Лабораторная работа №4

Выполнил студент группы ИДБ-18-09 Полс А.Д.

Вариант №8

Цель работы: Изучить метод ветвей и границ для решения задачи о коммивояжёре дискретного программирования и применить его на практическом примере

In [1]:
```
# необходимые зависимости

import string
import warnings

import numpy as np
import pandas as pd

from typing import Tuple, List
from ast import literal_eval as make_tuple
from IPython.display import display, Markdown
```

In [2]:
```
warnings.filterwarnings('ignore', r'All-NaN slice encountered')
```

In [3]:
```
# форматированный вывод

def fprint(text):
    if text is str:
        display(Markdown(text))
    else:
        display(Markdown(str(text)))
```

Матрица варианта

In [4]:
```
matrix = np.array(
    [
        [np.nan, 4, 7, 9, 4],
        [3, np.nan, 1, 3, 2],
        [10, 9, np.nan, 1, 7],
        [4, 6, 1, np.nan, 9],
        [5, 2, 3, 7, np.nan]
    ],
        dtype=float
)

graph = pd.DataFrame(
    matrix,
    columns=list(string.ascii_uppercase[:matrix.shape[0]]),
    index=list(string.ascii_uppercase[:matrix.shape[0]])
)

graph
```

Out[4]:

	A	B	C	D	E
A	NaN	4.0	7.0	9.0	4.0
B	3.0	NaN	1.0	3.0	2.0
C	10.0	9.0	NaN	1.0	7.0
D	4.0	6.0	1.0	NaN	9.0
E	5.0	2.0	3.0	7.0	NaN

Метод ветвей и границ

Класс дерева

In [5]:
```
class Node:
    def __init__(self, frame: pd.DataFrame = None, weight: float = N
one, path: str = 'root', parent = None):
        self.is_leaf = True
        self.left = None
        self.right = None
        self.frame = frame
        self.parent = parent
        self.weight = weight
        self.path = path
        self.is_forgoten = False

    def grow_left(self, frame: pd.DataFrame, weight: float, path: st
r) -> None:
        self.left = Node(frame, weight, path, self)
        self._check_growth()

    def grow_right(self, frame: pd.DataFrame, weight: float, path: s
tr) -> None:
        self.right = Node(frame, weight, path, self)
        self._check_growth()

    def _check_growth(self) -> None:
        if self.left is not None and self.right is not None:
            self.is_leaf = False

    def find_least_leaf(self):
        if self.is_leaf:

            return self
        elif self.left is not None and self.right is not None:
            left = self.left.find_least_leaf()
            right = self.right.find_least_leaf()
            if left.weight < right.weight:
                right.is_forgoten = True
                return left
            else:
                left.is_forgoten = True
                return right

    def backward(self):
        total_path = list()

        cursor = self
        while True:
            total_path.append(cursor.path)
            if cursor.parent is not None:
                cursor = cursor.parent
            else:
                return total_path
```

