CSE341T/CSE549T **10/13/2014**

Introduction to Graphs and Shortest Paths Algorithms

Lecture 12

Graphs are one of the most important abstractions in algorithms, and are used to solve many problems such as finding paths on maps, analyzing social networks, scene representation in graphs, etc. Today we look at some basics.

1 Formalities

Formally a *directed graph* or (*digraph*) is a pair $G = (V, E)$ where

- V is a set of *vertices* (or nodes), and
- $E \subseteq V \times V$ is a set of *directed edges* (or arcs).

Each arc is an ordered pair $e = (u, v)$ and a graph can have *self loops* (u, u). Directed graphs represent asymmetric relationships. An *undirected graph* is a pair $G = (V, E)$ where E is a set of unordered pairs over V. Each edge can be written as $e = \{u, v\}$, which is the same as $\{v, u\}$, and self loops are **not** allowed. Undirected graphs represent symmetric relationships.

Note that directed graphs are in some sense more general than undirected graphs since we can easily represent an undirected graph by a directed graph by placing an arc in each direction. Indeed this is often the way we represent directed graphs in data structures.

Graphs come with a lot of terminology. Fortunately most of it is intuitive once you understand the concept. At this point we will just talk about graphs that do not have any data associated with edges, such as weights. We will later talk about weighted graphs.

- A vertex u is a *neighbor* of or equivalently *adjacent* to a vertex v if there is an edge between them. For a directed graph we use the terms *in-neighbor* (if the arc points to u) and *out-neighbor* (if the arc points from u).
- The *degree* of a vertex is its number of neighbors and will be denoted as $d_G(v)$. For directed graphs we use *in-degree* ($d_G^-(v)$) and *out-degree* ($d_G^+(v)$) with the presumed meanings.
- For an undirected graph $G = (V, E)$ the *neighborhood* $N_G(v)$ of a vertex $v \in V$ is its set of neighbors, i.e. $N(v) = \{u : \{u, v\} \in E\}$. For a directed graph we use $N_G^+(v)$ to indicate the set of out-neighbors and $N_G^-(v)$ to indicate the set of in-neighbors of v. If we use $N_G(v)$ for a directed graph, we mean the out neighbors. The neighborhood of a set of vertices $U \subseteq V$ is the union of their neighborhoods, e.g. $N_G(V) = \cup_{v \in V} N_G(v)$, or $N_G^+(V) = \cup_{v \in V} N_G^+(v)$.

- A *path* in a graph is a sequence of adjacent vertices. More formally for a graph $G = (V, E)$, $Paths(G) = \{P \in V^+ : 1 \leq i < |P|, (P_i, P_{i+1}) \in E\}$ is the set of all paths in G, where V^+ indicates all positive length sequences of vertices (allowing for repeats). The length of a path is one less than the number of vertices in the path—i.e., it is the number of edges in the path.

- A vertex v is *reachable* from a vertex u in G if there is a path starting at v and ending at u in G. An undirected graph is *connected* if all vertices are reachable from all other vertices. A directed graph is *strongly connected* if all vertices are reachable from all other vertices.

- A *simple path* is a path with no repeated vertices. Often, however, the term simple is dropped, making it sometimes unclear whether path means simple or not (sorry). In this course we will almost exclusively be talking about simple paths and so unless stated otherwise our use of path means simple path.

- A *cycle* is a path that starts and ends at the same vertex. In a directed graph a cycle can have length 1 (i.e. a *self loop*). In an undirected graph we require that a cycle must have length at least three. In particular going from v to u and back to v does not count. A *simple cycle* is a cycle that has no repeated vertices other than the start vertex being the same as the end. In this course we will exclusively talk about simple cycles and hence, as with paths, we will often drop simple.

- An undirected graph with no cycles is a *forest* and if it is connected it is called a *tree*. A directed graph is a forest (or tree) if when all edges are converted to undirected edges it is undirected forest (or tree). A *rooted tree* is a tree with one vertex designated as the root. For a directed graph the edges are typically all directed toward the root or away from the root.

- For a graph $G = (V, E)$ the (unweighted) *shortest path length* between two vertices is the minimum length of a path between them : $SP_G(u, v) = \min\left\{|P| : P \in Paths(G), P_1 = u, P_{|P|} = v\right\}$.

