Homework 1: Algorithm Design Basics

Handed out Thu, Feb 7. Due at the start of class Tue, Feb 19. Late homeworks are not accepted, but you may drop your lowest homework score.

References: For reference information on asymptotics, summations, and recurrences, see either the text, by Cormen, Leiserson, Rivest, and Stein or the text by Kleinberg and Tardos. Also see the class handout on Background Information.

Notation: Throughout the semester, we will use $\lg x$ to denote logarithm of x base 2 ($\log_2 x$) and $\ln x$ to denote the natural logarithm of x. We will use $\log x$ when the base does not matter.

Problem 1. The eminent but flaky Professor Hubert J. Farnsworth[1] proposed the following simpler algorithm for the stable marriage problem. As in the standard problem, there are n men and n women, and each man and each woman has an n-element preference list that orders all the members of the opposite sex.

(1) Repeat the following for i running from 1 to n:
  - (a) Man m_i proposes to the highest woman on his preference list.
  - (b) This woman accepts this proposal, and is immediately removed from the preference lists of all remaining men.

(2) Output the resulting sets of mates.

(Note that in his algorithm, once a woman accepts a man's proposal, she will never break it off.) We will explore the correctness of Farnsworth's algorithm by answering the following questions.

(a) Is the matching produced by Farnsworth's algorithm guaranteed to be stable? If so, give a proof. If not, present a counterexample. (See the note below.)

(b) Suppose that all the woman in this system have exactly the same sets of preferences, and in particular, they rank the men in (decreasing preference) order $\langle m_1, m_2, \ldots, m_n \rangle$. The men's preferences are not constrained. (Of course, each man's list must contain all the women.) Under this restriction, is the matching produced by Farnsworth's algorithm guaranteed to be stable? As before, either give a proof or present a counterexample.

(Throughout the semester, whenever you are asked to present a counterexample, you should strive to make your counterexample as short and clear as possible. In addition to giving the input for the counterexample, briefly explain what the algorithm does when run on this input and why it is wrong.)

Problem 2. Consider the following summation, which holds for all $n \geq 0$,

$$\sum_{i=1}^{n} i^3 = \left(\sum_{i=1}^{n} i\right)^2$$

That is $(1^3 + 2^3 + \cdots + n^3) = (1 + 2 + \cdots + n)^2$.

(a) Prove this identity holds for all $n \geq 0$, by induction on n. (Recall that by convention, for $n = 0$ we have an empty sum, whose value is defined to be the additive identity, that is, zero.)

(b) The following figure provides an informal "pictorial proof" of this identity. Explain why.

[1] Famous inventor of the *Coolometer*, a hand-held device that measures a persons "coolness" as measured in "MegaFonzies."

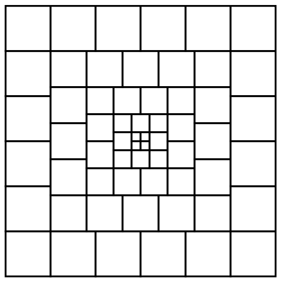


Figure 1: Problem 2.

Problem 3. For each of the parts below, list the functions in increasing asymptotic order. In some cases functions may be asymptotically equivalent (that is $f(n)$ is $\Theta(g(n))$). In such cases indicate this by writing $f(n) \equiv g(n)$. When one is asymptotically strictly less than the other (that is, $f(n)$ is $O(g(n))$ but $f(n)$ is not $\Theta(g(n))$), express this as $f(n) \prec g(n)$. For example, given the set:

$$n^2 \qquad n \log n \qquad 3n + n \log n,$$

the answer would be

$$n \log n \;\equiv\; 3n + n \log n \;\prec\; n^2.$$

Explanations are *not* required, but may be given to help in assigning partial credit. (Hint: Review Chapt. 3 in CLRS, especially the section on logarithms.)

(a)	$(3/2)^n$	$3^{(n/2)}$	$2^{(n/3)}$
(b)	$\lg n$	$\ln n$	$\lg(n^2)$
(c)	$n^{\lg 4}$	$2^{\lg n}$	$2^{(2 \lg n)}$
(d)	$\max(50n^2, n^3)$	$50n^2 + n^3$	$\min(50n^2, n^3)$
(e)	$\lceil n^2/20 \rceil$	$\lfloor n^2/20 \rfloor$	$n^2/20$

Problem 4. The purpose of this problem is to design a more efficient algorithm for the previous larger element problem, as introduced in class. Recall that we are given a sequence of numeric values, $\langle a_1, a_2, \ldots, a_n \rangle$. For each element a_i, for $1 \le i \le n$, we want to know the index of the rightmost element of the sequence $\langle a_1, a_2, \ldots, a_{i-1} \rangle$ whose value is strictly larger than a_i. If no element of this subsequence is larger than a_i then, by convention, the index will be 0. Here is naive the $\Theta(n^2)$ algorithm from class.

```
previousLarger(a[1..n]) {
    for (i = 1 to n)
        j = i-1;
        while (j > 0 and a[j] <= a[i]) j--;
        p[i] = j;
    }
    return p
}
```

There is one obvious source of inefficiency in this algorithm, namely the statement `j--`, which steps through the array one element at a time. A more efficient approach would be to exploit p-values that have already been constructed. (If you don't see this right away, try drawing a picture.) Using this insight, design a more efficient algorithm. For full credit, your algorithm should run in $\Theta(n)$ time. Prove that your algorithm is correct and derive its running time.

Challenge Problem. 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.

An *in-place algorithm* is one that is given its input in a chunk of memory (e.g., as an array) which it is allowed to modify in the course of the algorithm, it stores its output in this same chunk of memory, but it is only allowed to use $O(1)$ additional working storage. That is, it can use a constant number of primitive variables, but it cannot declare any additional arrays (including strings) nor can it allocate memory dynamically. It is also not allowed to make recursive calls, since doing so would allow it to use the recursion stack implicitly for working storage. The in-place restriction is often important in applications where the inputs are very large arrays, that barely fit into physical memory.

Suppose that you are given an array $X[1..n]$ of numbers, and you are given an index i, where $1 \leq i \leq n-1$. Consider the two subarrays, $X[1..i]$ and $X[i+1..n]$. Of course, these subarrays are not necessarily of the same size. Give an $O(n)$ time *in-place* algorithm which, given as input the array X and i, swaps these two subarrays within X, placing the contents of $X[i+1...n]$ before $X[1..i]$. The order of elements within each subarray should not be changed. An example is presented below.

	n	i	X
Input:	11	4	`[ 6  9  2 10  3  7 -4 18  5  1  0]`
Output:			`[ 3  7 -4 18  5  1  0  6  9  2 10]`

A Note about Writing Algorithms: When presenting algorithms, more detail is not necessarily better. Remember that your algorithm will be read by a human, not a compiler. You may assume that the reader is familiar with the problem. Be sure that your pseudo-code is sufficiently detailed that your intentions are clear and unambiguous. However, avoid adding extraneous details and confusing jargon. (E.g., It is much clearer to say "Insert x at the end of the list" rather than `list.insertAtEnd(x)`.) In addition to presenting the pseudo-code, explain your general strategy in English. (This way, if you make a coding error, the grader can ascertain your real intent and give partial credit.) It is also a good idea to provide an example to illustrate your approach. Even if you are not explicitly asked, you should always provide a justification of correctness of your algorithm and analysis of its running time.

Solutions to Homework 1: Algorithm Design Basics

Solution 1:

(a) No, it is not stable. The following set of preferences provides a counterexample.

Men		Women	
m_1	m_2	w_1	w_2
w_1	w_1	m_2	m_1
w_2	w_2	m_1	m_2

In the first step, man m_1 proposes to woman w_1, and they are married. This forces m_2 to marry his second choice, w_2. Observe that (m_2, w_1) is an instability, since m_2 prefers w_1 to his mate and w_1 prefers m_2 to her mate.

(b) In this case the algorithm is stable. To see this, suppose to the contrary that there was an instability (m, w). This means that there exist pairing (m, w') and (w, m') generated by the algorithm, where m prefers w to his mate w', and w prefers m to her mate m'. Since all the women prefer men in the order in which they are processed, the fact that w prefers m to m' implies that m would have been processed before m' was. Thus, m would have taken the highest remaining woman from his list. Clearly w has not yet been taken yet (because she was around for m' to take later), and so m' would have selected w over w', a contradiction.

Solution 2:

(a) For any $n \geq 0$, let $L(n) = \sum_{i=1}^{n} i^3$ and let $R(n) = \sum_{i=1}^{n} i$. We want to show that $L(n) = R(n)^2$, for all $n \geq 0$.

The basis case of the induction proof is for the smallest legal value of n, namely $n = 0$ (not $n = 1$). In this case, both sums are empty, and hence $L(0) = R(0)^2$. For the induction step, consider $n > 0$. Our induction hypothesis is that, $L(n') = R(n')^2$, where $0 \leq n' < n$. To prove that the identity holds for n itself, we will write the summations and remove the last term, so that the induction hypothesis can be applied.

$$\begin{aligned} L(n) &= \sum_{i=1}^{n} i^3 = \left(\sum_{i=1}^{n-1} i^3\right) + n^3 = L(n-1) + n^3 \\ R(n) &= \sum_{i=1}^{n} i = \left(\sum_{i=1}^{n-1} i\right) + n = R(n-1) + n. \end{aligned}$$

From the last line we see that $R(n-1) = R(n) - n$. From our induction hypothesis (where $n' = n - 1$) we have $L(n-1) = R(n-1)^2$, and by combining these two facts we obtain

$$\begin{aligned} L(n) &= L(n-1) + n^3 = R(n-1)^2 + n^3 = (R(n) - n)^2 + n^3 \\ &= R(n)^2 - 2nR(n) + n^2 + n^3. \end{aligned}$$

Observe that this is almost what we want, but we need to show that the last three terms above cancel. That is, it suffices to show that $2nR(n) = n^2 + n^3$. This follows from the formula for the arithmetic series:

$$\begin{aligned} R(n) &= \sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2+n}{2} \\ 2nR(n) &= n^3 + n^2, \end{aligned}$$

which completes the proof.

(b) Define the *size* of a square to be length of one of its sides. Let us group the squares of the figures into groups of equal size. Working from inside to outside, the square sizes grow as $1, 2, 3, \ldots, n$. The number of squares within the various size groups are $4, 8, 12, \ldots, 4n$. Using the fact that a square of side length x has area x^2, we can compute the area of the large square by summing the areas of the squares of each size group:

$$4 \cdot 1^2 + 8 \cdot 2^2 + 12 \cdot 3^2 + \ldots + 4n \cdot n^2 \;=\; 4 \sum_{i=1}^{n} i^3.$$

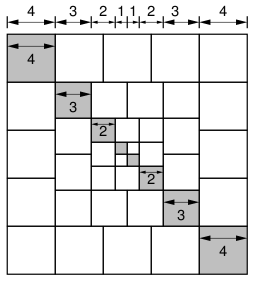


Figure 1: Solution to Problem 1(b).

Let's consider another way to compute this area. Consider the squares that lie along the diagonal. (See Fig. 1.) By observing that these squares do not overlap horizontally, and they cover the entire width of the large square, it follows that the top side length of the large square is

$$(n + (n-1) + \ldots + 2 + 1) \;+\; (1 + 2 + \ldots (n-1) + n) = 2(1 + 2 + 3 + \ldots + n),$$

Applying this to the vertical side as well, it follows that the area of the entire square is the square of this quantity, or

$$4 \left(\sum_{i=1}^{n} i \right)^2 .$$

These two areas must be equal, and so, after cancelling the common factor of 4, we get the desired result.

Solution 3:

$$\begin{array}{llclcl} \text{(a)} & 2^{(n/3)} & \prec & (3/2)^n & \prec & 3^{(n/2)} \\ \text{(b)} & \lg n & \equiv & \ln n & \equiv & \log_{1.2} n \\ \text{(c)} & 2^{\lg n} & \prec & n^{\lg 4} & \equiv & 2^{(2 \lg n)} \\ \text{(d)} & \min(50n^2, n^3) & \prec & 50n^2 + n^3 & \equiv & \max(50n^2, n^3) \\ \text{(e)} & \lfloor n^2/20 \rfloor & \equiv & n^2/20 & \equiv & \lceil n^2/20 \rceil \end{array}$$

Here are some explanations.

(a) We use the fact that $a^{bc} = (a^b)^c$, and hence $2^{(n/3)} = (2^{1/3})^n \approx 1.26^n$, $3^{(n/2)} = (3^{1/2})^n \approx 1.73^n$, and finally $(3/2)^n = 1.5^n$.

(b) The first two are equivalent because changing the base of a logarithm only changes the function by a constant factor. The last one can be rewritten as $\lg(n^2) = 2 \lg n \equiv \lg n$.

(c) $2^{\lg n} = n$, $n^{\lg 4} = n^2$, and finally $2^{2 \lg n} = (2^{\lg n})^2 = n^2$.

(d) A nice fact to use here is that $\max(f, g) \equiv f + g$, which follows from the observation that $\max(f, g) \leq f + g \leq 2 \max(f, g)$. This shows the second relationship. To see the first observe that for all sufficiently large n, $\min(50n^2, n^3) = 50n^2 = \Theta(n^2)$ for all sufficiently large n, whereas the middle term is $\Theta(n^2)$.

(e) The floor and ceiling are within an additive constant of 1 of the original value. Since all three values tend to infinity, this difference is negligible in the limit.

Solution 4:

The modified algorithm makes use of the observation about using the previously computed p-values by replacing the statement "`j--`" with "`j = p[j]`" in the while loop. To see why this is correct, observe that in order to get into the body of the while loop, it must be that $a[j] \leq a[i]$. Since $a[p[j]]$ is the previous element of the array to be larger than $a[j]$, all the elements from $a[p[j]]$ to $a[j-1]$ are not greater than $a[j]$, and hence they are not greater than $a[i]$. Thus, we may skip over all of them in one step. Here is the resulting algorithm:

```
previousLarger2(a[1..n]) {
    for (i = 1 to n)
        j = i-1;
        while (j > 0 and a[j] <= a[i]) j = p[j];
        p[i] = j;
    }
    return p
}
```

Fig. 2 illustrates this idea. The left side shows the output of the algorithm and the right side shows (in heavy lines) where jumps were made using the p-values.

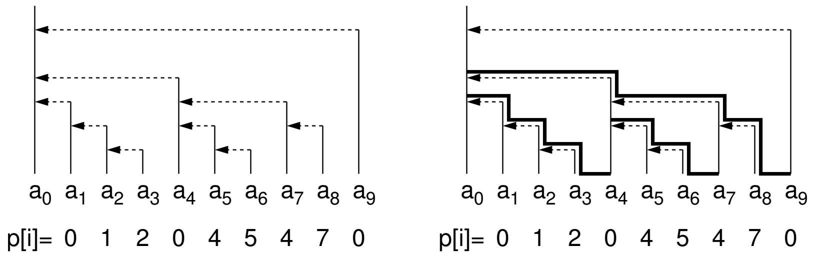


Figure 2: Solution to Problem 4.

By the comments made above this algorithm is correct, but is it asymptotically more efficient? You might be tricked into thinking not. Consider the following example, the elements $a[1..i-1]$ are in decreasing order, and thus, for $1 \leq j < i$, $p[j] = j - 1$. Suppose that $a[i]$ is larger than all these elements. Using the p-values is no help here, since we would have to go point by point back to the start of the array. But observe that this can happen only once! Once $a[i]$ is added, we will never again visit all the elements, because $p[i]$ will jump over all of them.

To make this insight more formal, we say that a pointer $p[j]$ is *touched* if the statement `j = p[j]` is executed for this value of j. We assert that each $p[j]$ can only be touched once. The reason is that if $p[j]$ is touched in the process of computing $p[i]$ $(i > j)$, then we know $p[i] \leq p[j]$, and therefore all subsequent searches will use $p[i]$ to "leap frog" over $p[j]$. Since each $p[j]$ value can be touched only once, and since each iteration of the while loop body touches one additional element, it follows that the total number of iterations of the while loop is $O(n)$. (Pretty neat!)

Solution to the Challenge Problem:

There are a number of different solutions that I can see for this problem. The first solution that comes to mind is not very easy to implement correctly. It involves cyclically moving items into their correct final position. Each movement causes another item to be displaced. This is repeated until returning to the initial item. The problem is that if the two subarray lengths i and $n - i$ share any common factors (e.g. when $i = 4$, $n = 10$), then this algorithm may skip over elements. To get a correct implementation, it is necessary to identify where the various "orbits" of shifted elements and show that each element of each orbit is only moved once. I will not attempt this solution, since the following two solutions are much simpler.

This first solution is based on swapping arrays of equal size. Let α and β denote the two subarrays $X[1..i]$ and $X[i+1..n]$, respectively, so that X is equal to the concatenated sequence $\alpha\beta$. If the two subarrays have equal size, then you can just interchange them element by element. If not, suppose that α is smaller than β. Then β can be written as $\beta = \beta_1\beta_2$, where β_2 has the same length as α. Then we swap α with β_2, which puts α into its proper position and leaves us with $X = \beta_2\beta_1\alpha$. Now, we can just continue to swap β_2 with β_1, and repeat this process until the two subarrays that remain to be swapped have equal size. A similar method is used if β is smaller than α. I will not give all the details, but I claim that this can be implemented to run in $O(n)$ total time. To see this, observe that with each swap of two elements, one of the elements being swapped (the element of α) is put in its final position, never to be moved again. This can only happen $O(n)$ times before all the elements are in their final positions.

This solution is the simplest of all. It is based on a general procedure $\text{Rev}(X, i, j)$, which reverses any subarray $X[i..j]$. This is easy to do in-place, by simply swapping opposite entries. Note that calling $\text{Rev}(X, 1, n)$ will swap the arrays α and β, but it will also reverse the order of elements, resulting in the sequence $\beta^R\alpha^R$ (where the superscript R denotes reversal). Thus, to swap α and β, we first reverse them, and then reverse the entire list. Clearly, this takes $O(n)$ time.

```
Swap(X, n, i) {                    // swaps X[1..i] with X[i+1..n]
    Rev(X,   1, i)                 // reverse X[1..i]
    Rev(X, i+1, n)                 // reverse X[i+1..n]
    Rev(X,   1, n)                 // reverse everything
}

Rev(X, i, j) {                     // reverses X[i..j]
    while (i < j) do { swap X[i++] with X[j--] }
}
```

Homework 2: BFS, DFS, and Greedy Algorithms

Handed out Thu, Feb 21. Due at the start of class Tue, Mar 4. Late homeworks are not accepted, but you may drop your lowest homework score.

As always, if you are asked to present an algorithm, in addition to the algorithm you must justify its correctness and derive its running time.

Problem 1. Consider the BFS algorithm presented in class. Given a digraph $G = (V, E)$ and a starting vertex $s \in V$, this algorithm computes for each vertex $u \in V$ the value $d[u]$, which is the length (number of edges) on the shortest path from s to u. For purposes of robust network communications, it is sometimes useful to know the number of shortest paths. The objective of this problem is to modify the BFS algorithm of class to compute the number of shortest paths from s to each vertex of G. We will do this in two steps.

(a) First run the standard BFS on G, starting from s. Explain how to use the result of this BFS to produce a new digraph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$, such that every path in G' that starts at s, is a shortest path in G starting from s, and conversely, every shortest path in G starting from s is a path in G'.

(b) Explain how to take the result of part (a) to compute for each vertex $u \in V'$, a quantity $c[u]$, which is the number of paths in G' from s to u. (Hint: This can be done by a modification of BFS.)

Both of your algorithms should run in $O(V + E)$ time.

Hint: Be careful in how you compute $c[u]$. In general, the number of shortest paths from one node to another can be exponential in the size of V. (You might think about why this is.) This means that your algorithm cannot generate the paths one by one. Rather, it must employ some method that can count up many paths at once.

Problem 2. Each of the following problems involves knowledge of cycles and paths in undirected and directed graphs. For all three parts, assume that the graph or digraph is represented using an adjacency list.

(a) Present an algorithm which, given an undirected graph $G = (V, E)$, determines whether G contains a cycle in $O(V)$ time. (It is important that the running time of your algorithm is independent of E, which may generally be as high as $\Omega(V^2)$.)

(b) The eminent but flaky Professor Hubert J. Farnsworth[1] claims that he has discovered an $O(V)$ time algorithm that determines whether a digraph has a cycle, assuming that the digraph is represented using a standard adjacency list. Prove that the Professor is wrong!

Hint: Assume that you have the professor's source code. Based on his code and given an arbitrary value V for the number of vertices, devise an digraph with $\Omega(V^2)$ edges, with the property that if his algorithm fails to look at *every* edge of your graph, you can trick his algorithm into producing the wrong result by altering one of the edges that his algorithm failed to look at.

(c) A digraph $G = (V, E)$ is said to be semi-connected if, given any two vertices, $u, v \in V$, there exists a path from u to v, or there exists a path from v to u, but not necessarily both. Give an algorithm which, given a directed acyclic graph (DAG), determines whether it is weakly connected.

