CPSC 335
Spring 2014
Project #2 — exhaustive search

Introduction

In this project you will design, implement, and analyze exhaustive search algorithms for the *traveling salesperson* and *longest common subsequence* problems.

The hypotheses

This experiment will test two hypotheses:

1. Exhaustive search and optimization algorithms produce correct outputs.
2. Algorithms with exponential (e.g. $O(2^n)$) and factorial (e.g. $O(n!)$) running times are extremely slow, probably too slow to be of practical use.

Traveling salesperson problem (TSP)

Section 4.16 of the lecture notes describe the TSP and an exhaustive search algorithm that solves it. The TSP's Wikipedia page contains some additional examples and background material.

Requirements

You are to:

1. Reuse the `tsp` algorithm given in Section 4.16, or design your own exhaustive search algorithm for solving TSP with exhaustive search. If you design your own algorithm you might start with `tsp` and modify it to generate permutations lazily with a factory. Or you might try to design a different algorithm whose running time is exponential in the number of edges m of the input graph, as described in question C-4.6.
2. Implement your algorithm as a Python 3 program. Your program should
   (a) Load a TSP instance.
   (b) Solve it, while measuring elapsed time.
   (c) Print out the elapsed time, sequence of vertices in an optimal solution, and total cost of an optimal solution.
3. Run your implementation on instances taken from TSPLIB (see below), and include its output in your project report (see below).
4. Collect timing data and analyze your algorithm empirically as we did in Project 1.

TSPLIB

TSPLIB is a corpus of TSP instances curated by the University of Heidelberg. The instances are available in various formats; we will only use their XML format. The XML instances are available at

```
http://www.iwr.uni-heidelberg.de/groups/comopt/
    software/TSPLIB95/XML-TSPLIB/instances/.
```

The number of vertices n is part of each instance's file name, so for example `br17.xml.zip` has $n = 17$ vertices and `att48.xml.zip` has $n = 48$ vertices.

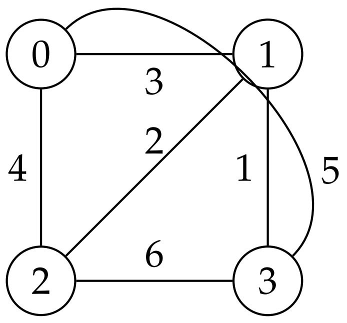
The total weight of optimal solutions for these instances are known and displayed at `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html`.

tsplib.py

I have provided a Python 3 module, `tsplib.py`, that can load a TSPLIB XML format file into a Python object. The usage of that module is documented in its comments.

Instances to use

I created a TSPLIB-format file corresponding to the 4-element graph shown in an Example in the exhaustive search lecture notes. It is in the provided `clique4.xml.zip`.



Run your algorithm on the following TSPLIB instances.

1. `clique4.xml.zip` (provided by me, not TSPLIB)
2. `burma14.xml.zip`
3. `br17.xml.zip`
4. The largest instance you can find that your program can complete in 5 minutes.

Since your algorithm probably runs in exponential time it may take a very long time to solve some of those instances. *You can stop waiting for your code to finish after waiting at least 5 minutes.*

Longest common subsequence (LCS)

The second problem that you will tackle is the longest common subsequence problem. This problem deals with finding *subsequences,* which are not the same thing as *substrings.*

Definition 1. *B is a* subsequence *of sequence U, written*

$$B \subseteq_S S,$$

when B may be transformed to U by inserting, but not reordering, elements into B.

All of the following are valid subsequences:

$$\begin{aligned}
\langle c,a,t\rangle &\subseteq_S \langle c,a,t,s\rangle \\
\langle c,a,t\rangle &\subseteq_S \langle c,a,r,t\rangle \\
\langle a,c,t\rangle &\subseteq_S \langle a,c,h,t,u,n,g\rangle \\
\langle y\rangle &\subseteq_S \langle z,e,p,h,y,r\rangle \\
\langle x\rangle &\subseteq_S \langle x\rangle \\
\langle\rangle &\subseteq_S \langle x\rangle
\end{aligned}$$

However

$$\langle a,c,t\rangle \not\subseteq_S \langle t,a,c,k\rangle$$

because $\langle a,c,t\rangle$ can only be transformed into $\langle t,a,c,k\rangle$ by reordering t.

The *longest common subsequence (LCS)* problem is:

input: two strings L, R of length n_L, n_R respectively
output: the longest string B such that B is a subsequence of L and also a subsequence of R

size: n_L, n_R

Example: The longest common subsequence of

$$L = \langle a, b, a, z, d, c \rangle$$

and

$$R = \langle b, a, c, b, a, d \rangle$$

is[1]

$$B = \langle a, b, a, d \rangle.$$

Requirements

We did not cover an algorithm for this problem in class. Designing an exhaustive search algorithm for the longest common subsequence problem is part of the assignment.

