is a lightweight object that stores an arbitrary element provided by the user (e.g., an airport code); and it supports a method, ***element()***, to retrieve the stored element.
- An ***Edge*** also stores an associated object (e.g., a flight number, travel distance, cost), retrieved with the ***element()*** method, and it supports the following methods:
  - ***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.
- The **Graph ADT** includes the following methods:
  - ***vertex_count()*****:** Return the number of vertices of the graph.
  - ***vertices()*****:** Return an iteration of all the vertices of the graph.
  - ***edge_count():*** Return the number of edges of the graph.
  - ***edges()*****:** Return an iteration of all the edges of the graph.
  - ***get_edge(u,v)*****:** Return the edge from vertex *u* to vertex *v*, if one exists; otherwise return None.
    - For an ***undirected graph***, there is no difference between ***get_edge(u,v)*** and ***get_edge(v,u)***.
  - ***degree(v, out=True)*****:**
    - For an ***undirected graph***, return the number of edges incident to vertex *v*.

The Graph ADT

- - For a ***directed graph***, return the number of ***outgoing*** (resp. ***incoming***) edges incident to vertex *v*, as designated by the optional parameter ***True*** (***False***).
  - ***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** (i.e., ***undirected graph***).
  - ***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

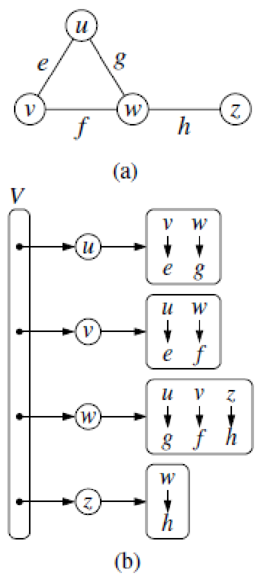
- Following are *four data structures* for representing a graph.
- In each representation, we maintain a collection to store the vertices of a graph.
- However, the four representations differ greatly in the way they organize the edges.
  - In an ***edge list***, we maintain an unordered list of all edges. This minimally suffices, but there is no efficient way to locate a particular edge *(u,v)*, or the set of all edges incident to a vertex *v*.
  - In an ***adjacency list***, we maintain, for each vertex, a separate list containing those edges that are incident to the vertex. The complete set of edges can be determined by taking the union of the smaller sets, while the organization allows us to more efficiently find all edges incident to a given vertex.

Data Structures for Graphs

  - An ***adjacency map*** is very similar to an adjacency list, but the secondary container of all edges incident to a vertex is organized as a ***map***, rather than as a ***list***, with the ***adjacent vertex*** serving as a ***key***. This allows for access to a specific edge *(u,v)* in $O(1)$ expected time.
  - An ***adjacency matrix*** provides worst-case $O(1)$ access to a specific edge *(u,v)* by maintaining an $n \times n$ matrix, for a graph with *n* vertices. Each entry is dedicated to storing a reference to the edge *(u,v)* for a particular pair of vertices *u* and *v*; if no such edge exists, the entry will be **None**.

Python Implementation of The Graph ADT

- Now, we provide an implementation of the **Graph ADT**.
- This implementation will support ***directed*** or ***undirected graphs***.
- We use a variant of the ***adjacency map*** representation.
- For each vertex *v*, we use a Python **dictionary** to represent the secondary ***incidence map I(v)***.
- Figure (a) shows an ***undirected graph*** *G*; Figure (b) shows 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***.



Python Implementation of The Graph ADT

- We do not explicitly maintain lists *V* and *E*.
- The ***list V*** is replaced by a top-level ***dictionary D*** that maps each ***vertex v*** to its ***incidence map I(v)***.
- We can iterate through all vertices by generating the set of keys for dictionary *D*. By using such a dictionary *D* to map vertices to the secondary incidence maps, we need not maintain references to those incidence maps as part of the vertex structures.
- Our implementation of the **graph ADT** is given below.
- **Classes Vertex** and **Edge** are nested within the **Graph class**.
- We define the ***hash method*** for both **Vertex** and **Edge** so that those instances can be used as keys in Python's ***hash-based sets*** and ***dictionaries***.

