Problems 1 through 9 are from material before Quiz 2 (up to dynamic programming), and problems 10-21 are from material after Quiz 2.

Problem 1. True or False
Answer True or False to the following questions:

(a) Every non-empty DAG has at least one source.

(b) Suppose we have a graph where each edge weight value appears at most twice. Then, there are at most two minimum spanning trees in this graph.

(c) A dynamic programming problem where we need to solve \( O(n^2) \) subproblems will take \( O(n^2) \) time.

Solution

(a) True. Start at an arbitrary vertex \( v \), and repeatedly go backwards in the graph. If there is no source, you can keep going forever. But that means some vertex will be visited twice, and in particular, you’ll have found a cycle in the graph - so it couldn’t have been a DAG in the first place.

(b) (Berkeley CS 170) False. Consider the following graph. The MST has weight 11, and can be any of \( b - a - c - d - e - f \), \( b - a - c - d - f - e \), \( a - b - c - d - f - e \), or \( a - b - c - d - e - f \).

(c) False. Each subproblem may take more than \( O(1) \) time to compute.

Problem 2. Big O Only
Is it possible to find two functions \( f(n), g(n) \) such that \( f(n) = O(g(n)) \), but \( f(n) \neq o(g(n)) \), \( f(n) \neq \Omega(g(n)) \)? If so, give an example. If not, give a proof.

Solution

The challenge here is to come up with 2 functions such that big O is the only relationship that holds between them. An example of such a pair of functions is \( f(n) = n\sin n \) and \( g(n) = n \). It is clear that \( f(n) \leq 1 \cdot g(n) \). It is also clear that \( f(n) \neq o(g(n)) \) because the limit of the ratio does not exist. We also have that \( f(n) \neq \Omega(g(n)) \) because \( f(n) \) can be arbitrarily small when \( \sin n = -1 \) for example.

Problem 3. Cyclic Sorted Array
You’re given an array \( A \) of \( n \) distinct numbers that is sorted up to a cyclic shift, i.e. \( A = (a_1, a_2, \ldots, a_n) \)
where there exists a $k$ such that $a_k < a_{k+1} < \ldots < a_n < a_1 < \ldots < a_{k-1}$, but you don’t know what $k$ is. Give an efficient algorithm to find the largest number in this array.

**Solution**

Here’s a recursive solution based on binary search that runs in $O(\log n)$ time.

First, we check that the array isn’t cycled at all if $a_n > a_0$, then we can return the last element (which will be the greatest), $a_n$. Otherwise, $a_0 > a_n$, and start at the middle element $a_m$ of the entire array. If $a_m > a_0$, then move to the right half of the array and recurse. Otherwise, if $a_m < a_0$, move to the left half of the array and recurse.

**Problem 4. Counting Permutations**

How many permutations of $\{1, 2, \ldots, N\}$ have exactly $K$ inversions? An inversion is a pair of numbers $i, j$ in the permutation such that $i > j$ but $i$ comes before $j$. For example, for $N = 3$ and $K = 1$ the answer is 2: $\{2, 1, 3\}$ and $\{1, 3, 2\}$.

**Solution**

We use dynamic programming on $N$ and $K$, and at each step, we decide where the smallest number goes.

- **Definition:** Define $X[i, j]$ to be the number of permutations of $\{1, 2, 3, \ldots, i\}$ that have exactly $j$ inversions. Our desired answer is $X[N, K]$.

- **Recurrence:** Consider the element $i$. Given the any permutation of $1, 2, \ldots, i - 1$, we can insert $i$ into any of the $i$ slots (i.e. before 1, between 1 and 2, between 2 and 3... after $i - 1$). Inserting all the way at the beginning causes $i - 1$ inversions, while inserting all the way at the end incurs 0 inversions. Therefore, we have:

  \[
  X[i, j] = \begin{cases} 
  0 & j < 0 \\
  1 & i = 0 \text{ or } j = 0 \\
  \sum_{\ell=0}^{i-1} X[i-1, j-\ell] & \text{otherwise}
  \end{cases}
  \]

- **Analysis:** The run-time is $O(N^2 K)$ because there are $O(NK)$ states of the DP and each takes $O(N)$ time to compute. The space complexity is naively $O(NK)$, but since we are only interested in the count of the number of permutations and not the permutations themselves, we can get away with $O(K)$ space by only storing each row $X[\cdot, K]$ of the table at a time.

**Problem 5. Dijkstra’s Variations**

In lecture, we showed that Dijkstra’s algorithm may not work correctly if negative edge weights are present. In this problem. Let $G = (V, E)$ be a weighted, directed graph with no negative-weight cycles. For each of the following property of $G$, give an algorithm that finds the length the shortest path from $s$ to all other vertices in $V$. In both cases, your algorithm should have the same run-time as that of Dijkstra’s.

(a) $G$ only has one negative edge. Give an algorithm to find length of the shortest path from $s$ to all other vertices in $V$.

