Lecture 23: Graph Data Structures

BT 3051 – Data Structures and Algorithms for Biology

Karthik Raman
Department of Biotechnology
Bhupat and Jyoti Mehta School of Biosciences
Indian Institute of Technology Madras

The Graph Abstract Data Type

The Graph ADT

- A graph is a collection of vertices and edges: the abstraction can be modelled as a combination of three data types: `Vertex`, `Edge`, and `Graph`

`Vertex`

- Light-weight object that stores arbitrary items provided by a user
- Needs a method such as `element()` to identify the node/vertex

`Edge`

- Stores a pair of `Vertex` elements
- In addition, can support `endpoints()` and `opposite(v)` methods

The Graph ADT

- A graph is a collection of vertices and edges: the abstraction can be modelled as a combination of three data types: `Vertex`, `Edge`, and `Graph`

`Vertex`

- Light-weight object that stores arbitrary items provided by a user
- Needs a method such as `element()` to identify the node/vertex

`Edge`

- Stores a pair of `Vertex` elements
- In addition, can support `endpoints()` and `opposite(v)` methods

The Graph ADT

- A graph is a collection of vertices and edges: the abstraction can be modelled as a combination of three data types: `Vertex`, `Edge`, and `Graph`

`Vertex`

- Light-weight object that stores arbitrary items provided by a user
- Needs a method such as `element()` to identify the node/vertex

`Edge`

- Stores a pair of `Vertex` elements
- In addition, can support `endpoints()` and `opposite(v)` methods

The Graph ADT

- A graph is a collection of vertices and edges: the abstraction can be modelled as a combination of three data types: `Vertex`, `Edge`, and `Graph`

`Vertex`

- Light-weight object that stores arbitrary items provided by a user
- Needs a method such as `element()` to identify the node/vertex

`Edge`

- Stores a pair of `Vertex` elements
- In addition, can support `endpoints()` and `opposite(v)` methods

The Graph ADT

- A graph is a collection of vertices and edges: the abstraction can be modelled as a combination of three data types: `Vertex`, `Edge`, and `Graph`

`Vertex`

- Light-weight object that stores arbitrary items provided by a user
- Needs a method such as `element()` to identify the node/vertex

`Edge`

- Stores a pair of `Vertex` elements
- In addition, can support `endpoints()` and `opposite(v)` methods

The Graph ADT

Graph

- `vertex_count ()`
- `vertices ()`
- `edge_count ()`
- `edges ()`
- `get_edge (u,v)`
- `degree (v, out=True)`
- `incident_edges (v, out=True)`
- `insert_vertex (v)`
- `insert_edge (u, v, x=None)`
- `remove_vertex (v)`
- `remove_edge (e)`

The Graph ADT

Graph

- vertex_count ()
- vertices ()
- edge_count ()
- edges ()
- get_edge (u,v)
- degree (v, out=True)
- incident_edges (v, out=True)
- insert_vertex (v)
- insert_edge (u, v, x=None)
- remove_vertex (v)
- remove_edge (e)

The Graph ADT

Graph

- `vertex_count ()`
- `vertices ()`
- `edge_count ()`
- `edges ()`
- `get_edge (u,v)`
- `degree (v, out=True)`
- `incident_edges (v, out=True)`
- `insert_vertex (v)`
- `insert_edge (u, v, x=None)`
- `remove_vertex (v)`
- `remove_edge (e)`

The Graph ADT

Graph

- vertex_count ()
- vertices ()
- edge_count ()
- edges ()
- get_edge (u,v)
- degree (v, out=True)
- incident_edges (v, out=True)
- insert_vertex (v)
- insert_edge (u, v, x=None)
- remove_vertex (v)
- remove_edge (e)

The Graph ADT

Graph

- vertex_count ()
- vertices ()
- edge_count ()
- edges ()
- get_edge (u,v)
- degree (v, out=True)
- incident_edges (v, out=True)
- insert_vertex (v)
- insert_edge (u, v, x=None)
- remove_vertex (v)
- remove_edge (e)

