Week 08: Graph Algorithms 2

Weighted Graphs

Graphs so far have considered
• edge = an association between two vertices/nodes
  may be a precedence in the association (directed)

Some applications require us to consider
• a cost or weight of an association
  modelled by assigning values to edges (e.g. positive reals)

Weights can be used in both directed and undirected graphs.

Example: major airline flight routes in Australia

Weights lead to minimisation-type questions, e.g.
1. Cheapest way to connect all vertices?
   • a.k.a. *minimum spanning tree* problem
   • assumes: edges are weighted and undirected

2. Cheapest way to get from A to B?
   • a.k.a. *shortest path* problem
   • assumes: edge weights positive, directed or undirected

Exercise #1: Implementing a Route Finder

If we represent a street map as a graph

• what are the vertices?
• what are the edges?
• are edges directional?
• what are the weights?
• are the weights fixed?

What kind of algorithm would ...

• help us find the "quickest" way to get from A to B?

Weighted Graph Representation

Weights can easily be added to:
• adjacency matrix representation
  (0/1 → int or float)
• adjacency lists representation
  (add int/float to list node)

An alternative representation useful in this context:
• edge list representation
  (list of (s,t,w) triples)

All representations work whether edges are directed or not.
Note: if undirected, each edge appears twice with same weight

### Weighted Graph Representation

Edge array / edge list representation with weights:

Note: not very efficient for use in processing algorithms, but does give a possible representation for min spanning trees or shortest paths

### Weighted Graph Representation

Sample adjacency matrix implementation in C requires minimal changes to previous Graph ADT:

**WGraph.h**

// edges are pairs of vertices (end-points) plus positive weight
typedef struct Edge {
    Vertex v;
    Vertex w;
    int weight;
} Edge;

// returns weight, or 0 if vertices not adjacent
int adjacent(Graph g, Vertex v, Vertex w);

**WGraph.c**

typedef struct GraphRep {
    int **edges; // adjacency matrix storing positive weights
    int nV; // #vertices
    int nE; // #edges
} GraphRep;

void insertEdge(Graph g, Edge e) {
    assert(g != NULL && validV(g,e.v) && validV(g,e.w));
    if (g->edges[e.v][e.w] == 0) { // edge e not in graph
        g->edges[e.v][e.w] = e.weight;
        g->edges[e.w][e.v] = e.weight;
        g->nE++;
    }
}

int adjacent(Graph g, Vertex v, Vertex w) {
    assert(g != NULL && validV(g,v) && validV(g,w));
    return g->edges[v][w];
}

### Minimum Spanning Trees

#### Minimum Spanning Trees

Reminder: Spanning tree $ST$ of graph $G(V,E)$

- spanning = all vertices, tree = no cycles
- $ST$ is a subgraph of $G$ ($G'=(V,E')$ where $E' \subseteq E$)
- $ST$ is connected and acyclic

Minimum spanning tree $MST$ of graph $G$

- $MST$ is a spanning tree of $G$
- sum of edge weights is no larger than any other $ST$

Applications: Computer networks, Electrical grids, Transportation networks …

**Problem:** how to (efficiently) find MST for graph $G$?

NB: MST may not be unique  (e.g. all edges have same weight => every ST is MST)

#### Minimum Spanning Trees

Brute force solution:

```
findMST(G):
   Input graph G
   Output a minimum spanning tree of G
   bestCost=∞
   for all spanning trees t of G do
      if cost(t)<bestCost then
         bestTree=t
         bestCost=cost(t)
   end if
```

Note: if undirected, each edge appears twice with same weight
Example of generate-and-test algorithm.

Not useful because #spanning trees is potentially large (e.g. $n^{n-2}$ for a complete graph with $n$ vertices)

---

### Minimum Spanning Trees

Simplifying assumption:
- edges in $G$ are not directed (MST for digraphs is harder)

---

### Kruskal's Algorithm

One approach to computing MST for graph $G$ with $V$ nodes:

1. start with empty MST
2. consider edges in increasing weight order
   - add edge if it does not form a cycle in MST
3. repeat until $V-1$ edges are added

Critical operations:
- iterating over edges in weight order
- checking for cycles in a graph

---

**Exercise #2: Kruskal's Algorithm**

Show how Kruskal's algorithm produces an MST on:

---

**Pseudocode:**

KruskalMST($G$):

1. **Input** graph $G$ with $n$ nodes
2. **Output** a minimum spanning tree of $G$
3. $MST$ = empty graph
sort edges(G) by weight
for each e \in \text{sortedEdgeList} do
  MST = MST \cup \{e\}
  if MST has a cycle then
    MST = MST \setminus \{e\}
  end if
  if MST has n-1 edges then
    return MST
  end if
end for

... Kruskal's Algorithm

Rough time complexity analysis ...

