NP Reduction Visualizer Documentation

May 5, 2025

1 Team Information

1. Alice Zhou (aliceyzh@usc.edu)
2. Alice Wei (alicewei@usc.edu)
3. Hao Jiang (hjiang86@usc.edu)
4. Wendy Wu (wuwendy@usc.edu)

2 Installation

Our NP Reduction Visualizer is a python based program with additional dependency such as numpy. To run the program, make sure you have python installed. If not, install the latest version of python here. To clone the project, run:

```
git clone git@github.com:wuwendyy/NP-reduction-visualization.git
```

(Optional) To configure an environment with anaconda, run command:

```
conda create -n npvis python=3.12
conda activate npvis
```

With or without conda setup, download the required dependency by pip:

```
cd NP-reduction-visualization
pip install -e .
```

3 Running Example Programs

We have implemented sample programs using our visualizer. To visualize a sample reduction of 3SAT to Independent Set, run in root directory (NP-reduction-visualization):

```
python -m tests.reduction_test.test_3sat_to_is
```

You will then see a visualizer window powered by Pygame with 3SAT problem on the left and Independent Set Problem on the right:

Figure 1: Initial Window

By clicking one element in 3SAT, the program will also highlight the corresponding element in independent set, and vice versa. Re-clicking the element disables its highlight:

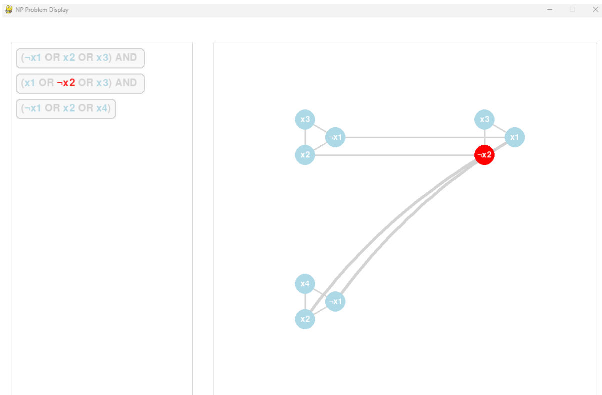


Figure 2: Clicking the Elements

Key S shows the solution of the two problems. We visualize the solution with a set of colors. Elements with the same colors belongs to the same group in solution, such as being assigned as true in 3SAT. Click key S again can disable the solution visualization.

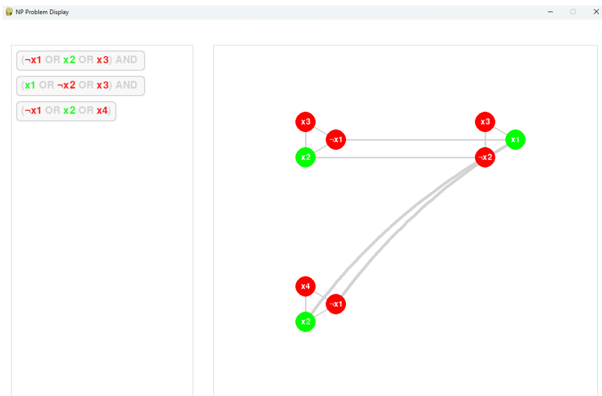
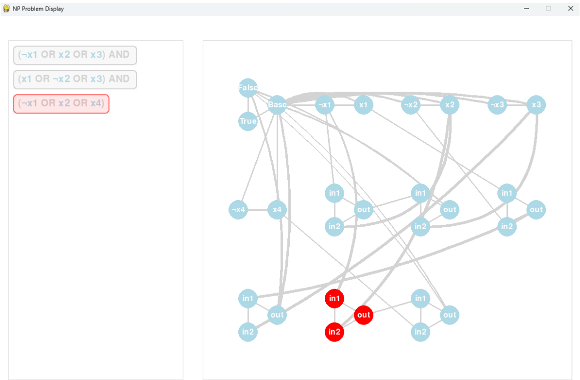
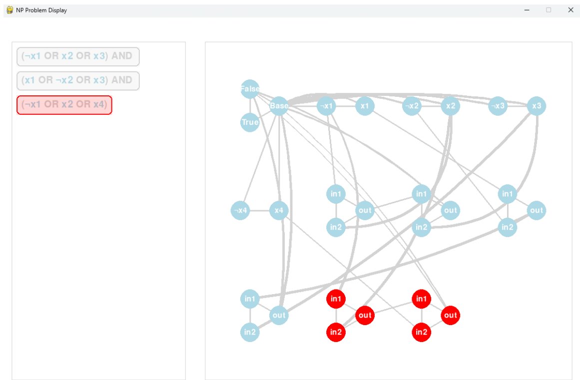