(b) The edges leaving from the source $s$ may have negative weights, but all other edges in $E$ have non-negative weights,
Solution

(a) Let’s say the negative-weight edge is \((u, v)\). First, remove the edge and run Dijkstra from \(s\). Then, check if \(d_s[u] + w(u, v) < d_s[v]\). If not, then we’re done because it implies that the shortest path tree does not use this negative edge.

If yes, then run Dijkstra from \(v\), with the negative-weight edge still removed. Let the distances obtained by that be \(d_v[.]\). Then, for any node \(t\), its shortest distance from \(s\) will be \(\min(d_s[t], d_s[u] + w(u, v) + d_v[t])\). This works because the shortest path to any node is either that where you use the negative edge or you do not. If you do use the negative edge, then the shortest path consists of the shortest path to \(u\), using \((u, v)\) and then the shortest path from \(v\) to \(t\).

(b) The claim is that running Dijkstra’s as usual will work. Let’s compare the execution of Dijkstra’s algorithm on \(G\) to the execution on a graph \(G'\) where we’ve added the same amount \(m\) to the weights of all edges coming out of the source so as to make them non-negative.

On the one hand, since all edge weights in \(G'\) are non-negative, we know that Dijkstra’s algorithm will find the correct shortest-path distances in \(G'\).

On the other hand, observe that the flow of Dijkstra’s algorithm is controlled by two things: the comparisons \(\text{dist}[w] > \text{dist}[v] + \text{length}[v, w]\), and the element with lowest priority in the heap.

Observe that at any step in Dijkstra’s algorithm, in the comparison \(\text{dist}[w] > \text{dist}[v] + \text{length}[v, w]\) we’re actually comparing the lengths of two paths from \(s\) \(w\). Whenever \(w \neq s\), both of these paths contain exactly one edge coming out of \(s\). So the truth value of the comparison will be the same in \(G\) and \(G'\) because the difference is \(m\) on both sides of the comparison. Moreover, once we pop \(s\) from the queue, the queue for \(G'\) will be a copy of the queue for \(G\), but with all priorities shifted up by \(m\). This preserves the minimal element.

It follows that Dijkstra’s algorithm is the same on \(G\) and \(G'\), up to the priorities of all vertices \(v \neq s\) shifted by \(m\). Thus, Dijkstra’s algorithm on \(G\) will find the shortest paths in \(G'\). But the shortest paths in \(G'\) to all \(v \neq s\) are the same as those in \(G\) by an argument similar to the above: if we have a path \(p : s \rightarrow v\) which is shortest in \(G'\), this means that \(l'(p) \leq l'(q)\) for any \(q : s \rightarrow v\), where \(l'\) is distance calculated in \(G'\), and \(l\) is distance calculated in \(G\). But then, \(l(p) = l'(p) - m \leq l'(q) - m = l(q)\) for all such \(q\) in \(G\), hence \(p\) is a shortest path in \(G\).

Finally, it remains to show that we find the right distance to \(s\) itself in \(G\) - which is clear because we assumed \(G\) has no negative cycles, so the true distance is 0, and this is what Dijkstra’s algorithm gives.

Problem 6. Largest Submatrix of 1’s
You are given a matrix \(M\) with \(m\) rows and \(n\) columns consisting of only 0’s and 1’s. Give an efficient algorithm for find the maximum size square sub-matrix consisting of only 1’s.

Solution
We will solve this via dynamic programming:

- **Definition:** Let \(S[i, j]\) be the size of largest square of 1’s submatrix containing \(M[i, j]\) in the
right-most and bottom-most corner. Note that if \( M[i, j] = 0 \), then \( S[i, j] = 0 \). The desired answer is \( \max_{i,j} S[i, j] \).

- **Recurrence:** We can then copy the first row and column from \( M \) to \( S \) because at most it will be a square of size \( 1 \times 1 \). This is equivalent to saying that if \( i = 1 \) or \( j = 1 \), then we have that \( S[i, j] = M[i, j] \). For the recursive case, we can then fill \( S \) according to

\[
S[i, j] = \begin{cases} 
\min( S[i, j - 1], S[i - 1, j], S[i - 1, j - 1]) + 1 & : M[i, j] = 1 \\
0 & : M[i, j] = 0 
\end{cases}
\]

It is pretty easy to see why this is correct after drawing an example.

- **Analysis:** This will take \( O(mn) \) time and \( O(m + n) \) space since we only need to keep the last two diagonals.

**Problem 7. Minimum Bottleneck Spanning Tree**

Let \( G = (V, E) \) be an undirected graph with weights \( w_e \) on the edges. A **minimum bottleneck spanning tree** of \( G \) is a spanning tree such that the weight of the heaviest edge is as small as possible. Note that for minimum bottleneck spanning trees, we do not care about the sum of the weights of the chosen edges—only the maximum edge.

