1. To reduce the problem of multiplying $n \times n$ matrices $A$ and $B$ to the problem of multiplying two lower triangular matrices $C$ and $D$, simply create a $3n \times 3n$ matrix:

$$C = D = \begin{bmatrix} 0 & 0 & 0 \\ B & 0 & 0 \\ 0 & A & 0 \end{bmatrix}$$

then we have,

$$CD = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ AB & 0 & 0 \end{bmatrix}$$

The matrix $AB$ can easily be obtained by isolating the lower left corner of the product matrix.

2. To reduce the problem of multiplying $n \times n$ matrices to the problem of inverting a single matrix, construct a $3n \times 3n$ matrix $C$ of the following form:

$$\begin{bmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix}$$

Where $I$ is the $n \times n$ identity matrix. By inverting $C$ we get:

$$\begin{bmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{bmatrix}^{-1} = \begin{bmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{bmatrix}$$

$AB$ can be easily obtained by extracting the upper right corner of $C^{-1}$.

3. Show that PolyMult $\leq$ PolySquare.

Using the identity

$$(A + B)^2 = A^2 + B^2 + 2AB$$

we have

$$AB = [(A + B)^2 - A^2 - B^2]/2$$

Procedure PolyMult

{ 
Read $A$ and $B$
$x = \text{PolySquare}(A + B)$
$y = \text{PolySquare}(A)$
$z = \text{PolySquare}(B)$
$AB = (x - y - z)/2$
}

The time for procedure Polymult is the time for procedure PolySquare plus $O(n)$.
4. To reduce the problem of sorting \( n \) numbers to the minimum Steiner tree problem, take the numbers \( x_1 \ldots x_n \) to sort and create the points \((x_1,0)\ldots(x_n,0)\) in the plane. The minimum Steiner tree of these points is the tree that contains \((x_1,0)\ldots(x_n,0)\) in order.

5. [This solution was adapted from Zhong Zhuang and Wen Gao’s homework writeup.] Refer the three problems as Problem A, B and C, respectively.

**Theorem: A \leq B**

**Proof:** we can directly feed Problem B solver with the circuit layout of an instance of Problem A, and take the result from Problem B solver as the result of A. Since A restricts the fan-in number of AND/OR gates to be 2 but B does not have such restriction, any circuit layout of A is legit and solvable to the solver of B. This is a linear reduction.

**Theorem: B \leq A**

**Proof:** we examine the circuit layout of an instance of Problem B, and replace any \( k \)-ary \((k > 2)\) AND/OR gate with \( k - 1 \) binary AND/OR gates. For example, if an AND gate \( AND_{\text{multi}} \) has \( k \) inputs \( a_1, a_2, \ldots, a_k \), we connect \( a_1 \) and \( a_2 \) to the inputs of a new binary AND gate \( AND_2 \), and connect the output of \( AND_2 \) and \( a_3 \) to the inputs of a new binary AND gate \( AND_3 \), and repeat until \( AND_k \). The output of \( AND_k \) is equivalent to \( AND_{\text{multi}} \) and they will have the same fan-out layout. Assume the original circuit layout has \( m \) gates and the maximum fan-in number of a gate is \( n \), the translation will take \( O(mn) \) time and the input of the new layout will have \( O(mn) \) gates. By the above translation the new circuit layout is legit and solvable to the solver of A, and we will take the result from Problem A solver as the result of Problem B.

**Theorem: A \leq C**

**Proof:** we examine the circuit layout of an instance of Problem A, and replace each gate with an operator, replace the inputs of each gate with operands. If one of the inputs is an output of another gate, the output of the preceding gate will be treated as an operand. The only trick is we need to have a variable with the same output as the proceeding gate. To do this, we add the logical equivalent of “if and only if” of the previous gate. Logically, we have \( z \iff f(x,y) \) is translated to \((z \lor \neg f(x,y)) \land (\neg z \lor f(x,y))\). For example, if the previous gate was an \( AND \) with inputs \( x \) and \( y \), we can set \( z \) to be the output of that gate using the logical statement \((z \lor \neg(x \land y)) \land (\neg z \lor (x \land y))\). We can use parenthesis to enforce the priority of operations. Assume the original layout has \( m \) gates, the translation takes \( O(m) \) and the length of final expression will be \( O(m) \). We will take the result from Problem C solver as the result of Problem A.
Theorem: $C \leq A$