You are to:

1. Design and analyze an exhaustive search algorithm for the longest common subsequence problem. (I am aware that there is a dynamic programming algorithm for this problem. Do not implement that algorithm; this assignment is about exhaustive search algorithms. We will study this dynamic programming algorithm later in the course.)
2. Implement your algorithm in Python 3. Your program should
   (a) Implement a longest common subsequence algorithm that takes two Python string objects as input and returns a Python string object.
   (b) Be able to load, for a given input size n, the first n characters of the text file `pg11.txt` as L, and the first n characters of `pg76.txt` as R. Remember that the project 1 template used the Python statement

```
L = open('pg11.txt').read()[:n]
```

[1]Example from `http://www.cc.gatech.edu/~ninamf/Algos11/lectures/lect0311.pdf`.

to do this sort of thing.

   (c) Compute the longest common subsequence of L and R, while measuring elapsed time.
   (d) Print out the elapsed time, output B, and length of B.

3. Collect timing data and analyze your algorithm empirically.

Instances to use

Run your algorithm on the following LCS instances.

1. The strings L=″abazdc″ and R=″bacbad″ with $n_L = 6$ and $n_R = 6$ (the example above).
2. The strings L=″abracadabra″ and R=″yabbadabbadoo″ with $n_L = 11$ and $n_R = 13$. [2]
3. The first $n = 10$ characters of `pg11.txt` and `pg76.txt`.
4. The first $n = 15$ characters of `pg11.txt` and `pg76.txt`.
5. The first $n = 20$ characters of `pg11.txt` and `pg76.txt`.
6. The largest value of n such that your program can complete in 5 minutes.

Again, you can terminate any run of your program that takes longer than 5 minutes.

Algorithm design hints

An output B is a subsequence of L and R. So each candidate solution in your exhaustive search algorithm should be some kind of subsequence. We have not covered a generator algorithm for subsequences. However, our subset generator can be used to generate subsequences. Generate all

[2] Example from `http://www.cs.umd.edu/~meesh/351/mount/lectures/lect25-longest-common-subseq.pdf`.

subsets of the *indices* of L (or R, your choice). Then you can form a subsequence from each set of indices. Very high-level pseudocode:

```
for indices in subsets(all valid indices of L):
  B = empty string
  for each valid index i of L:
    if i is in indices:
      B.append(L[i])
```

Candidate generation algorithms

I have provided a Python module `candidate.py` that include implementations of

- the eager subset generation algorithm we covered,
- the eager permutation generation algorithm,
- the lazy subset generation algorithm we covered, and
- a lazy permutation algorithm called the Steinhaus-Johnson-Trotter algorithm, which we did not cover.

The Steinhaus-Johnson-Trotter algorithm is described in Section 4.3 of the textbook by Levitin, and has a Wikipedia page.

Also, the itertools module in Python includes implementations of

- a lazy permutation generator called `permutations`, and
- an example shows how to implement a `powerset` function in one line using other functions in `itertools`.

You are free to use any of these to help generate your candidate solution objects, or to implement your own algorithms from scratch.

Sample output

The output of a correct TSP algorithm for the `clique4` instance:

```
TSP instance: clique4
  n = 4
  elapsed time = 0.001274119999834511
  optimal cycle = [0, 3, 1, 2, 0]
  optimal cost = 12.0
```

The output of a correct LCS algorithm for the first instance specified above:

```
LCS:
  a = "abazdc", length 6
  b = "bacbad", length 6
  elapsed time = 0.0007212339996840456
  longest common subsequence = "abad", length 4
```

Deliverables

Produce a written project report. Your report should include the following:

1. Your name(s), CWID(s), and an indication that the submission is for project 2.
2. Two scatter plots showing the elapsed time of
   (a) your TSP implementation, with the instance size n on the horizontal axis; and
   (b) your LCS implementation, with the instance size $\max(n_L, n_R)$ on the horizontal axis.
3. Your pseudocode for both algorithms.
4. Output from your programs, for all the instances you were able to solve.
5. Your complete Python source code.
6. Answers to the following questions, using complete sentences.

(a) Describe you TSP algorithm briefly. Did you use the eager exhaustive search algorithm we covered in class, modify that algorithm slightly, or modify it extensively? If you made changes, what were they and why did you make them? What is the time complexity of your algorithm?

(b) What is the largest TSP instance your implementation could solve, and how long did that take?

(c) Describe your LCS algorithm briefly. How did you generate candidate solutions? Does your algorithm generate them eagerly or lazily? What is the time complexity of your algorithm?

(d) What is the largest TSP instance your implementation could solve, and how long did that take?

(e) Which candidate generation code did you use: the provided eager generators, provided factory classes, or the generators built into `itertools`? Why did you choose that implementation over the alternatives?

(f) Did your implementations produce correct outputs? Is this evidence *consistent* or *inconsistent* with hypothesis 1?

(g) How would you characterize the efficiency of your implementations? Was it possible to solve realistically-sized inputs in a reasonable amount of time? Is this evidence *consistent* or *inconsistent* with hypothesis 2?

Your document *must* be uploaded to Titanium as a single PDF file.

Due Date

The project deadline is Friday, 3/21, 11:55 pm. Late submissions will not be accepted.