The Graph ADT

Graph

- `vertex_count ()`
- `vertices ()`
- `edge_count ()`
- `edges ()`
- `get_edge (u,v)`
- `degree (v, out=True)`
- `incident_edges (v, out=True)`
- `insert_vertex (v)`
- `insert_edge (u, v, x=None)`
- `remove_vertex (v)`
- `remove_edge (e)`

The Graph ADT

Graph

- `vertex_count ()`
- `vertices ()`
- `edge_count ()`
- `edges ()`
- `get_edge (u,v)`
- `degree (v, out=True)`
- `incident_edges (v, out=True)`
- `insert_vertex (v)`
- `insert_edge (u, v, x=None)`
- `remove_vertex (v)`
- `remove_edge (e)`

The Graph ADT

Graph

- `vertex_count ()`
- `vertices ()`
- `edge_count ()`
- `edges ()`
- `get_edge (u,v)`
- `degree (v, out=True)`
- `incident_edges (v, out=True)`
- `insert_vertex (v)`
- `insert_edge (u, v, x=None)`
- `remove_vertex (v)`
- `remove_edge (e)`

The Graph ADT

Graph

- vertex_count ()
- vertices ()
- edge_count ()
- edges ()
- get_edge (u,v)
- degree (v, out=True)
- incident_edges (v, out=True)
- insert_vertex (v)
- insert_edge (u, v, x=None)
- remove_vertex (v)
- remove_edge (e)

The Graph ADT

Graph

- `vertex_count ()`
- `vertices ()`
- `edge_count ()`
- `edges ()`
- `get_edge (u,v)`
- `degree (v, out=True)`
- `incident_edges (v, out=True)`
- `insert_vertex (v)`
- `insert_edge (u, v, x=None)`
- `remove_vertex (v)`
- `remove_edge (e)`

The Graph ADT

Graph

- vertex_count ()
- vertices ()
- edge_count ()
- edges ()
- get_edge (u,v)
- degree (v, out=True)
- incident_edges (v, out=True)
- insert_vertex (v)
- insert_edge (u, v, x=None)
- remove_vertex (v)
- remove_edge (e)

Data Structures for Graphs

Edge List

- Naïve representation
- We maintain an unordered list of all edges
- 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
- convenient way to store a graph in a file!

Edge List

- Naïve representation
- We maintain an unordered list of all edges
- 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
- convenient way to store a graph in a file!

Edge List

- Naïve representation
- We maintain an unordered list of all edges
- 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
- convenient way to store a graph in a file!

Edge List

- Naïve representation
- We maintain an unordered list of all edges
- 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
- convenient way to store a graph in a file!

Edge List

- Naïve representation
- We maintain an unordered list of all edges
- 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
- convenient way to store a graph in a file!

Edge List

- Naïve representation
- We maintain an unordered list of all edges
- 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
- convenient way to store a graph in a file!

Adjacency Matrix

- Let's assume $G(V, E)$, with $|V| = n$ and $|E| = m$
- We can represent G using an $n \times n$ *adjacency matrix* **A**
- $a_{ij} = 1$ iff $(i, j) \in E$, for $i, j \in V$
- How do we save space?
  - if the graph is undirected?
  - if the graph is very sparse?

Adjacency Matrix

- Let's assume $G(V, E)$, with $|V| = n$ and $|E| = m$
- We can represent G using an $n \times n$ *adjacency matrix* $\mathbf{A}$
- $a_{ij} = 1$ iff $(i, j) \in E$, for $i, j \in V$

- How do we save space?
  - if the graph is undirected?
  - if the graph is very sparse?

Adjacency Matrix

- Let's assume $G(V, E)$, with $|V| = n$ and $|E| = m$
- We can represent G using an $n \times n$ *adjacency matrix* $\mathbf{A}$
- $a_{ij} = 1$ iff $(i, j) \in E$, for $i, j \in V$

- How do we save space?
  - if the graph is undirected?
  - if the graph is very sparse?

