ECE 590.01
C++ Programming, Data structures, and Algorithms

Graphs I
Admin

- Reading
  - Chapter 9
  - Skipping 8 (optional reading if you want)

- Homework 4 out soon
  - Due April 5 (Almost 3 weeks from now)

- Project
  - Should be making some solid progress here
  - Don’t wait until the last minute!
What have we been talking about?

- What did we talk about last time?
What have we been talking about?

- What did we talk about last time?
  - Sorting
    - 6 different ways
    - And their tradeoffs
    - All in a ramp-up
  - For some reason, people love to ask sorting questions in interviews
    - Maybe because its so fundamental
Today: Graphs

- Graph ADT
  - Nodes + Edges
    - Edges connect two nodes
      - May or may not be weighted
      - May or may not be directional
Today: Graphs

- **Graph ADT**
  - Nodes + Edges
    - Edges connect two nodes
      - May or may not be **weighted**
      - May or may not be directional
Today: Graphs

- Graph ADT
  - Nodes + Edges
  - Edges connect two nodes
    - May or may not be weighted
    - May or may not be directional
Graphs: Many applications

- Graphs show up all over CS/CE
  - Networks (routing, etc)
  - Social Networks
  - Resource Allocation (register allocation)
  - Scheduling
  - Optimization
  - AI
  - ...
A few special types of graphs

- We’ve seen binary trees:
  - Trees are a special type of graphs
    - And binary trees are a special type of trees
  - Tree = undirected connected graph with no cycle
    - Connected: Path from every node to every other node
    - Cycle: No “loops”: You can’t get from a node back to itself without re-using an edge you already traversed
A few special types of graphs

- Forest:
  - Multiple trees
  - This one has 3
Directed Acyclic Graphs (DAGs)

- Directed graph with no cycle
  - Following edge directions: may look like it has loops ignoring the directions
  - Very common/useful
    - Ubiquitous in situations with dependencies
      - E depends on B and C
Directed Acyclic Graphs (DAGs)

- DAGs and computer architecture/compilers
  - Dataflow graph
    - Dependences between instructions
    - Could weight edges based on execution latency
      - Compilers use this for scheduling instructions

A: ld [r1] -> r2
B: ld 4[r2] -> r3
C: ld 8[r2] -> r4
D: ld 12[r2] -> r5
E: add r3 + r4 -> r6
F: addi r5 + 4 -> r7
G: neg r5 -> r8
H: addi r6 + 4 -> r1
I: mul r7 * r8 -> r9
J: stw [r1] <- r9
Graph: ADT

- Operations we might want:
  - void add_node(n)  //maybe void del_node(n) too
  - void add_edge(n1, n2, weight)
    - If weighted, if not, no weight
    - Directed? Adds on direction (n1 -> n2)
    - Un-directed: n1 <-> n2
Graph: ADT

- Operations we might want:
  - `void add_node(n)` //maybe `void del_node(n)` too
  - `void add_edge(n1,n2, weight)`
  - `void del_edge(n1,n2)`
  - `int get_edge_weight(n1,n2)` //infinity if no edge
    - `bool has_edge(n1,n2)` //if no weights
  - `Set<Node> get_adjacent_nodes(n)`
Graph: ADT

- Operations we might want:
  - void add_node(n) //maybe void del_node(n) too
  - void add_edge(n1,n2, weight)
  - void del_edge(n1,n2)
  - int get_edge_weight(n1,n2) //infinity if no edge
    - bool has_edge(n1,n2) //if no weights
  - Set<Node> get_adjacent_nodes(n)

How might we implement this?
Implementation 1: Adjacency Matrix

• One approach:
  • Adjacency matrix:
    • Row per node ("from")
    • Column per node ("to")
    • Entry is weight along that edge
# Implementation 1: Adjacency Matrix

Below is the adjacency matrix representation of the graph.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>∞</td>
<td>∞</td>
<td>2</td>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The graph visualizes the connections and weights between nodes.

- A to C: 2
- C to A: 3
- C to B: 9
- D to B: 7
- B to G: 17
- G to F: 1

The adjacency matrix shows the direct connections and their weights in the graph.
Implementation 1: Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>∞</td>
<td>∞</td>
<td>2</td>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>9</td>
<td>∞</td>
<td>∞</td>
<td>7</td>
<td>∞</td>
<td>2</td>
<td>∞</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Implementation 1: Adjacency Matrix