[1]Noted inventor of the *electric frankfurter* and winner of the Academy of Sciences award for "Advances in superconductivity properties of sandwich meats."

Problem 3. The following two problems involve algorithms for connectivity in graphs and digraphs.

(a) Recall that a *bridge* is an edge in a connected, undirected graph whose removal causes the graph to become unconnected. Present an $O(V+E)$ algorithm, which reports all the bridges in a graph.

(b) The concept of a bridge can be extended to directed graphs. (I don't know that it has any particular name, so I'll make one up.) Given a digraph $G = (V, E)$, an edge (u, v) is called a *span* if there is no path from v to u in G. Present an $O(V + E)$ algorithm, which determines all the tree edges of the DFS forest that are spans. (Note that cross edges and forward edges might be spans as well, but your algorithm should only report on the DFS tree edges that are spans.)

Hint: Both algorithms can be implemented by an appropriate modification of the algorithm we gave in class for biconnected components.

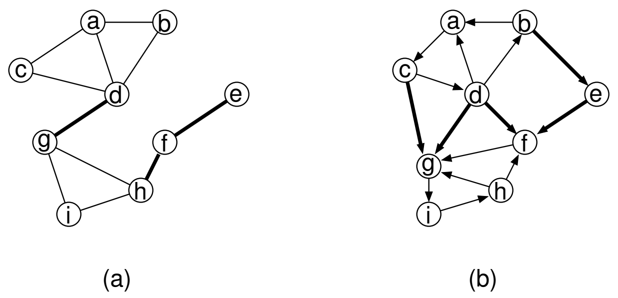
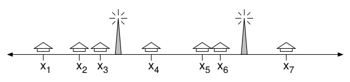
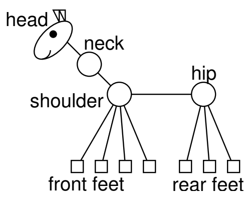


Figure 1: Problem 3(a) and 3(b). Bridge edges (shown as bold lines) in (a) are $\{d, g\}$, $\{f, h\}$, and $\{e, f\}$. Span edges (shown in bold in (b)) are (c, g), (d, g), (d, f), (b, e), and (e, f).

Problem 4. Consider a long straight road with houses scattered very sparsely along its length. Let $x_1 < x_2 < \ldots < x_n$ denote the locations of these houses. Design a greedy algorithm that will place the minimum number of cell towers along this road so that every house is within four miles of one of the towers. (Your algorithm should run in $O(n)$ time.) Prove that the result it produces is optimal. (Try to be rigorous in your proof, for example, by showing that any optimal solution that is not equal to the greedy solution can be converted into a solution of equal size that is closer to the greedy solution, in some sense.)



Problem 4



Challenge Problem

Challenge Problem. 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)-*goat* to be an undirected graph which consists of $n = i + j + 4$ vertices. The vertices consist of i front feet, j rear feet, and four extra vertices, the head, the neck, the shoulder, and the hip. These are connected as shown below (head-neck-shoulder-hip, and front feet to shoulder and rear feet to hip) (see the figure above).

You are given the $n \times n$ adjacency matrix for an n-vertex goat, but you are *not* told what the values of i and j are. Give an $O(n)$ time algorithm which, given the adjacency matrix of a goat, determines the values of i and j, and labels all the vertices according to their type (head, neck, shoulder, hip, front foot or rear foot). Note that because the adjacency matrix has n^2 entries, you cannot scan the entire matrix in $O(n)$ time. Explain how your algorithm works. (Try to keep it as simple as possible.)

Solutions to Homework 2: BFS, DFS, and Greedy Algorithms

Solution 1:

(a) Let $V' \subseteq V$ be the vertices that the BFS reaches. Observe that these are the only vertices reachable from s, and therefore the shortest paths from s can only use these vertices. For each edge $(u, v) \in E$, if both u and v are in V', and if $d[v] = d[u] + 1$, then put this edge in E'. (This will include all the tree edges of the BFS tree and may include some of the other edges as well.) Let $G' = (V', E')$ be the resulting digraph.

Observe that every path in the resulting graph that starts from s has the property that if the path consists of k edges, if and only if it terminates at a vertex whose d-value is k. Thus, all paths of G' are shortest paths in G. Conversely, any shortest path in G that starts from s has the property that the d-values increase by exactly 1 along the path, and therefore, all the edges of this path are in E'. Thus, all the shortest paths of G are paths of G'.

(b) For each vertex $u \in V'$, we will compute a quantity $c[u]$, which is the number of paths in G' from s to u. By part (a), this will be equal to the number of shortest paths from s to u in the original graph G. We do this by a modification to BFS. Initially we set $c[u] = 0$ for all vertices other than s and we set $c[s] = 1$. (The latter reflects the fact that there is a single path from s to itself, namely the trivial path of length 0.) Whenever the BFS encounters an edge $(u, v) \in E'$, we set $c[v] = c[v] + c[u]$.

Clearly the algorithm runs in the same $O(V + E)$ time as BFS. We claim that, on termination, for all $u \in V'$, $c[u]$ contains the number of paths from s to u in G'. The proof is by induction on the length of the path, that is, the d-value of the vertex. Consider any vertex v such that $d[v] = k$, and let us assume inductively that for all vertices whose d-value is $k - 1$ or smaller, the c-value is the correct count. By the construction of (a), we know that all the edges coming into v are of the form (u, v) where $d[u] = k - 1$, and by the induction hypothesis $c[u]$ is correctly computed as the number of paths from s to u. Clearly, every path from s to v involves going from s to one of these vertices u, and then following the edge from u to v. The number of paths from s to v going through the vertex u is $c[u]$ by the induction hypothesis. Therefore, the total number of paths from s to v is just the sum of these path counts overall the incoming vertices u. Therefore, the algorithm is correct.

Solution 2:

(a) Recall from class that a graph has a cycle if and only if a DFS (in fact any DFS) has a back edge. It is easy to modify DFS to determine the existence of a back edge. In particular, when processing the neighbors v of a vertex u in a call to DFSvisit(u), we need only check whether v's color is gray and v is not the parent of u (that is, pred[u] $\neq v$). If so, v is an ancestor, implying that (u, v) is a back edge, and we immediately terminate the algorithm. Recall that a tree with V vertices has at most $V - 1$ edges, and so we are guaranteed to discover the existence of a back edge (if it exists) after processing $O(V)$ edges of G.

(b) Given the number of vertices V, let the vertices be $\{1, 2, \ldots, V\}$ and create edges (u, v) for all $1 \leq u < v \leq V$. Clearly this graph has no cycles. Run Farnsworth's algorithm on this digraph. Since his algorithm runs in $O(V)$ time, for all sufficiently large V, it cannot look at all the edges of the graph, and in particular, there must be some vertex u such that Farnsworth's algorithm did not make it to the end of u's adjacency list. If his algorithm reported that the digraph has a cycle, then clearly it is wrong. If, on the other hand, it reported that the digraph is acyclic, replace the last entry of u's adjacency list with a self-loop (u, u) (or any other edge emanating from u that will induce a cycle).

Since his algorithm never saw this entry of the adjacency list, its behavior on your modified digraph cannot be different from the behavior on the original digraph. But its answer is now clearly wrong. (This type of proof is called an *adversary construction.* You play the role of an adversary, by showing that no matter what the algorithm does, you will cause it to fail. This technique is often useful in proving lower bounds.)

(c) The algorithm is quite simple. Apply topological sort to produce a topological ordering of the vertices, denoted $\langle u_1, u_2, \ldots, u_V \rangle$. If for every consecutive pair (u_i, u_{i+1}), this edge exists within the DAG, then report that the DAG is semi-connected. Otherwise, report that it is not.

This algorithm can be implemented in $O(V + E)$ time. (Observe that checking the edges can be done by walking down each of the adjacency lists, exactly once, which will take $O(V + E)$ time.)

To establish correctness, we assert that G is semi-connected if and only if all the consecutive edges exist. Clearly, if all these edges exist, there is a single path passing through all the vertices (this is called a *Hamiltonian path*), and if such a path exists, the graph is clearly semi-connected. Conversely, suppose that one of the edges (u_i, u_{i+1}) does not exist. Then we assert that there can be no path from u_i to u_{i+1} nor vice versa. Clearly, the reverse path from u_{i+1} to u_i cannot exist, since (by definition of topological sort) the edges of the DAG only proceed to vertices that are later in the sorted sequence. To see that there can be no path from u_i to u_{i+1}, observe that any edge leaving u_i must go to a vertex u_j, where $j \geq i + 2$. But (as we just argued) no path from u_j can go back to u_{i+1}.

Solution 3:

(a) We claim that an edge (u, v) is a bridge if and only if (1) (u, v) is a tree edge in the DFS, and (2) (assuming u is the parent and v is the child) there is no back edge from the subtree rooted at v to a proper ancestor of v (including u). To prove this first observe that (u, v) cannot be a back edge. The removal of even all the back edges would not result in a disconnected graph, since the tree edges provide all the connectivity. Thus, we may assume that any bridge must be a tree edge of the DFS tree. Let us assume without loss of generality that u is the parent of v. Then observe that if there were a back edge that "goes around" (u, v) to any proper ancestor of v (including u) then (u, v) cannot be a bridge, since this back edge would keep the graph connected. Conversely, if there is no such back edge, then the removal of (u, v) would imply that there is no path from u to v, and hence this is a bridge.

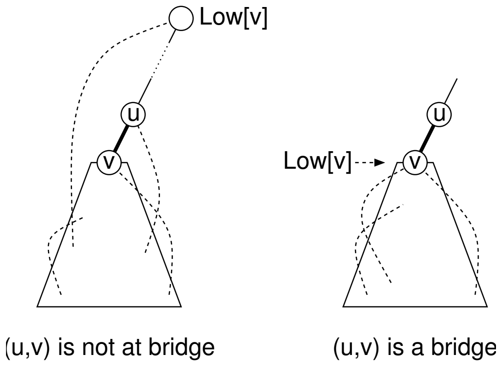


Figure 1: Solution to Problem 3(a).

To test this condition for given tree edge (u, v) we make use of Low$[v]$ (see the Lecture on Articulation Points). To check whether no back edge rises from the subtree rooted at v to a vertex that is strictly higher than v, it suffices to test that Low$[v] > d[u]$. If so (u, v) is a bridge. The pseudo-code is a minor modification of the articulation point algorithm. Note that there is no need for a special case for the root. As with all DFS algorithms, the running time is $O(V + E)$.

```
findBridges(u) {
  color[u] = gray;
  Low[u] = d[u] = ++time;          // set discovery time and init Low
  for each (v in Adj(u)) {
    if (color[v] == white) {       // (u,v) is a tree edge
      pred[v] = u;                 // v's parent is u
      findBridges(v);
      Low[u] = min(Low[u], Low[v]) // update Low[u]
      if (Low[v] > d[u]) Report that (u,v) is a bridge
    }
    else if (v != pred[u])        // (u,v) is a back edge
      Low[u] = min(Low[u], d[v])  // update Low[u]
  }
}
```

(b) The answer is tricky, so we will just sketch the solution. It is tempting to think that we can compute something analogous to Low$[u]$, as we did for bridges and apply a condition involving this value and discovery times of u and v. The value of For example we Low$[u]$ is the minimum of the discovery time of u and the vertex with the smallest discovery time that is reachable by a path of the special following form. Starting at u follow any number of tree or forward edges from u followed by a single cross or back edge. It is not hard to prove that this quantity could be computed as follows, within the context of DFS:

Initialization: Low$[u] = d[u]$.
Tree or forward edge (u, v)**:** Low$[u] = \min(\text{Low}[u], \text{Low}[v])$.
Back or cross edge (u, v)**:** Low$[u] = \min(\text{Low}[u], d[v])$.

How would we use this digraph-version of Low$[u]$? When we see a tree edge (u, v), we want to know whether v can reach u. Certainly if there is some path of the above special form that reaches an ancestor of u (including u itself), then there is a path from v to v' to u, implying that (u, v) is not a span edge. Unfortunately, although this condition is sufficient, it can be shown that there might be other ways (involving multiple back edges) that might complete the cycle.

Although I will not formally justify it formally (since it would take too long), there is a somewhat more complex solution. When we update Low$[u]$ on seeing a cross edge, we need to be careful to detect whether we are entering a subtree that can reach u or one that cannot. Suppose that (u, v) is a cross edge, and consider the path from v towards the root until first arriving at a vertex u' that is an ancestor of both v and u. If there is a span edge anywhere along this path, then v cannot reach u, and so we do not update Low$[u]$. On the other hand, if there is no span along this path, then we know that v and reach back to an ancestor of u, and hence v can reach u itself. Thus, in this case we update Low$[u]$. In order to check for span edges, our approach will be to maintain a stack of vertices that we have seen. When a span (u, v) is detected, we pop all the vertices on the stack until we get back to v, and mark them all. (We claim that none of these vertices can reach an ancestor of u, but proving this formally would take quite a bit of work.) The algorithm is presented in the code block below. Initially all vertices are unmarked, and their colors are white.

Solution 4: Assume that all distances are in miles. Initially let $i = 1$. Scan down the list of houses starting with $x_i, x_{i+1}, \ldots$, until we either go past the last house or else we come the first house x_j such that $x_j - x_i > 8$. (If we go past the last house, let $j = n + 1$.) This means that $x_{j-1} - x_i \leq 8$. Place the cell tower some place that will serve all the houses from i to $j - 1$ (e.g., at the midpoint, $(x_i + x_{j-1})/2$).

```
findSpans(u) {
  color[u] = gray;
  Low[u] = d[u] = ++time;          // set discovery time and init Low
  push u onto stack;
  for each (v in Adj(u)) {
    if (color[v] == white) {       // (u,v) is a tree edge
      pred[v] = u;                 // v's parent is u
      findSpans(v);
      Low[u] = min(Low[u], Low[v]) // update Low[u]
      if (Low[v] > d[u]) {         // this is a span edge
        Report that (u,v) is a span
        repeat {
          w = pop stack;
          mark w;
        until (w == v);
        }
      }
    }
    else if (d[v] < d[u] && w is unmarked) {
      Low[u] = min(Low[u], d[v])  // a back edge or a cross edge to an unmarked vertex
    }
  }
}
```

Clearly the running time is $O(n)$. To see that this is optimal, it will help to identify any solution to this problem with a sequence $\langle i_1, i_2, \ldots, i_k \rangle$, where i_j is the index of the rightmost house serviced by the jth tower. That is, tower j services houses at $i_{j-1}+1$ through i_j. (By convention, let $i_0 = 0$.) Let A be an optimal solution to the problem, and let G be the greedy solution. If $A = G$, then we are done. If not, then consider the associated sequences, and let j denote the first index at which these sequences differ:

$$\begin{aligned} A &= \langle a_1, a_2, \ldots, a_{j-1}, a_j, a_{j+1}, \ldots \rangle \\ G &= \langle a_1, a_2, \ldots, a_{j-1}, g_j, g_{j+1}, \ldots \rangle. \end{aligned}$$

By definition, G spans the greatest number of houses before placing a cell tower, and thus $g_j > a_j$. Since G is a valid solution, g_j covers all the houses from $a_{j-1}+1$ to g_j, and therefore it covers all the houses from $a_{j-1}+1$ to a_j. Let us form a new sequence A' by replacing a_j by g_j in sequence A.

$$A' = \langle a_1, a_2, \ldots, a_{j-1}, g_j, a_{j+1}, \ldots \rangle.$$

The resulting sequence has the same size as A, and so is also optimal. Also, as argued above, it covers all the same houses as A does, and so it is a feasible solution. Also, A' has one more interval in agreement with G, and so, in this sense, it is more like the greedy algorithm than A. By repeating this process a finite number of times, we will eventually have an optimal solution that is identical the greedy solution, which implies that the greedy solution is itself optimal.

Solution to Challenge Problem:

We need a constant number of applications of the following basic utilities. It is easy to see that each can be computed in $O(n)$ time.

- Compute the degree of a given vertex.
- Select an arbitrary neighbor of a given vertex.

- Mark the neighbors of a given vertex.
- Select any unmarked vertex.

Our algorithm will be a (rather long) straight-line program (no loops) where each step may invoke one of the above utilities. Thus, it will follow that its total running time is $O(n)$.

Our initial goal is to find the head. We'll see below how to do this, but assuming we have the head for now, we'll show how to finish the job off. First, we find the head's neighbor. This is the neck. The neck's other neighbor is the shoulder. Mark the head, neck, shoulder, and all the neighbors of the shoulder. Take any unmarked vertex. This must be a rear foot, and so we can identify its only neighbor as the hip. At this point, the head, neck, shoulder and hip have been identified, and all the front feet are marked and the rear feet are not marked. From this we can identify all the pieces of the animal in $O(n)$ time, and we are done.

All that remains is to explain how to identify the head. To begin, observe that feet and head have degree one, the neck has degree two, and the shoulder and hip have degree three or higher. Any vertex of degree one (head or feet) is adjacent to a vertex of degree strictly greater than one. If we ever encounter the head, we can verify this by checking that this vertex has degree one, and its only neighbor has degree two.

We first define a procedure (called the *neck test*) to check whether a vertex is the neck. The neck is uniquely identified as having degree two, such that one of its neighbors has degree one. This can be tested in $O(n)$ time. If it succeeds, we will have succeeded in identifying the head, and we are done. Otherwise, the procedure returns in failure.

We will locate the head through a process of elimination. Take any vertex u and compute its degree. If u's degree is equal to one then set u to its unique neighbor. Observe that u must be of degree greater than one. Apply the neck test to u. If it succeeds, we are done. If not, u is either the shoulder or hip. To determine which is the case, we will again apply a process of elimination. We mark u and all of u's neighbors. Select any unmarked vertex and call it v. If v's degree is greater than one, then set v to its unique neighbor. (Now v's degree must be greater than one.) Again we apply the neck test. If successful we are done, and otherwise v must be either the shoulder or hip.

At this point we know that u and v are both either shoulder or hip. Could they be the same? We claim not. If u had been the shoulder, then the only unmarked vertices are the head and rear feet. The initial choice for v could not be the head, since otherwise we would have moved v to the neck and the neck test would have caught this. Therefore v started as a rear foot and then moved to the hip. On the other hand, if u had been the hip, the only unmarked vertices are the head, neck, and front feet. For the same reason, v cannot be the neck, and therefore it follows that v must be the shoulder.

In conclusion, we know that u and v are the hip and shoulder, but we don't yet know which is which. Mark both u and v and all their neighbors. The only unmarked vertex remaining is the head, and we are done.

Homework 3: Kruskal, Dijkstra, and Dynamic Programming

Handed out Tue, Mar 11. Due at the start of class Thu, Mar 27. Late homeworks are not accepted, but you may drop your lowest homework score.

As always, if you are asked to present an algorithm, in addition to the algorithm you must justify its correctness and derive its running time.

Problem 1. Suppose that you wish to route flow through a network of pipes. We model the network as a connected, undirected graph $G = (V, E)$, in which each edge has a numeric value $c(u, v)$, which represents the edge's *capacity*, that is, the amount of flow it can take. Given any path $P = \langle u_1, u_2, \ldots, u_k \rangle$, its *capacity* is defined to be the minimum capacity of any edge on the path, that is $cap(P) = \min(c(u_1, u_2), c(u_2, u_3), \ldots, c(u_{k-1}, u_k))$. By convention, $c(u, u) = \infty$. For every $u, v \in V$, define $cap(u, v)$ to be the maximum capacity over all paths from u to v.

(a) (Single-Source Capacity) Given a source vertex $s \in V$, present an algorithm that computes $cap(s, u)$ for all $u \in V$. Your algorithm should run in $O(E \log V)$ time. (Hint: One solution that I know of involves modifying Dijkstra's algorithm.)

(b) (All-pairs capacity) Present an algorithm that computes $cap(u, v)$ for all $u, v \in V$. Your algorithm should run in $O(V^3)$ time. (Hint: One solution that I know of involves modifying the Floyd-Warshall algorithm.)

In both cases it is sufficient just to compute the capacity of the paths. It is not required to compute the actual paths.

Note: In both cases, be sure to establish the correctness of your algorithms. It is not sufficient to simply appeal to the correctness of related algorithms like Dijkstra's or Floyd-Warshall. Capacity is fundamentally a different optimization criterion than weighted path length, and so the proofs of these other algorithms will need to be modified to work in this context.

Problem 2. On the planet of Smurf, the adorable, gentle Smurf folk hold a convention every four years. In case you don't know, Smurf's never die, and they are all descended from a common ancestor, the venerable "Papa Smurf." It is ancient Smurf tradition that for every father-son pair at least one of the two must attend the convention, and both may attend. This means that, if a Smurf decides to stay home, his father and all his children are required to attend. Assume that you are given n Smurfs, named Smurf-1 through Smurf-n. For $1 \leq i \leq n$, let T_i denote the travel cost of sending Smurf-i to the convention. You may assume the Smurfs have been numbered, so that if Smurf-i is the father of Smurf-j, then $i < j$. (Thus, Papa Smurf is Smurf-1.)