(a) Show that any minimum spanning tree is necessarily a minimum bottleneck spanning tree.

(b) Give an example of a graph and a minimum bottleneck spanning tree that is not a minimum spanning tree.

**Solution**

(a) Assume that there is an MST \( T \) of a graph \( G \). Let \( T' \) be an MBST of \( G \). Define \( b(T) \) and \( b(T') \) to be the weight of the bottleneck edge in \( T \) and \( T' \). Let \( e \) be the bottleneck edge of \( T \). Consider the cut in the tree defined by \( e \). By the cut property every other edge crossing the cut necessarily has weight at least that of \( e \). Now look at the edges of \( T' \) which cross the cut. By the above statement they must too all have weight at least that of \( c(e) \). But then if \( b(T) \) is the bottleneck cost of a tree \( T \), \( b(T) = c(e) \). Thus, we know that \( b(T) \leq \) the cost of any edge of \( T' \) crossing the cut \( \leq b(T') \). Therefore, we have shown that \( b(T) \leq b(T') \). We also know that by definition of minimum bottleneck spanning tree that \( b(T') \leq b(T) \). Therefore, \( b(T') = b(T) \) and \( T \), a MST, is also an MBST.

\[ \leq \text{cost of any edge of } T_0 \text{ crossing the cut } \leq b(T_0) \leq b(T_0), \text{ and so since } T_0 \text{ has minimum bottleneck cost among trees } (b(T_0)) \leq b(T)), b(T) = b(T_0) - T \text{ is an MBST.} \]

(b) Consider the following graph:

```
     a --- d
       | 1 |
       |   |
       | 2 |
     b --- c
       | 2 |
```

4
Notice that the minimum spanning tree has weight 4 and has bottle-neck edge 2. However, $d - a - b - c$ is a minimum bottle neck spanning tree because it has heaviest edge 2 and it is impossible to find any other spanning trees that have a lighter bottle neck edge.

**Problem 8. Efficient Waiters**

A restaurant has $n$ tables and 2 waiters. Table $i$ is located at coordinate $(x_i, y_i)$ in the restaurant for $i = 1, 2, \ldots, n$. At the beginning, one waiter is at table 1 and the other waiter is at table 2. We will then receive a sequence of $m$ queries. Each query is simply a number in $\{1, 2, 3, \ldots, n\}$ representing that the customer at that table needs some kind of service. Hence, one of the two waiters needs to be service that table. The waiters are complaining of walking too much and so the restaurant has hired you to help them be more efficient.

Suppose an omniscient being has told you in advance the order in which the tables would need servicing. Let $t_1, t_2, \ldots, t_m$ be the sequence of queries (i.e. tables that need servicing). The goal is to minimize the total walking distance of the two waiters, where the distance between two tables is the standard Euclidean distance. Assume that there is enough time between the queries for whichever of the 2 waiters is chosen to walk to the table and service it.

(a) Consider the following greedy algorithm. In order to service table $t_i$, we will have whichever waiter is closer to service that table. Give an example to show that this greedy algorithm is not optimal.

(b) Give a $O(n^2m)$ time and $O(n^2)$ space solution that finds the value of the shortest total walking distance.

**Solution**

(a) The intuition behind why the greedy algorithm is incorrect is that it might be better to move a waiter that is further away to service a table because the next query is close to where that waiter was originally. Here’s a simple example:

Let $(x_1, y_1) = (0, 0), (x_2, y_2) = (5, 0)$ and $(x_3, y_3) = (2, 0)$. Recall that waiter 1 starts at table 1 $(0, 0)$ and waiter 2 starts at table 2 $(5, 0)$. Suppose $m = 2$ and there are 2 queries: $t_1 = 3$ and $t_2 = 1$.

The greedy algorithm would move waiter 1 to table 3 first to service that query. This has a distance of 2. Then, when table 1 is queried, the greedy algorithm will move waiter 1 back to table 1, costing a distance of 2. The total distance moved of the two waiters is 4. A more optimal solution would have been to just move waiter 2 to table 3 at the beginning. Then, when table 1 needs servicing, nothing needs to be moved at all!

(b) We will use dynamic programming:

- **Definition:** Let $f(k, a, b)$ be the smallest possible walking distance for servicing requests $t_k, t_{k+1}, \ldots, t_m$ given that waiter 1 is at table $a$ and waiter 2 is at table $b$. The final answer is $f(1, 1, 2)$.

- **Recurrence:** When we receive request $t_k$, we must choose one of two things to do: move waiter 1 there or waiter 2 there. Once that has happened, we have incurred some cost, but
then we are in another state of the same problem. Let \( d(i, j) \) be the Euclidean distance between \((x_i, y_i)\) and \((x_j, y_j)\).

\[
f(k, a, b) = \begin{cases} 
0 & k > m \\
\min \left[ f(k + 1, t_k, b) + d(a, t_k), f(k + 1, a, t_k) + d(b, t_k) \right] & k \leq m 
\end{cases}
\]

- There are \( O(n^2 m) \) states to the DP and each takes \( O(1) \) time to compute. Therefore, the run-time is \( O(n^2 m) \). The space complexity is \( O(n^2) \) because we each \( f(k, \cdot, \cdot) \) only depends on \( f(k - 1, \cdot, \cdot) \) and so we only need to store one row at a time.