Proof: this is the opposite of previous reduction. Examine the expression of an instance of Problem C, and replace each operator with a gate, each operand with an input. If one of the operand is an expression, the input will be connected to the output of the gates that compute such expression. Assume the original expression has $m$ operands, the translation takes $O(m)$ and the number of gates will be $O(m)$. The above translation assumes that any terminal input can be fed into arbitrary number of gates; otherwise if any variable appears in more than one operation, we could connect corresponding input serially to two NOT gates and connect the outputs of the second NOT gate to the desired inputs. This will still give $O(m)$ in translation time and number of gates. We will take the result from Problem A solver as the result of Problem B.

Conclusion: any of the three problems is polynomial time reducible to any of the other two. Therefore if anyone has a polynomial time solver, we can solve the other two in polynomial time.

6. No solution given

7. No solution given

8. To reduce the problem of finding a Hamiltonian cycle in a graph $G$ to the problem of determining whether one exists, create graph $G'$ by removing an arbitrary edge $(x, y)$. Call the HAM-decision algorithm on both $G$ and $G'$. If there is a Hamiltonian cycle in both, recurse on $G'$. If there is a Hamiltonian cycle in $G$ but not $G'$, then we know $(x, y)$ is in the Hamiltonian cycle. Recurse on $G$, taking care not to select edge $(x, y)$ again.

9. To reduce the clique optimization problem to the clique decision problem, we need to show that problem of finding a clique of maximum size is reducible to determine the size of the maximum clique. Algorithm for finding clique: Pick a vertex $v$. If the maximum clique size in $G - v$ is equal to the maximum clique size in $G$, we can recurse on $G - v$. Otherwise, add $v$ to final clique, delete $v$ and vertices in $G$ not adjacent to $v$, and recurse.

10. Given a polynomial time algorithm $VC(G, k)$ which returns true iff graph $G$ contains a vertex cover of size $k$, we can create an algorithm to find the maximum vertex cover by first calling $VC(G, i)$ for $i$ from $|V| - 1$ to $1$. Call $k$ the smallest $i$ such that $VC(G, i)$ returned true. We now consider each vertex, $v$, and call $VC(G - v, k - 1)$. If $VC(G - v, k - 1)$ returns true, then add $v$ to the cover and set $G = G - v$ and $k = k - 1$, otherwise, leave $G$ and $k$ alone. To see why this is correct observe that we never add
more than \(k\) vertices to the cover, we only remove edges from \(G\) if they are covered, and because \(VC(G - v, k - 1)\) is true, all remaining edges can be covered with \(k - 1\) vertices still in \(G - v\). It is not too hard to see that we cannot go through the entire graph more than once before \(VC(G - v, k - 1)\) returns true, thus we iterate at most \(|V|^2\) times, making our reduction polynomial if \(VC(G, k)\) is polynomial.

11. Not assigned

12. [This solution was adapted from Miguel Dickson and Eric Wiegandt’s homework writeup.]

(a) We will prove that determining if graph \(G\) contains a clique of size \(3n/4\), where \(n\) is the number of vertices in \(G\), is NP-hard. We will show that \(\text{clique} \leq 3n/4\) clique.

The input to the clique problem is graph \(H\) and integer \(k\). We need to transform these inputs into the input for \(3n/4\) clique, a single graph \(G\). There are three cases, in which \(p\) will represent the number of vertices in \(H\):

1) \(k = 3p/4\). This case is trivial — \(H\) requires no transformation, and the output to the \(3n/4\) problem is the same as the output to the clique problem.

2) \(k > 3p/4\). To transform \(H\) into a suitable input for the \(3n/4\) clique problem, add vertices to \(H\) until \(3p/4 = k\). These vertices have no edges connected to them, so they cannot be part of a clique. Thus, there will be a clique of size \(k = 3n/4\) in this new graph if and only if there was a clique of size \(k\) in \(H\).