Python Implementation of The Graph ADT

- Graphs are ***undirected*** by default, but can be declared as ***directed*** with an optional parameter to the ***constructor***.
- Internally, we manage the directed case by having two different top-level ***dictionary*** instances, ***_outgoing*** and ***_incoming***, such that **_outgoing[v]** maps to another dictionary representing **Iout(v)**, and **_incoming[v]** maps to a representation of **Iin(v)**.
- In order to unify our treatment of ***directed*** and ***undirected*** graphs, we continue to use the ***outgoing*** and ***incoming*** identifiers in the ***undirected*** case, yet as *aliases to the same dictionary*.
- For convenience, we define a utility named ***is_directed*** to allow us to distinguish between the two cases.

Python Implementation of The Graph ADT

- For methods ***degree*** and ***incident_edges***, which each accept an optional parameter to differentiate between the outgoing and incoming orientations, we choose the appropriate map before proceeding.
- For method ***insert_vertex***, we always initialize outgoing[v] to an empty dictionary for new vertex *v*.
  - In the directed case, we independently initialize incoming[v] as well.
  - For the undirected case, that step is unnecessary as outgoing and incoming are aliases.
- We leave the implementations of methods ***remove_vertex*** and ***remove_edge*** as exercises.
- Following is the code for **Graph class**, which includes the nested **Vertex** and **Edge classes**:

Python Implementation of The Graph ADT

```
class Graph:
 """Representation of a simple graph using an adjacency map."""

 #------------------------- nested Vertex class -------------------------
 class Vertex:
  """Lightweight vertex structure for a graph."""
  slots = '_element'
  def __init__(self, x):
   """Do not call constructor directly. Use Graph's insert_vertex(x)."""
   self._element = x
   def element(self):
    """Return element associated with this vertex."""
    return self._element
   def __hash__(self): # will allow vertex to be a map/set key
    return hash(id(self))
 #------------------------- nested Edge class -------------------------
 class Edge:
  """Lightweight edge structure for a graph."""
  slots = '_origin', '_destination', '_element'
  def __init__(self, u, v, x):
   """Do not call constructor directly. Use Graph's insert _edge(u,v,x)."""
   self._origin = u
   self._destination = v
   self._element = x
```


Python Implementation of The Graph ADT

```
  def endpoints(self):
   """Return (u,v) tuple for vertices u and v."""
   return (self._origin, self._destination)
  def opposite(self, v):
   """Return the vertex that is opposite v on this edge."""
   return self._destination if v is self._origin else self._origin
  def element(self):
   """Return element associated with this edge."""
   return self._element
  def __hash__(self): # will allow edge to be a map/set key
   return hash( (self._origin, self._destination) )

 #------------------------- Graph class -------------------------
 def __init__(self, directed=False):
  """Create an empty graph (undirected, by default).
  Graph is directed if optional paramter is set to True.
  """
  self._outgoing = { }
  # only create second map for directed graph; use alias for undirected
  self._incoming = { } if directed else self._outgoing
 def is_directed(self):
  """Return True if this is a directed graph; False if undirected.
  Property is based on the original declaration of the graph, not its contents.
  """
  return self._incoming is not self._outgoing # directed if maps are distinct
```

Note: **For undirected graph**, *incoming* and *outgoing* refer to the same map, but **for directed graph**, they refer to different maps.

