Approximation Algorithms

Daniel F. Savarese

Last Updated: 2012-05-23

Note

This article was published originally in the November 2002 issue of *Java Pro* with the title "Close to Correct." The code examples in the column were an afterthought. My objective was to give a high-level overview of NP-completeness and the motivation for approximation algorithms, with the hope that the reader would delve into the topic in more depth on his own.

The original article made enough of an impression to have been cited as a reference in the following paper:

> J. Martens, David V. Judge, and Jimmy A. Bigelow, "Uncertainties Associated With Many-Port (>4) *S*-Parameter Measurements Using a Four-Port Network Analyzer,"*IEEE Transactions on Microwave Theory and Techniques*, vol. 52, no. 5, pp. 1361-1368, May 2004.

The set-covering algorithm demonstrated in the code example was used to choose an optimal collection of vector network analyzer port calibrations.

Changes. The code examples have been updated to use generics and the so-called "enhanced for loop."

Algorithms vis-à-vis Everyday Programming

Programming is often more about writing business rules and less about implementing, or even inventing, algorithms to solve problems more efficiently. Programming involves using a third-party API or designing an API more than it involves devising data structures that make optimal use of available system resources. Enterprise development is concerned more with systems integration than systems optimization. Perhaps programming is now less about computer science and more about software engineering—if the two can even be considered separately.

It's all right that most programming tasks do not require deep mathematical or algorithmic insights. Most commonly required algorithms and data structures are already implemented in standard libraries such as the C++ Standard Template Library or the Java core APIs. When you write a business application, you don't want to get hung up on the details of open address hashing and linear probing; you just want access to a ready-made container that maps a key to a value relatively efficiently.

Modularization yields reuse and reuse yields greater productivity. Just look at how game programming has changed since the late 1980s and early '90s. Game programming used to require a more than superficial knowledge of computer graphics algorithms as well as platform-specific systems programming esoterica, such as how to use ModeX graphics modes with DOS. Today, you don't have to know a thing about computer graphics theory; pick your favorite graphics library and games programming API and you can focus on the business of writing a great game rather than great graphics routines. Heck, most of the optimization has moved into the graphics hardware anyway.

With a preponderance of libraries available for most common programming tasks, are we running the risk of forgetting how basic algorithms work? After all, you don't need to know anything about internal combustion engines to drive a car; and you don't need to know anything about red-black trees to use a `TreeMap`. I have worked with programmers who lacked a knowledge of fundamental algorithms and data structures, yet this did not seem to impair the quality of their work. Evidently, when you're translating Simple Object Access Protocol (SOAP) messages into SQL database queries, knowing how to implement Dijkstra's algorithm to solve the single-source shortest-paths problem doesn't help you at all. You can apply basic logical thought successfully to solve everyday programming problems; this may explain why so many programmers are self-taught.

[1] https://www.savarese.org/

Although I don't think programmers as a group will forget how fundamental algorithms work, the economics of software development dictates that only a small percentage of programmers will have experience implementing them. Otherwise, we'd all be spending our time reinventing the proverbial wheel. Besides, every resourceful programmer understands the benefit of maintaining a technical reference library. If you don't know how to do something, just look it up. Even if you're never going to implement a particular algorithm, however, it can be helpful to know in advance how it works so that, for example, you know when to use a `TreeMap` instead of a `HashMap`.

Polynomial-Time Algorithms

The ability to recognize when a particular programming task cannot be solved efficiently can also be helpful. Not every problem can be solved exactly without resorting to a brute-force or exponential-time algorithm. Many real-world applications can tolerate such an approach because they use small amounts of input data that prevent the algorithms from taking an inordinately long time to finish running. Equal or greater numbers of applications use large input data sets and cannot tolerate the excessive execution times of exponential-time algorithms. In these cases, it is necessary to settle for an approximately correct answer using what is known as an approximation algorithm.

How can you tell if a problem is so intractable as to necessitate the use of an approximation algorithm? Most fundamental algorithms have worst-case running times that can be expressed as a polynomial function of the size of their inputs. In other words, the running time can be represented as a polynomial of the form $an^k + bn^{k-1} + cn^{k-2} \ldots$, where n is the size of the input to the algorithm. For example, most sorting algorithms have a worst-case running time of knlog(n) (this is less than kn^2 and therefore polynomial time), where n is the number of items being sorted. The class of problems that can be solved by algorithms with polynomial running times is said to belong to the complexity class *P*. Algorithms in class *P* are considered tractable.

The composition of two polynomial-time algorithms yields a polynomial-time algorithm. This is an important property because it tells you that a program that makes use of only polynomial-time algorithms will have a running time no worse than the highest order polynomial of the running times of its component algorithms. Understanding polynomial-time algorithm composition helps you determine where to focus your attention when optimizing a program. There's little point in improving the running time of an algorithm from $2n^2$ to n^2 when your program's overall running time is dominated by an algorithm with a running time of n^3.