3) \(k < 3p/4\). This is the most complicated case. We cannot take vertices out of \(H\) without potentially changing an existing clique, so we must add vertices and edges to \(H\). We will add \(x\) vertices to \(H\), connecting each added vertex to every vertex already in \(H\). These vertices will form a clique of size \(x\), and can also be part of any existing clique (as every vertex added is mutually adjacent to all other vertices added and all vertices in the existing clique). In our new graph, there will be a clique of size \(x + k\) if and only if there was a clique of size \(k\) in \(H\). We will use the following equation to define \(x\):

\[
\frac{3(p + x)}{4} = k + x,
\]

which simplifies to \(x = 3p - 4k\).

program Transform(Graph \(H\), int \(k\)) {
    int \(p\) = # of vertices in \(H\);
    if \((k = 3p/4)\) {
        return \(H\);
    }
}
} else if \((k > 3p/4)\) {
  int \(x = \frac{4}{3}k - p\);
  \(G = H\) with \(x\) vertices and no edges added;
  return \(G\);
} else if \((k < 3p/4)\) {
  int \(x = 3p - 4k\);
  \(G = H\) with \(x\) vertices added and edges connecting each new vertex to all other vertices;
  return \(G\);
}

The code for the clique problem is now simple:

```c
program CLIQUE(Graph \(H\), int \(k\)) {
  Graph \(G\) = Transform(\(H\), \(k\));
  return \(3n/4\_Clique(G)\);
}
```

(b) The proof that the problem of determining if graph \(G\) with \(n\) vertices has an independent set of size \(3n/4\) is very similar to the proof in part (a). We will show that independent set \(\leq 3n/4\) independent set.

Graph \(H\) and integer \(k\) are the inputs to the independent set problem. Let \(p\) be the number of vertices in \(H\). There are three cases to transforming \(H\) and \(k\) into an input for the \(3n/4\) independent set problem:

1) \(k = 3p/4\). No transformation is needed; the output to the \(3n/p\) independent set problem with input \(H\) is the same as the output to the independent set problem with input \(H\) and \(k\).

2) \(k > 3p/4\). We want to add vertices to \(H\) until \(k = 3p/4\). We do not want these vertices to be part of any independent set. We need to add \(\frac{4}{3}k - p\) vertices to \(H\), along with edges connecting each of the new vertices to every other vertex. This will ensure that no new vertex can be part of an independent set, and this new graph will have an independent set of size \(k\) if and only if \(H\) has an independent set of size \(k\).

3) \(k < 3p/4\). We need to add vertices to \(H\) so that the \(3n/4\) independent set problem will output true if and only if \(H\) has an independent set of size \(k\). Let \(x = 3p - 4k\). Add \(x\) vertices to \(H\) and no edges. In this new graph, there will be an independent set of size \(x\). These new vertices can be part of any independent set, as they are not mutually
adjacent to any other vertex. Thus, the new graph will have an independent set of size \( x + k \) if and only if \( H \) has an independent set of size \( k \). The \( 3n/4 \) clique problem will output if the new graph has an independent set of size \( x + k \), as this problem will output whether or not the graph has an independent set of size \( 3n/4 \), as \( \frac{3(p + x)}{4} = k + x \).

program Transform(Graph \( H \), int \( k \)) {
    int \( p \) = \# of vertices in \( H \);
    if (\( k = 3p/4 \)) {
        return \( H \);
    } else if (\( k > 3p/4 \)) {
        int \( x = \frac{3}{4}k - p \);
        \( G = H \) with \( x \) vertices added and edges connecting each new vertex to all other vertices;
        return \( G \);
    } else if (\( k < 3p/4 \)) {
        int \( x = 3p - 4k \);
        \( G = H \) with \( x \) vertices and no edges added
        return \( G \);
    }
}

program IndependentSet(Graph \( H \), int \( k \)) {
    Graph \( G = Transform(H, k) \);  
    return \( 3n/4 \) IndependentSet(\( G \));
}

(c) If \( k > (num(\text{vertices}(G))/2) \) then \( G \) cannot contain a clique of size \( k \) AND an independent set of size \( k \) (since by the pigeonhole principle, some of the required vertices to be in the clique have to be included in the independent set - which proves they cannot be in the clique, and vice versa).

For \( k \leq (num(\text{vertices}(G))/2) \), we show that this problem is NP-hard. To do this, we will show that clique \( \leq \) clique and independent set.