- sorting edge list is \(O(E \cdot \log E)\)
- at least \(V\) iterations over sorted edges
  - getting next lowest cost edge is \(O(1)\)
  - checking whether adding it forms a cycle: cost = ??

Possibilities for cycle checking:

- use DFS ... too expensive?
- could use Union-Find data structure (see Sedgewick Ch.1)

Prim's Algorithm

Another approach to computing MST for graph \(G = (V,E)\):

1. start from any vertex \(s\) and empty MST
2. choose edge not already in MST to add to MST
   - must be incident on a vertex already connected to \(s\) in MST
   - must have minimal weight of all such edges
3. repeat until MST covers all vertices

Critical operations:

- checking for vertex being connected in a graph
- finding min weight edge in a set of edges

... Prim's Algorithm

Execution trace of Prim's algorithm (starting at \(s=0\)):

Exercise #3: Prim's Algorithm

Show how Prim's algorithm produces an MST on:

Start from vertex 0

After 1st iteration:

After 2nd iteration:

After 3rd iteration:

After 4th iteration:
After 8\textsuperscript{th} iteration (all vertices covered):

```
3 4 5
1
2 6 9
```

... Prim's Algorithm

Pseudocode:

\[\text{PrimMST}(G):\]
- **Input** graph \(G\) with \(n\) nodes
- **Output** a minimum spanning tree of \(G\)

\(\begin{align*}
\text{MST} &= \text{empty graph} \\
\text{usedV} &= \{0\} \\
\text{unusedE} &= \text{edges}(g)
\end{align*}\)

\[\text{while} \ |\text{usedV}| < n \text{ \ do} \]
- \(\text{find } e = (s, t, w) \in \text{unusedE} \text{ such that }\)
  - \(s \in \text{usedV} \land t \in \text{usedV} \land w \text{ is min weight of all such edges}\)
- \(\text{MST} = \text{MST} \cup \{e\}\)
- \(\text{usedV} = \text{usedV} \cup \{t\}\)
- \(\text{unusedE} = \text{unusedE} \setminus \{e\}\)

\[\text{end while}\]

\text{return } \text{MST}

Critical operation: finding best edge

---

... Prim's Algorithm

Rough time complexity analysis ...

- \(V\) iterations of outer loop
  - in each iteration ...
    - find min edge with set of edges is \(O(E) \Rightarrow O(V \cdot E)\) overall
    - find min edge with priority queue is \(O(\log E) \Rightarrow O(V \cdot \log E)\) overall

Note:

- Using a *priority queue* gives a variation of DFS (stack) and BFS (queue) graph traversal

---

... Sidetrack: Priority Queues

Some applications of queues require

- items processed in order of "key"

---

**Priority Queues** provide this via:

- **join**: insert item into PQ (replacing enqueue)
- **leave**: remove item with largest key (replacing dequeue)

---

Comparison of different Priority Queue representations:

<table>
<thead>
<tr>
<th></th>
<th>sorted array</th>
<th>unsorted array</th>
<th>sorted list</th>
<th>unsorted list</th>
</tr>
</thead>
<tbody>
<tr>
<td>space usage</td>
<td>MaxN</td>
<td>MaxN</td>
<td>(O(N))</td>
<td>(O(N))</td>
</tr>
<tr>
<td>join</td>
<td>(O(N))</td>
<td>(O(1))</td>
<td>(O(N))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>leave</td>
<td>(O(N))</td>
<td>(O(N))</td>
<td>(O(1))</td>
<td>(O(N))</td>
</tr>
<tr>
<td>is empty?</td>
<td>(O(1))</td>
<td>(O(1))</td>
<td>(O(1))</td>
<td>(O(1))</td>
</tr>
</tbody>
</table>

for a PQ containing \(N\) items

---

**Other MST Algorithms**

Boruvka's algorithm ... complexity \(O(E \cdot \log V)\)

- the oldest MST algorithm
- start with \(V\) separate components
- join components using min cost links
- continue until only a single component

Karger, Klein, and Tarjan ... complexity \(O(E)\)

- based on Boruvka, but non-deterministic
- randomly selects subset of edges to consider
- for the keen, here's the paper describing the algorithm

---

**Shortest Path**

**Path** = sequence of edges in graph \(G\)

\[p = (v_0, v_1, \ldots, v_m)\]

\(\text{cost}(\text{path}) = \text{sum of edge weights along path}\)

**Shortest path** between vertices \(s\) and \(t\)

- a simple path \(p(s, t)\) where \(s = \text{first}(p), t = \text{last}(p)\)
• no other simple path \(q(s,t)\) has \(\text{cost}(q) < \text{cost}(p)\)

Assumptions: weighted digraph, no negative weights.

