МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

Лабораторная работа №2.2

Задачи выбора

по дисциплине: Дискретная математика

Выполнил: студент ПВ-233
Мороз Роман Алексеевич

Проверил: Островский Алексей
Мичеславович

Белгород 2024 г.

Цель работы: приобрести практические навыки в использовании алгоритмов порождения комбинаторных объектов при проектировании алгоритмов решения задач выбора.

Задания

1. Ознакомиться с задачей (см. варианты заданий).

2. Определить класс комбинаторных объектов, содержащий решение задачи (траекторию задачи).

3. Определить, что в задаче является функционалом и способ его вычисления.

4. Определить способ распознавания решения по значению функционала.

5. Реализовать алгоритм решения задачи.

6. Подготовить тестовые данные и решить задачу.

8. Задана матрица Квайна. Найти тупиковую нормальную форму Кантора, в которой количество простых импликант не меньше, чем количество простых импликант в любой другой тупиковой нормальной форме Кантора этого множества.

ТНФК, в которой количество простых импликант не меньше, чем количество простых импликант в любой другой ТНФК, представляет собой подмножество множества всех ТНФК, поэтому траекториями данной задачи являются подмножества множества всех ТНФК.

Учитывая то, что требуется найти тупиковую нормальную форму Кантора, в которой количество простых импликант не меньше, чем количество простых импликант в любой другой тупиковой нормальной форме Кантора этого множества, целесообразно порождать подмножества в порядке неувеличения мощности.

Это можно сделать, применяя алгоритм порождения сочетаний. Сначала будем порождать сочетания из n по n (может быть, матрица будет иметь одну ТНФК), затем — сочетания из n по n-1 и так далее до порождения сочетаний из n по 1.

Функционалом будет являться множество крестиков,означающий покрытие i-ой импликантой j-ой конституенты, которыми владеет множество из k выбранных импликант.

Алгоритм вычисления функционала может быть следующим:

1. Порождаем всевозможные объединения импликант

2. Находим ядро Квайна и удаляем все комбинации без него

3. Каждой импликанте создаем множество, где элемент множества указывает на наличие покрытия для конституент в матрице квайна

4. Реализуем алгоритм отбора всех тупиковых форм среди всех оставшихся объединений импликант

5. Находим среди них тупиковую форму с наибольшим количесвтом импликант

Если окажется, что тупиковых форм нет, то вернется пустое множество

```
from pprint import pprint


def get_combinations(combination, k, original_set, i=0, b=0):
    all_combinations = []
    if i == k:
        return [combination.copy()]
    else:
        for x in range(b, len(original_set) - k + i + 1):
            combination[i] = original_set[x]
            all_combinations.extend(get_combinations(combination, k,
original_set, i + 1, x + 1))
    return all_combinations


def implicant_in_constituant(implicant, constituent):
    counter = 0
    count_equal = 0
    for i, elem in enumerate(constituent):
        if elem == implicant[i]:
            counter += 1
        if implicant[i] == '-':
            count_equal += 1

    return counter == 3 if count_equal == 1 else counter == 2


implicants = ['- 10-', '0-11', '01-1', '10-1', '1-01', '-011']
constituents = ['0011', '0100', '0101', '0111', '1001', '1011',
'1100', '1101']
kwayn_matrics = [['0' for _ in range(len(constituents))] for _ in
range(len(implicants))]
```

```
k = len(implicants)


def make_coverage(matrics, implicants, constituents):
    for i in range(len(implicants)):
        for j in range(len(constituents)):
            if implicant_in_constituant(implicants[i],
constituents[j]):
                matrics[i][j] = '+'


make_coverage(kwayn_matrics, implicants, constituents)
pprint(kwayn_matrics)
print()


def get_kwayn_core(matrics, implicants, constituents):
    kwayn_core = []
    for i in range(len(constituents)):
        counter = 0
        plus_index = 0
        for j in range(len(implicants)):
            if matrics[j][i] == '+':
                counter += 1
                plus_index = j
            if counter >= 2:
                break
            if j == len(implicants) - 1 and counter == 1 and
implicants[plus_index] not in kwayn_core:
                kwayn_core.append(implicants[plus_index])

    return kwayn_core


kwayn_core = get_kwayn_core(kwayn_matrics, implicants, constituents)


def check_kwayn_core(combination, kwayn_core):
    counter = 0
    for i in range(len(kwayn_core)):
        if kwayn_core[i] in combination:
            counter += 1

    return counter == len(kwayn_core)


def get_combinations_with_core(k, implicants, kwayn_core):
    result = []
    while k != 0:
        combination = [0] * k
        list_comb = get_combinations(combination, k, implicants)
        for comb in list_comb:
            if check_kwayn_core(comb, kwayn_core):
                result.append(comb)
        k -= 1
    return result
```

