Introduction to Graphs

http://people.cs.clemson.edu/~pargas/courses/cs212/common/notes/ppt/

Introduction

• Graphs are a generalization of trees
  – Nodes or vertices
  – Edges or arcs
• Two kinds of graphs
  – Directed
  – Undirected

Introduction: Formal Definition

• A directed graph, or digraph, is a graph in which the edges are ordered pairs
  – (v, w) \neq (w, v)
• An undirected graph is a graph in which the edges are unordered pairs
  – (v, w) \equiv (w, v)

Introduction: Directed Graphs

• In a directed graph, the edges are arrows.
• Directed graphs show the flow from one node to another and not vice versa.

Introduction: Undirected Graphs

• In a directed graph, the edges are lines.
• Directed graphs show a relationship between two nodes.
Terminology

- In the directed graph above, b is adjacent to a because \((a, b) \in E\). Note that a is not adjacent to b.
- A is a predecessor of node B
- B is a successor of node A
- The source of the edge is node A, the target is node B

Terminology

- An acyclic path is a path where each vertex is unique
- A cyclic path is a path such that
  - There are at least two vertices on the path
  - \(w_1 = w_n\) (path starts and ends at same vertex)

Terminology

- In the undirected graph above, a and b are adjacent because \((a,b) \in E\). a and b are called neighbors.

Test Your Knowledge

Cyclic or Acyclic?

- A directed graph that has no cyclic paths is called a DAG (a Directed Acyclic Graph).
- An undirected graph that has an edge between every pair of vertices is called a complete graph.

Note: A directed graph can also be a complete graph; in that case, there must be an edge from every vertex to every other vertex.
**Terminology**

- An undirected graph is **connected** if a path exists from every vertex to every other vertex.
- A directed graph is **strongly connected** if a path exists from every vertex to every other vertex.
- A directed graph is **weakly connected** if a path exists from every vertex to every other vertex, disregarding the direction of the edge.

**Various types of graphs**

- Connected/disconnected graphs
- The circled subgraphs are also known as connected components.
Various types of graphs

• Directed/undirected graphs

• You may treat each undirected edge as two directed edges in opposite directions.

Weighted/unweighted graphs

• You may treat unweighted edges to be weighted edges of equal weights.

Special graphs

• Tree: either one of the followings is the definition
  – A connected graph with $|V|-1$ edges
  – A connected graph without cycles
  – A graph with exactly one path between every pair of vertices

• Tree edges could be directed or undirected
• For trees with directed edges, a root usually exists

Special graphs

• Planar graphs
  – A graph that can be drawn on a plane without edge intersections
  – The following two graphs are equivalent and planar:

• To be discussed in details in Graph (III)

Special graphs

• Forest
  – All connected component(s) is/are tree(s)
• How many trees are there in the following forest?
How to store graphs in the program?

• Usually, the vertices are labeled beforehand
• 3 types of graph representations:
  – Adjacency matrix
  – Adjacency list
  – Edge list

Adjacency matrix

• Use a 2D array

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adjacency list

• N vertices, N linked lists
• Each list stores its adjacent vertices

Edge list

• A list of edges

<table>
<thead>
<tr>
<th>id</th>
<th>x</th>
<th>y</th>
<th>w</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>-2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
**Edge list**

- Memory complexity?
- Time complexity for:
  - Checking the weight of an edge between 2 given nodes?
  - Querying all adjacent nodes of a given node?

**Uses for Graphs**

- **Two-Player Game Tree:** All of the possibilities in a board game like chess can be represented in a graph. Each vertex stands for one possible board position. (For chess, this is a very big graph!)

**Which one should be used?**

- It depends on:
  - Constraints
  - Time Limit
  - Memory Limit
  - What algorithm is used

**Uses for Graphs**

- **Computer network:** The set of vertices V represents the set of computers in the network. There is an edge (u, v) if and only if there is a direct communication link between the computers corresponding to u and v.

**Uses for Graphs**

- **Precedence Constraints:** Suppose you have a set of jobs to complete, but some must be completed before others are begun. (For example, Atilla advises you always pillage before you burn.) Here the vertices are jobs to be done. Directed edges indicate constraints; there is a directed edge from job u to job v if job u must be done before job v is begun.

