دانشگاه صنعتی امیرکبیر

(پلی‌تکنیک تهران)

دانشکده ریاضی و علوم کامپیوتر

گزارش کد منطق

پیاده سازی $SAT\ Solver$

نگارش

محمدامین سیفی

استاد اول

دکتر مسعود پورمهدیان

تابستان ۱۴۰۳

چکیده

این پروژه یک حل‌کننده SAT (تست ارضاپذیری) پیاده‌سازی شده در پایتون را توسعه می‌دهد که برای رسیدگی به چالش‌های ارزیابی رضایت‌پذیری فرمول‌های منطقی گزاره‌ای طراحی شده است. این حل‌کننده استراتژی‌های الگوریتمی پیچیده‌ای را در چارچوب کلاسی ترکیب می‌کند که قابلیت تجزیه، تبدیل، ارزیابی و تعیین رضایت‌پذیری عبارات منطقی پیچیده را که به‌صورت رشته‌ها نشان داده می‌شوند، می‌کند.

واژه‌های کلیدی:

SAT، $LinearSATSolver$

فهرست مطالب

فصل اول

مقدمه

۱-۱ بررسی الگوریتم

الگوریتم Linear SAT Solver یک الگوریتم پیدا کردن مدل برای یک گزاره است. همانطور که از اسم آن پیداست این الگوریتم در زمان خطی عمل می‌کند اما ممکن است نتواند جواب آن را بیابد. در ادامه به یک ایده توسط من پرداخته می‌شود که احتمال پیدا شدن جواب را توسط این الگوریتم بالا می‌برد.

این الگوریتم برای اجرا بر روی درخت تجزیه فرمول های منطقی، به طور خاص بحث در مورد تبدیل فرمول ها به یک قطعه مناسب برای ایجاد یک گراف غیر چرخشی جهت دار (DAG) استفاده می شود. بدین ترتیب به مراحل الگوریتم اشاره‌ای می‌کنیم.

۱-۱-۱ تبدیل

فرمول ها در ابتدا به یک قالب خاص ترجمه می شوند، که ساختار فرمول را ساده می کند و آن را قابل کنترل تر می کند. قوانین تبدیل تعریف شده (T) ساختارهای منطقی را به اشکالی تبدیل می کند که فقط شامل نفی و ربط هستند (NAND). به عنوان مثال، تفکیک $(\phi_1 \vee \phi_2)$ و $(\phi_1 \implies \phi_2)$ با استفاده از نفی و ربط برای حفظ هم ارزی معنایی در حالی که ساختار را ساده می کنند، تبدیل می شوند. این تبدیل تضمین می کند که اگر فرمول اصلی ϕ ارضاپذیر باشد، فرمول تبدیل شده $T(\phi)$ نیز ارضاپذیر است و بالعکس. شکل زیر تابع تبدیل را نشان می‌دهد.

$$T(p) = p \qquad T(\neg\phi) = \neg T(\phi)$$
$$T(\phi_1 \wedge \phi_2) = T(\phi_1) \wedge T(\phi_2) \qquad T(\phi_1 \vee \phi_2) = \neg(\neg T(\phi_1) \wedge \neg T(\phi_2))$$
$$T(\phi_1 \to \phi_2) = \neg(T(\phi_1) \wedge \neg T(\phi_2))$$

