CS 4349.400 Final Exam—Problems and Instructions

December 12, 2018

Please read the following instructions carefully before you begin.

- Write your name and Net ID on the *answer sheets* cover page and your Net ID on each additional page. Answer each of the six questions on the answer sheets provided.
- Questions are not necessary given in order of difficulty, so read through them all before you begin writing!
- You're allowed to bring in one 8.5" by 11" piece of paper with notes written or printed on front and back.
- You have two hours and 45 minutes to take the exam.
- Please turn in these problem sheets, your answer sheets, scratch paper, and notes at the end of the exam period.
- Writing "I don't know" ***and nothing else*** for any question or lettered part of a question is worth 25% credit. If you leave the solution blank or write anything else, we will grade exactly what is written.
- If asked to describe an algorithm, you should state your algorithm clearly and briefly explain its asymptotic running time in big-O notation in terms of the input size. ***You do not have to justify (prove) correctness of the algorithm.***
- Feel free to ask for clarification on any of the problems.
- You can do this.

1. Most graphics hardware includes support for a low-level operation called *blit*, or **bl**ock transfer, which quickly copies a rectangular chunk of a pixel map (a two-dimensional array of pixel values) from one location to another.

   Suppose we want to rotate an $n \times n$ pixel map 90° clockwise. One way to do this is to split the pixel map into four $n/2 \times n/2$ blocks, move each block to its proper position using a sequence of five blits, and then recursively rotate each block.

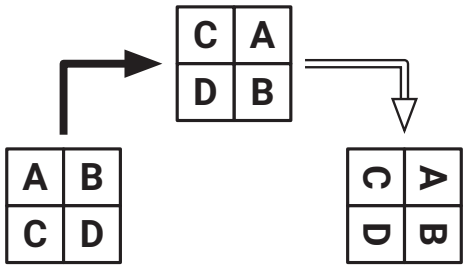


Figure 1. Rotating a pixel maps using blits and recursion.

   (a) **(5 out of 10)** Consider the partially defined procedure ROTATE($X[1 .. n, 1 .. n]$) which takes an $n \times n$ pixel map X and rotates it 90°. For simplicity, we assume n is a power of 2.

```
ROTATE(X[1 .. n, 1 .. n]):
  if n ≥ 2
      ⟨⟨Move blocks to final positions⟩⟩
      blit X[1 .. n/2, 1 .. n/2] to temp[1 .. n/2, 1 .. n/2]
      _____
      _____
      _____
      _____

      ⟨⟨Recursively rotate blocks⟩⟩
      ROTATE(X[1 .. n/2, 1 .. n/2])
      _____
      _____
      _____
```

   The procedure is missing four blits and three recursive calls. Fill in the missing lines to fully define the procedure.

   (b) **(3 out of 10)** Suppose a $k \times k$ blit takes $O(k^2)$ time. We can express the asymptotic running time of ROTATE($X[1 .. n, 1 .. n]$) using the recurrence

$$T(n) = 4T(n/2) + n^2.$$

   State the running time of ROTATE($X[1 .. n, 1 .. n]$) using big-O notation by solving the recurrence.

   (c) **(2 out of 10)** Let $B(n)$ be the number of blits used to rotate an image by calling ROTATE($X[1 .. n, 1 .. n]$). Give a recurrence definition for $B(n)$ including the base case. The recurrence should yield the ***exact*** number of blits used. You do not need to solve this recurrence.

2. Recall, a *subsequence* of a sequence A consists of a (not necessarily contiguous) collection of elements of A, arranged in the same order as they appear in A. If B is a subsequence of A, then A is a *supersequence* of B.

   In Homework 3, you were asked to design a simple recursive algorithm to compute, given two sequences $A[1..m]$ and $B[1..n]$, the length of the *shortest common supersequence* of A and B. For example, given the strings `ALGORITHM` and `ALTRUISTIC`, the algorithm would return 14, the length of the shortest common supersequence **ALGTORUISTHIMC**.

