Teoria współbieżności - laboratorium 12

Łukasz Jezapkowicz

8 stycznia 2021

1 Opis programu z komentarzami. Napisz program w dowolnym języku, który:

1.1 Wyznacza relację zależności D.

```
void getDependenceRelation() {
    for (String a : alphabet) {
        for (String b : alphabet) {
            if (!I.containsKey(a) || !I.get(a).contains(b)) {
                if (!D.containsKey(a)) {
                    HashSet<String> hs = new HashSet<>();
                    hs.add(b);
                    D.put(a, hs);
                } else {
                    D.get(a).add(b);
                }
            }
        }
    }
}
```

Powyższy kod wyznacza relację zależności D na podstawie alfabetu A oraz relacji niezależności I. Sprawdza on po kolei wszystkie kombinacje par alfabetu i zapisuje do relacji D , te które nie należą do I.

1.2 Wyznacza ślad $[w]$ względem relacji I.

```
void getTraces(String word) {
    traces.add(word);

    for (int i = 0; i < word.length() - 1; i++) {
        String key = Character.toString(word.charAt(i));
        String swapped = word.substring(0, i) + word.charAt(i + 1) + word.charAt(i);
        if (i + 2 < word.length())
            swapped += word.substring(i + 2);
        if (I.containsKey(key) && I.get(key).contains(Character.toString(word.charAt(i + 1)))
                && !traces.contains(swapped)) {
            getTraces(swapped);
            i++;
        }
    }
}
```

Powyższy kod wyznacza ślad $[w]$ względem relacji niezależności I. Dodaje on rekurencyjnie kolejne słowa (począwszy od słowa wyjściowego), w którym zamienione są pewne dwa znaki, jeżeli pozwala na to relacja

niezależności I. By uniknąć nieskończonej pętli funkcja sprawdza czy nowe słowo nie zostało już dodane do zbioru.

1.3 Wyznacza postać normalną Foaty $FNF([w])$ śladu $[w]$.

```
void getFoataForm() {
    LinkedHashMap<String, Stack<String>> stacks = new LinkedHashMap<>();
    for (String c : alphabet)
        stacks.put(c, new Stack<>());

    for (int i = word.length() - 1; i >= 0; i--) {
        String c = Character.toString(word.charAt(i));
        stacks.get(c).push(c);
        for (String ch : alphabet) {
            if (!c.equals(ch) && (!I.containsKey(c) || !I.get(c).contains(ch))) {
                stacks.get(ch).push( item: "*");
            }
        }
    }
```

Powyższy kod tworzy zbiór stosów (o liczności $|A|$) i uzupełnia stos zgodnie z algorytmem z książki "Handbook of Formal Languages" (czyli dodaje literę na odpowiedni stos i dodaje markery na stosach liter zależnych od dodanej litery).

```
    while (true) {
        StringBuilder new_foata_block = new StringBuilder("(");
        boolean anyStackNotEmpty = false;
        HashMap<String, Integer> poppedMarkers = new HashMap<>();
        for (String c : alphabet)
            poppedMarkers.put(c, 0);

        for (String key : stacks.keySet()) {
            if (stacks.get(key).empty())
                continue;
            anyStackNotEmpty = true;
            if (stacks.get(key).peek().equals("*") || poppedMarkers.get(key) != 0)
                continue;

            String top = stacks.get(key).pop();
            new_foata_block.append(top);

            for (String c : D.get(top)) {
                if (!c.equals(top) && !stacks.get(c).empty()) {
                    stacks.get(c).pop();
                    poppedMarkers.put(c, poppedMarkers.get(c) + 1);
                }
            }
        }

        new_foata_block.append(")");
        if (!anyStackNotEmpty) break;
        foataForm.append(new_foata_block);
    }
}
```

Powyższy kod tworzy postać normalną Foaty $FNF([w])$ śladu $[w]$. Ze stosów ściągane są kolejne zbiory liter (zbiory Foaty) zgodnie ze wspomnianym wcześniej algorytmem (ściągamy wszystkie litery z góry stosów i usuwamy markery ze stosów liter zależnych od ściągniętych liter).

