Computational Optimization
ISE 407

Lecture 5

Dr. Ted Ralphs

Reading for this Lecture

- Aho, Hopcroft, and Ullman, Chapter 1
- Miller and Boxer, Chapters 1 and 5
- Fountain, Chapter 4

Problems, Instances, and Algorithms

- A *problem* P is a mapping of a set of *inputs* to specified *outputs*.
- An *instance* is a problem along with a particular input.
- An *algorithm* is a procedure for computing the output expected from a given input.
- Example: Traveling Salesman Problem
  - Given an undirected graph $G = (N, A)$ and non-negative arc lengths d_{ij} for all $(i, j) \in A$, find a cycle that visits all nodes exactly once and is of minimum total length.
  - How do we specify an instance?

Algorithms

- An algorithm is a procedure for evaluating the function associated with a given problem.
- We assume that the input is a string of characters formed from an associated *alphabet* (usually denoted Γ).
- An algorithm *solves* a problem P if that algorithm produces the expected output for any input.
- In this case, the algorithm is said to be *correct*.
- In complexity theory, we typically consider simple problems where the output is either `YES` or `NO` (feasibility problems).
- The set of all input strings that produce the answer `YES` is the *language* associated with the problem.
- A correct algorithm is said to be one that "recognizes the language."
- Note that the alphabet of a machine language computer program is only two characters, 0 and 1 and this is the alphabet we usually assume.

Assessing Algorithms

- The property of an algorithm that we are interested in first and foremost is correctness.
- Beyond that, there will often be multiple correct algorithms (or implementations of the same algorithm) to choose from in solving a problem.
- When assessing each candidate, a primary basis for assessment is how "efficient" it is (usually in terms of "running time").
- In many cases, it is not possible to produce the solution exactly (problems with irrational solutions).
- In such cases, we also generally want know if we can obtain a guarantee for how close the approximate solution is to the true solution.
- This is the domain of *numerical analysis*, which we will later in the course.
- Often, there is a tradeoff between efficiency and accuracy that must be analyzed.

Example: Log Function in Python

```
>>> math.log10(1000)
3.0
>>> math.log(1000, 10)
2.9999999999999996
```

- Here, we are using two different algorithms for calculating the logarithm base 10 of 1000.
- One is more accurate than the other.
- We will look at issues of floating point error later in the course.

Proving Correctness

- Correctness of an algorithm must be proven mathematically.
- For the algorithms we'll study, this will not be easy in some cases.
- Many algorithms simply require that the output satisfy some easily verifiable criterion and follow one of the following paradigms.
  - Iterative: The algorithm executes a loop until a termination condition is satisfied.
  - Recursive: Reduce the problem to one or smaller instances of the same problem.
- In both cases, we must prove both that the algorithm terminates and that the result is correct by ensuring the criterion is satisfied.
  - Correctness of iterative algorithms is often proven by showing that there is an *invariant* that holds true after each iteration.
  - Recursive algorithms are almost always proven by an induction argument.
  - More complex algorithms can be formed by composition of simpler algorithms.

Proving Correctness for Optimization Problems

- Correctness of an optimization algorithm requires more work.
- Typically, we will need to ensure that some particular optimality criteria are achieved.
- Proving that the algorithm terminates will involve showing that a certain progress towards achieving optimality is made on each step.
- These proofs are often not very obvious.

Example: Insertion Sort

- A simple algorithm for sorting a list of numbers is insertion sort.
- The algorithm we show here is iterative, although there is also an obvious recursive implementation.
- All things being equal, iterative algorithms are usually preferred in practice to avoid the overhead of function calls (more on this later).

```
def insertion_sort(l):
    for i in range(1, len(l)):
        save = l[i]
        j = i
        while j > 0 and l[j - 1] > save:
            l[j] = l[j - 1]
            j -= 1
        l[j] = save
```

- Why is this algorithm correct? What is the invariant?