The input to the clique problem is a graph \( (H) \) and an integer \( (j) \). To transform \( H \) into the input for the AND problem, simply add \( j \) vertices and no edges to the graph, which we will now call \( G \). \( G \) will always have an independent set of size \( j \), as each of the \( j \) vertices we added have no edges and can be part of the set. \( G \) will have a clique if and only if \( H \) has a clique, as we did not add or remove edges. By making one of the conjuncts true, calling the AND code will return
the truth value of the other conjunct: whether $H$ has a clique of size $j$.

program CLIQUE(Graph $H$, int $j$) {
    $G = \text{Transform}(G)$;
    return AND($G$, $j$);
}

(d) Determining whether $G$ contains a clique of size $K$ OR an independent set of size $k$ is NP-hard; we demonstrate this by reduction from the independent set problem. That is, we show that independent set $\leq$ (independent set size $k$ or clique size $k$ [EITHER]). Take the input $G$ and run EITHER on it. If EITHER returns 0, return 0 for the independent set problem. If EITHER returns 1, then construct a new graph $H$ by adding totally disconnected vertices to the graph of $G$ equal to the number of vertices in $G$ (call that number $n$). Clearly, $H$ cannot have a clique of size $k+n$ since it cannot have a clique size greater than $n$ since even if $k = n$, none of the new points added to $H$ can be part of a clique. So run EITHER$(k+n)$ on graph $H$. If EITHER returns 1, then the original graph $G$ had to have an independent set of size $k$; since EITHER’s 1 result cannot be from a $n+k$ size clique it has to be from a $n+k$ size independent set of which $n$ points are the newly added disconnected points and the other $k$ are an independent set in graph $G$. If EITHER$(k+n)$ returns 0, then the original graph did NOT have an independent set of size $k$, and thus we have shown that independent set $\leq$ EITHER.

(e) There is a polynomial time algorithm for determining if graph $G$ with $n$ vertices contains a clique of size $3n/4$ and an independent set of size $3n/4$. If $n \geq 4$, it is impossible for $G$ to have both a clique of size $3n/4$ and an independent set of size $3n/4$. Assume that there is a graph $G$ with $n \geq 4$ and a clique of size $3n/4$ and an independent set of size $3n/4$ in $G$. There must be at least two vertices that are both in the clique and the independent set. These vertices must be connected, as they are in the clique. These vertices cannot be connected, as they are in the independent set. This is a contradiction, and our assumption that there is such a graph is false.

There are 3 cases left to consider: $n = 1$, $n = 2$, and $n = 3$:

$n = 1$. $3n/4 = 0$ in this case. Trivially, there is a clique of size 0 and an independent set of size 0. Always return true.

$n = 2$. $3n/4 = 1$. Trivially, there is a clique of size 1 and an independent set of size 1. Always return true.
\( n = 3 \), \( 3n/4 = 2 \). There are 3 possible edges in a graph of size 3. If there are 0 edges, there is no clique of size 2. With 1 edge, there will be a clique of size 2 (the two connected vertices) and an independent set of size of size 2 (one of the edge’s endpoint and the vertex not connected to the edge). With 2 edges, there is a clique of size 2 (any 2 connected vertices) and an independent set of size 2 (the two vertices without an edge connecting them). With 3 edges, there is not independent set of size 2, as all vertices are connected.

program \( 3n/4 \& \text{AND}(\text{Graph } G) \) {
    \( n = \# \text{ of vertices in } G \);
    \( \) if (\( n = 1 \)) {
        return true;
    } else if (\( n = 2 \)) {
        return true;
    } else if (\( n = 3 \)) {
        \( e = \# \text{ edges in } G \);
        \( \) if (\( e = 0 \) or \( e = 3 \)) {
            return false;
        } else if (\( e = 1 \) or \( e = 2 \)) {
            return true;
        }
    }
    \( \) else {
        return false;
    }
}

(f) This problem is NP-hard because we can reduce independent set to this problem, e.g., show that independent set \( \leq \) (independent set size \( 3/4k \) OR clique size \( 3/4k \)), again call this EITHER. The proof is essentially a combination of problems b) and d) above. First, apply the procedure in b). Then apply the procedure in d) with one slight modification: we add \( n \) more vertices total (where \( n \) is the number of vertices of the graph) but instead of making them all disconnected, only make \( 3/4 \) of them disconnected and make the other \( 1/4 \) of them connect completely with each other such that they cannot be part of any independent set. Then apply the EITHER algorithm. For analogous reasons to those described in d) this procedure returns 1 if and only if the new graph has an independent set of size \( (3/4n + 3/4n) \) which implies that the original graph has an independent set of size \( (3/4n) \), and thus we have shown that independent set \( \leq \) (independent set size \( 3/4k \) OR clique size \( 3/4k \)).

