Lecture 7 — Introduction to Graphs

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — September 20, 2011

Today:
- Example of using tables and sets for web searching (in the Lecture 6 notes)
- Introduction to Graphs

1 Graphs

Certainly one of the most important abstractions in the study of algorithms is that of a graph, also called a network. Graphs are an abstraction for expressing connections between pairs of items. Graphs can be very important in modeling data, and a large number of problems can be reduced to known graph problems. We already saw this in the case of reducing the shortest superstring problem to the traveling salesperson problem on graphs. A graph consists of a set of vertices with connections between them. Graphs can either be directed, with the directed edges (arcs) pointing from one vertex to another, or undirected, with the edges symmetrically connecting vertices. Graphs can have weights or other values associated with either the vertices and/or the edges.

16 Graph Applications. Here we outline just some of the many applications of graphs:

1. *Road networks.* Vertices are intersections and edges are the road segments between them. Such networks are used by google maps, Bing maps and other map programs to find routes between locations. They are also used for studying traffic patterns.
2. *Utility graphs.* The power grid, the Internet, and the water network are all examples of graphs where vertices represent connection points, and edges the wires or pipes between them. Analyzing properties of these graphs is very important in understanding the reliability of such utilities under failure or attack, or in minimizing the costs to build infrastructure that matches required demands.
3. *Document link graphs.* The best known example is the link graph of the web, where each web page is a vertex, and each hyperlink a directed edge. Link graphs are used, for example, to analyze relevance of web pages, the best sources of information, and good link sites.
4. *Protein-protein interactions graphs.* Vertices represent proteins and edges represent interactions between them that carry out some biological function in the cell. These graphs can be used, for example, to study molecular pathways—chains of molecular interactions in a cellular process. Humans have over 120K proteins with millions of interactions among them.
5. *Neural networks.* Vertices represent neurons and edges the synapses between them. Neural networks are used to understand how our brain works and how connections change when we learn. The human brain has about 10^{11} neurons and close to 10^{15} synapses.

6. *Network traffic graphs.* Vertices are IP (Internet protocol) addresses and edges are the packets that flow between them. Such graphs are used for analyzing network security, studying the spread of worms, and tracking criminal or non-criminal activity.

7. *Social network graphs.* Graphs that represent who knows whom, who communicates with whom, who influences whom or other relationships in social structures. An example is the twitter graph of who tweeted whom. These can be used to determine how information flows, how topics become hot, how communities develop, or even who might be a good match for who, or is that whom.

8. *Graphs in quantum field theory.* Vertices represent states of a quantum system and the edges the transitions between them. The graphs can be used to analyze path integrals and summing these up generates a quantum amplitude (yes, I have no idea what that means).

9. *Semantic networks.* Vertices represent words or concepts and edges represent the relationships among the words or concepts. These have been used in various models of how humans organize their knowledge, and how machines might simulate such an organization.

10. *Scene graphs.* In graphics and computer games scene graphs represent the logical or spacial relationships between objects in a scene. Such graphs are very important in the computer games industry.

11. *Finite element meshes.* In engineering many simulations of physical systems, such as the flow of air over a car or airplane wing, the spread of earthquakes through the ground, or the structural vibrations of a building, involve partitioning space into discrete elements. The elements along with the connections between adjacent elements forms a graph that is called a finite element mesh.

12. *Robot planning.* Vertices represent states the robot can be in and the edges the possible transitions between the states. This requires approximating continuous motion as a sequence of discrete steps. Such graph plans are used, for example, in planning paths for autonomous vehicles.

13. *Graphs in epidemiology.* Vertices represent individuals and directed edges the transfer of an infectious disease from one individual to another. Analyzing such graphs has become an important component in understanding and controlling the spread of diseases.

14. *Graphs in compilers.* Graphs are used extensively in compilers. They can be used for type inference, for so called data flow analysis, register allocation and many other purposes. They are also used in specialized compilers, such as query optimization in database languages.

15. *Constraint graphs.* Graphs are often used to represent constraints among items. For example the GSM network for cell phones consists of a collection of overlapping cells. Any pair of cells that overlap must operate at different frequencies. These constraints can be modeled as a graph where the cells are vertices and edges are placed between cells that overlap.

16. *Dependence graphs.* Graphs can be used to represent dependences or precedences among items. Such graphs are often used in large projects in laying out what components rely on other components and used to minimize the total time or cost to completion while abiding by the dependences.

There are many other applications of graphs.

1.1 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. $N(v) = \{u \mid \{u, v\} \in E\}$. 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 $G = (V, E)$, $Paths(G) = \{P \in V^+ \mid 1 \leq i < |P|, (P_i, P_{i+1}) \in E\}$ 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 : $SP_G(u, v) = \min \left\{ |P| \mid P \in Paths(G), P_1 = u, P_{|P|} = v \right\}$.
- 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.

1.2 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. 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. At the same time we can also loosen the restriction that vertices need to be labeled from 1 to n and instead allow for any labels. Conceptually, however, the representations we describe are not much different from adjacency lists and edge lists. We are just asking you to think parallel and rid yourself of the idea of lists.

When discussing the implementation of graphs it is important to consider the costs of various basic operations on graphs that are needed in various graph algorithms. In particular common useful operations include: finding the degree of a vertex, finding if an edge is in the graph, mapping or iterating over all edges in the graph, and mapping or iterating over all neighbors of a given vertex. In a dynamically changing graph we might also want to insert and delete edges.

Our first representation directly follows the mathematical definition of a graph and simply uses a set of edges $E \subseteq V \times V$. Such an *edge set* representation can be implemented with the set interface described in lecture 6. The representation is similar to an edge list representation mentioned above, but it abstracts away from the particular data structure used for the set—the set could be based on a list, an array, a tree, or a hash table. If we use the balance tree cost model for sets, for example, then determining if an edge is in the graph requires much less work than with an edge list – only $O(\log n)$ instead of $\Theta(n)$ (i.e. following the list until the edge is found). The problem with edge sets, as with edge lists, is that they do not allow for an efficient way to access the neighbors of a given vertex v. Selecting the neighbors requires considering all the edges and picking out the ones that have v as an endpoint. Although with and edge set (but not an edge list) this can be done in parallel with $O(\log m)$ span, it requires $\Theta(m)$ work even if the vertex has only a few neighbors.

To allow for more efficient access to the neighbors of a vertex our second representation uses a table of sets, which we will refer to as an *adjacency table*. The table simply maps every vertex to the set of its neighbors. Now accessing the neighbors of a vertex v is cheap, it just requires a lookup in the table. Assuming the balanced-tree cost model for tables, this can be done in $O(\log n)$ work and span. Once the neighbor set has been pulled out, mapping a constant work function over the neighbors can be done in $O(d_G(v))$ work and $O(1)$ span. Looking up if an edge is in the graph requires the same work and span as with edge sets: $O(\log n)$. This is because we can first look up one side of the edge in the table and then the second side in the set that is returned. We note that an adjacency list is a special case of adjacency table where the table of vertices is represented as an array and the set of neighbors is represented as a list.

The costs assuming the tree cost model for sets and tables can be summarized in the following table assuming the function being mapped uses constant work and span:

	edge set		adjacency table	
	Work	*Span*	*Work*	*Span*
`isEdge`$(G, (u, v))$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
`map over all edges`	$O(m)$	$O(\log n)$	$O(m)$	$O(\log n)$
`map over neighbors of` v	$O(m)$	$O(\log n)$	$O(\log(n) + d_G(v))$	$O(\log n)$
$d_G(v)$	$O(m)$	$O(\log n)$	$O(\log n)$	$O(\log n)$