**Problem 9. Significant Inversions (Kleinberg-Tardos 5.2)**

Call a significant inversion in an array \( a \) a pair \( i < j \) such that \( a_i > 2a_j \). Give an \( O(n \log n) \) algorithm to count the number of significant inversions in an array.

**Solution**

We can use a mergesort-like algorithm, only at each recursive level we perform two merges instead of one. Suppose we have the left half \( \ell \) and right half \( r \) at a current recursive step. Then let \( r' \) be the same list as \( r \) except with each element doubled; merging \( \ell \) and \( r' \) will allow us to count the number of significant inversions. Then we merge \( \ell \) and \( r \) and continue as usual.

This uses the same recurrence relation as mergesort, because two merges still takes \( O(n) \) time. Thus this algorithm takes \( O(n \log n) \) time.

**Problem 10. True or False**

Answer True or False to the following questions:

(a) If an iteration of the Ford-Fulkerson algorithm on a network places flow 1 through an edge \((u, v)\), then in every later iteration, the flow through \((u, v)\) is at least 1.

(b) If VertexCover reduces to some yes-no problem \( A \), \( A \) is NP-complete.

(c) If all of the edge capacities in a graph are an integer multiple of 7, then the value of the maximum flow will be a multiple of 7.

(d) Skip lists are more useful than balanced binary search trees in solving the predecessor problem because they can query in \( O(\log n) \) time.

**Solution**

(a) **False.** A later augmenting path may pass through \((v, u)\), causing the flow on \((u, v)\) to be decreased.

(b) **False.** \( A \) is NP-hard, but is not necessarily in NP.

(c) **True.** Each iteration of Ford-Fulkerson will increase the value of the flow by an integer multiple of 7.

(d) **False.** BBSTs also solve the predecessor in \( O(\log n) \) time, and in fact skip lists only achieve \( O(\log n) \) expected time. Skip lists provide a simpler randomized solution.
Problem 11.
Run Ford-Fulkerson on the following graph from $s$ to $t$, using the method of augmenting paths discussed in class. What is the final flow? Show the residual network at the final step of building the flow. It also may be helpful to show intermediate residual networks.

\[ \begin{array}{c}
\text{s} & \text{a} & \text{d} & \text{t} \\
6 & 3 & 2 & 1 \\
2 & 1 & 1 & 1 \\
\text{b} & \text{c} & & \\
\end{array} \]

Solution
First we send 3 units of flow along $(s, a)$ and $(a, t)$. Then we send 1 unit of flow along $(s, b), (b, c)$, and $(c, t)$. This gives us a total flow of 4.

1. 3 along $s \rightarrow a \rightarrow t$.

\[ \begin{array}{c}
\text{s} & \text{a} & \text{d} & \text{t} \\
3 & 3 & 3 & 2 \\
1 & 1 & 1 & 1 \\
\text{b} & \text{c} & & \\
\end{array} \]

2. 1 along $s \rightarrow b \rightarrow c \rightarrow t$.

Problem 12. Balls into Bins (CLRS 5.4-6)
Suppose that $n$ balls are tossed into $n$ bins, where each toss is independent and the ball is equally likely to end up in any bin. Hint: Use indicator random variables.

(a) What is the expected number of empty bins?

(b) What is the expected number of bins with exactly one ball?

Solution
(a) Let $X$ be the expected number of empty bins. Then we can write $X = \sum_{j=1}^{n} I_j$ where $I_j$ is the indicator of the $j$th bin being empty. Then by linearity of expectation, and the expectation of
an indicator being equivalent to the probability of the event:

\[ E[X] = E \left[ \sum_{j=1}^{n} I_j \right] = \sum_{j=1}^{n} E[I_j] = \sum_{j=1}^{n} P(\text{bin } j \text{ is empty}) = nP(\text{bin 1 is empty}) \text{ (by symmetry)} = n \times \left( \frac{n-1}{n} \right)^n, \]

because for bin 1 to be empty, each ball must not land in it. The probability of a specific ball not landing in bin 1 is \((n-1)/n\), and all the ball tosses are independent.