**Topological Sort**

Don't burn before you pillage!
**Topological Sort**

- Informally, a topological sort is a linear ordering of the vertices of a DAG in which all successors of any given vertex appear in the sequence after that vertex.

**Method to the Madness**

- One way to find a topological sort is to consider the in-degrees of the vertices. (The number of incoming edges is the in-degree). Clearly the first vertex in a topological sort must have in-degree zero and every DAG must contain at least one vertex with in-degree zero.

**Simple Topological Sort Algorithm**

- Repeat the following steps until the graph is empty:
  - Select a vertex that has in-degree zero.
  - Add the vertex to the sort.
  - Delete the vertex and all the edges emanating from it from the graph.

**Test Your Knowledge**

- Give a topological sort for this graph, it should be evident that more than one solution exists for this problem.

**Backtracking Algorithm**

**Depth-First Search**

Text

Read Weiss, § 9.6 Depth-First Search and § 10.5 Backtracking Algorithms

**Requirements**

- Also called Depth-First Search
- Can be used to attempt to visit all nodes of a graph in a systematic manner
- Works with directed and undirected graphs
- Works with weighted and unweighted graphs
Slide 43

**Walk-Through**

- **Task:** Conduct a depth-first search of the graph starting with node D.

Slide 44

- Visited array:
  - A
  - B
  - C
  - D
  - E
  - F
  - G
  - H

The order nodes are visited:
- D
- C

Slide 46

- The order nodes are visited:
- D
- C

Slide 47

- The order nodes are visited:
- D
- C

Slide 48

- The order nodes are visited:
- D
- C

Consider nodes adjacent to D, decide to visit C first (Rule: visit adjacent nodes in alphabetical order).

**Walk-Through**

- No nodes adjacent to C; cannot continue backtrack, i.e., pop stack and restore previous state.

- Back to D – C has been visited, decide to visit E next.
The order nodes are visited: D, C, E
Back to D – C has been visited,
decide to visit E next.

The order nodes are visited: D, C, E

Only G is adjacent to E

The order nodes are visited: D, C, E

Nodes D and H are adjacent to G. D has already been visited. Decide to visit H.

Visit G

The order nodes are visited: D, C, E, G

Visit H

The order nodes are visited: D, C, E, G, H

Nodes A and B are adjacent to F. Decide to visit A next.
The order nodes are visited: D, C, E, G, H, A

No unvisited nodes adjacent to B. Backtrack (pop the stack).

The order nodes are visited: D, C, E, G, H, A, B

No unvisited nodes adjacent to A. Backtrack (pop the stack).

The order nodes are visited: D, C, E, G, H, A, B

No unvisited nodes adjacent to H. Backtrack (pop the stack).
No unvisited nodes adjacent to G. Backtrack (pop the stack).

The order nodes are visited: D, C, E, G, H, A, B

No unvisited nodes adjacent to F. Backtrack (pop the stack).

The order nodes are visited: D, C, E, G, H, A, B, F

F is unvisited and is adjacent to D. Decide to visit F next.

The order nodes are visited: D, C, E, G, H, A, B, F

No unvisited nodes adjacent to F. Backtrack.
Walk-Through

Visited

\[ \begin{array}{c}
A \checkmark \\
B \checkmark \\
C \checkmark \\
D \checkmark \\
E \checkmark \\
F \checkmark \\
G \checkmark \\
H \checkmark \\
\end{array} \]

The order nodes are visited: Stack is empty. Depth-first traversal is done.

D, C, E, G, H, A, B, F

Requirements

- Can be used to attempt to visit all nodes of a graph in a systematic manner
- Works with directed and undirected graphs
- Works with weighted and unweighted graphs

Consider Trees

1. What depth-first traversals do you know?
2. How do the traversals differ?
3. In the walk-through, we visited a node just as we pushed the node onto the stack. Is there another time at which you can visit the node?
4. Conduct a depth-first search of the same graph using the strategy you came up with in #3.

