CSC373 Assignment 3

Terminology and Notation Table:

Notation	
$A: B$	A such that B holds
$x_{i,j\ldots k}$	Shorthand for the sequence $x_i, x_j, \ldots, x_k$ j is used for setting a "pattern precedent" (e.g. it's unclear that $\{1,2,\ldots\}$ represents the series $\{2^k\}$, but it is completely clear when we write something like $\{1,2,4,8,\ldots\}$ – we're setting a "precedent" for the pattern) If j is not included, we assume this to be the sequence $x_i, x_{i+1}, x_{i+2}, \ldots x_k$
$\langle x_{i\ldots k}\rangle$	Encoding of the variable sequence $x_{i\ldots k}$ (assumed to be binary encoding)
$\{\langle x, y\rangle \mid P\}$	The set of all encodings of x, y where predicate P holds
$A \leq_p B$	Problem A can be reduced in polynomial time to problem B A is poly-time reducible to B
$s \rightsquigarrow t$	Some path between nodes s and t
$s \rightarrow t$	Directed edge leading from node s to node t
$s \leftrightarrow t$	Undirected edge connecting nodes s, t
$(a\|b)$	Could be a or b
Symbols / Functions / Algorithms / Predicates	
L_P	Language of some problem P
γ_P	Some certificate for problem P
$V_P((x_{1\ldots j}), \gamma)$	Verifier for some problem P that checks if certificate γ is an appropriate solution to the input variables $x_{1\ldots j}$
$\alpha(H)$	Algorithm / function that solves problem H. In other words, $\alpha(H): I_H \mapsto O_H$ Alternative phrasing: algorithm that decides L_H
Problem Names	
BF	Binary Feasibility Problem ($\{0, 1\}$ −feasibility problem)
NDP	Node Disjoint Path decision problem
EDP	Edge Disjoint Path decision problem
EXS	Exam Scheduling decision problem
MSW	Mine Sweeper Satisfiability problem

1. Integer Programming
   a. Dual for Integer Programming
      - Definition:
        - Minimize: $y^T b$
        - Subject to:
          - $y^T A \geq c^T$
          - $y \geq 0$
          - $y \in \mathbb{Z}^n$ (horizontal n-dimensional integer-vector)
      - Integer Programming Weak Duality:
        - Definition: $c^T \bar{x} \leq \bar{y}^T b$
        - Proof:
          - Let:
            - $\bar{x}$ be a feasible solution to the **primal** integer program
            - $\bar{y}$ be a feasible solution to the **dual** integer program
          - Since $\bar{x}$ is feasible, according to the primal we have $A\bar{x} \leq b$
          - Now we multiply both sides by $\bar{y}^T$ to get: $\bar{y}^T A\bar{x} \leq \bar{y}^T b$
          - From the definition of the dual, $(c^T \leq \bar{y}^T A) \Rightarrow (c^T \bar{x} \leq \bar{y}^T A\bar{x})$
          - By transitivity: $c^T \bar{x} \leq \bar{y}^T A\bar{x} \leq \bar{y}^T b$
          - Thus we can conclude $c^T \bar{x} \leq \bar{y}^T b$.

b. Integer Programming Strong Duality Counter Example

- Definition of Strong Duality:
  - (Primal Feasible)∧(Primal Bounded)⇒(Optimal Primal Value = Optimal Dual Value)
  - To disprove an implication we need to find a case where $T \Rightarrow F$, so for this definition, we need to find:
    (Primal Feasible)∧(Primal Bounded)⇒(Optimal Primal Value ≠ Optimal Dual Value)
- Consider a primal-dual pair of integer programs with the following input values:
  - ♦ $(c = [1]) \Leftrightarrow (c^T = [1])$
  - ♦ $(A = [1]) \Leftrightarrow (A^T = [1])$
  - ♦ $(b = [4.5]) \Leftrightarrow (b^T = [4.5])$
- Primal:
  - Integer Program
    - ♦ Maximize x
    - ♦ Subject to:
      - $x \leq 4.5$
      - $x \geq 0$
      - $x \in \mathbb{Z}$
  - Clearly, we can maximize x with $x = 4$, as it is the highest integer that does not violate the constraint that $x \leq 4.5$.
  - Thus, the maximized value of the objective function here is 4
- Dual of Primal:
  - Integer Program:
    - ♦ Minimize $4.5y$
    - ♦ Subject to:
      - $y \geq 1$
      - $y \geq 0$
      - $y \in \mathbb{Z}$
  - Clearly, the lowest we can go with y is $y = 1$, which is the lowest integer that does not violate the constraint that $y \geq 1$.
  - Thus, the minimized value of the objective function here is 4.5.
- We can now say that strong duality does not hold under integer programming, as we have shown a case where:
  - The primal integer program was feasible and bounded
  - The primal's optimal value was **not** equal to the dual's optimal value.

c. $BF = \{0, 1\}$-Feasibility Problem

Inputs	$A \in \mathbb{Z}^{m \times n}$ $b \in \mathbb{Z}^m$
Outputs	Truth value of statement: $(\exists \hat{x} \in \{0, 1\}^n : A\hat{x} \leq b)$

- Language
  - $L_{BF} = \{\langle A, b\rangle \mid ((A \in \mathbb{Z}^{m \times n}) \wedge (b \in \mathbb{Z}^m) \wedge (\exists \hat{x} \in \{0, 1\}^n : A\hat{x} \leq b))\}$
- Proof of NP-Completeness
  - Proof that $L_{BF} \in NP$
    - ♦ Proof that γ_{BF} is polynomial sized
      - γ_{BF} is the $\hat{x}$ in $A\hat{x} \leq b$
      - We know that $\hat{x}$ is a binary vector of length n.
      - Thus, $\langle\gamma_{BF}\rangle$ is of length $n \in O(n) \in O(n^k)$, where k is a fixed constant
    - ♦ Proof that V_{BF} runs in polynomial time

```
# Some notation rules
A.row(i) = i-th row of matrix A (as a vector)
A x B = matrix multiplication of matrices A, B
A.num_rows = number of rows of matrix A
v.dot(u) = dot product of vectors v and u
v[i] = i-th element of vector v

def V_BF(A, b, gamma_BF):

    # Multiply A and certificate to get result
    result = A x gamma_BF

    # Check whether the result satisfies the constraint
    for i in range(len(result)):
        if gamma_BF.dot(A.row(i)) > b[i]:
            return False

    # All constraints must be true at this point
    return True
```