13. To reduce CNF-SAT to the linear inequality problem we create a set of inequalities \( S \) as follows: for each variable, \( x_i \), in boolean formula \( F \), we
create a integer variable, $x_i$, and constrain it to be 0 or 1 by adding the inequalities, $x_i \geq 0$ and $x_i \leq 1$. For each disjunction we create a sum: if the literal is $x_i$, we simply add $x_i$, and if the literal is $\bar{x}_i$, we add $(1 - x_i)$. We then require the sum to be at least 1. Call the set of inequalities, $S$. If we map true to 1 and false to 0, it’s not too hard to see that a satisfying assignment to $F$ gives a solution $S$, and a solution to $S$ gives a satisfying assignment to $F$.

14. To reduce the 3-coloring problem to the 4-coloring problem, create a new vertex $v$ that is connected to every other vertex in the original graph. Since $v$ is connected to every vertex, it must have a different color than the others. This new graph can be colored with 4 colors if and only if the original graph can be colored with 3 colors.

15. To reduce the 3-coloring decision problem to the SAT-CNF: Assume the colors are red, blue, and green. For each vertex $x$ in $G$, create the following variables: $x_{\text{red}}, x_{\text{blue}},$ and $x_{\text{green}}$. For each vertex $v \in G$, exactly one of $v_{\text{red}}, v_{\text{blue}},$ and $v_{\text{green}}$ must be true, and for each vertex $u$ adjacent to $v$, the $u$-variable corresponding to $v$’s color must be false (that is, $\bar{u}_{\text{color}} \lor \bar{v}_{\text{color}}$ must be true). The construction of the entire clause is straightforward; simply “and” together all the vertex clauses. If there is an assignment to these variables that makes the entire clause true, the graph can be colored with 3 colors.

16. To reduce the vertex-cover problem to the dominating set problem, create a new graph $G'$ by replacing each edge $e = (x, y)$ in $G$ by a new vertex $v_e$ and edges $(x, v_e), (y, v_e)$ and $(y, x)$. The vertex $v_e$ essentially forces the edge $e$ to be covered. Call the procedure for dominating set to get a minimum cardinality dominating set $D$. If a $v_e \in D$ then replace $v_e$ by $x$ (or $y$ doesn’t matter). Call this new set $D'$. Note that $D'$ is still dominating in $G'$. Then $D$ is a minimum cardinality vertex cover in $G$. (Note this reduction only handles the case when $G$ is a connected graph and has at least one edge. If $G$ is not connected, run this reduction on each non-trivial connected component.)

17. **Non-many-to-one reduction**: To reduce the Hamiltonian Cycle problem to the Fixed Hamiltonian path problem, check if there is a Hamiltonian path between $x$ and $y$ for each edge $e = (x, y) \in G$. Note that this is sufficient to answer the problem as stated.

**Many-to-one reduction**: Create graph $G'$ as follows: add a new vertex, $v'$, to the input graph $G$, and then pick any vertex $v$, in $G$, and connect $v'$ to all of the neighbors of $v$. To solve the Hamiltonian Cycle decision problem, $HC(G)$, return the value of the Fixed Hamiltonian path problem on $G'$ with start vertex $v$ and end vertex $v'$, $FHP(G', v, v')$. 
If $FHP(G', v, v')$ is true, then by definition, if we added an edge between $v$ and $v'$, we’d now have a hamiltonian cycle in $G'$. However, as $v$ and $v'$ have an identical set of neighbors, we know that $v'$’s neighbor in the path, call it $u$, is $v$’s neighbor in $G$. Thus the section of the path that is in $G$, would start at $v$ and end at $u$, because $v$ and $u$ are neighbors, we can complete the cycle.

If $HC(G)$ is true, then to construct the path for $FHP(G', v, v')$, for any $v$ in $G$, remove the edge (from the cycle) between $v$ and one of $v$’s neighbors in the cycle (there are two), call it $u$, then add the edge from $u$ to $v'$ to complete the path from $v$ to $v'$. The edge from $u$ to $v'$ must exist in $G'$ by the construction of $G'$.

