MCS 253p

Graphs

Graphs

- G = (V,E)
- V: vertices (or nodes). Notation u,v in V
- E: edges. Each connects a pair of vertices

  E : V x V; (u,v) in E
- Link

Example Applications of Graphs

- Airport system
  - Airports are vertices, flights are edges
  - Best path may mean shortest fly time or fewest edges
- Traffic flow
  - Intersection is vertex, roads are edges
  - Best path could be shortest distance, or fastest time
- Computer Network

Variations

- Undirected
  - e.g., electrical wire in a building
- Directed
  - streets (one way or two way), flights
- Weighted edges (notation w(u, v))
  - e.g., time/cost to travel a given flight leg
- Can also have weighted vertices
  - e.g., the time spent waiting at an airport
  - or the cost to stay in a hotel

Undirected Graphs

- degree of a vertex: # of edges connected to the vertex
- complete graph:
  - graph where all pairs of vertices are connected
- bipartite graph:
  - undirected graph whose $V = V_1 \times V_2$ and $E = V_1 \times V_2$
- multigraph: can have multiple edges between the same pair of vertices (including self edges)
- hypergraph: can have edges connect more than two vertices

Directed Graphs

- in-degree: # incoming edges
- out-degree: # outgoing edges
- path: $<v_0, v_1, \ldots, v_k>$ where $<v_i, v_{i+1}>$ in E
- simple path: path where all v_i are distinct
- cycle: a non-trivial simple path plus $<v_k,v_0>$ to close the path
- DAG: directed acyclic graph (contains no cycles)
- strongly connected: digraph whose vertices are all reachable from each other

Representations

- Adjacency list:
  - For each vertex, list its neighbors on outgoing edges
  - Good for sparse graphs
  - Can store weight in linked list node
- Adjacency matrix:
  - Bit matrix to represent presence of edge
  - Good for dense graphs
  - Can store weight in matrix
- Trade-offs
  - Adjacency lists - space efficient, easy to grow
  - Adjacency matrix - fast access, but $O(V^2)$ space

Breadth First Search

- “Distance” refers to number of vertices in path
- Visit vertices in increasing order of distance from starting point
- Use a queue to store vertices “to visit”
- Not necessarily unique
- Explore breadth and then depth
- May find the shortest distance to some node
- Breadth First Search

BFS(G, s)

```
for each vertex u in V[G] - [s] do
    color[u] = WHITE (White means undiscovered)
    distance[u] = Infinity (distance from s)
    previous[u] = NIL (previous vertex)
color[s] = GRAY, distance[s] = 0, previous[s] = NIL, Q = {}
Q.ENQUEUE(s)
while ! Q.IS_EMPTY() do
        u = Q.DEQUEUE()
        for each v in Adjacent[u] do
            if color[v] == WHITE then
                color[v] = GRAY (Gray means discovered, but not expanded)
                distance[v] = distance[u] + 1
                previous[v] = u
                Q.ENQUEUE(v)
        color[u] = BLACK (Black means expanded)
```

Depth first search

- Explores depth then neighbors
- Depth isn't necessarily the same as distance in BFS
- discover vertices before we visit
- Use a stack to store nodes "to visit"
- DFS order may not be unique!
- Variant: Iterative Deepening DFS
- Depth First Search

DFS(G)

```
for each u in V[G] do
   color[u] = WHITE
previous[u] = NIL
time = 0
for each u in V[G] do
   if color[u] == WHITE then
      DFS_VISIT(u)
```

Useful Graph Operations

- We will use these in some graph algorithms
- Time stamp each vertex with two time stamps
  - discover time discovered[u]
  - finish time finished[u]
- Can be used to detect cycles
- Assigned with DFS_VISIT (next slide)

DFS_VISIT(u)

```
color[u] = GRAY (discovered u)
discovered[u] = ++time
for each v in Adjacent[u] do
   if color[v] == WHITE then
      previous[v] = u
      DFS_VISIT(v)
color[u] = BLACK (done exploring from u)
finished[u] = ++time
```

Topological Sorting

- Directed Acyclic Graph
- Sorted order in which all of a node's predecessors appear before the node
- Useful if computing some property on graph that requires computations of predecessors
- E.g., useful for scheduling tasks that depend on other tasks

eat food, cook food, mix food, wash food, buy food

Topological Sort Algorithm

1 call DFS to compute finishing times for each vertex v

2 as each vertex is finished, insert it onto the front of a list

3 return the list of vertices

Minimum Spanning Trees (MST)

- Spanning tree
  - Spanning - connects each vertex
  - Tree - acyclic
  - Start with connected, undirected, edge-weighted graph G
  - Sub-graph with a subset of edges that connect all vertices
  - $|E| = |V| - 1$
- Minimum spanning tree
  - Sum total weight of edges is minimized
  - edge weights need not be distinct, MST need not be unique
- Example: wiring a house with minimal cable

