Zadanie domowe 1 - Teoria współbieżności
grupa czwartek 13:15

Piotr Albiński

18 listopada 2024

1 Informacje początkowe

- W części z rozwiązaniem, będę umieszczał kod, który uznaję za potrzebny do pokazania z opisem. Na samym końcu wklejony jest cały kod w całości.
- Zadanie zostało napisane w języku Rust.
- Dodatkowo korzystam z 1 zależności ułatwiającej manipulację grafem petgraph. Znajduje się ona w `cargo.toml` i jest automatycznie instalowana przy użyciu `cargo run`.
- Do rysowania grafu używam zaleconego w poleceniu Graphviz. Generuję postać grafu w formacie DOT i w kodzie uruchamiam odpowiednią komendę. Stąd do uruchomienia poprawnie programu, potrzebny jest zainstalowany program. W terminalu musi działać komenda `dot` (np. na windows wiąże się z dodaniem do `PATH`, o co jesteśmy pytani w trakcie instalacji).
- Dane wejściowe przyjmuję z plik tekstowego, którego ustaliłem następującą postać:

```
4       <- liczba transformacji
x <= x + y        <- transformacje
y <= y + 2z       ...
x <= 3x  +z       ...
z <= y - z        ...
abcd        <- alfabet (oznaczenia kolejnych równań)
baadcb      <- słowo wejściowe
```

- przyjąłem konwencję podczas przekształcania wejścia, że symbole w transformacjach spełniają funkcję `.is_alphabetic()`, a symbole alfabetu `.is_alphanumeric()`
- W celu uruchomienia programu wystarczy wpisać w folderze projektu np. `cargo run data1.txt`. Argument uruchomienia to nazwa pliku wejściowego. W przypadku braku podania argumentu, zostaniemy o wprowa-

dzenie jego nazwy w konsoli. W folderze zawarłem trzy pliki z danymi dostarczonymi danymi:

- Dane testowe 1 z pdf ⇒ `data1.txt`
- Dane testowe 2 z pdf, inaczej `case1.txt` ⇒ `data2.txt`
- `case2.txt` ⇒ `data3.txt`
- Następnie Zbiory D, I, postać Foaty są zapisywane do pliku `result.txt`. Dla grafu tworzony jest plik `graph.dot` oraz z niego `graph.png`.

2 Rozwiązanie zadania

2.1 Wyznaczenie relacji zależności D

2.2 Wyznaczenie relacji zależności I

Opis wyznaczenia zbiorów Najpierw znalazłem wszystkie zmienne, które występują w równaniach. Teraz dla każdej zmiennej w każdej transformacji ustaliłem jej *VariableSituation*. Przez to rozumiem następujące możliwości położenia:

```rust
#[derive(Debug)]
#[derive(PartialEq)]
enum VariableSituation {
    Left,
    Right,
    Both,
    Neither
}

impl VariableSituation {
    fn is_depend(&self, other:  &VariableSituation) -> bool {
        match (self, other) {
            (VariableSituation::Left | VariableSituation::Both,
            ↪  other)  if *other != VariableSituation::Neither
            ↪  => true,
            (VariableSituation::Right, VariableSituation::Left |
            ↪  VariableSituation::Both) => true,
            (_, _) => false
        }
    }
}
```

W sumie ta logika obejmuje 16 sytuacji, które sprawdził test:

```rust
#[doc = "Test sprawdzający wszystkie możliwe przypadki zależności zmiennych"]
#[test]
fn test_is_depend() {

    assert_eq!(VariableSituation::Left.is_depend(&VariableSituation::Right), true);

    assert_eq!(VariableSituation::Left.is_depend(&VariableSituation::Left), true);

    assert_eq!(VariableSituation::Left.is_depend(&VariableSituation::Both), true);

    assert_eq!(VariableSituation::Left.is_depend(&VariableSituation::Neither), false);

    assert_eq!(VariableSituation::Right.is_depend(&VariableSituation::Left), true);

    assert_eq!(VariableSituation::Right.is_depend(&VariableSituation::Right), false);

    assert_eq!(VariableSituation::Right.is_depend(&VariableSituation::Both), true);

    assert_eq!(VariableSituation::Right.is_depend(&VariableSituation::Neither), false);

    assert_eq!(VariableSituation::Both.is_depend(&VariableSituation::Left), true);

    assert_eq!(VariableSituation::Both.is_depend(&VariableSituation::Right), true);

    assert_eq!(VariableSituation::Both.is_depend(&VariableSituation::Both), true);

    assert_eq!(VariableSituation::Both.is_depend(&VariableSituation::Neither), false);

    assert_eq!(VariableSituation::Neither.is_depend(&VariableSituation::Left), false);

    assert_eq!(VariableSituation::Neither.is_depend(&VariableSituation::Right), false);
```

```
    assert_eq!(VariableSituation::Neither.is_depend(&VariableSituation::Both), false);

    assert_eq!(VariableSituation::Neither.is_depend(&VariableSituation::Neither), false);
}
```

