Announcements

• Upcoming talk
  • “The Many Careers of a Computer Scientist”
  • Or how a Computer Science degree empowers you to do much more than code
  • Dan Huttenlocher, Professor in the Department of Computer Science and Johnson Graduate School of Management
  • 5:00 PM, Wednesday, November 9th
  • Upson Lounge
  • FREE PIZZA!

• ACSU (Association of Computer Science Undergraduates)

This is not a Graph

These are Graphs

Applications of Graphs

• Communication networks
• Routing and shortest path problems
• Commodity distribution (flow)
• Traffic control
• Resource allocation
• Geometric modeling
• ...

Graph Definitions

• A directed graph (or digraph) is a pair \((V, E)\) where
  • \(V\) is a set
  • \(E\) is a set of ordered pairs \((u, v)\) where \(u, v \in V\)
    • Usually require \(u \neq v\) (no self-loops)
• An element of \(V\) is called a vertex (pl. vertices) or node
• An element of \(E\) is called an edge or arc

• \(|V| = \text{size of } V\), often denoted \(n\)
• \(|E| = \text{size of } E\), often denoted \(m\)
**Example Directed Graph**

Example:

\[
V = \{a,b,c,d,e,f\} \\
E = \{(a,b), (a,c), (a,e), (b,c), (b,d), (b,e), (c,d), (c,f), (d,e), (d,f), (e,f)\}
\]

|V| = 6, |E| = 11

---

**Example Undirected Graph**

An undirected graph is just like a directed graph, except the edges are unordered pairs (sets) \{(u,v)\}

Example:

\[
V = \{a,b,c,d,e,f\} \\
E = \{(a,b), (a,c), (a,e), (b,c), (b,d), (b,e), (c,d), (c,f), (d,e), (d,f), (e,f)\}
\]

---

**Some Graph Terminology**

- Vertices u and v are called the source and sink of the directed edge \((u,v)\), respectively
- Vertices u and v are called the endpoints of \((u,v)\)
- Two vertices are adjacent if they are connected by an edge
- The outdegree of a vertex u in a directed graph is the number of edges for which u is the source
- The indegree of a vertex v in a directed graph is the number of edges for which v is the sink
- The degree of a vertex u in an undirected graph is the number of edges of which u is an endpoint

---

**More Graph Terminology**

- A path is a sequence \(v_0,v_1,v_2,...,v_p\) of vertices such that \((v_i,v_{i+1}) \in E\), \(0 \leq i \leq p - 1\)
- The length of a path is its number of edges
  - In this example, the length is 5
- A path is simple if it does not repeat any vertices
- A cycle is a path \(v_0,v_1,v_2,...,v_p\) such that \(v_0 = v_p\)
- A cycle is simple if it does not repeat any vertices except the first and last
- A graph is acyclic if it has no cycles
- A directed acyclic graph is called a dag

---

**Is this a dag?**

- Intuition: If it’s a dag, there should be a “first” vertex (i.e., a vertex with indegree zero)
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

---

**Is this a dag?**

- Intuition: If it’s a dag, there should be a “first” vertex (i.e., a vertex with indegree zero)
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears
Is this a dag?

- Intuition: If it’s a dag, there should be a “first” vertex (i.e., a vertex with indegree zero)
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

Is this a dag?

- Intuition: If it’s a dag, there should be a “first” vertex (i.e., a vertex with indegree zero)
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

Is this a dag?

- Intuition: If it’s a dag, there should be a “first” vertex (i.e., a vertex with indegree zero)
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

Is this a dag?

- Intuition: If it’s a dag, there should be a “first” vertex (i.e., a vertex with indegree zero)
- This idea leads to an algorithm
  - A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears
Is this a dag?

• Intuition: If it’s a dag, there should be a “first” vertex (i.e., a vertex with indegree zero)
• This idea leads to an algorithm
  • A digraph is a dag if and only if we can iteratively delete indegree-0 vertices until the graph disappears

Topological Sort

• Just computed a topological sort of the dag
  • A numbering of the vertices such that all edges go from lower- to higher-numbered vertices

Graph Coloring

• A coloring of an undirected graph is an assignment of a color to each node such that no two adjacent vertices get the same color

• How many colors are needed to color this graph?

Graph Coloring

• A coloring of an undirected graph is an assignment of a color to each node such that no two adjacent vertices get the same color

• How many colors are needed to color this graph? • 3

An Application of Coloring

• Vertices are jobs
• Edge (u,v) is present if jobs u and v each require access to the same shared resource, and thus cannot execute simultaneously
• Colors are time slots to schedule the jobs
• Minimum number of colors needed to color the graph = minimum number of time slots required

Planarity

• A graph is planar if it can be embedded in the plane with no edges crossing

• Is this graph planar?
Planarity

• A graph is planar if it can be embedded in the plane with no edges crossing

• Is this graph planar?
  • Yes

Detecting Planarity

Kuratowski’s Theorem

K_5

K_{3,3}

A graph is planar if and only if it does not contain a copy of K_5 or K_{3,3} (possibly with other nodes along the edges shown)

The Four-Color Theorem

Every planar graph is 4-colorable
(Appel & Haken, 1976)

Bipartite Graphs

• A directed or undirected graph is bipartite if the vertices can be partitioned into two sets such that all edges go between the two sets

• The following are equivalent
  • G is bipartite
  • G is 2-colorable
  • G has no cycles of odd length

Bipartite Graphs
### Traveling Salesperson

Find a path of minimum distance that visits every city.