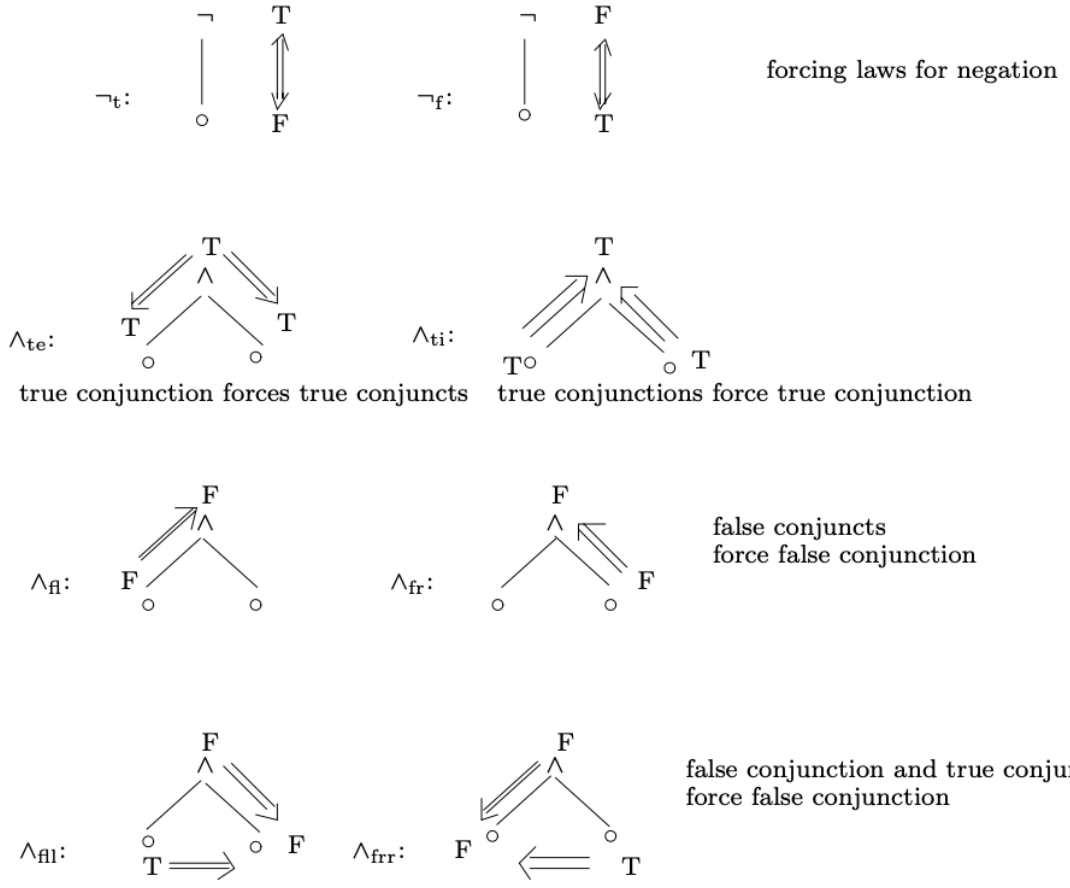
۱-۱-۲ ساخت گراف

پس از تبدیل شدن، ساب‌فرمول های رایج در درخت تجزیه به اشتراک گذاشته می شوند و به طور موثر درخت تجزیه را به یک DAG تبدیل می کنند. این اشتراک‌گذاری به بهینه‌سازی فرآیند ارزیابی با اجتناب از ارزیابی‌های اضافی از عبارات فرعی مشابه کمک می‌کند.

۱-۱-۳ اجرای الگوریتم ارزیابی

الگوریتم ارزیابی با این فرض عمل می کند که فرمول سطح بالا درست است (TOP DOWN) و مقادیر حقیقی را به صورت بازگشتی DAG منتشر می کند. ارزیابی شامل تعیین علائم بر روی گره ها بر اساس عملیات منطقی و مقادیر حقیقی منتشر شده است. برای مثال، اگر یک گره AND (∧) درست فرض شود،

هر دو گره فرزند آن نیز باید درست علامت گذاری شوند. تصویر زیر به Forcing Ruleها اشاره می‌کند که برای اجرای خطی الگوریتم به کار می‌روند.



فصل دوم

بررسی کد

۲-۱ کلاس *Node*

```python
class Node:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children if children else []
        self.id = None

    def evaluate(self, truth_assignment):
        if self.value.isalnum():  # Check if the node is a propositional variable
            return truth_assignment.get(self.value, False)
        elif self.value == '¬':  # Unary negation operator
            return not self.children[0].evaluate(truth_assignment)
        elif self.value == '∧':  # Logical AND operator
            return self.children[0].evaluate(truth_assignment) and
self.children[1].evaluate(truth_assignment)
        elif self.value == '∨':  # Logical OR operator
            return self.children[0].evaluate(truth_assignment) or
self.children[1].evaluate(truth_assignment)
        elif self.value == '→':  # Logical implication operator
            return not self.children[0].evaluate(truth_assignment) or
self.children[1].evaluate(truth_assignment)

    def __str__(self):
        if self.children:
            return f"{self.value}({', '.join(map(str, self.children))})"
        return f"{self.value}"
```

کلاس Node یک جزء اصلی ارزیابی کننده منطقی است که برای کپسوله کردن هر عنصر از فرمول منطقی مجازی در درخت تجزیه طراحی شده است. این کلاس از عملیات منطقی یکپارچه و باینری، علاوه بر مدیریت متغیرهای پیشنهادی، پشتیبانی می کند و هم‌چنین ساخت، پردازش و ارزیابی عبارات پیچیده منطقی را تسهیل می کند.

