

Teoria Współbierzości

Zadanie domowe 2

Piotr Kuchta

16 listopada 2023

1 Plik `file_reader.py`

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

1.1 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 obiekt `typeRelationGraph = Dict[str, List[str]]`

```
1 def read_file(path: str) -> Tuple[List[str], List[str], RelationGraph]:
2     if (not os.path.isfile(path)):
3         raise ValueError("No such file.")
4     with open(path, "r") as file:
5         symbols = file.readline()[:-1].split(",")
6         word = file.readline()[:-1]
7         line = file.readline()
8         all_expressions = []
9         while line:
10             line = line[:-1]
11             span = re.search(r"\(.(+)\)", line).span()
12             expression = []
13             expression.append(line[span[0]+1:span[1]-1])
14             expression.append(line[span[1]+1:])
15             expression[1] = " ".join(expression[1].split())
16             tmp = expression[1].split(":=")
17             expression[1] = tmp[0]
18             expression.append(tmp[1])
19             expression = tuple(expression)
20             all_expressions.append(expression)
21             line = file.readline()
22     graph = create_relation_graph(all_expressions)
23     return symbols, word, graph
```

1.2 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

```
1 def read_file(path: str) -> Tuple[List[str], List[str], RelationGraph]:
2     if (not os.path.isfile(path)):
3         raise ValueError("No such file.")
```

```

4 with open(path,"r") as file:
5     symbols = file.readline()[::-1].split(",")
6     word = file.readline()[::-1]
7     line = file.readline()
8     all_expressions = []
9     while line:
10         line = line[::-1]
11         span = re.search(r"\(.\+\)",line).span()
12         expression = []
13         expression.append(line[span[0]+1:span[1]-1])
14         expression.append(line[span[1]+1:])
15         expression[1] = " ".join(expression[1].split())
16         tmp = expression[1].split(":=")
17         expression[1] = tmp[0]
18         expression.append(tmp[1])
19         expression = tuple(expression)
20         all_expressions.append(expression)
21         line = file.readline()
22 graph = create_relation_graph(all_expressions)
23 return symbols, word, graph

```

2 Plik graph_node.py

W pliku zdefiniowana jest klasa reprezentująca wierzchołek grafu.

```

1 class GraphNode:
2
3     def __init__(self,id: int, symbol: str, edges: List[int]):
4         self.id = id
5         self.symbol = symbol
6         self.visited = False
7         self.edges = edges
8         self.current_path = []
9         self.time = -1
10
11     def __repr__(self):
12         return f"id={self.id}, visited={self.visited}, edges={self.edges}, path={self.current_path}"

```

3 Plik main.py

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

3.1 Funkcja get_relations

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

```

1 def get_relations(symbols: List[str], relation_graph: Dict[str,List[str]]) -> Tuple[List[Tuple],
2     List[Tuple]]:
3     in_rel = []
4     not_rel = []
5     for s1 in symbols:
6         for s2 in symbols:
7             if s2 in relation_graph[s1]:
8                 in_rel.append(tuple([s1,s2]))
9             else:
10                 not_rel.append(tuple([s1,s2]))
11     in_rel.sort()
12     not_rel.sort()
13     return in_rel, not_rel

```

3.2 Funkcja to_FNF

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

```

1 def to_FNF(graph: DirectedGraph) -> str:
2     time = 0
3     for v in graph:
4         time = max(time, v.time)
5     divided = [[] for i in range(time+1)]
6     for v in graph:
7         divided[v.time].append(v.symbol)
8     result_string = ""
9     for packet in divided:
10        result_string = f"{result_string}({"".join(packet)})"
11    return result_string

```

3.3 Funkcja to_Digraph

Funkcja tworząca obiekt graphviz.Digraph umożliwiające łatwe generowanie zapisu grafu w postaci dot.

```

1 def to_Digraph(graph: DirectedGraph, name: str) -> graphviz.Digraph:
2     dot = graphviz.Digraph(name)
3     for v in graph:
4         for new_v in v.edges:
5             dot.edge(str(v.id), str(new_v))
6     for v in graph:
7         dot.node(str(v.id), v.symbol)
8     return dot

```

3.4 Funkcja create_directed_graph

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

```

1 def create_directed_graph(word: str, relation_graph: RelationGraph) -> DirectedGraph:
2     directed = []
3     for i, letter in enumerate(word):
4         node = GraphNode(i, letter, [])
5         for j in range(i+1, len(word)):
6             letter2 = word[j]
7             if letter2 in relation_graph[letter]:
8                 node.edges.append(j)
9         directed.append(node)
10    return directed

```

3.5 Funkcja discarding_DFS

Funkcja realizująca przeszukiwanie 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.

```

1 def discarding_DFS(graph: DirectedGraph):
2     for node in graph:
3         sub_DFS(graph, node)
4     for node in graph:
5         node.visited = False
6     for node in graph:
7         nodes_to_check: List[GraphNode] = []
8         for node2 in graph:
9             if node.id in node2.edges:
10                nodes_to_check.append(node2)
11        for node_check1 in nodes_to_check:
12            for node_check2 in nodes_to_check:
13                if node_check1 == node_check2:
14                    continue
15                if node_check1 in node_check2.current_path and node.id in node_check2.edges:
16                    node_check2.edges.remove(node.id)

```

3.6 Funkcja sub_DFS

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

```

1 def sub_DFS(graph: DirectedGraph, node: GraphNode) -> List[GraphNode]:
2     if node.visited:
3         tmp = node.current_path.copy()
4         tmp.append(node)
5         return tmp
6     possible_paths = []
7     for new_verticle in node.edges:
8         node_tmp = graph[new_verticle]
9         possible_paths.extend(sub_DFS(graph, node_tmp))
10    node.current_path = possible_paths
11    node.visited = True
12    tmp = node.current_path.copy()
13    tmp.append(node)
14    return tmp

```

3.7 Funkcja BFS

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

```

1 def BFS(graph: DirectedGraph):
2     for v in graph:
3         if not v.visited:
4             q = queue.Queue()
5             q.put(v)
6             v.time = 0
7             while not q.empty():
8                 current = q.get()
9                 for verticle in current.edges:
10                    node = graph[verticle]
11                    node.visited = True
12                    node.time = current.time+1
13                    q.put(node)

```

3.8 Główny kod

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

```

1 if __name__ == "__main__":
2     if len(sys.argv) == 1:
3         name = "test1.txt"
4     else:
5         name = sys.argv[1]
6     symbols, word, relation_graph = read_file(name)
7     in_rel, not_rel = get_relations(symbols, relation_graph)
8     directed_graph = create_directed_graph(word, relation_graph)
9     discarding_DFS(directed_graph)
10    BFS(directed_graph)
11    print(f"D={in_rel}")
12    print(f"I={not_rel}")
13    print(f"Foata: {to_FNF(directed_graph)}")
14    dot = to_Diagraph(directed_graph, name.split(".")[0])
15    dot.render(directory='graph_output')
16    print(dot.source)

```

3.9 Uruchamianie

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