- All the code executed by V_{BF} runs in polynomial time
  - o $result = A \times \gamma_{BF}$: Matrix multiplication runs in polynomial time (DPV)
  - o $result$ is of length $O(m)$, the loop that iterates over it must take $O(m)$ time, which is clearly polynomial
- Thus the whole of V_{BF} must run in polynomial time as well

- Lemma: the following Boolean/Integer conversion system preserves truth values
  - Let:
    - $T, F = True, False$ respectively
    - $x_I = \begin{cases} 1, x = T \\ 0, x = F \end{cases}$ be the integer value of Boolean x
    - $x_B = \begin{cases} T, x \geq 1 \\ F, x = 0 \end{cases}$ be the Boolean value of Integer x
  - Conversion Table

<table>
<tr><th>Conversion</th><th>Truth/Integer Table</th></tr>
<tr><td>Negation/ Subtraction</td><td>

x	x_I	$\neg x$	$1 - x_I$	$(1 - x_I)_B$
T	1	F	0	F
F	0	T	1	T

</td></tr>
<tr><td>Or/ Addition</td><td>

x	y	x_I	y_I	$x \vee y$	$x_I + y_I$	$(x_I + y_I)_B$
T	T	≥ 1	≥ 1	T	≥ 1	T
T	F	≥ 1	0	T	≥ 1	T
F	T	0	≥ 1	T	≥ 1	T
F	F	0	0	F	0	F

</td></tr>
</table>

  - We note that the above conversions maintain truth values: i.e. the Boolean conversion of the integer conversion equals the Boolean value, **provided**:
    - The input to negations are within the set $\{0, 1\}$
    - The inputs of or statements are within the range $[0, \infty)$, where 0 indicates F and ≥ 0 indicates T
- Proof that $(\exists L_{NPC} \in NPC): (L_{NPC} \leq_p L_{BF})$ – pick $L_{NPC} = L_{3SAT}$. Proof that $(L_{3SAT} \leq_p L_{BF})$:
  - Let:
    - $x_{1 \ldots n}$ be the variables in the 3SAT circuit
    - $c_{1 \ldots m}$ be the clauses in the 3SAT circuit
  - An algorithm to solve L_{3SAT} that uses some $\alpha(L_{BF})$:
    - Convert inputs from $3SAT \to BF$ in polynomial time:
      - Convert clauses $c_{1 \ldots m}$ to integer equations:
        - Represent each clause $c_i \in \{c_{1 \ldots m}\}$ with an integer equation $f_i \in \{f_{1 \ldots m}\}$ using the conversion system described above
        - Create the following constraint for each $f_i \in \{f_{1 \ldots m}\}$: $f_i \geq 1$
        - The above is equivalent to each c_i being true, which we need to have if $(c_1 \wedge \ldots \wedge c_m)$ is to be true)
      - Once we have converted our Boolean variables and clauses to integer variables and equations, it is trivial to reformat them into standard form (needed for the integer program's inputs):
        - A is the coefficient matrix of size $m \times n$
          - j-th row represents the j-th equation's coefficients
          - i-th column represents the coefficient for x_i
        - b is the constraint vector of size $m \times 1$

            - j-th element represents the constraint of the j-th equation
        - Since there are and m clauses ($c_{1...m}$) of size 3 (each clause has 3 variables) to convert into linear equations ($f_{1...m}$), conversion into standard form should take time $O(m)$, which is clearly polynomial with respect to the input size
        - We can be sure that the Boolean/integer conversions are valid because all integer versions of variables will be between 0 and 1, which means:
            - All negation conversions are valid (since we only negate single variables, the lowest value of a negation is 0)
            - Since the lowest value of a negation is 0, we don't need to worry about OR statements outputting the wrong result (there will be no negative inputs)
    - Solve the converted inputs of $3SAT$ to BF with some $\alpha(L_{BF})$
        - If $\alpha(BF)$ returns true it means:
            - There exists some $\hat{x} \in \{0, 1\}^n$ such that $A\hat{x} \le b$.
            - By the constraints we set for each f_i, the above means that:

$$\forall f_i \in \{f_{1...m}\}, f_i \ge 1$$

            - We note that a way to get all the $f_{1...m}$ to be ≥ 1 was to give them the binary vector $\hat{x} = \hat{x}_{1...n}$ as input
            - We can use our Boolean/integer conversion system to translate $f_{1...m}$ back into $c_{1...m}$ (since each $f_i \ge 1$, each $c_i = T$)
            - To maintain the truth values of $c_{1...m}$ with respect to the integer values of $f_{1...m}$, the integer inputs to each $f_{1...m}$ must also be passed through the Boolean/integer conversion system as well
            - Thus for every c_j to return true, we can set the input variables $x_{1...n}$ to the (element-wise) Boolean equivalent of $\hat{x}$:
                - $\widehat{x_i} = 1 \Rightarrow x_i = T$
                - $\widehat{x_i} = 0 \Rightarrow x_i = F$
            - Thus, there exists a satisfying assignment of variables for the 3SAT circuit
        - If it returns false:
            - There is no $\hat{x}$ that satisfies the inequality in the converted BF instance
            - Thus there is no assignment of variables that satisfies the 3CNF circuit
    - Therefore, we can directly use the output of some $\alpha(BF)$ as the output of our $3SAT$ algorithm (this is constant time and is therefore a polynomial conversion)
- Therefore we can use an algorithm that solves L_{BF} to solve L_{3SA} , by converting between their respective inputs and outputs in polynomial time.
- Thus, we can say that $L_{3SAT} \le_p L_{BF}$. In other words, we have found a language in NPC that reduces in polynomial time to L_{BF}.