Mając tak poukładane dane teraz wystarczy stworzyć zbiory D i I

```
fn create_sets(transformations_with_variables: &Vec<HashMap<char, VariableSituation>>, alphabet: &Vec<char>) -> (HashSet<(char, char)>, HashSet<(char, char)>) {
    let mut D = HashSet::new();
    let mut I = HashSet::new();
    let mut has_been_added_to_D = false;
    let twv_len = transformations_with_variables.len();
    for i in 0..twv_len {
        for j in i..twv_len{
            if i == j {
                D.insert((alphabet[i], alphabet[j]));
                continue;
            }
            for (variable, situation_first) in &transformations_with_variables[i] {
                match transformations_with_variables[j].get(variable) {
                    Some(situation_second) => {
                        if situation_first.is_depend(situation_second) {
                            D.insert((alphabet[i], alphabet[j]));
                            D.insert((alphabet[j], alphabet[i]));
                            has_been_added_to_D = true;
                            break;
                        }
                    },
                    None => panic!("Variable not found in hashmap")
                }
            }
            if !has_been_added_to_D {
                I.insert((alphabet[i], alphabet[j]));
                I.insert((alphabet[j], alphabet[i]));
            }
```

```
                has_been_added_to_D = false;
        }

        }
        (D, I)
}
```

Widok na to co wykonaliśmy z main:

```
    //pobrania nazwy pliku z argumentów
    let args = std::env::args().collect::<Vec<String>>();
    let filename = if args.len() > 1 {
        args[1].clone()
    } else {
        String::new()
    };
    //odczyt z pliku
    let (n, transformations, alphabet, word) =
    ↪  read_from_file(filename.to_string());

    //przygotowywanie danych do obliczeń
    let  variables = find_variables(&transformations);
    let mut transformations_with_variables: Vec<HashMap<char,
    ↪  VariableSituation>> = create_matrix(&variables, n);
    fill_matrix_with_variables_status(&transformations, &mut
    ↪  transformations_with_variables);

    //tworzenie zbiorów D i I i zapis do pliku
    let (D, I) = create_sets(&transformations_with_variables,
    ↪  &alphabet);
    println!("Zbiór D: {:?}", D);
    println!("Zbiór I: {:?}", I);
    let D_string = format!("{:?}", D);
    let I_string = format!("{:?}", I);
    let mut file = File::create("result.txt").unwrap();
    write!(file, "D = {} \nI = {} \nFNF:", D_string,
    ↪  I_string).unwrap();

```

2.3 Wyznaczenie postaci normalnej Foaty FNF

2.4 A simple algorithm to compute normal forms

Let us describe a simple method which enables to compute normal forms. Let $\mathbb{M}(\Sigma, I)$ be a free partially commutative monoid, we use a stack for each letter of the alphabet Σ. Let x be a word of Σ^*, we scan x from right to left; when processing a letter a it is pushed on its stack and a marker is pushed on the stack of all the letters b $(b \neq a)$ which do not commute with a.

When all of the word has been processed we can compute either the lexicographic normal form or the Foata normal form.

- To get the lexicographic normal form: it suffices to take among the letters being on the top of some stack that letter a being minimal with respect to the given lexicographic ordering. We pop a marker on each stack corresponding to a letter b $(b \neq a)$ which does not commute with a. We repeat this loop until all stacks are empty.
- To get the Foata normal form we take within a loop the set formed by letters being on the top of stacks; arranging the letters in the lexicographic order yields a step. As previously we pop the corresponding markers. Again this loop is repeated until all stacks are empty.

For example, with (Σ, I) as in Ex. 2.1 and the word *badacb* we get the stacks given below. The lexicographic normal form is *baadbc*, and the Foata normal form is $(b)(ad)(a)(bc)$.

*	*b*		
a	*	*	*
a	*	*	*d*
*	*	*	*
*	*b*	*c*	*
a	*b*	*c*	*d*

Rysunek 1: Fragment z książki Diekert and Métivier [1997]

Skorzystałem z zalecenia w poleceniu i użyłem algorytmu z książki Diekert and Métivier [1997]. Algorytm polega na tym, że:

1. Tworzymy dla każdego symbolu alfabetu stos.
2. Wędrujemy od końca do początku słowa **w**.
3. Wpisujemy analizowaną literą na jej stos.
4. Dla innych liter wpisujemy gwiazdkę, jeżeli są w relacji z analizowaną literą, inaczej nic nie wpisujemy.
5. Po przeanalizowaniu całego stosu jesteśmy gotowi odczytać postać FNF.
6. Ściągamy od góry litery ze wszystkich stosów, te litery stanowią warstwę postaci FNF(jeżeli nie było na górze ani jednej liter to ściągamy warstwę gwiazdek tak długo, aż będą jakieś litery).
7. Teraz po zebraniu warstwy, dla każdej zebranej litery usuwamy gwiazdkę ze stosów tych liter, w którymi jest ona w zależności.
8. Kończymy, gdy opróżnimy stos.

```rust

#[doc = "Funkcja odpowiada za wyznaczenie zbiorów Foaty.
Korzystam z algorytmu z kopcami z
książki podanej w treści zadania."]
fn create_foata_normal_form(word: &String, I:
↪  &HashSet<(char,char)>,  alphabet: &Vec<char>) ->
↪  Vec<HashSet<char>> {
    let mut foata: Vec<HashSet<char>> = Vec::new();
    let mut stacks: HashMap<char, Vec<char>> =
    ↪  alphabet.iter().map(|&c| (c, Vec::new())).collect();
    fill_stacks(word, &mut stacks, I);
    fill_foata(&mut stacks, &mut foata, &I, &alphabet);
    foata
}


fn fill_foata(stacks: &mut HashMap<char, Vec<char>>, foata: &mut
↪  Vec<HashSet<char>>, I: &HashSet<(char, char)>, alphabet:
↪  &Vec<char>) {
    let mut to_be_popped = Vec::new();
    while !stacks.values().all(|stack| stack.is_empty()) {
        let mut set = HashSet::new();
        for element in &to_be_popped {
            let stack = stacks.get_mut(element).unwrap();
            stack.pop();
        }
        to_be_popped.clear();
        for stack_el in stacks.iter_mut() {
            let (_letter, stack) = stack_el;
            let current_sign = *stack.last().unwrap_or(&'*'); //
            ↪  '*' has here second meaning, it's a sign that
            ↪  stack is empty (normally is a special stack sign)
            if current_sign != '*' {
                set.insert(current_sign);
                stack.pop();
                for alphabet_letter in alphabet {
                    if !I.contains(&(current_sign,
                    ↪  *alphabet_letter)) {
                        if current_sign == *alphabet_letter {
                            continue;
                        }
                            to_be_popped.push(*alphabet_letter);
                        }
                    }
                }
            }
```

```
39
40          if set.is_empty() {
41              for stack_el in stacks.iter_mut() {
42                  let (letter, stack) = stack_el;
43                  stack.pop();
44              }
45          } else {
46              foata.push(set);
47          }
48      }
49      }
50
51
52  fn fill_stacks(word: &String, stacks: &mut HashMap<char,
    ↪  Vec<char>>, I: &HashSet<(char, char)>) {
53      for c in word.chars().rev() {
54          for stack_el in stacks.iter_mut() {
55              let (letter, stack) = stack_el;
56              if !I.contains(&(c, *letter)) {
57                  if c == *letter {
58                      stack.push(c);
59                  } else {
60                      stack.push('*');
61                  }
62              }
63
64          }
65
66      }
67  }
```

Fragment w main:

```
1
2       //tworzenie FNF i zapis do pliku
3       let foata = create_foata_normal_form(&word, &I, &alphabet);
4       println!("Zbiory Foaty: ");
5       for (i, set) in foata.iter().enumerate() {
6           println!("Zbiór nr {}: {:?}", i+1, set);
7           write!(file, "\nZbiór nr {}: {:?}", i+1, set).unwrap();
8       }
9
10      drop(file);
11
        ↪  println!("Zbiory i postać Foaty zostały dodane do result.txt");
12
```

2.4 Rysowanie grafu zależności w postaci minimalnej

Mając zbiór zależności stworzenie grafu jest trywialne. Problemem jest usunięcie redundantnych połączeń (Wikipedia link do zagadnienia). Skorzystałem tutaj z pomysłu zaimplementowanego w NetworkX. Dany mamy graf zależności ze wszystkimi krawędziami. Algorytm polega on na tym, że dla każdego wierzchołka, dla jego sąsiadów wywołujemy przeszukiwanie grafu(ja skorzystałem z BFS). Teraz, do nowego grafu dodajemy sąsiadów i tylko te krawędzie do których sąsiedzi nie byli w stanie się dostać.

```
1
2  fn create_dependency_graph(word: &String, D:
   ↪  &HashSet<(char,char)>) -> DiGraph<char, ()> {
3      let mut graph = DiGraph::new();
4      for c in word.chars() {
5          graph.add_node(c);
6      }
7      for node in graph.node_indices() {
8          for other_node in graph.node_indices() {
9              if node >= other_node {
10                 continue;
11             }
12             if D.contains(&(graph[node], graph[other_node])) {
13                 graph.add_edge(node, other_node, ());
14             }
15         }
16     }
17     graph
18 }
19
20 fn transitive_reduction(graph: &DiGraph<char, ()>) ->
   ↪  DiGraph<char, ()> {
21     let mut transitive_reducted_graph = DiGraph::new();
22     for node in graph.node_indices() {
23         transitive_reducted_graph.add_node(graph[node]);
24     }
25     let mut descendants = HashMap::new();
26     for node in graph.node_indices() {
27         let mut node_neigbours: HashSet<NodeIndex> =
           ↪  graph.neighbors(node).collect();
28         for neighbour in node_neigbours.clone() {
29             if node_neigbours.contains(&neighbour) {
30                 if !descendants.contains_key(&neighbour) {
31                     let descendants_set = bfs(graph, neighbour);
32                     descendants.insert(neighbour,
                       ↪  descendants_set);
33                 }
```

```
                node_neigbours.retain(|&x|
                    !descendants[&neighbour].contains(&x));
            }

        }
        for neighbour in node_neigbours {
            transitive_reducted_graph.add_edge(node, neighbour,
                ());
        }
    }

    transitive_reducted_graph
}

