Lecture 24: Graph Algorithms: Traversals

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

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

Graph Algorithms in Biology

Many biological problems map back on to graph problems

- Path finding in metabolic networks
- Identifying important proteins in networks
- Clusters of proteins in interaction networks
- Assembling reads of a genome from a next-generation sequencer
- Chemoinformatics problems

Graph Algorithms in Biology

Many biological problems map back on to graph problems

- Path finding in metabolic networks
- Identifying important proteins in networks
- Clusters of proteins in interaction networks
- Assembling reads of a genome from a next-generation sequencer
- Chemoinformatics problems

Graph Algorithms in Biology

Many biological problems map back on to graph problems

- Path finding in metabolic networks
- Identifying important proteins in networks
- Clusters of proteins in interaction networks
- Assembling reads of a genome from a next-generation sequencer
- Chemoinformatics problems

Graph Algorithms in Biology

Many biological problems map back on to graph problems

- Path finding in metabolic networks
- Identifying important proteins in networks
- Clusters of proteins in interaction networks
- Assembling reads of a genome from a next-generation sequencer
- Chemoinformatics problems

Graph Algorithms in Biology

Many biological problems map back on to graph problems

- Path finding in metabolic networks
- Identifying important proteins in networks
- Clusters of proteins in interaction networks
- Assembling reads of a genome from a next-generation sequencer
- Chemoinformatics problems

Graph Algorithms
Most Common Algorithms

- Dijkstra's algorithm (single source shortest path)
- A* search algorithm (Best-first search)
- Tree traversal
- Breadth-first search
- Travelling salesman problem
- Depth-first search
- Kruskal's algorithm
- Prim's algorithm
- Topological sorting
- Bellman–Ford algorithm (-ve weights)
- Floyd–Warshall algorithm (all pairs)

Graph Traversals

Graph Traversals

~~History~~ Mythology

Greek mythology tells of an elaborate labyrinth that was built to house the monstrous Minotaur, which was part bull and part man. This labyrinth was so complex that neither beast nor human could escape it. No human, that is, until the Greek hero, Theseus, with the help of the king's daughter, Ariadne, decided to implement a graph traversal algorithm. Theseus fastened a ball of thread to the door of the labyrinth and unwound it as he traversed the twisting passages in search of the monster. Theseus obviously knew about good algorithm design, for, after finding and defeating the beast, Theseus easily followed the string back out of the labyrinth to the loving arms of Ariadne.

Graph Traversals
Overview

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

Graph Traversals
Overview

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

Graph Traversals
Overview

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

Graph Traversals

Interesting Problems (Undirected Graphs)

- Is there a path from vertex u to vertex v? Can we compute it?
- 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
- Is G connected?
- Computing a spanning tree of G, if G is connected
- What are the connected components of G?
- Are there cycles in G?

Graph Traversals

Interesting Problems (Undirected Graphs)

- Is there a path from vertex u to vertex v? Can we compute it?
- 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
- Is G connected?
- Computing a spanning tree of G, if G is connected
- What are the connected components of G?
- Are there cycles in G?

Graph Traversals

Interesting Problems (Undirected Graphs)

- Is there a path from vertex u to vertex v? Can we compute it?
- 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
- Is G connected?
- Computing a spanning tree of G, if G is connected
- What are the connected components of G?
- Are there cycles in G?

Graph Traversals

Interesting Problems (Undirected Graphs)

- Is there a path from vertex u to vertex v? Can we compute it?
- 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
- Is G connected?
- Computing a spanning tree of G, if G is connected
- What are the connected components of G?
- Are there cycles in G?

Graph Traversals

Interesting Problems (Undirected Graphs)

- Is there a path from vertex u to vertex v? Can we compute it?
- 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
- Is G connected?
- Computing a spanning tree of G, if G is connected
- What are the connected components of G?
- Are there cycles in G?