(b) Similarly let \( Y \) be the expected number of bins with exactly one ball and \( I_j \) be the indicator of bin \( j \) having exactly one ball.

\[ E[Y] = E \left[ \sum_{j=1}^{n} I_j \right] = \sum_{j=1}^{n} E[I_j] = \sum_{j=1}^{n} P(\text{bin } j \text{ has 1 ball}) = nP(\text{bin 1 has 1 ball}) \text{ (by symmetry)} = n \times n \times \frac{1}{n} \times \left( \frac{n-1}{n} \right)^{n-1} = n \times \left( \frac{n-1}{n} \right)^{n-1} \]

Note that the number of balls in a bin is distributed Binomial\((n, 1/n)\), so the probability that bin 1 has 1 ball follows.

**Problem 13. Shortest Path LP**

Suppose you are given a weighted, directed graph \( G = (V, E) \), with weight function \( w \) mapping edges to positive real-valued weights, a source vertex \( s \), and a destination vertex \( t \). Show how to compute the value \( d[t] \), which is the weight of a shortest path from \( s \) to \( t \), by linear programming. 

*Hint*: Have variables \( x_{(u,v)} \) for each edge and think about constraints similar to that of flow.

**Solution**

Use a variable \( x_{(u,v)} \) for every edge. \( x_{(u,v)} \) is 1 when \((u, v)\) is in the shortest path from \( s \) to \( t \) and 0
otherwise:

\[
\text{minimize } \sum_{(u,v)} w_{(u,v)} x_{(u,v)}
\]

subject to

\[
\sum_{w} x_{(s,w)} = 1
\]

\[
\sum_{u} x_{(u,t)} = 1
\]

\[
\sum_{u} x_{(u,v)} - \sum_{w} x_{(v,w)} = 0 \text{ for every vertex } v \notin \{s, t\}
\]

\[
x_{(u,v)} \geq 0 \text{ for every edge } (u, v)
\]

Notice that this might give you a fractional solution. However, I claim that given any fractional solution, I can find an integral solution with the same value. Given a fractional solution \(\bar{x}_{(u,v)}\), construct a graph where edge \((u, v)\) has weight \(\bar{x}_{(u,v)}\). Notice that the max flow of this graph is 1 due to our LP constraints. Run a Ford-Fulkerson-like algorithm to decompose this flow into augmenting paths, where path \(p_i\) is the set of edges in the path, and path \(p_i\) has flow \(f_i\). Note that \(\sum_i f_i = 1\) because our max flow is 1.

Define \(\tilde{x}_{i,(u,v)}\) to be \(f_i\) if \((u, v) \in p_i\) and 0 otherwise. Now let \(i' = \arg \min_i \frac{1}{f_i} \sum_{(u,v)} w_{(u,v)} \tilde{x}_{i,(u,v)}\) = \(\arg \min_i \sum_{(u,v) \in p_i} w_{(u,v)}\). This gives us the shortest path \(p_{i'}\) of all the augmenting paths.

We can decompose the value of the objective function at \(\bar{x}_{(u,v)}\) as:

\[
\sum_{(u,v)} w_{(u,v)} \bar{x}_{(u,v)} = \sum_i \sum_{(u,v) \in p_i} \tilde{x}_{i,(u,v)} w_{(u,v)}
\]

\[
= \sum_i f_i \sum_{(u,v) \in p_i} w_{(u,v)}
\]

\[
\geq \sum_i f_i \sum_{(u,v) \in p_{i'}} w_{(u,v)} = \sum_{(u,v) \in p_{i'}} w_{(u,v)}
\]

Now we have an integer solution \(x_{(u,v)} = 1\) for \((u, v) \in p_{i'}\) with objective function less or equal to the fractional solution. Thus the objective function for the fractional and integral solutions are equal, so the value of the LP solution corresponds to the weight of the true shortest path.

**Alternative solution:** The hint didn’t suggest this formulation, but this is another way to solve the problem without worrying about integrality (but still won’t return the shortest path itself). Consider variables of the form \(d_u\) representing the shortest distance from \(s\) to the vertex \(u\). Then we can formulate the following LP:

\[
\text{maximize } d_t
\]

subject to

\[
d_s = 0
\]

\[
d_v \leq d_u + w_{(u,v)} \text{ for every edge } (u, v)
\]

The maximization as the objective function may seem weird, but intuitively the constraints are doing the minimizing rather than the objective function. If we minimized rather than maximized, we would get \(d_v = 0\) for all \(v\). This formulation is more analogous to our shortest-path algorithms than the flow-like formulation above.
Problem 14. Building Hospitals

Suppose we are given a set of cities represented by points in the plane, \( P = \{ p_1, \ldots, p_n \} \). We will choose \( k \) of these cities in which to build a hospital. We want to minimize the maximum distance that you have to drive to get to a hospital from any of the cities \( p_i \). That is, for any subset \( S \subset P \), we define the cost of \( S \) to be

\[
\max_{1 \leq i \leq n} \text{dist}(p_i, S) \quad \text{where} \quad \text{dist}(p_i, S) = \min_{s \in S} \text{dist}(p_i, s).
\]

This problem is known to be NP-hard, but we will find a 2-approximation. The basic idea: Start with one hospital in some arbitrary location. Calculate distances from all other cities — find the “bottleneck city,” and add a hospital there. Now update distances and repeat.