Python Implementation of The Graph ADT

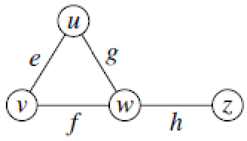
```
def vertex_count(self):
 """Return the number of vertices in the graph."""
 return len(self._outgoing)
def vertices(self):
 """Return an iteration of all vertices of the graph."""
 return self._outgoing.keys( )
def edge_count(self):
 """Return the number of edges in the graph."""
 total = sum(len(self._outgoing[v]) for v in self._outgoing)
 # for undirected graphs, make sure not to double-count edges
 return total if self.is_directed( ) else total // 2
def edges(self):
 """Return a set of all edges of the graph."""
 result = set( ) # avoid double-reporting edges of undirected graph
 for secondary_map in self._outgoing.values():
  result.update(secondary_map.values()) # add edges to resulting set
 return result
def get_edge(self, u, v):
 """Return the edge from u to v, or None if not adjacent."""
 return self._outgoing[u].get(v) # returns None if v not adjacent
```


Python Implementation of The Graph ADT

```
 def degree(self, v, outgoing=True):
 """Return number of (outgoing) edges incident to vertex v in the graph.
 If graph is directed, optional parameter used to count incoming edges.
 """
 adj = self._outgoing if outgoing else self._incoming
 return len(adj[v])
def incident_edges(self, v, outgoing=True):
 """Return all (outgoing) edges incident to vertex v in the graph.
 If graph is directed, optional parameter used to request incoming edges.
 """
 adj = self. outgoing if outgoing else self._incoming
 for edge in adj[v].values( ):
  yield edge
def insert_vertex(self, x=None):
 """Insert and return a new Vertex with element x."""
 v = self.Vertex(x)
 self._outgoing[v] = { }
 if self.is_directed( ):
  self._incoming[v] = { } # need distinct map for incoming edges
 return v
def insert_edge(self, u, v, x=None):
 """Insert and return a new Edge from u to v with auxiliary element x."""
 e = self.Edge(u, v, x)
 self._outgoing[u][v] = e
 self._incoming[v][u] = e
```

Python Implementation of The Graph ADT

- **Example:** The following code exercises the graph operations for the shown undirected grap




```
from Graph import Graph
# Test undirected graph operations
G = Graph()
u = G.insert_vertex('u')
v = G.insert_vertex('v')
w = G.insert_vertex('w')
z = G.insert_vertex('z')
G.insert_edge(u,v,'e')
G.insert_edge(u,w,'g')
G.insert_edge(v,w,'f')
G.insert_edge(w,z,'h')
print('Degree of vertex u:', G.degree(u,False))
print('Degree of vertex v:', G.degree(v,False))
print('Degree of vertex w:', G.degree(w,False))
print('Degree of vertex z:', G.degree(z,False))
print('is directed?', G.is_directed())
print('All (outgoing) edges incident to u:', end=' ')
for edge in G.incident_edges(u):
   print(edge.element(), end=' ')
print()
```


Python Implementation of The Graph ADT

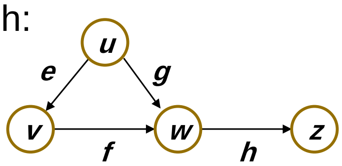
```
print('All (outgoing) edges incident to v:', end=' ')
for edge in G.incident_edges(v):
   print(edge.element(), end=' ')
print()
print('All (outgoing) edges incident to w:', end=' ')
for edge in G.incident_edges(w):
   print(edge.element(), end=' ')
print()
print('All (outgoing) edges incident z:', end=' ')
for edge in G.incident_edges(z):
   print(edge.element(), end=' ')
print()
print('Edge (u,v) is:', G.get_edge(u, v).element())
print('No. of Edges:', G.edge_count())
print('No. of vertices:', G.vertex_count())
print('The vertices are:', end=' ')
for vertex in G.vertices():
   print(vertex.element(), end=' ')
print()
print('All edges', end=' ')
for edge in G.edges():
   print(edge.element(), end=' ')
print()
```