18. [This solution was adapted from Brian Wongchaowart’s homework writeup.]
To solve the vertex cover problem for an undirected graph $G$ and an integer $m$, construct a new graph $H$ to be used as input to the problem. Begin by adding a new vertex $r$ to $H$. Then for each vertex $v_k$ in $G$, add a vertex to $H$ labeled $v_k$ and an edge between this vertex and $r$. For each edge $(v_i, v_j)$ in $G$, add a vertex to $H$ labeled $e_{ij}$ and add an edge between this vertex and the vertices in $H$ labeled $v_i$ and $v_j$.

Add vertex $r$ and all of the vertices $e_{ij}$ corresponding to edges of $G$ to the set $R$. Let $e$ be the number of edges in $G$. Use $H$, $R$, and $m + e + 1$ as the input to the problem. A set $U$ all of the vertices in $R$ and at most $m + e + 1$ vertices is a subgraph of $H$ if and only if $G$ has a vertex cover of size $m$:

$\left(\Leftarrow\right)$ If $G$ has a vertex cover $C$ of size $m$, such a $U$ can be found in $H$ by taking $r$ as the root, adding the $m$ vertices labeled with the vertices in $C$, and finally adding each of the $e$ vertices corresponding to an edge in $G$. There is a path from the vertex $r$ to every vertex $e_{ij}$ by using the edge connecting it to the vertex from $C$ (such an edge must exist by the definition of a vertex cover). This set has $m + e + 1$ vertices.

$\left(\Rightarrow\right)$ If, on the other hand, $H$ contains a set $U$ that includes paths between all of the vertices in $R$ and has at most $m + e + 1$ vertices, the vertices in $G$ that correspond to a vertex in $U$ form a vertex cover for $G$ of size at most $m$. To see why, note that $r$ is not connected directly to any vertex $e_{ij}$ in $H$, but by definition of $R$, these $e$ vertices must all be in $U$. So $U$ must include some number of the vertices in $H$ such that there is an edge from one of these vertices to each vertex $e_{ij}$. This corresponds exactly to choosing a set of vertices in $G$ so that each edge is incident to a vertex in the set, i.e., choosing a vertex cover. The number of these vertices, the size of this vertex cover, cannot be greater than $m$ because, $r$, as well as all $e$ vertices, each corresponding to an edge in $G$, must be in $U$. This reduction can clearly be carried out in polynomial time, as one vertex is added to $H$ for each vertex and each edge in $G$. 

10
19. [This solution was adapted from Brian Wongchaowart’s homework writeup.]

Suppose that $F$ is a Boolean formula in CNF with $k$ clauses and exactly three literals per clause. It is possible to construct a directed graph $G$ with pairs of vertices $(s_1, t_1), \ldots, (s_k, t_k)$ such that the disjoint paths problem with this input can be solved if and only if $F$ is satisfiable. The construction is as follows (refer to diagrams below): For each clause $C_i$ in $F$, there is a pair of vertices $(s_i, t_i)$ in $G$. There is an edge from each starting vertex $s_i$ to three literal gadgets corresponding to the three literals of clause $C_i$ and a unique path (to be described later) from each of these literal gadgets to the ending vertex $t_i$. The paths will be specified such that if there is a set of disjoints paths from each $s_i$ to each $t_i$, then each clause must have at least one true literal because the only paths from $s_i$ to $t_i$ are the ones through the three literal gadgets. Hence following the paths will indicate which literals are assigned a value of true.

```
"Clause Gadget"
(here for clause 1)

s1
/ | \  
/ | \\  
/ | \  
\ | /  
\ | /  
\ | /  
L1 L2 L3 <-- is a "Literal Gadget" of the form: --> v1--> v2--> v3-->

insert paths here to disable incompatible literals in other clauses

\ | /  
\ | /  
\ | /  
t1
```

The basic idea is that a path starts at each clause’s starting vertex $s_i$ and picks one of its three literal gadgets to be true, then continues on to end at the clause’s ending vertex $t_i$. Before terminating at the ending vertex, however, a path that has passed through a gadget for the literal $x$ should “disable” or pass through, the gadgets for the literal $\overline{x}$ in every other clause so that they cannot be used, and similarly, a gadget for the literal $\overline{x}$ should disable the gadgets for the literal $x$ in every other clause. So if there are $k$ clauses, it is possible for a literal to be disabled $k - 1$ times. These considerations suggest that a literal gadget (see figure on the right) should be a chain of $k$ vertices, $v_1, \ldots, v_k$, connected by edges $(v_1, v_2), (v_2, v_3), \ldots, (v_k, v_1)$, such that $v_i$ is the “disable switch” used by clause $C_i$. 