Give an algorithm to determine (1) the minimum total travel costs for all the Smurfs, subject to the above attendance requirement, and (2) output the set of Smurfs attending the convention (see Fig. 1). The inputs to your algorithm are two arrays C and T, where $C[i]$ is the list of the children of Smurf-i, and $T[i]$ is the associated travel cost.

Derive the running time of your algorithm. ($O(n)$ time is possible.) For example, in Figure 1, we give a sample example. The distances are listed next to each node of the Smurf family tree. The optimal solution is to send Smurfs 1 and 2 to the meeting, for a total transportation cost of $5.2 + 7 = 12.2$.

Hint: I know a few different solutions. One involves dynamic programming and another is a greedy solution. You are free to present any $O(n)$ time algorithm. In any case, be sure to show your algorithm's correctness.

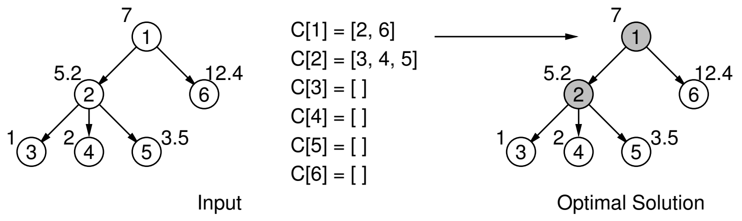


Figure 1: Example for Problem 2. Next to each node of the tree is its travel cost.

Problem 3. In this problem we will consider two variations of the longest common subsequence problem. You are given three sequences X, Y, and Z, where $m = |X|$, $n = |Y|$ and $|Z| = k$. You want to know whether the sequences X and Y can be shuffled together (without changing the relative order of elements within either sequence) in order to form the sequence Z. For example, if $X = \langle ABC \rangle$, $Y = \langle BACA \rangle$, and $Z = \langle ABBACCA \rangle$ the answer is "yes," ($Z = \langle A_x B_x B_y A_y C_x C_y A_y \rangle$) but if $Z = \langle ABCBAAC \rangle$ the answer is "no." You may assume that $k = m + n$, for otherwise answer is definitely "no."

(a) The eminent but flaky Professor Hubert J. Farnsworth[1] claims that the following simple algorithm works for this problem. First, compute the LCS of X and Z, and remove these elements from Z. Let Z' be the resulting sequence. If $Y = Z'$ then the answer is yes, and otherwise, the answer is no. Is Farnsworth's algorithm correct? Either prove that it is or present a counterexample.

(b) Present an $O(nm)$ time algorithm that answers this decision problem. (Whether right or wrong, you cannot reuse Farnsworth's algorithm.)

(c) Let us convert this decision problem into an optimization problem, by computing the smallest number of deletions from any of X, Y, or Z to make such a shuffle possible. For example, if $X = \langle ABC \rangle$, $Y = \langle BACA \rangle$, and $Z = \langle ABCBAAC \rangle$, then by deleting the last character of Y and the second to last character of Z, such a shuffle is possible. Thus, the answer would be 2. The running time of your algorithm should be $O(mnk)$ where $m = |X|$, $n = |Y|$, and $k = |Z|$. (Because characters can be deleted, you should not assume that that $k = m + n$.) Your algorithm should not only compute the minimum number of deletions needed, but it should also output the set of deletions to be performed.

Problem 4. A pizza delivery boy needs to make a series of deliveries. He is given the coordinates $p_i = (x_i, y_i)$ of a collection of n points $\{p_1, p_2, \ldots, p_n\}$. You may assume that the points are given in increasing order of x-coordinate, that is, $x_1 < x_2 < \ldots < x_n$. His pizza parlour is located at p_1, and his route is subject to the restriction that he first travels strictly from left to right (along increasing x-coordinates) until getting to p_n, and then he returns to p_1 strictly from right to left (along decreasing x-coordinates). (See Fig. 2.) Present an efficient algorithm that computes the cost of the shortest delivery tour that visits all the points subject to the above left-right-left restriction. (You do not need to compute the path, just the cost.) You may assume that you have access to a function $dist(i, j)$, which returns the distance from point p_i to p_j.

Hint: Use DP. Rather than thinking of a outbound and inbound path, consider a formulation that builds two paths, both starting at p_1 and both traveling from left to right, such that each point will either be added to one path or the other. In my formulation, I used two variables i and j, where one path ended at point i and the other ended at point j. For full credit, your algorithm should run in $O(n^2)$ time. I suspect that there is a somewhat simpler $O(n^3)$ time algorithm.

[1]Noted inventor of the *nose tuba*, a brass instrument that can be played with the nose. The nose tuba achieved notoriety as the only musical instrument to be featured in an article of the International Journal of the Medical Malpractice, because of its tendency to cause musicians to blow their brains out while attempting the tricky second movement of Mendelssohn's Tuba Concerto in A minor.

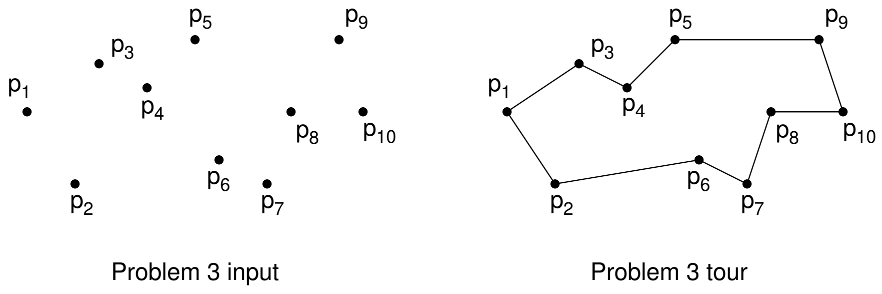


Figure 2: Problem 4.

Challenge Problem. 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.

The following two problems are similar in spirit, and both solutions involve a similar trick. You can do either one or both (or neither!).

(a) You are given a pointer to the head of a linked list of unknown length. Each node of the link list contains nothing but a pointer to the next node in the list. Either the linked list terminates in a null pointer, or else it loops back on itself (see Fig. 3(a).) Write an algorithm that determines which is the case. You are not allowed to alter the contents of any of the nodes of the linked list, and you only have access to a constant number of simple variables (no arrays or dynamic memory allocation allowed). Your algorithm should run in $O(n)$ time, where n is the (unknown) number of elements in the linked list. (For maximum credit you do not need any integer variables of more than constant size. My solution requires only two pointer variables and a boolean.)

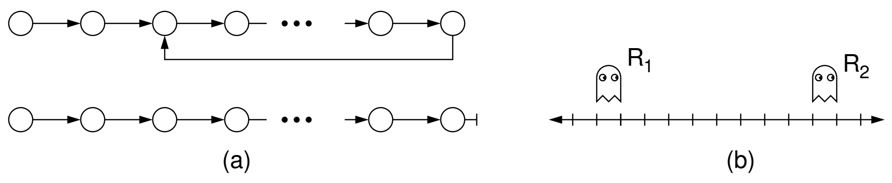


Figure 3: Challenge Problem.

(b) You have two robots R_1 and R_2 that can move along the integer line (which is infinite in both directions). The robots start at different positions along the line, but you have no idea what these positions are (see Fig. 3(b)). (In particular, a robot does not know which position it is in, it does not know whether it is to the left or right of the other robot, and it does not know how far away the other robot is.)

Write a program to help the robots find each other. The two robots are synchronized and execute one step every second. As a robot I may do any of the following things at each step of my program: move myself left/right one position, mark the spot that I am on (this can be done only once per robot), test whether the spot I am on has been marked, and test whether the other robot is currently on my spot. The robots have only a finite amount of local memory. Note that the initial distance between the two robots may be much much larger than any number that can be stored in the local memory.

Hints: It is possible to have both robots execute exactly the same program. Marks cannot be erased. Don't try to count, you don't have enough local memory. The running time should be $O(n)$, where n is the (unknown) distance between the robot's initial positions.

Solutions to Homework 3: Kruskal, Dijkstra, and Dynamic Programming

Solution 1: I will discuss two approaches, one based on modifying Dijkstra's algorithm and the other based on modifying the Floyd-Warshall algorithm.

(a) (Single-source) The algorithm is very similar to Dijkstra's algorithm. For each vertex u, we maintain a capacity estimate cap$[u]$, which stores the best known capacity from s to u, based on the vertices we have processed so far. Initially cap$[u] = -\infty$, indicating that we know of no path. (If we assume capacities are positive, we could have also used cap$[u] = 0$.) The key difference is the relaxation rule. Given an edge (u, v), if know that there is a path of capacity cap$[u]$ going from s to u, then we know that there is a path of capacity at least $\min(\text{cap}[u], c(u, v))$ going from s to v. One other difference with Dijkstra's algorithm is that we assume that we have a reverse order priority queue in which we can perform the operations extractMax and increaseKey in $O(\log V)$ time, and we also use the above modified relaxation rule. It is presented in the code block below.

Maximum Capacity Paths (Single-Source)

```
MaxCapSingleSource(G, c, s) {
    for each (u in V) {                    // initialization
        cap[u] = -infinity; pred[u] = null ;
    }
    cap[s] = +infinity                     // capacity to self is inf
    Q = new PriorityQueue(V)               // put all vertices in Q
    while (Q is nonEmpty) {                // until all vertices processed
        u = extractMax from Q;             // select u closest to s
        for each (v in Adj[u]) {
            if (min(cap[u], c(u, v)) > cap[v]) { // Relax(u, v)
                cap[v] = min(cap[u], c(u, v));
                Q.increaseKey(v, cap[v]);  // update v's position in Q
                pred[v] = u;
            }
        }
        color[u] = black
    }
    [The pred pointers define an "inverted" max capacity path tree]
}
```

The running time is clearly the same as Dijkstra's algorithm (since it is structurally identical). For each vertex $v \in V$, let $\gamma(v)$ denote the capacity of the maximum capacity path from s to v. To establish the algorithm's correctness, we argue as we did in Dijkstra's algorithm that when a vertex u comes out of the queue, we have cap$[u] = \gamma(u)$ is correct. Suppose that this were not the case, and let u be the first vertex for which cap$[u] \neq \gamma(u)$. Since we only generate capacities based on paths that we known about, it follows that cap$[u] < \gamma(u)$. Consider the true maximum capacity path from s to u. Since u is not yet processed, its color is still white, and hence there is an edge (x, y) where the path first goes from a vertex x that is black to a vertex y that is white. Since u was the first vertex for which we made a mistake, we know that cap$[x] = \gamma(x)$, and since we performed relax on (x, y), it follows that cap$[y] = \gamma(y)$ is correct. Because maximum capacity path from s to u passes through y, it follows that $\gamma(y) \geq \gamma(u)$. Thus, we have cap$[y] = \gamma(y) \geq \gamma(u) >$ cap$[u]$.

(b) (All-pairs) The algorithm is an almost trivial variant of the Floyd-Warshall algorithm where $+$ is replaced by min and min is replaced by max. Let us first define cap$^{(k)}[i, j]$ to be the maximum capacity path from i to j, where the intermediate vertices along the path are allowed to pass only through the vertices in the set $\{1, 2, \ldots, k\}$. The DP formulation is:

$$\text{cap}^{(0)}[i,j] \;=\; c(i,j) \qquad\qquad \text{cap}^{(k)}[i,j] \;=\; \max\begin{cases} \text{cap}^{(k-1)}[i,j] \\ \min(\text{cap}^{(k-1)}[i,k], \text{cap}^{(k-1)}[k,j]) \end{cases}$$

To establish the algorithm's correctness, we apply the same sort of argument that we did in the case of Floyd-Warshall algorithm. For the basis case for $i \neq j$, we have $\text{cap}^{(0)}[i,j] = c(i,j)$, since there are no intermediate vertices and hence the only option is to use the direct edge. Let $\text{cap}^{(0)}[i,i] = +\infty$, which reflects the fact that there is no capacity constraint from a vertex to itself. Now, assume inductively that for $1 \leq k \leq n$, we have already computed $\text{cap}^{(k-1)}[i,j]$ for all i and j, and we want to extend this to $\text{cap}^{k}[i,j]$. As in the Floyd-Warshall algorithm we observe that there are two possibilities. The above formulation arises by taking the better (i.e., maximum) of these two alternatives.

The max capacity path from i to j does not go through k: The cost is just $\text{cap}^{(k-1)}[i,j]$.

The max capacity path from i to j goes through k: There is no advantage in having the path revisit k a second time (since inserting a loop into any path can only decrease its capacity), and so the paths from i to k and from k to j do not visit vertex k. Since the capacity is the minimum of any edge on the path the final capacity is $\min(\text{cap}^{(k-1)}[i,k], \text{cap}^{(k-1)}[k,j])$.

Finally, as in Floyd-Warshall, we note that there is no harm in dropping the superscript k and computing everything in-place. The only entries of the array that could be polluted by overwriting are of the form $\text{cap}[i,k]$ and $\text{cap}[k,j]$, and it is easy to see that these values do not change during the kth iteration, due to our assumption that $\text{cap}[k,k] + \infty$. The algorithm is presented in the following code block. The running time is clearly $O(n^3)$.

Maximum Capacity Paths (All-Pairs)

```
MaxCapAllPairs(c, n) {
    for (i = 1 to n)
        for (j = 1 to n)
            cap[i, j] = (i == j ? +infinity : c[i, j]);
    for (k = 1 to n)
        for (i = 1 to n)
            for (j = 1 to n)
                cap[i, j] = max(cap[i, j], min(cap[i, k], cap[k, j]));
    return the cap array;
}
```

Solution 2: Let S_i denote the subtree rooted at Smurf-i. Let $G[i]$ be the minimal transportation cost for all the Smurf's in subtree S_i if Smurf-i goes to the meeting, and let $H[i]$ be the minimal transportation cost for this subtree if Smurf-i stays home. We compute the quantities $G[i]$ and $H[i]$ in a bottom-up manner, but compute the final set of attendees in a top-down manner.

In the case of $H[i]$, if Smurf-i stays home, all of his children must attend, and hence the cost is the sum of $G[j]$ for all children j of Smurf-i. In the case of $G[i]$, Smurf-i incurs a travel cost of $T[i]$. His children are free to go or stay home, so we take the sum of the minimum of $G[j]$ and $H[j]$ for each child j. Thus we obtain the following formulation.

$$H[i] \;=\; \sum_{j \in C[i]} G[j] \qquad\qquad G[i] \;=\; T[i] + \sum_{j \in C[i]} \min(G[j], H[j]).$$

Note that there is no need for a special basis case, if we use the standard convention that an empty sum is equal to zero.

To determine which Smurfs attend, we work top down. We maintain for each Smurf a boolean variable *going*$[i]$, which we set to true if Smurf-i is going, and false otherwise. The root (papa Smurf) goes only if $G[1] \le H[1]$. If the root stays home, then his sons are required to go. If the root goes, then his sons have the option of going or not. In such a case, son i will go only if $G[i] < H[i]$. Once we make this determination for all the children, we apply the same reasoning to each of the next lower level of descendents. By our numbering convention, a simple for-loop can be used to to visit nodes in this order. The complete pseudo code is presented in the code block below.

Solution to the Smurf Problem

```
Smurf(n, T) {
    for (i = n downto 1) {              // compute H[] and G[] bottom-up
        H[i] = 0;                       // initialize H and G
        G[i] = T[i];
        for each (j in C[i]) {          // compute H and G from children
            H[i] += G[j];
            G[i] += min(G[j], H[j]);
        }
    }
    for (i = 1 to n) going[i] = false;  // initialize
    for (i = 1 to n) {
        if (!going[i] && G[i] < H[i]) going[i] = true;
        if (!going[i]) {
            for each (j in C[i]) going[j] = true;
    }
    return min(G[1], H[1]);             // final cost of root
}
```

The total running time of the algorithm is proportional to the number of edges in the associated tree. We know that a free tree with n nodes has $n-1$ edges, so the total running time is $O(n)$.

Solution 3:

(a) No, his algorithm is not correct. Consider the sequences $X = \langle B \rangle$ $Y = \langle BC \rangle$ and $Z = \langle BCB \rangle$. The answer should be "yes," since we can append X to Y to obtain Z. However, $\text{LCS}(X, Z) = B$. If we remove the first B from Z, then we are left with $Z' = \langle CB \rangle$, which does not match Y, and hence his algorithm incorrectly answers "no." You might protest: What if Farnsworth's LCS algorithm was designed so that it matched X with the second B of Z not the first. Since his algorithm does not get to see Y, we will let $Y' = \langle CB \rangle$ instead. Again, the answer should be "yes," but with the sequences X, Y', and Z, it will answer "no." Thus, no matter which option Farnsworth's algorithm takes, one of the two instances provides a counterexample.

(b) There is an $O(m+n)$ time solution, based on tentatively merging the two strings X and Y into Z, but we will present a DP solution. For $0 \le i \le m$ and $0 \le j \le n$, let $C[i,j]$ be true if there is a way to shuffle X_i with Y_j to form Z_{i+j}. As the basis case $C[0,0] = \text{true}$, since empty sequences trivially match. To compute $C[i,j]$ in general we observe that if the last characters of Z_{i+j} and X_i match, we may match these characters with one another, and recursively test whether X_{i-1} and Y_j can be shuffled to form Z_{i+j-1}, which is given by $C[i-1,j]$. By a similar argument if the last characters of Z_{i+j} and Y_j match, we may match these characters with one another, and recursively test whether X_i and Y_{j-1} can be shuffled to form Z_{i+j-1}, which is given by $C[i,j-1]$. Thus we have the following formulation. $C[0,0] = \text{true}$ and otherwise:

$$C[i,j] \;=\; \text{true if and only if either} \left\{ \begin{array}{l} C[i-1,j] \text{ if } (i \ge 1 \;\wedge\; x_i = z_{i+j}) \\ C[i,j-1] \text{ if } (j \ge 1 \;\wedge\; y_j = z_{i+j}). \end{array} \right.$$

We will not give the actual program, but it follows in a straightforward manner. The running time is $O(nm)$, since it takes $O(1)$ time to fill each entry of the matrix.

(c) We define a matrix $C[0..m, 0..n, 0..k]$, such that $C[i, j, p]$ is the minimum number of changes to shuffle X_i with Y_j to form Z_p. We adopt an approach similar to the one above. If the last character of Z_p matches either the last character of X_i or Y_j, then we match them and recurse. Otherwise, we must skip one of the three characters. Since we don't know which, we try all three. Thus we have the following formulation. The code block below implements this formulation. $C[0, 0, 0] = 0$, and otherwise:

$$C[i,j,p] \quad = \quad \min \begin{cases} C[i-1,j,p-1] & \text{if } (i \geq 1 \ \wedge \ z_p = x_i) \\ C[i,j-1,p-1] & \text{if } (j \geq 1 \ \wedge \ z_p = y_j) \\ 1 + C[i-1,j,p] & \text{if } i \geq 0 \\ 1 + C[i,j-1,p] & \text{if } j \geq 0 \\ 1 + C[i,j,p-1] & \text{if } p \geq 0. \end{cases}$$

Shuffling Problem Solution

```
ShuffleCost(X, Y, Z) {
    C[0, 0, 0] = 0;
    for (i = 1 to m) {
        for (j = 1 to n) {
            for (p = 1 to k) {
                Cmin = +infinity;
                Dmin = null;
                if (i >= 1 && Z[p] == X[i]) update((Cmin, Dmin), (C[i-1, j, p-1], X));
                if (j >= 1 && Z[p] == Y[j]) update((Cmin, Dmin), (C[i, j-1, p-1], Y));
                if (i >= 0) update((Cmin, Dmin), (1 + C[i-1, j, p], skipX));
                if (j >= 0) update((Cmin, Dmin), (1 + C[i, j-1, p], skipY));
                if (p >= 0) update((Cmin, Dmin), (1 + C[i, j, p-1], skipZ));
            }
        }
        C[i, j, p] = Cmin;
        D[i, j, p] = Dmin;
    }
}
```

In order to extract the final shuffle, we maintain a parallel array $D[i, j, p]$ whose entries are the decisions made, which may be either X (meaning match the ith character of X with the pth character of Z), Y (meaning take the jth character of Y with the pth character of Z), skipX (meaning skip the ith character of X), skipY (meaning skip the jth character of Y) and skipZ (meaning skip the pth character of Z). We make use of a utility function update$((c, d), (c', d'))$, which is given the current cost-decision pair (c, d) and compares it to the current cost-decision pair (c', d') and sets $(c, d) \leftarrow (c', d')$ if $c' < c$. To extract the final sequence, we recursively traverse the D array backwards from the end back to the start. The initial call is to ShuffleSolution(D, m, n, k), given in the code block below.

Solution 4: We solve the problem through the application of dynamic programming. We construct an array $C[n, n]$. We only use entries $C[i, j]$, where $i \leq j$. The value of $C[i, j]$ is total length of the shortest pair of paths that start at p_1, and traverses all the points $\{p_1, p_2, \ldots, p_i, p_j\}$, such one path ends at point p_i and the other path ends at point p_j. Note that in this formulation we do not visit any of the vertices $\{p_{i+1}, p_{i+2}, \ldots, p_{j-1}\}$. This is not critical to the correctness of the method, but I found that it makes the analysis a bit easier. By symmetry, we may assume $i \leq j$, since $C[i, j] = C[j, i]$.

