Lecture Note Summary #1

Vincent Zimmer
Mark Pustilnik
Robert Laggart
Mark Autry

1 October 1997

1 Course Goals

The derivation of the word algorithms does is not etymologically:

algiros [painful] + arithmos [number] ==¿ "Painful Number"

but stems from Al-Khowarizmi "Rules of Restoration and Reduction", or "algebra." The *Oxford English Dictionary later defines "algorist" as "one skillful in reckonings or figuring."*

At the successful completion of CSE 589, the student should have accomplished the following goals:

- An appreciation for application of algorithmic techniques in the real world. – To utilize real world problems, generated from the student's work environments, for the purpose of demonstrating application of algorithmic techniques.
- A larger toolbox. – Increase the number of tools students have with which to find solutions for problems encountered. Self - explanatory.
- A better sense of how to model problems encountered in real life as well-known algorithmic problems. – Become able to map problems to algorithms rather than reinventing the wheel for each problem. *This is usually the hardest part of the process!*
- A deeper understanding of the issues and tradeoffs involved in algorithmic design.

The plan for the first lecture was to provide a whirlwind version of CSE 421, the undergraduate algorithms course. The first lecture was to cover an introduction to algorithm design and analysis (this corresponds to chapters 1 and 2 in the recommended text.[1]), dynamic programming (chapter 16), easy graph problems (chapters 23 - 26), and hard graph problems (roughly equivalent to chapter 36). Unfortunately time ran out and the material on hard graph problems will be covered in the next lecture.

2 Introduction to algorithm design and analysis

The primary goals for algorithm design are correctness and efficiency. An algorithm is said to be CORRECT if, for every input, it halts with the correct output. Contrary to what one might expect, incorrect algorithms can sometimes be useful if their error rate can be controlled. Developing a generic method with which to analyze algorithms and to compared their use of resources with each other will usually separate efficient algorithms from inferior algorithms. The inferior algorithms may still be correct and solve the given problem, however using more time and memory than more efficient algorithms. Algorithms are one of the core technologies of computer science. Usefulness of algorithms is independent of languages and technologies.

2.1 Random Access Machine model

Algorithms can be measured in a machine-independent way using the Random Access Machine (RAM) model. This model assumes a single processor. In the RAM model, instructions are executed one after the other, with no concurrent operations. This model of computation is an abstraction that allows us to compare algorithms on the basis of performance. The assumptions made in the RAM model to accomplish this are:

- Each simple operation takes 1 time step.
- Loops and subroutines are not simple operations.
- Each memory access takes one time step, and there is no shortage of memory.

For any given problem the running time of an algorithms is assumed to be the number of time steps. The space used by an algorithm is assumed to be the number of RAM memory cells.

2.2 Types of Complexity

Three types of complexity could be considered when analyzing algorithm performance. These are worst-case complexity, best-case complexity, and average-case complexity. Only worst-case complexity has found to be useful. Worst-Case running time for a function, $f(n)$ such that $f(n) = a * n^2 + b * n + c$ where a, b and c are constants can be described as a higher bounding limit for the shortest possible running time for any given "n" beyond some instance of "N"(big "n") such that $n \geq N$. In this case, $f(n) = O(g(n)) \Leftrightarrow c_1 * g(n)$ is the higher bounding limit on $f(n)$. $f(n) \geq c_1 * g(n)$. We will soon see how $g(n) = n^2$ for this example. More useful than best-case, since worst-case will tell how long an algorithm will run in the worst possible situation.

Average-Case running time for a function, $f(n)$ such that $f(n) = a*n^2+b*n+c$ where a, b and c are constants can be described as an average between best-case and worst-case. Average-case isn't useful because in most cases it's hard to define the average.

Best-Case running time for a function, $f(n)$ such that $f(n) = a * n^2 + b * n + c$ where a, b and c are constants can be described as a lower bounding limit for the shortest possible running time for any given "n" beyond some instance of "N"(big "n") such that $n \geq N$. In this case, $f(n) = \Omega(g(n)) \Leftrightarrow c_0 * g(n)$ is the lower bounding limit on $f(n)$. $f(n) \geq c_0 * g(n)$. We will soon see how $g(n) = n^2$ for this example. Best-case performance isn't really useful because it doesn't reflect normal operation of the algorithm.