۲-۱-۱ تابع ارزیابی

evaluate(truth_assignment): این روش به صورت بازگشتی عبارت منطقی نشان داده شده توسط زیردرختی که در این گره ریشه دارد را ارزیابی می کند. ارزیابی بر اساس یک تخصیص مقدار است - یک دیکشنری که نام متغیرها را به ۰ یا ۱ مپ می کند - که مقادیر متغیرهای مجازی را ارائه می دهد. این روش عملیات منطقی مختلفی را مطابق با تعاریف استاندارد آنها انجام می دهد.

۲-۲ کلاس SAT

```python
class SAT:

    def __init__(self, expression) -> None:
        self.expression = expression.replace('¬¬', '')
        self.atoms = re.findall(r'\w', expression)
        self.parse_tree = None
        self.transformed_tree = None
        self.last_id = 1
        self.ids = {}
```

کلاس SAT برای مدیریت تجزیه و تحلیل، تبدیل، ارزیابی و حل فرمول های منطقی پیشنهادی طراحی شده است که طیف وسیعی از تکنیک های محاسباتی برای پردازش عبارات منطقی پیچیده برای تعیین ارضاپذیر بودن آنها استفاده می کند.

۱-۲-۲ تابع *parse*

```python
    def parse(self):
        # Regular expression to match tokens: digits, words, parentheses, and operators
        token_pattern = r'\s*(?:(\d+|\w+|[()¬∧∨→]))\s*'
        tokens = re.findall(token_pattern, self.expression)
        # Stack for holding nodes that are being processed
        stack = []
        # Operator stack for handling operator precedence and parentheses
        ops = []

        def operator_precedence(op):
            """
                Returns the precedence of the logical operators, with higher values indicating higher
precedence.

                Args:
                    op (str): A string representing one of the logical operators.

                Returns:
                    int: The precedence of the operator.
            """
            precedences = {'¬': 3, '∧': 2, '∨': 2, '→': 1}
            return precedences.get(op, -1)

        def process_operator(op):
            """
                Processes operators by popping the required number of operands from the stack,
                applying the operator, and pushing the result back onto the stack.

                Args:
                    op (str): The operator to process.
            """
            if op == '¬':
                arg = stack.pop()
                stack.append(Node('¬', [arg]))
            elif op in {'∧', '∨', '→'}:
                right = stack.pop()
                left = stack.pop()
                stack.append(Node(op, [left, right]))

        # Process each token in the expression
        for token in tokens:
            if token.isalnum():  # Alphanumeric tokens are atoms
                stack.append(Node(token))
            elif token in {'¬', '∧', '∨', '→'}:
                # Handle operators with respect to precedence
                while (ops and ops[-1] != '(' and
                    operator_precedence(ops[-1]) >= operator_precedence(token)):
                    process_operator(ops.pop())
                ops.append(token)
            elif token == '(':
                # Opening parenthesis, push to stack to denote precedence change
                ops.append(token)
            elif token == ')':
                # Closing parenthesis, process all operators until the opening parenthesis
                while ops and ops[-1] != '(':
                    process_operator(ops.pop())
                ops.pop() # Remove the '(' from the stack

        # Process any remaining operators in the operator stack
        while ops:
            process_operator(ops.pop())

        self.parse_tree = stack[-1]
        return stack[-1]
```

این تابع ابتدا از یک عبارت منظم برای توکنایز کردن رشته ورودی استفاده می کند. این regex با

عملگرهای منطقی، پرانتزها و اتم‌ها (که نشان دهنده متغیرهای گزاره ای هستند) مطابقت دارد. سپس دو پشته مقداردهی اولیه می شوند

پشته: این اشیاء Node را نگه می دارد که به تدریج درخت تجزیه را ایجاد می کنند.

ops: این پشته به طور موقت عملگرها و پرانتزها را نگه می دارد تا اطمینان حاصل شود که عملگرها به ترتیب اولویت درست پردازش می شوند.

هنگامی که یک متغیر گزاره ای یا عملوند خوانده می شود (با کاراکترهای الفبایی مشخص می شود)، یک گره جدید ایجاد می شود و در پشته پوش می شود.