Python Implementation of The Graph ADT

Output

```
Degree of vertex u: 2
Degree of vertex v: 2
Degree of vertex w: 3
Degree of vertex z: 1
Is directed? False
All (outgoing) edges incident to u: e g
All (outgoing) edges incident to v: e f
All (outgoing) edges incident to w: g f h
All (outgoing) edges incident to z: h
Edge (u,v) is: e
No. of Edges: 4
No. of vertices: 4
The vertices are: u v w z
All edges f e h g
```

- **Example:** The following code exercises the graph operations for the shown directed graph:




```
from Graph import Graph
# Test directed graph operations
G = Graph(True)
u = G.insert_vertex('u')
v = G.insert_vertex('v')
w = G.insert_vertex('w')
z = G.insert_vertex('z')
```


Python Implementation of The Graph ADT

```
G.insert_edge(u,v,'e')
G.insert_edge(u,w,'g')
G.insert_edge(v,w,'f')
G.insert_edge(w,z,'h')
print('Degree of vertex u:', G.degree(u))
print('Degree of vertex v:', G.degree(v))
print('Degree of vertex w:', G.degree(w))
print('Degree of vertex z:', G.degree(z))
print('is_directed?', G.is_directed())
print('All (outgoing) edges incident to u:', end=' ')
for edge in G.incident_edges(u):
 print(edge.element(), end=' ')
print()
print('All (outgoing) edges incident to v:', end=' ')
for edge in G.incident_edges(v):
    print(edge.element(), end=' ')
print()
print('All (outgoing) edges incident to w:', end=' ')
for edge in G.incident_edges(w):
    print(edge.element(), end=' ')
print()
```

Python Implementation of The Graph ADT

```
print('All (outgoing) edges incident to z:', end=' ')
for edge in G.incident_edges(z):
  print(edge.element(), end=' ')
print()
print('Edge (u,v) is:', G.get_edge(u, v).element())
print('No. of Edges:', G.edge_count())
print('No. of vertices:', G.vertex_count())
print('The vertices are:', end=' ')
for vertex in G.vertices():
   print(vertex.element(), end=' ')
print()
print('All edges', end=' ')
for edge in G.edges():
   print(edge.element(), end=' ')
print()
```

Output

```
Degree of vertex u: 2
Degree of vertex v: 1
Degree of vertex w: 1
Degree of vertex z: 0
Is directed? True
All (outgoing) edges incident to u: e g
All (outgoing) edges incident to v: f
All (outgoing) edges incident to w: h
All (outgoing) edges incident to z:
Edge (u,v) is: e
No. of Edges: 4
No. of vertices: 4
The vertices are: u v w z
All edges g h e f
```


Python Implementation of The Graph ADT

- The following code displays all edges for the given undirected and directed graph:

```
for u in G.vertices():
 for v in G.vertices():
  if G.get_edge(u, v) != None:
   print('Edge (', u.element(), ',', v.element(), ') is:', G.get_edge(u, v).element())
```

Undirected Graph

```
Edge ( u , v ) is: e
Edge ( u , w ) is: g
Edge ( v , u ) is: e
Edge ( v , w ) is: f
Edge ( w , u ) is: g
Edge ( w , v ) is: f
Edge ( w , z ) is: h
Edge ( z , w ) is: h
```

Directed Graph

```
Edge ( u , v ) is: e
Edge ( u , w ) is: g
Edge ( v , w ) is: f
Edge ( w , z ) is: h
```

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.
- 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*** *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 *G* that are reachable from a given vertex *s*.
  - Determine whether *G* is acyclic.
  - Determine whether *G* is strongly connected.
- In the remainder of this lecture, we present two efficient graph traversal algorithms, called ***depth-first search*** and ***breadth-first search***, respectively.

