CS 505 Spring 2025 — Homework 2 (Sample Solutions)

Alex Block

1 Diagonalization (25 Points)

1.1 Part 1 (10 Points)

Prove that if $\mathbf{NEXP} \neq \mathbf{EXP}$ then $\mathbf{NP} \neq \mathbf{P}$.

Hint: You can use Problem 3 Part 2 from Homework 1 to help. We recall it here. Define a function $\mathrm{pad}\colon \{0,1\}^* \times \mathbb{N} \to \{0,1\}^* \times \{\#\}^*$ as follows. For $x \in \{0,1\}^*$ and $k \in \mathbb{N}$, $\mathrm{pad}(x,k) = (x, \#^j)$, where $j = \max\{0, (k - |x|)\}$ and $\#^j$ denotes writing the symbol $\#$ j times (e.g., $\#^3 = \#\#\#$).

Let $A \subseteq \{0,1\}^*$ be any language and let $T\colon \mathbb{N} \to \mathbb{N}$ be a function. Let $\mathbf{pad}(A,T)$ denote the following language:

$$\mathbf{pad}(A,T) = \{\mathrm{pad}(x, T(|x|)) \mid x \in A\}.$$

If $A \in \mathbf{DTIME}(n^6)$ then $\mathbf{pad}(A, n^2) \in \mathbf{DTIME}(n^3)$.

Proof of Problem 1 Part 1. We prove the contrapositive, namely that if $\mathbf{P} = \mathbf{NP}$ then $\mathbf{EXP} = \mathbf{NEXP}$. Note that since $\mathbf{EXP} \subseteq \mathbf{NEXP}$, all we have to show is that $\mathbf{NEXP} \subseteq \mathbf{EXP}$. Let $L \in \mathbf{NEXP}$. Then there exists a non-deterministic Turing machine M_L such that for any x, we have $x \in L$ if and only if $M_L(x) = 1$ and M_L runs in time $\exp(|x|)$. In particular, suppose that M_L runs in time 2^{n^c} for constant $c \geq 1$ and inputs of length n.

Consider the following language $\widehat{L} = \mathbf{pad}(L, 2^{n^c})$. Clearly, we can decide $\widehat{L}$ using an NTM, say $N_{\widehat{L}}$, running in exponential time as follows. Given any input y, $N_{\widehat{L}}$ first checks if $y = (x, \#^j)$ for $j = 2^{|x|^c} - |x|$ and rejects if it is not in this form. Otherwise, $N_{\widehat{L}}$ simulates $M_L(x)$, which takes non-deterministic time $2^{|x|^c}$ and outputs the same answer as M_L.

Now, let $|y| = n$. Then clearly $N_{\widehat{L}}(y)$ runs in polynomial time in n. This implies that $\widehat{L} \in \mathbf{NP}$, which by our assumption implies that $\widehat{L} \in \mathbf{P}$. This implies there also exists a deterministic Turing machine, say $D_{\widehat{L}}$, which decides $\widehat{L}$ in polynomial time. Given $D_{\widehat{L}}$, we construct an exponential time deterministic Turing machine to decide L. Let $\widetilde{M_L}$ be a deterministic Turing machine that operates as follows. On input x, $\widetilde{M_L}(x)$ constructs $y = \mathrm{pad}(x, 2^{|x|^c})$ and simulates $D_{\widehat{L}}(y)$. Clearly this takes exponential time in the length of the input x. Thus, $L \in \mathbf{EXP}$. $\square$

1.2 Part 2 (10 Points)

Define the *unique-SAT* problem as

$$\mathrm{USAT} = \{\phi : \phi \text{ is a Boolean formula with exactly 1 satisfying assignment}\}.$$

Prove that $\mathrm{USAT} \in \mathbf{P}^{\mathrm{SAT}}$.