2.3 Big Oh Notation

[1] The goals of big Oh-notation are to ignore constant factors which do not affect exponential orders of growth, and to ignore the details which do not impact the comparisons of algorithms. Abstract polynomial functions to their highest degree term while ignoring all constants. $f(n) = a * n^2 + b * n + c$ becomes $f(n) = O(n^2)$

- $f(n) = O(g(n)) \Leftrightarrow c_1 * g(n)$ is UPPER bound on $f(n) \Leftrightarrow \exists c_1, N$ such that for $n >= N, f(n) = c_1 * g(n)$
- $f(n) = \Omega(g(n)) \Leftrightarrow c_0 * g(n)$ is LOWER bound on $f(n) \Leftrightarrow \exists c_0, N$ such that for $n >= N, f(n) = c_0 * g(n)$
- $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n))$ and $f(n) = \Omega(g(n)) \Leftrightarrow \exists c_0, c_1, N$ such that for $n \geq N, c_0 * g(n) \leq f(n) \leq c_1 * g(n)$

In studying the ASYMPTOTIC efficiency of algorithms, use only input sizes large enough to make only the order of growth of the running time relevant. The running time of an algorithm increases with the size of the input, as the size of the input increases without bound. As input size becomes extremely large, only the highest degree polynomial terms are of any significance, and even with ignoring constant factors, the growth rates

[1]See chapter 2.1 of text[1], pages 23-31, for full discussion of aymptotic notation. (Also known as big Oh)

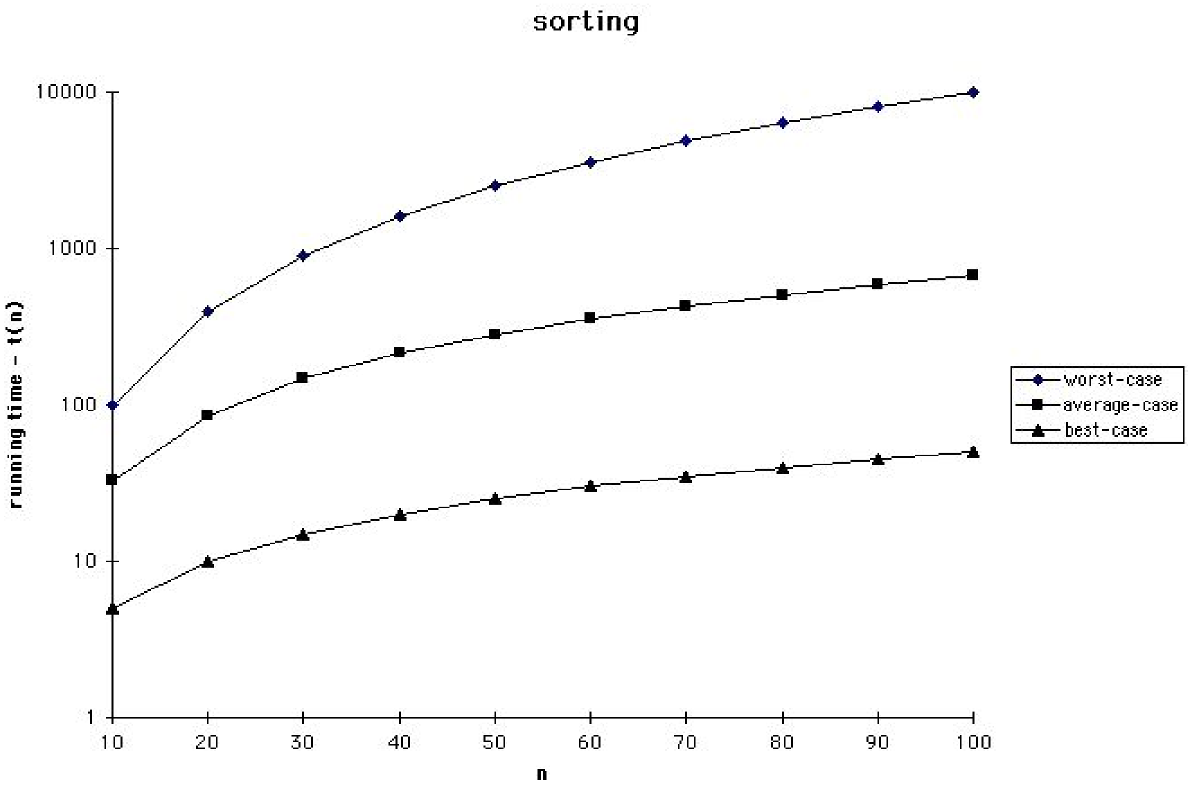

Figure 1: Relationship of Complexity Types

themselves will provide a excellent indicator of whether a given algorithm will be able to run in a reasonable amount of time on a problem of a given size. Big Oh notation and worst-case analysis are tools that greatly simplify our ability to compare the efficiency of algorithms.