- We have verified that $L_{BF} \in NP$ and $L_{3SAT} \in NPC: L_{3SA} \quad \leq_p L_{BF}$, which means that we can conclude that $L_{BF} \in NPC$

2. Disjoint Paths
    a. Language Formulation
        - $L_{NDP} = \{\langle G = (V, E), \{s_{1\ldots k}\}, \{t_{1\ldots k}\}\rangle |$
          $\forall i \neq j \in \{1 \ldots k\}, \left(\left(\exists (s_i \rightsquigarrow t_i), (s_j \rightsquigarrow t_j) \in G\right) : \left(\forall v_i \in (s_i \rightsquigarrow t_i), v_i \notin (s_j \rightsquigarrow t_j)\right)\right)\}$
    b. Proof that $L_{3SAT} \leq_p L_{NDP}$: we use some $\alpha(L_{NDP})$ to solve L_{3SAT} by converting the I/O in polynomial time
        - Input conversion from $L_{3SAT} \rightarrow L_{NDP}$:
            - Let:
                - $x_{1\ldots n}$ be the variables in the circuit, and x_i be an arbitrary variable
                - $c_{1\ldots m}$ be the clauses in the circuit, and c_j be an arbitrary clause
            - Create the following graph $G_{3SAT \rightarrow NDP}$:
                - Nodes :
                    - For each variable x_i, add the following nodes to $G_{3SAT \rightarrow NDP}$:
                        - $s(x_i)$ (source node for x_i)
                        - $t(x_i)$ (terminal node for x_i)
                        - $T(x_i, a), a \in [1 \ldots m]$ ($a - th$ "intermediate True" node for x_i)
                        - $F(x_i, a), a \in [1 \ldots m]$ ($a - th$ "intermediate False" node for x_i)
                    - For each clause c_j, add the following nodes to $G_{3SAT \rightarrow NDP}$:
                        - $s(c_j)$ (source node for c_j)
                        - $t(c_j)$ (terminal node for c_j)
                - Edges:
                    - For each variable x_i, make two $s(x_i) \rightsquigarrow t(x_i)$ by adding the following edges
                        - Add x_i's "true path", composed of the following edges:
                            - $\left(s(x_i) \rightarrow T(x_i, 1)\right)$
                            - $(\forall a \in [1, \ldots m - 1], T(x_i, a) \rightarrow T(x_i, a + 1))$
                            - $(T(x_i, m) \rightarrow t(x_i))$
                        - Add x_i's "false path", composed of the following edges:
                            - $\left(s(x_i) \rightarrow F(x_i, 1)\right)$
                            - $(\forall a \in [1, \ldots m - 1], F(x_i, a) \rightarrow F(x_i, a + 1))$
                            - $(F(x_i, m) \rightarrow t(x_i))$
                    - For each clause $c_j = (a \vee b \vee c)$, where (WLOG for literals a, b, c) $a \in \{x_i, \neg x_i\}$, add the following edges:
                        - Let d be one of the literals a, b, c.
                        - If d is negated, add edges:
                            - $(s(c_j) \rightarrow T(d, j))$
                            - $(T(d, j) \rightarrow t(c_j))$
                        - Otherwise, add edges:
                            - $(s(c_j) \rightarrow F(d, j))$
                            - $(F(d, j) \rightarrow t(c_j))$
                - Time to create this graph is polynomial:
                    - Each clause and each variable in the original 3SAT problem requires the introduction of:

- 1 node
- Some number of edges that is polynomial with respect to n, m
  - Since the number of nodes and edges is polynomial with respect to the input sizes of 3SAT, we can say that the total number of nodes and edges in $G_{3SAT \to NDP}$ is also polynomial with respect to the input sizes of 3SAT
- Let the source and terminal nodes be as follows:
  - $\{s_{1\ldots n+m}\} = \{s(r), r \in (\{c_{1\ldots m}\} \cup \{x_{1\ldots n}\})\}$
  - $\{t_{1\ldots n+m}\} = \{t(r), r \in (\{c_{1\ldots m}\} \cup \{x_{1\ldots n}\})\}$
  - Making these lists would require at most $O(n+m)$ time (we only need to loop over m clauses and n variables)
  - Thus, this process is also polynomial with respect to 3SAT's input sizes

- We execute $\alpha(L_{NDP})(G_{3SAT \to NDP}, \{s_{1\ldots n+m}\}, \{t_{1\ldots n+m}\})$ to solve NDP for the converted inputs, thereby solving $3SAT$. Proof that this call to $\alpha(L_{NDP})$ returns true $\Leftrightarrow$ $3SAT$ is satisfiable:
  - Assume that the input 3CNF circuit was satisfiable, i.e. $\exists\{x_{1\ldots n}\} \in \{True, False\}^n$ such that the circuit returns True
    - In this case, we can construct the $n+m$ disjoint paths as follows:
      - Between variable nodes:
        - If x_i was True in $\{x_{1\ldots n}\}$: $\left(s(x_i) \rightsquigarrow t(x_i)\right) = x_i$'s "true" path:
          $$(s(x_i) \to T(x_i, 1) \to \cdots \to T(x_i, m) \to t(x_i))$$
        - If x_i was False in $\{x_{1\ldots n}\}$: $\left(s(x_i) \rightsquigarrow t(x_i)\right) = x_i$'s "false" path:
          $$(s(x_i) \to F(x_i, 1) \to \cdots \to F(x_i, m) \to t(x_i))$$
        - By the structure of $G_{3SAT \to NDP}$, we know that all of the $s(x_i) \rightsquigarrow t(x_i)$ are node disjoint with respect to each other (they run "parallel" to each other)
      - Between clause nodes (for some $c_j = (a \vee b \vee c) \in \{c_{1\ldots m}\}$, WLOG for a):
        - By the structure of $G_{3SAT \to NDP}$, we know that there are three possibilities for $s(c_j) \rightsquigarrow t(c_j)$:
          - $s(c_j) \to F(a, j) \to t(c_j)$
          - $s(c_j) \to F(b, j) \to t(c_j)$
          - $s(c_j) \to F(c, j) \to t(c_j)$
        - Since c_j is an or-statement, at least one $d \in \{a, b, c\}$ must be true in order for c_j to be true
          - If literal d is not negated in c_j:
            - d is equal to the truth value of its underlying variable x_d, thus for d to be true we need x_d to be true as well
            - By the structure of $G_{3SAT \to NDP}$ the only path available to us is $s(c_j) \rightsquigarrow t(c_j) = s(c_j) \to F(x_d, j) \to t(c_j)$ (non-negated literals take the "false" path)
            - Clearly, this path does not intersect with the occupied "true" path of x_d, and these paths are node disjoint
          - If literal d is negated in c_j:

        - d is equal to the opposite truth value of its underlying variable x_d, thus for d to be true we need x_d to be false
        - By the structure of $G_{3SAT \to NDP}$ the only path available to us is $s(c_j) \rightsquigarrow t(c_j) = s(c_j) \to T(x_d, j) \to t(c_j)$ (negated literals take the "true" path)
        - Clearly, this path does not intersect with the occupied "false" path of x_d, and these paths are node disjoint
    - Thus we have constructed $n + m$ node-disjoint paths that connect each source-terminal pair, which means that our call to $\alpha(L_{NDP})$ will return True when the input circuit is satisfiable
  - Assume that there are $n + m$ node-disjoint paths connecting $s(x_i), t(x_i)$ and $s(c_j), t(c_j)$ in $G_{3SAT \to NDP}$.
    - We can now construct a satisfying variable assignment $\{x_{1\dots n}\}$ that makes the 3SAT circuit return true:
      - If $s(x_i) \rightsquigarrow t(x_i)$ uses the "true" path of x_i, we set x_i to be true.
        - In this case, all the clauses that contain a non-negated x_i will be true
      - If $s(x_i) \rightsquigarrow t(x_i)$ uses the "false" path of x_i, we set x_i to be false.
        - In this case, all the clauses that contain a negated x_i will be true
      - Since all variables are either be negated or non-negated in every clause, this assignment essentially sets all clauses to true, which makes the whole circuit true
    - Thus, if we have $n + m$ node disjoint paths, we can be certain that the input 3CNF circuit is satisfiable
- We have shown that the result of calling $\alpha(L_{NDP})(G_{3SAT \to NDP}, \{s_{1\dots n+m}\}, \{t_{1\dots n+m}\})$ is true iff the input circuit to 3SAT was satisfiable.
- Thus, the result of our call to $\alpha(L_{NDP})$ on a graph built in time polynomial with respect to n, m is logically equivalent as to whether the 3CNF circuit $(\{x_{1\dots n}\}, \{c_{1\dots m}\})$ is satisfiable
- Therefore, we have shown that $L_{3SAT} \leq_p L_{NDP}$ (the inputs from 3SAT can be converted to the inputs of MSW in poly-time, and the output of $\alpha(L_{NDP})$ called on these inputs is equal to whether the input circuit belongs in L_{3SAT}).

c. Proof of Corollary that $L_{NDP} \in NPC$:

- Proof that $L_{NDP} \in NP$
  - Proof that γ_{NDP} is polynomial sized
    - We let γ_{NDP} be the list of (supposed) node disjoint paths that link the k- many source and terminal nodes together
      - Each path is represented according to nodes:
        - Example: represent $(A \to E \to B)$, using list $[A, E, B]$
        - Representation has size $O(|V|)$ per-path (we can't have more than $|V|$ nodes in a path)
      - There are k-many paths in the path-list (we need one path per (s, t) pair, and there are k such pairings)
    - Thus, the total size of γ_{NDP} is $O(k) * O(|V|) \in O(k|V|)$, which is polynomial with respect to the size of inputs V and k
  - Proof that V_{NDP} runs in polynomial time
    - Algorithm:

```
# Keep a lookup array of intermediate nodes to keep track of what
# we've already seen for O(1) lookup time

# Assume that the graph's nodes have id numbers 1…|V|
# Assume that s_nodes and t_nodes are aligned and have indices 1…k
# Assume that the certificate's paths are aligned with
#   s_nodes/t_nodes (i.e. the i-th path connects s_i to t_i)

def V_NDP(graph, s_nodes, t_nodes, gamma_NDP):

    # Array of size |V| to keep track of seen nodes
    seen_nodes = [False] * len(graph.nodes)

    # This is k
    k = len(s_nodes)

    for i in range(k):
        s_t_pair = (s_nodes[i], t_nodes[i])
        cur_path = gamma_NDP[i]

        # Check if the path connects s_i with t_i
        if (cur_path[0], cur_path[-1]) != s_t_pair:
            return False # abort: path fails to connect s_i -> t_i

        # Check all the path's nodes
        else:
            # Check if any node has been seen in another path
            for node in cur_path:

                # Abort if we've seen this node in another path
                if seen_nodes[node.id_number]: return False

                # Indicate we've seen the node
                else: seen_nodes[node.id_number] = True

    # At this point we know that all the paths connect the s,t
    # nodes they say they're supposed to, and that none of them
    # have the same nodes
    return True
```

- This algorithm has runtime $O(k|V|)$, which is polynomial with respect to the size of inputs V, k:
  - Outer loop runs $O(k)$ times (one for each path in the certificate)
    - Inner loop runs $O(|V|)$ times (one for each node in a path, paths have length $O(|V|)$, and only executes constant-time instructions
  - Thus the outer loop runs in $O(k) * O(|V|) \in O(k|V|)$ time. This is the dominant runtime because the rest of the algorithm executes instructions with lower time-bounds.
- Proof that $(\exists L_{NPC} \in NPC)$: $(L_{NPC} \leq_p L_{NDP})$: verified in the previous sub-question, where we proved that $L_{3SAT} \leq_p L_{NDP}$ (we know $L_{3SAT} \in NPC$).