```

Fragment w main:

```

    let graph = create_dependency_graph(&word, &D);
    let graph = transitive_reduction(&graph);
    let dot_format = Dot::with_config(&graph,
    &[Config::EdgeNoLabel]);
    let dot_format = format!("{:?}", dot_format);


    let mut file = File::create("graph.dot").unwrap();
    write!(file, "{}", dot_format).unwrap();
    drop(file); //fixes problem with graph.dot not being present
    in the next step

    match Command::new("dot")
        .args(&["-Tpng", "graph.dot", "-o", "graph.png"])
        .output()
        {
            Ok(_) =>
                println!("Graf został wygenerowany do pliku graph.png"),
            Err(_) => println!("Nie udało się wygenerować grafu")
        }


```

3 Wyniki dla podanych danych

3.1 data1.txt

Dane wejściowe

```
4
x <= x + y
y <= y + 2z
x <= 3x + z
z <= y - z
abcd
baadcb
```

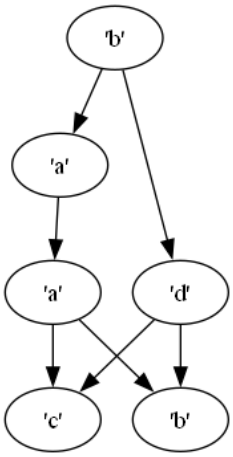
Dane wyjściowe

```
D = {('d', 'c'), ('a', 'a'), ('d', 'd'), ('b', 'd'), ('b', 'b'),
↪  ('a', 'c'), ('c', 'c'), ('a', 'b'), ('c', 'a'), ('c', 'd'),
↪  ('b', 'a'), ('d', 'b')}
I = {('c', 'b'), ('d', 'a'), ('a', 'd'), ('b', 'c')}
FNF:
Zbiór nr 1: {'b'}
Zbiór nr 2: {'a', 'd'}
Zbiór nr 3: {'a'}
Zbiór nr 4: {'c', 'b'}
```

Postać grafu

```
digraph {
    0 [ label = "'b'" ]
    1 [ label = "'a'" ]
    2 [ label = "'a'" ]
    3 [ label = "'d'" ]
    4 [ label = "'c'" ]
    5 [ label = "'b'" ]
    0 -> 3 [ ]
    0 -> 1 [ ]
    1 -> 2 [ ]
    2 -> 5 [ ]
    2 -> 4 [ ]
    3 -> 4 [ ]
    3 -> 5 [ ]
}
```

Graf



3.2 data2.txt

Dane wejściowe

```
6
x <= x + 1
y <= y + 2z
x <= 3x + z
w <= w + v
z <= y - z
v <= x + v
abcdef
acdcfbbe
```

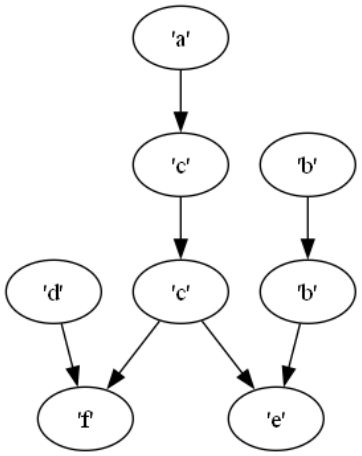
Dane wyjściowe

```
D = {('b', 'b'), ('f', 'a'), ('c', 'c'), ('e', 'c'), ('f', 'c'),
↪  ('b', 'e'), ('c', 'f'), ('f', 'd'), ('e', 'b'), ('d', 'd'),
↪  ('f', 'f'), ('c', 'a'), ('a', 'f'), ('e', 'e'), ('a', 'c'),
↪  ('d', 'f'), ('a', 'a'), ('c', 'e')}
I = {('e', 'a'), ('b', 'c'), ('f', 'b'), ('d', 'e'), ('d', 'c'),
↪  ('e', 'f'), ('e', 'd'), ('b', 'a'), ('d', 'a'), ('f', 'e'),
↪  ('d', 'b'), ('a', 'd'), ('b', 'd'), ('c', 'b'), ('c', 'd'),
↪  ('a', 'e'), ('a', 'b'), ('b', 'f')}
FNF:
Zbiór nr 1: {'b', 'd', 'a'}
Zbiór nr 2: {'c', 'b'}
Zbiór nr 3: {'c'}
Zbiór nr 4: {'e', 'f'}
```

Postać grafu

```
digraph {
    0 [ label = "'a'" ]
    1 [ label = "'c'" ]
    2 [ label = "'d'" ]
    3 [ label = "'c'" ]
    4 [ label = "'f'" ]
    5 [ label = "'b'" ]
    6 [ label = "'b'" ]
    7 [ label = "'e'" ]
    0 -> 1 [ ]
    1 -> 3 [ ]
    2 -> 4 [ ]
    3 -> 7 [ ]
    3 -> 4 [ ]
    5 -> 6 [ ]
    6 -> 7 [ ]
}
```

Graf



3.3 data3.txt

Dane wejściowe

```
6
x <= x + y
y <= z - v
z <= v * x
v <= x + 2y
x <= 3y + 2x
v <= v - 2z
```

```
abcdef
afaeffbcd
```

