Trace theory

Home assignment

How to run

1. Be in the root directory (`lab06`).
2. Run*:

```
./run.sh
```

**If you cannot run the script, grant an execution permission:*

```
chmod +x ./run.sh
```

Description

This project demonstrates, how concurrent tasks may be grouped by and run on different threads to achive better performance. **It only visualizes the method, doesn't implement it by running something concurrently**.

The program can be specified in 5 major steps (in chronological order):

- Creation of dependency relation (D).
- Creation of independency relation (I).
- Creation of minimal dependency relation DW of input word.
- Computation Foata normal form (FNF) of DW.
- Visualization of DW.

`main.py`

```
import os
import sys
from file_reader import FileReader
from relation import DependencyRelation, IndependencyRelation,
DependencyWordRelation
from normal_form import FoataNF
from drawer import GraphDrawer


if __name__ == "__main__":
  PATH           = os.path.dirname(__file__)
  PATH_DATA      = os.path.join(PATH, "../data")
  PATH_OUTPUT    = os.path.join(PATH, "../output")
```

```
  TEST_FILENAMES = [ "test1", "test2" ]

  test_filename = "test1"
  if len(sys.argv) > 1:
    test_filename = sys.argv[1]

  print(test_filename)

  # Read input file, first line is an alphabet, second one a word, rest
are expressions.
  # The word is also converted to integer list.
  alphabet, word, expressions = FileReader.read(os.path.join(PATH_DATA, f"
{test_filename}.txt"))
  word_int = list(map(lambda letter: ord(letter) - ord('a'), word))

  # Create and print dependency relation (D).
  dependency_relation = DependencyRelation(expressions, alphabet)
  dependency_relation.build()
  print(dependency_relation)

  # Create and print independency relation (I).
  independency_relation = IndependencyRelation(expressions, alphabet)
  independency_relation.build()
  print(independency_relation)

  # Create minimal dependency relation of the input word.
  dependency_relation_word = DependencyWordRelation(word_int,
dependency_relation, minimal=True)
  dependency_relation_word.build()

  # Create and print Foata normal form based on the input word.
  foata_normal_form = FoataNF(word_int, dependency_relation,
independency_relation)
  foata_normal_form.build()
  print(f"FNF: {foata_normal_form}")

  # Visualize results.
  drawer = GraphDrawer(dependency_relation_word.results, word)
  drawer.draw(os.path.join(PATH_OUTPUT, f"{test_filename}.gv"))
```

relation.py

```
class AbstractRelation(ABC):
  set_symbol = "AR"

  def __init__(self, expressions: list[tuple[str, list[str]]], alphabet:
list[str]) -> None:
    self.expressions = expressions
    self.alphabet    = alphabet
    self.results: list[list[int]] | None = None
```

```
    def __bool__(self) -> bool:
      return self.results != None

    def __str__(self) -> str:
      relation_set = f"{self.set_symbol} = {{%s}}"

      if not self.results or len(self.results) == 0:
        return relation_set % " "

      formatted_results: list[str] = []
      for i in range(len(self.alphabet)):
        for j in self.results[i]:
          i_str, j_str = self.__create_tuple_of_expressions(i, j)
          formatted_results.append(f"({i_str}, {j_str})")

      return relation_set % f" {', '.join(formatted_results)} "

    def __create_tuple_of_expressions(self, i: int, j: int) -> tuple[str,
  str]:
      return self.alphabet[i], self.alphabet[j]

    @abstractmethod
    def build(self) -> None:
      pass
```

Creation of dependency relation (D)

Build adjacency list of expressions of dependency relation D, which are represented by integer values (a = 0, b = 1, ...).

`main.py`

```
    # Create and print dependency relation (D).
    dependency_relation = DependencyRelation(expressions, alphabet)
    dependency_relation.build()
    print(dependency_relation)
```

`relation.py`

```
  class DependencyRelation(AbstractRelation):
    set_symbol = "D"

    # Override
    def build(self) -> None:
      self.results = [[] for _ in self.alphabet]
      expressions_enumerate_list = list(enumerate(self.expressions))
      for i, (assigned_var, operating_vars) in expressions_enumerate_list:
        for j, (assigned_var_inner, operating_vars_inner) in
```

```
expressions_enumerate_list:
        if (
             assigned_var == assigned_var_inner
          or assigned_var in operating_vars_inner
          or assigned_var_inner in operating_vars
        ):
          self.results[i].append(j)
```

Creation of independency relation (I)

Build adjacency list of expressions of independency relation I, which are represented by integer values as well. Expression e is independent, if and only if is not dependent ($e \notin D$). More clearly, just need to negate above if condition.