- Since we have shown that $L_{NDP} \in NP$ and that some NPC language (L_{3SAT}) reduced to L_{NDP}, we can conclude that $L_{NDP} \in NPC$

d. P-Space Classification of EDP

- Checking the validity of proposed solutions to EDP can be done in poly-time using a similar algorithm to γ_{NDP} (instead of keeping track of seen nodes, keep track of seen edges)
- Can do a poly-time reduction of $3SAT$ to EDP with a similar graph conversion, but with the following changes:
  - Add 2 more nodes to each true/false path of each x_i
  - Instead of "piping $s(c_j) \rightsquigarrow t(c_j)$ directly through some "true"/"false" path node, we pipe it through two neighbouring nodes in the "true"/"false" path node
  - The above ensures that our source-terminal paths share edges in the event of satisfiability
- Since we can verify L_{EDP} in polynomial time and $L_{3SAT} \in NPC \leq_p L_{EDP}$, we can say that $L_{EDP} \in NPC$

3. K-Coloring
   a. Poly-time Algorithm to Decide 2-Color-ability
      - English Description
         - We do a breadth first search from some arbitrary node R with the following modifications:
            - Color R with S. Let T be the opposite color of S, and vice versa.
            - For every uncolored node we encounter in the algorithm, color it using the color opposite to its parent
               - Example: we color the neighbours/"children" of R with T (since R has color S), and we color the "grandchildren" of R with S (as their parents had color T), and so on
            - If we encounter a node that has been colored already, and this node has the same color as the color we are about to use, we abort the algorithm – the graph cannot be 2-colored
            - If we don't run into the above case at all during our run of BFS, our graph can be 2-colored
      - Correctness
         - Assume we encounter a node that has the same color as a node we just painted
         - Then in order to properly color it with the current color, we'd have to alternate the colors of its neighbours
         - We can't do this as we would have to keep alternating colors of the neighbours' neighbours and so on to keep the coloration property
         - If we did the above, we'd effectively reverse the coloration of the entire graph
         - Thus there's no way paint this node
      - Running Time: $O(|V| + |E|)$
         - BFS has a runtime of $O(|V| + |E|)$ (CLRS 22.2)
         - We can store colors as attributes or in an array indexed by node ID number for $O(1)$ color inspection and modification time
         - The only modifications that we did to BFS are the addition of color assignments and neighbour/parent color inspections, which each take $O(1)$ time
         - These color operations do not need to be put inside their own loops
            - Assignment can be done after popping
            - Neighbour color inspections can be done within in the inner-for loop of BFS (exploring the current node)
            - Parent color is trivial to inspect (we can also keep track of parents/discoverers when we push nodes into the queue)
         - Thus, the runtime is still $O(|V| + |E|)$, as only $O(1)$ function calls were added

b. Exam Scheduling NP-Completeness (let λ be the curly l (number of students), I can't find it in Word)

- Language Formulation
  - Let:
    - $E_i \subset \{F_{1\ldots k}\}$ be a subset of exams scheduled for timeslot i
    - $\sigma_i \subset \{F_{1\ldots k}\}$ be a subset of exams that student i needs to take
  - $L_{EXS} = \{\langle F_{1\ldots k}, S_{1\ldots \lambda}, h\rangle \mid$
    $(\exists \mathcal{H} \leq h) \wedge (\exists E_{1\ldots \mathcal{H}} \subset \{F_{1\ldots k}\}) \colon \left(\forall S_i \in \{S_{1\ldots \lambda}\}, (|\sigma_i \cap E_i| < 2)\right)\}$
- Proof that $L_{EXS} \in NPC$
  - Proof that $L_{EXS} \in NP$:
    - Proof that γ_{EXS} is polynomial sized
      - We let our certificate for this problem be an exam schedule
      - We can represent an exam schedule as an 2D array $\gamma_{EXS} = S_{h \times k}$, where:
        - $S[i,j] = \begin{cases} 1 \; if \; final \; j \; is \; in \; timeslot \; i \\ 0 \; otherwise \end{cases}$
    - Proof that V_{EXS} runs in polynomial time
      - Algorithm:

```
# Let each student's final exams be represented as a bit-vector F
# - if they have a final in some course i,
# F[i] = 1, otherwise 0

# Let each timeslot be a bit-vector S, where S[i] = 1
# if F_i is running during this timeslot

def V_EXS(students, finals, gamma_EXS):
    # Go through every student
    for student in students:

        # Check the exam schedule
        for timeslot in gamma_EXS:

            # If the student has more than 1 final during the
            # selected timeslot, dot product of vectors
            # will exceed 1
            if dot(student.courses, timeslot) > 1:
                return False

    # At this point we've verified that no student has a conflict
    return True
```

      - Runtime of the above verifier is polynomial:
        - Student for-loop runs $O(\lambda)$ times
          - Timeslot for-loop runs $O(h)$ times
            - Dot-product of two k-length vectors takes $O(k)$ time
          - Thus time-slot for-loop runs in $O(hk)$ time
        - Thus student for-loop and therefore the whole algorithm runs in $O(hk\lambda)$ time, which is clearly in polynomial time with respect to input sizes h, k, λ
    - Since we have proven that γ_{EXS} is polynomial sized, and the runtime of V_{EXS} is polynomial, we can say that $L_{EXS} \in NP$