- The *diameter* of a graph is the maximum shortest path length over all pairs of vertices: $Dia(G) = \max\{SP_G(u, v) : u, v \in V\}$.

Sometimes graphs allow multiple edges between the same pair of vertices, called *multi-edges*. Graphs with multi-edges are called *multi-graphs*. We will allow multi-edges in a couple algorithms just for convenience.

By convention we will use the following definitions:

$$\begin{aligned} n &= |V| \\ m &= |E| \end{aligned}$$

Note that a directed graph can have at most n^2 edges (including self loops) and an undirected graph at most $n(n-1)/2$. We informally say that a graph is *sparse* if $m \ll n^2$ and *dense* otherwise. In

most applications graphs are very sparse, often with only a handful of neighbors per vertex when averaged across vertices, although some vertices could have high degree. Therefore the emphasis in the design of graph algorithms is typically on algorithms that work well when the graph is sparse.

2 Representation

Traditionally when discussing graphs three representations are used. All three assume that vertices are numbered from $1, 2, \ldots, n$ (or $0, 1, \ldots, n-1$). These are:

- **Adjacency matrix.** An $n \times n$ matrix of boolean values in which location (i, j) is true if $(i, j) \in E$ and false otherwise. Note that for an undirected graph the matrix is symmetric and false along the diagonal.
- **Adjacency list.** An array A of length n where each entry $A[i]$ contains a pointer to a linked list of all the out-neighbors of vertex i (we assume you can directly index $A[i]$). In an undirected graph with edge $\{u, v\}$ the edge will appear in the adjacency list for both u and v.
- **Edge list.** A list of pairs $(i, j) \in E$.

However, since lists are inherently sequential, in this course we are going to raise the level of abstraction so that parallelism is more natural. Lets assume that we have arrays that allow random access instead. This works very well if all your vertices are numbered from 1 to n, which is generally the case for static graphs. We won't worry about dynamic graphs too much in this course. However, if you do need a dynamic graph where you want to add and remove vertices and edges from the graph, then instead of arrays, you must use maps and tables.

We can now consider the cost of various operations with the different representations — here we assume that the edge array is sorted. If it is not, then the running times would be different.

	Adjacency Matrix		Edge Array		Adjacency Array	
	Work	*Span*	*Work*	*Span*	*Work*	*Span*
`isEdge`$(G, (u, v))$	$O(1)$	$O(1)$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
`map over all edges`	$O(n^2)$	$O(\lg n)$	$O(m)$	$O(\lg n)$	$O(m)$	$O(\lg n)$
`Neighbourhood of` v	$O(n)$	$O(\lg n)$	$O(\lg n + \lvert d_G(v)\rvert)$	$O(\lg n)$	$O(1 + \lvert d_G(v)\rvert)$	$O(\lg \lvert d_G(v)\rvert)$
$d_G(v)$	$O(n)$	$O(\lg n)$	$O(\lg n + \lvert d_G(v)\rvert)$	$O(\lg n)$	$O(1 + \lvert d_G(v)\rvert)$	$O(\lg \lvert d_G(v)\rvert)$

3 Single-Source Shortest Paths

You have learned about single source shortest paths in CSE241. The algorithms you learned were Dijkstra's and Bellman Ford. Recall that Dijkstra's Algorithm worked when there were no negative

weight edges and had a running time of $O(m + n \lg n)$. Bellman ford had running time of $O(mn)$.

First, can we parallelize Bellman Ford? Here's the algorithm:

```
BELLMANFORD(G)
1  for i ← 1 to |V|
2       do for all edges e
3             do RELAX(e)
4  for all edges e
5       do relax e
```

```
RELAX(u, v)
1  if d(v) ≤ d(u) + w(u, v)
2     then d(v) ← d(u) + w(u, v)
```

Where is the parallelism? Can I parallelize the outer for loop? Can I parallelize the inner for loop? It turns out that you can not parallelize either of them unless you allow some sort of concurrent write operation which leads to races. Therefore, we do parallelism based on nodes.

```
BELLMANFORD(G)
1  for i ← 1 to |V|
2       do parallel_for all nodes v
3             do for all edges (u, v)
4                   do RELAX(u, v)
```

