ACM Lab – HS 2015

Prof. Dr. Angelika Steger

October 15, 2015

Weighted Graphs

- If each edge is equipped with a weight, how do we store the graph?

- Adjacency matrix: store weight in matrix

```
vector<vector<int> > graph(n,vector<int>(n,INF));
for(int i = 0; i < m; ++i){
        int a,b,c; cin >> a >> b >> c;
        graph[a][b] = c;
        graph[b][a] = c;
    }
```

- Adjacency list: store weight with neighbor

```
vector<vector<pair<int,int> > > graph(n);
for(int i = 0; i < m; ++i){
        int a,b,c; cin >> a >> b >> c;
        graph[a].push_back(make_pair(b,c));
        graph[b].push_back(make_pair(a,c));
    }
```

Weighted Graphs

- If each edge is equipped with a weight, how do we store the graph?

- Adjacency matrix: store weight in matrix

```
vector<vector<int> > graph(n,vector<int>(n,INF));
for(int i = 0; i < m; ++i){
        int a,b,c; cin >> a >> b >> c;
        graph[a][b] = c;
        graph[b][a] = c;
    }
```

- Adjacency list: store weight with neighbor

```
vector<vector<pair<int,int> > > graph(n);
for(int i = 0; i < m; ++i){
        int a,b,c; cin >> a >> b >> c;
        graph[a].push_back(make_pair(b,c));
        graph[b].push_back(make_pair(a,c));
    }
```

Weighted Graphs

- If each edge is equipped with a weight, how do we store the graph?

- Adjacency matrix: store weight in matrix

```
vector<vector<int> > graph(n,vector<int>(n,INF));
for(int i = 0; i < m; ++i){
        int a,b,c; cin >> a >> b >> c;
        graph[a][b] = c;
        graph[b][a] = c;
    }
```

- Adjacency list: store weight with neighbor

```
vector<vector<pair<int,int> > > graph(n);
for(int i = 0; i < m; ++i){
        int a,b,c; cin >> a >> b >> c;
        graph[a].push_back(make_pair(b,c));
        graph[b].push_back(make_pair(a,c));
    }
```

Single Source Shortest Paths (SSSP)

- Task: given a weighted graph G and a source s, compute the shortest path from s to t for all $t \in V(G)$.
- If all weights are 1: BFS
- If weights are non-negative: Dijkstra's algorithm
- (Otherwise Bellman-Ford)

Dijkstra's Algorithm

Input: (G, w, s)
begin

$d[v] \longleftarrow \infty \qquad \forall v \in V(G)$
$d[s] \longleftarrow 0$
$S \longleftarrow \emptyset$
$Q \longleftarrow V(G)$
while $Q \neq \emptyset$ **do**
  $u \longleftarrow min(Q)$
  $S \longleftarrow S \cup \{u\}$
  foreach $v \in Adj[u]$ **do**
    if *d[v] > d[u] + w[(u,v)]* **then**
      $d[v] \longleftarrow d[u] + w[(u, v)]$
      $decreaseKey(Q, v)$

Dijkstra's Algorithm

Input: (G, w, s)
begin
- $d[v] \longleftarrow \infty \qquad \forall v \in V(G)$
- $d[s] \longleftarrow 0$
- $S \longleftarrow \emptyset$
- $Q \longleftarrow V(G)$
- **while** $Q \neq \emptyset$ **do**
  - $u \longleftarrow min(Q)$
  - $S \longleftarrow S \cup \{u\}$
  - **foreach** $v \in Adj[u]$ **do**
    - **if** $d[v] > d[u] + w[(u,v)]$ **then**
      - $d[v] \longleftarrow d[u] + w[(u,v)]$
      - $decreaseKey(Q, v)$

Sorting/Priority Queues of Structs

- How can we sort vectors of our own structs or have priority queues of our own structs?

- Define struct with “<” operator

```
struct s{
    int a;
    int b;

    friend bool operator<(const s &x, const s &y) {
      return x.a < y.a || (x.a == y.a && x.b < y.b)
    }
};
```

- The structs are sorted lexicographically increasing

```
vector<s> v; sort(v.begin(),v.end());
```

- The priority queue returns the lexicographical maximum

```
priority_queue<s>;
```

Sorting/Priority Queues of Structs

- How can we sort vectors of our own structs or have priority queues of our own structs?

- Define struct with "<" operator

```
struct s{
    int a;
    int b;

    friend bool operator<(const s &x, const s &y) {
      return x.a < y.a || (x.a == y.a && x.b < y.b)
    }
};
```

- The structs are sorted lexicographically increasing

```
vector<s> v; sort(v.begin(),v.end());
```

- The priority queue returns the lexicographical maximum

```
priority_queue<s>;
```

Sorting/Priority Queues of Structs

- How can we sort vectors of our own structs or have priority queues of our own structs?