- Proof that $(\exists L_{NPC} \in NPC): (L_{NPC} \leq_p L_{EXS})$ – pick $L_{NPC} = L_{COL}$. Proof that $(L_{COL} \leq_p L_{EXS})$:
  - An algorithm to solve L_{COL} that uses some $\alpha(L_{EXS})$:
    - Let the inputs to L_{COL} be:
      - Graph $G = (V, E)$, where $v \in V, e \in E$ are arbitrary nodes/edges
      - Number of colors $k \in \mathbb{Z}^+$
    - Convert inputs from $L_{COL} \rightarrow L_{EXS}$ in polynomial time:
      - Represent each node v as a final exam v (conversion takes $O(|V|)$ time, looping over nodes)
      - Represent each edge $(u \leftrightarrow v)$ as a student taking final exams u and v (conversion takes $O(|E|)$ time, looping over edges)
      - Let k (number of colors) be h (number of timeslots) (simple assignment takes $O(1)$ time)
    - Solve the converted $L_{COL} \rightarrow L_{EXS}$ with some $\alpha(L_{EXS})$
      - If it returns true it means:
        - Given h timeslots, it's possible to schedule exams such that every student has at most 1 exam per time slot
        - Rephrasing: given h timeslots, every student is "connected" to two exams, which each occupy separate timeslots
        - Reversing input translation: given k colors, every edge is connected to two nodes, which each have a separate color
        - Rephrasing the translation: given k colors, no two nodes of the same color are connected
        - The above sentence is logically equivalent to the input graph being k-colorable
      - If it returns false, it means:
        - It's not possible to construct such an exam schedule
        - Thus there exists some student that has more than 1 exam in the same time slot
        - Rephrasing: there exists some student that "connects" two exams in the same time slot together
        - Reversing the input translation: there exists some edge that connects 2 nodes of the same color together
        - This violates the k-coloring property, thus it's not possible to k-color the input graph
  - Therefore, we can use an algorithm that solves L_{EXS} to solve L_{COL}, by converting between their respective inputs and outputs in polynomial time.
  - Thus, we can say that $L_{COL} \leq_p L_{EXS}$. In other words, we have found a language in NPC that reduces in polynomial time to L_{EXS}.
- We have verified that $L_{EXS} \in NP$ and $L_{COL} \in NPC: L_{COL} \leq_p L_{EXS}$, which means that we can conclude that $L_{EXS} \in NPC$

4. Deciding Daggers

Note: $w[1 \dots i] = w_1 \dots w_i$ (inclusive slicing), and $w[1,0] = \mathcal{E}$

a. Semantic Array

- $C[i]$ = whether or not the word $w[1 \dots i]$ belongs to $L^\dagger$
- Answer: contained inside $C[n]$ (whether the whole word $w[1 \dots n]$ belongs to $L^\dagger$

b. Computational Array

- $C[i] = \begin{cases} True, i = 0 \\ \bigvee_{k \in \{1 \dots i\}} (\mathrm{C[k]} \wedge \mathrm{A(w[(k+1) \dots i])}), i > 0 \end{cases}$

c. Proof of Array Equivalence

- Base Case ($i = 0$): this subword is the empty string $\mathcal{E}$, which we know is inside $L^\dagger$ by convention, thus we return True
- Recursive Case ($i > 0$):
  - We note that $L^\dagger = L^\dagger L$. Proof:
    - $\bigcup_{i=0}^{\infty} L^i$ (definition from assignment)
    - $= \{\mathcal{E}\} \cup L \cup L^2 \cup L^3 \cup \dots$ (expanding the union)
    - $= \{\mathcal{E}\} \cup L \cup L^1 L \cup L^2 L \cup \dots$ (recursive definition of language powers)
    - $= \{\mathcal{E}\} \cup ((L^0 \cup L^1 \cup L^2 \cup \dots)L)$ ("factoring" out L)
    - $= \{\mathcal{E}\} \cup \left(\left(\bigcup_{i=0}^{\infty} L^i\right)L\right)$ (re-expressing the infinite union of powers)
    - $= \{\mathcal{E}\} \cup L^\dagger L$ (the infinite union is the dagger)
    - $= L^\dagger L$ (since $\mathcal{E}$ is already included in the definition of $L^\dagger$)
  - This means that any word in $L^\dagger$ is the concatenation of a word in $L^\dagger$ with a word in L
    - Formally: $\forall w_3 \in L^\dagger, w_3 = w_1 w_2, (w_1 \in L^\dagger \wedge w_2 \in L)$
  - Thus to check if a word is in $L^\dagger$, we need to have the following two conditions be true (for any way of splitting the word in two)
    - The first part of the word is in $L^\dagger$
    - The remainder of the word is in L
  - This is essentially what we are doing in the computational array: we check whether $\exists k \in \{1 \dots i\}$ such that the following two conditions are met:
    - $C[k]$ is true (i.e. the first part of the word up to index k is part of $L^\dagger$)
    - $A(w[k \dots i])$ is true (i.e. the remaining part of the word (from indices k to i) is part of L)

d. Runtime of Algorithm

- Let:
  - A's runtime be $O(f(m))$, where:
    - m is the size of the input word to A
    - f is some positive, non-decreasing function bounded by a polynomial (has to be bounded because question specifies that A is a poly-time algorithm)
  - w's size be $O(n)$ (input word)
- Each element in the computational array C essentially has to run a for-loop for $i \in O(n)$ iterations.
  - During this for-loop (call some arbitrary iteration k), we do the following:
    - Access $C[k]$. This takes $O(1)$ time
    - Call $A(w[k \dots i])$. This takes $O(f(n))$ time.

        - We call A on an instance of size $i - k$
        - This means that we experience a runtime of $O(f(i-k))$
        - We note that $i - k \leq n$, and since f is non-decreasing function, $f(i-k) \leq f(n)$
        - Thus this step takes $O(f(n))$ time
      - Clearly, the total of the steps inside this for-loop is in $O(1 + f(n))$ time
  - Since the for-loop runs $O(n)$ iterations, the total runtime of the loop is
    $O(n) * O\big(1 + f(n)\big) \in O(n + nf(n))$
- We note that the computational array C has $O(n)$ elements, and since each element has to run the aforementioned for-loop, we end up with a final running time of:
  $O(n) * O\big(n + nf(n)\big) \in O(n^2 + n^2 f(n))$
- This run-time is clearly polynomial with respect to the input size (n)
  - n^2 is polynomial with respect to n
  - $n^2 f(n)$ is polynomial with respect to n (if $g(n)$ is the bounding polynomial of $f(n)$, $n^2 g(n)$ is the bounding polynomial of $n^2 f(n)$)
- Therefore, if we can decide whether a word belongs in L in polynomial time, we can also decide whether it belongs in $L^\dagger$ in polynomial time