(a) Give a precise description of this algorithm, and prove that it runs in time \( O(nk) \).

(b) Prove that this is a 2-approximation.

Hint: Any two points within a circle of radius \( r \) are at most \( 2r \) away from each other.

Solution

(a) For each city \( p_i \) we let \( d_i \) denote its distance from the set of hospitals so far. Suppose we have just added the \( j \)-th hospital \( s_j \). To decide where to add the next hospital, we look at \( O(n) \) cities, updating the distances \( d_i \) by taking \( d_i = \min(d_i, \text{dist}(p_i, s_j)) \). The bottleneck city is the one with maximal \( d_i \) after this update, so we set \( s_{j+1} \) to be this city. We continue until \( k \) hospitals have been added, and we do a last update of the distances to find the final cost of our choice of \( S \).

(b) Consider the optimal choice of \( S^* = \{ s_1, \ldots, s_k \} \), and let \( D_j \) denote the disc with center \( s_j \) and radius \( r^* \), where \( r^* \) denotes the cost of \( S^* \). Now consider the \( S \) that we obtained from our approximation algorithm, which has cost \( r \). If every \( D_j \) contains some point of \( S \) then we are done, because then every \( p_i \) will be within distance \( 2r^* \) of \( S \). Suppose not: By the pigeonhole principle, two points \( s, t \in S \) must be contained in a single disc \( D = D_j \). But by the way we constructed \( S \), \( \text{dist}(s, t) \geq r \) because \( s \) and \( t \) both have hospitals. We also have \( \text{dist}(s, t) \leq 2r^* \), so again \( r \leq 2r^* \) which concludes the proof.

Problem 15. NP-Completeness

Show that the following 2 problems are NP-complete:

(a) **Subgraph isomorphism.** Consider graphs \( G = (V, E), H = (V_2, E_2) \). Does \( G \) contain a subgraph \((V_1, E_1)\) which is isomorphic to \( H \)?

(An isomorphism of graphs \((V_1, E_1)\) and \((V_2, E_2)\) is a 1-1 function \( f : V_1 \rightarrow V_2 \) such that the map \((u, v) \mapsto (f(u), f(v))\) is a 1-1 function \( E_1 \rightarrow E_2 \).)

(b) **4-Exact-cover.** Given a finite set \( X \) of size \( 4q \) and a collection \( C \) of 4-element subsets of \( X \), does \( C \) contain an exact cover (i.e. a partition) of \( X \)?

You may use without proof that 3-Exact-cover is NP-complete.

Solution
(a) This is clearly in NP because our certificate is the subgraph and isomorphism $f$: we can apply the isomorphism to the subgraph and check for equality of the subgraph and $H$. To show this is NP-complete, we reduce from clique: Asking whether there is a CLIQUE of size $\geq K$ is equivalent to asking whether there is a subgraph isomorphic to $C_K$, the complete graph on $K$ vertices.

Another possible reduction is from the HAMILTONIANPATH. Asking whether there is a Hamiltonian path in graph $G = (V, E)$ is equivalent to asking the subgraph isomorphism question on the graphs $G$ and the Hamiltonian cycle (simple cycle) on $|V|$ vertices.

(b) 4-EXACT-COVER is in NP because we can use the partition as a certificate and check if it is an exact cover. To prove NP-hardness, we reduce 3-EXACT-COVER to 4-EXACT-COVER. Expand $X$ to $X'$ by adding the elements $e_1, \ldots, e_q$. For each 3-element set $S \in \mathcal{C}$, define 4-element sets $S(i) = S \cup \{e_i\}$ for $1 \leq i \leq q$. Solving 4-EXACT-COVER for $X'$ with

$$C' = \{S(i) : S \in \mathcal{C}, 1 \leq i \leq q\}$$

will give the result for 3-EXACT-COVER.

Problem 16. Two Player Game

Consider the following zero-sum game, where the entries denote the payoffs of the row player:

$$\begin{bmatrix}
2 & -4 & 3 \\
1 & 3 & -3
\end{bmatrix}$$

Write the column player’s minimization as an LP. Then, write the dual LP, which is the row player’s maximization problem.

Solution

Let $p$ be the payoff of the row player. The column player wishes to minimize $p$ with to a strategy of $(y_1, y_2, y_3)$ (probabilities of picking column 1, 2, 3 respectively). Recall that for any given strategy of one player, the best response of the other player will occur when the other player plays one of their pure strategies. That’s why in the constraints, we only consider the opponent playing pure strategies.