For the basis case, we set $C[1, j] = \mathit{dist}(1, j)$, since the only point being hit is the jth. To compute $C[i, j]$ for $i \geq 2$, there are two cases. Clearly, one of the two paths must visit the point p_{i-1}. First, if the path from

Shuffling Problem Solution

```
ShuffleSolution(D, i, j, p) {
    switch (D[i, j, p]) {
    X:     ShuffleSolution(D, i-1, j, p-1);
           output "match X[i] with Z[p]"; break;
    Y:     ShuffleSolution(D, i, j-1, p-1);
           output "match Y[j] with Z[p]"; break;
    skipX: ShuffleSolution(D, i-1, j, p); break;
    skipY: ShuffleSolution(D, i, j-1, p); break;
    skipZ: ShuffleSolution(D, i, j, p-1); break;
    }
}
```

p_1 to p_i passes through p_{i-1}, then the path structure consists of one path from p_1 to p_j, the other path from p_1 to p_{i-1} followed by the direct edge (p_{i-1}, p_i). Second, if the path from p_1 to p_j passes through p_{i-1}, then the path structure consists of one path from p_1 to p_i, the other path from p_1 to p_{i-1} followed by the direct edge (p_{i-1}, p_j). Here is the final recursive rule:

$$C[i,j] \;=\; \min(C[i-1,j] + \mathit{dist}(i-1,i), C[i-1,i] + \mathit{dist}(i-1,j)).$$

To obtain the optimum result, we take the smaller of these two. We will not discuss recovery of the optimal path, but intuitively the way to do it is to maintain a parallel matrix P, and set $P[i,j]$ to either 0 or 1, depending on which of the two above cases occurs. We can then start at $P[n,n]$ and work our way backwards. When we visit $P[i,j]$, we prepend vertex $i-1$ to the same path as j if $P[i,j]=0$ and we prepend $i-1$ to the opposite path otherwise.

Solution to the Challenge Problem:

(a) There is simple, but tricky solution. The idea is to think of the linked list as a race track, and we will have two pointers race around this track. One runs twice as fast as the other. (Each time through the loop, the fast racer advances once, and the slow racer advances every other time through the loop.) If the list ends with a null pointer, then the faster racer will discover this first, and will discover it in $O(n)$ time. If the list contains a loop, then the faster racer will eventually pass the slower racer. When this happens, we announce that the list contains a loop. See the code block below.

Solution to the Challenge Problem (a)

```
checkList(head) {
    fast = slow = head;                   // racers start together
    doStep = true;
    while (true) {
        fast = fast->next;                // advance the fast racer
        if (doStep) slow = slow->next;    // advance slow every other time
        doStep = !doStep;                 // next time don't step
        if (fast == null) return "no loop"; // fast hits end of list
        if (fast == slow) return "looped";  // fast overtakes slow --> loop
    }
}
```

The question is: how long will this take? We claim that it will occur in $O(n)$ time. Let k denote the number of entries in the looped portion ($1 \le k \le n$). It will take the slow racer $2(n-k)$ steps to even reach the loop. Once it does, after the next $2k$ stages, the fast racer will go around the loop exactly twice, and the slow racer will go around the loop exactly once. It follows that no matter where the two started, the fast racer must pass the slow racer at least once before the $2k$ stages are over. Thus, the number of total stages before discovering the loop is at most $2(n-k)+2k = 2n \in O(n)$.

(b) We employ a trick that is similar to the one used in part (a). Both robots start by leaving a mark at their starting position, and then they move to the right. We know that one of the two robots (the one initially on the left) will eventually hit the mark left by the other robot (the one initially on the right). We cannot make the other robot stop, so what we would like to do is to make the left robot move at double speed, implying that it will eventually overtake the slower moving one. Unfortunately, there is no instruction to allow us to do this, so we start both robots off moving at half speed, by moving a unit to the right only every other time step. Then, in order to double our speed, we simply move every step, rather than every other step. The program for both robots is presented in the code block below. Note that we only need two bits of internal storage to store the two boolean variables *doStep* and *goFast*.

Solution to the Challenge Problem (b)

```
findRobot() {
    mark this spot;
    doStep = true;
    goFast = false;
    repeat {
        if (the other robot is on this spot) exit;
        if (goFast || doStep) {
            move right one unit;
            if (this spot is marked) {
                goFast = true;
            }
        }
        doStep = !doStep;
    } (forever);
}
```

To analyze this algorithm's correctness and running time, suppose that the robots are initially n units apart. If $n = 0$ then we will discover this as soon as we enter the repeat loop. If $n > 0$, then after $2n-1$ iterations, the left robot will discover the mark left by the right robot. During this time, both robots have moved n positions to the right, and so they are still separated by n units. Now, if n is even, after an additional $2n$ iterations the slow (right) robot will move exactly n additional steps to the right, and the fast (left) robot will move exactly $2n$ steps, which exactly closes the gap. If n is odd, after an additional $2n+1$ steps, the slow (right) robot will move exactly $n+1$ steps to the right, and the fast (left) robot will move exactly $2n+1$ steps, again exactly closing the gap. Thus the total number of iterations before termination is at most $(2n-1)+(2n+1) = 4n = O(n)$.

CMSC 451:Spring 2008 Dave Mount

Homework 4: Network Flow and NP-Completeness

Handed out Tue, Apr 15. Due at the start of class Tue, Apr 29. Late homeworks are not accepted, but you may drop your lowest homework score.

As always, if you are asked to present an algorithm, in addition to the algorithm you must justify its correctness and derive its running time.

Problem 1. The eminent but flaky Professor Hubert J. Farnsworth[1] has invented an alternative network flow algorithm. His algorithm is similar to Ford-Fulkerson, but Farnsworth's algorithm only *increases* flows along edges that have not yet reached their capacity—it never reduces flow on any edge by pushing flow along backward edges of the residual network.

- **(For partial credit)** Show that Farnsworth's algorithm is *not* optimal by presenting a counterexample, which consists of an s-t network G such that his algorithm terminates with a flow that is strictly less than the true maximum flow. You are free to select the augmenting paths however you like in forming your counterexample.
- **(For full credit)** Show that Farnsworth's algorithm is not merely suboptimal but can be *arbitrarily bad*. In particular, given any positive integer $b > 1$, show how to construct an s-t network G such that the ratio between the true maximum flow in G to Farnsworth's flow at least as large as b. (The size of your network and its capacities are allowed to be functions of b.)

Be sure to explain how your counterexample works.

Problem 2. Describe a polynomial time algorithm for the following problem. You are given a flow network $G = (V, E)$ with source s and sink t, and whose edges all have unit capacity, that is, $c(u, v) = 1$ for all $(u, v) \in E$. Your algorithm is also given a parameter k ($1 \leq k \leq |E|$). The objective is to delete k edges from G so as to reduce the maximum s-t flow value as much as possible. In other words, compute a subset $E' \subseteq E$ of size k such that the flow in $G' = (V, E - E')$ is as low as possible.

Justify your algorithm's correctness and derive its running time.

Problem 3. You are the manager of the computing staff at Mega State University, and one of your less pleasant duties involves assigning your staff to be on-call during holiday periods. There are n people in your staff and there are m holiday periods. Based on your calendar, you know that the jth holiday runs for $d[j]$ days. (You may assume that no two holidays overlap in time.) Here are the constraints that you must abide by:

- There must be one staff member on-call during every day of every holiday period.
- According to union rules, no staff member may be on-call for more than c total days over all the holiday periods.
- According to union rules, no staff member may be on-call for more than one day during any one holiday period.
- Each staff member $1 \leq i \leq n$ has given you a list $H[i]$ of holiday periods when they are available. (You may assume that if a person is listed as available for some holiday, he/she is available for any day of the holiday period.)

Describe a polynomial time algorithm, which given $d[1..m]$, c, and $H[1..n]$, generates an assignment of staff members to holidays. If no such assignment is possible, your algorithm should indicate this. Prove the correctness of your algorithm and give its running time.

[1]Famous inventor of the *laser-guided ketchup dispenser*, a device capable of squirting ketchup to within 2.5 microns of any desired spot on your hamburger. (The GPS-enabled version cannot be sold to minors, except in Texas.)

Problem 4. Consider a standard instance of the network flow algorithm, that is, an s-t network $G = (V, E)$, where each edge (u, v) has a nonnegative integer capacity $c(u, v)$. Present a polynomial time algorithm which determines whether G's minimum s-t cut is *unique*. For example, the network shown in Fig. 1 has two distinct minimum cuts, each of capacity 11.

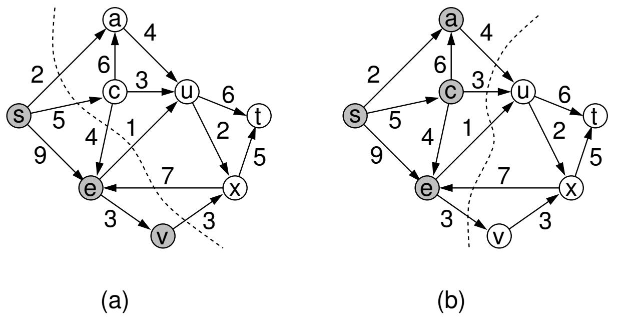


Figure 1: A network with (at least) two distinct minimum s-t cuts, each of value 11.

Justify your algorithm's correctness and derive its running time.

Hint: One approach involves invoking a network flow algorithm as a black-box (either applied to G directly or to some modification of G) followed by a subsequent analysis of the flow and/or the residual network. This process may be repeated a number of times.

Problem 5. This problem involves the observation that a solution to a decision problem can often be used to solve a related optimization problem.

Given an undirected graph $G = (V, E)$, it is said to be *3-colorable* if there is an assignment of the labels $\{1, 2, 3\}$ to the vertices of G such that no two adjacent vertices have the same label. (It is known that deciding 3-colorability is an NP-complete problem.) Suppose that (by some miracle) you have access to a procedure `is3Colorable(G)` which, given an undirected graph G, returns *true* if G is 3-colorable and *false* otherwise.

Using this procedure as a black box, present an algorithm which, given a undirected graph G determines whether G is 3-colorable, and if so outputs a valid coloring of the vertices. Your algorithm should run in polynomial time, and can make a polynomial number of calls to `is3Colorable`.

Hint: Consider how to make a series of modifications to the graph which successively restrict the possible ways of choosing colors, but without changing the fact that G is 3-colorable.

Challenge Problem. Some friends of yours who watch lots of movies have invented a new game. You start with a set X of n actresses and a set Y of n actors, and two players P_0 and P_1. Player P_0 names an actress $x_1 \in X$, player P_1 names an actor $y_1 \in Y$ who has appeared in a movie with x_1, player P_0 then names an actress x_2 who has appeared in a movie with y_1, and so on. Thus, P_0 and P_1 collectively generate a sequence $\langle x_1, y_1, x_2, y_2, \ldots \rangle$ such that each actor/actress in the sequence has appeared with the actress/actor immediately preceding. A player P_i $(i = 0, 1)$ loses when it is P_i's turn to move, and he/she cannot name a member of his/her set who has not been named before.

Suppose you are given a specific pair of such sets X and Y, with *complete information* on who has appeared in a movie with whom. A *strategy* for player P_i, in our setting, is an algorithm that takes a current sequence $\langle x_1, y_1, x_2, y_2, \ldots \rangle$ and generates a legal next move for P_i (assuming it is P_i's turn to move). Give a polynomial-time algorithm that decides which of the two players can force a win, in a particular instance of this game.

Hint: The material from Section 7.5 of KT on bipartite matchings, and Hall's Theorem in particular, may be useful.

Solutions to Homework 4: Network Flow and NP-Completeness

Solution 1: Our method for doing this is to generate a network which has many disjoint paths each of which can carry flow, but this network has a single s-t path that slashes through all these paths. By choosing this path first, it is impossible to route more flow through the network.

See the network shown in Fig. 1 below. In addition to s and t, the network consists of b pairs of vertices arrayed in the zig-zag pattern shown in the figure. All edges have a capacity of 1. If by bad luck, Farnsworth's algorithm augments one unit of flow along the zig-zag path shown in part (b), then it is impossible to route any more flow in this network, which yields a flow of value 1. On the other hand, if flow is routed through each of the b vertical edges, we achieve a flow of value b. Thus the ratio between Farnsworth's flow and the optimal is b.

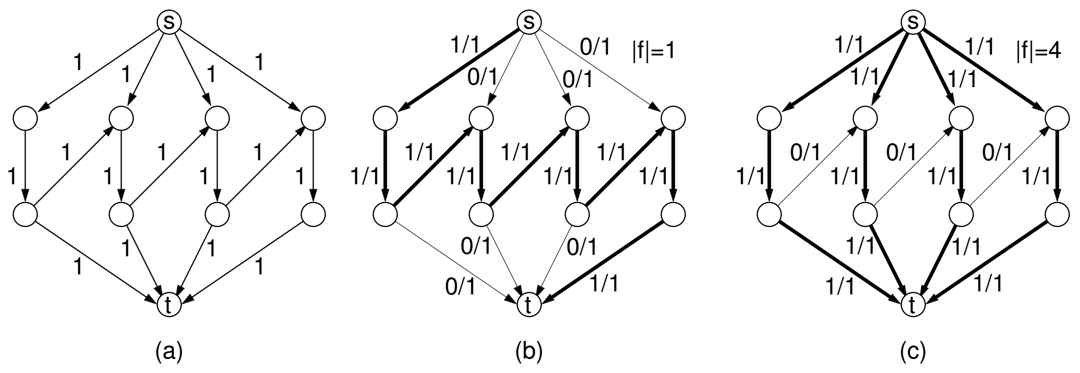


Figure 1: Solution to Problem 1.

Solution 2: We begin by observing that the removal of any k edges of unit capacity from G can reduce the network flow by at most k units. Further, if these edges are removed from the minimum cut, the flow will decrease by *exactly* k units. To see this, recall from the Max-Flow, Min-Cut Theorem that the value of the maximum flow is equal to the capacity of the minimum cut. Removing any edge of unit cost decreases the capacity of any cut by at most one, and so it reduces the flow by at most one unit. Let (X, Y) denote the minimum cut of G. Removing an edge (u, v), where $u \in X$ and $v \in Y$ clearly reduces the capacity of this cut by exactly one unit, and so it reduces the flow by exactly one unit. Also observe (and this is important) that if (X, Y) was a minimum capacity cut prior to removal of the edge, then it will be a minimum capacity cut in the modified network after removal. (Again the reason is that the capacity of (X, Y) decreases by one due to the removal, and no other cut's capacity by decrease by more.) Thus, removing k edges from the minimum cut decreases the capacity of the minimum cut by exactly k units, and so decreases the maximum flow by exactly k units.

This establishes the correctness of the following algorithm. First, compute the minimum s-t cut in G, call it (X, Y), and then eliminate any k edges from X to Y along this cut. (If there are fewer than k such edges, delete them all, thus reducing the flow to 0.) This involves one invocation of network flow and a simple DFS suffices to compute the cut. (Recall from the proof of the Max-Flow, Min-Cut Theorem we can let X be the set of vertices reachable from s, and let $Y = V - X$. This can be done in $O(V + E)$ time through DFS.) Thus, the overall running time is $O(V^3) + O(V + E) = O(V^3)$.

Solution 3: $N = \{1, \ldots, n\}$ be the staff members, and let $M = \{1, \ldots, m\}$ be holiday periods. Create an s-t network $G = (V, E)$ where $V = N \cup M \cup \{s\} \cup \{t\}$. Create a unit capacity edge (i, j) if staff member

$i \in N$ is available during holiday $j \in M$. Create an edge from s to each $i \in N$ with capacity c and create an edge from $j \in M$ to t with capacity $d[j]$.

Compute the maximum (integer) flow in this network. We claim that a feasible schedule exists if and only if all the edges entering t are saturated. To prove the "only if" part, we consider the following mapping between any valid schedule and a flow in G. If worker i is scheduled to be on-call during some day of holiday j, we apply one unit of flow along the edge (i, j). Since worker i is not scheduled to be on-call more than one day of this schedule, one unit of flow suffices. Since worker i is not scheduled to be on-call more than c holidays over the year, the flow along the edge (s, i) can be set to the total number of holidays this worker is on-call, which will not exceed this edge's capacity of c. Finally, observe that since each of the $d[j]$ days of holiday j has one worker on-call, there are $d[j]$ edges of unit flow entering into vertex j, and so we can apply $d[j]$ units of flow onto the edge (j, t), thus saturating this edge. Observe that the resulting flow satisfies flow balance and the capacity constraints, and so is a valid flow.

To prove the "if" part, suppose that G has a flow that saturates all the edges coming into t. We show how to map this to a valid schedule. Because of the capacity c on edges (s, i), worker i is not on-call for more than c holidays, thus satisfying the union rules. Since there is unit capacity on edge (i, j) and this edge is only present if worker i is available for holiday j, worker i is only scheduled for holidays he/she is available and only for one day of such a holiday. Since the edge (j, t) is saturated, every day of each holiday has exactly one person assigned to it. To produce the final schedule, we just take the workers allocated to (having flow into) each holiday vertex and schedule them (in any order) to the days of this holiday period.

The running time of the algorithm is dominated by the time for the network flow, which is $O(V^3)$, where $|V| = m + n$.

Solution 4: There is a fairly easy solution which involves $O(E)$ invocations of network flow. We first apply network flow to compute the max flow f, and (as seen in the Max-Flow, Min-Cut Theorem) we can use this to compute a min cut, call it (X, Y). For each edge that crosses the cut, we increase its capacity by any positive amount and recompute the network flow. If the flow does not increase, it must be because there is some other cut that is also constraining the flow, and we terminate and announce that the flow is not unique. We repeat this test for every edge crossing the cut. If the flow increases in every case, then we conclude that the minimum cut is unique. Naively, this would involve $O(E)$ invocations of the network flow algorithm. (Since only one edge has changed its capacity, this can be done more efficiently.)

Rather than dwell on the above approach, we present a simpler and more direct alternative, which involves only one application of network flow. Suppose that we have computed the max flow f in G. By the Max-Flow Min-Cut Theorem, $|f|$ is the capacity of any minimum cut. Let S be the subset of vertices that are reachable from s in the residual network G_f. In the proof of the Max-flow Min-cut Theorem we showed that $(S, V - S)$ is an s-t cut of value $|f|$, because every edge of G that crosses this cut from S to $V - S$ must be saturated, and clearly $s \in S$ and $t \in V - S$. Next, let T be the subset of vertices that can reach t in G_f. By a symmetrical argument $(V - T, T)$ is also an s-t cut of value $|f|$. If $(S, V - S) \neq (V - T, T)$, then since both are cuts of value $|f|$, we can answer immediately that the minimum cut is not unique. We claim that not only is is this a sufficient condition, but a necessary one as well.

To establish that the equality of these cuts is a necessary condition for the minimum cut to be unique, let us suppose to the contrary that $(S, V - S) = (V - T, T)$, but that there is some other cut (X, Y) of capacity $|f|$. Observe first that $(S, V - S) = (V - T, T)$ implies that every vertex u of the residual network G_f is either reachable from s or else it can reach t. The fact that (X, Y) has capacity $|f|$ implies that any edge that crosses this cut from X to Y must be saturated in f. (If not, we could decrease this edge's capacity by some strictly positive amount without altering the flow value. The result would be a cut whose capacity is strictly less than the maximum flow, which would contradict the Max-Flow, Min-Cut Theorem.) The fact that $(S, V - S) \neq (X, Y)$ implies that there is a vertex u that is either in $X - S$ or in $Y - (V - S) = Y - T$. If $u \in X - S$, then s cannot reach u in G_f (because $u \notin S$) and u cannot reach t in G_f (because getting to t would require crossing an edge from X to Y, but we have seen that all these edges are already saturated by f). This contradicts the above observation. The case where $u \in Y - T$ yields a similar contradiction by a symmetrical argument. Therefore, no such distinct cut (X, Y) can exists, which establishes the necessity of the condition.

Thus, we have the following algorithm. First compute the maximum flow in G. Next, compute the cuts $(S, V - S)$ and $(V - T, T)$ defined above. (It is an easy exercise to show that these can be computed in $O(V + E)$ time by DFS.) Finally, if these cuts are distinct from each other (which we can test in $O(V)$ time) we conclude that there are two or more distinct minimum cuts. Otherwise, by the above proof, the minimum cut is unique.

Solution 5: The idea is to attempt to insert as many edges into G as possible subject to the constraint that G remains 3-colorable. The final graph will have the property that two vertices are of the same color if and only if they are not adjacent (since otherwise we would have added the edge between them). Equivalently the complement of the final graph will consist of a collection of three disjoint cliques. These cliques are the color classes. The notation $G + (u, v)$ means the graph G with undirected edge (u, v) added. We will make $O(V^2)$ calls to `is3Colorable` and everything else can be done easily in polynomial time.