5. Minesweeper Satisfiability
   a. Language Formulation
      - Let:
         - $\lambda: V \mapsto \mathbb{Z}^{\geq -1}\}$ be the labelling function (nodes with no label are labelled -1)
         - $\mu: V \mapsto \{0,1\}$ be the mine-placement function (may only place on nodes labelled -1)
         - $\sigma: V \mapsto \mathbb{Z}^{\geq 0}$ be a function that returns the following:
            - ♦ $\sigma(v) = \sum_{\forall(v \leftrightarrow u) \in E} \mu(u)$ = number of mines placed on the neighbours of v
      - $L_{MSW} = \left\{ \langle G = (V,E), \lambda \rangle \middle| \left(\exists \mu : \left(\forall v \in V, (\lambda(v) \neq -1) \Rightarrow (\lambda(v) = \sigma(v)) \right) \right) \right\}$
   b. $3SAT$ Reduction
      - Input conversion from $L_{3SAT} \to L_{MSW}$:
         - Let:
            - ♦ $x_{1\ldots n}$ be the variables in the circuit, and x_i be an arbitrary variable
            - ♦ $c_{1\ldots m}$ be the clauses in the circuit, and c_j be an arbitrary clause
         - Create the following mine-graph $G_{3SAT \to MSW}$:
            - ♦ Nodes :
               - For each variable x_i, add the following nodes to $G_{3SAT \to MSW}$:
                  - $t(x_i)$ ("true" node for x_i)
                  - $m(x_i)$ ("middle" node for x_i)
                  - $f(x_i)$ ("false" node for x_i)
               - For each clause c_j, add the following nodes to $G_{3SAT \to NDP}$:
                  - $t(c_j)$ ("true" node for c_j)
                  - $m(c_j)$ ("middle" node for c_j)
                  - $f(c_j)$ ("false" node for c_j)
            - ♦ Edges:
               - For each variable x_i add the following edges:
                  - $t(x_i) \leftrightarrow m(x_i)$
                  - $m(x_i) \leftrightarrow f(x_i)$
               - For each clause $c_j = (a \vee b \vee c)$, where (WLOG for literals a, b, c) $a \in \{x_i, \neg x_i\}$, add the following edges:
                  - Between the clause's nodes:
                     - ♦ $t(c_j) \leftrightarrow m(c_j)$
                     - ♦ $m(c_j) \leftrightarrow f(c_j)$
                  - Between $m(c_j)$ and the three literals' nodes (where $d \in \{a, b, c\}$, and x_d is the underlying variable of d):
                     - ♦ If d is NOT negated: $m(c_j) \leftrightarrow t(x_d)$
                     - ♦ If d is negated: $m(c_j) \leftrightarrow f(x_d)$
            - ♦ Labelling:
               - For each variable x_i, set $\lambda(m(x_i)) = 1$
               - For each clause c_j, set $\lambda\left(m(c_j)\right) = 3$
            - ♦ Time to create this graph is polynomial:
               - Each variable x_i requires the introduction of:

- 3 nodes: $t(x_i), m(x_i), f(x_i)$
- 2 edges: $(t(x_i) \leftrightarrow m(x_i)), (m(x_i) \leftrightarrow f(x_i))$
- 1 labelling: $\lambda(m(x_i)) = 1$

- Each clause $c_j = (a \vee b \vee c)$ (with x_a, x_b, x_c being the underlying variables of literals a, b, c) requires the introduction of:
  - 3 nodes: $t(c_j), m(c_j), f(c_j)$
  - 5 edges:
    - $(t(c_j) \leftrightarrow m(c_j)), (m(c_j) \leftrightarrow f(c_j))$
    - $(m(c_j) \leftrightarrow (t|f)(a)), (m(c_j) \leftrightarrow (t|f)(b)), (m(c_j) \leftrightarrow (t|f)(c))$
  - 1 labelling: $\lambda\left(m(c_j)\right) = 3$
- Since each clause and variable introduces a constant number of nodes/edges/labels, this conversion from the inputs of $3SAT$ to the inputs of MSW clearly takes polynomial time

- We execute $\alpha(L_{MSW})(G_{3SAT \to MSW}, \lambda)$ to solve MSW for the converted outputs, thus solving $3SAT$. Proof that this call to $\alpha(L_{MSW})$ returns true $\Leftrightarrow$ the input circuit is satisfiable:
  - Assume that the input circuit was satisfiable. We can place mines on $G_{3SAT \to MSW}$ such that $\alpha(L_{MSW})$ returns true. Proof:
    - We can use the following partial mine placement plan to satisfy the MSW restrictions (for every variable x_i in a satisfying assignment $\{x_{1 \dots n}\}$):
      - $x_i = True \in \{x_{1 \dots n}\} \Rightarrow \mu(t(x_i)) = 1, \mu(f(x_i)) = 0$
      - $x_i = False \in \{x_{1 \dots n}\} \Rightarrow \mu(f(x_i)) = 1, \mu(t(x_i)) = 0$
    - Under this partial mine placement plan, if the circuit is satisfiable, mines can be placed around every $m(c_j)$ such that $\lambda(c_j) = 3$. Proof:
      - If the circuit was satisfiable, all c_j's contained a literal that was true
      - Let d be a true literal in c_j with underlying variable x_d.
        - If d was negated
          - x_d had to be false for d to be true
          - In this case $G_{3SA \ \to MSW}$ had node $m(c_j) \leftrightarrow f(x_d)$
          - Under the partial placement plan, if $x_d = F, \mu(f(x_d)) = 1$
        - If d was not negated
          - x_d had to be true for d to be true
          - In this case $G_{3SA \ \to MSW}$ had node $m(c_j) \leftrightarrow t(x_d)$
          - Under the partial placement plan, if $x_d = T, \mu(t(x_d)) = 1$
      - Thus, under the partial placement plan $\sigma\left(m(c_j)\right) \geq 1$
      - Since $m(c_j)$ is connected to the not yet mine-occupied nodes $t(c_j), f(c_j)$, we can always make $\sigma\left(m(c_j)\right) = 1$ to satisfy $\lambda = \sigma$ by placing the (at most 2) "missing" mines on $t(c_j), f(c_j)$