Proof of Problem 1 Part 2. To prove $\mathrm{USAT} \in \mathbf{P}^{\mathrm{SAT}}$, we give a polynomial time deterministic Turing machine that decides USAT when given an oracle to SAT. Recall that an oracle to SAT takes as input a formula ϕ and outputs 1 if ϕ is satisfiable and 0 if ϕ is unsatisfiable. At a high-level, our machine will decide USAT by actually finding the unique satisfying assignment.

Let D^{SAT} be a deterministic oracle Turing machine with oracle access to SAT. The machine D^{SAT} takes as input a SAT formula ϕ and operates as follows. Assume that $|\phi| = n$ and ϕ has n variables $x_1, \ldots, x_m$.

- $D^{\mathrm{SAT}}(\phi)$:
  - Query $b = \mathrm{SAT}(\phi)$. If $b = 0$, output 0. Else, continue.
  - Let $\varphi = \phi$. For $i \in [m]$:
    - Query $b_0 = \mathrm{SAT}(\varphi_{x_i=0})$ and $b_1 = \mathrm{SAT}(\varphi_{x_i=1})$.
    - If both $b_0 = 1$ and $b_1 = 1$, output 0.
    - Else if $b_0 = 1$ and $b_1 = 0$, set $\varphi = \varphi_{x_i=0}$.
    - Else if $b_0 = 0$ and $b_1 = 1$, set $\varphi = \varphi_{x_i=1}$.
  - Output 1.

We claim that D^{SAT} decides USAT in polynomial time. For any formula ϕ, D first uses the oracle SAT to check if ϕ is satisfiable. If it is not satisfiable (i.e., there is no satisfying assignment, which implies there is no unique satisfying assignment), D outputs 0.

Otherwise, suppose that ϕ is satisfiable. Then D uses SAT to try and build the unique satisfying assignment. It does this by iteratively checking if the formula ϕ with x_i set to 0 and x_i set to 1 is satisfiable. In particular, starting with $i = 1$, D checks if formulas $\phi_{x_1=0}$ and $\phi_{x_1=1}$ are satisfiable. Notably, if ϕ remains satisfiable under *both* assignments of x_1, D outputs 0 (i.e., reject). This is because the satisfying assignment can use either of these, so it is not unique. Otherwise, because ϕ is satisfiable, at least one of $x_1 = 0$ or $x_1 = 1$ is in the satisfying assignment. So D updates ϕ to $\phi_{x_1=b}$, where $b \in \{0,1\}$ is the value of x_1 that leads to a satisfiable formula, then continues with $i = 2$. If D terminates and outputs 1, then clearly ϕ has a unique satisfying assignment.

Now, to see that D operates in polynomial time, note that it takes $O(n)$ time to write ϕ (and its updated formulas with variables fixed) on the oracle tape, then a single computation step to get the answer from the SAT oracle. Moreover, for each i, to construct the sub formulas $\phi_{x_i=0}$ and $\phi_{x_i=1}$ takes $O(n)$ time. So the total time per $i \in [m]$ is $O(n)$. The final runtime then is $O(n \cdot m) = O(n^2)$. □

1.3 Part 3 (5 Points)

Prove that $\mathbf{DSPACE}(n) \neq \mathbf{NP}$.

Hint: Use the Space Hierarchy Theorem and the fact that **NP** *is closed under polynomial-time reductions.*

Proof of Problem 1 Part 3. Suppose by way of contradiction that $\mathbf{DSPACE}(n) = \mathbf{NP}$. Since **NP** is closed under polynomial time reductions, we know that for any two languages A, B, if $A \leq_p B$ and $B \in \mathbf{NP}$, then $A \in \mathbf{NP}$. By assumption, $\mathbf{NP} = \mathbf{DSPACE}(n)$, so this holds for $\mathbf{DSPACE}(n)$ as well: if $A \leq_p B$ and $B \in \mathbf{DSPACE}(n)$, we have $A \in \mathbf{DSPACE}(n)$.

Now, suppose we have $L \in \mathbf{DSPACE}(n^2)$ with decider D_L using $O(n^2)$ space for any input of length n. We construct a new language $\widehat{L} = \mathbf{pad}(L, n^2)$. We show that $\widehat{L} \in \mathbf{DSPACE}(n)$.