3-Colorability

```
Color(G) {                                // 3-color graph G
    if (!is3Colorable(G)) return "not colorable";
    for (each distinct pair {u,v} not in E) {
       if (is3Colorable(G + {u,v}) add {u,v} to G;
    }

    for (i = 1 to 3) {                    // assign color group i
        let u be any uncolored vertex in V;
        color[u] = i;
        for each (v not adjacent to u)      // color all non-neighbors the same
            color[v] = i;
    }
}
```

Solution to the Challenge Problem: Consider the bipartite $G = (V, E)$ where $V = X \cup Y$, where X are the actresses and Y are the actors, and an edge (x, y) denotes the fact that actress x and actor y were in the same film. For the network that we used in the bipartite matching algorithm, where all the edges are given unit capacity, and we create a source node with unit capacity edges into the vertices of X and unit capacity edges from Y to the sink vertex t. Compute the minimum cut (S, T) in this network (e.g., by computing the maximum flow and taking all the vertices reachable from s in the final residual network). Let $X' = X \cap S$. There are two cases, depending on whether X' is empty or no.

($X' = \emptyset$:) We claim that player P_1 has a winning strategy. If X' is empty, then every edge coming into a vertex of X is carrying flow, which implies that every $x_i \in X$ is carrying flow to some matching vertex $y_i \in Y$. (Note that it need not be the case that every vertex of Y has an incoming edge, since there may be more actors than actresses.) Here is how player P_1 wins. Whenever player P_0 selects an actress x_i, P_1 responds with matching actor y_i. Observe that P_1 always has a valid reply a (because each actor in Y can have at most one incoming edge carrying flow) P_1 will never repeat any actor. Eventually, player P_0 will run out of options. (Note that by the rules of the game, if there is perfect matching in the graph, eventually P_0 will lose simply because there are no more actresses left, even though this dos not seem very fair.)

For example, in Fig. 2(a), the sequence of play might be $\langle x_1, y_1, x_3, y_3, x_2, y_2 \rangle$, but now y_2's only neighbors are x_1 and x_2, who have already been listed. So P_0 loses.

($X' \neq \emptyset$:) We claim that player P_0 has a winning strategy. Because X' is nonempty, there exists at least one vertex $x_1 \in X$ that is connected to s by an edge carrying no flow. P_0 begins by selecting the

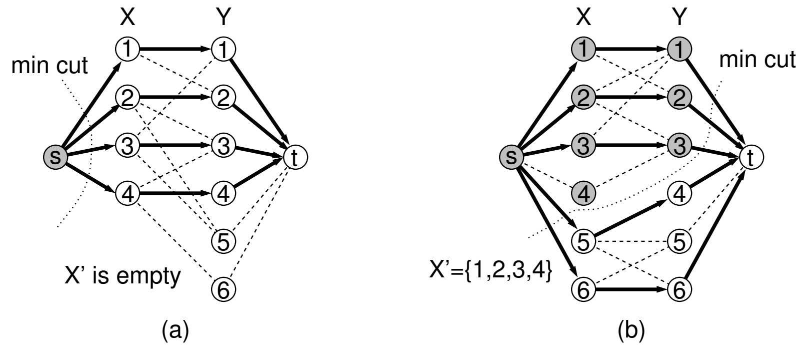


Figure 2: Solution to the Challenge Problem.

actress x_1. Either P_1 loses immediately (because there is no connecting edge in G) or he manages to respond with an adjacent actor $y_1 \in Y$. Observe that y_1 must be carrying flow from some other vertex $x_2 \in X$. The reason is that if y_1 had not been carrying flow, then we could increase the overall flow in the network by routing flow along the augmenting path $\langle s, x_1, y_1, t\rangle$. Next, player P_0 responds with this vertex x_2.

In the next round, either P_1 loses immediately (because there is no edge from x_2 to an unvisited actor) or he manages to respond with some new actor $y_2 \in Y$. Again, we claim that y_2 is carrying flow from some other vertex $x_3 \in X$. If not, then we could push more flow through the network by augmenting along the path $\langle s, x_1, y_1, x_2, y_2, t\rangle$. Player P_0 responds with x_3.

In general, after the ith move by P_0, we see that either P_1 loses immediately or manages to find an actor y_i. This vertex must have flow coming into it from some actress x_{i+1}, for otherwise there would be an augmenting path $\langle s, x_1, y_1, x_2, y_2, \ldots, x_i, y_i, t\rangle$, thus contradicting the fact that we have the maximum flow. Player P_0 responds with actress x_{i+1}. Each time P_0 responds with an actress along which new flow is coming, and so there is never any repetition. Eventually P_1 will run out of options, and P_0 wins.

For example, in Fig. 2(b), the play sequence might be $\langle x_4, y_3, x_3, y_1, x_1, y_2, x_2\rangle$, but the neighbors of x_2 are $\{y_1, y_2, y_3\}$, which have all been previously enumerated, implying that P_1 loses.

Homework 5: NP-Completeness and Approximation Algorithms

Handed out Tue, Apr 29. Due at the start of class Tue, May 13. Late homeworks are not accepted, but you may drop your lowest homework score.

As always, if you are asked to present an algorithm, in addition to the algorithm you must justify its correctness and derive its running time.

Problem 1. The eminent but flaky Professor Hubert J. Farnsworth[1] has stunned the scientific community by announcing that he has proven that P = NP. In particular, he has discovered a polynomial time algorithm for the CLIQUE problem.

For any fixed $k \geq 1$, the k-*CLIQUE* problem is as follows: Given an undirected graph $G = (V, E)$, does it contain a clique of size k? Farnsworth claims that for any $k \geq 1$, the following polynomial time algorithm solves k-CLIQUE.

Input the graph $G = (V, E)$ Let $n = |V|$. Enumerate all subsets of vertices of size k. There are $\binom{n}{k} = O(n^k)$ such subsets. In $O(n^2)$ time it is possible to check whether each such subset is a clique. Thus, in $O(n^{k+2})$ time, we can answer the k-CLIQUE problem. Since any instance of CLIQUE is an instance of k-CLIQUE for some k, CLIQUE can be solved in polynomial time, and therefore P = NP.

What is Farnsworth's mistake?

Problem 2. Consider the following problem, called the *zero-weight cycle* (ZWC). You are given a directed graph $G = (V, E)$ with weighted edges (which may be positive, negative or zero), and the question is whether there exists a simple cycle of total weight 0? (To prevent trivial solutions, we require that any solution consist of at least one edge.) Prove the ZWC is NP-complete. (Hint: Reduction from directed Hamiltonian cycle.)

Problem 3. This is a variant of the Independent Set problem. You are given an undirected graph $G = (V, E)$. We say that a subset $V' \subseteq V$ is a *strongly independent set* if for any two vertices $u, v \in V'$ the edge (u, v) is not in E and there is no path of two edges from u to v. The *Strongly Independent Set Problem* (SIS) is, given an undirected graph $G = (V, E)$ and an integer k, does G have a strongly independent set of size k. Prove that SIS is NP-complete. (Hint: Reduction from Independent Set (IS).)

Problem 4. This problem arises in applications of navigation, where a user wishes to go from one point to another, while escaping detection by avoiding revisiting the same regions twice. The problem is called the *evasive path problem* (EPP). You are given a directed graph $G = (V, E)$, a designated start node s, a target node t, and a collection of pairwise disjoint *zones* $Z_i \subseteq V$, for $1 \leq i \leq k$. The question is whether there exists a path from s to t that visits at most one vertex within each zone. Prove that EPP is NP-complete. (Hint: Reduction from directed Hamiltonian path.)

Problem 5. Suppose you are given an $n \times n$ *grid graph* $G = (V, E)$, as shown in Fig. 1. Each node is associated with a nonnegative integer weight. You may assume that all weights are distinct. Your objective is to compute an independent set $V' \subseteq V$ so that the sum of weights of the vertices of V' is as large as possible. Consider a simple greedy algorithm that works by selecting the vertex u of maximum weight from G and then deleting u and all its neighbors. This process is repeated until there are no more vertices left.

Prove that this algorithm produces an independent set for G whose total weight is at least $\frac{1}{4}$ the total weight of the optimal solution.

[1]Famous inventor of the *solar-powered night light* and recipient of the prestigious award for *Most Idiotic Application of Green Technology*.

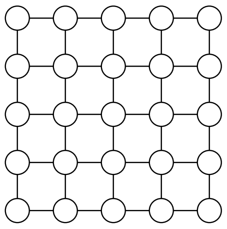


Figure 1: Problem 5.

Challenge Problem. The local video store has received a shipment consisting of a large number of video tapes. Among these is one special tape. Exactly 7 days after viewing this tape the viewer dies a mysterious death without warning or exception. (This might remind you of a scary movie, which came out a number of years ago.)

The video store manager wants to determine which of the tapes is the deadly one before the big sale coming up 8 days from now. He has managed to find a number of foolish people, who are willing to test the tapes for a hefty fee. Each tester will be given some subset of the tapes to view on the first day, and then he/she nervously waits for 7 days for the final results to develop. (There is no limit on the number of tapes that may be assigned to one tester, and each tape may be viewed many different testers.)

Suppose that there are n tapes total. The manager realizes that he can determine the deadly tape by arranging for n testers, each of whom will view one tape. A smart clerk informs him that he can do it with fewer testers. What is the minimum number of tape testers needed to determine which tape is deadly and how is the test conducted? (The stingy store manager does not care how many of the testers dies in the process; he just wants to pay the least number of testers.) If the deadly tape is random among the set of n tapes, then what is the expected number of testers that survive the process?

Solutions to Homework 5: NP-Completeness and Approximation Algorithms

Solution 1: The value of k in the CLIQUE problem is part of the input and is not fixed. Therefore $O(n^{k+2})$ is not polynomial time, as claimed by Farnsworth's proof.

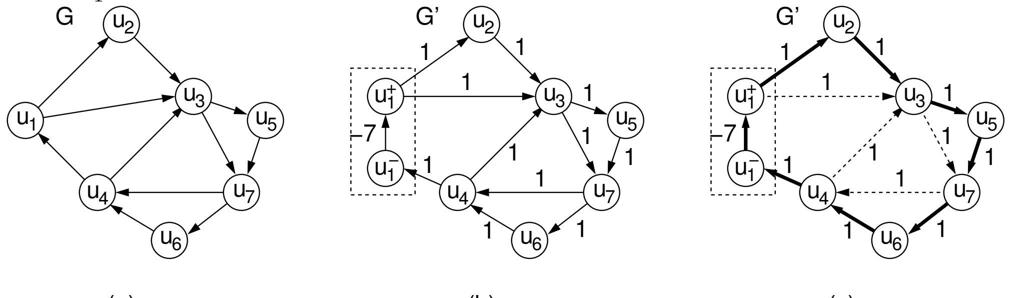
Solution 2: We first show that ZWC is in NP. The certificate is the sequence of vertices in the cycle. We can easily verify that this sequence forms a valid cycle and that its total weight is zero. To show that ZWC is NP-hard, we reduce directed Hamiltonian cycle (DHC) to ZWC. Unlike DHC, the ZWC problem does not require that all vertices be visited, and so we enforce this condition through the use of the weights. If we set the weights of all the edges to 1, then the total weight of a DHC solution will be n, where $n = |V|$. All we need to do is to figure out how to arrange for a bias of $-n$ so that the overall sum is 0.

Here is the reduction more formally. Given the directed graph $G = (V, E)$ for the DHC problem, we first generate the identical directed graph in which all the edges are given a weight of 1. Let u_1 be an arbitrary vertex of G. We replace u_1 with two vertices u_1^- and u_1^+. Each incoming edge $(w, u_1) \in E$ is replaced with the edge (w, u_1^-), and each outgoing edge $(u_1, v) \in E$ is replaced with the edge (u_1^+, v). Finally, we create an edge (u_1^-, u_1^+) of weight $-n$. Let $G' = (V', E')$ denote the resulting digraph. (An alternative approach involves not splitting u_1, but simply weighting all the edges coming into u_1 with a weight of $n - 1$.)

To establish the reduction's correctness we will show that G has a Hamiltonian cycle if and only if G' has a zero-weight cycle. First, suppose that G has the Hamiltonian cycle. Since the vertex visits all vertices, we may assume that it starts at u_1. Let $C = \langle u_1, u_2, \ldots u_n, u_1\rangle$ denote this cycle. Consider the cycle $C' = \langle u_1^-, u_1^+, u_1, u_2, \ldots, u_n, u_1^-\rangle)$. By our construction, all the edges of this cycle exist in G'. The total weight of C' is

$$\begin{aligned} w(C) &= w(u_1^-, u_1^-) + w(u_1^+, u_1) + w(u_1, u_2) + \ldots + w(u_{n-1}, u_n) + w(u_n, u_1^-) \\ &= -n + (1 + 1 + \ldots + 1) \;=\; -n + n \;=\; 0, \end{aligned}$$

as desired. Conversely, suppose that G' has a cycle of total weight 0. Since all but one of the edges is positive, clearly the negative edge (u_1^-, u_1^+) must be used. It's weight is $-n$, which implies that n other edges of weight must be used. Since the cycle is simple, it follows that it must visit every vertex in the graph before returning u_1^+.



(a) (b) (c)

Solution 3: We first show that SIS is NP-complete. The certificate consists of the subset V' of vertices in the SIS. We can compute the distances between each pair of vertices in the graph (e.g., through the use of the Floyd-Warshall algorithm) and then verify that every pair of vertices in V' are at distance three or more from each other.

To show that SIS is NP-hard, we reduce the standard IS problem to it. To do this we need to translate the notion of two vertices being nonadjacent to each other to the notion of two vertices being at distance three or higher. This suggests the idea of inserting a vertex into the middle of each edge, which has the effect of doubling distances in the graph. However, this is not a solution, since there is nothing forbidding us from placing these newly created vertices into the independent set. Our approach to handle this will be to connect all these newly created vertices to each other. The reduction is illustrated in Fig. 1.

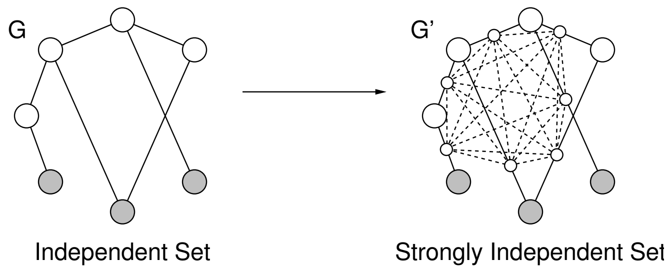


Figure 1: Reduction from IS to SIS.

Here is the reduction. Given the graph $G = (V, E)$ and integer k, we produce a new graph $G' = (W, E')$ and integer k' as follows. The vertices of V are all in W, and in addition, for each edge $\{u, v\} \in E$, we create an *edge-vertex* $w_{u,v}$ for W, and we create the edges $\{u, w_{u,v}\}$ and $\{w_{u,v}, v\}$ for E'. We also generate edge between each pair of edge-vertices. We set $k' = k$, and output the pair (G', k'). The resulting graph has $V + E$ vertices and $O(2E + E^2) = O(E^2)$ edges, and clearly can be constructed in polynomial time.

To establish correctness, we assert that G has an IS of size k if and only if G' has an SIS of size $k' = k$. It will simplify the proof to assume that $k \geq 2$ and that G has no isolated vertices. If $k = 1$, then the answer to both problems is trivially "yes" for any graph. If G has m isolated vertices, we may ignore them an imagine that k is smaller by m.

If G has an IS V' of size k, then we assert that the same vertices form an SIS in G'. The reason is that if two vertices $u, v \in V$ have no edge between them in G then the corresponding vertices of G' can be no closer in distance three. This is because u and v share no edge-vertex in common, and so any path from u to v must pass through at least two edge-vertices, for a total length of at least three. Therefore G' has an SIS of size k. Conversely, if G' has an SIS of size k, then we claim that the IS can contain none of the edge vertices. The reason is that $k \geq 2$ and (because there are no isolated vertices) every edge-vertex is within distance two of every other vertex of G'. If two vertices u and v are strongly independent in G' then clearly there cannot be an edge between them in G, which implies that any SIS for G' is an IS for G.

Solution 4: We first show that EPP is in NP. The certificate consists of the sequence of vertices along the s-t path. It is a simple matter to count the number of times each zone is visited by the path and verify that no zone is visited more than once.

To show that EPP is NP-hard, we reduce directed Hamiltonian path (DHP) to EPP. Given an instance $G = (V, E)$ for the directed Hamiltonian path, we create a directed graph $G' = (V', E')$ as follows. Let $n = |V|$. The vertex set V' consists of s, t. We create n copies of the vertices, organized in n layers, where edges lead from one layer to the next. More formally, in addition to s and t, the vertices V' consist of $v_{u,i}$, for $u \in V$ and $1 \leq i \leq n$ (see Fig. 2). Intuitively, visiting $v_{u,i}$ will correspond to the notion that u is the ith vertex of the Hamiltonian path. To get the path started, we create an edge from s to each vertex of the form $v_{u,1}$ for each $u \in V$. To get from level i to level $i + 1$, we create an edge $(v_{u,i}, v_{w,i+1})$, for each edge $(u, w) \in E$. To finish off the path, the create an edge from each vertex $v_{u,n}$ to t. Observe that every path of length n in G corresponds to a path in G' from s to t. In order for such a path to be a Hamiltonian path, we cannot allow the same vertex to be revisited. To enforce this condition, for each $u \in V$, we create zones $Z_u = \{v_{u,1}, v_{u,2}, \ldots, v_{u,n}\}$. Clearly, the graph G' consists of V^2 vertices and $V \cdot E$ edges, and so can be constructed in polynomial time.

To establish the correctness, we argue that G has a Hamiltonian path if and only if G' has an evasive path. First, suppose that G has a Hamiltonian path $\langle u_1, u_2, \ldots, u_n \rangle$. We map this to the s-t path

$\langle s, v_{u_1,1}, v_{u_2,2}, \ldots, v_{u_n,n}, t\rangle$ in G'. By our construction, each of the edges of this path exists, and it is evasive because each vertex u_i is visited exactly once and so each zone Z_{u_i} is visited exactly once. Therefore, G' has an evasive path. Conversely, suppose that G' has an evasive path. As observed earlier, any path from s to t in G' corresponds to an n-vertex path in G. The zone structure implies that no vertex of this path can be repeated, and hence it corresponds to a Hamiltonian path in G.

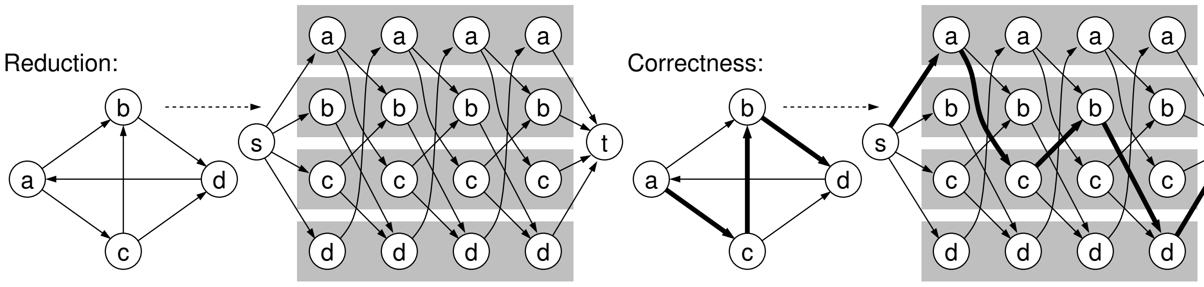


Figure 2: Reduction from DHP to EPP.

Solution 5: Let us assume that the edge weights are distinct. Let S be the solution produced by the greedy algorithm, and let S^* be the optimal solution. For each $u \in S^*$, observe that either $u \in S$, or u has a neighbor u' of higher weight that is in S. (To see this note that if u is not in S then the reason is that one of its neighbors was taken, and this caused u to be deleted. The weight of this vertex must be higher that of u, since it was selected prior to u.) For each vertex $u' \in S$, if $u' \in S^*$, define w'_u to be $w(u')$. Otherwise ($u' \notin S^*$) let w'_u be the sum of the weights of the neighboring vertices of u'. By the above observation, it follows that the total weight of the optimal solution is:

$$W(S^*) \;\leq\; \sum_{u' \in S} w'_u.$$

Clearly $w'_u < 4w(u)$. Thus we have

$$W(S^*) \;\leq\; \sum_{u' \in S} w'_u \;<\; \sum_{u' \in S} 4w(u') \;=\; 4 \sum_{u' \in S} w(u') \;=\; 4W(S).$$

Therefore, $W(S) \geq \frac{1}{4} W(S^*)$.

Solution to the Challenge Problem: The answer is $\lceil \log_2 n \rceil$ testers. To simplify things, let us assume that n is a power of 2. To see that $\log_2 n$ is the minimum possible, observe that there are n possible results, you only have time to run one test for each tester, and each test provides two possible results. Thus, at least $\log_2 n$ tests are needed to distinguish between the n possibilities.

