Teoria Współbieżności
Zadanie domowe 2

Piotr Kuchta

20 listopada 2023

1 Uruchamianie

Aby przetestować program na obu plikach testowych wystarczy uruchomić `./run.sh`.

2 Plik `file_reader.py`

Plik zawiera 2 funkcje, przy czym interesuje nas najbardziej funkcja `read_file`

2.1 Funkcja `read_file`

Funkcja czyta pliki zawierające dane w formacie:
znaki alfabetu oddzielone przecinkiem
analizowane słowo
produkcje, każda w osobnych liniach
Przykład:
a,b,c,d
baadcb
(a) x := x + y
(b) y := y + 2z
(c) x := 3x + z
(d) z := y - z

Zwraca krotkę składającą się z listy symboli, listy przedstawiającej słowo
oraz objekt $typeRelationGraph = Dict[str, List[str]]$

```
def read_file(path: str) -> Tuple[List[str],List[str],RelationGraph]:
  if (not os.path.isfile(path)):
    raise ValueError("No such file.")
  with open(path,"r") as file:
    symbols = file.readline()[:-1].split(",")
    word = file.readline()[:-1]
    line = file.readline()
    all_expressions = []
    while line:
      line = line[:-1]
      span = re.search(r"\(.+\)",line).span()
      expression = []
      expression.append(line[span[0]+1:span[1]-1])
      expression.append(line[span[1]+1:])
      expression[1] = "".join(expression[1].split())
      tmp = expression[1].split(":=")
      expression[1] = tmp[0]
      expression.append(tmp[1])
      expression = tuple(expression)
      all_expressions.append(expression)
      line = file.readline()
  graph = create_relation_graph(all_expressions)
  return symbols, word, graph
```

2.2 Funkcja `create_relation_graph`

Funkcja tworzy graf relacji pomiędzy symbolami. Graf reprezentowany jest przez słownik, którego kluczami są symbole alfabetu, a wartościami: listy zawierające symbole które są w relacji z kluczami

```
def read_file(path: str) -> Tuple[List[str],List[str],RelationGraph]:
  if (not os.path.isfile(path)):
    raise ValueError("No such file.")
  with open(path,"r") as file:
    symbols = file.readline()[:-1].split(",")
    word = file.readline()[:-1]
    line = file.readline()
    all_expressions = []
    while line:
      line = line[:-1]
      span = re.search(r"\(.+\)",line).span()
      expression = []
      expression.append(line[span[0]+1:span[1]-1])
      expression.append(line[span[1]+1:])
      expression[1] = "".join(expression[1].split())
      tmp = expression[1].split(":=")
      expression[1] = tmp[0]
      expression.append(tmp[1])
      expression = tuple(expression)
      all_expressions.append(expression)
      line = file.readline()
  graph = create_relation_graph(all_expressions)
  return symbols, word, graph
```

3 Plik `graph_node.py`

W pliku zdefiniowana jest klasa reprezentująca wierzchołek grafu.

```
class GraphNode:

  def __init__(self,id: int, symbol: str, edges: List[int]):
    self.id = id
    self.symbol = symbol
    self.visited = False
    self.edges = edges
    self.current_path = []
    self.time = -1

  def __repr__(self):
    return f"id={self.id}, visited={self.visited}, edges={self.edges}, path={self.current_path}"
```

4 Plik `main.py`

W pliku zdefiniowane są funkcje działające na grafach oraz przetwarzające dane do odpowiedniej formy czytelnej dla ludzi.

4.1 Funkcja `get_relations`

Funkcja, która na podstawie grafu relacji pomiędzy symbolami oraz listy symboli tworzy tablicę wypełnioną relacjami zależności oraz tablicę relacji niezależności.

```
def get_relations(symbols: List[str], relation_graph: Dict[str,List[str]]) -> Tuple[List[Tuple],
    List[Tuple]]:
  in_rel = []
  not_rel = []
  for s1 in symbols:
    for s2 in symbols:
      if s2 in relation_graph[s1]:
        in_rel.append(tuple([s1,s2]))
      else:
        not_rel.append(tuple([s1,s2]))
  in_rel.sort()
  not_rel.sort()
  return in_rel, not_rel
```

4.2 Funkcja `to_FNF`

Funkcja zwracająca postać normalną Foaty na podstawie grafu skierowanego w którym każdy wierzchołek ma zapisany czas odwiedzin czasem odwiedzin.

```
def to_FNF(graph: DirectedGraph) -> str:
  time = 0
  for v in graph:
    time = max(time,v.time)
  divided = [[] for i in range(time+1)]
  for v in graph:
    divided[v.time].append(v.symbol)
  result_string = ""
  for packet in divided:
    result_string = f"{result_string}({"".join(packet)})"
  return result_string
```

