Project 2: Greedy versus Exhaustive (50 points)

CPSC 335 - Algorithm Engineering
Fall 2021
Instructor: Doina Bein (dbein@fullerton.edu)
Last updated: Sun Aug 22 00:54:10 PDT 2021

Abstract

In this project you will implement and compare two algorithms that solve the same problem. For this problem, you will design two separate algorithms, describe the algorithms using clear pseudocode, analyze them mathematically, implement your algorithms in C++, measure their performance in running time, compare your experimental results with the efficiency class of your algorithms, and draw conclusions. The first is a greedy algorithm with a fast (i.e. polynomial) running time, while the second is an exhaustive ~~search~~ **optimization** algorithm with a slow (i.e. exponential) running time.

Both algorithms solve the problem of maximizing the number of minutes spent at the attractions at a county fair. More specifically, given a set of many different tickets for attractions available to purchase, these algorithms pick a subset of attractions that fit within a given budget while maximizing the number of minutes spent on the rides.
Sources: available at https://pxhere.com/en/photo/917877

Checking This Assignment Prompt For Updates

Assignment prompts are often living documents. Check back here regularly (and especially before your final submission), to make sure your submission complies with any changes or edits. You may check the top of the document, where the date of the last edit will be noted.

The Hypothesis

This experiment will test the following hypotheses:

1. Exhaustive search **and exhaustive optimization** algorithms are feasible to implement, and produce correct outputs.
2. Algorithms with exponential running times are extremely slow, probably too slow to be of practical use.

The Problem

Both algorithms will be used to solve an interesting problem. Suppose the following.

You are a trucking company responsible for transporting goods to a seaport, to be loaded in a container ship. Bringing containers to the port or from the port is called "drayage".

Tugboats guide a container ship at the Yangshan Deepwater Port on Oct. 4, 2019 in Shanghai, China. (Credit: Ji Haixin/VCG via Getty Images. Available at https://ktla.com/news/nationworld/global-trade-disrupted-as-companies-reduce-shipments-from-china-amid-coronavirus-outbreak/

A container is loaded and then brought to the port by a trucking company. A container is kept at the port in the container stacks until the designated ship arrives. Container ships, a type of cargo ship, have revolutionised the manner in which cargo supplies are ferried and transported across the world. By grouping cargo into containers, 1,000 to 3,000 cubic feet (28 to 85 m^3) of cargo, or up to about 64,000 pounds (29,000 kg), is moved at once and each container is secured to the ship once in a standardized way. The most common containers are 20 foot dry and 40 foot dry containers. Containerization has increased the efficiency of moving traditional break-bulk cargoes significantly, reducing shipping time by 84% and costs by 35%. Container ship capacity is measured in *twenty-foot equivalent units* (TEU). Today, about 90% of non-bulk cargo worldwide is transported by container, and modern container ships can carry over 23,000 TEU (e.g., MSC Gülsün). (https://en.wikipedia.org/wiki/Container_ship)

Your trucking company has been contracted to transport a list of goods, and you are tasked to use any of your available trucks to transport these goods to be loaded in a container ship. Each item has a weight and a volume. We are restricted by the total volume that can be loaded in the ship, while trying to maximize the total weight. One can transport any number of items and they add up as volume. For example, if you choose two different Soybean boxes of volumes +2 cubic meters and +3 cubic meters, you would be able to transport +5 cubic meters.

You are limited by the total volume that can be loaded on the container ship so you cannot transport everything you want. There are two algorithms to help you pick a subset of the available goods, so that you can maximize total weight for the ship while staying within the given volume.

