CS35X: Competitive Programming

Lecture 8: Priority Queues, Dijkstra's Algorithm

Joshua Brody

Warmup Kattis Problem: vemvinner

Problem debrief: Button Bashing

Graphs revisited

- A graph **G = (V,E)** is a set of *vertices* **V** along with a set of *edges* **E**.
- Edges can be *directed* or *undirected*
  - *Directed* graph: edges ***u—>v*** are asymmetric
  - *Undirected* graph: edges ***u—v*** are symmetric
- Edges can be *weighted* or *unweighted*

- Which type of edges you have depends on application.

Motivation:
Emergency Room Triage

Priority Queue ADT

- Maintain collection of (***priority***, ***value***) pairs.
- Support the following operations:
  - Initialize an empty priority queue.
  - **Insert** a new (***priority***, ***value***) pair.
  - **Get** the highest priority pair.
  - **Remove** the highest priority pair.
  - Check to see if a PQ is **empty**.
- **Note:** “high priority” can be minimum PQ or maximum PQ

Example Syntax

```
#include <queue>
priority_queue<int> pq;
pq.push(5);                    // insert 5
pq.push(3);                    // insert 3
cout << pq.top() << endl;      // 5
pq.pop();                      // remove 5
cout << pq.top() << endl;      // 3
pq.push(6);                    // insert 6
cout << pq.top() << endl;
```

C++ priority_queue details

- Stores just priorities
- To support (***priority***, ***value***) pairs: use **pair** class.
  - `priority_queue<pair<int,string>> myPQ;`
- priority_queue is maximum priority by default.
- Make a minimum PQ by changing the comparison operator:
  - `priority_queue<int, vector<int>, std::greater> minPQ;`
- Sometimes it is useful to create your own comparison operator:
  - ```
    bool operator<(const Edge& rhs) const {
      return weight > rhs.weight;
    }
    ```
    ```

Exercise: Fancy Dog Show winners!
week8/dogshow

Application: Dijkstra's algorithm

- Find shortest path in *weighted* graph from source to all other vertices.
- Idea: maintain dictionary of distance of shortest path to u we've seen
- Needs:
  - `dist`: map of current best distances from source to each vertex
  - `minPQ`: minimum PQ of candidate distances
- At each step, lock in distance from source to one node

```
map<int,int> dist;
priority_queue<pair<int,int>, vector<pair<int,int>>,
                    greater<pair<int,int>>> minPQ;
dist[source]=0;
minPQ.push(pair(0,source))
while(!minPQ.empty()) {
  int u = minPQ.top().second;
  int d = minPQ.top().first;
  minPQ.pop();
  if(d > dist[u]) continue;          // distance more than current best
  for(int j=0; j<g[u].size(); j++) {              // for each neighbor of u
    Edge e = g[u][j];
    v = e.dest;
    newcost = dist[u]+e.weight;
    if(!dist.count(v) || newcost < dist[v]) {
      dist[v] = newcost;
      minPQ.push(pair(newcost,v));
    }
  }
}
```

Dijkstra's Algorithm implementation details

- Two ways to handle Infinity:
  - `const int INF = 1e9;  // define INF to be a ridiculously large number`
  - `vertex not in dictionary == distance is infinite`
- Defining an Edge class can be useful, e.g.:

```
class Edge{
  Public:
    int src, dest, weight;
  Edge(int src, int dest, int weight) {
    this->src = src;
    this->dest = dest;
    this->weight = weight;
  }
};
```

- Create your own comparison operator:
  -
```
bool operator<(const Edge& rhs) const {
    return weight > rhs.weight;
 }
```

Kattis Problem: bumped