Depth-First Search (DFS)

- The ***depth-first search*** of a graph first visits a vertex, then recursively visits all vertices adjacent to that vertex.
- The graph may contain cycles, which may lead to an infinite recursion. To avoid this problem, you need to track the vertices that have already been visited.
- The search is called ***depth-first***, because it searches "deeper" in the graph as much as possible.
- The search starts from some vertex v. After visiting v, it next visits the first unvisited neighbor of v. If v has no unvisited neighbor, backtrack to the vertex from which we reached v.
- The pseudo-code for a ***depth-first search traversal*** starting at a vertex *u:*

Depth-First Search (DFS)

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 visiting 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)

DFS Implementation

- Now, we provide a Python implementation of the basic ***depth-first search algorithm***, described with above pseudo-code.
- The code for recursive ***DFS function*** that implements ***depth-first search*** on a graph, starting at a designated vertex *u* is presented below.

Depth-First Search (DFS)

```
def DFS(g, u, visited):
 """Perform DFS of the unvisited portion of Graph g starting at Vertex u.
 visited is a dictionary mapping each vertex to the edge that was used to
 discover it during the DFS. (u should be "visited" prior to the call.)
 Newly visited vertices will be added to the dictionary as a result.
 """
 for e in g.incident_edges(u): # for every outgoing edge from u
  v = e.opposite(u)
  if v not in visited: # v is an unvisited vertex
   visited[v] = e    # e is the tree edge that visited v
   DFS(g, v, visited) # recursively explore from v
```

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

Depth-First Search (DFS)

- We assume that the source vertex *u* occurs as a ***key*** of the ***dictionary***, with **None** as its ***value***.
- Thus, a caller might start the traversal as follows:

```
visited = {u : None}      # a new dictionary, with u trivially discovered
DFS(g, u, visited)
```

- The ***dictionary*** 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*** augments this dictionary as it proceeds, and thus the values within the dictionary are the ***DFS tree*** edges at the conclusion of the process.

Depth-First Search (DFS)

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 ***visiting dictionary*** during the traversal.
- The following code fragment provides an implementation of a secondary function that produces an ordered list of vertices on the path from *u* to *v*.
- To reconstruct the path, we begin at the *end* of the path, examining the ***visiting dictionary*** to determine what edge was used to reach vertex *v*, and then what the other endpoint of that edge is.

Depth-First Search (DFS)

- We add that vertex to a list, and then repeat the process to determine what edge was used to discover it.
- Once we have traced the path all the way back to the starting vertex *u*, we can reverse the list so that it is properly oriented from *u* to *v*, and return it to the caller.

```
def construct_path(u, v, visited):
 path = [ ]          # empty path by default
 if v in visited:
  # we build list from v to u and then reverse it at the end
  path.append(v)
  walk = v
  while walk is not u:
   e = visited[walk]          # find edge leading to walk
   parent = e.opposite(walk)
   path.append(parent)
   walk = parent
  path.reverse( )             # reorient path from u to v
 return path
```

Depth-First Search (DFS)

- **Example:**

Figure 1 shows an undirected graph, where the vertices represent cities and the edges represent roads and distances between two adjacent cities.

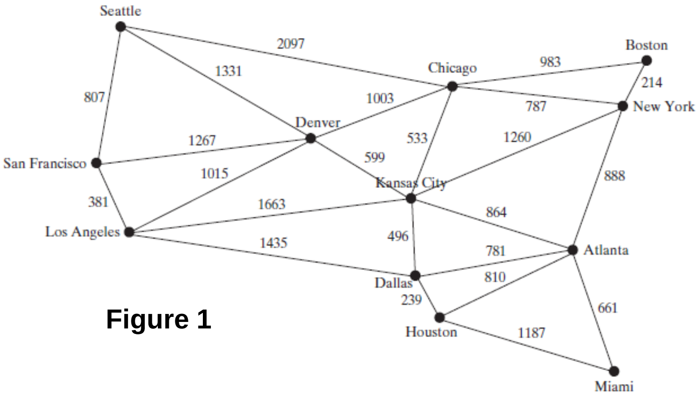


Figure 1

- The following test program displays a ***DFS*** for this graph starting from *Chicago*.