Dane wyjściowe

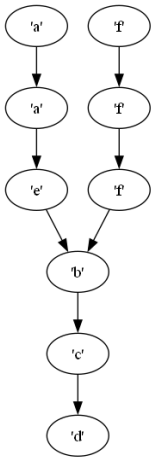
```
D = {('b', 'c'), ('d', 'c'), ('c', 'd'), ('e', 'b'), ('e', 'd'),
↪  ('e', 'c'), ('e', 'e'), ('e', 'a'), ('c', 'a'), ('d', 'd'),
↪  ('f', 'b'), ('f', 'f'), ('b', 'b'), ('f', 'c'), ('d', 'a'),
↪  ('b', 'a'), ('a', 'e'), ('d', 'e'), ('b', 'e'), ('a', 'a'),
↪  ('d', 'b'), ('b', 'f'), ('a', 'c'), ('c', 'b'), ('c', 'f'),
↪  ('b', 'd'), ('d', 'f'), ('c', 'e'), ('f', 'd'), ('c', 'c'),
↪  ('a', 'd'), ('a', 'b')}
I = {('f', 'e'), ('e', 'f'), ('f', 'a'), ('a', 'f')}
FNF:
Zbiór nr 1: {'f', 'a'}
Zbiór nr 2: {'f', 'a'}
Zbiór nr 3: {'e', 'f'}
Zbiór nr 4: {'b'}
Zbiór nr 5: {'c'}
Zbiór nr 6: {'d'}
```

Postać grafu

```
digraph {
    0 [ label = "'a'" ]
    1 [ label = "'f'" ]
    2 [ label = "'a'" ]
    3 [ label = "'e'" ]
    4 [ label = "'f'" ]
    5 [ label = "'f'" ]
    6 [ label = "'b'" ]
    7 [ label = "'c'" ]
    8 [ label = "'d'" ]
    0 -> 2 [ ]
    1 -> 4 [ ]
    2 -> 3 [ ]
    3 -> 6 [ ]
    4 -> 5 [ ]
    5 -> 6 [ ]
    6 -> 7 [ ]
    7 -> 8 [ ]
}

