Graphs: Structures and Algorithms

- **How do packets of bits/information get routed on the internet**
  - Message divided into packets on client (your) machine
  - Packets sent out using routing tables toward destination
    - Packets may take different routes to destination
    - What happens if packets lost or arrive out-of-order?
  - Routing tables store local information, not global (why?)

- **What about The Oracle of Bacon, Six Degrees of Separation, Erdos Numbers, and Word Ladders?**
  - All can be modeled using graphs
  - What kind of connectivity does each concept model?

- **Graphs are everywhere in the world of algorithms (world?)**
Vocabulary

- **Graphs are collections of vertices and edges**
  - Vertex is sometimes called a *node*
  - An edge connects two *vertices*
    - Direction is sometimes important, other times not so
    - Sometimes edge has a weight/cost associated with it

- A sequence of vertices $v_0, v_1, v_2, \ldots, v_{n-1}$ is a *path* where $v_k$ and $v_{k+1}$ are connected by an edge.
  - If some vertex is repeated, the path is a *cycle*
  - Trees are cycle-free graphs with a root
  - A graph is *connected* if there is a path between any pair of vertices
    - Non-connected graphs have *connected components*
Graph Traversals

- **Connected?**
  - Why?
  - Indegrees? Outdegrees?

- **Starting at 7 where can we get?**
  - *Depth-first* search, envision each vertex as a room, with doors leading out
    - Go into a room, choose a door, mark the door and go out
    - Don’t go into a room you’ve already been in
    - Backtrack and open the next unopened door
  - Rooms are stacked up, backtracking is really recursion
  - One alternative uses a queue: *breadth-first* search
Pseudo-code for depth-first search

```cpp
void depthfirst(const string& vertex)
// post: depth-first search from vertex complete
{
    if (! alreadySeen(vertex))
    {
        markAsSeen(vertex);
        cout << vertex << endl;
        for (each v adjacent to vertex)
        {
            depthfirst(v);
        }
    }
}
```

- Clones are stacked up, problem? When are all doors out of vertex opened and visited? Can we make use of stack explicit?
Graph implementations

• Typical operations on graph:
  ➤ Add vertex
  ➤ Add edge (parameters?)
  ➤ AdjacentVerts(vertex)
  ➤ AllVerts(..)
  ➤ String->int (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

Adjacency list
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?
Other graph questions/operations

- What vertices are reachable from a given vertex
  - Can depth-first search help here?

- What vertex has the highest in-degree (out-degree)?
  - How can we use a map to answer this question?

- Shortest path between any two vertices
  - Breadth first search is storage expensive
  - Dijkstra’s algorithm will offer an alternative, uses a priority queue too!

- Longest path in a graph
  - No known efficient algorithm
Breadth first search

- In an unweighted graph this finds the shortest path between a start vertex and every vertex
  - Visit every node one away from start
  - Visit every node two away from start
    - This is every node one away from a node one away
  - Visit every node three away from start

- Like depth first search, but use a queue instead of a stack
  - What features of a queue ensure shortest path?
  - Stack can be simulated with recursion, advantages?
  - How many vertices on the stack/queue?
void breadthfirst(const string& vertex)
    // post: breadth-first search from vertex complete
{
    tqueue<string> q;
    q.enqueue(vertex);
    while (q.size() > 0)
    {
        q.dequeue(current);
        for (each v adjacent to current)
        {
            if (distance[v] == INFINITY) // not seen
            {
                distance[v] = distance[current] + 1;
                q.enqueue(v);
            }
        }
    }
}
What about word ladders

- **Find a path from white->house changing one letter**
  - Real world? Computer vs. human?
    - white write writs waits warts parts ports forts forte
    - ... rouse house
  - See ladder.cpp program

- **How is this a graph problem? What are vertices/edges?**
- **What about spell-checking, how is it similar?**
  - Edge from accomodate to accommodate
  - Can also use tries with wild-cards, e.g., acc*date
What about connected components?

- What computers are reachable from this one? What people are reachable from me via acquaintanceship?
  - Start at some vertex, depth-first search (why not breadth)?
    - Mark nodes visited
    - Repeat, starting from an unvisited vertex (until all visited)

- What is minimal size of a component? Maximal size?
  - What is complexity of algorithm in terms of V and E?

- What algorithms does this lead to in graphs?
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
Edsger Dijkstra

- Turing Award, 1972
- Operating systems and concurrency
- Algol-60 programming language
- Goto considered harmful
- Shortest path algorithm
- Structured programming
  “Program testing can show the presence of bugs, but never their absence”
- A Discipline of programming
  “For the absence of a bibliography I offer neither explanation nor apology”
Dijkstra’s Shortest Path Algorithm

• Similar to breadth first search, but uses a priority queue instead of a queue. Code below is for breadth first search

```c
q.dequeue(vertex w)
foreach (vertex v adjacent to w)
    if (distance[v] == INT_MAX) // not visited
    {
        distance[v] = distance[w] + 1;
        q.enqueue(v);
    }
```

• Dijkstra: Find minimal unvisited node, recalculate costs through node

```c
q.deletemin(vertex w)
foreach (vertex v adjacent to w)
    if (distance[w] + weight(w,v) < distance[v])
    {
        distance[v] = distance[w] + weight(w,v);
        q.enqueue(vertex(v, distance[v]));
    }
```
Shortest paths, more details

- **Single-source shortest path**
  - Start at some vertex $S$
  - Find shortest path to every reachable vertex from $S$

- **A set of vertices is processed**
  - Initially just $S$ is processed
  - Each pass processes a vertex
  
  *After each pass, shortest path from $S$ to any vertex using just vertices from processed set (except for last vertex) is always known*

- Next processed vertex is closest to $S$ still needing processing
Dijkstra’s algorithm works (greedily)

- Choosing minimal unseen vertex to process leads to shortest paths

```plaintext
q.deleteMin(vertex w)
foreach (vertex v adjacent to w)
    if (distance[w] + weight(w,v) < distance[v])
        { distance[v] = distance[w] + weight(w,v);
          q.enqueue(vertex(v, distance[v]));
        }
```

- We always know shortest path through processed vertices
  - When we choose $w$, there can’t be a shorter path to $w$ than distance[w] – it would go through processed $u$, then we would have chosen $u$ instead of $w$
Topological sort

- Given a directed acyclic graph (DAG)
  - Order vertices so that any if there is an edge (v,w), then v appears before w in the order

- Prerequisites for a major, take CPS 100 before CPS 130
  - Edge(cps100,cps130)
  - Topological sort gives an ordering for taking courses

- Where does ordering start?
  - First vertex has no prereqs
  - “remove” this vertex, continue
  - Depends on in-degree