A description of the problem at hand is as follows:

Container ship weight-maximization problem

input: A positive "volume limit" budget V (floating point of TEUs); and a vector G of n "goods" objects, containing one or more goods where each cargo item $a = (w, v)$ has floating point weight $w > 0$ and volume in cubic meters $v >= 0$

output: A vector K of goods drawn from G, such that the sum of volumes of goods from K is within the prescribed volume limit V and the sum of the goods' weight is maximized. In other words:

$\sum_{(w,vt)\in G} v \le V$;and the sum of all goods' weights $\sum_{(w,v)} w$ is maximized

Note that we require each good's volume $v > 0$ to be positive; we are omitting zero-volume goods that have been canceled.

The Algorithms

You must implement the following two algorithms for the *Container ship weight-maximization problem*. The first algorithm uses the greedy pattern. The greedy heuristic is to always choose the "best" (highest weight-per-volume) item that fits within the volume V, and keep selecting the best items until ran out of space (aka volume):

```
greedy_max_time(V, goods):
      todo = goods
      result = empty vector
      result_volume = 0
      while todo is not empty:
            Find the good "a" in todo of maximum weight per its volume
            Remove "a" from todo
            Let v be a's volume in TEUs
            if (result_volume + v) <= V:
                  result.add_back(a)
                  result_volume += v
      return result
```

The time complexity of the greedy algorithm depends on the data structures that are used to implement it. A naive approach using unsorted vectors and sequential search to find "a" takes $O(n^2)$ time. This is acceptable but not ideal. Using a heap, binary search tree, or sorting algorithm in a fairly straightforward way can speed this up to $O(n \log n)$.

The second algorithm uses a ~~proper exhaustive search~~ **exhaustive optimization**:

```
exhaustive_max_weight(V, goods):
      best = None
      for candidate in subsets(goods):
            if total_weight(candidate) <= V:
                  if best is None or
                     total_weight(candidate) > total_weight(best):
                        best = candidate
      return best
```

As discussed in section 7.5.4 of ADITA, `subsets(goods)` can be implemented using bitwise operations.

```
exhaustive_max_weight(V, goods):
```

```
n = |goods|
best = None
for bits from 0 to ($2^n$ -1):
      candidate = empty vector
      for j from 0 to n-1:
            if ((bits >> j) & 1) == 1:
                  candidate.add_back(goods[j])

      if total_volume(candidate) <= V:
            if best is None or
               total_weight(candidate) > total_weight(best):
                  best = candidate
return best
```

For this to work, the `bits` loop counter variable needs to be able to store the quantity $2^n - 1$. A good way of ensuring that is to use the largest integer data type in C++, which is the uint64_t type that is 64 bits wide. This creates a limitation that the exhaustive ~~search~~ **optimization** algorithm can only handle $n < 64$. This is unlikely to be a practical problem, because the time complexity of this algorithm is $O(2^n \cdot n)$.

Our theory predicts that the $O(2^n \cdot n)$ exhaustive ~~search~~ **optimization** algorithm will be far slower than the greedy algorithm with its $O(n^2)$ or $O(n \log n)$ time complexity. Your experiment will show whether this is the case.

Implementation

You are provided with the following files.

1. `goods.csv` contains over 8,000 goods.
2. `maxweight.hh` is a C++ header that defines skeleton functions for the two algorithms described above. In addition, there is a skeleton function `filter_cargo_vector` that filters down a large vector of goods into a smaller and more manageable set suitable for the exhaustive search algorithm. You are responsible for implementing these three functions (`filter_cargo_vector` and the two algorithms). `filter_cargo_vector` is intended to be a warmup to get you familiar with the `CargoItem` and `CargoVector` data types used throughout `maxweight.hh`. There are also some provided pre-written functions, for example to load and print a `CargoVector`.
3. `maxweight_scatterplot.cc` is a C++ program with a `main()` function that computes and stores execution times in `csv` files. Make changes to it to get your data, then plot it using Excel or similar. To compile it, use: `g++ -std=c++17 maxweight_scatterplot.cc`
4. `maxweight_test.cc` is a C++ program with a `main()` function that performs unit tests on the functions defined in `maxweight.hh` to see whether they work, prints out the outcome, and calculates

a score for the code. You can run this program to see whether your algorithm implementations are working correctly.