Consider $D_{\widehat{L}}$ which takes as input y and operates as follows. First, $D_{\widehat{L}}(y)$ checks if $y = (x, \#^j)$ for $j = |x|^2$. Note this only requires $O(\log(|x|) + \log(j)) = O(\log(|x|))$ bits to check (we just count). If y does not have the correct form, we reject. Otherwise, $D_{\widehat{L}}(y)$ simulates $D_L(x)$. Using the fact that universal simulation only incurs a constant overhead (i.e., to simulate a space S machine, we only need $O(S)$ space), this simulation only uses $O(|x|^2)$ space.

Setting $|y| = n$, we have that $|x| = O(\sqrt{n})$, so simulating $D_L(x)$ uses $O(\sqrt{n}^2) = O(n)$ space. This implies that $\widehat{L} \in \mathbf{DSPACE}(n)$. Now, notice that given any $x \in L$, we can construct $y \in \widehat{L}$ in $O(|x|^2)$ time. Moreover, by the above discussion, we can see that $x \in L$ if and only if $y \in \widehat{L}$. So we have that $L \leq_p \widehat{L}$. This implies that $L \in \mathbf{DSPACE}(n)$. However, this contradicts the space hierarchy theorem, which states that $\mathbf{DSPACE}(f) \subsetneq \mathbf{DSPACE}(g)$ for $f = o(g)$. Clearly, $n = o(n^2)$, so this is not possible. Thus, $\mathbf{DSPACE}(n) \neq \mathbf{NP}$. □

2 Space Complexity (25 points)

2.1 Part 1 (10 Points)

Let UCYCLE $= \{G \colon G$ is an undirected graph that contains a simple cycle.$\}$. Recall that in an undirected graph, a simple cycle is a path of vertices $(v_1, \dots, v_k)$ such that $v_1 = v_k$ and for all $i, j \in \{1, \dots, k-1\}$, we have $v_i \neq v_j$ whenever $i \neq j$ (i.e., all nodes are unique except the first and last). Prove that UCYCLE $\in \mathbf{L}$.

Note: G may not be a connected graph.

Proof of Problem 2 Part 1. The main difficulty in showing that UCYCLE $\in \mathbf{L}$ is that all the "clever" algorithms one might employ in practice use at least linear space. For example, doing a breadth-first search, depth-first search, or Djikstra's algorithm could use at least $O(n)$ space for an n vertex graph G.

So, instead, we give a "dumb" algorithm which decides UCYCLE using only logarithmic space. The key ideas behind the algorithm are as follows.

1. If we find any non-simple cycle in G, there must exist a simple cycle in G. So we do not actually have to check if the cycle we've found is simple or not.

2. To check if a vertex v is part of a cycle, it suffices to find a path leaving v through some edge e and then return to v *through a different edge* $e' \neq e$.

In general, we let D be a deterministic Turing machine that operates as follows. Note for $G = (V, E)$, we assume that $V = [n]$ and note that if $(i, j) \in E$ then $(j, i) \in E$ (since G is undirected). We can also assume that for each $v \in V$, the edges of v are ordered in E. That is, if v is connected to $u_1 < u_2 < \cdots < u_m$, then in E this is listed as $(v, u_1), (v, u_2), \dots, (v, u_m)$.

```
• $D(G)$:
    – For $v \in [n]$ :
        1. If $v$ has at most one edge, continue to next iteration of the loop.
        2. Set $v_0 = v$.
        3. Else let $e = (v, u_1) \in E$ be the smallest edge of $v$; i.e., the first edge in its edge list. Set $e_0 = e$
           and $u = u_1$ and $e_u = (u, v)$.
        4. While $u \neq v_0$:
            (a) If $u$ has a single edge $e^* = (u, u^*)$, set $u = u^*$ and $e_u = (u^*, u)$.
            (b) Else for $e_u = (u, u_i)$ (for some $i$), let $e = (u, u_{i+1})$. Set $e_u = (u_{i+1}, u)$ and $u = u_{i+1}$.
                Where, we wrap $i + 1$ to 1 if $i + 1$ is greater than the number of edges of $u$ (e.g., if $u$ has
                $m$ edges and $i = m$, then $i + 1 = 1$).
        5. If $e_u \neq e_0$, output 1 (accept).
    – Output 0 (reject).
```