Figure 3: Solution Visualization

The program can also show correspondence between one element from problem one to multiple elements in problem two. Here in 3SAT to 3Colors reduction, you can click multiple elements in problem two that matches the same 3SAT clause. The more you click, the brighter the color for the clause:



(a) 3 Elements Selected



(b) 6 Elements Selected

Run this example by:

```
python -m tests.reduction_test.test_3sat_to_3color
```

4 Included Classes

4.1 Element and SubElement

The element classes are the underlying data structures that we build problems and reductions with. All elements inherit from the base Element class, and need to implement the display() and parse() functions.

4.1.1 SubElement

SubElement is an abstract class with 3 attributes and no methods.

id: an int, unique identifier

name: a string, used in display

color: current color when displayed

Node

Nodes are used in graphs. Nodes are displayed as a circle, with the name in the center.

Attributes:

selected: boolean to determine if a node is selected

location: coordinates (x, y)

default_color: color when not highlighted

neighbors: a list of all neighbor node's id

Methods:

toggle_highlight(self, highlight_color): toggle the state of the highlight to the given color (default is light pink)

add_neighbor(self, neighbor): append an id to neighbors

__repr__(): print node ID

__lt__(): less than operator, compare nodes by ID

Edge
Each edge is made up of 2 nodes.

Attributes:

node1: first Node element
node2: second Node element
selected: boolean to determine if an edge is selected
default_color: color when not highlighted

Variable
Each variable can be true or false.

Attributes:

is_negated: if the variable is false, is_negated is true
clause_id: the clause this variable belongs to

Methods:

toggle_highlight(self, highlight_color): toggle the state of the highlight to the given color (default is light pink)

Clause
Each clause contains a list of variables

Attributes:

variables: list of Variable elements

Methods:

evaluate(self, highlight_color): given a solution, evaluate if this clause is true. Solution should be a dict where each variable is assigned either True or False.

4.1.2 Element

Element is an abstract class with 3 methods.

parse(self, filename): read in input from a file
display(self, screen): display this element
handle_event(self, event): determine which part of the Element is being clicked on

Graph
The Graph Element can be used to represent various different types of graphs. A separate file, `graph_drawing_utils.p` contains helper functions for displaying the graph that determine intersection, edge thickness, edge shape, etc. Examples of using graphs can be found in the `graph_introduction.py` file.

Input File Format:

The Graph class' default parser takes in a file name and creates a Graph.

Node: Each node's name should be written on its own line.

Edge: Each edge on a new line with parentheses, with the two node names (ex: (X1, X2) represents an edge between node X1 and node X2).