```

Graf



4 Kod źródłowy

4.1 main.rs

```rust
#![allow(non_snake_case)]
#![allow(unused)]

#[cfg(test)]
mod tests;


use std::collections::HashSet;
use std::collections::HashMap;
use std::fs;
use std::fs::File;
use std::io::Write;
use std::process::Command;
use petgraph::graph::{DiGraph, NodeIndex};
use petgraph::dot::{Dot, Config};

fn main(){
    //pobrania nazwy pliku z argumentów
    let args = std::env::args().collect::<Vec<String>>();
    let filename = if args.len() > 1 {
        args[1].clone()
    } else {
        String::new()
    };
    //odczyt z pliku
```

```
26      let (n, transformations, alphabet, word) =
        ↪  read_from_file(filename.to_string());
27
28      //przygotowywanie danych do obliczeń
29      let  variables = find_variables(&transformations);
30      let mut transformations_with_variables: Vec<HashMap<char,
        ↪  VariableSituation>> = create_matrix(&variables, n);
31      fill_matrix_with_variables_status(&transformations, &mut
        ↪  transformations_with_variables);
32
33      //tworzenie zbiorów D i I i zapis do pliku
34      let (D, I) = create_sets(&transformations_with_variables,
        ↪  &alphabet);
35      println!("Zbiór D: {:?}", D);
36      println!("Zbiór I: {:?}", I);
37      let D_string = format!("{:?}", D);
38      let I_string = format!("{:?}", I);
39      let mut file = File::create("result.txt").unwrap();
40      write!(file, "D = {} \nI = {} \nFNF:", D_string,
        ↪  I_string).unwrap();
41
42      //tworzenie FNF i zapis do pliku
43      let foata = create_foata_normal_form(&word, &I, &alphabet);
44      println!("Zbiory Foaty: ");
45      for (i, set) in foata.iter().enumerate() {
46          println!("Zbiór nr {}: {:?}", i+1, set);
47          write!(file, "\nZbiór nr {}: {:?}", i+1, set).unwrap();
48      }
49
50      drop(file);
51
        ↪  println!("Zbiory i postać Foaty zostały dodane do result.txt");
52
53      let graph = create_dependency_graph(&word, &D);
54      let graph = transitive_reduction(&graph);
55      let dot_format = Dot::with_config(&graph,
        ↪  &[Config::EdgeNoLabel]);
56      let dot_format = format!("{:?}", dot_format);
57
58
59      let mut file = File::create("graph.dot").unwrap();
60      write!(file, "{}", dot_format).unwrap();
61      drop(file); //fixes problem with graph.dot not being present
        ↪  in the next step
62
63      match Command::new("dot")
```

```
64          .args(&["-Tpng", "graph.dot", "-o", "graph.png"])
65          .output()
66          {
67              Ok(_) =>
                ↪  println!("Graf został wygenerowany do pliku graph.png"),
68              Err(_) => println!("Nie udało się wygenerować grafu")
69          }
70
71
72  }
73
74
75  fn create_dependency_graph(word: &String, D:
    ↪  &HashSet<(char,char)>) -> DiGraph<char, ()> {
76      let mut graph = DiGraph::new();
77      for c in word.chars() {
78          graph.add_node(c);
79      }
80      for node in graph.node_indices() {
81          for other_node in graph.node_indices() {
82              if node >= other_node {
83                  continue;
84              }
85              if D.contains(&(graph[node], graph[other_node])) {
86                  graph.add_edge(node, other_node, ());
87              }
88          }
89      }
90      graph
91  }
92
93  fn transitive_reduction(graph: &DiGraph<char, ()>) ->
    ↪  DiGraph<char, ()> {
94      let mut transitive_reducted_graph = DiGraph::new();
95      for node in graph.node_indices() {
96          transitive_reducted_graph.add_node(graph[node]);
97      }
98      let mut descendants = HashMap::new();
99      for node in graph.node_indices() {
100         let mut node_neigbours: HashSet<NodeIndex> =
            ↪  graph.neighbors(node).collect();
101         for neighbour in node_neigbours.clone() {
102             if node_neigbours.contains(&neighbour) {
103                 if !descendants.contains_key(&neighbour) {
104                     let descendants_set = bfs(graph, neighbour);
```

```
105                     descendants.insert(neighbour,
                        ↪  descendants_set);
106                 }
107                 node_neigbours.retain(|&x|
                    ↪  !descendants[&neighbour].contains(&x));
108             }
109
110         }
111         for neighbour in node_neigbours {
112             transitive_reducted_graph.add_edge(node, neighbour,
                    ↪  ());
113         }
114     }
115
116     transitive_reducted_graph
117 }
118
119 fn bfs(graph: &DiGraph<char, ()>, node:
    ↪  petgraph::graph::NodeIndex) ->
    ↪  HashSet<petgraph::graph::NodeIndex> {
120     let mut visited = HashSet::new();
121     let mut queue = Vec::new();
122     queue.push(node);
123     while !queue.is_empty() {
124         let current_node = queue.pop().unwrap();
125         if !(current_node == node) {
126             visited.insert(current_node);
127         }
128         for neighbour in graph.neighbors(current_node) {
129             if !visited.contains(&neighbour) {
130                 queue.push(neighbour);
131             }
132         }
133     }
134     visited
135
136 }
137
138
139
140 #[doc =
    ↪  "Funkcja odpowiada za wyznaczenie zbiorów Foaty. Korzystam z algorytmu z kopcami z ksią
141 fn create_foata_normal_form(word: &String, I:
    ↪  &HashSet<(char,char)>,  alphabet: &Vec<char>) ->
    ↪  Vec<HashSet<char>> {
142     let mut foata: Vec<HashSet<char>> = Vec::new();
```

```rust
    let mut stacks: HashMap<char, Vec<char>> =
    ↪  alphabet.iter().map(|&c| (c, Vec::new())).collect();
    fill_stacks(word, &mut stacks, I);
    fill_foata(&mut stacks, &mut foata, &I, &alphabet);
    foata
}


fn fill_foata(stacks: &mut HashMap<char, Vec<char>>, foata: &mut
↪  Vec<HashSet<char>>, I: &HashSet<(char, char)>, alphabet:
↪  &Vec<char>) {
    let mut to_be_popped = Vec::new();
    while !stacks.values().all(|stack| stack.is_empty()) {
        let mut set = HashSet::new();
        for element in &to_be_popped {
            let stack = stacks.get_mut(element).unwrap();
            stack.pop();
        }
        to_be_popped.clear();
        for stack_el in stacks.iter_mut() {
            let (_letter, stack) = stack_el;
            let current_sign = *stack.last().unwrap_or(&'*'); //
            ↪  '*' has here second meaning, it's a sign that
            ↪  stack is empty (normally is a special stack sign)
            if current_sign != '*' {
                set.insert(current_sign);
                stack.pop();
                for alphabet_letter in alphabet {
                    if !I.contains(&(current_sign,
                    ↪  *alphabet_letter)) {
                        if current_sign == *alphabet_letter {
                            continue;
                        }
                            to_be_popped.push(*alphabet_letter);
                        }
                    }
                }
            }

        if set.is_empty() {
            for stack_el in stacks.iter_mut() {
                let (letter, stack) = stack_el;
                stack.pop();
            }
        } else {
            foata.push(set);
```

```
183             }
184         }
185         }
186
187
188     fn fill_stacks(word: &String, stacks: &mut HashMap<char,
        ↪  Vec<char>>, I: &HashSet<(char, char)>) {
189         for c in word.chars().rev() {
190             for stack_el in stacks.iter_mut() {
191                 let (letter, stack) = stack_el;
192                 if !I.contains(&(c, *letter)) {
193                     if c == *letter {
194                         stack.push(c);
195                     } else {
196                         stack.push('*');
197                     }
198                 }
199
200             }
201
202         }
203     }
204
205
206     fn read_file_name() -> String {
207         println!("Podaj nazwę pliku z danymi: ");
208         let mut filename = String::new();
209         std::io::stdin().read_line(&mut
            ↪  filename).expect("Failed to read filename");
210         let filename = filename.trim();
211         filename.to_string()
212     }
213     fn read_from_file(mut filename: String) ->(i32, Vec<String>,
        ↪  Vec<char>, String) {
214         if filename.is_empty() {
215             filename = read_file_name();
216         }
217         let content =