1.4 Wyznacza graf zależności w postaci minimalnej dla słowa w.

```
void createDependenceGraph(final String word, final LinkedHashMap<String, HashSet<String>> D) {
    for (int i = 0; i < word.length(); i++) {
        vertices.add(i + 1);
        mapping.put(i + 1, Character.toString(word.charAt(i)));
        // now adding edges
        for (int j = 0; j < i; j++) {
            String tau1 = mapping.get(j + 1);
            String tau2 = mapping.get(i + 1);
            if (D.containsKey(tau1) && D.get(tau1).contains(tau2)) {
                if (!edges.containsKey(j + 1)) {
                    HashSet<Integer> hs = new HashSet<>();
                    hs.add(i + 1);
                    edges.put(j + 1, hs);
                } else {
                    edges.get(j + 1).add(i + 1);
                }
            }
        }
    }
    this.minimizeDependenceGraph();
}
```

Powyższy kod tworzy graf zależności dla słowa w zgodnie z algorytmem podanym na wykładzie (iteracyjnie buduje graf dodając kolejne podgrafy i łącząc krawędzie zależnych liter). Graf musi zostać zminimalizowany.

```
private void minimizeDependenceGraph() {
    for (Integer vertex : vertices) {  // for every vertex we do DFS of all his neighbours
        if (!edges.containsKey(vertex))
            continue;
        HashSet<Integer> toDelete = new HashSet<>();
        for (Integer neighbour : edges.get(vertex)) {
            if (toDelete.contains(neighbour))
                continue;
            HashSet<Integer> visited = new HashSet<>();
            this.DFS(visited, neighbour);
            for (Integer vertexVisited : visited) {
                if (!vertexVisited.equals(neighbour)) {
                    toDelete.add(vertexVisited);
                }
            }
        }

        for (Integer neighbour : toDelete) {
            edges.get(vertex).remove(neighbour);
        }
    }
}
```

```java
private void DFS(HashSet<Integer> visited, int vertex) {
    visited.add(vertex);
    if (!edges.containsKey(vertex))
        return;
    for (Integer neighbour : edges.get(vertex)) {
        if (!visited.contains(neighbour))
            this.DFS(visited, neighbour);
    }
}
```

Powyższy kod minimalizuje graf zależności przy pomocy algorytmu DFS. Dla każdego wierzchołka u oraz wszystkich jego sąsiadów v szukamy wszystkich osiągalnych wierzchołków v' z v oraz usuwamy wszystkie istniejące krawędzie (u, v').

1.5 Wyznacza postać normalną Foaty na podstawie grafu.

```java
private void topologicalDFS(HashSet<Integer> visited, Stack<Integer> tsort, int vertex) {
    visited.add(vertex);
    if (edges.containsKey(vertex)) {
        for (Integer neighbour : edges.get(vertex)) {
            if (!visited.contains(neighbour))
                this.topologicalDFS(visited, tsort, neighbour);
        }
    }
    tsort.push(vertex);
}

void createTopologicalSort() {
    HashSet<Integer> visited = new HashSet<>();
    Stack<Integer> tsort = new Stack<>();
    for (Integer vertex : vertices) {  // for every vertex we do DFS of all his neighbours
        if (!visited.contains(vertex))
            this.topologicalDFS(visited, tsort, vertex);
    }
    while (!tsort.empty())
        topologicalSort.add(mapping.get(tsort.pop()));
}
```

Powyższy kod tworzy sortowanie topologiczne stworzonego wcześniej grafu. Używa do tego celu algorytmu DFS, w którym wrzucamy na stos kolejne wierzchołki (zaczynając od wierzchołków, z których nie ma krawędzi lub nie ma krawędzi do wierzchołka nieodwiedzonego). Odwrócony stos to szukane sortowanie topologiczne.

```
void getFoataFromGraph() {
    dgraph.createTopologicalSort();
    ArrayList<String> topological = dgraph.topologicalSort;
    int index = 0;
    while (index < word.length()) {
        StringBuilder new_foata_chunk = new StringBuilder("(");
        new_foata_chunk.append(topological.get(index));

        int start_index = index++;
        while (index < word.length()) {
            boolean isIndependent = true;
            for (int i = start_index; i < index; i++) {
                if (!I.containsKey(topological.get(index))
                        || !I.get(topological.get(index)).contains(topological.get(i))) {
                    isIndependent = false;
                    break;
                }
            }
            if (isIndependent)
                new_foata_chunk.append(topological.get(index++));
            else
                break;
        }
        new_foata_chunk.append(")");
        foataFromGraphForm.append(new_foata_chunk);
    }
}
```

