﻿Department of Mathematics & Computer Science, Whitworth University

Algorithm Design & Analysis CS 473

Fall 2019

What’s been taken vs. what is still available

**Candidate Project ideas/topics/problems:**

1. Strassen’s Algorithm (divide-n-conquer algorithm for matrix-matrix multiplication)

* Input: two square matrices of size n x n with 4 <= n <= 256
* It’s fine to consider either integer or floating point matrix entries
* If n is not a power of 2, “pad with zeros” as briefly discussed in class
* Implement Strassen as well as regular (“high school”) matrix multiplication for square matrices of size up to n = 256
* Compare performance (who runs faster) as a function of size n: does Strassen’s speed overtake regular matrix multiplication as n grows? Run on the same hardware, plot speeds as a function of n, and discuss briefly what you found when it comes to efficiency/relative speeds of the two algorithms

1. Fast Fourier Transform (FFT) Algorithm (also divide-n-conquer based, essentially for a particular type of matrix-by-vector multiplication)

* Input: an array of integers or reals/floats, of size n (where n is a power of 2)
* Output: the Discrete Fourier Transform (DFT) of that array (mathematically, it’s a vector to which you apply a linear transformation as specified by the DFT matrix of size nxn; computationally, you’re multiplying an array or column-matrix by a square matrix)
* How you do it: by applying a divide-n-conquer, O(n log n) matrix-vector multiplication (n x n DFT matrix multiplying an array represented as a n x 1 column-matrix)
* Why is it called “fast”: because it beats the “normal”, linear algebra matrix-vector multiplication (which, for sizes of n x n and n x 1, would take how long?)

1. Playing Generalized (game of) 3-Heap Nim **\*\*\* Taken**

* Input: four positive integers, n, m1, m2, m3: where mi  stands for the size (number of marbles) in each of the three piles, i = 1, 2, 3; and n as before is the max. # of marbles each player can remove at each move (that is, you have to remove at least 1 marble, and no more than n marbles)
* See slide deck / read Levitin for the single-pile version of Nim; and read Wikipedia / do a bit of online search for how Nim generalizes to more piles in general; and exactly 3 piles, in particular
* Instead of a single “pile” of marbles or peanuts (like in the textbook and the lectures), in this project you have three such piles.
* You should implement the game, prompt the user to play either first or second, and then play optimally as “the other player” (that is, you make it an interactive game where human user plays against the computer)
* Simple GUI is fine; your Nim simulator should do moves one-at-a-time like in a real game: the user first chooses whether to go first or second, and then the players (user vs. your program) alternate moves
* So, after each move by the computer, you prompt the user for their next move, until someone picks the last marble/wins the game
* Your program should report who won the game, as well as (based on values of m and n) **who \*should have won\***, had both players played optimally (and of course, your program / “the computer” should play optimally!)

1. Tic-Tac-Toe **\*\*\* Taken**

* You are to implement, from the scratch, the classic kids’ game of Tic-Tac-Toe (3x3 board, two players alternate placing ‘X’ and ‘O’ in one of the empty squares, whoever places three of the same kind in a row be it a vertical, horizontal or diagonal row, wins! Otherwise, the game is a tie)
* Simple GUI is fine; your implementation should be so that 3rd graders can use it to “play against the computer” and have fun doing it
* Your program should prompt the human user whether (s)he wants to go first or second; your program then plays against the human
* The program should play optimally, and in particular never lose; and it should punish errors by the human opponent (the TA or I can easily test you on this aspect, obviously! ☺ )

1. Knapsack

* For the version of Knapsack problem originally defined in Ch. 3 (brute-force algorithms) and then revisited in Chapter on Dynamic Programming (yet to be covered), implement three algorithms: a brute-force (per Ch. 3), dynamic programming (read Ch. 8), and a third algorithms of your choice
* The 3rd algorithm can either be your original idea, or an implementation of an algo. found online/in the literature (if not original, you need to properly attribute where you found it / whose algorithm you are implementing). It needs to be “smarter” than brute-force, but not based on DP as found in Ch. 8 of Levitin.
* Your input should be positive integer set of weights w[i], set of values of those items v[i], and overall knapsack capacity W. It’s fine to bound all w[i]’s and W to be (say) <= 100. The # of items should be in the range from 1 to 20.
* Should any of your algorithms run for too long on a certain input, you should define a ‘deadline’ so your algorithm halts and outputs the best feasible solution found so far (which obviously need not be optimal).
* You should do a comparative analysis of the three algorithms and write a brief report summarizing your findings.

