Lecture: Analysis of Algorithms (CS583 - 002)

Amarda Shehu

Fall 2017
1. **Graphs**
   - Definition of a Graph
   - Omnipresence of Graphs

2. **Graph Representations**
   - Adjacency List Representation
   - Adjacency Matrix Representation
   - Alternative Graph Representations

3. **Solving Problems with Graph Algorithms**

4. **Elementary Graph Algorithms for Searching**
   - Uninformed Graph Search
     - Depth-first Search
     - Breadth-first Search (BFS)
   - Graph Algorithms for Graph Problems
   - Topological Sort
What is a Graph?

Graph \( G = (V, E) \)

- \( V \) : set of vertices
- \( E \) : set of edges consisting of pairs of vertices from \( V \)

\[
V = \{v_0, v_1, v_2, v_3, v_4\} \\
E = \{(v_0, v_1), (v_0, v_3), (v_1, v_2), (v_1, v_4)\}
\]
First Graph Problem

Seven Bridges of Koenigsberg [1736]:
Find a route that crosses each bridge exactly once.
Posed by Leonard Euler [1707 - 1783].

modified from wikipedia

What is the minimum number of bridges that need to be added so that there exists a route that crosses each bridge exactly once?
Road Networks as Graphs
Outline of Today’s Class
Graphs
Graph Representations
Solving Problems with Graph Algorithms
Elementary Graph Algorithms for Searching

Definition of a Graph
Omnipresence of Graphs

Airline Routes as Graphs

Figure: http://www.airlineroutemaps.com/
Social Networks as Graphs

Figure: http://hbr.idnet.net/images/
The Internet as a Graph

Visualization of the various routes through a portion of the Internet.

Figure: Credit: Matt Britt

Figure: Credit: Young Hyun, CAIDA
Websites as Graphs

Figure: http://www.google.com

Figure: http://www.cs.gmu.edu

Figure: Credit: Marcel Salathe
http://www.aharef.info

Figure: http://www.apple.com
Biological Networks as Graphs

Figure: Adapted from A. Barabasi, University of Notre Dame
Applications of Graphs

- Compilers
- Databases
- Neural Networks
- Machine Learning
- Artificial Intelligence
- Robotics
- Computational Biology
- ...
Formal Definition of a Graph

A graph \( G = (V, E) \) is a pair consisting of:

- a set \( V \) of vertices (or nodes)
- a set \( E \subseteq V \times V \) of edges (or arcs)
  - edge \( e_i \in E \) is a pair \((u, v)\) connecting vertices \( u \) and \( v \)

A graph \( G = (V, E) \) is:

- directed (referred to as a digraph) if \( E \) is a set of ordered pairs of vertices. The edges here are often referred to as directed edges or arrows.
- undirected if \( E \) is a set of unordered pairs of vertices.
- weighted if there are weights associated with the edges.
Illustrations of Types of Graphs

Figure: undirected graph

Figure: directed graph

Figure: multigraph

Figure: weighted graph
General Definition of a Graph

In a graph $G = (V, E)$:

- $E$ may be a set of unordered pairs of vertices not necessarily distinct. More than one edge can connect two vertices.
- An edge in $E$ may connect more than two vertices.
- These graphs are referred to as multigraphs or pseudo-graphs.
Focusing on Simple Graphs

**Simple Graphs**

- A simple graph, or a strict graph, is an unweighted, undirected graph containing no loops or multiple edges.
- Given that $E \subseteq V \times V$, $|E| \in O(|V|^2)$.
- If a graph is connected, $|E| \geq |V| - 1$.
- Combining the two, show that $\lg(|E|) \in \theta(\lg(|V|))$. 

---

Outline of Today’s Class
- Graphs
- Graph Representations
- Solving Problems with Graph Algorithms
- Elementary Graph Algorithms for Searching

Definition of a Graph
- Omnipresence of Graphs
A subgraph $H$ of $G = (V, E)$ is $H = (V_1, E_1)$ where $V_1 \subseteq V$ and $E_1 \subseteq E$, where $\forall e = (k, j) \in E_1$, $k, j \in V_1$.

A path is a sequence of vertices, where each pair of successive vertices is connected by an edge.

The length of the path is the number of edges in the path.

A simple path contains unique vertices.

A cycle is a simple path with the same first and last vertex.

Two vertices are adjacent if they are connected by an edge.

The neighbors of a vertex are all the vertices adjacent to it.

The degree of a vertex is the number of its neighbors.

A graph is connected if $\exists$ a path between every pair of vertices.