First, we argue that D decides UCYCLE. Actually, we argue that if we flip the output of D, we get a decider for $\overline{\text{UCYCLE}}$, which implies the result since $\mathbf{L}$ is closed under complements. In particular, $\overline{\text{UCYCLE}}$ is the set of all undirected graphs that are forests (i.e., one or more disjoint trees).

Suppose first that G is a tree. Then, $D(G)$ is actually performing a depth-first search of G, starting at a vertex v. So $D(G)$ performs a DFS starting at each vertex v. If it returns to v through a different edge, then clearly it has a cycle. Otherwise, $D(G)$ will always return to v through the first edge e_0 it left through. This argument readily extends to when G is a forest (a collection of disjoint trees).

Now suppose that G has a cycle and suppose v is part of this cycle. This leads to two possible outcomes.

1. We return to v through a different edge $e \neq e_0$. Then we are done.

2. We return to v through the same edge e_0.

The second case can happen if v is part of a cycle but is also connected to another vertex u such that if we remove the edge (v, u) from the graph, then u is part of a tree. However, if this case happens, then there is some other vertex v' on the cycle which this does not happen (even if v' is connected to the same type of vertex u). This is because the edges are ordered, and we always leave from the smallest vertex/edge; it cannot be the case that all the smallest edges lead to a tree, otherwise the graph is a forest, which we assumed was not the case.

So D decides UCYCLE. Moreover, clearly D uses $O(\log(n))$ space since we need $O(\log(n))$ bits to store a vertex and an edge, along with a counter, and D only stores a constant number of vertices and edges. $\square$

2.2 Part 2 (15 Points)

Recall the language $2\mathrm{SAT} = \{\phi\colon \phi \text{ is a satisfiable 2CNF formula.}\}$. Show that 2SAT is **NL**-complete.

Proof of Problem 2 Part 2. First, we show that $2\mathrm{SAT} \in \mathbf{NL}$. We do this by showing $\overline{2\mathrm{SAT}} \in \mathbf{NL}$. Then, the result follows since $\mathbf{NL} = \mathbf{coNL}$.

To show that $\overline{2\mathrm{SAT}} \in \mathbf{NL}$, we actually show that $\overline{2\mathrm{SAT}} \leq_L \mathrm{PATH}$, where

$$\mathrm{PATH} = \{(G, s, t)\colon \exists \text{ path from } s \text{ to } t \text{ in directed graph } G\}.$$

Then, by transitivity of logspace reductions, this shows that $\overline{2\mathrm{SAT}} \in \mathbf{NL}$.

Given a formula ϕ, we give a logspace reduction to a directed graph. First, the reduction verifies in ϕ is a valid 2CNF formula; if not, the reduction just outputs an empty graph. Now, assume that $\phi = \phi_1 \wedge \cdots \wedge \phi_n$ and has m variable $x_1, \ldots, x_m$.

We construct a directed graph G_ϕ as follows. First, the graph G_ϕ will have $V = [2m]$, which consists of each variable and its negation $x_i, \overline{x}_i$ for all $i \in [m]$. To make things easy, we label variable x_i as number $2i - 1$ and variable $\overline{x}_i$ as $2i$ in the vertex set. Now, to construct the edges of the graph, we scan the formula ϕ clause by clause. For each $j \in [n]$, suppose that $\phi_j = (a_j \vee b_j)$. Then, we add two edges to the graph: $(\overline{a}_j, b_j)$ and $(\overline{b}_j, a_j)$. For example, if $\phi_1 = (x_3 \vee \overline{x}_5)$, then we add the edges $(\overline{x}_3, \overline{x}_5)$ and (x_5, x_3) to the graph.