Overview

Breadth-first search starts with given node

Task: Conduct a breadth-first search of the graph starting with node D

Breadth-First Search

Text
Read Weiss, § 9.3 (pp. 299-304) Breadth-First Search Algorithms

Overview

Breadth-first search starts with given node

Then visits nodes adjacent in some specified order (e.g., alphabetical)

Like ripples in a pond

Nodes visited: D
Overview

Breadth-first search starts with given node
Then visits nodes adjacent in some specified order (e.g., alphabetical)
Like ripples in a pond

Nodes visited: D, C

Overview

When all nodes in ripple are visited, visit nodes in next ripples

Nodes visited: D, C, E, F, G

Overview

Breadth-first search starts with given node
Then visits nodes adjacent in some specified order (e.g., alphabetical)
Like ripples in a pond

Nodes visited: D, C, E

Overview

When all nodes in ripple are visited, visit nodes in next ripples

Nodes visited: D, C, E, F, G, H

Overview

Breadth-first search starts with given node
Then visits nodes adjacent in some specified order (e.g., alphabetical)
Like ripples in a pond

Nodes visited: D, C, E, F

Overview

When all nodes in ripple are visited, visit nodes in next ripples

Nodes visited: D, C, E, F, G, H

Overview

Breadth-first search starts with given node
Then visits nodes adjacent in some specified order (e.g., alphabetical)
Like ripples in a pond

Nodes visited: D, C, E, F, H

Overview

When all nodes in ripple are visited, visit nodes in next ripples

Nodes visited: D, C, E, F, G, H, A
**Overview**

When all nodes in ripple are visited, visit nodes in next ripples.

Nodes visited: D, C, E, F, G, H, A, B

---

**Walk-Through**

Enqueued Array

How is this accomplished? Simply replace the stack with a queue. Rules: (1) Maintain an enqueued array. (2) Visit node when dequeued.

Nodes visited: D, C

Enqueue D. Notice, D not yet visited.

Nodes visited: D, C, E

Dequeue E. Visit E. Enqueue unenqueued nodes adjacent to E.
**Walk-Through**

Nodes visited: D, C, E, F

Dequeue F. Visit F. Enqueue unenqueued nodes adjacent to F.

Nodes visited: D, C, E, F

Dequeue G. Visit G. Enqueue unenqueued nodes adjacent to G.

Nodes visited: D, C, E, F, G

Dequeue H. Visit H. Enqueue unenqueued nodes adjacent to H.

Nodes visited: D, C, E, F, G, H

Dequeue A. Visit A. Enqueue unenqueued nodes adjacent to A.

Q empty. Algorithm done.

Nodes visited: D, C, E, F, G, H, A, B

Q empty.
Consider Trees

1. What do we call a breadth-first traversal on trees?

Requirements

- Works with directed and undirected graphs
- Works with weighted and unweighted graphs
- Rare type of algorithm
  A greedy algorithm that produces an optimal solution

Dijkstra’s Algorithm

Text
Read Weiss, § 9.3
Dijkstra’s Algorithm
Single Source Multiple Destination
Shortest Path Algorithm

Walk-Through

Initialize array

Start with G

Slide 91
Slide 94
Slide 92
Slide 95
Slide 93
Slide 96
Update unselected nodes

Select minimum distance

Update unselected nodes

Select minimum distance

Update unselected nodes

Select minimum distance

Update unselected nodes

Select minimum distance

Update unselected nodes
Update unselected nodes

Select minimum distance

Update unselected nodes

Select minimum distance

Update unselected nodes

Select minimum distance

Update unselected nodes

Select minimum distance
Update unselected nodes

Select minimum distance

Done

Order of Complexity

- Analysis
  - findMin() takes $O(V)$ time
  - outer loop iterates $(V-1)$ times
  $\Rightarrow O(V^2)$ time
- Optimal for dense graphs, i.e., $|E| = O(V^2)$
- Suboptimal for sparse graphs, i.e., $|E| = O(V)$

Order of Complexity

If the graph is sparse, i.e., $|E| = O(V)$
- maintain distances in a priority queue
- insert new (shorter) distance produced by line 10 of Figure 9.32
  $\Rightarrow O(|E| \log |V|)$ complexity

Dijkstra’s Algorithm

Read § 9.3.3
Dijkstra’s algorithm as shown in Figure 9.32 does not work!
Why?
Acyclic Graphs

• Read § 9.3.4
• Combine topological sort with Dijkstra’s algorithm

All-Pairs Shortest Paths

• One option: run Dijkstra’s algorithm |V| times → O(V^3) time
• A more efficient O(V^3) time algorithm is discussed in Chapter 10