![Graph Diagram]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>2</td>
<td>3</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td><strong>B</strong></td>
<td>9</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>7</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>2</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>1</td>
<td>4</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td><strong>D</strong></td>
<td>$\infty$</td>
<td>7</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td><strong>E</strong></td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>4</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td><strong>G</strong></td>
<td>5</td>
<td>17</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>
Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>∞</td>
<td>∞</td>
<td>2</td>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>9</td>
<td>∞</td>
<td>∞</td>
<td>7</td>
<td>∞</td>
<td>2</td>
<td>∞</td>
</tr>
<tr>
<td>C</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>1</td>
<td>4</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
<td>7</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>E</td>
<td>∞</td>
<td>∞</td>
<td>4</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>F</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>5</td>
<td>17</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

- How must space does this take and fast are these?
  - void add_node(n)  //maybe void del_node(n) too
  - void add_edge(n1, n2, weight)
  - void del_edge(n1, n2)
  - int get_edge_weight(n1, n2) //infinity if no edge
    - bool has_edge(n1, n2) //if no weights
  - Set<Node> get_adjacent_nodes(n)
Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>∞</td>
<td>∞</td>
<td>2</td>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>B</td>
<td>9</td>
<td>∞</td>
<td>∞</td>
<td>7</td>
<td>∞</td>
<td>2</td>
<td>∞</td>
</tr>
<tr>
<td>C</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>1</td>
<td>4</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
<td>7</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>E</td>
<td>∞</td>
<td>∞</td>
<td>4</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>F</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>5</td>
<td>17</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

- How much space does this take and how fast are these? $O(V^2)$
- void add_node(n) // maybe void del_node(n) too $O(V^2)$
- void add_edge(n1,n2, weight) $O(1)$
- void del_edge(n1,n2) $O(1)$
- int get_edge_weight(n1,n2) // infinity if no edge $O(1)$
- bool has_edge(n1,n2) // if no weights
- Set<Node> get_adjacent_nodes(n) $O(V)$
Adjacency List

- Each node holds a list (vector, array, linked list) of edges
  - Each edge has the weight and a pointer to the “to” node

- How much space does this take and fast are these?
  - void add_node(n)  //maybe void del_node(n) too
  - void add_edge(n1,n2, weight)
  - void del_edge(n1,n2)
  - int get_edge_weight(n1,n2) //infinity if no edge
    - bool has_edge(n1,n2) //if no weights
  - Set<Node> get_adjacent_nodes(n)
Adjacency List

- Each node holds a list (vector, array, linked list) of edges
  - Each edge has the weight and a pointer to the “to” node

- How much space does this take and how fast are these? \(O(V+E)\)
  - void add_node(n) \(O(1)\) // maybe void del_node(n) too \(O(V+E)\)
  - void add_edge(n1,n2, weight) \(O(1)\)
  - void del_edge(n1,n2) \(O(V)\)
  - int get_edge_weight(n1,n2) // infinity if no edge \(O(V)\)
    - bool has_edge(n1,n2) // if no weights
  - Set<Node> get_adjacent_nodes(n) \(O(V) \text{ or } O(1)\)
Adjacency List

- Each node holds a list (vector, array, linked list) of edges
  - Each edge has the weight and a pointer to the “to” node

Might be able to improve with Adjacency Tree or Hash Table? (Really, each “from” node is storing a map from “to” Nodes -> weights)

- How must space does this take and fast are these? $O(V+E)$
  - void add_node(n) $O(1)$ //maybe void del_node(n) too $O(V+E)$
  - void add_edge(n1,n2, weight) $O(1)$
  - void del_edge(n1,n2) $O(V)$
  - int get_edge_weight(n1,n2) //infinity if no edge $O(V)$
    - bool has_edge(n1,n2) //if no weights
  - Set<Node> get_adjacent_nodes(n) $O(V)$ or $O(1)$
Graphs: our plans for them

- Rest of today: a few things you should be conversant in
  - Traveling Salesperson
  - NP completeness
  - Graph coloring
- Next two lectures:
  - Depth First Search (DFS)
  - Breadth First Search (BFS)
  - Dijkstra’s Shortest Path Algorithm
  - Minimum Spanning Tree
  - Strongly Connected Components
  - Topological Sort (DAGs)
Traveling Salesperson

- Need to travel to some cities to make sales
  - Start at home (e.g., RDU) and return there
  - Visit each city once
  - Want to minimize cost of entire trip (know cost between each pair)

[not all shown due to space on slide]
Traveling Salesperson

- How would you do this?
  - What would your algorithm look like?
  - What would its runtime be?
Traveling Salesperson

- Exhaustive search: try all possibilities
  - Runtime?
Traveling Salesperson

- Exhaustive search: try all possibilities
  - Runtime: start/end city is fixed
    - Then 8 choices for 1st city
    - Each of those leaves 7 for the 2nd...
  - $8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 40,320$ not bad...