Adjacency Matrix

- Let's assume $G(V, E)$, with $|V| = n$ and $|E| = m$
- We can represent G using an $n \times n$ *adjacency matrix* $\mathbf{A}$
- $a_{ij} = 1$ iff $(i, j) \in E$, for $i, j \in V$

- How do we save space?
  - if the graph is undirected?
  - if the graph is very sparse?

Adjacency Matrix

- Let's assume $G(V, E)$, with $|V| = n$ and $|E| = m$
- We can represent G using an $n \times n$ *adjacency matrix* $\mathbf{A}$
- $a_{ij} = 1$ iff $(i, j) \in E$, for $i, j \in V$
- How do we save space?
  - if the graph is undirected?
  - if the graph is very sparse?

Adjacency Matrix

- Let's assume $G(V, E)$, with $|V| = n$ and $|E| = m$
- We can represent G using an $n \times n$ *adjacency matrix* $\mathbf{A}$
- $a_{ij} = 1$ iff $(i, j) \in E$, for $i, j \in V$

- How do we save space?
  - if the graph is undirected?
  - if the graph is very sparse?

Adjacency Lists and Maps

Adjacency Lists

- For each vertex, we maintain a separate (linked) list containing those edges that are incident to the vertex
- Complete set of edges can be determined by taking the union of the smaller sets
- Allows us to more efficiently find all edges incident to a given vertex

Adjacency Maps

- Secondary container of all edges incident on a vertex is organised as a *map* (`dict`), rather than a list, with adjacent vertex serving as key
- Allows for access to a specific edge (u, v) in $O(1)$ expected time

Adjacency Lists and Maps

Adjacency Lists

- For each vertex, we maintain a separate (linked) list containing those edges that are incident to the vertex
- Complete set of edges can be determined by taking the union of the smaller sets
- Allows us to more efficiently find all edges incident to a given vertex

Adjacency Maps

- Secondary container of all edges incident on a vertex is organised as a *map* (`dict`), rather than a list, with adjacent vertex serving as key
- Allows for access to a specific edge (u, v) in $O(1)$ expected time

Adjacency Lists and Maps

Adjacency Lists

- For each vertex, we maintain a separate (linked) list containing those edges that are incident to the vertex
- Complete set of edges can be determined by taking the union of the smaller sets
- Allows us to more efficiently find all edges incident to a given vertex

Adjacency Maps

- Secondary container of all edges incident on a vertex is organised as a *map* (`dict`), rather than a list, with adjacent vertex serving as key
- Allows for access to a specific edge (u, v) in $O(1)$ expected time

Adjacency Lists and Maps

Adjacency Lists

- For each vertex, we maintain a separate (linked) list containing those edges that are incident to the vertex
- Complete set of edges can be determined by taking the union of the smaller sets
- Allows us to more efficiently find all edges incident to a given vertex

Adjacency Maps

- Secondary container of all edges incident on a vertex is organised as a *map* (`dict`), rather than a list, with adjacent vertex serving as key
- Allows for access to a specific edge (u, v) in $O(1)$ expected time

Adjacency Lists and Maps

Adjacency Lists

- For each vertex, we maintain a separate (linked) list containing those edges that are incident to the vertex
- Complete set of edges can be determined by taking the union of the smaller sets
- Allows us to more efficiently find all edges incident to a given vertex

Adjacency Maps

- Secondary container of all edges incident on a vertex is organised as a *map* (`dict`), rather than a list, with adjacent vertex serving as key
- Allows for access to a specific edge (u, v) in $O(1)$ expected time

Comparison of Graph Data Structures

Operation	Edge List	Adj. List	Adj. Map	Adj. Mat.
`vertex_count()`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`edge_count()`	$O(1)$	$O(1)$	$O(1)$	$O(1)$
`vertices()`	$O(n)$	$O(n)$	$O(n)$	$O(n)$
`edges()`	$O(m)$	$O(m)$	$O(m)$	$O(m)$
`get_edge(u,v)`	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ exp.	$O(1)$
`degree(v)`	$O(m)$	$O(1)$	$O(1)$	$O(n)$
`incident_edges(v)`	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
`insert_vertex(x)`	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
`remove_vertex(v)`	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
`insert_edge(u,v,x)`	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$
`remove_edge(e)`	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$

Graph Algorithms: Overview

Graph Algorithms

Many many problems in science and engineering can be cast back on to a graph!

