Graph implementations

- **Typical operations on graph:**
  - Add vertex
  - Add edge (parameters?)
  - getAdjacent(vertex)
  - getVertices(..)
  - String->Vertex (vice versa)

- **Different kinds of graphs**
  - Lots of vertices, few edges, *sparse* graph
    - Use adjacency list
  - Lots of edges (max # ?) *dense* graph
    - Use adjacency matrix
Graph implementations (continued)

- Adjacency matrix
  - Every possible edge represented, how many?

- Adjacency list uses $O(V+E)$ space
  - What about matrix?
  - Which is better?

- What do we do to get adjacent vertices for given vertex?
  - What is complexity?
  - Compared to adjacency list?

- What about weighted edges?
Memoization

- **How do we avoid solving the same problem?**
  - Consider APT BSTs
  - Review student submission
  - Consider similarities to Fibonacci
    - See next slide

- **How to avoid cost?**

- **How is this relevant to APT?**
  - Create map of parameter to solution
  - Avoid recursion/solving when problem already solved
Fibonacci: Don’t do this recursively

```java
public long recFib(int n) {
    // precondition: 0 <= n
    // postcondition: returns the n-th Fibonacci number
    if (0 == n || 1 == n) {
        return 1;
    } else {
        return recFib(n-1) + recFib(n-2);
    }
}
```

- How many clones/calls to compute F(5)?
- How many calls of F(1)?
- How many total calls?
Shortest path in weighted graph

- We need to modify approach slightly for weighted graph
  - Edges have weights, breadth first by itself doesn’t work
  - What’s shortest path from A to F in graph below?

- Use same idea as breadth first search
  - Don’t add 1 to current distance, add ???
  - Might adjust distances more than once
  - What vertex do we visit next?

- What vertex is next is key
  - Use greedy algorithm: closest
  - Huffman is greedy, ...
Greedy Algorithms

- **A greedy algorithm makes a locally optimal decision that leads to a globally optimal solution**
  - Huffman: choose two nodes with minimal weight, combine
    - Leads to optimal coding, optimal Huffman tree
  - Making change with American coins: choose largest coin possible as many times as possible
    - Change for $0.63, change for $0.32
    - What if we’re out of nickels, change for $0.32?

- **Greedy doesn’t always work, but it does sometimes**
- **Weighted shortest path algorithm is Dijkstra’s algorithm, greedy and uses priority queue**
Shortest Path (Unweighted)

1. Mark all vertices with infinity (*) except starting vertex with 0.
2. Place starting vertex in queue.
3. Repeat until queue is empty:
   1. Remove a vertex from front of queue.
   2. For each adjacent vertex marked with *,
      i. process it,
      ii. mark it with source distance + 1
      iii. place it on the queue.
Shortest Path (Unweighted)

- Mark all vertices with infinity (*)
- Mark starting vertex with 0
- Place starting vertex in queue
- Repeat until queue is empty:
  1. Remove a vertex from front of queue
  2. For each adjacent vertex marked with *,
     i. process it,
     ii. mark it with source distance + 1
     iii. place it on the queue.

How do we get actual “Path”?
Shortest Path (Weighted): Dijkstra

- Unmark all vertices and give infinite weight
- Set weight of starting vertex at 0 and place in priority queue
- Repeat until priority queue is empty:
  1. Remove a vertex from priority queue
     i. Process and mark (weight now permanent)
  2. For each adjacent unmarked vertex
     i. Set weight at lesser of current weight and (source weight + path weight).
        - May involve reducing previous weight setting
     ii. Place in priority queue (if not there already)
Shortest Path (Weighted): Dijkstra

Diagram showing the shortest path algorithm with nodes labeled v0, v1, v2, v3, v4, v5, and v6. The path follows the sequence 0 → 1 → 2 → 3 → 4 → 6 with weights as indicated by the numerical values in the diagram.
Shortest Path (Weighted): Dijkstra

1. Mark all vertices with infinity (*).
2. Mark the starting vertex with 0.
3. Place the starting vertex in the queue.
4. Repeat until the queue is empty:
   a. Remove a vertex from the front of the queue.
   b. For each adjacent vertex marked with *,
      i. Process it,
      ii. Mark it with the source distance + 1,
      iii. Place it on the queue.

How do we get actual "Path"?