2.4 Algorithm design

Accurately modeling a problem is an art, and the key to applying algorithm design techniques to real-world problems. Most algorithms are designed to work on rigorously defined abstract structures. Algorithms and data structures are inseparable, however, by data structures, we do not mean data structure implementation for any specific OS or language.

Two basic paradigms used for modeling problems are divide and conquer and dynamic programming.

- Divide and conquer usually splits the problem in half and solves for each half, then combines the result to get the full solution.
- Dynamic programming is typically applied to optimization problems. The development of a dynamic-programming algorithm can be divided into a series of four steps:
  1. Characterize the structure of an optimal solution.
  2. Recursively define value of an optimal solution.
  3. Compute the value of an optimal solution in bottom-up fashion.
  4. construct an optimal solution from computed information.

2.5 Dynamic Programming

2.5.1 Example: Approximate String Matching

- P – a pattern string
- T – a text string
- Edit distance between P and T is smallest number of changes to transform a substring of T into P, where changes are:
  - Substitution ...KAT... C for K ...CAT...
  - Insertion ...CT... A inserted ...CAT...
  - Deletion ...CAAT... A deleted ...CAT...

2.5.2 Dynamic Program for Approximate String Matching

- $D[i,j] = \min \#$ differences between $P[1..i]$ and segment of T ending at j.
-

$$D[i,j] = \left\{ \begin{array}{ll} \min[D[i-1,j-1], D[i-1,j-1]+1, D[i-1,j]+1] & \text{if } T_j = P_i \\ \min[D[i-1,j-1]+1, D[i-1,j-1]+1, D[i-1,j]+1] & \text{if } T_j \neq P_i \\ D[0,j] = 0 & \\ D[i,0] = i & \end{array} \right\}$$

2.5.3 Dynamic Programming Summary

1. Formulate answer as recurrence relation or recursive algorithm.
2. Show that number of values of recurrence are bounded by a polynomial function.
3. Specify order of evaluation for recurrence so have partial results when you need them.

Dynamic programming works particularly well for optimization problems with inherent left to right ordering among elements. The resulting global optimum often much better than solution found with heuristics. And once understood, dynamic programming is usually easier to work algorithm out from scratch than to look up algorithm.

3 Easy Graph Problems

3.1 Graph Terminology

Graph $G = (V, E)$ is a set of vertices V, with set of edges E.

Directed Edges which have a specific direction associated with them so that each edge can be traversed in one direction only.

Undirected edges do not have a direction associated so that each edge can be traversed in each direction.

Path The set of edges and vertices which connect one vertex to another vertex.

Connected Two vertices which are connected via a valid path.

Strongly Connected Connected in more than one path, such that no single vertex or single path can be remove and make the graph unconnected.

Degree Equals the number of edges going out/in from/to a single vertex.

Cycle A path within a graph which can loop back to an ancestor node/vertex.

Tree A graph which contains no cycles.

3.2 Traversing a Graph

Key idea: mark each vertex when it's visited, keep track of what's been explored. Vertex can exist in one of three distinct states; UNDISCOVERED, DISCOVERED, and FULLY-EXPLORED. Undiscovered means a vertex has not previously been visited. Discovered means the vertex has been visited before, however not all edges leading away from the vertex have been explored. Fully-explored means that the vertex has no unexplored edges attached to it. Maintain a structure of vertices which are discovered but not fully-explored. A queue structure, FIFO, would mean performing a breadth first search, while a stack structure, LIFO, would mean performing a depth first search. Regardless of what order, every edge must be considered twice.

3.3 Depth-First Search

```
DFS(G,U)
  state[u] = ``discovered'';
  for each v in Adj[u] do
    if state[v] = ``undiscovered''
      parent[v] = u;
      DFS(G,v);
  state[u] = ``fully-explored''
```

```
DFS-graph(G)
  for each vertex u in V[G] do
    state[u] = ``undiscovered'';
  for each vertex u in V[G] do
    if state[u] = ``undiscovered''
      DFS[G,u];
```

The following definitions for depth-first-search graphs were taken from the recommended text.[1]

Tree edges are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was discovered by exploring edge (u, v).

Back edges are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Self-loops are considered to be back edges.

Forward edges are those nontree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.

Cross edges are all other edges. They can go between vertices in the same depth-first tree, as long as one vertex is not an ancestor of the other, or they can go between vertices in different depth-first trees.

Depth-first-searches on undirected graphs can only have tree and back edges in their trees. Depth-first-searches on directed graphs can have all four types of edges.