`main.py`

```
  # Create and print independency relation (I).
  independency_relation = IndependencyRelation(expressions, alphabet)
  independency_relation.build()
  print(independency_relation)
```

`relation.py`

```
class IndependencyRelation(AbstractRelation):
  set_symbol = "I"

  # Override
  def build(self) -> None:
    self.results = [[] for _ in self.alphabet]
    expressions_enumerate_list = list(enumerate(self.expressions))
    for i, (assigned_var, operating_vars) in expressions_enumerate_list:
      for j, (assigned_var_inner, operating_vars_inner) in
expressions_enumerate_list:
        if not (
             assigned_var == assigned_var_inner
          or assigned_var in operating_vars_inner
          or assigned_var_inner in operating_vars
        ):
          self.results[i].append(j)
```

Creation of minimal dependency relation DW of input word

Build adjacency list of letters (integers) of minimal dependency relation DW of input word. Directed acyclic graph, which is formed may be in 2 shapes: full and minimal. It uses dependency relation I to create a full graph, remembering about potential cycles and moving forward every iteration by 1 to avoid creating bidirectional edges. Then, if user wants to, reduces the full graph using `NetworkX` library.

main.py

```
  # Create minimal dependency relation of the input word.
  dependency_relation_word = DependencyWordRelation(word_int,
dependency_relation, minimal=True)
  dependency_relation_word.build()
```

relation.py

```
"""
Uses NetworkX library to create minimal directed graph.
"""
class DependencyWordRelation(AbstractRelation):
  set_symbol = "DW"

  def __init__(
      self,
      word: list[int],
      dependency_relation: DependencyRelation,
      **kwargs: dict[str, any]
  ) -> None:
    super().__init__(dependency_relation.expressions,
dependency_relation.alphabet)
    self.word                = word
    self.dependency_relation = dependency_relation
    self.results: list[list[int]] | None = None

    self.minimal = kwargs.get("minimal", False)

  # Override
  def __str__(self) -> str:
    return self.set_symbol

  # Override
  def build(self) -> None:
    if not self.dependency_relation:
      raise ValueError("Dependency relation is not builded.")

    # Create full directed acyclic relation graph of the input word.
    self.results = [[] for _ in self.word]
    for i, letter in enumerate(self.word):
      for j in range(i + 1, len(self.word)):
        if self.word[j] in self.dependency_relation.results[letter]:
          self.results[i].append(j)

    if self.minimal:
      # Reduce the graph.
      edge_list: list[tuple[int, int]] = []
      for u in range(len(self.word)):
```

```
          for v in self.results[u]:
            edge_list.append((u, v))

        digraph = nx.DiGraph(edge_list)
        reduced_digraph = nx.transitive_reduction(digraph)

        self.results = [[] for _ in self.word]
        for u, v in reduced_digraph.edges:
          self.results[u].append(v)
```

Computation Foata normal form (FNF) of DW

Uses algorithm from chapter *2.4. a simple algorithm to compute normal forms* from **"Partial Commutation and Traces"** by V. Diekert and Yves Metivier. The algorithm consists of creating stack for every letter of alphabet, using the prevously computated dependency relation DW of the word. From right to left fills appropriate stacks with letters of word I, next fills with empty markers every stack, which letter I is dependent with (different than I). Repeat until stacks are not empty.

main.py

```
  # Create and print Foata normal form based on the input word.
  foata_normal_form = FoataNF(word_int, dependency_relation,
independency_relation)
  foata_normal_form.build()
  print(f"FNF: {foata_normal_form}")
```

normal_form.py

```
from relation import DependencyRelation, IndependencyRelation

"""
Uses algorithm from chapter _2.4. a simple algorithm to compute normal
forms_
from "Partial Commutation and Traces" by V. Diekert and Yves Metivier.
"""
class FoataNF:
  def __init__(
      self,
      word: list[int],
      dependency_relation: DependencyRelation,
      independency_relation: IndependencyRelation
  ) -> None:
    self.word                 = word
    self.dependency_relation   = dependency_relation
    self.independency_relation = independency_relation
    self.results: list[list[int]] | None = None
```

```
  def __str__(self) -> str:
    return "".join(map(self.__group_letters, self.results))

  def __group_letters(self, letters: list[int]) -> str:
    return f"({''.join(map(lambda letter:
self.dependency_relation.alphabet[letter], letters))})"

  def build(self) -> None:
    stacks: list[list[int | None]] = [[] for _ in
self.dependency_relation.alphabet]

    # Build stacks of input alphabet.
    for letter_word in reversed(self.word):
      stacks[letter_word].append(letter_word)
      dependent_letters = self.__get_dependent_letters(letter_word)
      for dependent_letter in dependent_letters:
        if dependent_letter == letter_word:
          continue
        stacks[dependent_letter].append(None)

    # Evaluate Foata normal from.
    self.results = []
    while not self.__are_stacks_empty(stacks):
      self.results.append([])
      # Letters to pop from top of the stacks.
      letters_to_pop = list(
        map(
          lambda stack: stack[-1],
          filter(lambda stack: len(stack) > 0 and stack[-1] is not None,
stacks)
        )
      )
      # Pop desired letters and "stones", which are dependent with them.
      for letter_pop in letters_to_pop:
        self.results[-1].append(stacks[letter_pop].pop())
        dependent_letters = self.__get_dependent_letters(letter_pop)
        for dependent_letter in dependent_letters:
          if dependent_letter == letter_pop:
            continue
          stacks[dependent_letter].pop()

  def __get_dependent_letters(self, letter: int) -> list[int]:
    return self.dependency_relation.results[letter]

  @staticmethod
  def __are_stacks_empty(stacks: list[list[int | None]]) -> bool:
    for stack in stacks:
      if len(stack) > 0:
        return False
    return True
```