ECE 590.01 (Hilton): Graphs I
Traveling Salesperson

• Exhaustive search: try all possibilities
  • Runtime: start/end city is fixed
    • Then 8 choices for 1\textsuperscript{st} city
    • Each of those leaves 7 for the 2\textsuperscript{nd}...
  • $8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 40,320$ not bad...

• Expand it to 20 cities?
  • $19 \times 18 \times 17 \times 16 \times \ldots = 19!$
    • Can someone work that out really quickly for me?
Traveling Salesperson

- Exhaustive search: try all possibilities
  - Runtime: start/end city is fixed
    - Then 8 choices for 1\textsuperscript{st} city
    - Each of those leaves 7 for the 2\textsuperscript{nd}...
  - $8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 40,320$ not bad...

- Expand it to 20 cities?
  - $19 \times 18 \times 17 \times 16 \times \ldots = 19! = 24,329,020,081,766,400$
    - Can someone work that out really quickly for me?
      - That’s 24.3 quadrillion (i.e., 24,329 trillion)
    - If our computer can do 1 Billion possible options per second, how long will this take?
Traveling Salesperson

- Exhaustive search: try all possibilities
  - Runtime: start/end city is fixed
    - Then 8 choices for 1\textsuperscript{st} city
    - Each of those leaves 7 for the 2\textsuperscript{nd}...
  - $8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 40,320$ not bad...

- Expand it to 20 cities?
  - $19 \times 18 \times 17 \times 16 \times \ldots = 19 \times 24,329,020,081,766,400$
    - Can someone work that out really quickly for me?
      - That’s 24.3 quadrillion (i.e., 24,329 trillion)
    - If our computer can do 1 Billion possible options per second, how long will this take?
      - **About 9 months** (plan your trip well in advance)
        - Adding one more city $\Rightarrow$ 20x as long $\Rightarrow$ 15 years
TSP: continued

- Ok, so our first algorithm only works for ~10 cities
  - How much traveling do we really want to do?

- Can we do better?
TSP: continued

- Ok, so our first algorithm only works for ~10 cities
  - How much traveling do we really want to do?
    - Actual traveling on a plan: maybe this is fine
    - But other real problems are the “same” math problem

- Can we do better?
TSP: continued

- Ok, so our first algorithm only works for ~10 cities
  - How much traveling do we really want to do?
    - Actual traveling on a plan: maybe this is fine
    - But other real problems are the “same” math problem
  - Can we do better?
    - Ok, so we thought about this for 5 minutes, maybe if we think harder, we’ll come up with a nice fast algorithm...
TSP: continued

• Ok, so our first algorithm only works for ~10 cities
  • How much traveling do we really want to do?
    • Actual traveling on a plan: maybe this is fine
    • But other real problems are the “same” math problem

• Can we do better?
  • Ok, so we thought about this for 5 minutes, maybe if we think harder, we’ll come up with a nice fast algorithm...
    • No, we probably won’t do better than exponential 😞
NP Completeness

• There is a very large (and very useful) class of NP complete problems
  • To be precise: these are decision problems (yes/no answer)
    • TSP as a decision problem: is there a travel plan with cost at most N?
  • Best known algorithm: exponential time
    • Many very smart people have tried to come up with better
  • A “yes” answer can be verified in polynomial time with some (polynomial amount of) extra information
    • E.g., “look here is a path of cost N”
  • Solving one problem in polynomial time, solves them ALL!
    • Even though very diverse
      • http://en.wikipedia.org/wiki/List_of_NP-complete_problems
NP Completeness continued

• Two non-graph NP-complete problems
  • Both good for the traveling salesperson to use for his/her trip

• Bin packing:
  • Have many suitcases of fixed volume
  • Need to pack lots of stuff in them
  • Want to minimize number of suitcases used

• Knapsack problem:
  • Have only one suitcase (with fixed volume)
  • Can assign a value (importance) to your items
  • Pick set of items to pack that
    • Fits in suitcase
    • Maximizes total value
Graph Coloring

- Back to graphs (and not travel related):
  - Graph coloring
    - Color each node so that no two adjacent nodes are the same color
    - Use as few colors as possible (here: 4)
Graph Coloring

- Also NP complete
  - Decision problem: Can a graph be $k$ colored?
    - Can this one be four colored: yes (see above)
    - Can this one be three colored: no
Graph Coloring Applications

- Graph Coloring:
  - Common in resource allocation problems
  - Node = thing that uses resource
  - Edge = conflict between uses of same resource
  - Color = resource

- Valid coloring:
  - No conflicting uses of same resource
    - Adjacent nodes different colors
  - Few colors:
    - Use fewer resources

- Example: compilers allocate registers
  - Variables can’t be in use at same time if in same register
Graphs: Searching

• Next time: graph search
  • Finding a path from one node to another.