Depth-First Search (DFS)

```
def main():
 G = Graph()
 #vertices
 v0 = G.insert_vertex('Seattle')
 v1 = G.insert_vertex('San_Francisco')
 v2 = G.insert_vertex('Los_Angeles')
 v3 = G.insert_vertex('Denver')
 v4 = G.insert_vertex('Kansas_City')
 v5 = G.insert_vertex('Chicago')
 v6 = G.insert_vertex('Boston')
 v7 = G.insert_vertex('New_York')
 v8 = G.insert_vertex('Atlanta')
 v9 = G.insert_vertex('Miami')
 v10 = G.insert_vertex('Dallas')
 v11 = G.insert_vertex('Houston')
 # edges
 G.insert_edge(v0,v1)
 G.insert_edge(v0,v3)
 G.insert_edge(v0,v5)
 G.insert_edge(v1,v2)
 G.insert_edge(v1,v3)
 G.insert_edge(v2,v3)
 G.insert_edge(v2,v4)
 G.insert_edge(v2,v10)
 G.insert_edge(v3,v4)
 G.insert_edge(v3,v5)
 G.insert_edge(v4,v5)
 G.insert_edge(v4,v7)
 G.insert_edge(v4,v8)
 G.insert_edge(v4,v10)
 G.insert_edge(v5,v6)
 G.insert_edge(v5,v7)
 G.insert_edge(v6,v7)
 G.insert_edge(v7,v8)
 G.insert_edge(v8,v9)
 G.insert_edge(v8,v10)
 G.insert_edge(v8,v11)
 G.insert_edge(v9,v11)
 G.insert_edge(v10,v11)
```

Depth-First Search (DFS)

```
visited = {v5 : None} # a new dictionary, with v5 'Chicago' trivially visited
DFS(G, v5, visited)
print(G.vertex_count(), ' vertices are searched in this DFS order:')
for v in visited:
  print(v.element(), end=' ')
 print()
 print('The path followed from Chicago to Dallas')
 path = construct_path(v5, v10, visited)
 for v in path:
  print(v.element(), end=' ')
 print()
main()
```

Output

12 vertices are searched in this DFS order:
Chicago Seattle San_Francisco Los_Angeles Denver Kansas_City New_York Boston Atlanta Miami Houston Dallas

The path followed from Chicago to Dallas
Chicago Seattle San_Francisco Los_Angeles Denver Kansas_City New_York Atlanta Miami Houston Dallas

Depth-First Search (DFS)

- The graphical illustration of the ***DFS*** starting from Chicago is shown in Figure 2.

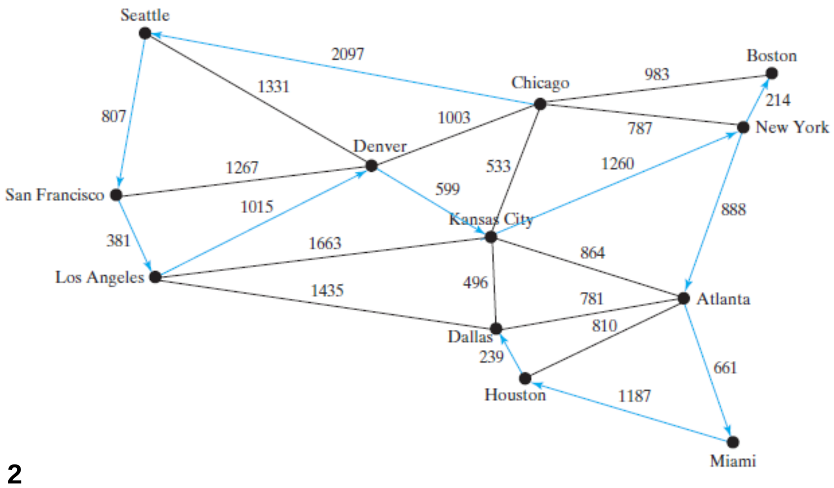


Figure 2

Breadth-First Search (BFS)

- The ***breadth-first search*** of a graph first visits a vertex, then all its adjacent vertices, then all the vertices adjacent to those vertices, and so on. To ensure that each vertex is visited only once, skip a vertex if it has already been visited.
- A Python implementation of ***BFS*** is given in the following code fragment.
- We follow a convention similar to that of ***DFS***, using a visited dictionary both to recognize visited vertices, and to record the visiting edges of the ***BFS tree***.