   Today, we'll design a faster algorithm using dynamic programming. Let $SCS(i, j)$ be the length of the shortest common supersequence between $A[1 .. i]$ and $B[1 .. j]$. We can recursively define $SCS(i, j)$ as follows:

$$SCS(i,j) = \begin{cases} j & \text{if } i = 0 \\ i & \text{if } i > 0 \text{ and } j = 0 \\ 1 + \min\{SCS(i-1,j), SCS(i,j-1)\} & \text{if } i,j > 0 \text{ and } A[i] \neq B[j] \\ 1 + SCS(i-1,j-1) & \text{otherwise} \end{cases}$$

   (a) **(2 out of 10)** In what kind of ***memoization data structure*** should we store the solutions to all subproblems $SCS(i, j)$? If you're using a (multidimensional) array, be sure to state the indices we use. For example, the input to our algorithm is two arrays $A[1 .. m]$ and $B[1 .. n]$.

   (b) **(2 out of 10)** What is a good ***evaluation order*** for solving the subproblems so each subproblem is solved after the ones it is dependent upon?

   (c) **(2 out of 10)** What will be the final ***space*** and ***time*** complexity of the dynamic programming algorithm? Give your solutions in terms of both *m and n*.

   (d) **(4 out of 10)** Write the iterative algorithm that computes the length of the shortest common supersequence between $A[1 .. m]$ and $B[1 .. n]$.

3. Let X be a set of intervals on the real line. Note that some intervals may have identical endpoints. A subset of intervals $Y \subseteq X$ is called a ***tiling cover*** if the intervals in Y cover the intervals in X, that is, any point that is contained in some intervals in X is also contained in some interval in Y. The ***size*** of a tiling cover is just the number of intervals in the cover.

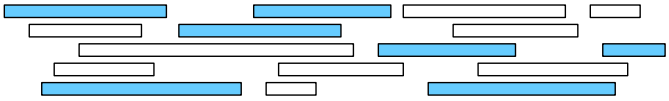


Figure 2. A set of intervals. The seven shaded intervals form a tiling cover.

We want to compute a smallest tiling cover of X as quickly as possible using a greedy algorithm.

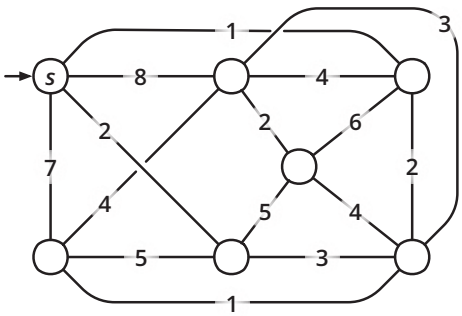
(a) **(4 out of 10)** Give a very small counterexample showing the following strategy *does not* lead to a smallest tiling cover: Take the *longest* interval x, remove any points in x from each of the other intervals (i.e., each interval y becomes $y \setminus x$), and recurse.

(b) **(4 out of 10)** The following strategy *does* lead to a smallest tiling cover: Let p be the leftmost point in any interval, and let x^* be the longest interval *starting at* p. Take interval x^*, remove any points in x^* from each of the other intervals, and recurse.

We want to do an exchange argument to show this strategy works. Suppose some smallest tiling cover Y does not contain interval x^*. Describe an interval y we can safely remove from Y and replace with x^* so that $Y - y + x^*$ is still a smallest tiling cover. Briefly describe why your choice is correct.

(c) **(2 out of 10)** Now suppose each interval $x \in X$ has a non-negative weight $w(x)$ assigned to it. Give a very small counterexample showing the strategy from part (b) *does not* find a tiling cover of *minimum total weight*.

4. Consider the weighted graph pictured below.



(a) **(2.5 out of 10)** Draw a depth-first spanning tree rooted at s.

(b) **(2.5 out of 10)** Draw a breath-first spanning tree rooted at s.

(c) **(2.5 out of 10)** Draw a shortest-path tree rooted at s.

(d) **(2.5 out of 10)** Draw a minimum spanning tree.

Some of these subproblems may have more than one correct answer.

