Laboratorium 10 cz. 2 - Teoria śladów

Piotr Karamon

13.01.2025.

Treść zadania

Dane są:

- Alfabet A, w którym każda litera oznacza akcje.
- Relacja niezależności I, oznaczająca które akcje są niezależne (przemienne, tzn. można je wykonać w dowolnej kolejności i nie zmienia to wyniku końcowego).
- Słowo w oznaczające przykładowe wykonanie sekwencji akcji.

Napisz program w dowolnym języku, który:

1. Wyznacza relację niezależności I
2. Wyznacza ślad $[w]$ względem relacji I
3. Wyznacza postać normalną na Foaty FNF($[w]$) śladu $[w]$
4. Wyznacza graf zależności dla słowa w
5. Wyznacza postać normalną Foaty na podstawie grafu

Rozwiązanie

Kod został napisany w języku Java. Klasą która rozwiązuje zadane problemy jest `TraceSolver`. Otrzymuje na wejściu alfabet, zbiór par informujących o tym, które akcje są niezależne oraz słowo.

```
record Pair(char a, char b) {
    @Override
    public String toString() {
        return "(%s, %s)".formatted(a, b);
    }
}
class TraceSolver {
    private final Set<Character> alphabet;
    private final Set<Pair> independenceRelation;
    private final Set<Pair> dependenceRelation;
    private final String word;

    public TraceSolver(Set<Character> alphabet,
                       Set<Pair> independentPairs,
                       String word

    ) {
        this.alphabet = alphabet;
        this.independenceRelation = computeIndependenceRelation(independentPairs);
        this.dependenceRelation = computeDependenceRelation(independenceRelation);
        this.word = word;
    }
}
```

Wyznaczenie relacji niezależności i zależności

Relację niezależności otrzymujemy poprzez stworzenie zbioru, który zawiera pary (a, b), (b, a) dla każdej takiej pary w `independentPairs`.

```
private static Set<Pair> computeIndependenceRelation(Set<Pair> independentPairs) {
    return independentPairs
            .stream()
            .flatMap(
                    pair -> Stream.of(new Pair(pair.a(), pair.b()), new Pair(pair.b(), pair.a()))
            )
            .collect(Collectors.toSet());
}

public Set<Pair> getIndependenceRelation() {
    return independenceRelation;
}
```

Relację zależności tworzymy iterując po wszystkich parach $(a, b) \in A^2$, sprawdzamy czy para ta nie należy do I, jeżeli tak to należy ona do D i dodajemy ją do tworzonego zbioru.

```
private Set<Pair> computeDependenceRelation(Set<Pair> independentRelation) {
    Set<Pair> dependenceRelation = new HashSet<>();
    for (char a : alphabet) {
        for (char b : alphabet) {
            Pair pair = new Pair(a, b);
            if (!independentRelation.contains(pair)) {
                dependenceRelation.add(pair);
            }
        }
    }

    return dependenceRelation;
}

public Set<Pair> getDependenceRelation() {
    return dependenceRelation;
}
```

Wyznaczanie śladu $[w]$

Używamy następującego algorytmu:

1. Tworzymy zbiór `trace` i dodajemy do niego słowo w.
2. Próbujemy powiększyć zbiór `trace` dodając do niego wyrazy, które powstały w wyniku zamiany dwóch sąsiednich liter, które są niezależne.
   (a) Tworzymy zbiór `newTrace`
   (b) Dla każdego wyrazu w `trace`:
      i. Dodajemy go do `newTrace`
      ii. Przechodzimy przez wyraz i jeżeli dwie litery obok siebie są niezależne, to zamieniamy je miejscami i dodajemy je do `newTrace`
   (c) Jeżeli `newTrace = trace` to oznacza, że kończymy. Jeżeli jest to fałsz to znów próbujemy powiększyć `trace`

```java
public Set<String> getTrace() {
    Set<String> trace = new HashSet<>();
    trace.add(word);

    while (true) {
        Set<String> newTrace = new HashSet<>();

        for (String word : trace) {
            newTrace.add(word);
            for (int i = 0; i < word.length() - 1; i++) {
                char a = word.charAt(i);
                char b = word.charAt(i + 1);
                Pair pair = new Pair(a, b);
                if (independenceRelation.contains(pair)) {
                    String newWord = word.substring(0, i) + b + a + word.substring(i + 2);
                    newTrace.add(newWord);
                }
            }
        }
        if (newTrace.equals(trace)) {
            break;
        }
        trace = newTrace;
    }
    return trace;
}
```