Graph Traversals

Interesting Problems (Undirected Graphs)

- Is there a path from vertex u to vertex v? Can we compute it?
- 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
- Is G connected?
- Computing a spanning tree of G, if G is connected
- What are the connected components of G?
- Are there cycles in G?

Graph Traversals

Interesting Problems (Directed Graphs)

- Is there a directed path from vertex u to vertex v? Can we compute it?
- What are all the vertices of G that are reachable from a given vertex s?
- Is G acyclic?
- Is G strongly connected?

Graph Traversals

Interesting Problems (Directed Graphs)

- Is there a directed path from vertex u to vertex v? Can we compute it?
- What are all the vertices of G that are reachable from a given vertex s?
- Is G acyclic?
- Is G strongly connected?

Graph Traversals

Interesting Problems (Directed Graphs)

- Is there a directed path from vertex u to vertex v? Can we compute it?
- What are all the vertices of G that are reachable from a given vertex s?
- Is G acyclic?
- Is G strongly connected?

Graph Traversals

Interesting Problems (Directed Graphs)

- Is there a directed path from vertex u to vertex v? Can we compute it?
- What are all the vertices of G that are reachable from a given vertex s?
- Is G acyclic?
- Is G strongly connected?

Traversal Algorithms

Basic Ideas

- Mazes are naturally represented by graphs
  - each vertex denotes a junction
  - each edge denotes a hallway
- Any graph traversal algorithm must be powerful enough to get us out of an arbitrary maze!
- For *efficiency*, we must make sure we don't get trapped in the maze and visit the same place repeatedly
- For *correctness*, we must do the traversal in a systematic way to guarantee that we get out of the maze
- Our search must take us through every edge and vertex in the graph

Traversal Algorithms

Basic Ideas

- Mazes are naturally represented by graphs
  - each vertex denotes a junction
  - each edge denotes a hallway
- Any graph traversal algorithm must be powerful enough to get us out of an arbitrary maze!
- For *efficiency*, we must make sure we don't get trapped in the maze and visit the same place repeatedly
- For *correctness*, we must do the traversal in a systematic way to guarantee that we get out of the maze
- Our search must take us through every edge and vertex in the graph

Traversal Algorithms

Basic Ideas

- Mazes are naturally represented by graphs
  - each vertex denotes a junction
  - each edge denotes a hallway
- Any graph traversal algorithm must be powerful enough to get us out of an arbitrary maze!
- For *efficiency*, we must make sure we don't get trapped in the maze and visit the same place repeatedly
- For *correctness*, we must do the traversal in a systematic way to guarantee that we get out of the maze
- Our search must take us through every edge and vertex in the graph

Traversal Algorithms

Basic Ideas

- Mazes are naturally represented by graphs
  - each vertex denotes a junction
  - each edge denotes a hallway
- Any graph traversal algorithm must be powerful enough to get us out of an arbitrary maze!
- For *efficiency*, we must make sure we don't get trapped in the maze and visit the same place repeatedly
- For *correctness*, we must do the traversal in a systematic way to guarantee that we get out of the maze
- Our search must take us through every edge and vertex in the graph

Traversal Algorithms

Basic Ideas

- Mazes are naturally represented by graphs
  - each vertex denotes a junction
  - each edge denotes a hallway
- Any graph traversal algorithm must be powerful enough to get us out of an arbitrary maze!
- For *efficiency*, we must make sure we don't get trapped in the maze and visit the same place repeatedly
- For *correctness*, we must do the traversal in a systematic way to guarantee that we get out of the maze
- Our search must take us through every edge and vertex in the graph

Traversal Algorithms

Basic Ideas

- Mazes are naturally represented by graphs
  - each vertex denotes a junction
  - each edge denotes a hallway
- Any graph traversal algorithm must be powerful enough to get us out of an arbitrary maze!
- For *efficiency*, we must make sure we don't get trapped in the maze and visit the same place repeatedly
- For *correctness*, we must do the traversal in a systematic way to guarantee that we get out of the maze
- Our search must take us through every edge and vertex in the graph