5. `rubrictest.hh` is the unit test library used for the test program; you can ignore this file.
6. `timer.hh` contains a small `Timer` class that implements a precise timer using the `std::chrono` library in C++17. `timer.hh` may be used together with `filter_cargo_vector`, and other helper functions to run your algorithm implementations against various lengths of the ride database, in order to gather empirical data for your report.
7. `README.md` contains a brief description of the project,and a place to write the names and CSUF email addresses of the group members. You need to modify this file to identify your group members.

Obtaining and Submitting Code

This document explains how to obtain and submit your work:

GitHub Education Instructions

Invitation links for the project:

https://classroom.github.com/g/csZ68df4

What to Do

First, add your group member names to `README.md`. Implement all the skeleton functions in the provided header file. Use the test program to check whether your code works.

When you go to commit files in git, make sure you are not checking any generated executable/object files into git. Use the *.gitignore* file to accomplish this.

Once you are confident that your algorithm implementations are correct, do the following:

1. Analyze your greedy algorithm code mathematically to determine its big-O efficiency class, probably $O(n^2)$ or $O(n \log n)$.
2. Analyze your exhaustive optimization algorithm code mathematically to determine its big-O efficiency class, probably $O(2^n \cdot n)$.
3. Gather empirical timing data by running your implementations for various values of n.
4. Draw a scatter plot for each algorithm and fit line for your timing data. The instance size $n$should be on the horizontal axis and elapsed time should be on the vertical axis. Each plot should have a title; and each axis should have a label and units of measure.
5. Conclude whether or not your empirically-observed time efficiency data is consistent, or inconsistent, with your mathematically-derived big-O efficiency class for each algorithm.

Finally, produce a brief written project report ***in PDF format***. Your report should be submitted as a checked-in file in GitHub. Your report should include the following:

1. Your names, CSUF-supplied email address(es), and an indication that the submission is for project 2.
2. Two scatter plots meeting the requirements stated above.
3. Answers to the following questions, using complete sentences.
a. Is there a noticeable difference in the performance of the two algorithms? Which is faster, and by how much? Does this surprise you?
b. Are your empirical analyses consistent with your mathematical analyses? Justify your answer.
c. Is this evidence consistent or inconsistent with hypothesis 1? Justify your answer.
d. Is this evidence consistent or inconsistent with hypothesis 2? Justify your answer.

Grading Rubric

Your grade will consist of three parts: *Form, Function,* and *Analysis.*

Function refers to whether your code works properly as defined by the test program. We will use the score reported by the test program as your Function grade.

Form refers to the design, organization, and presentation of your code. A grader will read your code and evaluate these aspects of your submission.

Analysis refers to the correctness of your mathematical and empirical analyses, scatter plots, question answers, and the presentation of your report document.

The grading rubric is below.

1. Function - 16 points:
    a. `load_cargo_database` passes all tests: 2 points (this function is given)
    b. `filter_cargo_vector` passes all tests: 2 points
    c. `greedy_max_weight` trivial tests: 2 points
    d. `greedy_max_weight` passes all additional tests: 4 points
    e. `exhaustive_max_weight` trivial tests: 2 points
    f. `exhaustive_max_weight` passes all additional tests: 4 points
2. Form - 12 points
    a. README.md complete: 3 points
    b. Style (whitespace, variable names, comments, etc.): 3 points
    c. Design (where appropriate, uses encapsulation, helper functions, data structures, etc.): 3 points
    d. Craftsmanship (no memory leaks, gross inefficiency, taboo coding practices, etc.): 3 points
3. Analysis - 22 points
    a. Report document presentation: 6 points
    b. Two scatter plots: 6 points each plot
    c. Question answers: 4 points

Deadline

The project deadline is Friday October 15, 11:59 pm. You will be graded based on what you have pushed to GitHub as of the deadline. Commits made after the deadline will not be considered. No late submissions.