اگر توکن یک عملگر باشد، تابع اولویت خود را با عملگر در بالای پشته عملیات مقایسه می کند. عملگرهایی که دارای اولویت بالاتر یا مساوی از قبل در پشته عملیات هستند، ظاهر می شوند و پردازش می شوند، و اطمینان حاصل می شود که عبارات با ترتیب صحیح عملیات ساخته شده اند. سپس عملگر فعلی به پشته پوش می شود. پرانتزهای باز کردن مستقیماً در پشته ops پوش می‌شوند، در حالی که بسته شدن پرانتز باعث می‌شود عملگرها ظاهر شوند و پردازش شوند تا زمانی که یک پرانتز باز شود.

و در انتها پس از خواندن همه نشانه ها، هر عملگر باقی مانده در پشته عملیات پردازش می شود. این مرحله تضمین می‌کند که تمام بخش‌های عبارت به درستی در درخت تجزیه ساخته شده‌اند.

یک مرحله‌ی بهینه‌سازی حذف دو نفی پشت سر هم است زیرا می‌دانیم دو نفی پشت سر هم هم‌دیگر را خنثی می‌کند، بنابراین همان موقع پارس شدن حذف می‌شوند.

۲-۲-۲ تابع $transform$

```python
def _transform(self, node, ):
    # Handle atomic propositions directly without transformation
    if node.value in self.atoms:
        return Node(node.value, [])
    # Simplify negations by recursively transforming the child node
    elif node.value == '¬':
        subnode = self._transform(node.children[0])
        return Node('¬', [subnode])
    # Simplify conjunctions by recursively transforming each child
    elif node.value == '∧':
        left = self._transform(node.children[0])
        right = self._transform(node.children[1])
        return Node('∧', [left, right])
    # Convert disjunctions to negations of conjunctions (De Morgan's Law)
    elif node.value == '∨':
        left = self._transform(node.children[0])
        right = self._transform(node.children[1])
        return Node('¬', [Node('∧', [Node('¬', [left]), Node('¬', [right])])])
    # Convert implications to disjunctions using logical equivalences
    elif node.value == '→':
        left = self._transform(node.children[0])
        right = self._transform(node.children[1])
        return Node('¬', [Node('∧', [left, Node('¬', [right])])])
    else:
        return node  # Base case for other atomic propositions
```

تابع `_transform` در کلاس SAT برای استانداردسازی ساختار یک درخت تجزیه با اعمال معادل‌های منطقی برای ساده‌سازی ارزیابی فرمول‌های منطقی گزاره‌ای طراحی شده است. این تابع درخت تجزیه را به گونه‌ای تغییر می‌دهد که از مجموعه محدودتری از عملیات منطقی استفاده می‌کند، که در درجه اول بر نفی‌ها و ربط‌ها تمرکز دارد، که می‌تواند پردازش بیشتر را یکنواخت‌تر و به طور افیشنت کارآمدتر کند.

۲-۲-۳ تابع *propagate*

```python
def propagate_truth(self, node, assumed_true=True):
    """
        Recursively propagates truth values through the parse tree based on the assumption that
        the formula is true. This method assigns truth values to propositional variables and
        applies logical operations accordingly.

        Args:
            node (Node): The current node in the parse tree to process.
            assumed_true (bool): Indicates whether the current branch of the parse tree is
                                 assumed to be true or false based on logical deductions.

        Note:
            This function operates in a linear time complexity with respect to the number of nodes,
            as it traverses each node exactly once. However, it might not find a solution if the
            formula is unsatisfiable or if contradictions are encountered during propagation.
    """
    if self.falsity:
        return

    if node.value.isalnum() and self.ids.get(node.value):
        node.id = self.ids.get(node.value)
    elif node.value.isalnum():
        self.ids[node.value] = self.last_id
        node.id = self.last_id
        self.last_id += 1
    else:
        node.id = self.last_id
        self.last_id += 1

    if node.value == '∧' and assumed_true:
        # If AND and assumed true, both sides must be true
        for child in node.children:
            self.propagate_truth(child, True)
    elif node.value == '¬':
        # If NOT, propagate the opposite of the current assumption
        self.propagate_truth(node.children[0], not assumed_true)
    elif node.value in self.atoms:  # Base case for variables
        # Mark the variable based on the propagated truth value
        if self.assignments.get(node.value) and self.assignments.get(node.value) != assumed_true:
            self.falsity = True
            return
        self.assignments[node.value] = assumed_true
```