11
In order for clause $C_i$’s path from $s_i$ to $t_i$ to pass through one of its literal gadgets, this path must pass through all of the vertices in the literal gadget $v_1, \ldots, v_k$. Another clause $C_j$ can “activate” its disable switch in this literal gadget of clause $C_i$ by sending its path (from $s_j$ to $t_j$) through vertex $v_j$ in the gadget. In the disjoint paths problem, if $C_j$’s path passes through $v_j$, it becomes impossible for $C_i$’s path to pass through this gadget. In this way a solution to the disjoint paths problem for $G$ and $(s_1, t_1), \ldots, (s_k, t_k)$ cannot make use of two opposing literals $x$ and $\bar{x}$ in different clauses (i.e. cannot assign $x$ to be both true and false).

Thus it is easy to see that a solution to the disjoint paths problem can be turned into a satisfying assignment for $F$ and that a satisfying assignment for $F$ allows one to pick disjoint paths in $G$. This reduction can clearly be carried out in polynomial time, as $G$ contains two vertices for each clause in $F$ and a gadget for each literal, with the size of the literal gadgets being proportional to the number of clauses.

20. We show that the triangle problem is hard by reduction from 3SAT. It will be sufficient to show how to construct in polynomial time an instance of the triangle problem that has a feasible solution iff a particular 3SAT formula $F$ is satisfiable.

We first show that we can assume without loss of generality that for each variable $v$ in $F$, the literal $v$ and the literal $\bar{v}$ appear an equal number of clauses. If $v$ appears more frequently than $\bar{v}$, then add clauses of the form $(\bar{v} \lor a \lor \bar{a})$, where $a$ is a new variable until the number of occurrences of $v$ and $\bar{v}$ are equal. Similarly add clauses of the form ($v \lor a \lor \bar{a}$) if there are more occurrences of $\bar{v}$ than $v$. This does not affect the satisfiability, and equals out the number of positive and negative occurrences of the variables.

The construction of the triangle instance: Then for each variable $v$ in $F$ that appears in $2k$ clauses, there will be $k$ elements $x_{v,0}, \ldots, x_{v,k-1}$ in $X$ and $k$ elements $y_{v,0}, \ldots, y_{v,k-1}$ in $Y$. For each clause $C$ in $F$, there will be an element $z_C$ in $Z$. An additional $2c$ garbage elements are added to $Z$, where $c$ is number of clauses in $F$. Let $v$ be a variable that occurs $k$ times in $F$. Number both the $k$ positive occurrences and the $k$ negative occurrences arbitrarily. For the $j$th occurrence of $v$ we add a triangle of the form $(x_{v,j}, y_{v,j}, C)$, where $C$ is the clause in which contains the $j$th occurrence of $v$. We also add $2c$ garbage triangles of the form $(x_{v,j}, y_{v,j}, g)$, one for each garbage element $g$ in $Z$. Call these the $v$-true triangles. For the $k$th occurrence of the literal $\bar{v}$ we add a triangle of the form $(y_{v,j}, x_{v,j+1 \text{mod} k}, C)$, where $C$ is the clause in which contains the $j$th occurrence of $\bar{v}$. We also add $2c$ garbage triangles of the form
(y_{v,j}, x_{v,j+1 \mod k}, g)$, one for each garbage element $g$ in $Z$. Call these the v-false triangles.