NP-Complete Problems

Another class of computational problems commonly encountered cannot be solved in polynomial time, but the correctness of their solutions can be verified in polynomial time. These problems are said to belong to the complexity class *NP*, which stands for nondeterministic polynomial time. All elements of class *P* are a subset of *NP*. That is, any problem that can be solved by a polynomial-time algorithm can have its solution verified by a polynomial-time algorithm. Elements of a special set of problems in *NP* possess the property that if they can be solved in polynomial time, then all problems in *NP* can be solved in polynomial time; and therefore it will be true that *NP=P*. These problems are called NP-complete and can be thought of as the hardest problems in *NP*. It is believed that *NP≠P*, and no polynomial time algorithm has yet been found to solve an NP-complete problem. But it has not been proven that none exists.

When you run into an NP-complete problem, you know you can't compute an exact answer in a reasonable amount of time; but how do you know you've run into an NP-complete problem? The handy thing about NP-complete problems is that if you can show that a problem is equivalent to another problem already known to be NP-complete (a process known as reduction), you've proven the problem is NP-complete. That leads to the crux of this column. You can't really figure out if a problem is NP-complete if you aren't already familiar with NP-complete problems. It behooves us as programmers, whether self-trained or formally instructed, to become familiar with the theory of algorithms. Even if you never have to implement classic algorithms or work with concepts such as NP-completeness, a familiarity with a wide variety of algorithms gives you the ability to recognize related problems and adapt an algorithm to solve a new problem you encounter. You can also identify which algorithm implementations are most appropriate to use in your programs if you have a good understanding of how their running times and memory use vary based on input size and, in the case of data structures, the application of specific operations.

Three frequently occurring NP-complete problems are the travelling-salesman problem, the subset-sum problem, and the set-covering problem.

The Travelling-Salesman

In the travelling-salesman problem, a salesman has to visit a set of cities, but can visit each city only once, except for the last city in the tour, which must be the same as the first city he started from. The total cost of visiting all of the cities must be the minimal possible cost. The cost may be in terms of distance, fuel, money, or whatever metric is appropriate to the real-world situation. Independent of the nature of the cost metric, the problem is modeled as a set of vertices connected by edges with associated weights, forming a graph. The travelling-salesman problem is encountered in logistics systems that must deliver goods or pick up goods and return to a central distribution location.

Subset-Sum

The subset-sum problem asks if a subset of numbers in a set of positive integers adds up exactly to a given value. A relaxed version of the problem tries to identify a subset of numbers that adds up to a maximum value no greater than a given value. This problem is, again, encountered in logistics systems, where you may be trying to load up vehicles with as many packages as possible without exceeding the weight limit each vehicle can carry. UPS and FedEx are probably masters at approximating solutions to the travelling-salesman and subset-sum problems.

Set-Covering

Explaining the set-covering problem is a little more verbose. Suppose you have a set of items called *X* and a family of subsets of those items called *F*. The set-covering problem attempts to find a set of subsets contained in *F* that together contain each of the elements in *X* at least once. Furthermore, this covering set of subsets must contain the minimal number of subsets necessary to cover the elements in *X*. This problem occurs in many resource-allocation activities. A common example is where you have a mission, project, or assignment that requires a certain set of skills to complete. Given a set of available personnel with different sets of skills, you want to assemble a team with the necessary skills composed of the least number of personnel. It is probably acceptable to solve this instance of the problem by brute force, but it is probably not acceptable for other instances, such as when trying to assemble a balanced stock portfolio based on various stock properties instead of personnel skills.

After you've identified a problem as being intractable, you can set about developing an approximation algorithm that yields a reasonably accurate and useful solution. Figure 1 and Figure 2 implement an approximate solution to the set-covering problem, where the Enterprise computer selects an away party based on a set of skills required for a mission. Figure 1 defines an `Employee` class that associates a name with a set of skills, represented as strings. Figure 2 implements a greedy algorithm in the `setCover()` method to solve the problem. Greedy algorithms solve problems by making decisions that seem optimal at each stage of the algorithm. In this case, `setCover()` always selects the employee with the greatest number of skills that have yet to be filled by another employee. It does not guarantee a solution that uses a minimal number of personnel, but it does guarantee that all the required skills will be covered if possible. The running time of the algorithm is linear and proportional to the sum of the number of skills possessed by each employee.