Traversal Algorithms

Basic Ideas

- Mazes are naturally represented by graphs
  - each vertex denotes a junction
  - each edge denotes a hallway
- Any graph traversal algorithm must be powerful enough to get us out of an arbitrary maze!
- For *efficiency*, we must make sure we don't get trapped in the maze and visit the same place repeatedly
- For *correctness*, we must do the traversal in a systematic way to guarantee that we get out of the maze
- Our search must take us through every edge and vertex in the graph

Traversal Algorithms

Basic Ideas

- Key idea behind graph traversal is to *mark* each vertex when we first *visit* it and keep track of what we have not yet completely *explored*
- Although bread-crumbs or unraveled threads have been used to mark visited places in fairy-tale mazes, we will rely on *Boolean flags*

Traversal Algorithms

Basic Ideas

- Key idea behind graph traversal is to *mark* each vertex when we first *visit* it and keep track of what we have not yet completely *explored*
- Although bread-crumbs or unraveled threads have been used to mark visited places in fairy-tale mazes, we will rely on *Boolean flags*

Traversal Algorithms

Basic Ideas

- Each vertex will exist in one of three states:
  - *undiscovered* — the vertex is in its initial state
  - *discovered* — the vertex has been found, but we have not yet checked out all its incident edges
  - *processed* — the vertex after we have visited all its incident edges
- Obviously, a vertex cannot be processed until after we discover it, so the state of each vertex progresses over the course of the traversal from undiscovered to discovered to processed
- We must also maintain a structure containing the vertices that we have discovered but not yet completely processed

Traversal Algorithms

Basic Ideas

- Each vertex will exist in one of three states:
  - *undiscovered* — the vertex is in its initial state
  - *discovered* — the vertex has been found, but we have not yet checked out all its incident edges
  - *processed* — the vertex after we have visited all its incident edges
- Obviously, a vertex cannot be processed until after we discover it, so the state of each vertex progresses over the course of the traversal from undiscovered to discovered to processed
- We must also maintain a structure containing the vertices that we have discovered but not yet completely processed

Traversal Algorithms

Basic Ideas

- Each vertex will exist in one of three states:
  - *undiscovered* — the vertex is in its initial state
  - *discovered* — the vertex has been found, but we have not yet checked out all its incident edges
  - *processed* — the vertex after we have visited all its incident edges
- Obviously, a vertex cannot be processed until after we discover it, so the state of each vertex progresses over the course of the traversal from undiscovered to discovered to processed
- We must also maintain a structure containing the vertices that we have discovered but not yet completely processed

Traversal Algorithms

Basic Ideas

- Each vertex will exist in one of three states:
  - *undiscovered* — the vertex is in its initial state
  - *discovered* — the vertex has been found, but we have not yet checked out all its incident edges
  - *processed* — the vertex after we have visited all its incident edges
- Obviously, a vertex cannot be processed until after we discover it, so the state of each vertex progresses over the course of the traversal from undiscovered to discovered to processed
- We must also maintain a structure containing the vertices that we have discovered but not yet completely processed

Traversal Algorithms

Basic Ideas

- Each vertex will exist in one of three states:
  - *undiscovered* — the vertex is in its initial state
  - *discovered* — the vertex has been found, but we have not yet checked out all its incident edges
  - *processed* — the vertex after we have visited all its incident edges
- Obviously, a vertex cannot be processed until after we discover it, so the state of each vertex progresses over the course of the traversal from undiscovered to discovered to processed
- We must also maintain a structure containing the vertices that we have discovered but not yet completely processed

Traversal Algorithms

Basic Ideas

- Each vertex will exist in one of three states:
  - *undiscovered* — the vertex is in its initial state
  - *discovered* — the vertex has been found, but we have not yet checked out all its incident edges
  - *processed* — the vertex after we have visited all its incident edges
- Obviously, a vertex cannot be processed until after we discover it, so the state of each vertex progresses over the course of the traversal from undiscovered to discovered to processed
- We must also maintain a structure containing the vertices that we have discovered but not yet completely processed

