CSE 326: Data Structures
Lecture #16
Graphs I: DFS & BFS

Henry Kautz
Winter Quarter 2002

Outline

• Graphs (TO DO: READ WEISS CH 9)
• Graph Data Structures
• Graph Properties
• Topological Sort
• Graph Traversals
  – Depth First Search
  – Breadth First Search
  – Iterative Deepening Depth First
• Shortest Path Problem
  – Dijkstra’s Algorithm

Graph ADT

Graphs are a formalism for representing relationships between objects

- a graph G is represented as
  \[ G = (V, E) \]
  - V is a set of vertices
  - E is a set of edges

- operations include:
  - iterating over vertices
  - iterating over edges
  - iterating over vertices adjacent to a specific vertex
  - asking whether an edge exists connected two vertices

What Graph is THIS?

ReferralWeb
(co-authorship in scientific papers)
Graph Representation 1:
Adjacency Matrix
A $|V| \times |V|$ array in which an element $(u, v)$ is true if and only if there is an edge from $u$ to $v$

Runtime:
iterate over vertices
iterate over edges
iterate edges adj. to vertex
edge exists?

Space requirements:

Graph Representation 2:
Adjacency List
A $|V|$-ary list (array) in which each entry stores a list (linked list) of all adjacent vertices

Runtime:
iterate over vertices
iterate over edges
iterate edges adj. to vertex
edge exists?

Space requirements:

Directed vs. Undirected Graphs
• In directed graphs, edges have a specific direction:

• In undirected graphs, they don’t (edges are two-way):

• Vertices $u$ and $v$ are adjacent if $(u, v) \in E$

Graph Density
A sparse graph has $O(|V|)$ edges

A dense graph has $\Theta(|V|^2)$ edges

Anything in between is either sparse or dense depending on the context.

Weighted Graphs
Each edge has an associated weight or cost.

There may be more information in the graph as well.
Paths and Cycles

A path is a list of vertices \( \{v_1, v_2, ..., v_n\} \) such that \((v_i, v_{i+1}) \in E\) for all \(0 \leq i < n\).

A cycle is a path that begins and ends at the same node.

\[
p = \{\text{Seattle, Salt Lake City, Chicago, Dallas, San Francisco, Seattle}\}
\]

Path Length and Cost

Path length: the number of edges in the path
Path cost: the sum of the costs of each edge

\[
\text{length}(p) = 5 \quad \text{cost}(p) = 11.5
\]

Connectivity

Undirected graphs are connected if there is a path between any two vertices.

Directed graphs are strongly connected if there is a path from any one vertex to any other.

Directed graphs are weakly connected if there is a path between any two vertices, ignoring direction.

A complete graph has an edge between every pair of vertices.

Trees as Graphs

- Every tree is a graph with some restrictions:
  - the tree is directed
  - there are no cycles (directed or undirected)
  - there is a directed path from the root to every node

Directed Acyclic Graphs (DAGs)

DAGs are directed graphs with no cycles.

if program call graph is a DAG, then all procedure calls can be in-lined

Trees \(\subset\) DAGs \(\subset\) Graphs

Application of DAGs: Representing Partial Orders

- check in airport
- reserve flight
- call taxi
- take flight
- taxi to airport
- locate gate
- pack bag
- load bags
Topological Sort

Given a graph, $G = (V, E)$, output all the vertices in $V$ such that no vertex is output before any other vertex with an edge to it.

Topo-Sort Take One

Label each vertex’s in-degree (# of inbound edges)

While there are vertices remaining

Pick a vertex with in-degree of zero and output it

Reduce the in-degree of all vertices adjacent to it

Remove it from the list of vertices

runtime:

Topo-Sort Take Two

Label each vertex’s in-degree

Initialize a queue (or stack) to contain all in-degree zero vertices

While there are vertices remaining in the queue

Remove a vertex $v$ with in-degree of zero and output it

Reduce the in-degree of all vertices adjacent to $v$

Put any of these with new in-degree zero on the queue

runtime:

Depth-First Search

- Both Pre-Order and Post-Order traversals are examples of depth-first search
  - nodes are visited deeply on the left-most branches before any nodes are visited on the right-most branches
  - visiting the right branches deeply before the left would still be depth-first! Crucial idea is “go deep first!”
- In DFS the nodes “being worked on” are kept on a stack (where?)
- Recursion is a clue that DFS may be lurking…

Level-Order Tree Traversal

- Consider task of traversing tree level by level from top to bottom (alphabetic order)
- Is this also DFS?
Breadth-First Search

- No! Level-order traversal is an example of Breadth-First Search
- BFS characteristics
  - Nodes being worked on maintained in a FFO Queue, not a stack
  - Iterative-style procedures often easier to design than recursive procedures
  - Put root in a Queue
  - Repeat until Queue is empty:
    - Dequeue a node
    - Process it
    - Add it’s children to queue

Graph Traversals

- Depth first search and breadth first search also work for arbitrary (directed or undirected) graphs
  - Must mark visited vertices so you do not go into an infinite loop!
- Either can be used to determine connectivity:
  - Is there a path between two given vertices?
  - Is the graph (weakly) connected?
- Important difference: Breadth-first search always finds a shortest path from the start vertex to any other (for unweighted graphs)
  - Depth first search may not!

Single Source, Shortest Path for Weighted Graphs

Given a graph $G = (V, E)$ with edge costs $c(e)$, and a vertex $s \in V$, find the shortest (lowest cost) path from $s$ to every vertex in $V$

- Graph may be directed or undirected
- Graph may or may not contain cycles
- Weights may be all positive or not
- What is the problem if graph contains cycles whose total cost is negative?

The Trouble with Negative Weighted Cycles
Edsger Wybe Dijkstra

Legendary figure in computer science; now a professor at University of Texas.

Supports teaching introductory computer courses without computers (pencil and paper programming)

Also famous for refusing to read e-mail; his staff has to print out messages and put them in his mailbox.

Dijkstra’s Algorithm for Single Source Shortest Path

- Classic algorithm for solving shortest path in weighted graphs (with only positive edge weights)
- Similar to breadth-first search, but uses a priority queue instead of a FIFO queue:
  - Always select (expand) the vertex that has a lowest-cost path to the start vertex
  - A kind of “greedy” algorithm
- Correctly handles the case where the lowest-cost (shortest) path to a vertex is not the one with fewest edges

Pseudocode for Dijkstra

Initialize the cost of each vertex to \(\infty\)
\[
cost[s] = 0;
heap.insert(s);
\]
While (!heap.empty())
\[
n = heap.deleteMin();
\]
For (each vertex a which is adjacent to n along edge e)
\[
if (\text{cost}[n] + \text{edge}_\text{cost}[e] < \text{cost}[a]) \text{ then}
\]
\[
cost[a] = \text{cost}[n] + \text{edge}_\text{cost}[e]
previous_on_path_to[a] = n;
if (a is in the heap) then heap.decreaseKey(a)
else heap.insert(a)

Important Features

- Once a vertex is removed from the head, the cost of the shortest path to that node is known
- While a vertex is still in the heap, another shorter path to it might still be found
- The shortest path itself from s to any node a can be found by following the pointers stored in previous_on_path_to[a]

Dijkstra’s Algorithm in Action

<table>
<thead>
<tr>
<th>vertex</th>
<th>previous</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Demo

Dijkstra’s
Data Structures for Dijkstra’s Algorithm

\[ |V| \text{ times:} \]
Select the unknown node with the lowest cost

\[ |E| \text{ times:} \]
\[ a’s \text{ cost} = \min(a’s \text{ old cost}, \ldots) \]
\[ \text{findMin/deleteMin} \quad O(\log |V|) \]
\[ \text{decreaseKey} \quad O(\log |V|) \]

runtime: \( O(|E| \log |V|) \)

Fibonacci Heaps

- A complex version of heaps - Weiss 11.4
- Used more in theory than in practice
- Amortized \( O(1) \) time bound for decreaseKey
- \( O(\log n) \) time for deleteMin

Dijkstra’s uses \(|V|\) deleteMins and \(|E|\) decreaseKeys

runtime with Fibonacci heaps: \( O(|E| + |V| \log |V|) \)

for dense graphs, asymptotically better than \( O(|E| \log |V|) \)