Implementacja algorytmu dla równań słów

Kacper Solecki

1 Wstęp?

2 Opis algorytmu

2.1 Ogólny zarys

Algorytm jako wejście przyjmuje równanie słów, w którym litery są reprezentowane przez małe litery angielskiego alfabetu, a zmienne przez wielkie litery. Rozwiązaniem takiego równania, o ile istnieje jest podstawienie (ciąg liter) za każdą zmienną występującą w równaniu, takie, że po zamianie zmiennych na ich podstawienia dwie strony równania rzeczywiście są równe jako słowa. W języku gramatyk bezkontekstowych litery to symbole terminalne, zmienne – nieterminalne.
Przykładowo jednym z rozwiązań równania

$$abYX = XYba$$

jest

$$\begin{cases} X = aba \\ Y = a \end{cases}$$

Algorytm przeszukuje wszerz graf, którego wierzchołkami są równania słów. W każdym wierzchołku wykonywane są dwa rodzaje transformacji, które przekształcają równianie w inne. W języku grafów tłumaczy się to na istnienie dwóch rodzajów krawędzi z każdego wierzchołka:

krawędzie łączenia maksymalnych bloków

Przez maksymalny blok litery a w równaniu r będziemy rozumieć nierozszerzalny, spójny ciąg litery a w r. Nierozszerzalny znaczy taki, że z każdej jego strony jest brzeg równania lub inna litera lub zmienna i jeżeli jest to zmienna to jej podstawianie nie ma litery a ze strony tego bloku (tzn. nie zaczyna lub nie kończy się na a).

Taka krawędź jest pomiędzy wierzchołkami u i v, jeżeli z równanie u można przetransformować w równanie v łącząc wszystkie maksymalne bloki jakiejś litery występującej w u. Przez złączenie bloku rozumiemy zastąpienie go nową literą, niewystępującą w u. Oczywiście bloki tej samej długości powinny zostać złączone w tę samą literę, a różnej długości – w różne litery. Różne podstawiania za zmienne mogą implikować różne równości pomiędzy blokami, zatem trzeba będzie rozpatrzeć wszystkie przypadki.

krawędzie łączenia par liter

Innym rodzajem transformacji równania u jest wybranie z niego dwóch liter a i b, a następnie zastąpienie każdego wystąpienia pary ab poprzez nową literę spoza u.

Przy czym nie chcemy, by jedna litera z pary ab mogła znaleźć się w podstawieniu za jakąś zmienną, a druga nie, tzn. jeżeli X jest zmienną, której podstawianie zaczyna się literą b, nie chcemy skracać pary ab jeśli w u jest fragment aX (analogicznie jeśli podstawianie X kończy się na a, nie chcemy skracać ab, jeśli w u jest fragment Xb). W przeciwnym przypadku w dalszym przeszukiwaniu grafu stracilibyśmy informację, że litera do której złączyliśmy ab jest rzeczywiście równa tej parze liter. Aby móc skrócić taką parę ab, będziemy musieli dla każdej zmiennej rozważyć przypadki czy jej podstawianie zaczyna się na b i czy kończy się na a.

Przeszukiwanie grafu kończy się, gdy dojdziemy do wierzchołka, który reprezentuje równanie, które w oczywisty sposób jest spełnione np. $a = a$. Może też być tak, że po przeszukaniu całego grafu nie znajdziemy rozwiązania - znaczy to, że żadne nie istnieje.
Algorytm zaimplementowany jest w języku Python, jego fragmenty będą się pojawiały w dalszej części pracy.

2.2 reprezentacja słów

Słowo, czyli ciąg liter i zmiennych jest reprezentowane jako lista następujących obiektów:

```
class Variable(object):
    def __init__(self, nr: int) -> None:
        self.nr = nr

    def __repr__(self) -> str:
        return f'var_{self.nr}'

    def __eq__(self, other) -> bool:
        return type(self) == type(other) and self.nr == other.nr

    def __hash__(self):
        return hash(self.nr)


class Letter(object):
    def __init__(self, nr: int, cnt: int) -> None:
        self.nr = nr
        self.cnt = cnt

    def __repr__(self) -> str:
        return f'{self.cnt}({self.nr})'

    def __eq__(self, other) -> bool:
        return type(self) == type(other) and self.nr == other.nr

    def __hash__(self) -> int:
```

```
        return hash((self.nr, self.cnt))
```

Zmienne posiadają tylko jeden atrybut – numer, który je indetyfikuje. Litery oprócz indetyfikującego numeru posiadają atrybut cnt, który służy krótszemu zapisowi wielokrotnego, spójnego wystąpienia danej litery. Np. zakładając, że litera a ma numer 97 słowo aaa będzie skrótowo zapisane, jako pojedynczy obiekt Letter o $nr = 97$ i $cnt = 3$.