3.4 Topological Sorting

[2] Perform a depth-first-search. Label vertices in reverse order as they are marked fully-explored. Consider the edge (u, v), encountered during exploration of u. If v undiscovered $\Rightarrow$ DFS(v) before continuing with $u \Rightarrow v$ marked fully-explored before u. If v discovered, but not fully-explored $\Rightarrow$ this can't happen because topological sorting can only be performed on directed-acyclic-graphs(DAG). That is, a directed graph without cycles. If v is fully-explored $\Rightarrow$ labeled before $u \Rightarrow u$ before v in topological order.

3.5 Planarity Testing

Given a graph G, can G be drawn in the plane such that no two edges cross? This algorithm is $O(|V| + |E|)$, using depth-first-search in subtle and clever way.

3.6 Minimum Spanning Tree

[3] The spanning tree of a graph G is a subgraph of G that has same set of vertices as G but is a tree. The Minimum spanning tree of a weighted graph G is a spanning tree of G whose edges sum to the minimum weight for all spanning trees of G. Greedy algorithms give optimal solution for minimum spanning trees.

3.7 Prim's algorithm

[4]

[2] See chapter 23.4, pages 485-487 of Cormen et.al.[1]
[3] See chapter 24 of Cormen [1] pages 498-502.
[4] See chapter 24.2 of text [1] pages 505-510.

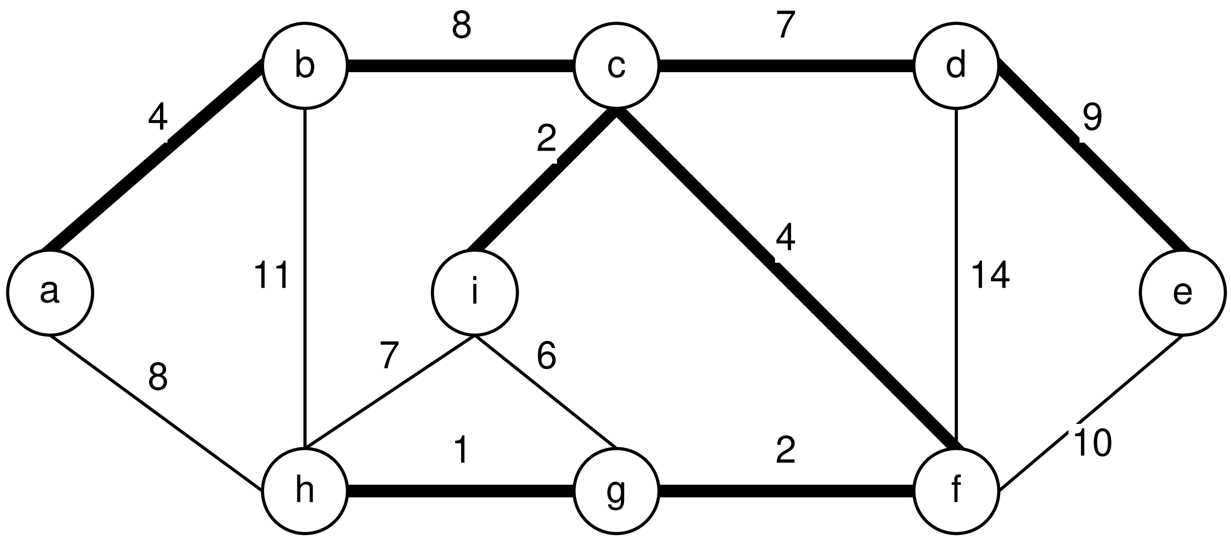

Figure 2: Minimum Spanning Tree resulting from Prim's Algorithm

```
Start with arbitrary vertex
While there are vertices in tree connected to vertices outside
  select minimum weight edge between tree and outside
  add selected edge and vertex to tree.
```

Prim's algorithm is a simple greedy algorithm. The class notes indicate that Prim's algorithm as described above has a performance complexity of $O(nm)$. However, Cormen et.al.[1] claim a running time performance of $O(E \lg V)$ assuming an algorithm that uses a priority queue implemented using a binary heap. Cormen et. al.[1] claim a better performance for Prim's algorithm when the priority queue is implemented using Fibonacci heaps. In this case the running time can be reduced to $O(E + V \lg V)$.

3.8 Dijkstra's Algoritm for shortest path problems

[5] This algorithm applies for cases where all edge weights are non-negative. This algorithm finds the shortest path from a given vertex v to other vertices on the graph. At the beginning of an iteration, have a set of vertices S to which we know the shortest path from $v \Rightarrow$ grow set by one. This algorithm exhibits dynamic programming-like strategy.

[5] See chapter 25.2 of text [1], pages 527-531.

References

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.