More Graphs

Lecture 22
CS211 - Fall 2006

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 graphs (i.e., lots of edges)

- **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 graphs (i.e., fewer edges)

Goal: Find Shortest Path in a Graph

- Finding the shortest (min-cost) path in a graph is a problem that occurs often
  - Find the least-cost route between Ithaca and West Lafayette, IN
  - 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

```java
bfsDistance(s):
  // $s$ is the start vertex
  // $dist[v]$ is length of $s$-to-$v$ path
  // Initially $dist[v] = \infty$ for all $v$
  $dist[s] = 0$;
  Q.insert(s);
  while (Q nonempty) {
    $v = Q$.get();
    for (each $w$ adjacent to $v$) {
      if ($dist[w] == \infty$) {
        $dist[w] = dist[v]+1$;
        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: $O(n)$
    - Adjacency List: $O(m)$
  - Time:
    - $O(n^2)$ for adj matrix
    - $O(m+n)$ for adj list

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 we just jump ahead to the next real node?
  - Intuition
    - Edges are threads; vertices are beads
    - Pick up at $s$; mark each node as it leaves the table
    - Rule: always do the closest-to-$s$ node first
  - Use the array $dist[]$ to:
    - 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

\[ \text{dijkstra}(s): \]
\[ \text{dist}[s] = 0; \]
\[ \text{while (some vertices are unmarked)} { \]
\[ \quad v = \text{unmarked node with smallest dist}; \]
\[ \quad \text{Mark } v; \]
\[ \quad \text{for (each } w \text{ adj to } v) { \]
\[ \quad \quad \text{dist}[w] = \min\{\text{dist}[w],\text{dist}[v]+c(v,w)\}; \]
\[ \quad } \]
\[ } \]

Proof for Dijkstra's Algorithm

- Claim: When vertex \( v \) is marked, \( \text{dist}[v] \) is the length of the shortest path from \( s \) to \( v \)
- Proof
  - Suppose there is a shorter path \( P \) from \( s \) to \( v \)
  - Consider the first edge of \( P \) that links a marked vertex to an unmarked vertex
    - Such an edge must exist because we know \( s \) is marked and \( v \) is not
    - Call this edge \( (u',v') \)
  - Note that the length of the path from \( s \) to \( u' \) to \( v' \) is less than the length of \( P \)
  - Thus \( v' \) would be chosen in the algorithm instead of \( v \)
  - Contradiction!

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 \((v,\text{dist}[v])\) in PQ whenever \( \text{dist}[v] \) 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 \( O(m + n \log n) \)

Dijkstra's Algorithm for Digraphs

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

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 algorithms are 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 ⇒ greedy strategy may fail
      - Example: suppose the US introduced a 4¢ coin