ICS 260—Fall, 2000
Class Notes 1: Asymptotic Analysis and Introduction to Graphs

Notes Originally Developed by Sandy Irani
Revised by Michael B. Dillencourt

1 Analysis of algorithms [CLR, 1.1, 1.2]

Most of this class will focus on the development and analysis of algorithms which involves determining how much of a given resource is used by the algorithm. We will primarily be concerned with the time the algorithm takes to run but we will also be sometimes be interested in the amount of space the algorithm uses as well. Much of the time, we will perform a formal analysis of an algorithm in which its running time is mathematically proven to be bounded by a certain amount. Naturally, the running time of an algorithm depends on the particular programming language used to code the algorithm as well as the particular machine on which the program is run. The running time may also depend on a number of other diverse factors ranging from implementation details to the particular compiler used. It is our goal here to analyze algorithms in a manner which is independent of many of these concerns. For this reason, we will be interested in understanding how the running time of an algorithm grows as the size of the input grows and we will be interested in the order of growth instead of the exact growth. That is, we will be concerned with how the running time of the algorithm grows in the limit as the size of the input grows without bound. Generally (but not always), an algorithm that is asymptotically more efficient will be the fastest for all but very small inputs.

The running time of an algorithm is determined by counting the number of steps it takes given an input of a certain size. We will generally define the notion of 'steps' in a machine-independent way and will vary from algorithm to algorithm. The important point is that each step takes a constant amount of time to execute. The best way to define the size of the input will also vary from application to application.

For the most part, we will also be concerned with performing a worst-case analysis. That is, the running time of an algorithm may vary depending on the actual input (not just the size of the input). For example, an algorithm which sorts n numbers may take less time if the numbers are presented to the algorithm already sorted than if they are given in non-sorted order. We will be interested in bounding (from above and below) the running time of the algorithm on the most time consuming input of a given size.

2 Asymptotic notation [CLR, Section 2.1]

Suppose that an algorithm takes $f(n)$ steps on an input of size n, where $f : \mathcal{N} \rightarrow \mathcal{N}$. ($\mathcal{N}$ denotes the set of natural numbers $\mathcal{N} = \{0, 1, 2, \ldots\}$.) We will not be interested in the exact function f, but instead in the dominant terms in f as n grows.

Definition (Asymptotic Upper Bound): Let f and g be two functions: $f, g : \mathcal{N} \rightarrow \mathcal{N}$.

Suppose that $\exists c$ and $\exists n_0 \in \mathcal{N}$ such that

$$\forall n \geq n_0, \qquad f(n) \leq c \cdot g(n).$$

Then $f = O(g)$.

For example: $n \log n + 20 = O(n \log n)$. $500n^2 \log_{100} n = O(n^3)$.

Note that the O notation serves as an upper bound. That is, it is also the case that $500n^2 \log_{100} n = O(n^2 \log n)$.

Typically, the goal is to say that the running time of an algorithm is $O(g(n))$ where g is a relatively simple function.

Definition (Asymptotic Non-tight Upper Bound): Let f and g be two functions: $f, g : \mathcal{N} \rightarrow \mathcal{N}$. Suppose that $\forall c > 0$ and $\exists n_0 \in \mathcal{N}$ such that

$$\forall n \geq n_0, \qquad f(n) \leq c \cdot g(n).$$

Then $f = o(g)$.

For example, it is the case that $500n^2 \log_{100} n = o(n^3)$, but not the case that $500n^2 \log_{100} n = o(n^2 \log n)$. Note that constant factors, no matter how larger, do not matter: $10^4 n = o(n \log n)$.

Definition (Asymptotic Lower Bound):

Let f and g be two functions: $f, g : \mathcal{N} \rightarrow \mathcal{N}$. Suppose that $\exists c > 0$ and $\exists n_0 \in \mathcal{N}$ such that

$$\forall n \geq n_0, \qquad f(n) \geq c \cdot g(n).$$

Then $f = \Omega(g)$.

Note that $f = O(g)$ if and only if $g = \Omega(f)$.