            ↪  fs::read_to_string(filename).expect("Failed to read file");
218         let mut lines = content.lines();
219         let n: i32 =
            ↪  lines.next().expect("Failed to read n").parse().expect("Failed to parse n");
220         let mut transformations = Vec::new();
221         for _ in 0..n {
222
                ↪  transformations.push(lines.next().expect("Failed to read transformation").to_str
```

```
223     }
224     let alphabet =
        ↪  parse_alphabet(&lines.next().expect("Failed to read alphabet").to_string());
225     let word = lines.next().expect("Failed to read word");
226     (n, transformations, alphabet, word.to_string())
227 }
228
229
230 fn read_from_console() ->(i32, Vec<String>, Vec<char>) {
231     println!("Podaj liczbę równań, które chcesz wprowadzić: ");
232     let mut n = String::new();
233     std::io::stdin().read_line(&mut n).unwrap();
234     let n: i32 = match n.trim().parse() {
235         Ok(num) => num,
236         Err(_) => panic!("This must be a number!"),
237     };
238     let transformations = read_transformations(n);
239     let alphabet = get_alphabet_from_input();
240     if alphabet.len() != n as usize {
241
            ↪  panic!("Alphabet size has to be equal to number of transformations!");
242     }
243     (n, transformations, alphabet)
244 }
245
246 fn get_alphabet_from_input() -> Vec<char> {
247     println!("Podaj alfabet: ");
248     let mut alphabet = String::new();
249     std::io::stdin().read_line(&mut
        ↪  alphabet).expect("Failed to read alphabet");
250     let alphabet: Vec<char> = parse_alphabet(&alphabet);
251     alphabet
252 }
253 fn parse_alphabet(alphabet: &String) -> Vec<char> {
254     let alphabet: Vec<char> = alphabet.chars().filter(|&c|
        ↪  c.is_alphanumeric()).collect();
255     alphabet
256
257
258 }
259
260 fn create_sets(transformations_with_variables: &Vec<HashMap<char,
    ↪  VariableSituation>>, alphabet: &Vec<char>) -> (HashSet<(char,
    ↪  char)>, HashSet<(char, char)>) {
261     let mut D = HashSet::new();
262     let mut I = HashSet::new();
```

```
263     let mut has_been_added_to_D = false;
264     let twv_len = transformations_with_variables.len();
265     for i in 0..twv_len {
266         for j in i..twv_len{
267             if i == j {
268                 D.insert((alphabet[i], alphabet[j]));
269                 continue;
270             }
271             for (variable, situation_first) in
                ↪  &transformations_with_variables[i] {
272                 match
                    ↪  transformations_with_variables[j].get(variable)
                    ↪  {
273                     Some(situation_second) => {
274                         if
                            ↪  situation_first.is_depend(situation_second)
                            ↪  {
275                             D.insert((alphabet[i], alphabet[j]));
276                             D.insert((alphabet[j], alphabet[i]));
277                             has_been_added_to_D = true;
278                             break;
279                         }
280                     },
281                     None =>
                        ↪  panic!("Variable not found in hashmap")
282                 }
283             }
284             if !has_been_added_to_D {
285                 I.insert((alphabet[i], alphabet[j]));
286                 I.insert((alphabet[j], alphabet[i]));
287             }
288             has_been_added_to_D = false;
289     }
290
291     }
292     (D, I)
293 }
294 fn read_transformations(n: i32) -> Vec<String> {
295     let mut transformations: Vec<String> = Vec::new();
296     for i in 0..n {
297         println!("Podaj równanie nr {}: ", i+1);
298         let mut equation = String::new();
299         std::io::stdin().read_line(&mut
            ↪  equation).expect("Failed to read line");
300         transformations.push(equation);
301     }
```

```
302     transformations
303 }
304 
305 fn find_variables(transformations: &Vec<String>) -> HashSet<char>
    ↪ {
306     let mut variables:HashSet<char> = HashSet::new();
307     for transformation in transformations {
308         for c in transformation.chars() {
309             if c.is_alphabetic() {
310                 variables.insert(c);
311             }
312         }
313     }
314     variables
315 }
316 
317 fn create_matrix(variables: &HashSet<char>, n: i32) ->
    ↪ Vec<HashMap<char, VariableSituation>> {
318     let mut transformation_with_variables: Vec<HashMap<char,
        ↪ VariableSituation>> = Vec::new();
319     for _ in 0..n {
320         transformation_with_variables.push(HashMap::new());
321         for variable in variables {
322 
                ↪ transformation_with_variables.last_mut().expect("Error creating matrix!").in
                ↪ VariableSituation::Neither);
323         }
324     }
325     transformation_with_variables
326 }
327 
328 fn fill_matrix_with_variables_status(transformations:
    ↪ &Vec<String>, transformations_with_variables: &mut
    ↪ Vec<HashMap<char, VariableSituation>>)  {
329     for (i, transformation) in transformations.iter().enumerate()
        ↪ {
330         let mut last_char = ' ';
331         let mut current_site = CurrentSite::Left;
332         for c in transformation.chars() {
333             if last_char == '<' && c == '=' {
334                 current_site = CurrentSite::Right;
335             }
336             last_char = c;
337 
338             if c.is_alphabetic() {
```

```
339                 match
                    ↪  transformations_with_variables[i].get_mut(&c)
                    ↪  {
340                     Some(situation) => {
341                         match situation {
342                             VariableSituation::Neither => {
343                                 match current_site {
344                                     CurrentSite::Left => {
345                                         *situation =
                                            ↪  VariableSituation::Left;
346                                     },
347                                     CurrentSite::Right => {
348                                         *situation =
                                            ↪  VariableSituation::Right;
349                                     }
350                                 }
351                             },
352                             VariableSituation::Left => {
353                                 match current_site {
354                                     CurrentSite::Left => {
355                                         *situation =
                                            ↪  VariableSituation::Left;
356                                     },
357                                     CurrentSite::Right => {
358                                         *situation =
                                            ↪  VariableSituation::Both;
359                                     }
360
361                                 }
362                                 *situation =
                                    ↪  VariableSituation::Both;
363                             },
364                             VariableSituation::Right => {
365                                 //*situation =
                                    ↪  VariableSituation::Both;
366                             },
367                             VariableSituation::Both => {
368                                 //*situation =
                                    ↪  VariableSituation::Both;
369                             }
370                         }
371                     },
372                     None => {
373                         panic!("Variable not found in hashmap");
374                     }
375                 }
```

```
            }
        }
    }
}

#[derive(Debug)]
#[derive(PartialEq)]
enum VariableSituation {
    Left,
    Right,
    Both,
    Neither
}

impl VariableSituation {
    fn is_depend(&self, other:  &VariableSituation) -> bool {
        match (self, other) {
            (VariableSituation::Left | VariableSituation::Both,
            ↪  other)  if *other != VariableSituation::Neither
            ↪  => true,
            (VariableSituation::Right, VariableSituation::Left |
            ↪  VariableSituation::Both) => true,
            (_, _) => false
        }
    }
}

enum CurrentSite {
    Left,
    Right
}
```