Finding a Simple Path in a Graph

- Here, we try to find a simple path in a graph using a recursive algorithm.
- We must pass in a vector of bools to track which nodes have been visited.

```
def SPath(G, v, w, visited = None)
    if visited == None:
       visited = {}
    if v == w:
       return [v]
    visited[v] = True
    for n in v.get_neighbors():
       if n not in visited:
          path = SPath(G, n, w, visited)
             if path != None
                path.append(n)
                return path
    return None
```

- How would we formally prove the correctness of this algorithm?

Design of Algorithms

- The *design* of an algorithm involves assess and improving its efficiency.
- How do we know if an algorithm is correct and what do we mean by "efficient"?
- A single mathematical algorithm may have multiple different implementations with different efficiencies.
- A single problem may have multiple different algorithms.
- How do we compare?

Analysis of Algorithms

- The goal of algorithm analysis is to determine the resources required to execute a given algorithm in practice.
- Ultimately, we want to use thi sinformation to choose the *right* algorithm for particular use cases.
- Generally speaking, *time*, *space* (memory), and *number of cores* are the most important resources to consider.
- The comparison between algorithms is based on comparison of the probability distributions of certain "measures of effectivness."
- We are usually (but not always) interested in applying the algorithms on a certain well-defined *class*.
- The class usually contains an infinite number of instances, so understanding the distributions can be extremely challenging.

Empirical Analysis

- An apparently "simple" approach to evaluating effectiveness in practice is to do an *empirical analysis*.
- This approach essentially aims to construct approximations of the relevant distributions.
- Rigorous empirical analyses are more difficult than they appear because of factors that are difficult to take into account.
  - What instances should we use to do the testing?
  - How should we measure "effectiveness" (CPU time, wallclock time)?
  - How should we take into account variability?
  - What platform should we do our testing on?
- It is not obvious how to perform a rigorous analysis in the presence of so many factors.
- Practical considerations prevent complete testing.
- Even more challenges arise in the analysis of parallel algorithms.

Theoretical Analysis

- Formal theoretical analysis of algorithms attempts to abstract away some of these issues.
  - We consider a formal model of a computer called the *model of computation*.
  - We use summary measures that account for effectiveness across all instances.
  - We explicitly account for the dependence of efficiency on important properties of individual instances.
- The result of the most common type of analysis is a *running time* function.
- This function represents execution time as a function of properties of the instance (typically *size*).

Measures of Effectiveness (Single Instance)

- For the time being, we focus on sequential algorithms.
- What is an appropriate measure of effectiveness for a single instance?
- What does the end user care about?
  - Running time (resources required to complete computation)?
  - Best result given limited resources (progress)?
  - Accuracy?
- What is the goal of the analysis?
  - Compare two algorithms.
  - Improve the implementation of a single algorithm.
- How should we measure resource usage? What proxies are most appropriate?
  - Empirical running time (CPU time, wallclock)
  - Theoretical operation counts
- We will discuss measure of effectiveness in more detail in future lectures.

Summary Statistics

- What we really want is a way to compare algorithms by their distributions.
- Summarization is the easiest way to compare (although there are better ways).
- Empirical: Given a measure of effectiveness and a test set, we construct an empirical CDF over a test set and then compute a statistic.
  - Arithmetic mean
  - (Shifted) Geometric mean
  - Variance
- These simple statistics hide information useful for comparison.
- We will see alternatives in later lectures.
- Theoretical: We try to summarize across "all possible" instances or at least some statistical distribution of instances.
  - Average-case: Try to determine the expected running time an algorithm analytically.
  - Worst-case analysis: Provide an upper bound on the running time of an algorithm for *any* instance in a given set.

Drawbacks of Three Approaches

Empirical	1. Depends on programming language, compiler, etc. 2. Time consuming and expensive 3. Often inconclusive
Average-Case	1. Depends on probability distribution 2. Difficult to determine appropriate distribution 3. Intricate mathematical analysis 4. No information on distribution of outcomes
Worst-Case	1. Influenced by pathological instances