Definition (Asymptotic Tight Bound):

$$f = \Theta(g) \Leftrightarrow f = O(g) \text{ and } f = \Omega(g).$$

3 Introduction to graphs

Many of the algorithms which we will discuss in this class are graph algorithms. Graphs are used to represent many concepts in computer science. They are used to represent physical phenomena such as circuits, networks and roadways. They are also used to represent more abstract phenomena such as the space of all feasible solutions to a problem, processes in a communication graph or data dependencies in a program.

A graph G is specified by a pair (V, E), where V is the set of nodes and E is the set of edges. Graphs come in two varieties: *undirected* and *directed*. In an undirected graph, each edge in E is specified by an unordered pair $\{u, v\}$, where $u, v \in V$. In a directed graph, each edge in E is specified by an ordered pair (u, v), where $u, v \in V$. Although, technically an edge in an undirected graph should be represented using curly braces $\{u, v\}$ to denote the fact that the vertices are not ordered, typically normal braces are used (u, v). Usually it is understood from the context whether the graph is directed or undirected.

Here are some definitions which we will use throughout the course.

Undirected graph definitions: if (u, v) is an edge, we say that u is *adjacent* to v (and that v is adjacent to u). A **path** is a sequence of vertices $p = v_1, v_2, \ldots, v_n$ such that v_i is adjacent to v_{i+1}. We say that v_1 is **connected to** v_n. A **cycle** is a path that starts and ends at the same vertex (i.e. $c = v_1, v_2, \ldots, v_n$ such that $v_1 = v_n$). A **simple cycle** is a cycle with at least three edges in which no two vertices are repeated except the first and the last ($c = v_1, v_2, \ldots, v_n$ such that $n \geq 2$ and $v_1 = v_n$ and $v_i \neq v_j$ for $i \neq j$ in the range $1, \ldots, n-1$). The **length** of a path is the number of edges in it. A **connected component** is a set of vertices such that each pair of vertices in the set is connected. A **maximal connected component** is a component such that you can not add another vertex to the set and have it still be connected. Any undirected graph can be partitioned into maximal connected components. The **adjacency set** or **neighborhood set** for a vertex x is the set of vertices in the graph which are adjacent to it and is denoted V_x. The **degree** of a vertex is the number of nodes in its neighborhood set.

Directed graph definitions: An edge (v_1, v_2) is specified as an ordered pair. v_1 is the **origin** and v_2 is the **terminus** of the edges (v_1, v_2). A **path** is a sequence of vertices, $p = v_1, v_2, \ldots, v_n$, such that for every consecutive pair of vertices v_i, v_{i+1}, there is an edge from v_i to v_{i+1}. We say that v_1 is **connected to** v_n. The **length** of a path is the number of edges in it. A **cycle** is a path that starts and ends at the same vertex (i.e. $c = v_1, v_2, \ldots, v_n$ such that $v_1 = v_n$). A **simple cycle** is a cycle with at least three edges in which no two vertices are repeated except the first and the last ($c = v_1, v_2, \ldots, v_n$ such that $n \geq 3$ and $v_1 = v_n$ and $v_i \neq v_j$ for $i \neq j$ in the range $1, \ldots, n-1$). A **strongly connected** is a set of vertices such that for every two vertices $v_1, v_2 \in S$, there is a path from v_1 to v_2 **and** a path from v_2 to v_1. A **maximal strongly connected component** is a component such that you can not add another vertex to the set and have it still be strongly connected. Any directed graph can be partitioned into maximal strongly connected components. The **Successors of** x (denoted $Succ(x)$) is the set of vertices y such that there is an edge (x, y) in the graph. The **Predecessors of** x (denoted $Pred(x)$) is the set of vertices y such that there is an edge (y, x) in the graph. The **Outdegree** of x, denotes $Outdegree(x)$, is the number of vertices in $Succ(x)$. The **Indegree** of x, denotes $Indegree(x)$, is the number of vertices in $Pred(x)$.

3.1 Graph Representations

There are two primary methods for representing a graph in a data structure. The first is an **Adjacency Matrix**. Suppose we have a graph $G = (V, E)$ where the number of nodes is n. We'll start with an undirected graph. An Adjacency matrix for G is an $n \times n$ matrix. The entries are all 0 or 1. $A[i, j]$ will denote the entry in row i and column j. We put a 1 in row i, column j if (v_i, v_j) is an edge of G. Otherwise we put a 0. Note that the adjacency matrix of an undirected graph is **symmetric** (i.e. for all i and j, $A[i, j] = A[j, i]$). For a directed graph, we do the same thing: We put a 1 in row i, column j if (v_i, v_j) is an edge of G. Otherwise we put a 0. Note that with an adjacency matrix is no longer necessarily symmetric.

