Lecture 9 — Graphs Introduction

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2012)

Lectured by Guy Blelloch — September 25, 2012

Material in this lecture:

- Graph Introduction
- Graph Representation
- Graph Search

1 Graphs

Graphs (sometimes referred to as networks) offer a way of expressing relationships between pairs of items, and are one of the most important abstractions in the study of algorithms. Graphs can be very important in modeling data. Furthermore a large number of problems can be reduced to known graph problems. 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.

Here we outline just some of the many applications of graphs.

16 Graph Applications.

1. *Social network graphs: to tweet or not to tweet.* 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 follows 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.
2. *Transportation networks.* In road networks vertices are intersections and edges are the road segments between them, and for public transportation networks vertices are stops and edges are the links between them. Such networks are used by many map programs such as google maps, Bing maps and now Apple IOS 6 maps (well perhaps without the public transport) to find the best routes between locations. They are also used for studying traffic patterns, traffic light timings, and many aspects of transportation.
3. *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.

†Lecture notes by Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

4. *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.

5. *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.

6. *Network packet 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. *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.

8. *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.

9. *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.

10. *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.

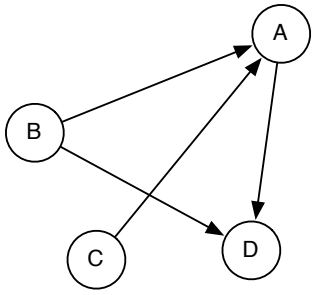
11. *Graphs in quantum field theory.* Vertices represent states of a quantum system and the edges the transitions between them. The graphs can be used to analyze path integrals and summing these up generates a quantum amplitude (yes, I have no idea what that means).

12. *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.

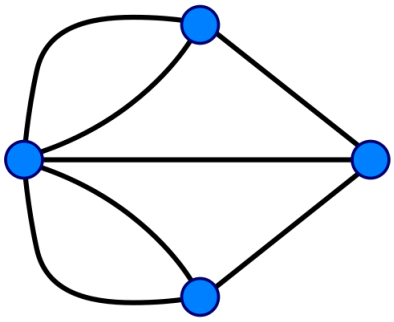
13. *Graphs in epidemiology.* Vertices represent individuals and directed edges the transfer of an infectious disease from one individual to another. Analyzing such graphs has become an important component in understanding and controlling the spread of diseases.

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



An example of a directed graph on 4 vertices.



An undirected graph on 4 vertices[1]

Figure 1: Example Graphs.

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.

1.1 Definitions

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

- V is a set of *vertices* (or nodes), and
- $A \subseteq V \times V$ 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 (i.e., $E \subseteq \binom{V}{2}$). Each edge can be written as $e = \{u, v\}$, which is the same as $\{v, u\}$, and self loops are generally *not* allowed. Undirected graphs represent symmetric relationships.

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 undirected graphs in data structures.

Graphs come with a lot of terminology, but fortunately most of it is intuitive once we 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 in a graph $G = (V, E)$ if there is an edge $\{u, v\} \in E$. For a directed graph we use the terms *in-neighbor* if $(u, v) \in E$ and *out-neighbor* if $(v, u) \in E$.

- 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. $N_G(v) = \{u \mid \{u, v\} \in E\}$. 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 $N_G^+(V) = \cup_{v \in V} N_G^+(v)$.

- The *degree* $d_G(v)$ of a vertex $v \in V$ in a graph $G = (V, E)$ is the size of the neighborhood ($|N_G(v)|$). For directed graphs we use *in-degree* ($d_G^-(v) = |N_G^-(v)|$) and *out-degree* ($d_G^+(v) = |N_G^+(v)|$). We will drop the subscript G when it is clear from the context which graph we're talking about.

- A *path* in a graph is a sequence of adjacent vertices. More formally for a graph $G = (V, E)$, $Paths(G) = \left\{P \in V^+ \mid 1 \leq i < |P|, (P_i, P_{i+1}) \in E\right\}$ 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 path in a finite graph can have infinite length.

- 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 vertex v is *reachable* from a vertex u in G if there is a path starting at v and ending at u in G. We use $R_G(v)$ to indicate the set of all vertices reachable from v 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 *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.

- A directed graph with no cycles is a *directed acyclic graph* (DAG).

- The *distance* $\delta_G(u, v)$ from a vertex u to a vertex v in a graph G is the shortest path (minimum number of edges) from u to v. It is also referred to as the *shortest path length* from u to v.

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

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, at least for this class, is typically on algorithms that work well for sparse graphs.

2 How should we represent a graph?

How we want to represent a graph largely depends on the operations we intend to support. For example we might want to do the following on a graph $G = (V, E)$:

(1) Map over the vertices $v \in V$.

(2) Map over the edges $(u, v) \in E$.

(3) Map over the neighbors of a vertex $v \in V$, or in a directed graph the in-neighbors or out-neighbors.

(4) Return the degree of a vertex $v \in V$.

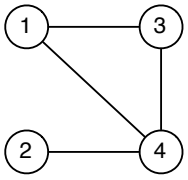
(5) Determine if the edge (u, v) is in E.

(6) Insert or delete vertices.

(7) Insert or delete edges.

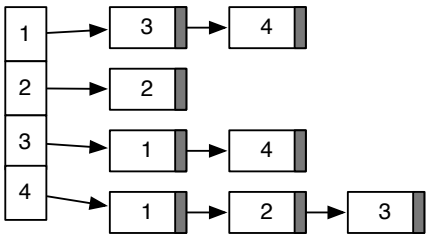
Traditionally, there are four main representations, all of which assume that vertices are numbered from $1, 2, \ldots, n$ (or $0, 1, \ldots, n-1$):

- **Adjacency matrix.** An $n \times n$ matrix of binary values in which location (i, j) is 1 if $(i, j) \in E$ and 0 otherwise. Note that for an undirected graph the matrix is symmetric and 0 along the diagonal.