Traversal Algorithms

Basic Ideas

- Initially, only the single start vertex is considered to be discovered
- To completely explore a vertex v, we must evaluate each edge leaving v
- If an edge goes to an undiscovered vertex x, we mark x discovered and add it to the *list of work to do*
- Ignore an edge that goes to a *processed* vertex — it will tell us nothing new about the graph
- Also ignore any edge going to a discovered but not processed vertex, because the destination already resides on the list of vertices to process
- Each undirected edge will be considered exactly twice, once when each of its endpoints is explored
- Directed edges will be considered only once, when exploring the source vertex

Traversal Algorithms

Basic Ideas

- Initially, only the single start vertex is considered to be discovered
- To completely explore a vertex v, we must evaluate each edge leaving v
- If an edge goes to an undiscovered vertex x, we mark x discovered and add it to the *list of work to do*
- Ignore an edge that goes to a *processed* vertex — it will tell us nothing new about the graph
- Also ignore any edge going to a discovered but not processed vertex, because the destination already resides on the list of vertices to process
- Each undirected edge will be considered exactly twice, once when each of its endpoints is explored
- Directed edges will be considered only once, when exploring the source vertex

Traversal Algorithms

Basic Ideas

- Initially, only the single start vertex is considered to be discovered
- To completely explore a vertex v, we must evaluate each edge leaving v
- If an edge goes to an undiscovered vertex x, we mark x discovered and add it to the *list of work to do*
- Ignore an edge that goes to a *processed* vertex — it will tell us nothing new about the graph
- Also ignore any edge going to a discovered but not processed vertex, because the destination already resides on the list of vertices to process
- Each undirected edge will be considered exactly twice, once when each of its endpoints is explored
- Directed edges will be considered only once, when exploring the source vertex

Traversal Algorithms

Basic Ideas

- Initially, only the single start vertex is considered to be discovered
- To completely explore a vertex v, we must evaluate each edge leaving v
- If an edge goes to an undiscovered vertex x, we mark x discovered and add it to the *list of work to do*
- Ignore an edge that goes to a *processed* vertex — it will tell us nothing new about the graph
- Also ignore any edge going to a discovered but not processed vertex, because the destination already resides on the list of vertices to process
- Each undirected edge will be considered exactly twice, once when each of its endpoints is explored
- Directed edges will be considered only once, when exploring the source vertex

Traversal Algorithms

Basic Ideas

- Initially, only the single start vertex is considered to be discovered
- To completely explore a vertex v, we must evaluate each edge leaving v
- If an edge goes to an undiscovered vertex x, we mark x discovered and add it to the *list of work to do*
- Ignore an edge that goes to a *processed* vertex — it will tell us nothing new about the graph
- Also ignore any edge going to a discovered but not processed vertex, because the destination already resides on the list of vertices to process
- Each undirected edge will be considered exactly twice, once when each of its endpoints is explored
- Directed edges will be considered only once, when exploring the source vertex

Traversal Algorithms

Basic Ideas

- Initially, only the single start vertex is considered to be discovered
- To completely explore a vertex v, we must evaluate each edge leaving v
- If an edge goes to an undiscovered vertex x, we mark x discovered and add it to the *list of work to do*
- Ignore an edge that goes to a *processed* vertex — it will tell us nothing new about the graph
- Also ignore any edge going to a discovered but not processed vertex, because the destination already resides on the list of vertices to process
- Each undirected edge will be considered exactly twice, once when each of its endpoints is explored
- Directed edges will be considered only once, when exploring the source vertex