تابع `propagate_truth` در کلاس SAT برای انتشار بازگشتی مقادیر حقیقت از طریق درخت تجزیه طراحی شده است و به ساختار منطقی فرمولی که نشان می‌دهد پایبند است. این تابع نقش مهمی در تعیین رضایت‌پذیری فرمول‌های منطق گزاره‌ای با تلاش برای تخصیص مقادیر صدق منسجم در سراسر درخت بر اساس یک فرض اولیه که فرمول سطح بالا درست است، ایفا می‌کند. این تابع بدین صورت عمل می‌کند که در ابتدا وقتی از درخت تجزیه عبور می‌کند، به هر گره یک شناسه منحصربه‌فرد جدید اختصاص داده می‌شود یا اگر گره یک گزاره اتمی مکرر را نشان دهد، به شناسه موجود نگاشت می‌شود. این به مدیریت هویت گره ها در طول فرآیند استقرار بازگشتی کمک می کند. سپس قرارداد ارتباط (∧): اگر گره یک AND منطقی را نشان می‌دهد و درست فرض می‌شود، تابع به صورت بازگشتی بیان می‌کند

که هر دو فرزند ربط نیز باید درست باشند، که منعکس کننده ماهیت عملیات AND است که در آن هر دو عملوند باید درست باشند. گره های نفی: برای نفی، تابع خلاف فرض حقیقت فعلی را به گره فرزند منتشر می کند. این مرحله شامل فرآیند نفی منطقی است که در آن مقدار صدق عملوند معکوس می شود. گزاره‌های اتمی: وقتی به یک گره که یک متغیر مجازی را نشان می‌دهد (گزاره اتمی) دسترسی پیدا می‌شود، تابع بررسی می‌کند که آیا مقدار گزاره فعلی با هر تخصیص قبلی در تضاد است یا خیر. اگر تضاد تشخیص داده شود (مقدار صدق متفاوتی قبلاً به متغیر اختصاص داده شده است)، فلگ خطا روی $true$ تنظیم می شود که نشان دهنده ارضانپذیر بودن است.

این قسمت یکی از قسمت‌هایی است که بهینه‌سازی در آن انجام شده زیرا با بررسی کردن مقدار قبلی اساین شده به متغیر و در صورت مغایرت آن دیگر به ارزیابی ادامه نمی‌دهد و از فضای جستجو می‌کاهد.

۴-۲-۲ تابع $solve$

```
    def solve(self):
        self.assignments = {}
        self.falsity = False
        self.propagate_truth(self.transformed_tree, True)
        if self.falsity:
            return False  # Return False if a contradiction is detected indicating unsatisfiability.
        if self.transformed_tree.evaluate(self.assignments) == True:
            return self.assignments  # Return assignments if the formula evaluates to True.

        # This section demonstrates the novelity of this work, taking the log2 of count of atoms, then
add a constant like 5 to it and make guesses randomly
        for _ in range(int(log2(len(self.atoms))) + 5):
            new_assignment = self._generate_non_assigned_atoms(self.assignments)
            if self.transformed_tree.evaluate(new_assignment) == True:
                return new_assignment  # Return assignments if the formula evaluates to True.
        return None  # Return None if the formula's truth cannot be determined.
```

تابع سالو، فرآیند تعیین اینکه آیا یک مدل وجود دارد که عبارت منطقی داده شده را برآورده کند، هماهنگ می کند. در صورت لزوم، سپس به یک رویکرد اکتشافی جدید برای یافتن یک مدل می‌پردازد. وجه تمایز این پروژه با سایر پروژه‌ها این است که اگر این الگوریتم موفق به پیدا کردن جواب نشد، سپس به تعداد لگاریتم تعداد اتم‌ها بر مبنای ۲ به علاوه‌ی یک عدد ثابت به اساین کردن مقادیر مختلف به اتم‌هایی که به طور دترمینیستیک مقداردهی نشده‌اند می‌پردازد. مشاهده می‌شود تست‌کیس‌هایی که

نیز توسط این الگوریتم به روش سنتی قابل حل نیستند، با روشی که من پیشنهاد کرده‌ام حل می‌شوند.

فصل سوم

نتایج

۳-۱ تست کیس ها

۳-۱-۱ تست کیس ۱

```
----- Test 1 -----
Original formula: p ∧ ¬(q ∨ ¬p)
Parsed tree: ∧(p, ¬(∨(q, ¬(p))))
Translated formula: ∧(p, ¬(¬(∧(¬(q), ¬(¬(p))))))
Satisfying assignments where the formula is true:
{'p': True, 'q': False}
--- Parse Tree ---
└── ∧
    ├── p
    └── ¬
        └── ¬
            └── ∧
                ├── ¬
                │   └── q
                └── ¬
                    └── ¬
                        └── p
-- DAG ---
└── Node 1: ∧
    ├── Node 2: p
    └── Node 3: ¬
        └── Node 4: ¬
            └── Node 5: ∧
                ├── Node 6: ¬
                │   └── Node 7: q
                └── Node 8: ¬
                    └── Node 9: ¬
                        └── Node 2: p (shared)
```