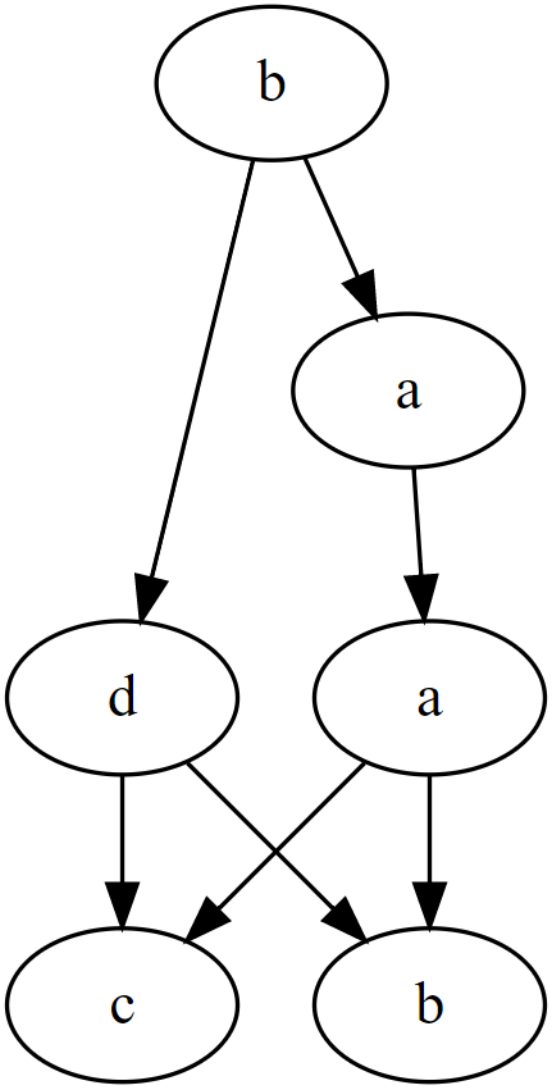
Powyższy kod tworzy postać normalną Foaty na podstawie sortowania topologicznego. Tworzenie kolejnych bloków Foaty wygląda następująco:
- bierzemy kolejne znaki słowa (odpowiadające kolejnym wierzchołkom sortowania topologicznego)
- dobieramy kolejne znaki dopóki wszystkie znaki są parami niezależne