Visualization of DW

main.py

```
    # Visualize results.
    drawer = GraphDrawer(dependency_relation_word.results, word)
    drawer.draw(os.path.join(PATH_OUTPUT, f"{test_filename}.gv"))
```

drawer.py

```
import graphviz


class GraphDrawer:
  def __init__(self, digraph: list[list[int]], word: list[str]) -> None:
    self.digraph  = digraph
    self.word     = word

  def draw(self, file: str) -> None:
    digraph_dot = graphviz.Digraph(name="Hesse diagram")

    for u in range(len(self.digraph)):
      digraph_dot.node(str(u), self.word[u])

    for u in range(len(self.digraph)):
      for v in self.digraph[u]:
        digraph_dot.edge(str(u), str(v))

    print(digraph_dot.source)

    digraph_dot.render(file, format="png", overwrite_source=True)
```

Tests

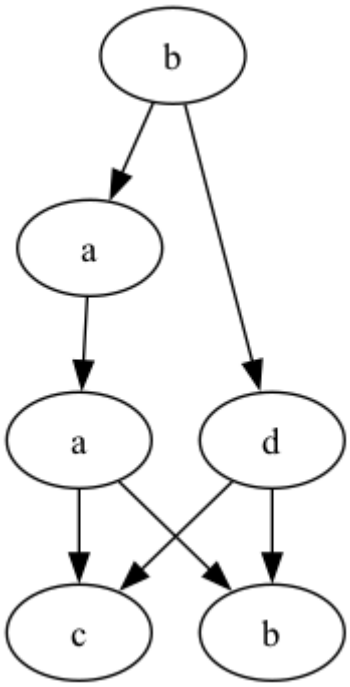
Test 1

Input

- Alphabet: $A_1 = { a, b, c, d }$
- Word: $w_1 = baadcb$
- Expression / transactions:
    1. $x := x + y$
    2. $y := y + 2z$
    3. $x := 3x + z$
    4. $z := y - z$

Output

- Dependency relation: $D = { (a, a), (a, b), (a, c), (b, a), (b, b), (b, d), (c, a), (c, c), (c, d), (d, b), (d, c), (d, d) }$
- Independency relation: $I = { (a, d), (b, c), (c, b), (d, a) }$
- Foata normal form (FNF): $(b)(ad)(a)(bc)$
- Digraph of DW with source code (DOT Language):




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

Test 2

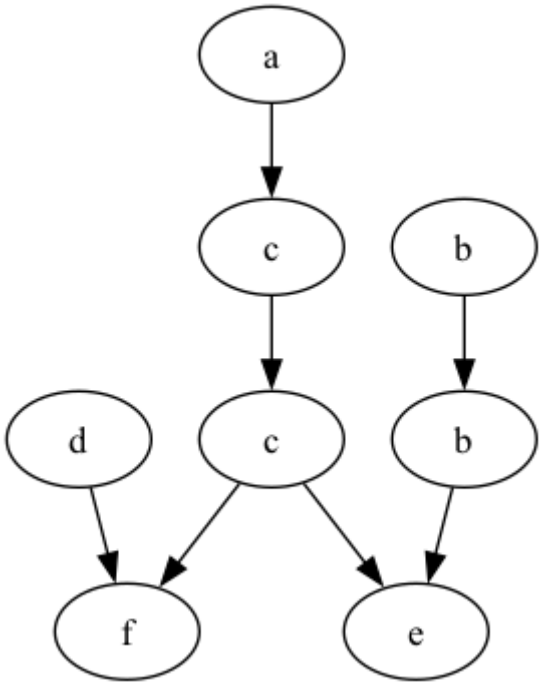
Input

- Alphabet: $A_2 = { a, b, c, d, e, f }$
- Word: $w_2 = acdcfbbe$
- Expression / transactions:
    1. $x := x + 1$
    2. $y := y + 2z$
    3. $x := 3x + z$

4. $w := w + v$
5. $z := y - z$
6. $v := x + v$

Output

- Dependency relation: $D = { (a, a), (a, c), (a, f), (b, b), (b, e), (c, a), (c, c), (c, e), (c, f), (d, d), (d, f), (e, b), (e, c), (e, e), (f, a), (f, c), (f, d), (f, f) }$
- Independency relation: $I = { (a, b), (a, d), (a, e), (b, a), (b, c), (b, d), (b, f), (c, b), (c, d), (d, a), (d, b), (d, c), (d, e), (e, a), (e, d), (e, f), (f, b), (f, e) }$
- Foata normal form (FNF): $(abd)(bc)(c)(ef)$



- Digraph of DW with source code (DOT Language):

```
digraph "Hesse diagram" {
        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 -> 4
        3 -> 7
        5 -> 6
        6 -> 7
}
```