5. A ***polygonal path*** is a sequence of line segments joined end-to-end; the endpoints of these line segments are called the ***vertices*** of the path. The ***length*** of a polygonal path is the sum of the lengths of its segments. A polygonal path with vertices $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$ is ***monotonically increasing*** if $x_i < x_{i+1}$ and $y_i < y_{i+1}$ for every index i—informally, each vertex of the path is above and to the right of its predecessor.

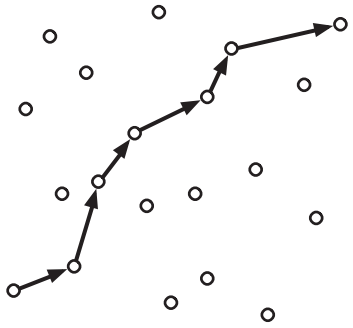


Figure 3. A monotonically increasing polygonal path with seven vertices through a set of points.

Suppose we are given a set S of n points in the plane, represented as two arrays $X[1 .. n]$ and $Y[1 .. n]$ and a subroutine $\textsc{Length}(x, y, x', y')$ that returns the length of the segment from (x, y) to (x', y'). Our goal is to compute the length of the maximum-length monotonically increasing path with vertices in S that begins at a given point (x_s, y_s) and ends at a given point (x_t, y_t). To do so, we'll perform a reduction to ***single source shortest paths*** in a DAG. We need to begin by constructing a directed acyclic graph G.

(a) **(2 out of 10)** What should we use for the vertices of G? Which vertex should be used for s? *[Hint: Read the question again.]*

(b) **(2 out of 10)** What should we use for the edges of G? Be sure to describe in which direction they are oriented. *[Hint: A path in G contains a subset of its edges.]*

(c) **(2 out of 10)** Briefly explain why G is a DAG.

(d) **(2 out of 10)** What weights should we assign to each edge? *[Hint: A longest monotonically increasing path with vertices in S needs to correspond to a shortest path in G.]*

(e) **(2 out of 10)** In Homework 7, we saw how single source shortest paths in a DAG can be computed in $O(V + E)$ time using dynamic programming. ***In terms of n***, how long does it take to construct G and find single source shortest paths using this $O(V + E)$ time subroutine?

6. Both parts ask you to design an algorithm for different problems. Neither part depends upon the other.

   (a) **(4 out of 10)** Let $G = (V, E)$ be an arbitrary directed graph with non-negative capacities $c : E \to \mathbb{R}_{\geq 0}$ on the edges and two special vertices s and t. Suppose we assign a non-negative ***limit*** $\ell : V \setminus \{s, t\} \to \mathbb{R}_{\geq 0}$ for the amount of flow that can pass through each vertex other than s or t. Formally, a flow $f : E \to \mathbb{R}_{\geq 0}$ is feasible *with respect to both c and ℓ* if for all edges $e \in E$ we have $f(e) \leq c(e)$ and for all vertices $v \in V \setminus \{s, t\}$ we have $\sum_u f(u \to v) \leq \ell(v)$.

   Describe and analyze an algorithm to compute a graph $G' = (V', E')$ with non-negative edge capacities $c' : E' \to \mathbb{R}_{\geq 0}$ *but no vertex limits* so that the value of the maximum feasible flow in G' with respect to c' is equal to the value of the maximum feasible flow in G with respect to both c and ℓ.

   (b) **(6 out of 10)** Suppose you are taking a particularly intense class in the computer science department that requires a large time investment to complete the homework assignments (no comment on what class that might be). You know your own ability to do the assignemnts very well. For each integer k, you'll earn $Score[k]$ points for doing homework k. Unfortunately, completing homework k means you'll be behind in your other classes, forcing you to skip the next *two* assignments to catch up on your other work (in other words, you cannot do assignments $k + 1$ or $k + 2$ if you do assignment k).

   Describe and analyze a dynamic programming algorithm to compute the maximum total score you can achieve doing homework for this class. The input to your algorithm is the array $Score[1 .. n]$. You'll receive full credit for describing a recursive solution, giving a suitable evaluation order for solving the subproblems, and stating the total time needed to evaluate those subproblems.