Finding shortest path between two given nodes known as \textit{source-target} SPP problem

Variations: \textit{single-source, all-pairs}

Applications: robot navigation, routing in data networks, …

---

**Single-source Shortest Path (SSSP)**

Given: weighted digraph \(G\), source vertex \(s\)

Result: shortest paths from \(s\) to all other vertices

- \(\text{dist}[]\) \(V\)-indexed array of cost of shortest path from \(s\)
- \(\text{pred}[]\) \(V\)-indexed array of predecessor in shortest path from \(s\)

Example:

---

**Edge Relaxation**

Assume: \(\text{dist}[]\) and \(\text{pred}[]\) as above  
(but containing data for shortest paths \textit{discovered so far})

\(\text{dist}[v]\) is length of shortest known path from \(s\) to \(v\)

\(\text{dist}[w]\) is length of shortest known path from \(s\) to \(w\)

Relaxation updates data for \(w\) if we find a shorter path from \(s\) to \(w\):

Relaxation along edge \(e=(v,w,weight)\):

- if \(\text{dist}[v]+\text{weight} < \text{dist}[w]\) then update \(\text{dist}[w]:=\text{dist}[v]+\text{weight} \text{ and } \text{pred}[w]:=w\)

---

**Dijkstra’s Algorithm**

One approach to solving single-source shortest path …

Data: \(G, s, \text{dist}[], \text{pred}[]\) and

- \(\text{vSet}\): set of vertices whose shortest path from \(s\) is known

Algorithm:

\[
\begin{align*}
\text{dijkstraSSSP}(G, \text{source}): \quad & \text{Input graph } G, \text{ source node} \\
& \text{initialise } \text{dist}[] \text{ to all } \infty, \text{ except } \text{dist}[\text{source}]=0 \\
& \text{initialise } \text{pred}[] \text{ to all } -1 \\
& \text{vSet}=\text{all vertices of } G \\
& \text{while } \text{vSet} \neq \emptyset \text{ do} \\
& \quad \text{find } s \in \text{vSet with minimum } \text{dist}[s] \\
& \quad \text{for each } (s,t,w) \in \text{edges}(G) \text{ do} \\
& \quad \quad \text{relax along } (s,t,w) \\
& \quad \text{end for} \\
& \quad \text{vSet}=\text{vSet}\{s\} \\
& \text{end while}
\end{align*}
\]

---

**Exercise #4: Dijkstra’s Algorithm**

Show how Dijkstra’s algorithm runs on (source node = 0):
**Dijkstra's Algorithm**

Why Dijkstra's algorithm is correct:

*Hypothesis.*

(a) For visited $s$ ... $dist[s]$ is shortest distance from source

(b) For unvisited $t$ ... $dist[t]$ is shortest distance from source via visited nodes

*Proof.*

Base case: no visited nodes, $dist[source]=0$, $dist[s]=\infty$ for all other nodes

Induction step:

1. If $s$ is an unvisited node with minimum $dist[s]$, then $dist[s]$ is shortest distance from source to $s$:
   - if $\exists$ shorter path via only visited nodes, then $dist[s]$ would have been updated when processing the predecessor of $s$ on this path
   - if $\exists$ shorter path via an unvisited node $u$, then $dist[u]<dist[s]$, which is impossible if $s$ has min distance of all unvisited nodes

2. This implies that (a) holds for $s$ after processing $s$

3. (b) still holds for all unvisited nodes $t$ after processing $s$:
   - if $\exists$ shorter path via $s$ we would have just updated $dist[t]$
   - if $\exists$ shorter path without $s$ we would have found it previously

---

**Time complexity analysis ...**

Each edge needs to be considered once $\Rightarrow O(E)$.

Outer loop has $O(V)$ iterations.

Implementing "**find s\in\text{vSet with minimum dist[s]}"**

1. try all $s\in\text{vSet} \Rightarrow \text{cost} = O(V) \Rightarrow \text{overall cost} = O(E + V^2) = O(V^2)$
2. using a PQueue to implement extracting minimum
   - can improve overall cost to $O(E + V\cdot\log V)$ (for best-known implementation)

---

**Summary**

- Weighted graph representations
- Minimum Spanning Tree (MST)
  - Kruskal, Prim
- Single source shortest path problem
  - Dijkstra

- Suggested reading:
  - Sedgewick, Ch.19.3, 20-20.4, 21-21.3

Produced: 11 Sep 2017