1. Planar version of TSP

* Traveling Salesman Problem (TSP) is notoriously difficult in general, even to solve approximately (as it is an optimization problem); however, in some restricted cases, it can be solved efficiently at least insofar as good approximate solutions (meaning, finding a tour that’s guaranteed to be close to optimal – you need to do some background reading to make a good sense of this problem!)
* Your input: an integer matrix (non-negative entries only) representing a weighted undirected graphs; the number of nodes n should be a parameter between 2 and 50
* Your algorithm is to first check whether this graph is planar; if it is not, you should report “Graph not planar, TSP on it too hard to solve!” or similar and stop. (You will need to do some literature review to figure out, how to determine if a graph is planar.)
* If the input graph is planar, you need to implement two different approximation algorithms/heuristics that find an optimal, or close-to-optimal, TSP tour. You can do an online search and you should appropriately attribute which/whose algorithms you are implementing (you are also welcome to try to devise an original one on your own – however, it is challenging to come up with any novel, reasonably “smart” heuristic or algorithm for TSP even in restricted planar case!)
* You then need to do a comparative analysis of relative performances (both speeds and how good approximations they find) for the two approximation algorithms you have implemented.

1. (Minimum Spanning Trees) Prim’s vs. Kruskal’s Algorithm: Comparison & Contrast

* Read about Greedy Algorithms in Ch. 9 and find some online resources on finding MSTs (“Minimum Spanning Trees”) in weighted undirected graphs
* Given an integer matrix representing a weighted undirected CONNECTED graph, implement both Prim’s and Kruskal’s algorithms and compare their performances (cf. w.r.t. relative speeds). Assume all edge weights / adjacency matrix entries to be nonnegative. Your graphs should have up to n = 1000 nodes.
* Make sure your input graph is connected! Do some online research on how to assure that a randomly (or not-so-randomly) generated graph of a nontrivial size is actually connected.
* Write a report comparing performances on Prim’s and Kruskal’s on various graphs with up to n = 1,000 nodes. Is either algorithm consistently better than the other? If not, based on your simulations, which of the two would you prefer in various scenarios (depending on input graph’s size, density, and/or other properties)?

1. Closest Pairs of Points **\*\*\* Taken**

* Implement Divide-and-Conquer algorithm for finding, among N points in x-y Cartesian plane, the two points that are closest to each other. (This problem was briefly mentioned in the lectures but not covered in the context of divide-and-conquer; there is a fairly detailed description of it in A. Levitin,)
* Also implement brute-force algorithm for the same closest pair problem, as described in Ch. 3.
* Compare-and-contrast the two algorithms for various randomly distributed sets of N points, where N varies from 3 to 500. Plot speeds of the two algorithms as a function of N. For each N you choose (say, N = 10, 20, 30, … ) generate five different distributions of N points, and include in your plot and analysis the average run time across those five runs.
* Write a brief report summarizing the results. Is the divide-and-conquer algorithm as superior as predicted by theory?

1. Network Flow Problem

* Read *Chapter 10* in Levitin on network flows, and in particular make sure you understand what Max-Flow-Min-Cut Theorem says.
* Implement two classical network flow algorithms (whose detailed descriptions and pseudo-codes can easily be found online): Ford-Fulkerson and Edmunds-Karp
* Run those algorithms on network flow problems defined on capacity networks (weighted graphs with edge capacities) of up to n = 200 nodes. Assume all edge capacities to be non-negative integers. Feel free to use a network generator you found online, as long as you properly attribute what you are using.
* Do a comparative analysis between these two classical network flow algorithms. Which one performs better for larger networks? Are your simulation results consistent with theoretical predictions? What is your explanation for your findings?