Intuitively, we have constructed what is called an implication graph. Recall that the Boolean function $x \Rightarrow y$ is logically equivalent to $(\overline{x} \vee y)$. Moreover, because $(a \vee y) = (y \vee x)$, we also know that $x \Rightarrow y$ is equivalent to $\overline{y} \Rightarrow \overline{x}$. So, we have encoded implications in the graph. The purpose of this is as follows. Every clause ϕ_j encodes some implication. If $\phi_j = (a_j \vee b_j)$, then this encodes $\overline{a}_j \Rightarrow b_j$ and $\overline{b}_j \Rightarrow a_j$. What does this mean in a satisfying assignment? Well, recalling from the implication function $x \Rightarrow y$, if $x = 0$, then the implication is true and evaluates to 1; i.e., y can be whatever value. However, if $x = 1$, then the implication can only be true if $y = 1$ as well.

Now, under this light, a 2CNF formula is a bunch of implications. What does it mean for the 2CNF to be unsatisfiable? It means that for some variable x, the implications give us $x \Rightarrow \overline{x}$ and $\overline{x} \Rightarrow x$. That is, $x = \overline{x}$, which is impossible.

Now, turning back to the graph G_ϕ, we want to capture this implication as paths in the graph G_ϕ. Towards this, for two vertices $a, b \in V(G_\phi)$, let $a \mapsto b$ denote that there is a path from a to b in G_ϕ. We show two facts about the constructed graph G_ϕ.

Claim 1. *For $a, b \in V(G_\phi)$, if $a \mapsto b$, then $a \Rightarrow b$ is encoded in the formula ϕ.*

Proof of Claim 1. By construction of the graph, the edge $(u, v) \in E(G_\phi)$ if $u \Rightarrow v$ in ϕ. Using the fact that if $u \Rightarrow v$ and $v \Rightarrow w$, then $u \Rightarrow w$, if $a \mapsto b$ then by continuously applying the transitivity of $\Rightarrow$, we have that $a \Rightarrow b$ in the formula ϕ. $\square$

Claim 2. *If $a \mapsto b$ in G_ϕ, then $\overline{b} \mapsto \overline{a}$ in G_ϕ.*

Proof of Claim 2. The claim holds by symmetry in the construction of the graph G_ϕ. Namely, if $a \Rightarrow b \in \phi$, then $\overline{b} \Rightarrow a \in \phi$ and we add edges $(\overline{a}, b)$ and $(\overline{b}, a)$ to the graph G_ϕ. $\square$

Now, we need to show two things about the above reduction. First, that ϕ is an unsatisfiable 2CNF if and only if there exists $x \in V(G_\phi)$ such that $x \mapsto \overline{x}$ and $\overline{x} \mapsto x$, and that the reduction can be done in (implicit) logspace. We show the first part of the reduction.

First, suppose that there is a vertex $x \in V(G_\phi)$ such that $x \mapsto \overline{x}$ and $\overline{x} \mapsto x$. Suppose by way of contradiction that there is a satisfying assignment $u \in \{0,1\}^m$ such that $\phi(u) = 1$. Suppose under this assignment, the variable x is assigned 1. Then, by Claim 1, since $x \mapsto \overline{x}$, we know that ϕ encodes the Boolean formula $x \mapsto \overline{x}$, which gives us $1 \mapsto 0$ under this assignment, which means $\phi(u) = 0$. The same holds true if $x = 0$ since we have assumed $\overline{x} \mapsto x$ as well, which would give us $\phi(u) = 0$. So ϕ cannot have any satisfying assignments. Another way to see this, since we have assumed $x \mapsto \overline{x}$ and $\overline{x} \mapsto x$, by Claim 1, this implies that ϕ encodes $x \iff \overline{x}$, which is always false, so ϕ has no satisfying assignments.