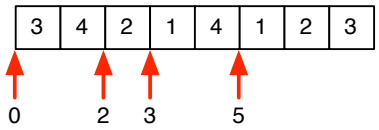


$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

- **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.



- **Adjacency array.** Similar to an adjacency list, an adjacency array keeps the neighbors of all vertices, one after another, in an array `adj`; and separately, keeps an array of indices that tell us where in the `adj` array to look for the neighbors of each vertex.



- **Edge list.** A list of pairs $(i, j) \in E$.

Since operations on linked 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 also want to loosen the restriction that vertices need to be labeled from 1 to n and instead allow for any labels. Conceptually, though, the representations we describe are not much different from adjacency lists and edge lists. We are just going to think parallel and base our view on data types that are parallel.

Here we discuss two representations.

Edge Sets and Tables. Suppose we want a representation that directly follows the mathematical definition of a graph and simply uses a set of edges $E \subseteq \binom{V}{2}$. A moment's thought shows that we can actually support these basic operations rather efficiently using sets. With our ADT for sets, we can implement an *edge set* representation directly. The representation is similar to an edge list representation mentioned before, 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 want to associate data with each edge, we can use and table instead of a set, which maps each edge to its value.

If we use the balanced-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 an 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.

Adjacency Tables. In our second representation, we aim to get more efficiency in accessing to the neighbors of a vertex. This representation, which we refer to as an *adjacency table*, is a table that maps every vertex to the set of its neighbors. Simply put, it is an edge-set table. If we want to associate data with each edge, we can use an edge-table table, where the table associated with a vertex v maps each neighbor u to the value on the edge (u, v).

In this representation, 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 $O(\log d_G(v))$ 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. 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.

Cost Summary. For these two representations the costs assuming the tree cost model for sets and tables can be summarized in the following table. This assumes the function being mapped uses constant work and span.

	edge set		**adj 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	$O(m)$	$O(\log n)$	$O(\log n + d_G(v))$	$O(\log n)$
$d_G(v)$	$O(m)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

3 Graph Search

One of the most fundamental tasks on graphs is searching a graph by starting at some source vertex, or set of vertices, and visiting new vertices by crossing (out) edges until there is nothing left to search. In such a search we need to be systematic to make sure that we visit every vertex that is reachable from the source exactly once. This will require recording what vertices have already been visited so they are not visited a second time. Graph searching can be used to determine various properties of graphs, such as whether the graph is connected or whether it is bipartite, as well as various properties relating vertices, such as whether a vertex u is reachable from v, or finding the shortest path between vertices u and v. In the following discussion we use the notation $R_G(u)$ to indicate all the vertices that can be *reached* from u in a graph G (i.e., vertices v for which there is a path from u to v in G).

For all graph search methods we will consider the vertices can be partitioned into three sets at any time during the search:

1. vertices already *visited* (often denoted as X in the notes),
2. the unvisited neighbors of the visited vertices, called the *frontier* (F in the notes),
3. and the rest.

There are three standard graph search methods: breadth first search (BFS), depth first search (DFS), and priority first search (PFS). All these methods visit every vertex that is reachable from a source exactly once, but the order in which they visit the vertices can differ.

Search methods when starting on a single source vertex generate a rooted *search tree*, either implicitly or explicitly.[2] This tree is a subset of the edges from the original graph. In particular a search visits a vertex v by entering from one of its neighbors u via an edge (u, v). This visit to v adds the edge (u, v) to the tree. These edges form a tree (i.e., have no cycles) since no vertex is visited twice and hence there will never be an edge that wraps around and visits a vertex that has already been visited. We refer to the source vertex as the *root* of the tree. Figure 2 gives an example of a graph along with two possible search trees. The first tree happens to correspond to a BFS and the second to a DFS.

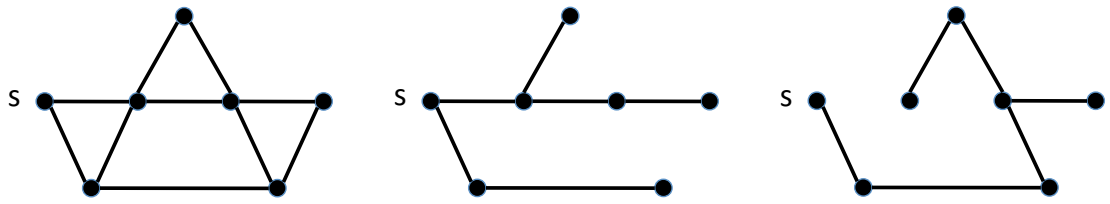


Figure 2: An undirected graph and two possible search trees.

Graph searching has played a very important role in the design of sequential algorithms, but the approach can be problematic when trying to achieve good parallelism. Depth first search (DFS) has a wealth of applications, but it is inherently sequential. Because of this, one often uses other techniques in designing good parallel algorithms. We will cover some of these techniques in upcoming lectures. Breadth first search (BFS), on the other hand, can be parallelized effectively as long as the graph is shallow (the longest shortest path from the source to any vertex is reasonably small). In fact, the depth of the graph will show up in the bounds for span. Fortunately many real-world graphs are shallow. But if we are concerned with worst-case behavior over any graph, then BFS is also sequential.

[2]Note that in assignment 4 you might want to create something other than a tree.