Tapes are assigned to testers as follows. Number the tapes from 0 to $n-1$. For the kth tape, consider the binary expansion of k. Tester i views this tape if and only if the ith bit of this binary number is 1. For example, suppose that there are 8 tapes. Then tester 0 views tapes $\{1, 3, 5, 7\}$, tester 1 views tapes $\{2, 3, 6, 7\}$, and tester 2 views tapes $\{4, 5, 6, 7\}$. After the tests have been run, we wait 7 days, and construct a binary string whose ith position is 1 if the ith tester died and 0 otherwise. We claim that the resulting binary number is the index of the deadly tape. For example, if the deadly tape was tape $5 = 101_2$, then testers 0 and 2 would have viewed it, and would have died, so the resulting number would be $101_2 = 5$.

To see why this works more formally, suppose that the kth tape is deadly one, and consider the binary expansion of k, where b_i denotes the ith bit. For each position i of this binary number, if $i = 1$ then tester i views the tape and dies, and if $i = 0$ then tester i does not view the tape and lives. This, the binary number constructed based on which testers live and die gives the final answer.

How many testers survive the ordeal? Observe that each tester views half of the tapes (assuming n is a power of 2). So, each tester has a 1/2 probability of survival.

CMSC 451:Spring 2008 Dave Mount

Practice Problems for the Midterm

The midterm will be on Tues, Apr 1. The exam will be closed-book and closed-notes, but you will be allowed one cheat-sheet (front and back).

Disclaimer: These are practice problems, which have been taken from old homeworks and exams. They do not necessarily reflect the actual length, difficulty, or coverage for the exam. For example, we have covered some topics this year that were not covered in previous semesters. So, just because a topic is not covered here, do not assume it will not be on the exam.

Problem 0. You should expect one problem in which you will be asked to work an example of one of the algorithms we have presented in class on a short example.

Problem 1. Short answer questions.

(a) Consider the code $a = \langle 0\rangle$, $b = \langle 01\rangle$, $c = \langle 11\rangle$, $d = \langle 101\rangle$. Is this a prefix code? Explain.

(b) Consider a breadth-first search (BFS) of a *directed graph* $G = (V, E)$. Let (u, v) be any edge in G. Is it possible that $d[v] \geq d[u] + 2$? Explain briefly.

(c) As a function of n, give an exact closed-form solution (no embedded sums) to the following summation:

$$T(n) = \sum_{i=n}^{2n} i.$$

(d) In the DFS of a digraph there is an edge (u, v) such that $d[u] < d[v]$. What are the possible types of this edge? (tree, back, forward, cross.)

(e) Given a connected undirected graph $G = (V, E)$ in which all edges have weight 1, what is the fastest and simplest way (that we know of) for computing a minimum spanning tree for G?

(f) What is the maximum number of edges in an undirected graph with n vertices, in which each vertex has degree at most k?

(g) You are given a graph with V vertices and E edges as an adjacency list. How fast can you sort the vertices in increasing order of their degrees? (Explain briefly.)

(h) True or False: If a digraph has k distinct cycles, then for any depth-first search (DFS) of this digraph, there are at least k back edges.

Problem 2. Given two strings, $X = x_1x_2\ldots x_m$ and $Y = y_1y_2\ldots y_n$, the *shortest common supersequence* (SCS) is a minimum length string Z such that both X and Y are subsequences of Z. For example, if $X = \langle ABCBABA\rangle$ and $Y = \langle BCAABAB\rangle$, then $Z = \langle ABCAABABA\rangle$ is an SCS of both X and Y.

Give an $O(mn)$ dynamic programming algorithm which given X and Y computes the length of the SCS of X and Y. You do not need to determine the actual SCS, just its length. Be sure to give the DP formulation, and explain its correctness. (Hint: There is a cute, tricky solution. But to gain practice in doing DP problems, even if you see it, work through the full DP formulation anyway.)

Problem 3. A pharmacist has W pills and n empty bottles. Let $\{p_1, p_2, \ldots, p_n\}$ denote the number of pills each bottle can hold.

(a) Describe a greedy algorithm, which given W and the p_i's determines the fewest number of bottles needed to store all the pills. (An informal description is sufficient.)

(b) Argue that the *first* bottle chosen by your greedy algorithm will be in some optimal solution.

(c) How would you modify your algorithm if each bottle also has an associated cost c_i, and you want to minimize the total cost of the bottles used to store all the pills? (No need to prove correctness.)

Problem 4. Recall that in the interval scheduling problem, we are given a set of n requests that are to be scheduled to use some resource, where each request is defined by a start/finish time interval (s_i, f_i). The objective is to schedule the maximum number of nonoverlapping requests. Consider the following greedy strategies. In both instances, we sort the tasks according to the given criterion. Then, we repeatedly schedule the next task from the order that does not overlap in time with a previously scheduled task.

(a) *Shortest request first:* Sort the requests in increasing order of duration $f_i - s_i$. Give a counterexample to show that this is *not* optimal.

(b) *Latest start time first:* Sort the requests in decreasing order of start time. Give a short proof that this is optimal. (You may use any results proved in class.)

Problem 5. Let $G = (V, E)$ be an undirected graph. Write an $O(V + E)$ time algorithm to determine whether it is possible to direct the edges of G such that the indegree of every vertex is at least one. If it is possible, then your algorithm should show a way to do this. (Hint: Use DFS.)

Problem 6. For each part, either give a short proof of the correctness of your claim (if true) or give a counterexample (if false).

(a) Consider a weighted undirected graph G. Suppose you replace the weight of every edge with its negation (e.g. $w(u, v)$ becomes $-w(u, v)$), and compute the minimum spanning tree of the resulting graph using Kruskal's algorithm. True or False: The resulting tree is a *maximum* cost spanning tree for the original graph.

(b) Consider a weighted digraph G and source vertex s. Suppose you replace the weight of every edge with its negation and compute the shortest paths using Dijkstra's algorithm. True or False: The resulting paths are the *longest* (i.e., highest cost) simple paths from s to every vertex in the original digraph.

Problem 7. You are given an undirected graph $G = (V, E)$ where each vertex is a gas station and each edge is a road with an associated weight $w(u, v)$ indicating the distance from station u to v. The brilliant but flaky Professor Farnsworth wants to drive from vertex s to vertex t. Since his car is old and may break down, he does not like to drive along long stretches of road. He wants to find the path from s to t that minimizes the *maximum* weight of any edge on the path. Give an $O(E \log V)$ algorithm to do this. Briefly justify your algorithm's correctness and derive its running time.

Solutions to the Practice Midterm Problems

Solution 1:

(a) It is not a prefix code, since $a = \langle 0 \rangle$ is a prefix of $b = \langle 01 \rangle$.

(b) No. This would mean that v is more than two levels deeper than u. When u is visited, it would put v on the queue (or else it is already there), implying that $d[v] \leq d[u] + 1$.

(c)

$$T(n) \;=\; \sum_{i=n}^{2n} i \;=\; \sum_{i=0}^{2n} i - \sum_{i=0}^{n-1} i \;=\; \frac{2n(2n+1)}{2} - \frac{n(n-1)}{2} \;=\; \frac{3n(n+1)}{2}.$$

(d) It could be a tree edge or forward edge. (Back edges and cross edges go to vertices with earlier discovery times.)

(e) Let V denote the number of vertices. The trick is to observe that since *any* spanning tree has $V - 1$ edges, and since all edges have weight 1, all spanning trees have the same weight. So it suffices to find any spanning tree. BFS and DFS trees are both spanning trees, and both are equally easy to compute. Both take $\Theta(V + E)$ time.

(f) Since each vertex has degree at most k, the sum of vertex degrees is at most nk. But we saw in class that the sum of vertex degrees is twice the number of edges, and thus $2E \leq nk$ from which we have $E \leq nk/2$.

(g) You can compute the vertex degrees in $O(V + E)$ time, by a simple traversal of the adjacency list. Through the use of some fast integer sorting algorithm (e.g., CountingSort), the vertices can be sorted in additional $O(V)$ time.

(h) False. Consider a complete digraph. It has an exponential number of different cycles, but there can be no more than $O(n^2)$ back edges, since the entire graph has at most $O(n^2)$ edges.

Solution 2: Here is a DP algorithm. Let $c[i, j]$ denote the length of the SCS of X_i and Y_j. As a basis, observe that $c[i, 0] = i$ and $c[0, j] = j$, since if one sequence is empty, the SCS is just the other sequence. In general, to compute $c[i, j]$, we observe that if $x_i = y_j$ then we put this common character into the SCS and recurse on remaining strings X_{i-1} and Y_{j-1}, for a cost of $1 + c[i - 1, j - 1]$. Otherwise either x_i or y_j must be added to the SCS (perhaps both). If we add x_i to the SCS, the cost is $1 + c[i - 1, j]$, and a similar case holds if we add y_j instead. Since we do not know which is better, we try both.

```
SCS(x[1..m], y[1..n]) {
    for i = 0 to m do c[i,0] = i;          // initialize first column
    for j = 0 to n do c[0,j] = j;          // initialize first row
    for i = 1 to m do {
        for j = 1 to n do {
            if (x[i] == y[j]) c[i,j] = 1 + c[i-1,j-1];
            else              c[i,j] = 1 + min(c[i,j-1], c[i-1,j]);
        }
    }
    return c[m,n];                         // return final length
}
```

There is a very short and tricky solution, by the way. It consists of computing the sum of the lengths of X and Y and then subtracting from this the length of LCS(X, Y). It is a little tricky to prove this is correct, but intuitively the SCS consists of the union of characters in X and Y, but each character in the LCS need only appear once in the LCS. Since the LCS still takes $O(nm)$ time to compute, this is no faster.

Solution 3:

(a) Sort the bottles in decreasing order of size. Then fill each bottle in order until all the pills are gone.

(b) We claim that the largest bottle will be part of some optimal solution. Let B denote the optimum set of bottles, and let p_o denote the largest bottle used B, and let p_l denote the largest bottle overall. If $p_o = p_l$ then we are done. Otherwise, convert B into an alternative solution B' by replacing p_o with p_l. Since $p_l \geq p_o$, we can fit all the pills from p_o into p_l. Thus, this is a feasible solution, which uses the same number of bottles as the optimum solution.

(c) If the bottles have associated costs c_i, then sort the bottles by increasing order of the ratio c_i/p_i (cost per capacity). Then select bottles in this order.

Solution 4:

(a) One counterexample consists of two long nonoverlapping activities, with a short activity in between that overlaps both. Consider the three activities with the start/finish times $(0, 10)$, $(9, 11)$, $(10, 20)$. The shortest-duration first would schedule the second activity only, but the optimum would schedule the first and last.

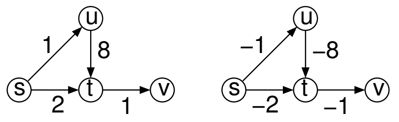
(b) The *sneaky proof* of optimality is that it corresponds to a simple time reversal of the earliest-finish-time strategy given in class. But this doesn't indicate that you know anything about how to prove the optimality of greedy algorithms, so here is a more complete proof. Suppose to the contrary that some nongreedy schedule A was indeed optimal. Let G be the (latest start time) greedy schedule. Order the activities in decreasing order of start time. Suppose that A and G agree up until the jth activity, that is, $A = \langle x_1, \ldots, x_{j-1}, x_j, \ldots, x_k\rangle$ and $G = \langle x_1, \ldots, x_{j-1}, g_j, \ldots, g_m\rangle$. Since G selects the activity with the latest possible start time, we know that g_j does not conflict with any of the earlier activities of A. Consider the modified "greedier" schedule A' that results by replacing x_j with g_j in the schedule A, that is, $A' = \langle x_1, x_2, \ldots, x_{j-1}, g_j, x_{j+1}, \ldots\rangle$. This is a feasible schedule, and it has the same number of activities as A. Therefore, A' is also optimal. By repeating this process, we will eventually convert A into G, without decreasing the number of activities. Therefore, G is also optimal.

Solution 5: Let us assume that the graph is connected. If not, apply the solution to each connected component. We claim that it is possible to direct the edges in this way if and only if the graph has at least one cycle. If the graph has no cycle, then it is a free tree and has exactly $V - 1$ edges. By the pigeonhole principal, it is impossible to direct $V - 1$ edges so that all V vertices have an incoming edge. Conversely, suppose that the graph has a cycle. Let u be any vertex on the cycle. The vertex u can be found by applying DFS, finding a back-edge, and then letting u be either endpoint of the back-edge. Then start a second DFS at vertex u. There must be a back-edge coming into u. Direct this back-edge into u, direct all the other tree edges in the DFS tree from parent to child. (All other edges may be directed arbitrarily.) It follows that every vertex has at least one incoming edge. It takes $\Theta(V + E)$ time to run the two DFS's.

Solution 6:

(a) True. Recall that both Kruskal and Prim's algorithms are correct with negative edge weights. Let G and G' denote the original graph and the one with negated edge weights. Note that a spanning tree of G is also a spanning tree of G'. For any spanning tree T, let $c(T)$ denote the total cost (sum of edge weights) in G, and let $c'(T)$ denote the corresponding cost in G'. It follows directly that $-c(T) = c'(T)$ for all T. Thus, finding the spanning tree T that minimizes c', we find the spanning tree that minimizes $-c$, and such a tree maximizes c.

(b) False. Consider the digraph shown below. When Dijkstra's is run on the negated graph, it will relax vertices in the order s (distance 0), t (distance -2), v (distance -3), and u (distance -1), which leaves v's distance at -3. But the longest simple path from s to v has length 10 in the original graph. In fact, it can be shown that computing maximum length simple paths in a digraph is an NP-hard problem, so no simple modification will work.



Solution 7: One solution is based on modifying the relaxation rule for Dijkstra's algorithm. Define the *cost* of a path to be the maximum weight edge on the path, and vertices u and v, define $\mu(u, v)$ to be the cost of the minimum cost path from u to v. We maintain an array m, which for each $u \in V$, $m[u]$ is equal to smallest cost we know of from s to u so far. Given that $m[u]$ is the cost of getting from s to u, and if v is a neighbor of u, then there is a path from s to v of cost $\max(m[u], w(u, v))$. Thus the relaxation rule for edge (u, v) is changed as follows:

```
Relax(u,v) {
    if (max(m[u], w(u,v)) < m[v]) {
        m[v] = max(m[u], w(u,v));
        pred[v] = u;
    }
}
```

Otherwise Dijkstra's algorithm is unchanged (except that the priority queue is sorted by m-value). On termination, return the reversal of the path formed by following the predecessor pointer from t back to s. The running time is the same as Dijkstra's algorithm. The correctness comes about by a simple modification of the correctness of Dijkstra's algorithm, which is based on contradiction. If, to the contrary, the algorithm ever finalizes the cost of a vertex u for which $m[u]$ is incorrect, then there must be a lower cost path. Let y be the first vertex on this path which is not finalized. We would then have $\mu(s, y) \leq \mu(s, u) < m[u]$, implying that $m[y] < m[u]$, and hence y would have been chosen before u, a contradiction.

Here is an alternative solution, which is trickier, but somewhat more direct. Compute a MST for G (using, say Prim's algorithm) and take the unique path from s to t. It is a bit tricky to see why this works. One way is to observe that the above relaxation rule is exactly the same as the update rule in Prim's algorithm. A different way to establish correctness is by contradiction. Assume, for simplicity, that all edge weights are distinct. If the MST path does not minimize the maximum edge weight, then remove the maximum edge weight along the path. This splits the MST into two trees. The min-max path must contain at least one edge that crosses the cut from one tree to the other. Add it. Because this is the min-max path, the new edge is of lower cost than the one that was removed. The result is a spanning tree with lower weight, which contradicts the minimality of the original MST.

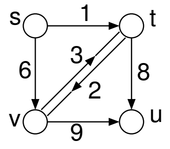
CMSC 451:Spring 2008 Dave Mount

Midterm Exam

This exam is closed-book and closed-notes. You may use one sheet of notes (front and back). Write all answers in the exam booklet. You may use any algorithms or results given in class. If you have a question, either raise your hand or come to the front of class. Total point value is 100 points. Good luck!

Problem 1. (15 points)

Show the execution of Dijkstra's shortest path algorithm on the digraph in the figure, where s is the source vertex. Indicate each of the following:



(a) The order in which the vertices are removed from the priority queue,
(b) The final distance value `d[u]` for each vertex u,
(c) The final predecessor pointer `pred[u]` for each vertex u.

Problem 2. (30 points; 4–7 points each.) Short answer questions. Explanations are not always required, but may be given for partial credit.

(a) List the following functions in increasing asymptotic order. (No explanations required.)

$$f(n) \;=\; n^{\lg 8} \qquad g(n) \;=\; 2^{(2\lg n)} \qquad h(n) \;=\; (\min(n,2))^5.$$

If two functions are asymptotically equal (e.g., if $f(n) = \Theta(g(n))$), then indicate this. (Recall that lg denotes logarithm base 2.)

(b) Recall that a digraph $G = (V,E)$ is *semi-connected* if, given any two vertices, $u, v \in V$, there exists a path from u to v, or there exists a path from v to u, but not necessarily both. Suppose you are given a semi-connected DAG G. True or false: The topological ordering of the vertices of G is unique. Explain briefly.

(c) Suppose that you run DFS on a directed graph $G = (V,E)$ and compute both discovery and finish times for each vertex. For some edge (u,v) you observe that $f[v] < d[u]$. What type of edge might this be? (List all the apply. Give a short explanation.)

(i) tree edge (ii) back edge (iii) forward edge (iv) cross edge

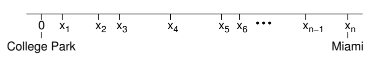
(d) You are given a connected, undirected graph $G = (V,E)$ with positive edge weights. You form another graph G' by squaring the weight of every edge of G. True or false: A spanning tree T is a minimum spanning tree of G if and only if T is a minimum spanning tree of G'. Explain briefly.

(e) Recall the following two rules used in the Floyd-Warshall algorithm for computing shortest paths in a directed graph with edge weights $w(i,j)$.

$$d_{ij}^{(0)} \;=\; w(i,j) \qquad d_{ij}^{(k)} \;=\; \min\left(d_{ij}^{(k-1)},\; d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$$

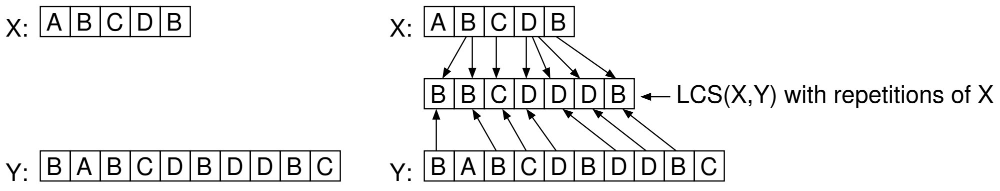
Explain how to modify these rules for the following related problem. (You do not need to modify the entire algorithm, just the rules.) For each edge $(i,j) \in E$ you are given the probability $0 < p(i,j) < 1$ that the edge is usable. The probability that a path is usable is the *product* (i.e., multiplication) of the probabilities of the edges on the path. For each i and j, the objective of your algorithm is to compute the path whose probability of usability is *maximimum.*

Problem 3. (20 points) Professor Farnsworth drives from College Park to Miami Florida along I-95. He starts with a full tank and can go for 100 miles on a full tank. Let $x_1 < x_2 < \ldots < x_n$ denote the locations of the various gas stations along the way, measured in miles from College Park. Present an algorithm that determines the *fewest number* of gas stations he needs to stop at to make it to Miami without running out of gas along the way. Give a short *proof of the correctness.*



Problem 4. (20 points) For each of the following two problems, present a short algorithm, briefly justify its correctness and derive its running time. (If you present a DP solution, it is sufficient to present the recursive DP formulation, rather than an entire algorithm.)

(a) Consider the following modification of the longest common subsequence (LCS) problem. You are given two strings $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. The objective is to compute the length of the longest common subsequence, in which each character of X (but not Y) is allowed to be *repeated* any number of times in the LCS.



Your program need only output the *length* of the LCS, not the actual subsequence.

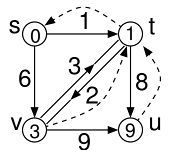
(b) Consider a further modification. In addition to X and Y, you are give a positive integer k. The problem is the same as in part (a), but each character of X can be repeated *at most k times* in the final LCS. (Characters of Y cannot be repeated.)

For example, if $k = 2$ in the above figure, you would only be allowed to use 'D' twice, rather than three times. (Hint: Add another index to your formulation of part (a).)

Problem 5. (15 points) You are given a connected undirected graph $G = (V, E)$ in which each edge's weight is either 1 or 2. Present an $O(V + E)$ time algorithm to compute a minimum spanning tree for G. Explain your algorithm's correctness and derive its running time. (Hint: This can be done by a variant of DFS.)

Solutions to the Midterm Exam

Solution 1: The vertices are removed from the priority queue in the order $\langle s, t, v, u\rangle$. The final d values for the vertices are indicated in the figure, and the predecessor pointers are shown with dashed lines. (The predecessor for s is null.)