Group: Each group should be surrounded by brackets, with all nodes in a comma separated list (ex: [X1, X2, X4] will place nodes X1, X2, and X4 in a group).

```
X1
X2
X3
X4
X5
X6
X7
X8
(X1, X3)
(X2, X3)
(X2, X6)
(X4, X8)
(X7, X8)
[X1, X2, X3]
[X4, X5, X6, X7, X8]
```

Figure 5: sample .txt file input for a graph with 8 nodes, 5 edges, 2 groups

Attributes:

nodes: a set of Node elements

edges: a set of Edge elements

groups: a list of lists of Node elements. Nodes in a group will be displayed together.

node_dict: a dict storing each id: Node for accessing nodes by id

original_bounding_box: an `np.array()` that determines the bounds of the graph

node_radius: the size of each node element in the display

Methods:

add_node(self, node: Node): add a node to the set of nodes; updates the node_dict

add_edge(self, edge: Edge): add an edge to the set of edges

add_group(self, group: [Node]): add a group to the list of groups

has_edge(self, n1: Node, n2: Node): return True if there is any edge between the two given nodes

get_node_by_id(self, node_id): returns the Node element with specified id

set_bounding_box(self, bounding_box): set the bounding box

determine_node_positions(self): assigns each node a position for display

_determine_node_positions_grouped(self): called when there are groups used; nodes in a group are placed together, and groups are separated from each other

_determine_node_positions_nx(self): randomize all node locations; nodes are spaced to minimize edge intersections

_create_node_dictionary(self): create the node_dict after all nodes have been added

Figure 6: A standard graph; A graph using grouped nodes

Formula
The Formula Element can be used to represent a character based formula. It is currently designed to be used with 3SAT style formulas. Examples of using formulas can be found in the `formula_introduction.py` file.

Input File Format

The Formula class' default parser takes in a file name and creates a Formula. The input style must match the spelling and capitalization exactly.

```
(X1 OR X2 OR NOT X3) AND (X1 OR NOT X2 OR NOT X4) AND (NOT X1 OR NOT X2 OR X4)
```

Figure 7: sample .txt file input for a formula with 3 clauses; 4 variables will be automatically initialized, named X1, X2, X3, and X4

Attributes:

clauses: a list of Clause elements

literal_rects: a dict containing (clause_index, literal_index) : Rectangle, used for display

clause_rects: a dict containing clause_index : Rectangle, used for display.

font: font used to display text; default is Pygame Font

Methods:

load_formula_from_tuples(self, list_of_clause_tuples): optional method for loading in Formula from Python tuples instead of a file

get_as_list(self): return self.clauses

Figure 8: The formula display for a 3SAT style problem; colors of Clause bounding rectangles and each Variable can be changed

4.2 NP_Problem

The problem classes represent one type of NP Complete Problem. Each NP_Problem contains an Element and a solution. Each class must have a way to evaluate a solution.

Attributes:

element: an Element type

solution: a solution to the problem that can be evaluated

Methods:

display_solution(self): overlay the solution display on the problem
disable_solution(self): return to problem display

4.2.1 IndependentSetProblem

Given a graph, the Independent Set problem asks whether a specific set of nodes has no connected edges. Our implementation of this problem is based on a Graph element. This problem is based on a Graph element. More information can be found in `independent_set_introduction.py`

Attributes:

next_node_id: a counter to ensure unique Node id

Methods:

evaluate(self, node_ids): return True if the list of given Node ids form a valid independent set

set_solution(self, solution): select the Nodes corresponding to the given solution when solution is given as a list of Nodes

set_solution_by_id(self, solution): select the Nodes corresponding to the given solution when solution is given as a list of ids