Thus, the column player’s LP is

$$\begin{align*}
\min p \\
& p \geq 2y_1 - 4y_2 + 3y_3 \\
& p \geq y_1 + 3y_2 - 3y_2 \\
y_1 + y_2 + y_3 &= 1, \\
y_1, y_2, y_3 &\geq 0.
\end{align*}$$
On the other hand, let the row player’s strategy be \((x_1, x_2)\). Then, the dual LP is

\[
\begin{align*}
\text{max } p \\
p &\leq 2x_1 + x_2 \\
p &\leq -4x_1 + 3x_2 \\
p &\leq 3x_1 - 3x_2 \\
x_1 + x_2 &\leq 1 \\
x_1, x_2 &\geq 0.
\end{align*}
\]

**Problem 17. Edge Connectivity**

The edge connectivity of an undirected graph is the minimum number \(k\) of edges that must be removed to disconnect the graph. For example, the edge connectivity of a tree is 1, and the edge connectivity of a cyclic chain of vertices is 2. Let \(V = \{v_0, v_1, \ldots, v_n\}\). For each \(i\), consider the flow network \(F_i\) where \(v_0\) is the source, \(v_i\) is the sink, and both \((u, v)\) and \((v, u)\) have capacity 1 for each \((u, v) \in E\). Let \(f_i\) be the total flow of \(F_i\). Prove that the edge connectivity is \(\min_i f_i\).

**Solution**

We will show the inequality in both directions:

(a) \(\min_i f_i \geq \) edge connectivity: consider flow network \(F_j\) that has the minimal flow. Now consider the min-cut derived from that maximum flow. When the edges in that cut had capacity zero, the sink is not reachable from the source. Hence, erasing those edges would render a disconnected graph.

(b) edge connectivity \(\geq \min_i f_i\): Suppose \(k\) is the minimum amount of edges to render \(G\) disconnected and consider the graph \(G'\) which is \(G\) without the \(k\) edges that makes it disconnected. Let \(v_j\) be a vertex that is not reachable from \(v_0\) in \(G'\). Then the flow network \(f'_j\) defined analogous to \(f_j\) for the graph \(G'\) must have flow 0, otherwise there would be a path connecting source \(v_0\) and sink \(v_j\). However, notice that the flow of \(f_j\) is at most \(k + \) the flow of \(f'_j\), so the flow of \(f_j\) \(\leq k\). So: \(\min_i f_i \leq f_j \leq k\).

**Problem 18. MST Bound**

Suppose that we are given a graph with the following guarantee: for any two disjoint sets \(S, T\) of vertices, there exists some edge between a vertex in \(S\) and one in \(T\) with length no more than \(\frac{1}{\max(|S|,|T|)}\). Find a bound on the size of the minimum spanning tree for this graph, in terms of the number of vertices. *Hint:* Think Prim’s algorithm. You may find the following approximation useful:

\[
\sum_{i=1}^{n} \frac{1}{i} \approx \ln(n)
\]

**Solution**

We create a tree similarly to that of Prim’s algorithm. In particular, start with sets of vertices of size 1 and \(n - 1\); we know that there exists an edge of size no more than \(1/(n - 1)\) from the single vertex to one of the other vertices. Add that edge. Now, we have sets of size 2 (connected) and \(n - 2\)
(disconnected), and can find an edge of size no more than $1/(n - 2)$ from one of the two vertices into a disconnected one. We can continue this process until we have one connected set of size $\lceil n/2 \rceil$ (the tree so far) and one entirely disconnected set of $\lfloor n/2 \rfloor$ vertices.

This is the point at which we switch from using the bound $1/($# disconnected vertices$)$ for a bound on the edge length to $1/($# vertices in tree so far$)$.

Now there is some disconnected vertex of distance at most $1/\lceil n/2 \rceil$ from our tree so far, then one of $1/\lceil n/2 + 1 \rceil$, etc. Finishing out this process gives a total edge length of

$$2 \sum_{i=1}^{n-1} \frac{1}{i} \leq 2 \sum_{i=1}^{n-1} \frac{1}{i} - 2 \sum_{i=1}^{\lceil \frac{n-1}{2} \rceil} \frac{1}{i} \approx 2 \ln(n - 1) - 2 \ln \left( \lceil \frac{n}{2} \rceil - 1 \right) \approx 2 \ln 2$$

This does not depend on $n$!

**Problem 19. $s$ – $t$ Min Cut**

Suppose we try to use Karger’s contraction algorithm not for global min-cut, but for $s – t$ min cut in an undirected, unweighted graph. We do the same algorithm, but we modify it so that if it ever tries to contract an edge from the $s$-supervertex to the $t$-supervertex, we don’t do it (i.e. we always choose a random edge to contract amongst the edges which don’t connect $s$ to $t$).

(a) Prove that there are graphs such that the probability that this modified Karger’s algorithm finds an $s – t$ min cut is exponentially small.

(b) Show that unlike with global min-cut, the number of $s – t$ min cuts is not $O(n^2)$ but can be much bigger.