Wyznaczanie postaci normalnej Foaty

Algorytm:

1. Dla każdej litery $a \in A$ tworzymy stos.
2. Przechodzimy przez wyraz w od prawej do lewej:
   (a) Dodajemy literę na odpowiadający jej stos.
   (b) Dla wszystkich $c \in A$, które są zależne z literą dodajemy znacznik $*$ na odpowiadający im stos
3. Dopóki wszystkie stosy nie są puste:
   (a) Zdejmujemy litery na górze stosów. Te zdjęte litery tworzą klasę Foaty.
   (b) Dla każdej ze zdjętych liter, zdejmujemy ich znaczniki.

```java
public String getFoataNormalForm() {
    Map<Character, ArrayList<Character>> stacks = getStacks();

    List<String> foataNormal = new ArrayList<>();
    while (stacks.values().stream().anyMatch(stack -> !stack.isEmpty())) {
        List<Character> tops = stacks.values()
                .stream()
                .filter(stack -> !stack.isEmpty() && stack.getLast() != '*')
                .map(ArrayList::removeLast)
                .toList();

        for (char letter : tops) {
            for (Map.Entry<Character, ArrayList<Character>> entry : stacks.entrySet()) {
                if (entry.getKey() != letter && !independenceRelation.contains(new Pair(entry.getKey(),
                ↪  letter))) {
                    entry.getValue().removeLast();
                }
            }
        }

        foataNormal.add(tops.stream().map(String::valueOf).collect(Collectors.joining()));
```

```
    }

    return formatFoataNormalForm(foataNormal);
}

private Map<Character, ArrayList<Character>> getStacks() {
    Map<Character, ArrayList<Character>> stacks = new HashMap<>();
    for (int i = word.length() - 1; i >= 0; i--) {
        char c = word.charAt(i);

        stacks.putIfAbsent(c, new ArrayList<>());
        stacks.get(c).add(c);

        for (char a : alphabet) {
            if (a != c && !independenceRelation.contains(new Pair(a, c))) {
                stacks.putIfAbsent(a, new ArrayList<>());
                stacks.get(a).add('*');
            }
        }

    }
    return stacks;
}

private String formatFoataNormalForm(List<String> foataNormal) {
        return foataNormal.stream().map("(%s)"::formatted).collect(Collectors.joining());
}
```

Wyznaczanie grafu zależności dla słowa w

Używamy następującego algorytmu:

1. Tworzymy graf w którym wierzchołkami są wszystkie litery ze słowa w.
2. Następnie dla każdej litery a ze słowa w, idąc od lewej do prawej:

   Przechodzimy przez litery będące dalej w słowie w. Jeżeli a jest zależne z b(litera dalej w słowie) to dodajemy krawędź skierowaną z a do b.
3. W ten sposób otrzymujemy graf, ale z przechodnimi krawędziami.
4. Usuwamy krawędzie przechodnie:
   (a) Tworzymy nowy graf `reducedGraph`.
   (b) Dla każdej krawędzi (a, b) w oryginalnym grafie. sprawdzamy czy istnieje niebezpośrednia droga z a do b. Jeżeli nie istnieje taka krawędź to dodajemy (a, b) do `reducedGraph`.
5. Na koniec konwertujemy graf do formatu `dot`.

```
public String getDependenceGraphInDotFormat() {
    List<List<Integer>> graph = createDependenceGraphWithTransitiveEdges();
    List<List<Integer>> reducedGraph = removeTransitiveEdges(graph);
    return convertGraphToDot(reducedGraph);
}


private List<List<Integer>> createDependenceGraphWithTransitiveEdges() {
    List<List<Integer>> graph = word.chars()
            .mapToObj(__ -> new ArrayList<Integer>())
            .collect(Collectors.toList());

    for (int i = 0; i < word.length(); i++) {
        char a = word.charAt(i);
        for (int j = i + 1; j < word.length(); j++) {
```

```
            char b = word.charAt(j);
            if (!independenceRelation.contains(new Pair(a, b))) {
                graph.get(i).add(j);
            }
        }
    }
    return graph;
}


private List<List<Integer>> removeTransitiveEdges(List<List<Integer>> graph) {
    List<List<Integer>> reducedGraph = word.chars()
            .mapToObj(__ -> new ArrayList<Integer>())
            .collect(Collectors.toList());

    for (int v = 0; v < word.length(); v++) {
        for (int neighbour : graph.get(v)) {
            if (!hasIndirectPath(graph, v, neighbour)) {
                reducedGraph.get(v).add(neighbour);
            }
        }
    }
    return reducedGraph;
}


private boolean hasIndirectPath(List<List<Integer>> graph, int u, int v) {
    Stack<Integer> stack = new Stack<>();
    stack.push(u);
    Set<Integer> visited = new HashSet<>();
    visited.add(u);

    while (!stack.isEmpty()) {
        int current = stack.pop();
        for (int neighbor : graph.get(current)) {
            if (current == u && neighbor == v) continue;
            if (neighbor == v) return true;
            if (!visited.contains(neighbor)) {
                stack.push(neighbor);
                visited.add(neighbor);
            }
        }
    }
    return false;
}

private String convertGraphToDot(List<List<Integer>> reducedGraph) {
    StringBuilder sb = new StringBuilder();
    sb.append("digraph G {\n");

    for (int i = 0; i < word.length(); i++) {
        for (int j : reducedGraph.get(i)) {
            sb.append("\t%s -> %s\n".formatted(i, j));
        }
    }

    for (int i = 0; i < word.length(); i++) {
        sb.append("\t%s[label = \"%s\"]\n".formatted(i, word.charAt(i)));
    }

    sb.append("}");

    return sb.toString();
}
```