Another way to represent a graph is to have an array of linked lists. This is the **Adjacency List Representation**. There will be one linked list for each node in the graph. Node v_j will be in the list for v_i if (v_i, v_j) is an edge in the graph (i.e. The list for x has each vertex in V_x).

Note that if G is undirected, then if v_i is in v_j's list, then v_j is in v_i's list. If the graph is directed, then node v_j will be in the list for v_i if (v_i, v_j) is an edge in the graph. The list of x is $Succ(x)$. The number of nodes in v_i's list is the outdegree of v_i.

The type of representation one chooses will depend on the operations one will typically perform. For most of the applications in this class, the adjacency list representation will be the most useful.

To answer a query of the form: 'is $\{v_i, v_j\}$ an edge?' requires $O(1)$ time for the adjacency matrix, and $\Theta(degree(v_i))$ in adjacency list.

To run through the neighbors of a given node requires $\Theta(n)$ time for adjacency matrix, and $\Theta(degree(x))$ time for adjacency list.

4 Searching a graph

A fundamental operation that will be used frequently throughout this course is to determine whether there is a path from u to another vertex v. The following procedure will finish with the set S containing all those nodes v such that there is a path from s to v, where s is the initial vertex on which the procedure is called. It can be run on a directed or undirected graph.

```
GraphExplore(G,s)
Mark all edges as unused
S = {s}
while (there is an unused edge (u,v) connected to any node u ∈ S)
    choose edge (u,v)
    mark edge (u,v) used
    S = S ∪ {v}
```

For an undirected graph, one may want to find all the maximal connected components of a graph G. This can be achieved by repeatedly calling the procedure GraphExplore:

```
ConnectedComponents(G)
i = 0
while (nodes remain in V)
    Pick a node v ∈ V.
    Increment i
    Call GraphExplore(G,v)
    Return set S_i
    Delete nodes in S_i from V
```

What is the running time of this algorithm? ConnectedComponents takes $O(n)$ time, except for the calls to GraphExplore. Suppose that there are n_i nodes and m_i edges in the i^{th} connected component. Then the i^{th} call to GraphExplore takes $\Theta(n_i + m_i)$ time. Since $\sum n_i = n$ and $\sum m_i = m$, the total running time is $\Theta(n + m)$.

4.1 Breadth First Search

There is a lot of latitude in the procedure GraphExplore as to how the unused edges are chosen and in what order they are used. In this first algorithm, called *Breadth First Search*, the edges will be put on a queue. That is, they are inserted and deleted from a data structure in first-in-first-out order.

```
GraphExploreBFS(G,s)
S = {s}
Put all edges (s,v) on the queue.
while (the queue is not empty)
    dequeue edge (u,v)
    if (v ∉ S)
        S = S ∪ {v}
        Enqueue all edges (v,w)
```

We will see later that this breadth-first-search is useful for computing shortest paths from s to all other nodes in the graph.

4.2 Depth First Search

The next algorithm we will consider stores edges using a stack. That is, edges are inserted and deleted in a last-in-first-out order.

```
GraphExploreDFS(G,s)
S = {s}
Push all edges (s,v) on the stack.
while (the stack is not empty)
    pop edge (u,v)
    if (v ∉ S)
        S = S ∪ {v}
        Push all edges (v,w) onto the stack.
```

Depth first search has many uses. DFS can be used to compute the strongly connected components of a directed graph. It can also be used to determine whether or not a graph has a cycle. For a acyclic graphs (i.e. graphs without a cycle), it is sometimes useful to compute a *topological ordering* of the vertices. This is an ordering such that if u precedes v in the ordering, then there is no edge from v to u. For example, suppose we have an acyclic graph representing precedence constraints on the operations is a program. Each node represents an operation and there is an edge from node u to node v if the operation represented by node u must precede the operation represented by node v. A topological ordering is an ordering of the operations such that if the operations are executed in that order, then none of the precedence constraints will be violated.