1. Computational Geometry – Computing Convex Hulls **\*\*\* Taken**

* Read Levitin (and look up some online sources, as appropriate) on classic computational geometry problem, that of finding a convex hall of a set of points in Euclidean plane.
* We skipped this in the lectures; however, Levitin covers it / convex hull algorithms do illustrate some of the techniques we have covered this semester.
* Implement, compare and contrast two different algorithmic approaches to finding a convex hull of a set of N points in the plane. Include several test-cases (examples) and how each of your two algorithms finds the convex hull of those points; your examples should have between 10 and 100 points.
* Do a careful analysis of the computational efficiency of each of your two implementations, including generalizing to what would be the asymptotic run-times as function of N, the number of points (as N grows large).

1. String Matching Algorithms **\*\*\* Taken**

* Implement, compare and contrast at least two, and ideally three or more, of the following string-matching algorithms: 1) Rabin-Karp, 2) Knuth-Morris-Pratt, 3) Boyer-Moore, 4) Two-way algorithm and 5) the Horspool algorithm.
* You should submit both several test cases of running your implementations on concrete strings of non-trivial sizes, and an asymptotic complexity analysis of both run-time and space (i.e., memory) requirements of your algorithms as a function of pattern length *n* and text length *m*. Your analysis should include a brief discussion of time-space tradeoffs.
* You should gain the necessary background on string matching by reading *Chapter 7* in A. Levitin, esp. the sections on Horspool and Boyer-Moore algorithms; and then expand your understanding of string matching, by doing some online research to familiarize yourself with other popular string-matching algorithms. (Note: in the lectures, we will skip most of Ch. 7.)

12. **Shortest Paths** – comparison and analysis of at least three shortest path algorithms (Dijkstra, Bellman-Ford, and the third one is up to you!) **\*\*\* Taken**

* Dijsktra’s algorithm is a computer science “classic” that everyone with a CS degree should know, and hence is a must to be included in this comparison-n-contrast of 3 (or more) algorithms for shortest paths in weighted graphs
* Bellman-Ford’s algorithm is very important to understand, as well; and it can handle certain types of weighted graphs that Dijkstra’s algorithm has trouble with (what kind of weighted graphs are those? Explain briefly in your project report!)
* The 3rd algorithm (and beyond if you are ambitious and want to compare-n-contrast more than three shortest path algo’s) is up to you; one possible choice is Floyd-Warshall’s algorithm, which is a paradigmatic Dynamic Programming technique, and hence interesting to compare vs. greedy algorithms such as Dijkstra’s on the same problem type (in this case, shortest paths in weighted graphs)

Unconventional Techniques

Use an interesting technique not covered in the class, to solve an interesting optimization, search or other type of combinatorial/mathematical/computational problem. Examples of interesting techniques that are seldom or never covered in “standard” undergraduate (or even graduate-level) courses on Algorithms include evolutionary programming (EP), genetic algorithms (GA), particle swarm optimization (PSO), ant colony optimization, and so on.

The same rules apply to these projects using techniques not studied in the class, with one caveat: if, say, two teams each propose a project using *Genetic Algorithms* but they work on considerably different problems using the same technique (GA in this case), that’s allowed, i.e., those would still be two sufficiently different projects. Therefore, it is very important, should you choose a “non-traditional” technique such as those listed in the previous paragraph (EP, GA, PSO, etc.) , that you discuss with the Instructor, what specific problem(s) you want to address or solve using that technique. In particular, just saying, “My team will do a project using GA” or “We will do a project with PSO” is similar to saying, “My team will do a project with dynamic programming” or “We will do divide-n-conquer algorithm(s) and/or application(s)”: it is way too generic, and you need to scope it down and come up with (and discuss with, get an approval from me, the instructor) a reasonably specific problem or problems that you would attempt to solve with your “exotic” technique of choice.