Wyznaczanie postaci normalnej Foaty na podstawie grafu

Algorytm:

1. Tworzymy słownik `inDegree`, gdzie kluczem jest indeks wierzchołka, a wartością jest liczba krawędzi wchodzących do tego wierzchołka.
2. Dopóki słownik `inDegree` nie jest pusty:
   (a) Dla każdego wierzchołka a takiego, że `inDegree[a] == 0`:
      i. Znajdujemy sąsiadów i zmniejszamy im `inDegree[b]` o 1.
      ii. Usuwamy a z `inDegree`.
   (b) Usunięte wierzchołki tworzą klasę Foaty.

```
public String getFoataNormalFormFromGraph() {
    List<List<Integer>> graph = createDependenceGraphWithTransitiveEdges();
    List<List<Integer>> reducedGraph = removeTransitiveEdges(graph);
    Map<Integer, Integer> inDegree = computeInDegrees(reducedGraph);

    List<String> foataNormal = new ArrayList<>();
    while (!inDegree.isEmpty()) {
        List<Integer> nodesWithZeroInDegree = inDegree.entrySet()
                .stream()
                .filter(entry -> entry.getValue() == 0)
                .map(Map.Entry::getKey)
                .toList();

        for (int node : nodesWithZeroInDegree) {
            for (int neighbor : reducedGraph.get(node)) {
                inDegree.put(neighbor, inDegree.get(neighbor) - 1);
            }
            inDegree.remove(node);
        }

        foataNormal.add(
                nodesWithZeroInDegree
                        .stream()
                        .map(i -> String.valueOf(word.charAt(i)))
                        .collect(Collectors.joining())
        );
    }


    return formatFoataNormalForm(foataNormal);
}

private Map<Integer, Integer> computeInDegrees(List<List<Integer>> reducedGraph) {
    Map<Integer, Integer> inDegree = new HashMap<>();
    for (int i = 0; i < word.length(); i++) {
        inDegree.putIfAbsent(i, 0);
        for (int j : reducedGraph.get(i)) {
            inDegree.put(j, inDegree.getOrDefault(j, 0) + 1);
        }
    }
    return inDegree;
}
```

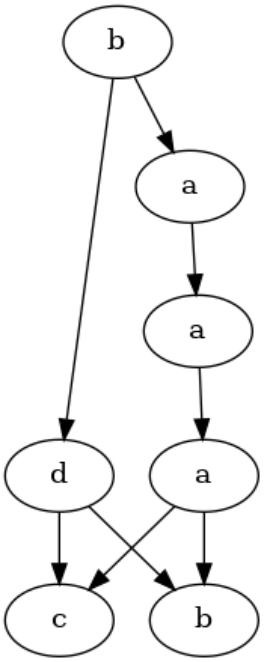
Wyniki dla przykładowych danych testowych

```
public static void main(String[] args) {
    System.out.println("DANE TESTOWE 1\n");
    printResults(new TraceSolver(
            Set.of('a', 'b', 'c', 'd'),
            Set.of(new Pair('a', 'd'), new Pair('b', 'c')),
            "baaadcb"
    ));

    System.out.println("\nDANE TESTOWE 2\n");
    printResults(new TraceSolver(
            Set.of('a', 'b', 'c', 'd', 'e', 'f'),
            Set.of(new Pair('a', 'd'), new Pair('b', 'e'),
                   new Pair('c', 'd'), new Pair('c', 'f')
            ),
            "acdcfbbe"
    ));
}
private static void printResults(TraceSolver traceSolver) {
    System.out.printf("I = %s%n", traceSolver.getIndependenceRelation());
    System.out.printf("D = %s%n", traceSolver.getDependenceRelation());
    System.out.printf("[w] = %s%n", traceSolver.getTrace());
    System.out.printf("FNF([w]) = %s%n", traceSolver.getFoataNormalForm());
    System.out.printf("FNF([w]) from graph = %s%n", traceSolver.getFoataNormalFormFromGraph());
    System.out.println("Dependence graph in DOT format:");
    System.out.println(traceSolver.getDependenceGraphInDotFormat());
}
```