2.3 reprezentacja podstawień

W trakcie działania algorytmu w każdym wierzchołku musi on pamiętać wszystkie aktualne podstawiania za zmienne. Oprócz tego, poniważ przechodząc każdą krawędzią w grafie tworzymy nowe litery, zatem musi też pamiętać ich rozwinięcia, tzn. jakim słowom złożonym z literek wyjściowego równania odpowiadają litery obecne w danym wierzchołku. Np. po złączeniu pary ab w literkę c musimy pamiętać rozwinięcie $c \to ab$.

Aby uprościć implementację i oszczędzać zasoby algorytm używa trzech rodzajów rozwinięć liter:

1)

Rozwinięciem może być dosłownie zapisane słowo złożone z liter z wyjściowego równania. Takie rozwinięcia są używane tylko dla tych właśnie liter, więc zawsze mają postać $a \to a$.
(Algorytm pamięta je jako pary $(-1, a)$.)

2)

Rozwinięciem może być konkatenacja dwóch innych rozwinięć – tak dzieje się, gdy łączymy parę liter w inną literę.
(Algorytm pamięta je jako pary $(0, (p1, p2))$, gdzie $p1$ i $p2$ ro rozwinięcia.)

3)

Rozwinięciem może być wielokrotne powtórzenie innego rozwinięcia - tak dzieje się, gdy łączymy maksymalne bloki litery do innych liter.
(Algorytm pamięta je jako pary (cnt, p), gdzie $cnt > 0$ jest liczbą powtórzeń, a p jest powtórzeniem.)

Podstawianie za zmienną zawsze będzie ciągiem rozwinięć liter. Podstawiania są reprezentowane jako para list – jedna oznacza początek podstawienia, druga koniec. Na przykład, gdy stwierdzamy, że zmienna X zaczyna się literą a, do listy początkowej podstawienia za X dodajemy rozwinięcie litery a.

Zamianę takiej reprezentacji podstawienia w napis (zwykły, pythonowy) realizuje funkcja:

```
def read_result(result: tuple) -> str:
    # reads substitution for a variable
    def read_part(part: tuple) -> str:
        cnt, res = part
        if cnt == -1:
```

```
            return res
        if cnt == 0:
            return read_part(res[0]) + read_part(res[1])
        return cnt * read_part(res)
    return reduce(lambda a, b: a + b,
                  [read_part(part) for part in result[0]], '')\
           + reduce(lambda a, b: a + b,
                    [read_part(part) for part in reversed(result[1])], '')
```

Zauważmy, że lista oznaczająca koniec podstawienia musi być odwrócona, ponieważ jeżeli dla zmiennej X najpierw zdecydowaliśmy, że kończy się na a, a potem że pozostała część kończy się na b, lista końcowa X będzie wyglądała $[<$ rozwinięcie a $>, <$ rozwinięcie b $>]$, natomiast taka kolejność decyzji implikuje, że X kończy się na ba.

2.4 upraszczanie równań

———-[tutaj będzie opis funkcji simplify_eq() i pochodnych]———-

2.5 reprezentacja grafu

Wierzchołki przeszukiwaneg grafu są reprezentowane przez obiekt Node, którego początek implementacji wygląda tak:

```
class Node(object):
    def __init__(self, L: list, R: list,
                 letter_unwrap: dict, variable_subs: dict) -> None:
        self.L = L
        self.R = R
        self.letter_unwrap = letter_unwrap
        self.variable_subs = variable_subs

    def __eq__(self, other) -> bool:
        if len(self.L) != len(other.L) or len(self.R) != len(other.R):
            return False
        for l0, l1 in list(zip(self.L, other.L)):
            if l0 != l1:
                return False
            elif is_letter(l0) and l0.cnt != l1.cnt:
                return False
        for r0, r1 in list(zip(self.R, other.R)):
            if r0 != r1:
                return False
            elif is_letter(r0) and r0.cnt != r1.cnt:
                return False
        return True

    def __len__(self) -> int:
        return len(self.L) + len(self.R)
```

```
    (...)
```