Solution 2:

(a) Simplifying we have $f(n) = n^{\lg 8} = n^3$, $g(n) = 2^{(2 \lg n)} = 2^{\lg(n^2)} = n^2$, and $h(n) = (\min(n, 20{,}000))^5 = O(1)$, thus the order is $h(n) \prec g(n) \prec f(n)$, and none are equivalent.

(b) True. We saw in Homework 2 that a graph is semi-connected if and only if there exists a single path that passes through all the vertices. The topological ordering is order of vertices along this path.

(c) Cross edge: The fact that v finished before u was first discovered can only mean that u and v are unrelated to each other as ancestor or descendant, which rules out all the other possibilities.

(d) True: Observe that the execution of Kruskal's algorithm (which is correct) depends only on the relative order of the edge weights. Squaring the positive edge weights preserves their relative order.

(e) We replace addition and minimization with multiplication and maximization, respectively.

$$d_{ij}^{(0)} = p(i,j) \qquad\qquad d_{ij}^{(k)} = \max\left(d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} \cdot d_{kj}^{(k-1)}\right)$$

Solution 3: We assume no gap exceeds 100 miles. We greedily go as far as possible before stopping to refuel. The variable `lastStop` indicates the location that he last got fuel. We find the farthest station from this that is within 100 miles and get fuel there. The code block below provides a sketch of the algorithm. To avoid subscripting out of bounds, let us assume that $x[n+1] = x[n]$.

Greedy Algorithm for Refueling

```
Fueling(x, n) {
    lastStop = 0;
    for (i = 1 to n) {
        if (x[i+1] > lastStop) > 100) {
            lastStop = x[i];  print "Refuel at ", lastStop;
        }
    }
}
```

Clearly the running time is $O(n)$. Observe that this produces a feasible sequence, since we never go more than 100 miles before stopping. To establish optimality, let $F = \langle f_1, f_2, \ldots, f_k\rangle$ be the indices of an optimal sequence of refueling stops, and let $G = \langle g_1, g_2, \ldots, g_{k'}\rangle$ be the greedy sequence. If the two sequences are the same, then we are done. If not, let i be the smallest index such that $g_i \neq f_i$. Because greedy algorithm selects the last possible gas station, we know that $g_i > f_i$. Consider an alternative solution F' which comes by replacing f_i with g_i. We claim that F' is a also a feasible solution. To see this observe that sequence up to g_i is the same as G (which we know is feasible) and because we have delayed fueling, for the rest of the trip we have at least as much gas as we had with F (which we know is feasible). The sequence F' has the same number of stops as F and so is also optimal, and it has one more segment in common with G. By repeating this, eventually we will have an optimal solution that is identical to G.

Solution 4:

(a) We define a matrix $c[0..m, 0..n]$ where $c[i,j]$ is the length of the LCS with repetitions for $X_i = \langle x_1, \ldots, x_i \rangle$ and $Y_j = \langle y_1, \ldots, y_j \rangle$. The basis case is the same as in the standard LCS problem, as is the case when $x_i \neq y_j$. When $x_i = y_j$, the old formulation consumes both of these characters (by defining $c[i,j] = 1 + c[i-1, j-1]$). Our modification is to consume y_j, but not x_i (thus allowing x_i to be used again). We have $c[i,j] = 1 + c[i, j-1]$. The final DP formulation is given below:

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ 1 + c[i, j-1] & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

(b) We define a matrix $c[0..m, 0..n, 0..k]$ where $c[i,j,r]$ is equal to the length of the LCS (as defined in part (a)) with at most r repetitions of the last character. Our approach is that, every time we elect to match the last two characters and $r > 0$, we decrement both the j component (thus consuming y_j) and the r component (thus allowing one fewer repetition of x_i). If not, we recurse as we do in the standard LCS formulation by skipping either x_i or y_j. Whenever we skip x_i (by decrementing i) we reset the repetition component to its maximum value of k, since we are starting with a new character of X. Here is the final formulation.

$$c[i,j,r] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ 1 + c[i, j-1, r-1] & \text{if } i, j, r > 0 \text{ and } x_i = y_j, \\ \max(c[i, j-1, r], c[i-1, j, k]) & \text{if } i, j > 0 \text{ and } (r = 0 \text{ or } x_i \neq y_j). \end{cases}$$

Solution 5: Note that Kruskal's algorithm is *not* easy to modify. Although it is possible to sort the edges in $O(E)$ time by Counting Sort, it is not clear how to perform the Union-Find operations in linear time. There are two $O(V+E)$ solutions that I know of. One is based on performing two DFS's, first using just the weight-1 edges and the second using the weight-2 edges. The other, which I describe below, is based on a variant of BFS.

Since G is connected, BFS will produce a single BFS tree, that is, a spanning tree for G. However, this is not necessarily a minimum spanning tree. The problem is that BFS selects the next vertex to visit arbitrarily. To fix this, our algorithm will give preference to visiting edges of weight 1 over edges of weight 2. At any time, there will be a collection of vertices waiting to be processed (colored gray). We will visit only those that are accessible via paths consisting of edges of weight 1, from a previously processed vertex. If there are none, then we will visit any unprocessed vertex.

To implement this, rather than using a traditional first-in first-out queue, we will use a two-sided queue. Vertices reachable through edges of weight 1 will be enqueued at the front of the queue and vertices reachable by edges of weight 2 will be enqueued at the back of the queue. In this way, whenever we dequeue (always from the front of the queue) we will be favoring the former over the latter. Clearly these queueing operations can be done in $O(1)$ time each (in contrast to a standard priority queue, which would require $O(\log V)$ time per operation.)

To keep track of the two different types of vertices, each vertex u is associated with a priority value, denoted $d[u]$. When processing a vertex u, and seeing the edge (u,v) to a vertex v, we set $d[v] = \min(d[v], w(u,v))$. If v is in the queue, and its weight decreases, we need to move it from the back part to the front part. (This can be done in $O(1)$ time.)

As with BFS, the running time is $O(V+E)$. We claim that the BFS tree that results is an MST for G. Suppose to the contrary the algorithm generates a BFS tree that is not an MST. Let (u,v) be the first edge that the algorithm outputs that is not in any MST of G. Let T be an MST that agrees with all the decisions that our BFS algorithm has made prior to outputting (u,v). Adding (u,v) to T creates a cycle. Removing any edge from this cycle restores a spanning tree. Since (u,v) is not in any MST, it follows that $w(u,v) = 2$,

```
BFS(G, s) {
    for each (u in V) {                     // initialization
        color[u] = white;
        d[u] = infinity;
    }
    color[s] = gray;  pred[s] = null;  d[s] = 1;
    store all vertices in Q with s in front and all others in back;
    while (Q is nonempty) {
        u = dequeue from head of Q          // u is the next to visit
        for each (v in Adj[u]) {
            if (color[v] != black) {        // process v
                if (w(u,v) < d[v]) {
                    pred[v] = u;
                    d[v] = w(u,v);
                    move v from back of queue to front;
                }
                color[v] = gray;
            }
        }
        color[u] = black;
    }
}
```

and every other edge of this cycle has weight 1 (for otherwise, replacing it would have no impact on the tree's cost). Thus, there is a path from u to v consisting entirely of edges of weight 1. However, if this were true, when the BFS first visited u, it will eventually encounter v and mark it gray before ever using any of the weight-2 neighbors of u. This contradicts the hypothesis that the edge (u, v) has been output by the algorithm.

Practice Problems for the Final Exam

The final will be on Fri, May 16, 8-10am. The exam will be closed-book and closed-notes, but you will be allowed two pages of notes (front and back of each page).

Disclaimer: These are practice problems, which have been taken from old homeworks and exams. They do not necessarily reflect the actual length, difficulty, or coverage for the exam. For example, we have covered some topics this year that were not covered in previous semesters. So, just because a topic is not covered here, do not assume it will not be on the exam.

Problem 0. You should expect one problem in which you will be asked to work an example of one of the algorithms we have presented in class or some NP-complete reduction we covered.

Problem 1. Short answer questions.

(a) Technically advanced aliens come to Earth and show us that some known NP-hard problem cannot be solved faster than $O(n^{100})$ time. Does this resolve the question of whether P = NP? (Explain briefly.)

(b) Suppose that $A \leq_P B$, the reduction runs in $O(n^2)$ time, and B can be solved in $O(n^4)$ time. What can we infer about the time needed to solve A? (Explain briefly.)

(c) True or False: If a graph G has a vertex cover of size k, then it has a dominating set of size k or smaller.

(d) Suppose that $A \leq_P B$, and there is a factor-2 approximation to problem B, then which of the following necessarily follows:

(i) There is a factor-2 approximation for A.

(ii) There is a constant factor approximation for A, but the factor might not be 2.

(iii) We cannot infer anything about the approximability of A.

Problem 2. (Bucket redistribution) You are given a collection of n blue buckets, and n red buckets. These are denoted B_i and R_i for $0 \leq i \leq n-1$. Initially each of the blue buckets contains some number of balls and each red bucket is empty. The objective is to transfer all the balls from the blue buckets into the red buckets, subject to the following restrictions.

The input to the problem consists of two sequences of integers, $\langle b_0, b_1, \ldots, b_{n-1}\rangle$ and $\langle r_0, r_1, \ldots, r_{n-1}\rangle$. Blue bucket B_i holds b_i balls initially, and at the end, red bucket R_i should hold exactly r_i balls. The balls from blue bucket B_i may be redistributed only among the red buckets R_{i-1}, R_i, and R_{i+1} (indices taken modulo n). You may assume that $\sum_i b_i = \sum_i r_i$.

Design a polynomial time algorithm which given the lists $\langle b_0, b_1, \ldots, b_{n-1}\rangle$ and $\langle r_0, r_1, \ldots, r_{n-1}\rangle$, determines whether it is possible to redistribute the balls from the blue buckets into the red buckets according to these restrictions. (Hint: Use network flow.)

Problem 3. Given an undirected graph $G = (V, E)$, a *feedback vertex set* is a subset of vertices such that every simple cycle in G passes through one of these vertices. The *feedback vertex set problem* (FVS) is, given an undirected graph G and an integer k, does G contain a feedback vertex set of size at most k? For example, the graph shown in Fig. 1 has a feedback vertex set of size 2 (shaded).

Show that FVS is in NP. That is, given a graph G that has a FVS of size k, give a certificate, and show how you would use this certificate to verify the presence of a FVS in polynomial time. (This is one of the few NP problems where verifying a certificate is not obvious.)

(You are *not* asked to show that FVS is NP-hard, but if you wanted to try this, as a hint the reduction is from Vertex Cover.)

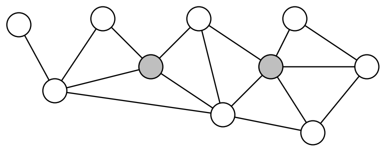


Figure 1: A feedback vertex set (FVS) of size 2.

Problem 4. A *tournament* is a digraph $G = (V, E)$ in which for each pair of vertices u and v, either there is an edge (u, v) or an edge (v, u) but not both. (See Fig. 2.) A directed *Hamiltonian path* is a path that visits every vertex in a digraph exactly once.

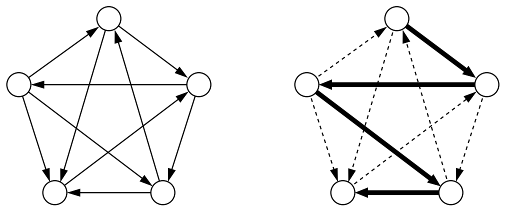


Figure 2: Hamiltonian path in a tournament.

(a) Prove that for all $n \geq 1$, every tournament on n vertices has a directed Hamiltonian path. (Hint: Use induction on the number of vertices.)

(b) Give an $O(n^2)$ algorithm which given a tournament, finds a Hamiltonian path. (You may assume either an adjacency list or an adjacency matrix representation of G.)

Problem 5. Prove that the following problem, called the *acyclic subgraph problem* (AS) is NP-complete. Given a directed graph $G = (V, E)$ and an integer k, determine whether G contains a subset V' of k vertices such that the induced subgraph on V' is acyclic. Recall that the *induced subgraph* on V' is the subgraph $G' = (V', E')$ whose vertex set is V', and for which $(u, v) \in E'$ if $u, v \in V'$ and $(u, v) \in E$. (Hint: Reduction from Independent Set. Think of a reduction that maps undirected edges to directed cycles.)

Problem 6. Show that the following problem is NP-complete. Remember to show (1) that it is in NP and (2) that some known NP-complete problem can be reduced to it.

Balanced 3-coloring (B3C): Given a graph $G = (V, E)$, where $|V|$ is a multiple of 3, can G can be 3-colored such that the sizes of the 3 color groups are all equal to $|V|/3$. That is, can we assign an integer from $\{1, 2, 3\}$ to each vertex of G such that no two adjacent vertices have the same color, and such that all the colors are used equally often.

Hint: Reduction from the standard 3-coloring problem (3COL).

Problem 7. The *set cover* optimization problem is: Given a pair (X, S), where X is a finite set and a $S = \{s_1, s_2, \ldots, s_n\}$ is a collection of subsets of X, find a minimum sized collection of these sets C whose union equals X. Consider a special version of the set-cover problem in which each element of X occurs in *at most* three sets of S. Present an approximation algorithm for this special version of the set cover problem with a ratio bound of 3. Briefly derive the ratio bound of your algorithm

Problem 8. Prove that the Hamiltonian Path problem is NP-complete for undirected graphs. (Hint: Reduction from Directed Hamiltonian Path. Replace each vertex with a small gadget that splits the incident edges into two groups, such that if a path enters from one group of edges then it must leave along the others.)

Solutions to Final Exam Practice Problems

Solution 1:

(a) No. Had they done this for a problem in NP, then it would imply that P $\neq$ NP. But NP-hard problems may generally lie outside of NP.

(b) The reduction runs in $O(n^2)$ time, which means that it might take an input of size n and produce an output of size $m = O(n^2)$. The solution for B runs in $O(m^4)$ time, and so the overall algorithm runs in $O((n^2)^4) = O(n^8)$ time.

(c) False. This would be true if there were no isolated vertices. If there are k' isolated vertices, they must all be included in the dominating set, but serve no purpose for the vertex cover.

(d) (iii). Since reductions do not preserve approximation bounds, we cannot infer anything about A's approximability.

Solution 2: We create an s-t network as follows. (Alternately, this could be expressed as a circulation problem.) We create a source vertex s and sink t. For each blue bucket we create a vertex B_i, and an edge (s, B_i) of capacity b_i. For each red bucket we create a vertex R_i and create an edge (R_i, t) of capacity r_i. Finally, for each i, we create directed edges $(B_i, R_{i-1}), (B_i, R_i), (B_i, R_{i+1})$ (indices taken modulo n), each of infinite capacity. We run any network flow algorithm on this network. If there exists a flow that saturates all the edges coming into t, then the redistribution is possible. In particular, the flow on edge (B_i, R_j) indicates the number of balls to take from B_i and transfer to R_j. If the capacity is smaller, then such a redistribution is not possible, and the total flow equals the maximum number of balls that can be redistributed subject to the rules.

Solution 3: Suppose that G is a graph that has a feedback vertex set of size k. The certificate is the set V' of vertices that are in the feedback vertex set. To determine whether V' is indeed a feedback vertex set, we need to test whether every cycle in G passes through at least one of these vertices. Note that there are exponentially many cycles in a graph, so testing each cycle would not run in polynomial time. Instead we delete all the vertices of V' from G, along with any incident edges. Let G'' denote the resulting graph. Then we test whether G'' has a cycle, by running DFS and checking whether the resulting tree has at least one backedge. If so, V' is not a feedback edge set (since the cycle in G'' does not pass through any vertex of V') and otherwise it is.

Solution 4:

(a) The proof is by induction on the number of vertices n in the tournament. If $n = 1$, then there is a trivial Hamiltonian path consisting of the vertex itself. Otherwise, if $n > 1$, we assume inductively that all tournaments with strictly less than n vertices have a Hamiltonian path. Delete any vertex v from the tournament. By the induction hypothesis, the remaining tournament on $n - 1$ vertices has a Hamiltonian path $\langle v_1, v_2, \ldots, v_{n-1} \rangle$. Now, consider the sequence of edges going between v and each of these vertices. If the first edge is directed out of v ((v, v_1)) then we may complete the Hamiltonian path by adding v to the front of the path. If the last edge is directed into v ((v_{n-1}, v)) then we may complete the path by adding v to the end of the path.

Otherwise, observe that the first edge is directed into v and the last edge is directed out of v, and there are edges going either to or from every other vertex. It follows, somewhere in between, there must be a pair of consecutive vertices v_i and v_{i+1}, such that there exists edge (v_i, v) directed into v and (v, v_{i+1}) directed out of v. We complete the path by inserting v between v_i and v_{i+1}.

(b) The algorithm just simulates the proof above. Let the vertex set by $V = \{1, 2, \ldots, n\}$. The procedure HamPath(A, n) is given an $n \times n$ boolean adjacency matrix A for the tournament, and returns a list P consisting of the vertices on the Hamiltonian path. At each iteration it adds vertex i into the path.

Hamiltonian Path in a Tournament

```
HamPath(A, n) {
    P = <1>                                 // initialize path
    for i = 2 to n do {                     // add vertex i to path
        j = 1;
        while (j < i && !A[i, P[j]]) j++   // find first edge leaving i
        if (j < i) insert i immediately before P[j] in P;
        else insert i at end of P;
    }
    return P
}
```

The inner while-loop can be executed at most $i-1$ times, so the total running time is proportional to $\sum_{i=2}^{n}(i-1) \in \Theta(n^2)$.

Solution 5:

AS $\in$ NP: (This is essentially the same as the solutions to Problem 3 above.) The certificate consists of the vertices of V'. In $O(n+e)$ time we can build an adjacency list for the induced subgraph on V' (basically by deleting any references to the vertices that are not in V' from the original adjacency list), and then run DFS and search for the presence of a backedge. If there is no backedge, then V' is an AS.

IS $\leq_P$ AS: Given an input to the independent set problem, (G, k), where G is an undirected graph and k is an integer, we will generate a pair (G', k') such that G has a IS of size k if an only if the digraph G' has an AS of size k'. The intuition is to map each edge of G into a cycle in G'.

In particular, G' has the same vertex set as G. For each undirected edge $(u, v) \in E(G)$ we create two directed edges (u, v) and (v, u) for $E(G')$. We set $k' = k$. We output the pair (G', k'). Clearly this can be done in polynomial time.

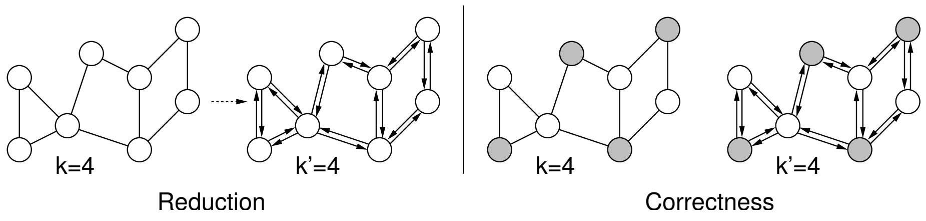


Figure 1: IS to AS Reduction.

We claim that G has an IS of size k if and only if G' has an AS of size k'.

($\Rightarrow$) If V' are the vertices of an IS, then we know that the induced subgraph on V' has no edges in either G or G', implying that there are no edges between the vertices of V' in G', and so V' is an AS for G'.

($\Leftarrow$) Conversely, suppose that V' is an AS for G'. We claim that V' is an IS for G. If not, there would be two vertices $u, v \in V'$, such that the edge (u, v) is in G. But, by our reduction, this would imply that there are the two directed edges (u, v) and (v, u), and these would form a cycle. This contradicts the choice of V'.

Solution 6:

B3C $\in$ NP: The certificate is the assignment of colors to vertices. In polynomial time we can check that the coloring is legal and that each color is used $|V|/3$ times.

3COL $\leq_P$ B3C: Let $G = (V, E)$ be an instance of the 3-coloring problem. We want to produce an equivalent instance of B3C, consisting of a graph G'. The trick is to modify G so that we may assume that every color is used equally often. Let $n = |V|$. Let G' consist of the graph G together with $2n$ additional vertices. Each of these $2n$ vertices is isolated in the sense that it is not adjacent to any other vertex. Clearly G' can be produced from G in polynomial time.

We claim that G is 3-colorable if and only if G' can be 3-colored with colors each occurring equally often.

($\Leftarrow$) Suppose that G' can be 3-colored with balanced color groups. Then, if we discard the $2n$ newly added vertices, then this also a coloring for the remaining graph, namely G.

($\Rightarrow$) Suppose that G is 3-colorable. Let k'_1, k'_2, k'_3 be the number of times each of the colors appears. We have $k'_1 + k'_2 + k'_3 = n$. We can color $n - k'_i$ of the isolated vertices with color i. The total number of vertices with color i is $k'_i + (n - k'_i) = n$. Since G' has $3n$ vertices, each color is used equally often.