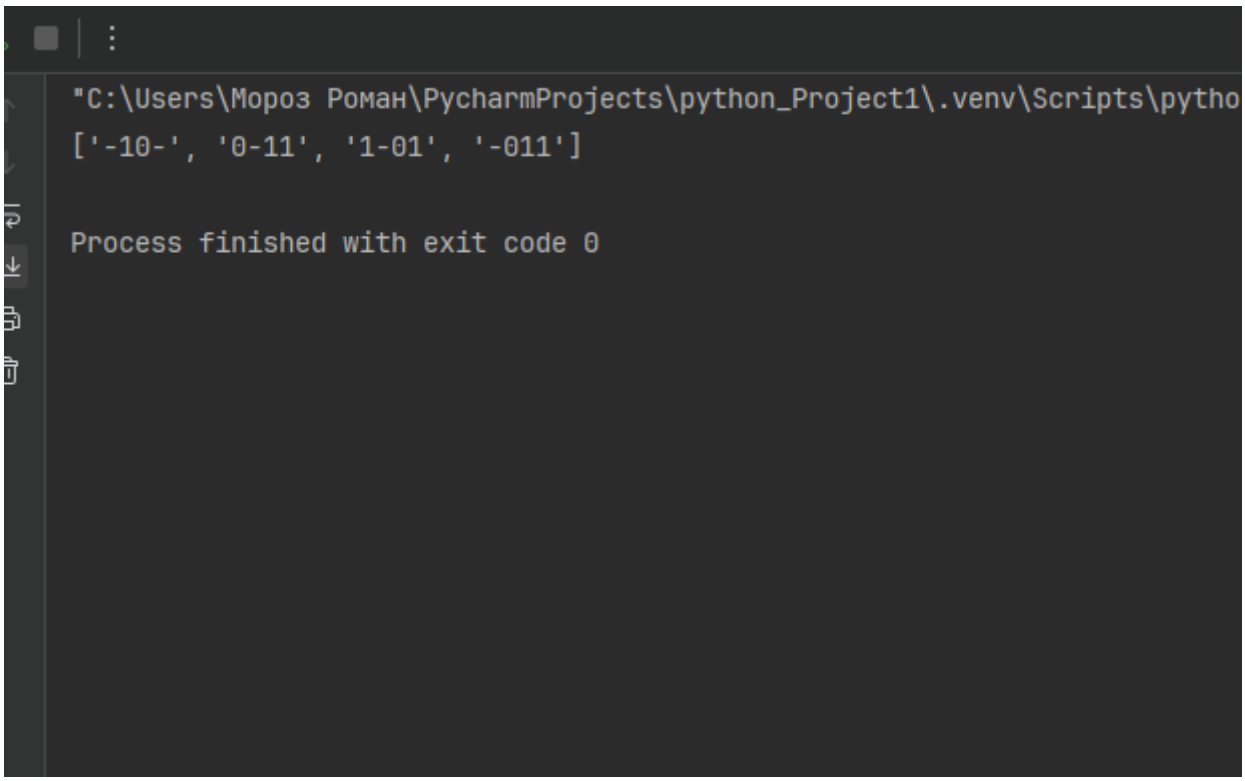
```
list_comb = get_combinations_with_core(k, implicants, kwayn_core)
pprint(list_comb)


def get_sets_for_implicants(matrics, implicants, implicant_sets=[]):
    for i in range(len(implicants)):
        implicant_sets.append(get_set(matrics[i], set()))

    return implicant_sets


def get_set(list, res_set=set()):
    for i in range(len(list)):
        if list[i] == '+':
            res_set.add(i + 1)
    return res_set


list_sets = get_sets_for_implicants(kwayn_matrics, implicants)

def get_dict_for_implicants(implicants, implicant_sets, res_dict =
{}):
    for i in range(len(implicants)):
        res_dict[implicants[i]] = implicant_sets[i]

    return res_dict


U = set(i for i in range(1, len(constituents) + 1))

print(get_dict_for_implicants(implicants, list_sets))
```

```
"C:\Users\Мороз Роман\PycharmProjects\python_Project1\.venv\Scripts\python.exe" "C:\Users\Мороз Роман\Pyc
[['0', '0', '0', '0', '0', '0', '0', '0'],
 ['+', '0', '0', '+', '0', '0', '0', '0'],
 ['0', '0', '+', '+', '0', '0', '0', '0'],
 ['0', '0', '0', '0', '+', '+', '0', '0'],
 ['0', '0', '0', '0', '+', '0', '0', '+'],
 ['+', '0', '0', '0', '0', '+', '0', '0']]

[['- 10-', '0-11', '01-1', '10-1', '1-01', '-011'],
 ['- 10-', '0-11', '01-1', '10-1', '1-01'],
 ['- 10-', '0-11', '01-1', '1-01', '-011'],
 ['- 10-', '01-1', '10-1', '1-01', '-011'],
 ['0-11', '01-1', '10-1', '1-01', '-011'],
 ['- 10-', '0-11', '01-1', '1-01'],
 ['- 10-', '01-1', '10-1', '1-01'],
 ['- 10-', '01-1', '1-01', '-011'],
 ['0-11', '01-1', '10-1', '1-01'],
 ['0-11', '01-1', '1-01', '-011'],
 ['01-1', '10-1', '1-01', '-011'],
 ['- 10-', '01-1', '1-01'],
 ['0-11', '01-1', '1-01'],
 ['01-1', '10-1', '1-01'],
 ['01-1', '1-01', '-011'],
 ['01-1', '1-01']]
{'- 10-': set(), '0-11': {1, 4}, '01-1': {3, 4}, '10-1': {5, 6}, '1-01': {8, 5}, '-011': {1, 6}}
```

