Homework 1: DFS, Paths and Greedy

Handed out Thu, Feb 6. **Due at the start of class (11am), Thu, Feb 20**. (Submission through Gradescope.) Solutions will be discussed in class, so no late homeworks will be accepted. In other words, turn in whatever you have finished by the due date. Total point value 50.

Problem 1. (12 points) In this problem you will simulate the strong-components algorithm given in class on the digraph shown in Fig. 1.

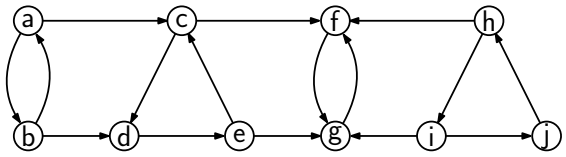


Figure 1: Computing strong components.

(a) (2 points) Draw the reverse graph G^R. (Be careful to note the directions of the edges.)

(b) (4 points) Show the result of applying DFS to G^R. Label each vertex u with its discovery and finish times ($d[u]/f[u]$). You need only show the *final* DFS tree, not the intermediate results.

To make the grader's life easier please draw your DFS forest in the same manner as in Fig. 5 of Lecture 3. Whenever you have a choice of which vertex to visit next, select the lowest vertex in alphabetic order.

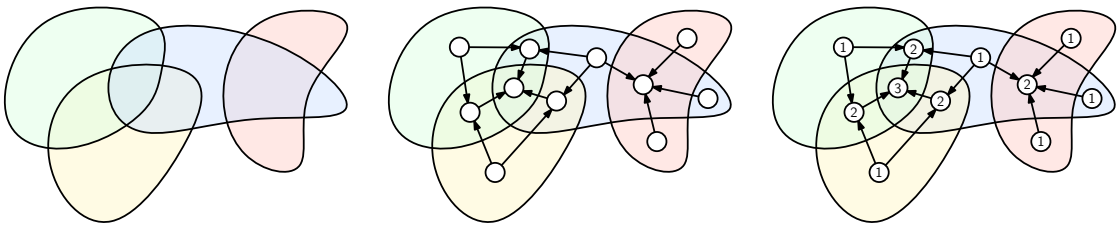
(c) (4 points) Show the result of running DFS to G, where the main program selects the next vertex to visit based on decreasing order of finish times from part (b).

(d) (2 points) Illustrate (e.g., by circling them or listing them out) the strong components of G.

Problem 2. (14 points) Graph theory has many applications in geometry. Suppose that you have a collection of overlapping geometric shapes in the plane (see Fig. 2(a)). The subdivides the plane into connected regions, such that the same set of objects overlap all the points in each region. Define the *depth* of a region to be the number of shapes that overlap it. (Let's assume that the shape boundaries intersect "nicely" in the sense that no two boundaries are tangent to each other. This implies that whenever a boundary is crossed, the depth increases or decreases by exactly 1.)

We can define a digraph to represent the overlap structure (see Fig. 2(b)). Each region is associated with a vertex of this graph. (For simplicity, we ignore the infinite external region is not overlapped by any shape.) There is an edge (u, v) between two vertices if the associated regions are adjacent to each other. The edge is directed from the region of lower depth to the region of greater depth. We call this a *layered digraph*.

(a) (4 points) Prove that the layered digraph of any finite collection of ("nicely intersecting") planar shapes is a DAG.



(a) (b) (c)

Figure 2: Layered digraph.

(b) (6 points) Present an efficient algorithm that assigns depth numbers to the vertices of a general layered digraph (see, for example, Fig. 2(c)). (Note that this can be done based solely on the digraph's structure, without reference to the geometry of the shapes.) Your algorithm should run in time $O(n+m)$, where n is the number of vertices and m is the number of edges.

(c) (4 points) Given any layered digraph, suppose that we ignore the edge directions, and consider the resulting undirected graph. Prove that this graph is *bipartite*, meaning that its vertex set can be partitioned into two sets V_1 and V_2 such that all edges go between these sets (never within the same set).

Problem 3. (12 points) Dijkstra's algorithm assumes that all the edge weights in a digraph are nonnegative.

(a) (4 points) Present an (ideally small) example that shows that Dijkstra's algorithm may fail to produce a correct result if the digraph has even a single negative-weight edge. (Note: It is not hard to find examples on the internet. Before doing this, try to come up with your own example to get a better understanding of why this is not utterly trivial.)

(b) (8 points) Suppose that your digraph has negative-weight edges, but these edges all emanate from the source vertex. Also, the digraph has not negative-cost cycles. Prove that Dijkstra's algorithm produces a correct result on such a digraph.

Problem 4. (12 points) A common optimization problem for word processors and typesetters is splitting a string of text into lines. The input to this problem consists of a text string T, which consists of a sequence of n words. For $1 \leq i \leq n$, let w_i denote the length of the ith word in T (see Fig. 3). Let's simplify matters by assuming that there are no spaces between words and words cannot be hyphenated. Given a line of length L, we want add line breaks so that no line has length greater than L, and each line is as full as possible. We call this a *segmentation*.

The simple greedy approach to segmentation is to fill each line with as many words as possible, so long as the total length does not exceed L (see Fig. 3). Then a new line is started with the next word in the string. This is called the *greedy segmentation*. In this problem, we will explore whether the greedy segmentation is optimal with respect to some metrics.