Complexity theory and the analysis of algorithms is widely shunned—even by computer science students—which is why I've used the absolute minimum amount of math to convey the general ideas behind the need for approximation algorithms. A problem I've found with the presentation of approximation problems is that NP-complete problems are often presented in terms of graphs and vertices or other very abstract terms. It's easy to come away thinking you'll never use the stuff. But consider the development of personalization systems in e-commerce.[2] If a customer has indicated a set of interests and you want to make recommendations of products that cover those interests with some constraint, be it a minimal set of products or a set of products with a maximal price, you've run right into a variation of the set-covering problem. Once you become familiar with NP-complete problems, or a variety of problems solvable by polynomial-time algorithms, you see them pop up all over the place.

[2]E-commerce personalization was trendy in 2002; in 2012 an example relating to social media would be more relevant.

Figure 1. Employee Class

```
package example;

import java.util.Collections;
import java.util.Set;
import java.util.HashSet;

public class Employee {
  String name;
  Set<String> skills;

  public Employee(String name) {
    this.name = name;
    skills = new HashSet<String>();
  }

  public Employee addSkill(String skill) {
    skills.add(skill);
    return this;
  }

  public Set<String> getSkills() {
    return Collections.unmodifiableSet(skills);
  }

  public String toString() {
    StringBuffer buffer = new StringBuffer();

    buffer.append(name);
    buffer.append("\nSkills:\n");

    for(String skill : skills) {
      buffer.append("\t");
      buffer.append(skill);
      buffer.append("\n");
    }

    return buffer.toString();
  }
}
```

Figure 2. SetCover Class

```
package example;

import java.util.Collection;
import java.util.Set;
import java.util.HashSet;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.Arrays;

public final class SetCover {

  public static
  <T> Set<T> maximizingSubset(Set<T> set, Collection<Set<T>> subsets) {
    int maxSize = 0;
    Set<T> setCopy = new HashSet<T>();
    Set<T> maximizingSet = null;

    for(Set<T> subset : subsets) {
      setCopy.clear();
      setCopy.addAll(set);
      setCopy.retainAll(subset);

      if(setCopy.size() > maxSize) {
        maxSize = setCopy.size();
        maximizingSet = subset;
      }
    }

    return maximizingSet;
  }

  public static
  <T> Collection<Set<T>> setCover(Set<T> set, Collection<Set<T>> subsets) {
    Set<T> setCopy  = new HashSet<T>();
    Collection<Set<T>> subsetsCopy = new LinkedList<Set<T>>();
    Collection<Set<T>> setCover = new LinkedList<Set<T>>();

    setCopy.addAll(set);
    subsetsCopy.addAll(subsets);

    while(setCopy.size() > 0) {
      Set<T> max;
      if(subsetsCopy.size() <= 0) {
        // no set cover exists
        return null;
      }

      max = maximizingSubset(setCopy, subsetsCopy);

      if(max == null) {
        // no set cover exists
        return null;
      }

      subsetsCopy.remove(max);
      setCopy.removeAll(max);
      setCover.add(max);
    }

    return setCover;
  }

  public static
```

```
Set<Employee> awayParty(Set<String> skills, Set<Employee> employees)
{
  HashMap<Set<String>, Employee> map = new HashMap<Set<String>, Employee>();
  Collection<Set<String>> subsets = new LinkedList<Set<String>>();

  // Kluge to recover employees from their skill sets.
  for(Employee employee : employees) {
    Set<String> eskills = employee.getSkills();
    subsets.add(eskills);
    map.put(eskills, employee);
  }

  Collection<Set<String>> cover = setCover(skills, subsets);

  if(cover == null) {
    return null;
  }

  // Recover employees from their skill sets.
  Set<Employee> party = new HashSet<Employee>();

  for(Set<String> skillSet : cover) {
    party.add(map.get(skillSet));
  }

  return party;
}

public static void printParty(Set<Employee> party) {
  if(party == null) {
    System.out.println("Empty party.");
    return;
  }

  for(Employee employee : party) {
    System.out.println(employee.toString());
  }
}

// args[] is a sequence of skill names required for a mission.
public final static void main(String args[]) {
  Employee a, b, c, d, e;
  Set<String> requiredSkills = new HashSet<String>();
  Set<Employee> employees = new HashSet<Employee>();

  requiredSkills.addAll(Arrays.asList(args));

  a = new Employee("James T. Kirk")
    .addSkill("management").addSkill("tactics").addSkill("diplomacy");

  b = new Employee("Spock")
    .addSkill("logic").addSkill("science").addSkill("diplomacy");

  c = new Employee("Leonard H. McCoy")
    .addSkill("medicine").addSkill("science");

  d = new Employee("Montgomery Scott")
    .addSkill("engineering").addSkill("management");

  e = new Employee("New Ensign").addSkill("cannon fodder");

  employees.add(a);
  employees.add(b);
  employees.add(c);
  employees.add(d);
```

```
    employees.add(e);

    printParty(awayParty(requiredSkills, employees));
  }
}
```