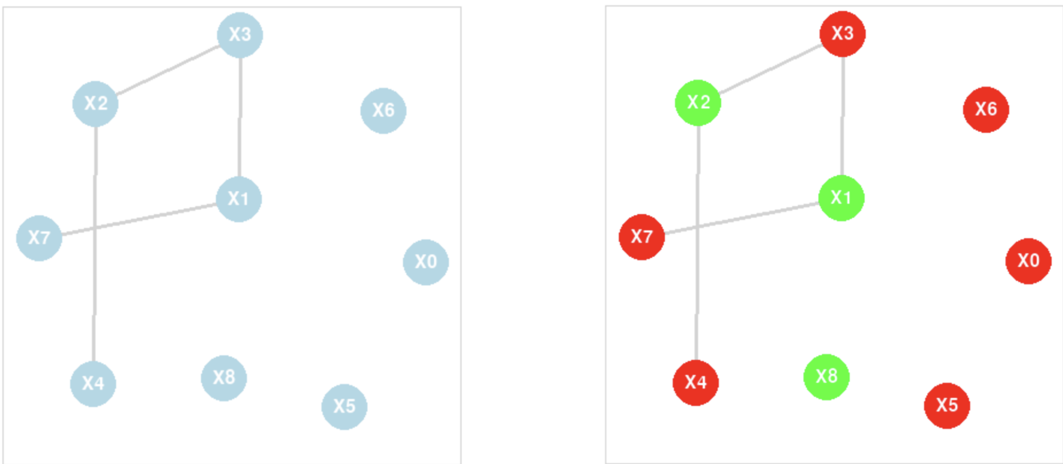


Figure 9: An Independent Set Problem displayed without and with solution mapping

4.2.2 ThreeSATProblem

The 3SAT problem asks whether a Boolean formula in conjunctive normal form, where each clause contains exactly 3 literals, can be satisfied by some assignment of true/false values to its variables. In our implementation, this problem is based on a Formula element.

Methods:

load_formula_from_file(self, filename: str): load a formula from a file

load_formula_from_tuples(self, list_of_clause_tuples): load a formula from a list of tuples

evaluate(self, assignment): Evaluates this 3-SAT formula given a variable assignment.

get_variables(self): Returns a set of all variable IDs in the formula.

get_as_list(self): Returns the formula as a list of lists of (var_id, is_negated).

set_solution(self, assignment): assign the variables True/False values according to the specified solution

4.2.3 ThreeColoringProblem

The 3-Coloring problem asks whether the vertices of a given graph can be colored using at most 3 colors such that no adjacent vertices share the same color. Our implementation of this problem is based on a Graph element.

Methods:

reset_coloring(self): reset all nodes to the default color

evaluate(self, assignment): Evaluates whether the current solution (a list of 3 node-sets) is a valid 3-coloring.

4.3 Reduction

Each Reduction takes in 2 different NP_Problems, it is up to the programmer to define the reduction process.

Attributes:

problem1: first NP Problem (reduce from)

problem2: second NP Problem (reduce to)

input1_to_input2_dict: dict that maps a set of elements/subelements from the first NP problem to a set of elements/subelements in the second NP problem. A single element can belong to multiple sets, as some reduction problems do not produce 1:1 mappings.

highlighted: list of highlighted items

Methods:

input1_to_input2(self): map the inputs of problem1 to inputs of problem2; must be overloaded by child

solution1_to_solution2(self): method to map solution of problem1 to a solution in problem2; must be overloaded by child

solution2_to_solution1(self): method to map solution of problem2 to a solution in problem1; must be overloaded by child

display_input_to_input(self, clicked_set): iterate through the input1_to_input2_dict to find which sets of related inputs should be selected. This method controls the opacity of the highlight, depending on how many relevant set items there are.

For more information, please read the 3SAT to Independent Set Annotated Guide.

4.3.1 GameManager

The GameManager is the class that manages the base pygame display functionality. It is also responsible for updating the display, processing events (clicks/key presses), and connecting displays across elements. It can be used to help manage interactive Reduction displays.

4.3.2 Creating a New Reduction

1. Identify the 2 required NP Problems, create classes for them if needed
2. Create any necessary Elements or SubElements if needed
   (a) Element: override display(), parse(), and handle_event()
   (b) SubElement: init parent constructor with id, name, color

3. Create a solution to the NP Problem and create a function to evaluate it
4. Create the reduction class, inheriting from `Reduction`
   (a) Implement the input1_to_input2, solution1_to_solution2, and solution2_to_solution1 methods
   (b) input1_to_input2() must update the input1_to_input2_dict; this is what controls the second problem's display to be highlighted when clicking parts of the first element.
5. To run your reduction, create a new Python file. Initialize the components of the reduction, and connect the GameManager to enable interactivity.