### Implementing Digraphs

- **Adjacency Matrix**
  - $g[u][v]$ is true if there is an edge from $u$ to $v$
  - Example:

### Implementing Weighted Digraphs

- **Adjacency Matrix**
  - $g[u][v]$ is $c$ if there is an edge of cost $c$ from $u$ to $v$
  - Example:

### Implementing Undirected Graphs

- **Adjacency Matrix**
  - $g[u][v]$ is true if there is an edge from $u$ to $v$
  - Example:

### Adjacency Matrix or Adjacency List?

- **Adjacency Matrix**
  - Uses space $O(n^2)$
  - Can iterate over all edges in time $O(n^2)$
  - Can answer “Is there an edge from $u$ to $v$?” in $O(1)$ time
  - Better for dense (i.e., lots of edges) graphs

- **Adjacency List**
  - Uses space $O(m+n)$
  - Can iterate over all edges in time $O(m+n)$
  - Can answer “Is there an edge from $u$ to $v$?” in $O(m_u)$ time
  - Better for sparse (i.e., fewer edges) graphs

### Goal: Find Shortest Path in a Graph

- **Adjacency List**
  - The list for $u$ contains $v$ if there is an edge from $u$ to $v$
  - Example:

- **Adjacency Matrix**
  - $g[u][v]$ is true if there is an edge from $u$ to $v$
  - Example:

- **Adjacency List**
  - The list for $u$ contains $v$ if there is an edge from $u$ to $v$
  - Example:

- Finding the shortest (min-cost) path in a graph is a problem that occurs often
  - Find the least-cost route between Ithaca and Detroit
  - Result depends on our notion of cost
    - least mileage
    - least time
    - cheapest
    - least boring
  - All of these “costs” can be represented as edge costs on a graph
  - How do we find a shortest path?

Shortest Paths for Unweighted Graphs

bfsDistance(s):
// s is the start vertex
// dist[v] is length of s-to-v path
// Initially dist[v] = ∞ for all v
dist[s] = 0;
Q.insert(s);
while (Q nonempty) {
v = Q.get();
for (each w adjacent to v) {
if (dist[w] == ∞) {
dist[w] = dist[v] + 1;
Q.insert(w);
}
}
}

Analysis for bfsDistance
• How many times can a vertex be placed in the queue?
• How much time for the for-loop?
  • Depends on representation
    • Adjacency Matrix: O(n^2)
    • Adjacency List: O(m + n)

If There are Edge Costs?
• Idea #1
  • Add false nodes so that all edge costs are 1
  • But what if edge costs are large?
  • What if the costs aren’t integers?
• Idea #2
  • Nothing “interesting” happens at the false nodes
  • Can’t we just jump ahead to the next “real” node
  • Rule: always do the closest (real) node first
  • Report answers
    • Keep track of what to do next

Dijkstra’s Algorithm
• Intuition
  • Edges are threads; vertices are beads
  • Pick up at s, mark each node as it leaves the table
• Note: Negative edge-costs are not allowed
  • s is the start vertex
  • c(i,j) is the cost from i to j
  • Initially, vertices are unmarked
  • dist[v] is length of s-to-v path
  • Initially, dist[v] = ∞, for all v
dijkstra(s):
dist[s] = 0;
while (some vertices are unmarked) {
v = unmarked vertex with smallest dist;
Mark v;
for (each w adj to v) {
dist[w] = min ( dist[w], dist[v] + c(v,w) );
}
}

Dijkstra’s Algorithm using Adj Matrix
• While-loop is done n times
• Within the loop
  • Choosing v takes O(n) time
  • Could do this faster using PQ, but no reason to
  • For-loop takes O(n) time
• Total time = O(n^2)

Dijkstra’s Algorithm using Adj List
• Looks like we need a PQ
• Problem: priorities are updated as algorithm runs
  • Can insert pair (v.dist[v]) in PQ whenever dist[v] is updated
• At most m things in PQ
• Time: O(n + m log m)
  • Using a more complicated PQ (e.g., Pairing Heap), time can be brought down to O(n log n)

Dijkstra’s Algorithm for Digraphs

- Algorithm works on both undirected and directed graphs without modification.
- As before: Negative edge-costs are not allowed.

\[ s \] is the start vertex
\[ c(i,j) \] is the cost from i to j
Initially, vertices are unmarked
\[ \text{dist}[v] \] is length of \( s \)-to-\( v \) path
Initially, \( \text{dist}[v] = \infty \), for all \( v \)

\[
\text{dijkstra}(s):
\begin{align*}
\text{dist}[s] &= 0; \\
\text{while} \ (\text{some vertices are unmarked}) \ {\{} & \notag \\
\text{Mark} \ v; \\
\text{for} \ (\text{each} \ w \ \text{adj to} \ v) \ {\{} & \notag \\
\text{dist}[w] &= \min ( \text{dist}[w], \text{dist}[v] + c(v,w) ); \\
\text{dist}[w] &= 0; \\
\text{while} \ (\text{some vertices are unmarked}) \ {\{} & \notag \\
\text{Mark} \ v; \\
\text{for} \ (\text{each} \ w \ \text{adj to} \ v) \ {\{} & \notag \\
\text{dist}[w] &= \min ( \text{dist}[w], \text{dist}[v] + c(v,w) ); \\
\text{dist}[w] &= 0;
\end{align*}
\]

Greedy Algorithms

- Dijkstra’s Algorithm is an example of a Greedy Algorithm.
- The Greedy Strategy is an algorithm design technique.
- Like Divide & Conquer
- The Greedy Strategy is used to solve optimization problems.
- The goal is to find the best solution.
- Works when the problem has the greedy-choice property.
- A global optimum can be reached by making locally optimum choices.

- Problem: Given an amount of money, find the smallest number of coins to make that amount.
- Solution: Use a Greedy Algorithm.
- Give as many large coins as you can.
- This greedy strategy produces the optimum number of coins for the US coin system.
- Different money system \( \Rightarrow \) greedy strategy may fail.
- For example: suppose the US introduces a 4¢ coin.