۳-۱-۲ تست کیس ۲

```
----- Test 2 -----
Original formula: p ∧ ¬(q → ¬p)
Parsed tree: ∧(p, ¬(→(q, ¬(p))))
Translated formula: ∧(p, ¬(¬(∧(q, ¬(¬(p))))))
Satisfying assignments where the formula is true:
{'p': True, 'q': True}
--- Parse Tree ---
└── ∧
    ├── p
    └── ¬
        └── ¬
            └── ∧
                ├── q
                └── ¬
                    └── ¬
                        └── p
-- DAG ---
└── Node 1: ∧
    ├── Node 2: p
    └── Node 3: ¬
        └── Node 4: ¬
            └── Node 5: ∧
                ├── Node 6: q
                └── Node 7: ¬
                    └── Node 8: ¬
                        └── Node 2: p (shared)
```

۳-۱-۳ تست کیس ۳

```
----- Test 3 -----
Original formula: p ∧ ¬¬(¬q ∧ ¬¬p)
Parsed tree: ∧(p, ∧(¬(q), p))
Translated formula: ∧(p, ∧(¬(q), p))
Satisfying assignments where the formula is true:
{'p': True, 'q': False}
--- Parse Tree ---
└── ∧
    ├── p
    └── ∧
        ├── ¬
        │   └── q
        └── p
-- DAG ---
└── Node 1: ∧
    ├── Node 2: p
    └── Node 3: ∧
        ├── Node 4: ¬
        │   └── Node 5: q
        └── Node 2: p (shared)
```

۳-۱-۴ تست کیس ۴

```
----- Test 4 -----
Original formula: p ∧ ¬p
Parsed tree: ∧(p, ¬(p))
Translated formula: ∧(p, ¬(p))
There are no satisfying assignments; the formula is always false.
--- Parse Tree ---
└── ∧
    ├── p
    └── ¬
        └── p
-- DAG ---
└── Node 1: ∧
    ├── Node 2: p
    └── Node 3: ¬
        └── Node 2: p (shared)
```

۳-۱-۵ تست کیس ۵

```
----- Test 5 -----
Original formula: ¬p ∧ (q ∨ r)
Parsed tree: ∧(¬(p), ∨(q, r))
Translated formula: ∧(¬(p), ¬(∧(¬(q), ¬(r))))
Satisfying assignments where the formula is true:
{'p': False, 'q': True, 'r': True}
--- Parse Tree ---
└── ∧
    ├── ¬
    │   └── p
    └── ¬
        └── ∧
            ├── ¬
            │   └── q
            └── ¬
                └── r
-- DAG ---
└── Node 1: ∧
    ├── Node 2: ¬
    │   └── Node 3: p
    └── Node 4: ¬
        └── Node 5: ∧
            ├── Node None: ¬
            │   └── Node None: q (shared)
            └── Node None: ¬ (shared)
```

۳-۱-۶ تست کیس ۶

```
----- Test 6 -----
Original formula: ¬p ∧ (q ∨ r) ∧ (s ∧ r)
Parsed tree: ∧(∧(¬(p), ∨(q, r)), ∧(s, r))
Translated formula: ∧(∧(¬(p), ¬(∧(¬(q), ¬(r)))), ∧(s, r))
Satisfying assignments where the formula is true:
{'p': False, 's': True, 'r': True}
--- Parse Tree ---
└── ∧
    ├── ∧
    │   ├── ¬
    │   │   └── p
    │   └── ¬
    │       └── ∧
    │           ├── ¬
    │           │   └── q
    │           └── ¬
    │               └── r
    └── ∧
        ├── s
        └── r
-- DAG ---
└── Node 1: ∧
    ├── Node 2: ∧
    │   ├── Node 3: ¬
    │   │   └── Node 4: p
    │   └── Node 5: ¬
    │       └── Node 6: ∧
    │           ├── Node None: ¬
    │           │   └── Node None: q (shared)
    │           └── Node None: ¬ (shared)
    └── Node 7: ∧
        ├── Node 8: s
        └── Node 9: r
```

توجه شود علت وجود $None$ در گراف به علت بهینه‌سازی‌هایی است که موقع پروپگیت کردن مقادیر ارزیابی صورت گرفته است.