2 Wyniki działania dla przykładowych danych.

2.1 Przykład 1

$A = \{a, b, c, d\}$
$I = \{(a, d), (d, a), (b, c), (c, b)\}$
$w = baadcb$

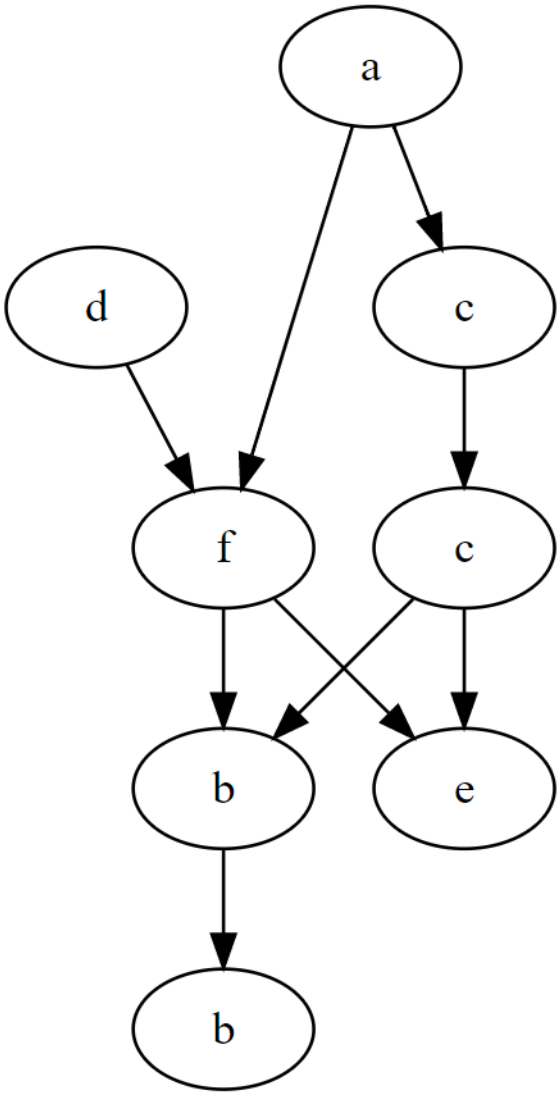
```
SUMMARY:
Alphabet: {a, b, c, d}
I: {(a,d), (d,a), (b,c), (c,b)}
Input word: baadcb
D: {(a,a), (a,b), (a,c), (b,a), (b,b), (b,d), (c,a), (c,c), (c,d), (d,b), (d,c), (d,d)}
Traces: {baadcb, badacb, bdaacb, bdaabc, badabc, baadbc}
Foata form: (b)(ad)(a)(bc)
Foata (from Graph) form: (b)(da)(a)(bc)
```



2.2 Przykład 2

$A = \{a, b, c, d, e, f\}$
$I = \{(a, d), (d, a), (b, e), (e, b), (c, d), (d, c), (c, f), (f, c)\}$
$w = acdcfbbe$

```
SUMMARY:
Alphabet: {a, b, c, d, e, f}
I: {(a,d), (d,a), (d,c), (b,e), (e,b), (c,d), (c,f), (f,c)}
Input word: acdcfbbe
D: {(a,a), (a,b), (a,c), (a,e), (a,f), (b,a), (b,b), (b,c), (b,d), (b,f), (c,a), (c,b),
(c,c), (c,e), (d,b), (d,d), (d,e), (d,f), (e,a), (e,c), (e,d), (e,e), (e,f), (f,a),
(f,b), (f,d), (f,e), (f,f)}
Traces: {acdcfbbe, adccfbbe, daccfbbe, dacfcbbe, adcfcbbe, acdfcbbe, acdfcbeb, adcfcbeb, dacfcbeb,
dafccbeb, adfccbeb, adfccebb, dafccebb, dacfcebb, adcfcebb, acdfcebb, acdcfebb, adccfebb,
daccfebb, daccfbeb, adccfbeb, acdcfbeb, accdfbeb, accdfebb, dafccbbe, adfccbbe}
Foata form: (ad)(cf)(c)(be)(b)
Foata (from Graph) form: (da)(fc)(c)(eb)(b)
```



2.3 Przykład 3

$A = \{a, b, c, d, e\}$
$I = \{(a, c), (c, a), (a, d), (d, a), (b, d), (d, b), (b, e), (e, b)\}$
$w = acebdac$

```
SUMMARY:
Alphabet: {a, b, c, d, e}
I: {(a,c), (a,d), (c,a), (d,a), (d,b), (b,d), (b,e), (e,b)}
Input word: acebdac
D: {(a,a), (a,b), (a,e), (b,a), (b,b), (b,c), (c,b), (c,c), (c,d), (c,e), (d,c), (d,d),
(d,e), (e,a), (e,c), (e,d), (e,e)}
Traces: {acebdac, caebdac, cabedac, acbedac, acbeadc, cabeadc, caebadc, acebadc, cabedca,
acbedca, acebdca, caebdca, caedbca, acedbca, acedbac, caedbac}
Foata form: (ac)(be)(ad)(c)
Foata (from Graph) form: (ca)(be)(ad)(c)
```