3–SAT → INDEPENDENT–SET Reading guide

Concrete Reduction

3–SAT → INDEPENDENT–SET

The high-level idea of the reduction is the standard one taught in any introductory complexity course:

1. For every clause $(\ell_1 \vee \ell_2 \vee \ell_3)$ create **three nodes**—one per *literal occurrence*—and **fully connect** them so that at most one can be picked into an independent set.
2. For every pair of complementary literals that appear in the formula (e.g. x and $\neg x$ in *different* clauses) connect the corresponding nodes, preventing a satisfying assignment from choosing both.
3. The number k for the INDEPENDENT-SET instance is simply the number of clauses; a satisfying assignment lets us pick exactly one literal per clause, hence k nodes in total.

This class provides *bidirectional* conversions so the visualiser can highlight how solutions correspond.

`__init__()`

Source code

```
             three_sat_problem: ThreeSATProblem,
             ind_set_problem: IndependentSetProblem,
             debug: bool = False):
        """Create a new reduction object.

        Parameters
        ----------
        three_sat_problem : ThreeSATProblem
           The *source* instance () in 3CNF we want to reduce.
        ind_set_problem : IndependentSetProblem
           The *target* graph G (initially empty  nodes/edges added later).
        debug : bool, optional
           If *True*, print verbose tracing information to *stdout*.
        """
```

```
        super().__init__(three_sat_problem, ind_set_problem)

        # -----------------------------------------------------------------
        # Forward / backward maps  they let the GUI jump between layers.
        # Each map stores **single objects** so we can colour a literal *x*
        # and immediately know which graph node to flash (and viceversa).
        # -----------------------------------------------------------------
        self.input1_to_input2_pairs = {} # SATliteral  graphnode
        self.input2_to_input1_pairs = {} # graphnode  SATliteral
        self.output1_to_output2_pairs = {} # sat solution  indset node
        self.output2_to_output1_pairs = {} # indset node  sat literal

        # Whether debug printing is enabled.
        self.DEBUG = debug\n
```

Walk-through.

- Stores references to the source (3-SAT) and target (Independent-Set) problem objects by calling the parent constructor.
- Initialises four dictionaries that map SAT literals and graph nodes in both directions (used later by the GUI and the two conversion routines).
- Caches the `debug` flag so that verbose tracing can be switched on/off globally.

`_debug_print()`

Source code

```
        """Emit *msg* prefixed with [DEBUG] iff self.DEBUG is *True*."""
        if self.DEBUG:
            print("[DEBUG]", msg)\n
```

Walk-through.

- If debugging is enabled, prints the supplied message with a distinctive `[DEBUG]` prefix.
- Otherwise does nothing — a zero-overhead logger in production runs.

