Flipped Lecture Concepts

Week 8
Ananda Guna
11.06.14

Topics this week

- directed graphs (digraphs)
  - API
  - Search
  - Topological sort
  - Strong components
- Minimum spanning trees (MST's)
  - The cut
  - Kruskalls
  - Prim's

API

```
public class Digraph
                  Digraph(int V)
                  Digraph(In in)
            void  addEdge(int v, int w)
Iterable<Integer> adj(int v)
             int  V()
             int  E()
         Digraph  reverse()
          String  toString()
```

Complexity of operations

representation	space	insert edge from v to w	edge from v to w?	iterate over vertices adjacent from v?
list of edges	E	1	E	E
adjacency matrix	V^2	$1^\dagger$	1	V
adjacency lists	$E + V$	1	*outdegree*(v)	*outdegree*(v)

Digraph search

DFS (to visit a vertex v)

Mark vertex v as visited.
Recursively visit all unmarked vertices w adjacent from v.

- Reachability.
- Path finding.
- Topological sort.
- Directed cycle detection.

BFS (from source vertex s)

Put s onto a FIFO queue, and mark s as visited.
Repeat until the queue is empty:
- remove the least recently added vertex v
- for each unmarked vertex adjacent from v: add to queue and mark as visited.

Proposition. BFS computes shortest paths (fewest number of edges) from s to all other vertices in a digraph in time proportional to $E + V$.

Topological sort

- Given a DAG, arrange vertices so that, edges only go forward
- Algorithm
  - Use the DFS to find the postorder of vertices
  - Reverse postorder provides a topological sort of vertices
- Intuition
  - First node in a postorder (last in reverse postorder) has outdegree 0
  - Second-to-last vertex in postorder can only point to last vertex

Directed Cycle Detection

Proposition. A digraph has a topological order iff no directed cycle.
Pf.
- If directed cycle, topological order impossible.
- If no directed cycle, DFS-based algorithm finds a topological order.

orderings

Orderings.
- Preorder: order in which `dfs()` is called.
- Postorder: order in which `dfs()` returns.
- Reverse postorder: reverse order in which `dfs()` returns.

Strong components

Def. Vertices v and w are strongly connected if there is both a directed path from v to w and a directed path from w to v.

Kosaraju-Sharir algorithm: intuition

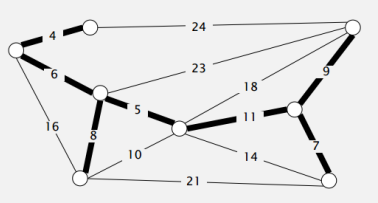
Reverse graph. Strong components in G are same as in G^R.

Kernel DAG. Contract each strong component into a single vertex.

Idea. (how to compute?)
- Compute topological order (reverse postorder) in kernel DAG.
- Run DFS, considering vertices in reverse topological order.

Proposition. Kosaraju-Sharir algorithm computes the strong components of a digraph in time proportional to $E + V$.

Minimum Spanning Tree



Proposition: A connected graph with distinct edge weights has a unique MST

Cut property

Def. A cut in a graph is a partition of its vertices into two (nonempty) sets.
Def. A crossing edge connects a vertex in one set with a vertex in the other.

Cut property. Given any cut, the crossing edge of min weight is in the MST.

Greedy MST algorithm

- Start with all edges colored gray.
- Find cut with no black crossing edges; color its min-weight edge black.
- Repeat until $V - 1$ edges are colored black.

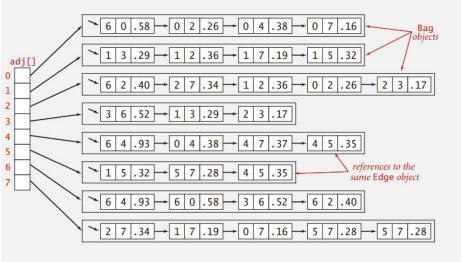
Greedy MST Algorithms

Efficient implementations. Choose cut? Find min-weight edge?
Ex 1. Kruskal's algorithm. [stay tuned]
Ex 2. Prim's algorithm. [stay tuned]
Ex 3. Borüvka's algorithm.