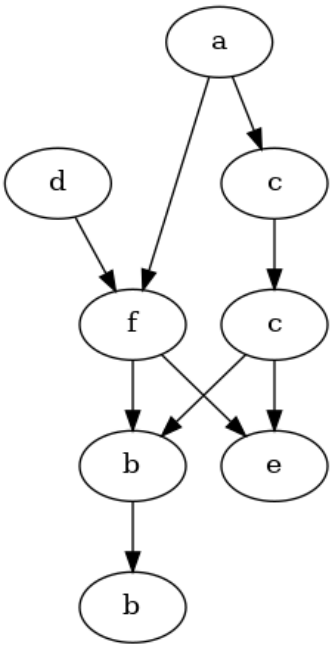
Wynik programu:

```
DANE TESTOWE 1

I = [(b, c), (a, d), (d, a), (c, b)]
D = [(d, d), (c, c), (b, b), (a, a), (c, d), (a, b), (b, d), (a, c), (d, b), (c,
↪  a), (d, c), (b, a)]
[w] = [baadacb, badaabc, badaacb, bdaaacb, baaadcb, bdaaabc, baaadbc, baadabc]
FNF([w]) = (b)(ad)(a)(a)(bc)
FNF([w]) from graph = (b)(ad)(a)(a)(cb)
Dependence graph in DOT format:
digraph G {
    0 -> 1
    0 -> 4
    1 -> 2
    2 -> 3
    3 -> 5
    3 -> 6
    4 -> 5
    4 -> 6
    0[label = "b"]
    1[label = "a"]
    2[label = "a"]
    3[label = "a"]
    4[label = "d"]
    5[label = "c"]
    6[label = "b"]
}
```

```
DANE TESTOWE 2

I = [(c, d), (c, f), (b, e), (a, d), (f, c), (e, b), (d, a), (d, c)]
D = [(f, f), (d, d), (b, b), (d, e), (b, c), (d, f), (b, d), (b, f), (e, a), (e,
↪  c), (c, a), (e, d), (c, b), (e, e), (c, c), (a, a), (e, f), (a, b), (c, e), (a,
↪  c), (a, e), (a, f), (f, a), (f, b), (f, d), (d, b), (f, e), (b, a)]
[w] = [acdcfbbe, acdcfebb, acdfcbbe, acdfcebb, adccfbeb, adccfbbe, adccfebb,
↪  adcfcbbe, adcfcebb, dafccbeb, acdcfbeb, daccfbbe, daccfebb, accdfbeb, acdfcbeb,
↪  adfccbeb, dacfcbbe, dacfcebb, dacfcbeb, daccfbeb, adfccbbe, adfccebb, accdfbbe,
↪  accdfebb, dafccebb, dafccbbe, adcfcbeb]
FNF([w]) = (ad)(cf)(c)(be)(b)
FNF([w]) from graph = (ad)(cf)(c)(be)(b)
Dependence graph in DOT format:
digraph G {
    0 -> 1
    0 -> 4
    1 -> 3
    2 -> 4
    3 -> 5
    3 -> 7
    4 -> 5
    4 -> 7
    5 -> 6
    0[label = "a"]
    1[label = "c"]
    2[label = "d"]
    3[label = "c"]
    4[label = "f"]
    5[label = "b"]
    6[label = "b"]
    7[label = "e"]
}
```



Rysunek 1: Graf zależności dla danych testowych 1.



Rysunek 2: Graf zależności dla danych testowych 2.

Wnioski

Wyniki algorytmów teorii śladów są zgodne z podanymi wynikami dla danych testowych. W obu przypadkach poprawność potwierdza również fakt, że niezależnie od sposobu obliczenia postać normalna Foaty jest identyczna.

Bibliografia

- Volker Diekert, Yves Metivier : Partial Commutation and Traces
- Dot Language Documentation