`build_graph_from_formula()`

Source code

```
        """Populate *self.problem2* (an IndependentSetProblem) with nodes and
        edges so that **IndependentSet(G, k=len(clauses))** is equivalent to
        the original **3SAT()** instance.
        """
        # -- Grab the list of (clause, literals) objects --------------------
        self._debug_print("Starting build_graph_from_formula")
        formula_list = self.problem1.element.clauses
```

```python
        self._debug_print(f"Retrieved formula_list with {len(formula_list)}
            clause(s).")

        # Iterate over each clause *C* and perform steps (node creation &
        # intraclause clique).
        for c_idx, clause in enumerate(formula_list, start=1):
            # Pretty banner for human readers
            self._debug_print(f"Processing Clause #{c_idx} with {len(clause.
                variables)} variable(s).")

            clause_nodes = [] # Nodes we create for this clause
            clause_fs = set() # Literal objects for GUI crosshighlighting

            # ---- 1(a) Node creation ------------------------------------
            for literal in clause.variables:
                clause_fs.add(literal) # Remember for group mapping later

                # Create a *brand new* node in the target graph whose name is
                # the repr() of the literal (e.g. 'x', 'x').
                node = self.problem2.add_node(repr(literal))

                # Store bidirectional mapping for future conversions / UI.
                self.input1_to_input2_pairs[literal] = node
                self.input2_to_input1_pairs[node] = literal
                self.add_input1_to_input2_by_pair(literal, node) # Framework
                    helper

                # Trace what we just did
                self._debug_print(f" -- Added literal/node pair [{literal} : {
                    node}] to maps")
                self._debug_print(f" Created node '{node.id}' with label '{node.
                    name}' for literal {literal}.")

                # Remember the node so we can fullyconnect them momentarily.
                clause_nodes.append(node)

            # ---- 1(b) Tag nodes that belong to the same clause ----------
            # The visualiser can later colour them as a unit (triangle).
            self.problem2.add_group(clause_nodes)
            self._debug_print(f" Added group for Clause #{c_idx}: node IDs {[n.
                id for n in clause_nodes]}.")

            # ---- 1(c) Intraclause **clique** --------------------------
            # Connect every pair inside the clause so that only **one** can
            # be chosen in an independent set. Because each clause contains
            # exactly 3 literals (3CNF) we always create a triangle.
            for i in range(len(clause_nodes)):
                for j in range(i + 1, len(clause_nodes)):
                    self.problem2.add_edge(clause_nodes[i], clause_nodes[j])
```

```
            self._debug_print(f" Fully connected the nodes within Clause #{c_idx}.")

        # ----------------------------------------------------------------
        # 2. Interclause edges between *complementary* literals
        # (x vs x) so they cannot both be selected in the IS.
        # GoalFor every variable x we create an edge between *each* positive
        # occurrence (x) and *each* negative occurrence (x) that live
        # in **different** clauses. This guarantees the graph never lets
        # us pick both literalnodes for the same variable, because that
        # would violate the independentset property.
        #
        #  Miniexample
        # -----------------------------------------------------------------
        # Formula: (x  y  z)  (x  y  z)
        #
        # Literal list in one pass through `items` might look like:
        # i=0 : literal_A = x , node_A = v
        # i=1 : literal_A = y , node_A = v
        # i=2 : literal_A = z , node_A = v
        # i=3 : literal_A = x , node_A = v
        # i=4 : literal_A = y , node_A = v
        # i=5 : literal_A = z , node_A = v
        #
        #  Keys we create
        # 'x'  { x } ('x' positive bucket)
        # 'x_neg'  { x } ('x' negative bucket)
        #  and similarly for y / z
        #
        #  Edges added
        # v  v (x  x)
        # v  v (y  y)
        # v  v (z  z)
        # ----------------------------------------------------------------
        self._debug_print("Connecting complementary literal occurrences across clauses")

        # Because we need to crosscompare every literal against *later* ones
        # we copy the items into a list first (O(m) but m=3|clauses|, fine).
        # input1_to_input2_pairs : { literal_obj  node_obj }
        items = list(self.input1_to_input2_pairs.items())
        #
        # each element is (literal, node)


        # These helper dicts collect **all** positive / negative occurrences
        # so the GUI can colour or highlight them together later.
        name_literal_dict = {} # 'x'  {literal objects for x}, 'x_neg'  {x, }
        name_node_dict = {} # 'x'  {node objects for x}, 'x_neg'  {v, }
```

```
        for i in range(len(items)):
            literal_A, node_A = items[i]

            # Normalise the key so occurrences fall into exactly two buckets:
            # 'x'  positive literal x
            # 'x_neg'  negative literal x
            name = literal_A.name if not literal_A.is_negated else f"{literal_A.name}_neg"

            # Record this literal / node under its bucket for later GUI use
            name_literal_dict.setdefault(name, set()).add(literal_A)
            name_node_dict.setdefault(name, set()).add(node_A)

            # Compare with every *later* literal_B (j > i) so each pair is handled once
            for j in range(i + 1, len(items)):
                literal_B, node_B = items[j]

                # SAME variable label && OPPOSITE polarity?  connect!

                # Example  Complementary literals  condition is **True**
                # --------------------------------------------------------------
                # literal_A = x2 (name='x2', is_negated=False)
                # literal_B = x2 (name='x2', is_negated=True)
                #
                # literal_A.name == literal_B.name  'x2' == 'x2'
                # literal_A.is_negated != literal_B.is_negated  False != True
                # Both tests pass, so we add an edge between the two nodes.
                if literal_A.name == literal_B.name and literal_A.is_negated != literal_B.is_negated:
                    self.problem2.add_edge(node_A, node_B)
                    self._debug_print(
                        f" Connected complementary literals '{literal_A}'  '{literal_B}' "
                        f"via nodes {node_A.id} and {node_B.id}.")

        # The gathered *samesign* buckets are now inserted into helper maps
        # so the visualiser can flash *all* positive occurrences of x together.
        for name, literals in name_literal_dict.items():
            # (Framework call omitted  uncomment if the UI expects it)
            self._debug_print(
                f" -- Added same_name_literals/same_name_nodes pair "
                f"[{literals} : {name_node_dict[name]}] to maps")

        self._debug_print("Finished build_graph_from_formula.\n")\n
```