4.3 Funkcja `to_Digraph`

Funkcja tworząca obiekt graphviz.Diagraph umożliwiający łatwe generowanie zapisu grafu w postaci dot.

```
def to_Diagraph(graph: DirectedGraph, name: str) -> graphviz.Digraph:
  dot = graphviz.Digraph(name)
  for v in graph:
    for new_v in v.edges:
      dot.edge(str(v.id),str(new_v))
  for v in graph:
    dot.node(str(v.id),v.symbol)
  return dot
```

4.4 Funkcja `create_directed_graph`

Na podstawie grafu relacji pomiędzy symbolami oraz słowa utworzonego z tych symboli, tworzy graf skierowany.

```
def create_directed_graph(word: str, relation_graph: RelationGraph) -> DirectedGraph:
  directed = []
  for i,letter in enumerate(word):
    node = GraphNode(i,letter,[])
    for j in range(i+1,len(word)):
      letter2 = word[j]
      if letter2 in relation_graph[letter]:
        node.edges.append(j)
    directed.append(node)
  return directed
```

4.5 Funkcja `discarding_DFS`

Funkcja realizująca przeszukanie grafu za pomocą DFSa, a następnie odrzucająca niepotrzebne krawędzie. Każdy wierzchołek A przechowuje w polu current_path wszystkie wierzchołki do których można utworzyć ścieżkę z wierzchołka A. Następnie dla każdego wierzchołka B, funkcja sprawdza czy wierzchołki których krawędzie prowadzą do B należą do current_path innych wierzchołków prowadzących do B. Jeśli z wierzchołka C prowadzącego do B da się dojść do innego wierzchołka prowadzącego do B to krawędź (C,B) jest zbędna.

```
def discarding_DFS(graph: DirectedGraph):
  for node in graph:
    sub_DFS(graph, node)
  for node in graph:
    node.visited = False
  for node in graph:
    nodes_to_check: List[GraphNode] = []
    for node2 in graph:
      if node.id in node2.edges:
        nodes_to_check.append(node2)
    for node_check1 in nodes_to_check:
      for node_check2 in nodes_to_check:
        if node_check1 == node_check2:
          continue
        if node_check1 in node_check2.current_path and node.id in node_check2.edges:
          node_check2.edges.remove(node.id)
```

4.6 Funkcja `sub_DFS`

Funkcja pomocnicza realizująca sam podstawowy algorytm DFS. Wypełnia dodatkowo listę w polu current_path każdego wierzchołka

```
def sub_DFS(graph: DirectedGraph, node: GraphNode) -> List[GraphNode]:
  if node.visited:
    tmp = node.current_path.copy()
    tmp.append(node)
    return tmp
  possible_paths = []
  for new_verticle in node.edges:
    node_tmp = graph[new_verticle]
    possible_paths.extend(sub_DFS(graph, node_tmp))
  node.current_path = possible_paths
  node.visited = True
  tmp = node.current_path.copy()
  tmp.append(node)
  return tmp
```

4.7 Funkcja `BFS`

Funkcja realizująca algorytm BFS, który odwiedza także już odwiedzone wierzchołki.

```
def BFS(graph: DirectedGraph):
  for v in graph:
    if not v.visited:
      q = queue.Queue()
      q.put(v)
      v.time = 0
      while not q.empty():
        current = q.get()
        for verticle in current.edges:
          node = graph[verticle]
          node.visited = True
          node.time = current.time+1
          q.put(node)
```

4.8 Główny kod

Program domyślnie czyta wejście z pliku `test1.txt`. Można podać nazwę pliku jako argument wywołania by wybrać inny plik.

```
if __name__ == "__main__":
  if len(sys.argv) == 1:
    name = "test1.txt"
  else:
    name = sys.argv[1]
  symbols, word, relation_graph = read_file(name)
  in_rel,not_rel = get_relations(symbols, relation_graph)
  directed_graph = create_directed_graph(word,relation_graph)
  discarding_DFS(directed_graph)
  BFS(directed_graph)
  print(f"D={in_rel}")
  print(f"I={not_rel}")
  print(f"Foata: {to_FNF(directed_graph)}")
  dot = to_Diagraph(directed_graph,name.split(".")[0])
  dot.render(directory='graph_output')
  print(dot.source)
```