**Solution**

(a) Consider a graph where $s$ and $t$ are each connected to the other “middle” $n – 2$ vertices, and these other vertices are also arranged in a cycle. Then there are only 2 $s – t$ min cuts: $\{s\}, V - \{s\}$ and $\{t\}, V - \{t\}$ of size $n – 2$. Every other cut has weight at least $n$, because for every middle vertex has one edge cut (to either $s$ or $t$, each cut must be at least weight $n – 2$), and if the entire cycle isn’t on the same side of the cut, then at least 2 edges will cross the cut.

Then to return a min $s – t$ cut, the algorithm must either (1) never cut an edge touching $s$ or (2) never cut an edge touching $t$. This happens with exponentially small probability. Let $C$ be the $\{s\}, V - \{s\}$ cut. Then analogously to in lecture:

$$P(C \text{ is output}) = P(C \text{ survives every recursive level})$$

$$= \prod_{i=0}^{n-2} P(C \text{ survives level } i|C \text{ survived levels } 0, 1, \ldots, i – 1)$$

$$= \prod_{i=0}^{n-2} \frac{2(n-2) - i}{3(n-2) - i} < \left( \frac{2}{3} \right)^{n-2} = O \left( \left( \frac{2}{3} \right)^{n-2} \right).$$

Note there are originally $3(n-2)$ edges, and $n – 2$ of them come out of $s$.  

13
(b) Note the maximum number of $s - t$ min cuts is $2^{n-2}$, because every vertex which isn’t $s$ or $t$ needs to choose to either be on the $s$ side or the $t$ side. This upper bound is achievable: consider the graph where $s$ has an edge to every non-$t$ vertex and $t$ also has an edge to every non-$t$ vertex. Then the size of the min cut is $n - 2$, and every valid cut achieves this.

**Problem 20. Scheduling Jobs**

There are $n$ jobs that we would like to schedule on $m$ machines. Job $i$ requires processing time $p_i$ and each job must be exactly assigned to one machine. The completion time of a machine is then the sum of processing time of each job assigned to it and the load of the assignment is the maximum completion time of any machines in this assignment. We want to minimize load. Prove that there is a greedy algorithm that achieves completion time at most $(2 - \frac{1}{m})$ times the optimal completion time.

**Solution**

Let $p_{\text{max}}$ be the maximum processing time of any job. Then $OPT \geq \max\{p_{\text{max}}, \frac{1}{m} \sum_{i=1}^{n} p_i\}$ because $OPT$ at least needs to cost $p_{\text{max}}$ time and if each machine takes less than average time then, you can’t finish all the jobs. Now consider a greedy algorithm that loops over $1$ to $n$ and assign job $i$ to the machine that has the least load so far. In the worst case scenario, $p_{\text{max}} = p_n$, and $p_1, \ldots, p_{n-1}$ are perfectly distributed over the $m$ machines after $n - 1$ jobs are assigned. The load after $n - 1$ jobs is then $\frac{1}{m} (p_{\text{max}} + \sum_{i=1}^{n} p_i)$, and then the load after adding the $n$th job the load will be:

$$\frac{1}{m} \left( -p_{\text{max}} + \sum_{i=1}^{n} p_i \right) + p_{\text{max}} = \frac{1}{m} \sum_{i=1}^{n} p_i + \frac{m-1}{m} p_{\text{max}} \leq \left( 1 + \frac{m-1}{m} \right) OPT = \left( 2 - \frac{1}{m} \right) OPT$$

**Problem 21. Reduce the Flow (Berkeley CS 170)**

Given a directed graph $G = (V, E)$, where each edge has capacity 1. You are also given vertices $s, t \in V$, and a number $k \in \mathbb{N}$.

(a) Is it possible to compute the maximum $s - t$ flow in $O(|V||E|)$? How or why not?

(b) Give an algorithm that returns $k$ edges that, when deleted, reduce the maximum $s - t$ flow by as much as possible. Hint: the max flow is the same as what?

**Solution**

(a) Yes! We can just run Ford-Fulkerson. The runtime is $O(|E| f^*)$ where $f^*$ is the value of the max flow, but since the edge weights are all 1, we have $f^* \leq |V| - 1$, giving the desired runtime.

(b) First we run Ford-Fulkerson to compute the max flow $f$ and construct the residual graph $G_f$. We define $S = \{ v \in V : v \text{ is reachable from } s \text{ in } G_f \}$ and then $T = \{(v, w) \in E : v \in S, w \notin S \}$. In other words, we find the set $T$ of edges that cross the cut $(S, V - S)$. Then we return any $k$ edges in $T$. Because of Ford-Fulkerson, $(S, V - S)$ is a min cut equal in value to the max flow. By removing any $k$ edges from $T$, we reduce the capacity of the cut by $k$ because each edge is unit weight, and the cut remains minimal in the modified graph because we’ve only made it smaller. Thus, the max flow also decreases by $k$, and because we can only delete $k$ edges, this is the most by which we can decrease the max flow so the algorithm is optimal.