Traversal Algorithms

Basic Ideas

- Initially, only the single start vertex is considered to be discovered
- To completely explore a vertex v, we must evaluate each edge leaving v
- If an edge goes to an undiscovered vertex x, we mark x discovered and add it to the *list of work to do*
- Ignore an edge that goes to a *processed* vertex — it will tell us nothing new about the graph
- Also ignore any edge going to a discovered but not processed vertex, because the destination already resides on the list of vertices to process
- Each undirected edge will be considered exactly twice, once when each of its endpoints is explored
- Directed edges will be considered only once, when exploring the source vertex

Traversal Algorithms

- Every edge and vertex in the connected component must eventually be visited
- Why?
- Suppose that there exists a vertex u that remains unvisited, whose neighbour v was visited
- This neighbour v will eventually be explored, after which we will certainly visit u
- Thus, we must find everything that is there to be found

Traversal Algorithms

- Every edge and vertex in the connected component must eventually be visited
- Why?
- Suppose that there exists a vertex u that remains unvisited, whose neighbour v was visited
- This neighbour v will eventually be explored, after which we will certainly visit u
- Thus, we must find everything that is there to be found

Traversal Algorithms

- Every edge and vertex in the connected component must eventually be visited
- Why?
- Suppose that there exists a vertex u that remains unvisited, whose neighbour v was visited
- This neighbour v will eventually be explored, after which we will certainly visit u
- Thus, we must find everything that is there to be found

Traversal Algorithms

- Every edge and vertex in the connected component must eventually be visited
- Why?
- Suppose that there exists a vertex u that remains unvisited, whose neighbour v was visited
- This neighbour v will eventually be explored, after which we will certainly visit u
- Thus, we must find everything that is there to be found

Traversal Algorithms

- Every edge and vertex in the connected component must eventually be visited
- Why?
- Suppose that there exists a vertex u that remains unvisited, whose neighbour v was visited
- This neighbour v will eventually be explored, after which we will certainly visit u
- Thus, we must find everything that is there to be found

Graph Traversals: BFS

Breadth-First Search

- At some point during the course of a traversal, every node in the graph changes state from *undiscovered* to *discovered*
- In a breadth-first search of an undirected graph, we assign a direction to each edge, from the discoverer u to the discovered v
- We thus denote u to be the parent of v
- Since each node has exactly one parent, except for the root, this defines a tree on the vertices of the graph

Breadth-First Search

- At some point during the course of a traversal, every node in the graph changes state from *undiscovered* to *discovered*
- In a breadth-first search of an undirected graph, we assign a direction to each edge, from the discoverer u to the discovered v
- We thus denote u to be the parent of v
- Since each node has exactly one parent, except for the root, this defines a tree on the vertices of the graph

Breadth-First Search

- At some point during the course of a traversal, every node in the graph changes state from *undiscovered* to *discovered*
- In a breadth-first search of an undirected graph, we assign a direction to each edge, from the discoverer u to the discovered v
- We thus denote u to be the parent of v
- Since each node has exactly one parent, except for the root, this defines a tree on the vertices of the graph

Breadth-First Search

- At some point during the course of a traversal, every node in the graph changes state from *undiscovered* to *discovered*
- In a breadth-first search of an undirected graph, we assign a direction to each edge, from the discoverer u to the discovered v
- We thus denote u to be the parent of v
- Since each node has exactly one parent, except for the root, this defines a tree on the vertices of the graph

Breadth-First Search

```
BFS(G, s)
    for each vertex u ∈ V[G] − {s} do
        state[u] = "undiscovered"
        p[u] = nil, i.e. no parent is in the BFS tree
    state[s] = "discovered"
    p[s] = nil
    Q = {s}
    while Q ≠ ∅ do
        u = dequeue[Q]
        process vertex u as desired
        for each v ∈ Adj[u] do
            process edge (u, v) as desired
            if state[v] = "undiscovered" then
                state[v] = "discovered"
                p[v] = u
                enqueue[Q, v]
        state[u] = "processed"
```

Graph Traversals: DFS

Depth-First Search

- Can be done in recursive fashion
- How?

Depth-First Search

- Can be done in recursive fashion
- How?