Now, to see the other direction, we actually prove the contrapositive. Suppose there does not exist a vertex x such that both $x \mapsto \overline{x}$ and $\overline{x} \mapsto x$. We construct a satisfying assignment for ϕ. We iteratively build up the assignment $u \in \{0,1\}^m$.

- For $i \in [m]$:
  - If x_i has not been assigned, continue to the next iteration of the loop.
  - Check if $x_i \mapsto \overline{x}_i$. If not, set $v = x_i$. Otherwise, set $v = \overline{x}_i$.
  - For each vertex $j \in [2m]$ that is reachable from v in G_ϕ, if $j = 2k$, then assign $\overline{x}_k = 1$, otherwise if $j = 2k-1$, assign $x_k = 1$.

We argue that the above algorithm successfully outputs a satisfying assignment for ϕ. First, consider when the above algorithm does not output a valid satisfying assignment. There are two cases to consider.

First, let v be the unassigned variable we choose and suppose that $v \mapsto x_j$ and $v \mapsto \overline{x}_j$. The algorithm above would try to set $x_j = 1$ and $\overline{x}_j = 1$, which is impossible. However, such a case cannot occur by construction of the graph G_ϕ. First, if $v \mapsto x_j$, by Claim 2, we have that $\overline{x}_j \mapsto \overline{v}$. Similarly, if $v \mapsto \overline{x}_j$, we have that $x_j \mapsto \overline{v}$. So this implies that $v \mapsto x_j \mapsto \overline{v}$, and thus $v \mapsto \overline{v}$. Now our choice of v explicitly assumed that there was no path from v to $\overline{v}$ in G_ϕ, which is a contradiction, so there cannot be both x_j and $\overline{x}_j$ reachable from v in G_ϕ.

Next, suppose that x_j is reachable from v, but we have already assigned $x_j = 0$ in a previous iteration of the loop. Suppose this was due to some other vertex v'. That is, $\overline{x}_j$ was reachable from $v' < v$. Thus, we have $v \mapsto x_j$, which by Claim 2, implies that $\overline{x}_j \mapsto \overline{v}$. This implies that $v' \mapsto \overline{v}$, and we would have assigned $\overline{v} = 1$ in a previous loop. So this is a contradiction, since we choose v that has not already been assigned.

Finally, we argue that the produced assignment is a satisfying assignment. First, the procedure terminates after m iteration. Next, we know that each CNF clause encodes an implication. For each literal in the CNF, if the literal was assigned 1, then every literal following the chain of implications in the graph is assigned 1. So by Claim 1, this yields the final implication of $1 \implies 1$, which evaluates to true. Conversely, for every literal that is assigned 0, all of its predecessors in the graph are also assigned 0 (by Claim 2). So we never get an implication of the form $1 \implies 0$. Thus, ϕ is satisfied by the generated assignment. Therefore, we have shown that $\phi \in \overline{\text{2SAT}}$ if and only if there exists x such that $(G, x, \overline{x}) \in \text{PATH}$ and $(G, \overline{x}, x) \in \text{PATH}$.

Finally, we argue that the reduction is implicitly logspace. That is, we only need $O(\log(n))$ extra space to write the graph G_ϕ to the output tape (recall that we do not pay for the space costs of writing G_ϕ on the output tape, as $|G_\phi| = O(n^2)$). Clearly, it takes $O(\log(n))$ space to write the vertices to the output tape; we simply keep a counter of size $\log(2m) = O(\log(n))$ bits and enumerate the vertices on the output. Now, generating the edges also only takes $O(\log(n))$ space, since we scan the clause $\phi_j = (a_j \vee b_j)$, and then simply use $O(\log(m))$ bits to copy a_j and b_j to the work tape, and write the edges $(\overline{a}_j, b_j)$ and $(\overline{b}_j, a_j)$ to the output tape. Since the reduction is implicitly logspace, we have shown that $\overline{\text{2SAT}} \leq_L \text{PATH}$ and thus $\overline{\text{2SAT}} \in \mathbf{NL}$.