Breadth-First Search (BFS)

```
def BFS(g, s, visited):
 """Perform BFS of the unvisited portion of Graph g starting at Vertex s.
 visited is a dictionary mapping each vertex to the edge that was used to
 visit it during the BFS (s should be mapped to None prior to the call).
 Newly visited vertices will be added to the dictionary as a result.
 """
 level = [s] # first level includes only s
 while len(level) > 0:
  next_level = [ ] # prepare to gather newly found vertices
  for u in level:
   for e in g.incident_edges(u): # for every outgoing edge from u
    v = e.opposite(u)
    if v not in visited: # v is an unvisited vertex
     visited[v] = e # e is the tree edge that visited v
     next_level.append(v) # v will be further considered in next pass
  level = next_level # relabel 'next' level to become current
```

- The following test program displays a ***BFS*** for the graph shown in Figure 1 starting from *Chicago*.

Breadth-First Search (BFS)

```
def main():
 G = Graph()
 #vertices as in test program of DFS
 # edges as in test program of DFS
visited = {v5 : None} # a new dictionary, with v5 'Chicago' trivially visited
 BFS(G, v5, visited)
 print(G.vertex_count(), ' vertices are searched in this BFS order:')
 for v in visited:
  print(v.element(), end=' ')
 print()
 print('The path followed from Chicago to Dallas')
 path = construct_path(v5, v10, visited)
 for v in path:
  print(v.element(), end=' ')
 print()
main()
```

Output

```
12 vertices are searched in this BFS order:
Chicago Seattle Denver Kansas_City Boston New_York
San_Francisco Los_Angeles Atlanta Dallas Miami Houston

The path followed from Chicago to Dallas
Chicago Kansas_City Dallas
```


Breadth-First Search (BFS)

- The graphical illustration of the *BFS* starting from *Chicago* is shown in Figure 3.

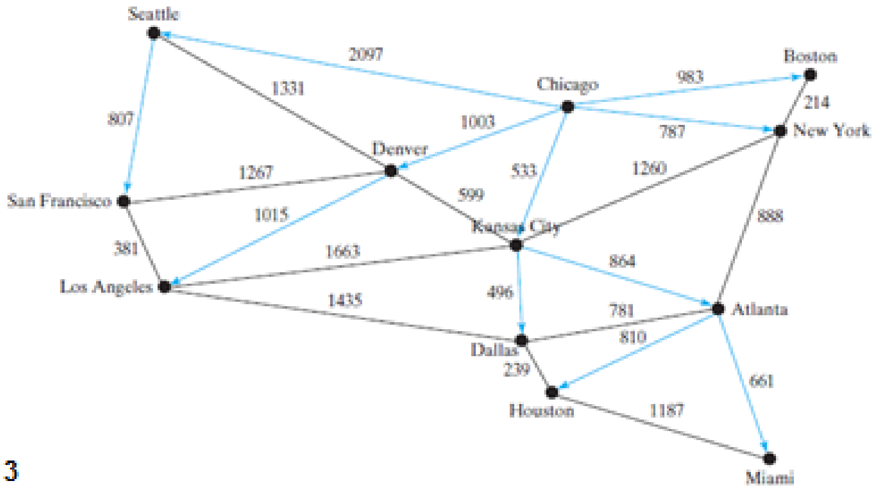


Figure 3