Walk-through.

- Fetches the list of clauses from the 3-SAT instance.
- For **each clause**:
- *(a) Node creation* — makes three graph nodes, one per literal occurrence, and records the literal ↔ node mapping.
- *(b) Group tagging* — stores the three nodes as a GUI group so the visualiser can draw a triangle around them.
- *(c) Intra-clause edges* — fully connects the three nodes to enforce "pick at most one".
- After all clauses are handled, scans every pair of *complementary* literals (x vs $\neg x$) that lie in *different* clauses and links their nodes, preventing the solution from choosing both.

solution1_to_solution2()

Source code

```
        """Given a satisfying *sat_assignment* (dict varbool) return the set
        of graph nodes that constitutes the corresponding **independent set**.

        Notes
        -----
         Exactly one *satisfied* literal is picked per clause (triangle).
         Because complementary literals are connected, the set is indeed
          independent as long as the assignment satisfies the formula.
        """
        self._debug_print("Starting sol1tosol2 (SAT  IS) conversion")

        independent_set = set() # The resulting node set
        formula_list = self.problem1.element.clauses

        # Iterate clausebyclause to choose *one* node per satisfied clause.
        for clause_idx, clause in enumerate(formula_list, start=1):
           chosen_node = None # Reset for this clause
           self._debug_print(f" Evaluating Clause #{clause_idx}")

           for literal in clause.variables:
              # Get the node info corresponding to this literal
              node = self.input1_to_input2_pairs[literal]
              var_id = literal.name
              is_negated = literal.is_negated
              assigned_val = sat_assignment[var_id]

              self._debug_print(
                 f" Checking literal {literal}: assignment[{var_id}]={
                     assigned_val}, "
                 f"is_negated={is_negated}")

              # A literal is satisfied (true) when its sign matches the
                  assignment.
```

```
            #  Positive literal x  true if assigned_val == True
            #  Negated literal x  true if assigned_val == False
            # Hence we include the node exactly when
            # assigned_val != is_negated
            # where `is_negated` is True for x and False for x.
            if assigned_val != is_negated:
               chosen_node = node

               # Store for reverse lookup when we later highlight answers.
               self.output1_to_output2_pairs[literal] = node

               self._debug_print(
                  f"  Literal {literal} is satisfied; picking node {node.id}.")
               break # Only need *one* per clause

         # After scanning the three literals:
         if chosen_node:
            independent_set.add(chosen_node)
         else:
            # Should not happen if *sat_assignment* truly satisfies
            self._debug_print(f" No satisfied literal found in Clause #{clause_idx}.")

      self._debug_print(f"Constructed Independent Set: {[n.id for n in independent_set]}\n")
      self._debug_print("Finished sol1tosol2.\n")
      return independent_set\n
```

Walk-through.