Solution 7: The approximation algorithm is a generalization of the vertex cover approximation. Initially all elements are uncovered. Select any uncovered element $x \in X$, and consider the (at most 3) sets that contain x. We know that one of these sets must be in the optimum set cover. Put all three sets into the set cover. Then mark all the elements in the union of these sets as covered. Repeat until no more uncovered elements remain. Observe that for each element x we select, there must be at least one set in the optimum set cover to cover this element, and we select at most 3 sets. Hence, our set cover is at most 3 times larger than the optimum set cover.

Solution 8: It will be a bit easier to prove the claim for Undirected Hamiltonian Cycle (UHC), rather than Hamiltonian path, and it illustrates all the important points. To show that Unidrected Hamiltonian Cycle problem (UHC) is in NP, we use the same verification as for the Directed Hamiltonian Cycle problem (DHC), that is, we are given the sequence of vertices in the cycle and we can easily verify that this sequence forms a cycle that visits each vertex exactly once.

To show that UHC is NP-complete, we reduce DHC to UHC. Let $G = (V, E)$ be a directed graph for the DHC problem and we will show how to constructe an undirected graph $G' = (V', E')$ for the UHC problem, so that G has a Hamiltonian cycle if and only if G' does.

The key to the reduction is showing how to simulate the notion of direction in the edges of the graph. Let $u \in V$ be a vertex of G. We replace the vertex u in G' with a gadget consisting of three vertices u^-, u^0, and u^+. Intuitively, any cycle passing through u in G will enter the gadget through u^-, next pass through u^0, and finally exit the gadget through u^+. We refer to u^- as the gadget's *input port* and u^+ as the gadget's *output port.*

Here is how we construct this gadget. First, for each vertex $u \in V$, we add the vertices u^-, u^0, u^+ to V', and we add the undirected edges $\{u^-, u^0\}$ and $\{u^0, u^+\}$. Next, for each directed edge $(u, v) \in E$, we add the undirected edge $\{u^+, v^-\}$ to E' (thus joining the output port of u's gadget to the input port of v's gadget). We output the resulting graph G'. (See Fig. 2 for an example.)

To establish the correctness of this reduction, we show that G has a directed Hamiltonian cycle if and only if G' has an undirected Hamiltonian cycle. To see the first part, consider any directed cycle $\langle u_0, u_1, \ldots, u_n, u_0\rangle$ in G. This can be mapped to the following cycle in G', which passes through each of the gadgets in the same sequence

$$\langle u_0^-, u_0^0, u_0^+,\ u_1^-, u_1^0, u_1^+, \ldots, u_k^-, u_k^0, u_k^+,\ u_0^-\rangle$$

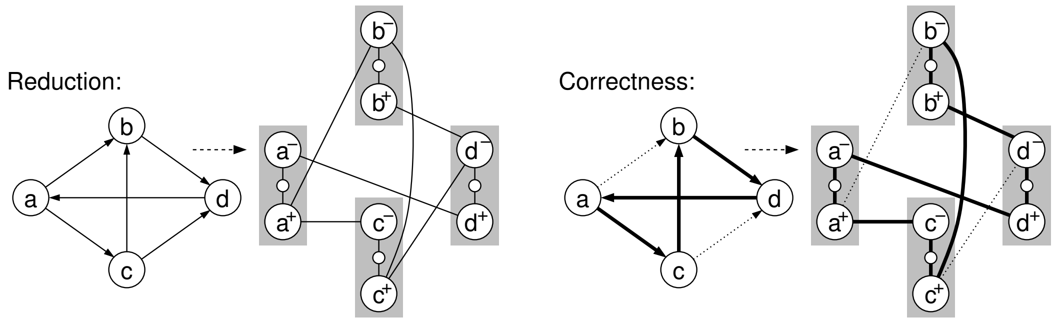


Figure 2: DHC to UHC reduction.

(See Fig. 2 for an example.) Since each such cycle visits all the vertices of the gadget, it is easy to see that any Hamiltonian cycle in G maps to a Hamiltonian cycle in G'.

Conversely, suppose that G' has a Hamiltonian cycle. We say that a cycle traverses a gadget in the *positive direction* if it visits the vertices in the order $\langle -, 0, + \rangle$. We need to show that it visits the gadgets in the appropriate manner. In particular, we assert that such a cycle satisfies the following properties:

(i) It visits the three vertices of each gadget consecutively.

(iii) Any Hamiltonian cycle of G' can be oriented so that it visits all gadgets in the positive direction.

To see that (i) holds, suppose that a cycle entered a gadget along either its entry or exit port u^- or u^+ and then jumped to another gadget without going immediately to the middle vertex u^0. There is no way for such a path to come back and visit u^0 without repeating this same vertex, which would violate the definitoin of a Hamiltonian cycle. Once it goes to u^0, it must then proceed to the other vertex of the gadget.

To establish (ii), fix any vertex $u_0 \in V$. By (i) we know that the Hamiltonian cycle visits all three vertices of the corresponding gadget in consecutive order. Clearly it must visit the middle vertex in between the other two. By reversing the path (which we may do because G' is undirected) we may assume that this gadget is visited in the positive direction, $\langle u_0^-, u_0^0, u_0^+ \rangle$. The edges incident to u_0^+ go only to the entry ports of other gadgets, and hence the next gadget in the sequence must be visited in the positive direction as well. By extending this argument inductively, we see that all gadgets are entered along their input port and exitted along their output port, and hence are all traversed in the positive direction.

Suppose now that G' has a Hamiltonian cycle. By the above properties, we know that this cycle visits all the vertices of one gadget before going on to the next, and it visits all the gadgets in the positive direciton. Since the edges leaving any gadget's output port u^+ go to the input port v^- of another gadget if and only if there is an directed edge (u, v) in the original graph G, it is easy to see that the order in which any Hamiltonian cycle in G' visits the gadgets corresponds to a Hamiltonian cycle of the original vertices of G.

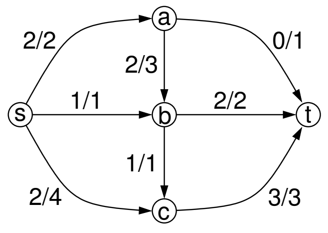
CMSC 451:Spring 2008 Dave Mount

Final Exam

This exam is closed-book and closed-notes. You may use two sheets of notes (front and back). Write all answers in the exam booklet. You may use any algorithms or results given in class. If you have a question, either raise your hand or come to the front of class. Total point value is 100 points. Good luck!

Problem 1. (15 points)

Consider the s-t network shown in the figure to the right. For each edge (u, v) the edge label f/c indicates a flow of f along this edge and a capacity of c.



(a) What is the *value* of this flow?

(b) Draw the *residual network* for this flow.

(c) Does there exist an *augmenting path*? If so, give an example of one, and show the new flow after augmentation. If not, explain how you know that there is none.

Problem 2. (30 points; 3–7 points each.) Short answer questions. Unless otherwise specified, explanations are not required, but may be given for partial credit.

(a) Consider the following summation, $f(n) = \sum_{i=1}^{n} i^3$. Which of the following three assertions are true? (Select any/all that apply.)

$$\text{(i): } f(n) = O(n^3) \qquad \text{(ii): } f(n) = O(n^4) \qquad \text{(iii): } f(n) = O(n^5)$$

(b) Suppose you are given an undirected graph which has n vertices, and each vertex has exactly six incident edges. As a function of n, what is the total number edges in this graph? (Give an exact answer for full credit, asymptotic answer for partial credit.)

(c) Suppose that you perform a DFS on a directed graph $G = (V, E)$, and for each vertex $u \in V$, we compute the discovery time $d[u]$ and finish time $f[u]$. Given that u is a descendent of v in the DFS tree, what can be said about the relationship between the discovery and finish times of these two vertices?

(d) You are given a connected, undirected graph $G = (V, E)$ in which each edge has a numeric edge weight, and all edge weights are distinct. Let e_1, e_2, and e_3 be the edges with smallest, second smallest, and third smallest weights among all the edges of G. Among these three edges, which *must* be in the minimum spanning tree (MST) of G and which *might* be in the MST.

(e) What does it mean to have a *memoized implementation* of a dynamic programming algorithm?

(f) The worst-case running time of the Ford-Fulkerson network flow algorithm is: (select any/all that apply.)

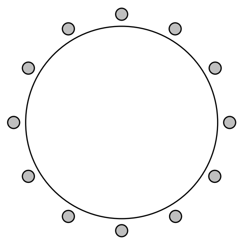
(i): $O(V^3)$ (ii): $O(V^2(E + V \log V))$ (iii): Depends on the edge capacities.

Problem 3. (15 points) You are given a collection of n points $U = \{u_1, u_2, \ldots, u_n\}$ in the plane, each of which is the location of a cell-phone user. You are also given the locations of m cell-phone towers, $C = \{c_1, c_2, \ldots, c_m\}$. A cell-phone user can connect to a tower if it is within distance Δ of the tower. For the sake of fault-tolerance each cell-phone user must be connected to at least three different towers. For each tower c_i you are given the maximum number of users, m_i, that can connect to this tower.

Give a polynomial time algorithm, which determines whether it is possible to assign all the cell-phone users to towers, subject to these constraints. Prove its correctness. (You may assume you have a function that returns the distance between any two points in $O(1)$ time.)

Problem 4. (10 points)

In ancient times, King Arthur had a large round table around which all the knights would sit. Unfortunately, some knights hate each other, cannot be seated next to each other. There are n knights altogether, $\{v_1, v_2, \ldots, v_n\}$, and the king has given you a list of pairs of the form $\{v_i, v_j\}$, which indicates that knights v_i and v_j hate each other.



You are asked to write a program to determine if it is possible to seat the knights about the table, called the *angry knight seating problem* (AKS). Prove that AKS is NP-complete. (Hint: The reduction is simple, and involves one of the following NP-complete problems: 3-coloring (3COL), independent set (IS), or undirected Hamiltonian cycle (UHC).)

Problem 5. (15 points) Recall the following problem, called the *Interval Scheduling Problem.* We are given a set $S = \{1, 2, \ldots, n\}$ of n *activity requests*, each of which has a given start and finish time, $[s_i, f_i]$. The objective is to compute the maximum number of activities whose corresponding intervals do not overlap. In class we presented an optimal algorithm greedy algorithm. We will consider some alternatives here.

(a) **Earliest Activity First** (EAF): Schedule the activity with the earliest start time. Remove all activities that overlap it. Repeat until no more activities remain.

Give an example to show that, not only is EAF not optimal, but it may be arbitrarily bad, in the sense that its approximation ratio may be arbitrarily high.

(b) **Shortest Activity First** (SAF): Schedule the activity with the smallest duration $(f_i - s_i)$. Remove all activities that overlap it. Repeat until no more activities remain.

Give an example to show that SAF is not optimal.

(c) Prove that SAF has an approximation ratio of 2, that is, it schedules at least half as many activities as the optimal algorithm.

Problem 6. (15 points) This is a variant of the Clique problem. You are given an undirected graph $G = (V, E)$. We say that a subset $V' \subseteq V$ is a *pseudo-clique* if for any two vertices $u, v \in V'$ the distance from u to v in G is at most two (that is, either there is an edge between them or they share a common neighbor). The *Pseudo-Clique Problem* (PC) is, given an undirected graph $G = (V, E)$ and an integer k, does G have a pseudo-clique of size k. Prove that PC is NP-complete. (Hint: Reduction from Clique.)

Solutions to the Final Exam

Solution 1:

(a) The value of the flow is the sum of flows coming out of s which is $2 + 1 + 2 = 5$.

(b) See the Fig. 1(a).

(c) The only augmenting path is $\langle s, c, b, a, t\rangle$ of bottleneck capacity 1. The resulting flow (which is maximum) is in Fig. 1(b).

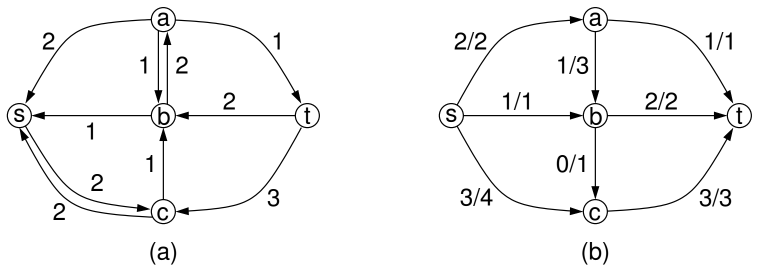


Figure 1: Solution to Problem 1.

Solution 2:

(a) Both (ii) and (iii) are true. First, to see that (i) is false, observe that there are at least half of the terms of the sum have value at least as large as $(n/2)^3$, which means that the sum is at least $(n/2)(n/2)^3 = n^4/16$, which is not $O(n^3)$. The largest term is n^3, and there are n terms, so the sum is at most n^4, and thus (ii) is true. Finally, any function that is $O(n^4)$ is $O(n^5)$, since O-notation simply provides an upper bound.

(b) Recall that the sum of vertex degrees in an undirected graph is equal to twice the number of edges (since summing the degrees of each vertex counts each edge twice). The sum of degrees is $6n$, and therefore there are $6n/2 = 3n$ edges in the graph.

(c) By the Parenthesis Lemma, u is a descendent of v if and only if $[d[u], f[u]] \subseteq [d[v], f[v]]$, that is $d[v] < d[u] < f[u] < f[v]$.

(d) The edges e_1 and e_2 must be in the MST, since Kruskal's algorithm will add them both, and these two edges cannot form a cycle. The edge e_3 might be in the MST, provided that it does not form a cycle with edges e_1 and e_2.

(e) In a memoized implementation, the dynamic program is implemented as a recursive function, but whenever a value is computed, it is stored (e.g., in a table) so it never needs to be computed again.

(f) (iii): As seen in the example from class, the number of augmentations may depend on the sizes of the edge capacities.

Solution 3: We will do this by reduction to network flow. (It is also possible to reduce to the circulation problem.) Create an s-t network $G = (V, E)$, where $V = U \cup C \cup \{s\} \cup \{t\}$. Create a unit capacity edge

(u_i, c_j) if cell-phone user $u_i \in U$ is within distance Δ of cell-phone tower $c_j \in C$. Create an edge from s to each $u_i \in U$ with capacity 3, and create an edge from $c_j \in C$ to t with capacity m_j.

Compute the maximum (integer) flow in this network. We claim that a feasible schedule exists if and only if all the edges exiting s are saturated in the maximum flow. To prove the "only if" part, we consider the following mapping between any valid assignment and a flow in G. We first observe that if there is a valid assignment, there is a valid assignment in which each user is assigned to exactly three towers (since if it was assigned to more than three, removing the additional assignments cannot violated any of the constraints of the problem). If user u_i is assigned to tower c_j, then $\text{dist}(u_i, c_j) \leq \Delta$, and so an edge exists between them. We apply one unit of flow along the edge (u_i, c_j). Since u_i is assigned to three towers, it has three units of flow entering it, and so the incoming edge (s, u_i) is saturated. Finally, since this is a valid assignment, the number of users assigned to any tower c_j is at most m_j, and so the flow leaving each c_j is at most m_j.

To prove the "if" part, suppose that G has a flow that saturates all the edges coming out of s. We show how to map this to a valid schedule. Because of the edge (s, u_i) is saturated, and the outgoing edges are of unit capacity, each user has three outgoing edges carrying flow. We assign u_i to these three towers. Since edges are created only when users and towers are within distance Δ, these are valid assignments. Since the flow coming out of tower j is at most m_j, tower j is assigned to more than m_j users. Thus, this is a valid assignment.

The running time of the algorithm is dominated by the time for the network flow, which is $O(V^3)$, where $|V| = m + n$.

Solution 4: We first model AKS as a graph problem. The vertices of the graph are the knights $V = \{v_1, v_2, \ldots, v_n\}$ and the edges are the pairs $\{v_i, v_j\}$ of knights who dislike each other. Let $G = (V, E)$ be the associated graph. Let $\overline{G}$ be the complement graph, in which two knights are connected by an edge if and only if they like each other. Any valid seating corresponds to a cycle in $\overline{G}$ that visits all the vertices of G, where consecutive vertices are joined by an edge (that is, the corresponding knights get along with each other). Thus, AKS is equivalent to the undirected Hamiltonian path problem (UHP) in $\overline{G}$. We won't bother to give all the details, because the NP-completeness of AKS follows directly from the NP-completeness of UHP.

Solution 5:

(a) To see that EAF may be arbitrarily bad, suppose that the first activity to start is very long and overlaps all the others. EAF will succeed in scheduling exactly one activity. If all the other tasks are non-overlapping, this will produce an approximation ration of $(n-1)/1 = n-1$, which can be arbitrarily high.

(b) Consider three intervals: $[0, 4]$, $[3, 6]$, and $[5, 9]$. The middle interval is the shortest, but overlaps both the other two, and thus SAF schedules one activity. The optimal algorithm, however, schedules both the first and last.

(c) To prove that SAF has an approximation ratio of 2, we will see that every time SAF schedules an activity, it might overlap at most two activities of the optimal schedule. Suppose that SAF manages to schedule k activities, and let $A = \langle a_1, a_2, \ldots, a_k \rangle$ be these activities. Let O be the optimum set of activities. We will show that $|O|/|A| \leq 2$.

When a_1 is selected, it is the shortest activity. Consider the subset O_1 of activities of O that overlap a_1 (it is possible that $O_1 = \{a_1\}$). Because the activities of O_1 are non-overlapping and of equal or longer duration than a_1, there can be at most two activities in O_1 (one overlapping a_1's start and the other overlapping a_1's finish). Now, remove a_1 and all overlapping activities from S (including the activities O_1). We can apply the same argument to the remaining set of activities, producing successive subsets

$O_2, O_3, \ldots, O_k$ of O, each of size at most 2. We have

$$|O| \;=\; \sum_{i=1}^{k} |O_i| \;\le\; 2k \;=\; 2|A|.$$

Therefore, $|O|/|A| \le 2$, as desired.

Solution 6: We first show that PC is NP-complete. The certificate consists of the subset V' of vertices in the PC. We can compute the distances between each pair of vertices in the graph (e.g., through the use of the Floyd-Warshall algorithm) and then verify that every pair of vertices in V' are at at most distance two from each other.

To show that PC is NP-hard, we reduce the standard Clique problem to it. To do this we need to translate the notion of two vertices being adjacent to the notion of two vertices being at distance two or less. This suggests the idea of inserting a vertex into the middle of each edge, which has the effect of doubling distances in the graph. However, this is not a solution, since there is nothing that controls the number of these new vertices that might be placed into the pseudo-clique. placing these newly created vertices into the pseudo-clique. (To see that this is a problem, suppose there is a vertex u in G of degree k. The k newly added vertices adjacent to u would form a pseudo-clique, even if G has no clique.) Rather than trying to keep these vertices out of the pseudo-clique, our approach will be to try to put them all in. To do this, we will connect them all together. The reduction is illustrated in Fig. 2.

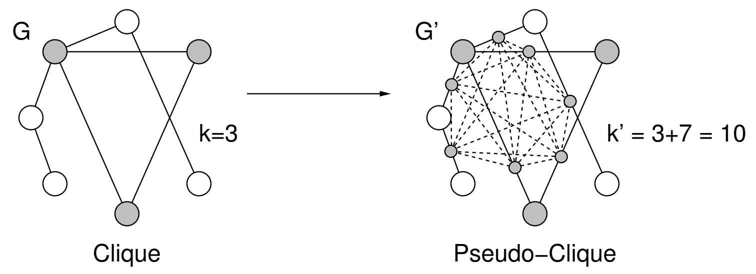


Figure 2: Reduction from Clique to PC.

Here is the reduction. Given the graph $G = (V, E)$ and integer k, we produce a new graph $G' = (W, E')$ and integer k' as follows. For each edge $\{u, v\} \in E$, we create an *middle-vertex* $m_{u,v}$. Let M be the set of all these middle vertices, and let $W = V \cup M$. For each edge $\{u, v\}$, we generate the edges $\{u, m_{u,v}\}$ and $\{m_{u,v}, v\}$ for E'. We also generate an edge between each pair of middle-vertices. We set $k' = k + |M|$, and output the pair (G', k'). The resulting graph has $V + E$ vertices and $O(2E + E^2) = O(E^2)$ edges, and clearly can be constructed in polynomial time.

To establish correctness, we assert that G has a clique of size k if and only if G' has a PC of size $k' = k + |M|$. It will simplify the proof to assume that G has no isolated vertices. (If G has any isolated vertices, we may ignore them, since they cannot contribute to the clique.)

If G has an clique V' of size k, then we assert that the union $V' \cup M$ forms a PC in G'. Since all the vertices of V' are adjacent in G, they are within distance two of each other in G'. Also, all the middle-vertices are adjacent to each other and (because there are no isolated vertices) they are within distance two of all the vertices of V, and hence within distance two of all the vertices of V'. Thus $V' \cup M$ is a PC in G'.

Conversely, if G' has a PC of size k', we claim that the PC may be assumed to contain all the middle-vertices. The reason is every middle vertex is within distance two of every vertex in G', and hence they may all be freely added to any PC. If we ignore all these vertices, the remaining vertices, call them V' consist of k vertices of G. These must be within distance two of each other, which implies that they must have been adjacent in G (for otherwise they would be at distance at least three) in G'. Therefore, V' forms a clique of size k in G.