Weighted Edge API

```
public class Edge implements Comparable<Edge>
              Edge(int v, int w, double weight)
        int  either()
        int  other(int v)
        int  compareTo(Edge that)
     double  weight()
     String  toString()
```

Adjacency List



MST API

```
public class MST

                  MST(EdgeWeightedGraph G)      constructor
Iterable<Edge>    edges()                       edges in MST
        double    weight()                      weight of MST
```

Kruskal's Algorithm

- Order edges by weight or maintain a minPQ
- add the minimum edge first to MST
- Continue to add edges as long as they don't create a cycle.
- When |V|-1 edges are added, MST is created

Implementation

- Maintain a minPQ of edges
  - |E| to build
  - |E| ln|E| in the worst case to deleteMin
- Use a union-find
  - To test if adding an edge creates a cycle
    - log|V| in the worst case (weighted UF)
- **Proposition:** Kruskal's algorithm computes MST in time proportional to E log E (in the worst case).

Kruskal code

```
public KruskalMST(EdgeWeightedGraph G)
{
   MinPQ<Edge> pq = new MinPQ<Edge>(G.edges());

   UF uf = new UF(G.V());
   while (!pq.isEmpty() && mst.size() < G.V()-1)
   {
      Edge e = pq.delMin();
      int v = e.either(), w = e.other(v);
      if (!uf.connected(v, w))
      {
         uf.union(v, w);
         mst.enqueue(e);
      }
   }
}
```

Kruskal Summary

operation	frequency	time per op
build pq	1	E
delete-min	E	$\log E$
union	V	$\log^* V$ †
connected	E	$\log^* V$ †

Prim's Algorithm

- Start with vertex 0 and greedily grow tree T.
- Add to T the min weight edge with exactly one endpoint in T.
- Repeat until V - 1 edges.

Prim's Algorithm (lazy implementation)

Challenge. Find the min weight edge with exactly one endpoint in T.

Lazy solution. Maintain a PQ of edges with (at least) one endpoint in T.

- Key = edge; priority = weight of edge.
- Delete-min to determine next edge $e = v$–w to add to T.
- Disregard if both endpoints v and w are marked (both in T).
- Otherwise, let w be the unmarked vertex (not in T):
  - add to PQ any edge incident to w (assuming other endpoint not in T)
  - add e to T and mark w

Implementation

```
public LazyPrimMST(WeightedGraph G)
{
   pq = new MinPQ<Edge>();
   mst = new Queue<Edge>();
   marked = new boolean[G.V()];
   visit(G, 0);

   while (!pq.isEmpty() && mst.size() < G.V() - 1)
   {
      Edge e = pq.delMin();
      int v = e.either(), w = e.other(v);
      if (marked[v] && marked[w]) continue;
      mst.enqueue(e);
      if (!marked[v]) visit(G, v);
      if (!marked[w]) visit(G, w);
   }
```

```
private void visit(WeightedGraph G, int v)
{
   marked[v] = true;
   for (Edge e : G.adj(v))
      if (!marked[e.other(v)])
         pq.insert(e);
}
```

Prim's runtime

Proposition. Lazy Prim's algorithm computes the MST in time proportional to $E \log E$ and extra space proportional to E (in the worst case).

Pf.

operation	frequency	binary heap
delete min	E	$\log E$
insert	E	$\log E$

Prim's (eager implementation)

Eager solution. Maintain a PQ of vertices connected by an edge to T, where priority of vertex v = weight of shortest edge connecting v to T.

- Delete min vertex v and add its associated edge $e = v$–w to T.
- Update PQ by considering all edges $e = v$–x incident to v
  - ignore if x is already in T
  - add x to PQ if not already on it
  - decrease priority of x if v–x becomes shortest edge connecting x to T

Implementing a PQ with decreaseKey

```
public class IndexMinPQ<Key extends Comparable<Key>>
```

	`IndexMinPQ(int N)`	*create indexed priority queue with indices 0, 1, ..., N – 1*
`void`	`insert(int i, Key key)`	*associate key with index i*
`void`	`decreaseKey(int i, Key key)`	*decrease the key associated with index i*
`boolean`	`contains(int i)`	*is i an index on the priority queue?*
`int`	`delMin()`	*remove a minimal key and return its associated index*
`boolean`	`isEmpty()`	*is the priority queue empty?*
`int`	`size()`	*number of keys in the priority queue*

The idea

- Maintain parallel arrays `keys[]`, `pq[]`, and `qp[]` so that:
  - `keys[i]` is the priority of `i`
  - `pq[i]` is the index of the key in heap position `i`
  - `qp[i]` is the heap position of the key with index `i`
- Use `swim(qp[i])` to implement `decreaseKey(i, key)`.

i	0	1	2	3	4	5	6	7	8
keys[i]	A	S	O	R	T	I	N	G	-
pq[i]	-	0	6	7	2	1	5	4	3
qp[i]	1	5	4	8	7	6	2	3	-