- Define struct with “<” operator

```
struct s{
    int a;
    int b;

    friend bool operator<(const s &x, const s &y) {
      return x.a < y.a || (x.a == y.a && x.b < y.b)
    }
};
```

- The structs are sorted lexicographically increasing

```
vector<s> v; sort(v.begin(),v.end());
```

- The priority queue returns the lexicographical maximum

```
priority_queue<s>;
```

Sorting/Priority Queues of Structs

- How can we sort vectors of our own structs or have priority queues of our own structs?

- Define struct with "<" operator

```
struct s{
    int a;
    int b;

    friend bool operator<(const s &x, const s &y) {
      return x.a < y.a || (x.a == y.a && x.b < y.b)
    }
};
```

- The structs are sorted lexicographically increasing

```
vector<s> v; sort(v.begin(),v.end());
```

- The priority queue returns the lexicographical maximum

```
priority_queue<s>;
```

Dijkstra's Algorithm with STL

- Unfortunately, the STL priority queue does not have a decreaseKey operation
- We can implement a "Lazy Deletion" variant of Dijkstra
- In this version, there can be duplicates in the priority queue

Minimum Spanning Tree (MST)

- Task: given a weighted graph G, compute a spanning tree such that the sum of edge weights of the spanning tree is minimal
- Prim's algorithm (similar to Dijkstra's algorithm)
- Kruskal's algorithm

Minimum Spanning Tree (MST)

- Task: given a weighted graph G, compute a spanning tree such that the sum of edge weights of the spanning tree is minimal
- Prim's algorithm (similar to Dijkstra's algorithm)
- Kruskal's algorithm

Minimum Spanning Tree (MST)

- Task: given a weighted graph G, compute a spanning tree such that the sum of edge weights of the spanning tree is minimal
- Prim's algorithm (similar to Dijkstra's algorithm)
- Kruskal's algorithm

Minimum Spanning Tree (MST)

- Task: given a weighted graph G, compute a spanning tree such that the sum of edge weights of the spanning tree is minimal
- Prim's algorithm (similar to Dijkstra's algorithm)
- Kruskal's algorithm

Kruskal's Algorithm

Input: (G, w)
begin
$T \longleftarrow \emptyset$
$Q \longleftarrow E(G)$
while $Q \neq \emptyset$ **do**
$\{u, v\} \longleftarrow min(Q)$
if $\{u, v\}$ *does not close a cycle in* T **then**
$T \longleftarrow E \cup \{\{u, v\}\}$

Kruskal's Algorithm

Input: (G, w)
begin
$T \longleftarrow \emptyset$
$Q \longleftarrow E(G)$
while $Q \neq \emptyset$ **do**
$\{u, v\} \longleftarrow min(Q)$
if $\{u, v\}$ *does not close a cycle in* T **then**
$T \longleftarrow E \cup \{\{u, v\}\}$

Union Find Disjoint Sets (UFDS)

- Data structure to represent a partition $\{S_1, \ldots, S_i\}$ of $\{1, \ldots, n\}$
- Each subset S_j has some representative $r \in S_j$
- $find(x)$: returns the representative of the set x is contained in
- $union(x, y)$: merges the sets x and y are contained in

Union Find Disjoint Sets (UFDS)

- Data structure to represent a partition $\{S_1, \ldots, S_i\}$ of $\{1, \ldots, n\}$
- Each subset S_j has some representative $r \in S_j$
- $\mathit{find}(x)$: returns the representative of the set x is contained in
- $\mathit{union}(x, y)$: merges the sets x and y are contained in

Union Find Disjoint Sets (UFDS)

- Data structure to represent a partition $\{S_1, \ldots, S_i\}$ of $\{1, \ldots, n\}$
- Each subset S_j has some representative $r \in S_j$
- $find(x)$: returns the representative of the set x is contained in
- $union(x, y)$: merges the sets x and y are contained in

Union Find Disjoint Sets (UFDS)

- Data structure to represent a partition $\{S_1, \ldots, S_i\}$ of $\{1, \ldots, n\}$
- Each subset S_j has some representative $r \in S_j$
- $\mathit{find}(x)$: returns the representative of the set x is contained in
- $\mathit{union}(x, y)$: merges the sets x and y are contained in

Kruskal's Algorithm Revisited

Input: (G, w)
begin

- $T \longleftarrow \emptyset$
- $Q \longleftarrow E(G)$
- **while** $Q \neq \emptyset$ **do**
  - $\{u, v\} \longleftarrow min(Q)$
  - **if** $\{u, v\}$ *does not close a cycle in* T **then**
    - $T \longleftarrow E \cup \{\{u, v\}\}$

- The Union Find Disjoint Set data structure will keep track of the components of T
- If u and v are in the same component (i.e., $find(u) == find(v)$), then adding the edge $\{u, v\}$ would close a cycle