4.2 test.rs

```

use super::*;


#[doc =
↪  "Test sprawdzający wszystkie możliwe przypadki zależności zmiennych"]
#[test]
fn test_is_depend() {

    ↪  assert_eq!(VariableSituation::Left.is_depend(&VariableSituation::Right),
    ↪  true);
```

```
9
        ↪ assert_eq!(VariableSituation::Left.is_depend(&VariableSituation::Left),
        ↪ true);
10
        ↪ assert_eq!(VariableSituation::Left.is_depend(&VariableSituation::Both),
        ↪ true);
11
        ↪ assert_eq!(VariableSituation::Left.is_depend(&VariableSituation::Neither),
        ↪ false);
12
        ↪ assert_eq!(VariableSituation::Right.is_depend(&VariableSituation::Left),
        ↪ true);
13
        ↪ assert_eq!(VariableSituation::Right.is_depend(&VariableSituation::Right),
        ↪ false);
14
        ↪ assert_eq!(VariableSituation::Right.is_depend(&VariableSituation::Both),
        ↪ true);
15
        ↪ assert_eq!(VariableSituation::Right.is_depend(&VariableSituation::Neither),
        ↪ false);
16
        ↪ assert_eq!(VariableSituation::Both.is_depend(&VariableSituation::Left),
        ↪ true);
17
        ↪ assert_eq!(VariableSituation::Both.is_depend(&VariableSituation::Right),
        ↪ true);
18
        ↪ assert_eq!(VariableSituation::Both.is_depend(&VariableSituation::Both),
        ↪ true);
19
        ↪ assert_eq!(VariableSituation::Both.is_depend(&VariableSituation::Neither),
        ↪ false);
20
        ↪ assert_eq!(VariableSituation::Neither.is_depend(&VariableSituation::Left),
        ↪ false);
21
        ↪ assert_eq!(VariableSituation::Neither.is_depend(&VariableSituation::Right),
        ↪ false);
22
        ↪ assert_eq!(VariableSituation::Neither.is_depend(&VariableSituation::Both),
        ↪ false);
23
        ↪ assert_eq!(VariableSituation::Neither.is_depend(&VariableSituation::Neither),
        ↪ false);
24  }
```

```rust

#[doc = "Test sprawdzający czy funkcja read_from_file zwraca poprawne dane"]
#[test]
fn test_read_from_file() {
    let (n, transformations, alphabet,word ) = read_from_file("data1.txt".to_string());
    assert_eq!(n, 4);
    assert_eq!(transformations, vec!["x <= x+y", "y <= y+2z", "x <= 3x+z", "z <= y-z"]);
    assert_eq!(alphabet, vec!['a', 'b', 'c', 'd']);
    assert_eq!(word, "baadcb".to_string());
}
```

Literatura

Volker Diekert and Yves Métivier. *Partial Commutation and Traces*, page 465. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997. ISBN 978-3-642-59126-6. doi: 10.1007/978-3-642-59126-6_8. URL https://doi.org/10.1007/978-3-642-59126-6_8.

NetworkX. Implementacja transitive_reduction w bibliotece networkx. URL https://networkx.org/documentation/stable/_modules/networkx/algorithms/dag.html#transitive_reduction.