- Assume that the call to $\alpha(L_{MSW})$ returned true. Proof that this implies the input circuit is satisfiable:
  - Let:
    - $c_j = (a \lor b \lor c)$
    - x_a, x_b, x_c = underlying variables of literals a, b, c
    - $d \in (a, b, c)$ be some arbitrary literal, with underlying variable x_d
  - Each $m(c_j)$ has a mine on some "variable node" $(t|f)(x_d)$. Proof:
    - By the structure of $G_{3SAT \to MSW}$, $m(c_j)$ is connected to the following 5 nodes:
      - "Clause nodes": $t(c_j), f(c_j)$
      - "Variable nodes": $(t|f)(x_a), (t|f)(x_b), (t|f)(x_c)$
    - We have to place three mines on these nodes for $\lambda(c_j) = 3$ to be satisfied
    - We note that two of the nodes are "clause nodes"
    - Thus, we always have at least one leftover mine that needs to be put on a "variable node"
  - Recall the labelling λ for $G_{3S \quad \to MSW}$ that we constructed in the poly-time conversion from $3SAT \to MSW$:
    - $\forall x_i \in \{x_{1\ldots n}\}$: $\lambda(m(x_i)) = 1$
    - Thus, $\forall x_i \in \{x_{1\ldots n}\}, (\mu(t(x_i)) = 1) \oplus (\mu(f(x_i)) = 1)$ (otherwise the labelling above would be violated)
  - We can use the μ values of each $(t|f)(x_i)$ to create the circuit-satisfying assignment $\{x_{1\ldots n}\}$:
    - $\mu(t(x_i)) = 1 \Rightarrow x_i = T$
    - $\mu(f(x_i)) = 1 \Rightarrow x_i = F$
    - Since only one of the "true"/"false" nodes of x_i can have a mine, these assignment, these assignment rules always result in a certain value for x_i
  - Proof that the above assignment rule satisfies the circuit:
    - At least one of the "variable nodes" connected to $m(c_j)$ has a mine on it
    - This node is either has to be $t(x_d)$ or $f(x_d)$.
      - Assume that the node is $t(x_d)$.
        - By our $3SAT \to MSW$ input conversion, $m(c_j)$ is only connected to $t(x_d)$ when d is NOT negated in c_j
        - If d is NOT negated, x_d needs to be true for d to be true
        - We have assumed that $\mu(t(x_d)) = 1$
        - Thus $\mu(t(x_d)) = 1$ where x_d needs to be true
      - Assume that the node is $f(x_d)$.
        - By our $3SAT \to MSW$ input conversion, $m(c_j)$ is only connected to $f(x_d)$ when d is negated in c_j
        - If d is negated, x_d needs to be false for d to be true
        - We have assumed that $\mu(f(x_d)) = 1$
        - Thus $\mu(f(x_d)) = 1$ where x_d needs to be false

- We have shown that the result of calling $\alpha(L_{MSW})(G_{3SAT \to MSW}, \lambda)$ is true iff the input circuit to 3SAT was satisfiable.
- Thus, the result of our call to $\alpha(L_{MSW})$ on a graph built in time polynomial with respect to n, m is logically equivalent as to whether the 3CNF circuit $(\{x_{1 \dots n}\}, \{c_{1 \dots m}\})$ is satisfiable
- Therefore, we have shown that $L_{3SAT} \leq_p L_{MSW}$ (the inputs from 3SAT can be converted to the inputs of MSW in poly-time, and the output of $\alpha(L_{MSW})$ called on these inputs is equal to whether the input circuit belongs in L_{3SAT}).

c. Proof of Corollary that $L_{MSW} \in NPC$:

- Proof that $L_{MSW} \in NP$
  - Proof that γ_{MSW} is polynomial sized
    - We let γ_{MSW} be the mine-placement function μ that makes this graph (supposedly) mine-consistent
      - It can be represented as a hash-map/array that maps nodes μ values (whether there is a mine on the given node)
      - This map is of size $O(|V|)$, as at most $O(|V|)$ nodes are unlabelled
      - Each element has size $O(1)$ (1 or 0 is 1 bit)
    - Thus, the total size of γ_{MSW} is $O(1) * O(|V|) \in O(|V|)$, which is polynomial with respect to the size of input V
  - Proof that V_{MSW} runs in polynomial time
    - Algorithm:

```
# Function to count the number of mine nodes around a given node
def sigma(graph, node, mu):
    num_mines = 0
    for neighbour in graph.neighbours_of(node):
        num_mines += mu[neighbour.id]
    return num_mines

# Verification function
def V_MSW(graph, lambda, gamma_MSW):

    # Check each labelled node to see if sigma aligns with lambda
    for node in graph.nodes:
        if lambda[node.id] != -1:
            if sigma(graph, node, gamma_MSW) != lambda[node.id]:
                # Abort: labels don't line up!
                return False

    # At this point all labelled nodes are mine-consistent
    return True
```

    - This algorithm has runtime $O(|V|^2)$, which is polynomial with respect to the size of inputs V, E:
      - Main for-loop does $O(|V|)$ iterations (in the worst case it has to iterate over each node)
        - Sigma function runs in $O(|V|)$ time
          - Each node in the graph has at most $|V| - 1 \in O(|V|)$ neighbours
          - Looking up the μ value of the neighbour takes $O(1)$ time (since we can represent μ as a hashmap/array that maps nodes to μ values
      - Thus, the main-for-loop has a total runtime of $O(|V|) * O(|V|) \in O(|V|^2)$
  - Proof that $(\exists L_{NPC} \in NPC): (L_{NPC} \leq_p L_{MSW})$: verified in the previous sub-question, where we proved that $L_{3SAT} \leq_p L_{MSW}$ (we know $L_{3SAT} \in NPC$).
- Since we have shown that $L_{MSW} \in NP$ and that some NPC language (L_{3SAT}) reduced to L_{MSW}, we can conclude that $L_{MSW} \in NPC$