A tree is a connected graph with no cycles.
Graph Representations

- A graph can be represented as an **adjacency list**.
- A graph can be represented as an **adjacency matrix**.
Adjacency List Representation

```
struct elist
{
  int vto;
  struct elist *next;
};
```

```
struct vlist
{
  int v;
  elist *edges;
  struct vlist *next;
};
```
## Basic Graph Functionality

<table>
<thead>
<tr>
<th>Function</th>
<th>Adjacency List</th>
</tr>
</thead>
<tbody>
<tr>
<td>find($v$)</td>
<td>$O(</td>
</tr>
<tr>
<td>hasVertex($v$)</td>
<td>$O(\text{find}(v))$</td>
</tr>
<tr>
<td>hasEdge($v_i$, $v_j$)</td>
<td>$O(\text{find}(v_i) + \text{deg}(v_i))$</td>
</tr>
<tr>
<td>insertVertex($v$)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertEdge($v_i$, $v_j$)</td>
<td>$O(\text{find}(v_i))$</td>
</tr>
<tr>
<td>removeVertex($v$)</td>
<td>$O(</td>
</tr>
<tr>
<td>removeEdge($v_i$, $v_j$)</td>
<td>$O(\text{find}(v_i) + \text{deg}(v_i))$</td>
</tr>
<tr>
<td>outEdges($v$)</td>
<td>$O(\text{find}(v) + \text{deg}(v))$</td>
</tr>
<tr>
<td>inEdges($v$)</td>
<td>$O(</td>
</tr>
<tr>
<td>overall memory</td>
<td>$O(</td>
</tr>
</tbody>
</table>

**In undirected graphs:**

In undirected graphs: 
$|\text{elist}[v]| = \text{degree}(v)$.

**In digraphs:**

In digraphs:
$|\text{elist}[v]| = \text{out-degree}(v)$.

**Handshaking Lemma:** $\sum_{v \in V} |\text{elist}(v)| = 2|E|$ for undirected graphs. $O(|V| + |E|)$ storage $\Rightarrow$ **sparse** representation.
The adjacency list of a vertex can be implemented as a linked list.
The list of vertices themselves can be implemented using:
- A linked list
- A binary search tree
- A hash table

In a standard implementation, each edge list has two fields, a data field and a pointer:
- The data field contains adjacent vertex name and edge information
- The pointer points to next adjacent vertex
### Adjacency Matrix Representation

**Definition:**

\[ M[i][j] = 1 \ \text{iff} \ (v_i, v_j) \in E \]

**Example:**

<table>
<thead>
<tr>
<th></th>
<th>(v_0)</th>
<th>(v_1)</th>
<th>(v_2)</th>
<th>(v_3)</th>
<th>(v_4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v_0)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>(v_1)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>(v_2)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(v_3)</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(v_4)</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**C++ Code:**

```cpp
bool M[n][n];
bool **M;
using namespace std;

vector<vector<bool>> M;
```
# Basic Graph Functionality

<table>
<thead>
<tr>
<th>Function</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>find(v)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>hasVertex(v)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>hasEdge(v_i, v_j)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertVertex(v)</td>
<td>$O(</td>
</tr>
<tr>
<td>insertEdge(v_i, v_j)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>removeVertex(v)</td>
<td>$O(</td>
</tr>
<tr>
<td>removeEdge(v_i, v_j)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>outEdges(v)</td>
<td>$O(</td>
</tr>
<tr>
<td>inEdges(v)</td>
<td>$O(</td>
</tr>
<tr>
<td>overall memory</td>
<td>$O(</td>
</tr>
</tbody>
</table>

$O(|V|^2)$ storage ⇒ **dense** representation.
## Comparing The Two Representations

<table>
<thead>
<tr>
<th>Function</th>
<th>Adjacency List</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>find(v)</code></td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td><code>hasVertex(v)</code></td>
<td>$O(\text{find}(v))$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>hasEdge(v_i, v_j)</code></td>
<td>$O(\text{find}(v_i) + \deg(v_i))$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>insertVertex(v)</code></td>
<td>$O(1)$</td>
<td>$O(</td>
</tr>
<tr>
<td><code>insertEdge(v_i, v_j)</code></td>
<td>$O(\text{find}(v_i))$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>removeVertex(v)</code></td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td><code>removeEdge(v_i, v_j)</code></td>
<td>$O(\text{find}(v_i) + \deg(v_i))$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>outEdges(v)</code></td>
<td>$O(\text{find}(v) + \deg(v))$</td>
<td>$O(</td>
</tr>
<tr>
<td><code>inEdges(v)</code></td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td>overall memory</td>
<td>$O(</td>
<td>V</td>
</tr>
</tbody>
</table>
## Alternative Graph Representations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast to query</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>[hasVertex, hasEdge]</td>
<td></td>
</tr>
<tr>
<td>Fast to scan</td>
<td>$O(</td>
</tr>
<tr>
<td>[outEdges]</td>
<td></td>
</tr>
<tr>
<td>Fast to insert</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>[insertVertex, insertEdge]</td>
<td></td>
</tr>
<tr>
<td>Fast to remove</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>[removeEdge]</td>
<td></td>
</tr>
</tbody>
</table>
Graph Representation: Hash Map