Consider the greedy segmentation for a given string T for a line of length L. Let m denote the total number of lines, and for $1 \leq j \leq m$, let r_j denote the *remaining space* on line j,

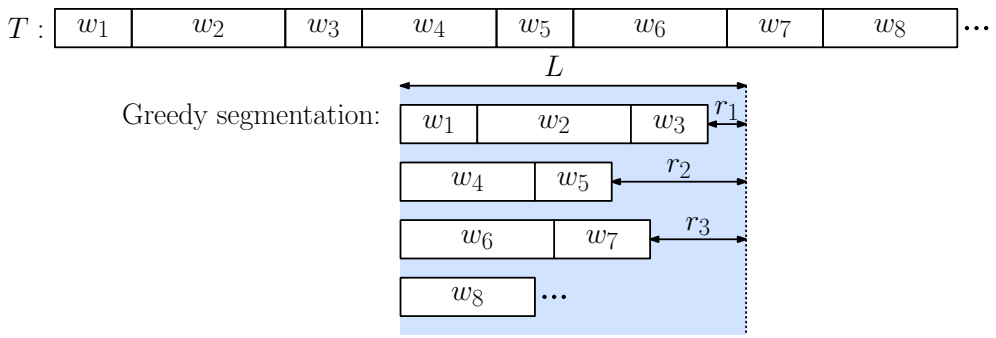


Figure 3: The greedy text segmentation for a text string T on a page of width L.

that is, L minus the space used by the words on this line. Define the *total penalty* and the *max penalty* of the greedy segmentation to be

$$P_t = \sum_{j=1}^{m} r_j \qquad \text{and} \qquad P_m = \max_{1 \le j \le m} r_j,$$

respectively. Answer the following questions about these two metrics.

(a) (8 points) Is the greedy segmentation optimal with respect to *total penalty*? Either give a proof or present a counterexample.

(b) (4 points) Is the greedy segmentation optimal with respect to the *max penalty*? Either give a proof or present a counterexample.

(Note: When presenting a counterexample, try to keep it as short as possible. Explain how the greedy segmentation differs from the optimum on your specific example. It is not necessary to philosophize as to why this happens.)

Challenge Problem. (Note: Challenge problems count for extra credit points. These additional points are factored in only *after* the final cutoffs have been set and can only increase your final grade.)

Given integers i and j, both greater than or equal to 2, we define an (i, j)-*horse* to be an undirected graph which consists of $n = i + j + 4$ vertices arranged in the following manner. There i vertices that form the horse's *front feet*, j vertices which form the *hind feet*, and four additional vertices, called the *head*, the *neck*, the *shoulder*, and the *hip*. These are connected as shown in Fig. 4 (that is, head-neck-shoulder-hip joined in a chain, and the front feet are joined to the shoulder and the rear feet are joined to the hip). In this problem, the objective is to design an algorithm that, given such a graph, can efficiently classify the vertices based on their body part.

The input to your algorithm is the $n \times n$ adjacency matrix for a horse. You are *not* told the values of i and j are. Note that the input size is n^2, but the objective of the problem is to classify the horse's body parts in only $O(n)$ time. This means that your algorithm does not even have time to look at every entry in the matrix! (Such an algorithm is called a *sublinear*

time algorithm, and this is only possible because we assume that the input is given to us and is of the proper format.)

Given the $n \times n$ adjacency matrix of a horse, your algorithm should determine the values of i and j, and identify the indices of the head, neck, shoulder, and hip.

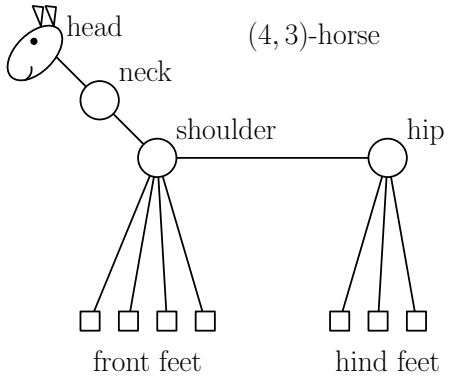


Figure 4: Challenge Problem - Classifying the vertices of an (i, j)-horse.

Explain why your algorithm is correct and derive its running time. (Hint: This can be done in $O(n)$ time.)

Some tips about writing algorithms: Throughout the semester, whenever you are asked to present an "algorithm," you should present three things: (1) a description of the algorithm, (2) an informal proof/explanation of its correctness, and (3) an explanation/analysis of its running time. Remember that your description is intended to be read by a human, not a compiler, so conciseness and clarity are preferred over technical details. I would prefer pseudocode over actual Java/Python code, and if it is clearer, I would prefer an English explanation over pseudocode. When doing analyses of running times, a high-level explanation is almost always sufficient (unless the analysis relies on a tricky recurrence or summation). As an guide, consider how algorithms and analyses are presented in lecture notes.

Giving careful and rigorous proofs can be quite cumbersome, and so you are encouraged to use intuition and give illustrations whenever appropriate. Your principal objective is to convince the grader that you clearly understood why your algorithm is correct. Beware, however, that a poorly drawn figure or one that is too simplistic can make certain erroneous hypotheses appear to be "obviously correct" when they are not.

Unless otherwise stated, you may use any results from class, or results from any standard textbook on algorithms and data structures. If you use sources from outside of class, including generative AI, you must cite your sources. (There is no penalty for using external materials, but there is a stiff penalty if you are found responsible of academic dishonesty.) If you have collaborated with classmates, that is fine provided that you credit your collaborators and write everything in your own words. (Do not copy!) This can be very quick. (For example, "I got this idea from StackExchange," or "Mary Smith and I worked on this together.") If you are unsure, please feel free to check with me.