5 Przykładowe wyniki

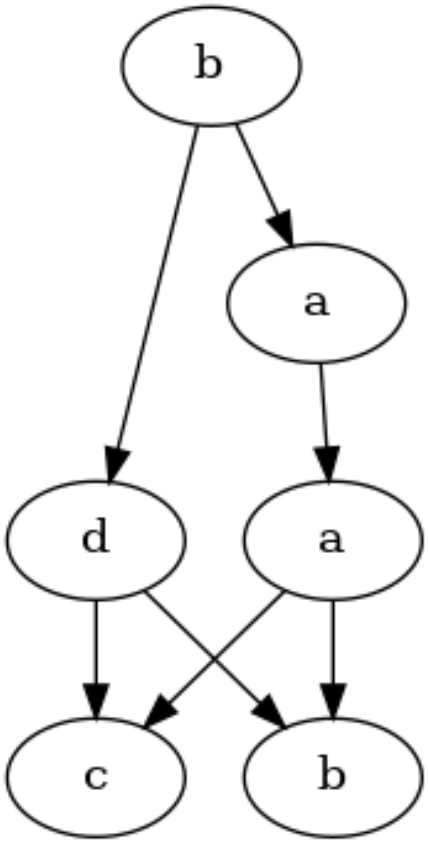
5.1 Plik `test1.txt`

Dane wejściowe:
a, b, c, d
$baadcb$
$(a)x := x + y$
$(b)y := y + 2z$

$(c)x := 3x + z$
$(d)z := y - z$

Wyniki:
$D = [('a','a'), ('a','b'), ('a','c'), ('b','a'), ('b','b'), ('b','d'), ('c','a'), ('c','c'), ('c','d'), ('d','b'), ('d','c')]$
$I = [('a','d'), ('b','c'), ('c','b'), ('d','a'), ('d','d')]$
$Foata : (b)(ad)(a)(cb)$

```
$digraph test1 {
        0 -> 1
        0 -> 3
        1 -> 2
        2 -> 4
        2 -> 5
        3 -> 4
        3 -> 5
        0 [label=b]
        1 [label=a]
        2 [label=a]
        3 [label=d]
        4 [label=c]
        5 [label=b]
}$
```



Rysunek 1: Wynikowy graf skierowany

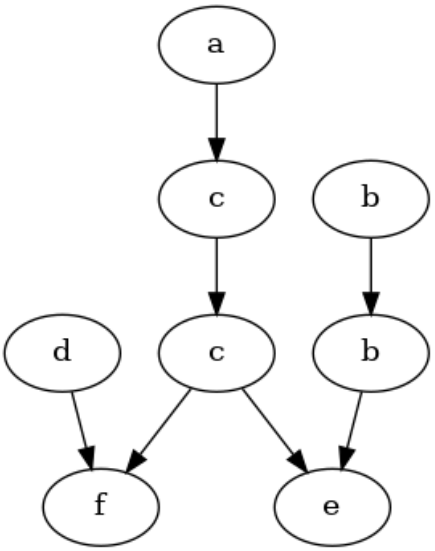
5.2 Plik `test1.txt`

Dane wejściowe:
a, b, c, d, e, f
$acdcfbbe$
$(a)x := x + 1$
$(b)y := y + 2z$
$(c)x := 3x + z$

$(d) w := w + v$
$(e) z := y - z$
$(f) v := x + v$

Wyniki:
$D = [('a','a'), ('a','c'), ('a','f'), ('b','b'), ('b','e'), ('c','a'), ('c','c'), ('c','e'), ('c','f'),$
$('d','d'), ('d','f'), ('e','b'), ('e','c'), ('e','e'), ('f','a'), ('f','c'), ('f','d')]$
$I = [('a','b'), ('a','d'), ('a','e'), ('b','a'), ('b','c'), ('b','d'), ('b','f'), ('c','b'), ('c','d'),$
$('d','a'), ('d','b'), ('d','c'), ('d','e'), ('e','a'), ('e','d'), ('e','f'), ('f','b'), ('f','e'), ('f','f')]$
$Foata : (adb)(cfb)(ce)$

```
$digraph test2 {
        0 -> 1
        1 -> 3
        2 -> 4
        3 -> 4
        3 -> 7
        5 -> 6
        6 -> 7
        0 [label=a]
        1 [label=c]
        2 [label=d]
        3 [label=c]
        4 [label=f]
        5 [label=b]
        6 [label=b]
        7 [label=e]
}$
```



Rysunek 2: Wynikowy graf skierowany