- Vertex set as a hash map
  - key: vertex
  - data: outgoing edges
- Outgoing edges of each vertex as a hash set

```cpp
using namespace std_ext;

hash_map<key, hash_set<key> >

vertex outgoing edges
```

```
<table>
<thead>
<tr>
<th>vertex</th>
<th>hash_set</th>
</tr>
</thead>
<tbody>
<tr>
<td>v0</td>
<td>v1, v2, v4</td>
</tr>
<tr>
<td>v1</td>
<td>v1</td>
</tr>
<tr>
<td>v2</td>
<td>v1, v3</td>
</tr>
<tr>
<td>v3</td>
<td>v4</td>
</tr>
<tr>
<td>v4</td>
<td>v0, v2, v3</td>
</tr>
</tbody>
</table>
```
### Comparing The Three Representations

<table>
<thead>
<tr>
<th>Function</th>
<th>Adj. List</th>
<th>Adj. Matrix</th>
<th>Hash Map</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>find(v)</code></td>
<td>(O(</td>
<td>V</td>
<td>))</td>
</tr>
<tr>
<td><code>hasVertex(v)</code></td>
<td>(O(</td>
<td>V</td>
<td>))</td>
</tr>
<tr>
<td><code>hasEdge(v_i, v_j)</code></td>
<td>(O(</td>
<td>V</td>
<td>+ \text{deg}(v_i)))</td>
</tr>
<tr>
<td><code>insertVertex(v)</code></td>
<td>(O(1))</td>
<td>(O(</td>
<td>V</td>
</tr>
<tr>
<td><code>insertEdge(v_i, v_j)</code></td>
<td>(O(</td>
<td>V</td>
<td>))</td>
</tr>
<tr>
<td><code>removeVertex(v)</code></td>
<td>(O(</td>
<td>V</td>
<td>+</td>
</tr>
<tr>
<td><code>removeEdge(v_i, v_j)</code></td>
<td>(O(</td>
<td>V</td>
<td>+ \text{deg}(v_i)))</td>
</tr>
<tr>
<td><code>outEdges(v)</code></td>
<td>(O(</td>
<td>V</td>
<td>+ \text{deg}(v)))</td>
</tr>
<tr>
<td><code>inEdges(v)</code></td>
<td>(O(</td>
<td>V</td>
<td>+</td>
</tr>
<tr>
<td>overall memory</td>
<td>(O(</td>
<td>V</td>
<td>+</td>
</tr>
</tbody>
</table>
Graph modeling: Problem Solving with Graph Algorithms

- Identify the vertices and the edges in your problem formulation
- Identify the objective of the problem
- State this objective in graph terms
- Implementation:
  - Construct the graph from the input instance
  - Run the suitable graph algorithm on the graph
  - Convert the output into a suitable/required format
Find a path from initial vertex to goal vertex

- Uninformed graph search
  - Depth-first Search (DFS)
  - Breadth-first Search (BFS)
  - Depth-limited search (DLS)
  - Iterative Deepening Search (IDS)

Find the shortest path from initial vertex to goal vertex

- Informed graph search
  - Dijkstra [Edsger Dijkstra 1959]
  - B* [Hans Berliner 1979]
  - Best-First Search [Judea Pearl 1984]
Characteristics of Uninformed Graph Search/Traversal:

- There is no additional information about states/vertices beyond what is provided in the problem definition.
- All that the search does is generate successors/neighbors and distinguish a goal state from a nongoal state.

Correctness A systematic search is needed to traverse all paths from initial vertex so the goal vertex is not missed.

Efficiency No edge should be traversed more than twice.