Atrybuty L oraz R oznaczają słowa będące odpowiednio lewą i prawą stroną równania reprezentowanego przez ten wierzchołek. Atrybut letter_unwrap to słownik zawierający dla każdego numeru litery z równania jej rozwinięcie. variable_subs to również słownik, który dla każdego numeru zmiennej z równiania trzyma jej aktualne podstawianie.

———-[tutaj będzie opis funkcji Node.children() i pochodnych]———-

2.6 Przeszukiwanie grafu

Graf trawersowany jest za pomocą przeszukiwania wszerz - takim sposobem, mimo, że cały graf ma długość wykładniczą względem długości początkowego równania, możemy po nim przechodzić i robić jakikolwiek postęp.

```
def bfs(start: Node, max_len: int) -> dict:
    # breadth first search
    if start.terminal():
        return start.variable_subs
    q = Queue()
    q.put(start)
    visited = {hash(start)}
    while not q.empty():
        node = q.get()
        for child in node.children():
            child_hash = hash(child)
            if child_hash not in visited and len(child) <= max_len:
                visited.add(child_hash)
                if child.terminal():
                    return child.variable_subs
                q.put(child)
    return {}
```

Metoda Node.terminal() sprawdza, czy wierzchołek reprezentuje trywialnie spełnione równanie, w którym każde podstawianie za zmienną jest niepuste:

```
class Node(object):
    (...)
    def terminal(self) -> bool:
        # checks if node represents correct solution
        for substitution in self.variable_subs.values():
            if not substitution[0] and not substitution[1]:
                return False

        if all(map(is_variable, self.L + self.R)):
            left_side = reduce(lambda a, b:
                            a + read_result(self.variable_subs[b.nr]),
                          self.L, '')
            right_side = reduce(lambda a, b:
                             a + read_result(self.variable_subs[b.nr]),
```

```
                                  self.R, '')
            return left_side == right_side

        return not self.L and not self.R
    (...)
```

Są dwie możliwości, kiedy równanie uznajemy za trywialnie spełnione

1)

Równanie jest puste. Ponieważ w każdym kroku upraszczamy równania, każde równanie postaci $w = w$, gdzie w t odowolne słowo natychmiast upraszcza się do pustego.

2)

Równanie zawiera tylko zmienne, a po podstawieniu za nie aktualnych podstawnień otrzymujemy równanie postaci $w = w$.

Dodatkową optymalizacją w przeszukiwaniu grafu jest implementacja metody Node.__hash__(), dzięki której za takie same uznawane są wierzchołki zawierające równania takie same co do zamiany numerów zmiennych – takie równania nazwiemy izomorficznymi.

```
class Node(object):
    (...)
    def __hash__(self) -> int:
        rep_L, rep_R = representative_eq(self.L, self.R)
        return hash((tuple(rep_L), tuple(rep_R)))

    (...)


def representative_eq(L: list, R: list) -> Node:
    # for given equation build represantative node of it
    def translate(side: list, letter_nr_trans: dict, free_nr: int):
        result = []
        for symbol in side:
            if is_variable(symbol):
                result.append(copy(symbol))
            else:
                if symbol.nr in letter_nr_trans:
                    result.append(Letter(letter_nr_trans[symbol.nr], symbol.cn
                else:
                    letter_nr_trans[free_nr] = symbol.nr
                    result.append(Letter(free_nr, symbol.cnt))
                    free_nr += 1
        return result, free_nr

    letter_nr_trans = {}
    rep_L, free_nr = translate(L, letter_nr_trans, 0)
```

```
    rep_R, _ = translate(R, letter_nr_trans, free_nr)
    return rep_L, rep_R
```

Funkcja representative_eq() zwraca równanie izomorficzne z danym równaniem, taki że spośród wszystkich takich, jego ciąg numerów kolejnych wystąpień liter jest najmniejszy leksykograficznie.