**Correctness:** Let us first assume $F$ is satisfiable, and then show that the resulting instance of the triangle problem is feasible. For each clause, pick out one true literal, and call this the satisfying literal. If variable $v$ is true in the assignment, then pick the triangle $(x_{v,j}, y_{v,j}, C)$ for each clause $C$ in which $v$ is the satisfying literal. Cover the remaining elements of the form $x_{v,0}, \ldots, x_{v,k-1}$ in $X$ and $y_{v,0}, \ldots, y_{v,k-1}$ in $Y$ with garbage v-true triangles. If the variable $v$ is false in the assignment, then pick the triangle $(y_{v,j}, x_{v,j+1 \mod k}, C)$, for each clause $C$ in which $\overline{v}$ is the satisfying literal. Cover the remaining vertices of the form $x_{v,0}, \ldots, x_{v,k-1}$ in $X$ and $y_{v,0}, \ldots, y_{v,k-1}$ in $Y$ with garbage v-false triangles. The non-garbage elements in $Z$ are covered because the assignment is satisfying.

Now assume that the triangle instance has a feasible solution. First note that because the v-true and v-false triangles are arranged alternatively in a cycle around the vertices $x_{v,0}, \ldots, x_{v,k-1}$ in $X$ and $y_{v,0}, \ldots, y_{v,k-1}$ in $Y$, and no other triangles cover these vertices, its not possible that both a v-true and a v-false triangle are selected. This selection then corresponds to an assignment to the variables in $F$ in the natural way. This assignment is satisfying because the non-garbage vertices in $Z$ are covered.

21. No solution given

22. We show that the Fox, Goose, and Bag of Beans problem, $FGBoB(G, k)$, is NP-hard by reduction from the Vertex Cover problem, $VC(G, k)$. To solve $VC(G, k)$, create a $G'$ that is identical to $G$ with two additional vertices, $v_1$ and $v_2$, that have an edge between them and no other edges. Return $FGBoB(G', k + 1)$.

If $VC(G, k)$ is true, then create a solution to $FGBoB(G', k + 1)$ by putting the vertex cover of $G$ in the boat, then put $v_1$ in the boat. By the definition of a vertex cover and $G'$, an independent set remains on the left shore. Take the boat across and drop $v_1$ on the right side, now all vertices not in the vertex cover of $G$ can be transported, one at a time, from the left shore to the right. Lastly, $v_2$ can be transported and now all vertices are on the right shore.

If $FGBoB(G', k + 1)$ is true, then consider the first trip across. It must be that at least one of $v_1$ or $v_2$ are in the boat, thus there can be at most $k$ vertices from $G$ in the boat. Of the vertices that remain on the left shore from $G$, they must be an independent set, thus the at most $k$ vertices in the boat from $G$ must be a vertex cover on $G$.

23. [This solution was adapted from Jesse Davis, Andrew Menzes, and You Zhou’s homework writeup.]
Reduction of 3SAT to this problem (which we will call the Set Splitting Problem): Suppose we have an algorithm $F$ that decides whether a given instance of the Set Splitting Problem has a solution. Given a Boolean formula $B$ as input to 3SAT, create an initially empty collection $C$ of sets and set $S$ of individual elements. For each distinct variable $v_i$ in $B$, add two elements to $S$: $v_i$ and $¬v_i$. Add a set $V_i$ to $C$ that consists of only $v_i$ and $¬v_i$. Also, add a single element $d$ to $S$ representing “false”. For each clause $c_i$ in $B$, add a set $C_i$ to $C$ containing $d$ and the literals that appear in that clause. (For instance, if a clause in $B$ is $(v_1 \lor v_2 \lor ¬v_3)$, then add the set $\{v_1, v_2, ¬v_3, d\}$ to $C$.) Finally, run $F$ on $(S, C)$ and return the same result.

Explanation: If the instance of Set Splitting has a solution, then element $d$ is assigned to either $S_1$ or $S_2$. Suppose WLOG that it is assigned to $S_1$. For each variable $v_i$, one of the literals containing it (either the positive or negative) must be assigned to $S_1$ and the other to $S_2$, because the set $V_i$ contains only those two elements. Since every clause set $C_i$ contains $d$, which is in $S_1$, each clause needs to contain at least one element in $S_2$. This is equivalent to the requirement that each clause in a 3CNF formula must contain at least one true literal, with the literals assigned to $S_2$ corresponding to true literals in a truth assignment and the literals assigned to $S_1$ corresponding to false literals.

The reduction takes polynomial time (creating a 2-element set per variable and a 3-element set per clause) so, because 3SAT is NP-hard, so is Set Splitting.