С помощью этих функций обрабатываем данные в ответ данные:

```
comb = [['0-11', '-011', '01-1', '10-1', '1-01', '-10-'],
 ['0-11', '-011', '01-1', '10-1', '-10-'],
 ['0-11', '-011', '01-1', '1-01', '-10-'],
 ['0-11', '-011', '10-1', '1-01', '-10-'],
 ['0-11', '01-1', '10-1', '1-01', '-10-'],
 ['-011', '01-1', '10-1', '1-01', '-10-'],
 ['0-11', '-011', '01-1', '-10-'],
 ['0-11', '-011', '10-1', '-10-'],
 ['0-11', '-011', '1-01', '-10-'],
 ['0-11', '01-1', '10-1', '-10-'],
 ['0-11', '01-1', '1-01', '-10-'],
 ['0-11', '10-1', '1-01', '-10-'],
 ['-011', '01-1', '10-1', '-10-'],
 ['-011', '01-1', '1-01', '-10-'],
 ['-011', '10-1', '1-01', '-10-'],
 ['01-1', '10-1', '1-01', '-10-'],
 ['0-11', '-011', '-10-'],
 ['0-11', '01-1', '-10-'],
 ['0-11', '10-1', '-10-'],
 ['0-11', '1-01', '-10-'],
 ['-011', '01-1', '-10-'],
 ['-011', '10-1', '-10-'],
 ['-011', '1-01', '-10-'],
 ['01-1', '10-1', '-10-'],
 ['01-1', '1-01', '-10-'],
 ['10-1', '1-01', '-10-'],
 ['0-11', '-10-'],
```

```
 ['-011', '-10-'],
 ['01-1', '-10-'],
 ['10-1', '-10-'],
 ['1-01', '-10-'],
 ['-10-']]

kwayn_core = '-10-'
c = ['0-11', '-011', '01-1', '10-1', '1-01', '-10-']
impl_dict = {'-10-': {8, 2, 3, 7}, '0-11': {1, 4}, '01-1': {3, 4},
'10-1': {5, 6}, '1-01': {8, 5}, '-011': {1, 6}}


def create_set_from_implicants(list_implicants, dict_implicants, U):
    res_set = set()
    res_form = []
    for key in dict_implicants.keys():
        if key in list_implicants and not
dict_implicants[key].issubset(res_set):
            res_set |= dict_implicants[key]
            res_form.append(key)

    if res_set == U:
        return res_form

U = set(i for i in range(1, 9))

res = []

def print_max_TNFK(impl_dict, U):
    for imlicants in comb:
        current_form = create_set_from_implicants(imlicants,
impl_dict, U)
        if current_form not in res and current_form != None:
            res.append(current_form)

    max_len = 0
    index_max_TNFK = 0
    for i, element in enumerate(res):
        if len(element) > max_len:
            max_len = len(element)
            index_max_TNFK = i

    print(res[index_max_TNFK])

print_max_TNFK(impl_dict, U)
```



Данные функции будут являться функционалом:

```
def create_set_from_implicants(list_implicants, dict_implicants, U):
    res_set = set()
    res_form = []
    for key in dict_implicants.keys():
        if key in list_implicants and not
dict_implicants[key].issubset(res_set):
            res_set |= dict_implicants[key]
            res_form.append(key)

    if res_set == U:
        return res_form

U = set(i for i in range(1, 9))

def print_max_TNFK(impl_dict, U):
    for imlicants in comb:
        current_form = create_set_from_implicants(imlicants, impl_dict, U)
        if current_form not in res and current_form != None:
            res.append(current_form)

    max_len = 0
    index_max_TNFK = 0
    for i, element in enumerate(res):
        if len(element) > max_len:
            max_len = len(element)
            index_max_TNFK = i

    print(res[index_max_TNFK])

print_max_TNFK(impl_dict, U)
```

Вывод: приобрели практические навыки в использовании алгоритмов порождения комбинаторных объектов при проектировании алгоритмов решения задач выбора.