The systematic search “lays out” all paths from initial vertex; it traverses the spanning tree of the graph.
Uninformed Graph Search

S: search data structure — parent: edge comes from

1: S.insert(v)
2: parent[v] ← true
3: while not S.isEmpty do
4: u ← S.extract()
5: if isGoal(u) then
6: return true
7: for each v in outEdges(u) do
8: if no parent[v] then
9: S.insert(v)
10: parent[v] ← u
Depth-first Search (DFS)

1. $S$.insert($v$)
2. $parent[v] \leftarrow$ true
3. while not $S$.isEmpty do
   4. $u \leftarrow S$.extract()
   5. if $isGoal(u)$ then
      6. return true
   7. for each $v$ in outEdges($u$) do
      8. if no $parent[v]$ then
         9. $S$.insert($v$)
   10. $parent[v] \leftarrow u$
Characteristics and Problems with DFS

Basic Behavior:
- Expands the deepest node in the tree
- Backtracks when reaches a non-goal node with no descendants

Problems:
- Make a wrong choice and can go down along an infinite path even though the solution may be very close to initial vertex
- Hence, DFS is not optimal
- If subtree is of unbounded depth and contains no solutions, DFS will never terminate.
- Hence, DFS is not complete
- Let $b$ be the maximum number of successors of any node (known as branching factor), $d$ be depth of shallowest goal, and $m$ be maximum length of any path in the spanning tree
- Time complexity is $O(b^m)$ and space complexity is $O(b \cdot m)$
Pros of DFS

RecursiveDFS(v)

1: if v is unmarked then
2: mark v
3: for each edge v, u do
4: RecursiveDFS(u)

Color arrays can be kept to indicate that a vertex is undiscovered, the first time it is discovered, when its neighbors are in the process of being considered, and when all its neighbors have been considered.

DFS can be used to timestamp vertices with when they are discovered and when they are finished. These start and finish times are useful in various applications of DFS such as topological sorting. We will come back to topological sorting.
Breadth-first Search (BFS)

S: queue(first-in, first-out) — parent: edge comes from

1: S.insert(v)
2: parent[v] ← true
3: while not S.isEmpty do
4: u ← S.extract()
5: if isGoal(u) then
6: return true
7: for each v in outEdges(u) do
8: if no parent[v] then
9: S.insert(v)
10: parent[v] ← u
Characteristics and Problems with BFS

Basic Behavior:
- Expands all nodes at depth $d$ before those at depth $d + 1$
- The sequence is root, then children, then grandchildren in the spanning tree.

Problems:
- If the path cost is a non-decreasing function of the depth of the goal node, then BFS is optimal.
- A graph with no weights can be considered to have edges of weight 1. In this case, BFS is optimal.
- BFS will find shallowest goal after expanding all shallower nodes (if branching factor is finite). Hence, BFS is complete.
- BFS is not very popular because time and space complexity are exponential: $O(b^{d+1})$ and $O(b^{d+1})$, respectively.
- Memory requirements of BFS are a bigger problem.
Depth-limited Search (DLS)

- Problem with DFS is presence of infinite paths
- DLS limits the depth of a path in DFS
- Modifies DFS by using a predetermined depth limit \( d_l \)
- DLS is still incomplete if the shallowest goal is beyond the depth limit \( d_l \)
- DLS is not optimal if \( d < d_l \)
- Time complexity is \( O(b^{d_l}) \) and space complexity is \( O(b \cdot d_l) \)
Iterative Deepening Search (IDS)

- Finds the best depth limit by incrementing $d_l$ until goal is found at $d_l = d$
- Can be viewed as running DLS with consecutive values of $d_l$
- IDS combines the benefits of both DFS and BFS
- Like DFS, its space complexity is $O(b \cdot d)$
- Like BFS, it is complete when the branching factor is finite, and it is optimal if the path cost is a non-decreasing function of the depth of the goal node
- Its time complexity is $O(b^d)$
- IDS is the preferred uninformed search when the search space is large and the depth of the solution is not known
First Graph Problem

Seven Bridges of Koenigsberg [1736]:
Find a route that crosses each bridge exactly once.
Posed by Leonard Euler [1707 - 1783].

If and only if there are exactly two or zero nodes of odd degree
Graph-Search Problem: Knight’s Tour

Knight must visit all cells but exactly once

Figure: http://www.chess-poster.com

Figure: http://www.chilton-computing.org.uk
Graph-Search Problem: 8-Queens

Place eight queens such that none of them can capture another.

Figure: http://www.sics.se/sicstus
Graph-Search Problem: 8-Puzzle

Given an initial configuration of eight sliding tiles on a 3 x 3 board, slide them so that in the end configuration the tiles are placed in a specific order.

Start State

Goal State

Figure: http://cs.millersville.edu/webster/cs406java/8puzzle.jpg
Graph-Search Problem: Sudoku
Graph-Search Problem: Maze Search
The pirates of a ship in the Mediterranean just looted a small island off the Adriatic coast. They got away with a bag of $N$ items.

Realizing that some items are more precious than others, they are trying to come up with a fair scheme to distribute them.

Surprisingly, these pirates are fair and sensitive to bullying. They think the rule should be simple: if pirate $A$ has ever bullied pirate $B$, $B$ gets to choose before $A$.

Can you help them out?

This problem is known as Topological Sort. We will come back to it next class.