Again, there are races in this algorithm as well. Some loop iteration may be updating $d(u, v)$ while some other loop iteration is reading it. How can we fix this? We create a new array of d for each outer iteration.

```
BELLMANFORD(G)
1  for i ← 1 to |V|
2       do parallel_for all nodes v
3             do for all edges (u, v)
4                   do RELAX(u, v, i)
```

```
RELAX(u, v, i)
1  if d_i(v) ≤ d_i(u) + w(u, v)
2     then d_{i+1}(v) ← d_i(u) + w(u, v)
```

We can optimize further by just keeping two of these arrays and switching between them. As it is, this algorithm has work $O(mn)$ and span $O(nd)$, where d is the maximum degree, which can be as much as n. To combat this small parallelism, the relaxation step for each node can be a reduce operation, since it is simply taking a min over many quantities. Therefore, we get work $O(mn)$ and span $O(n \lg n)$.

4 All Pairs Shortest Paths

4.1 Johnson's Algorithm

If there are no negative weight edges, we can run dijkstra's algorithm n times from each vertex. Each of these runs can happen in parallel, giving us work $O(mn+n^2 \lg n)$ and span $O(m+n \lg n)$. What if we do have negative weight edges, but no negative weight cycles?

Graph Re-weighting

For sparse graphs, there is actually another solution that works with negative weights and matches the cost of using multiple instances of Dijkstra's algorithm. The approach boils down to first using Bellman-Ford to convert the graph into one that has no negative weights, and then using Dijkstra's algorithm in parallel across the vertices. The graph is converted in a way such that the shortest path taken between every pair of vertices does not change although the weights of the edges on the path might. Before going further, it is worth thinking about what systematic changes on edges will not change the shortest path taken between two vertices. You might think of simply increasing all the weights on all the edges equally. Unfortunately, this does not work.

Exercise 1 *Come up with a small example that shows that increasing the weight of all edges in the graph equally, changes the shortest paths.*

We could consider multiplying all the weights by any positive number. This will not affect the paths taken, but unfortunately, it does not help us since the negative edges will remain negative. What if we take a vertex and increase every out edge by some constant p_v and decrease every in-edge by the same constant. Does this maintain the shortest paths? Let us consider the possible cases. Firstly, if a path goes through v, then the weight of the path is decreased by v on the edge into v and increased by the same amount on the edge out of v. Therefore, the weight of the path is not changed. Secondly, a path could start at v. In this case all paths out of v increased by the same amount so the shortest path is not affected (although its weight will be). Similarly, for the paths that finishes at v: all paths will be decreased by the same amount. Therefore, this transformation would appear not to change the shortest path between any pair of vertices.

If we can modify the in- and out- weights of one vertex, then we can modify them all. This suggests an idea of assigning values to all vertices and adjusting the edges accordingly. More specifically, we'll assign a real valued "potential" to each vertex, which we indicate as $p(v)$. Now each directed edge (u, v) will be reweighted so that its new weight is

$$w'(u, v) = w(u, v) + p(u) - p(v).$$

(i.e. we add the potential of u going out of u, and subtract the potential of v coming in to v). This leads to the following lemma:

Lemma 1 *Given a weighted directed graph $G = (V, E, w)$ with weight function $w : E \to \mathcal{R}$, and "potential" function $p : v \to R$, then for a graph $G' = (V, E, w')$ with weights*

$$w'(u, v) = w(u, v) + p(u) - p(v),$$

we have that for every path from s to t,

$$W_{G'}(s, t) = W_G(s, t) + p(s) - p(t)$$

where $W_G(s, t)$ is the weight of the path from s to t in graph G.

Proof. Each s-t path is of the form $\langle v_0, v_1, ..., v_k \rangle$, where $v_0 = s$ and $v_l = t$. For every vertex $v_i, 0 < i < k$ in the path the conversion to w' removed p_{v_i} from the weight when entering v_i and added it back in when leaving, so they cancel (the overall sum is a telescoping sum). Therefore, we need only consider the two endpoints, giving the desired result.