Now, to show that 2SAT is **NL**-complete, we show that $\overline{\text{2SAT}}$ is **NL**-complete. In particular, we actually just do the above reduction in reverse: we show that $\text{PATH} \leq_L \overline{\text{2SAT}}$. Suppose that (G, s, t) is an input to the PATH problem. Without loss of generality, suppose that $V(G) = [m]$ (m vertices labeled 1 to m). Let E be the directed edge set of G. We will actually construct a 2CNF formula by treating G as an implication graph, just as we did for $\overline{\text{2CNF}} \leq_L$ PATH. In particular, for every $(u, v) \in E(G)$, we add the clause $u \Rightarrow v$ to the formula we are constructing ϕ_G. That is, we add $(\overline{x}_u \vee x_v)$ as a clause in ϕ_G. Finally, we add a clause $(x_s \vee x_s)$ and $(x_t \Rightarrow \overline{x}_s) = (\overline{x}_t \vee \overline{x}_s)$ to the formula ϕ_G. Note that it is clear we can construct the formula ϕ_G using only logarithmic space since we only need to construct two literals per edge of E, which requires $O(\log(m)) = O(\log(n))$ bits.

Now, if $|E| = n = O(m^2)$, the formula ϕ_G has $n + 2$ clauses and m variables. By the previous work we did for showing the reverse direction, it is straightforward to show that $s \mapsto t$ in G if and only if ϕ_G is unsatisfiable.

Suppose that there is a path from s to t in G ($s \mapsto t$). Then we show ϕ_G is unsatisfiable. Note that by construction of ϕ_G, Claim 1 still holds in formula ϕ_G. Thus, if $s \mapsto t$, we have that ϕ_G encodes $s \Rightarrow t$ (formally, $x_s \Rightarrow x_t$). Also by construction of ϕ_G, we have encoded $(x_t \Rightarrow \overline{x}_s) = (\overline{x}_t \vee \overline{x}_s)$. So we also have that $x_s \Rightarrow \overline{x}_s$. Now, we have also added the clause $(x_s \vee x_s)$ to ϕ_G. As an implication, this is equivalent to $\overline{x}_s \Rightarrow x_s$. Together, this means ϕ_G encodes $x_s \iff \overline{x}_s$, which always evaluates to false. Thus, ϕ_G must be unsatisifiable.

Now for the other direction, suppose that there is no path from s to t in the graph G. We construct a satisfying assignment for ϕ_G as follows. Assign $x_s = 1$ (note we must do this since $(x_s \vee x_s)$ is in ϕ_G). Now, for every $u \in V$ such that u is reachable from s, we assign the variable $x_u = 1$. Assign all other variables as false.

To see that this generates a satisfying assignment for ϕ_G, first let S be the set of all vertices reachable from S. Notably, $t \notin S$. Now, when we generate the 1's in the assignment, this is done by setting $x_s = 1$ and by following all paths from s to other vertices $u \in S$. This gives a chain of implications $x_s \Rightarrow x_u$ for all $u \in S$. Since we set all of these to 1, then we know that all of these implications are satisfied.

Now, let $T = V \setminus S$ be the set of all vertices not reachable from S in G. For each $v \in T$, we set $x_v = 0$. Consider all edges with starting points in T. That is, any edge of the form (v, u) for $v \in T$ and $u \in V$. In particular, this encodes the implication $x_v \Rightarrow x_u$. There are two cases.

First, suppose that $u \in S$. In this case, $x_u = 1$, we assign $x_v = 0$, which is the implication $0 \Rightarrow 1$, which evaluates to true. Crucially, $(u, v) \notin E$ by assumption since v is not reachable from s, and u is reachable from s, so this edge existing would imply v is reachable from s, but this contradicts our definition of T. Now, in the second case, suppose that $u \notin S$. So $u \in T$. Importantly, we have the implication $x_v \Rightarrow x_u$, which we set to be $0 \Rightarrow 0$, which evaluates to true. Thus, the assignment we have generated assigns all variables and it evaluates to true, so ϕ_G is satisfiable. $\square$