- Iterates through every clause and selects the first literal that is satisfied by the provided assignment σ.
- The corresponding node is inserted into the independent set and cached for reverse look-up.
- Because complementary literals are adjacent in the graph, the resulting set is guaranteed to be independent whenever σ satisfies the formula.

solution2_to_solution1()

Source code

```
      """Recover a concrete truth assignment for the original 3SAT instance
      from the *independent_set* returned by the graph solver.
      """
      self._debug_print("Starting sol2tosol1 (IS  SAT) conversion\n")

      sat_assignment = {}
      formula_list = self.problem1.element.clauses
```

```

        # ---- 4(a) Positive information: variables forced by selected nodes
        self._debug_print("Assigning variables for selected nodes in the Independent Set.")
        for literal, node in self.input1_to_input2_pairs.items():
            # We only care about nodes that survived in the solvers answer.
            if node in independent_set:
                var = literal.name # e.g. "x3"
                is_negated = literal.is_negated # True for x3, False for x3

                # Keep a *reverse* lookup for GUI / animation layers:
                # graphnode  corresponding literal occurrence
                self.output2_to_output1_pairs[node] = literal

                sat_assignment.setdefault(var, not is_negated)

                # Verbose debug trace
                self._debug_print(
                    f" Selected node {node.id}  literal {literal}; "
                    f"setting {var} = {not is_negated}")

        # ---- 4(b) Default remaining variables to *False* so assignment is total
        all_vars = {lit.name for clause in formula_list for lit in clause.variables}
        self._debug_print("\nEnsuring all variables are assigned (default = False).")
        for var in sorted(all_vars):
            if var not in sat_assignment:
                sat_assignment[var] = False
                self._debug_print(f" {var} absent from IS; defaulting {var}=False ")

        self._debug_print(f"\nFinal recovered SAT Assignment: {sat_assignment}")
        self._debug_print("Finished sol2tosol1.\n")
        return sat_assignment\n
```

Walk-through.

- Starts with an empty assignment dictionary.
- For each node that appears in the independent set, retrieves its literal and fixes the underlying variable:
- positive literal $x \;\Rightarrow\;$ set $x = \text{True}$
- negative literal $\neg x \;\Rightarrow\;$ set $x = \text{False}$
- Uses `setdefault` to keep the first value when a variable appears multiple times (the reduction guarantees no contradictions).

- Any variable that was never forced defaults to `False` so the assignment is total.

`test_solution()`

Source code

```
        """Verify a *sat_assignment* by checking **both** sides:
        1. Does the assignment satisfy the original 3CNF ?
        2. Is the image under the reduction an \emph{actual} independent set?
        Returns
        -------
        (bool, bool)
            satisfied  whether (sat_assignment) = True
            valid_independent  whether chosen set is independent in G
        """
        self._debug_print("Starting test_solution")

        # Step 1: formula evaluation
        satisfied = self.problem1.evaluate(sat_assignment)
        self._debug_print(f" Formula satisfied? {satisfied}")

        # Step 2: graph evaluation
        chosen_set = self.solution1_to_solution2(sat_assignment)
        valid_independent = self.problem2.evaluate(chosen_set)
        self._debug_print(f" Independent set valid? {valid_independent}")
        self._debug_print("Finished test_solution.\n")

        return satisfied, valid_independent\n
```

Walk-through.

- Checks the candidate assignment directly against the 3-SAT formula (step 1).
- Feeds the assignment through `solution1_to_solution2` and asks the graph object to verify independence (step 2).
- Returns a pair of booleans (formula_ok, IS_ok).