MST Algorithms

- Two common algorithms:
  - Prim's
  - Kruskal's
- Both are "Greedy algorithms"
  - take the best choice at each opportunity
- Difference is in how next edge is selected

Prim's Algorithm

- Start with single vertex, called the root
- Select the minimum edge connecting u, v such that u is in the MST and v is not
- Grow MST by adding one vertex and one edge at a time
- $O(|V|^2)$ without heaps for priority queue
- $O(|E| \log |V|)$ with heaps
- Link

Prim's Algorithm

```
def PRIM(G):
    for each u in V[G]:
        key[u] = INFINITY;   prev[u] = nil
    Q.ENQUEUE(G.V);  F = {G.V}
    while ! Q.ISEMPTY():
        u = Q.EXTRACT-MIN()
        foreach v in Adjacent[u]:
            if v in F   &&  G.w(u, v) < key[v]:
                F = F - u
                pr[v] = u
                key[v] = G.w(u,v)
```

Kruskal’s Algorithm

- Start with single-node trees
- Identify cheapest edge
- If edge links different trees && doesn’t cause a cycle
  - add the edge
- Adding an edge merges two trees into one
- Minimum Spanning Tree Animation
- O(|V| log |E|) - much better in practice
- Link

Kruskal's Algorithm in pseudo code

```
def KRUSKAL(G):
    A = ∅
    foreach v ∈ G.V:
        MAKE-SET(v)
    foreach (u, v) in G.E ordered by weight(u, v), increasing:
        if FIND-SET(u) ≠ FIND-SET(v):
            A = A ∪ {(u, v)}
            UNION(u, v)
    return A
```

Disjoint Set

- Sets are required for Kruskal’s MST algorithms
- Link

Kruskal's Algorithm in pseudo C++

```
// complexity is O(|E| lg |E|)  or O(|E| lg |V|) if using binary heap for PriorityQueue

Vector<Edges> Kruskals(const Graph & g)
{
   for ( auto v : g.vertices() )  {
      vertex.setID = Set::makeSet();
   }
   int N = g.vertices().size();
   PriorityQueue Q;
   for ( auto e : g.edges() )
      Q.insert(e); // Sort Key is edge weight
   Vector<Edges> T;
   while ( T.size() < N-1 ) {
      Edge e = Q.extractMin();
      if ( e.u.setID != e.v.setID ) { // i.e., not in the same set
         T.push_back(e);  // add edge e to solution
         Set::union( e.u, e.v );  // join two sub trees into one tree
      }
   }
   return T;
}
```

Single Source Shortest Path

- Given
  - A graph, G = (V,E)
  - A single starting vertex, s
- Find lowest cost path to all other vertices
- For un-weighted graph, use BFS
- For weighted graph, we have choices
  - Dijkstra’s Algorithm - positive weights
  - Bellman-Ford Algorithm - negative too
  - A* is heuristic, but must know max
    - http://en.wikipedia.org/wiki/A*#Pseudocode

Applications

- Maze games
- Street traffic routing
- Plane trip planning
- Network packet routing
- Robot walking through obstacles
- Dijkstra Animation

Naïve Solution

- "Brute Force" or "Exhaustive Search"
- Enumerate all routes from A to B
- Add up the distances of each route
- Select the shortest
- There could be millions of possibilities
  - cycles make it infinite

Dijkstra’s Algorithm

- A greedy algorithm published in 1959
- Keep distance estimates to neighbors of S
- Add neighbor with shortest distance to S and update other neighbors
- Doesn’t work with negative edges
- O((|E|+|V|) log |V|)
    - with heap for PriorityQ
- Dijkstra Algorithm Presentation
- Wikipedia entry
    - http://en.wikipedia.org/wiki/Dijkstra's_algorithm
- Link STL Link

Dijkstras Algorithm in pseudo C++

```
// complexity is O((|E|+|V|) lg |V|) with binary heap for PriorityQueue
typedef int Vertex;
struct Edge {Vertex dest; int weight;};

void dijkstras(Graph g, Vertex s, int dist[], int prev[]) {
    PriorityQueue<Vertex> Q;
    for (  Vertex v : g.getVertices() )  { // initialize
        dist[v] = INFINITY;            // INT_MAX
        prev[v] = UNDEFINED;    // -1
        Q.insert(v);
    }
    dist[s] = 0; // distance from start to start is zero, first one extracted below
    while ( ! Q.isEmpty() ) {
        Vertex u = Q.extractMin(); // take out vertex with min dist
        for (  Edge edge : g.outgoingEdges(u)  )  {  // from adjacency list
            Vertex v = edge.dest;
            if ( dist[v] > dist[u] + edge.weight ) { // relax, use edge u to v
                dist[v] = dist[u] + edge.weight;
                prev[v] = u;
                Q.decreaseKey(v); // dist[v] has new value, must move up in Q
            }
        }
    }
}
```

Sample Graph Problems

1.