- Shortest path problem
- Travelling salesperson problem
- Finding [strongly] connected components
- Graph isomorphism
- Vertex cover problem
- Minimum spanning tree problem
- Hamiltonian path problem
- Eulerian path problem
- k-shortest path problem
- Centrality measures

Graph Algorithms

Many many problems in science and engineering can be cast back on to a graph!

- Shortest path problem
- Travelling salesperson problem
- Finding [strongly] connected components
- Graph isomorphism
- Vertex cover problem
- Minimum spanning tree problem
- Hamiltonian path problem
- Eulerian path problem
- k-shortest path problem
- Centrality measures

Graph Algorithms

Many many problems in science and engineering can be cast back on to a graph!

- Shortest path problem
- Travelling salesperson problem
- Finding [strongly] connected components
- Graph isomorphism
- Vertex cover problem
- Minimum spanning tree problem
- Hamiltonian path problem
- Eulerian path problem
- k-shortest path problem
- Centrality measures

Graph Algorithms

Many many problems in science and engineering can be cast back on to a graph!

- Shortest path problem
- Travelling salesperson problem
- Finding [strongly] connected components
- Graph isomorphism
- Vertex cover problem
- Minimum spanning tree problem
- Hamiltonian path problem
- Eulerian path problem
- k-shortest path problem
- Centrality measures

Graph Algorithms

Many many problems in science and engineering can be cast back on to a graph!

- Shortest path problem
- Travelling salesperson problem
- Finding [strongly] connected components
- Graph isomorphism
- Vertex cover problem
- Minimum spanning tree problem
- Hamiltonian path problem
- Eulerian path problem
- k-shortest path problem
- Centrality measures

Graph Algorithms

Many many problems in science and engineering can be cast back on to a graph!

- Shortest path problem
- Travelling salesperson problem
- Finding [strongly] connected components
- Graph isomorphism
- Vertex cover problem
- Minimum spanning tree problem
- Hamiltonian path problem
- Eulerian path problem
- *k*-shortest path problem
- Centrality measures

Graph Algorithms

Many many problems in science and engineering can be cast back on to a graph!

- Shortest path problem
- Travelling salesperson problem
- Finding [strongly] connected components
- Graph isomorphism
- Vertex cover problem
- Minimum spanning tree problem
- Hamiltonian path problem
- Eulerian path problem
- k-shortest path problem
- Centrality measures

Graph Algorithms

Many many problems in science and engineering can be cast back on to a graph!

- Shortest path problem
- Travelling salesperson problem
- Finding [strongly] connected components
- Graph isomorphism
- Vertex cover problem
- Minimum spanning tree problem
- Hamiltonian path problem
- Eulerian path problem
- k-shortest path problem
- Centrality measures

Graph Algorithms

Many many problems in science and engineering can be cast back on to a graph!

- Shortest path problem
- Travelling salesperson problem
- Finding [strongly] connected components
- Graph isomorphism
- Vertex cover problem
- Minimum spanning tree problem
- Hamiltonian path problem
- Eulerian path problem
- *k*-shortest path problem
- Centrality measures

Graph Algorithms

Many many problems in science and engineering can be cast back on to a graph!

- Shortest path problem
- Travelling salesperson problem
- Finding [strongly] connected components
- Graph isomorphism
- Vertex cover problem
- Minimum spanning tree problem
- Hamiltonian path problem
- Eulerian path problem
- k-shortest path problem
- Centrality measures