Класс алгоритма решения

In [6]:
```
class BranchAndBoundSolver:

    def __init__(self, initial_frame: pd.DataFrame):
        self.root = Node(
            *BranchAndBoundSolver.reduction(initial_frame)
        )

    def solve(self):

        while self.root.find_least_leaf().frame.to_numpy().shape !=
(2, 2):
            leaf = self.root.find_least_leaf()

            max_element_value, max_element_position, pos_names = Bra
nchAndBoundSolver.max_element_analysis(
                leaf.frame)

            if leaf.is_forgoten:
                leaf.frame, _ = BranchAndBoundSolver.ban_element(lea
f.frame, pos_names)
                max_element_value, max_element_position, pos_names =
BranchAndBoundSolver.max_element_analysis(
                    leaf.frame)

            leaf.grow_left(*BranchAndBoundSolver.remove_path(leaf.fr
ame, max_element_position, leaf.weight, pos_names))
            leaf.grow_right(leaf.frame.copy(), leaf.weight + max_ele
ment_value, f'skip: {pos_names}')

        last_leaf = self.root.find_least_leaf()
        last_frame = last_leaf.frame.copy(deep=True)

        pre, last = BranchAndBoundSolver.special_max_element_analysi
s(last_frame)

        self.path = list(reversed(last_leaf.backward()))
        self.path.append(f'add: {pre}')
        self.path.append(f'add: {last}')

        return self.path, last_leaf.weight

    @staticmethod
    def remove_path(frame: pd.DataFrame, max_element_position, last_
penalty, pos_names):

        local_frame = frame.copy(deep=True)
        i, j = max_element_position
        begin, end = pos_names

        try:
            local_frame.loc[begin][end] = np.nan
        except:
            pass

        try:
            local_frame.loc[end][begin] = np.nan
        except:
            pass
        local_matrix = local_frame.to_numpy().copy()

        local_matrix = np.delete(local_matrix, i, axis=0)
        local_matrix = np.delete(local_matrix, j, axis=1)

        columns = local_frame.columns[local_frame.columns != end]
        index = local_frame.index[local_frame.index != begin]

        new_frame = pd.DataFrame(local_matrix.copy(), columns=column
s, index=index)

        reduced_new_frame, penalty = BranchAndBoundSolver.reduction
(new_frame)

        return reduced_new_frame.copy(deep=True), penalty + last_pen
alty, f'add: {pos_names}'

    @staticmethod
    def special_max_element_analysis(frame: pd.DataFrame):
        local_matrix = frame.to_numpy().copy()
        new_matrix = np.zeros(shape=local_matrix.shape)

        for i in np.arange(local_matrix.shape[0]):
            for j in np.arange(local_matrix.shape[1]):
                stash = local_matrix[i, j]
                local_matrix[i, j] = np.nan
                axis0_min = np.nanmin(local_matrix[i])
                axis1_min = np.nanmin(local_matrix.T[j])
                new_matrix[i, j] = axis0_min + axis1_min
                local_matrix[i, j] = stash

        max_value = np.max(new_matrix)

        first_value_pos = np.argwhere(np.isnan(new_matrix)).T[0]
        second_value_pos = np.argwhere(np.isnan(new_matrix)).T[1]

        first_max_value_indecies = tuple(first_value_pos)
        second_max_value_indecies = tuple(second_value_pos)

        local_frame = pd.DataFrame(new_matrix, columns=frame.column
s.copy(), index=frame.index.copy())

        first_begin = local_frame.index[first_max_value_indecies[0]]
        first_end = local_frame.columns[first_max_value_indecies[1]]

        second_begin = local_frame.index[second_max_value_indecies
[0]]
        second_end = local_frame.columns[second_max_value_indecies
[1]]

        first_pos_names = (first_begin, first_end)
        second_pos_names = (second_begin, second_end)

        return first_pos_names, second_pos_names

    @staticmethod
    def max_element_analysis(frame: pd.DataFrame):
        local_matrix = frame.to_numpy().copy()
        new_matrix = np.zeros(shape=local_matrix.shape)
        for i in np.arange(local_matrix.shape[0]):
            for j in np.arange(local_matrix.shape[1]):
                stash = local_matrix[i, j]
                local_matrix[i, j] = np.nan

                axis0_min = np.nanmin(local_matrix[i])
                axis1_min = np.nanmin(local_matrix.T[j])

                new_matrix[i, j] = axis0_min + axis1_min

                local_matrix[i, j] = stash

        max_value = np.nanmax(new_matrix)

        max_value_position = np.array(
            np.where(new_matrix == max_value)
        ).T[0]

        max_value_indecies = tuple(max_value_position)

        local_frame = pd.DataFrame(new_matrix, columns=frame.column
s.copy(), index=frame.index.copy())

        begin = local_frame.index[max_value_indecies[0]]
        end = local_frame.columns[max_value_indecies[1]]

        pos_names = (begin, end)

        return max_value, max_value_position, pos_names

    @staticmethod
    def reduction(frame: pd.DataFrame):
        local_matrix = frame.to_numpy().copy()
        weight = 0

        axis1_mins = np.nanmin(local_matrix, axis=1).reshape(-1, 1)
        weight += axis1_mins.sum()
        local_matrix -= axis1_mins

        axis0_mins = np.nanmin(local_matrix, axis=0)
        weight += axis0_mins.sum()
        local_matrix -= axis0_mins

        new_frame = pd.DataFrame(local_matrix, columns=frame.column
s, index=frame.index)

        return new_frame, weight

    @staticmethod
    def ban_element(frame: pd.DataFrame, pos_names):
        local_frame = frame.copy(deep=True)
        local_frame.loc[pos_names[0]][pos_names[1]] = np.nan

        return BranchAndBoundSolver.reduction(local_frame)
```

Результат решения

```
In [7]: graph
```

Out[7]:

	A	B	C	D	E
A	NaN	4.0	7.0	9.0	4.0
B	3.0	NaN	1.0	3.0	2.0
C	10.0	9.0	NaN	1.0	7.0
D	4.0	6.0	1.0	NaN	9.0
E	5.0	2.0	3.0	7.0	NaN

```
In [8]: solver = BranchAndBoundSolver(graph)
        branch, weight = solver.solve()
```

Ветвь дерева с решением

```
In [9]: branch

Out[9]: ['root',
         "add: ('C', 'D')",
         "add: ('D', 'A')",
         "add: ('B', 'C')",
         "add: ('A', 'E')",
         "add: ('E', 'B')"]
```

Конечная длинна маршрута

```
In [10]: weight

Out[10]: 12.0
```

Конечный маршрут

In [11]:
```
def remove_prefix(text, prefix):
    if text.startswith(prefix):
        return text[len(prefix):]
    return text

verts = list(map(lambda t: make_tuple(remove_prefix(t, 'add: ')), br
anch[1:]))

chain = 'A'

while len(chain) < graph.shape[0]:
    for v in verts:
        if chain[-1] == v[0]:
            chain += v[1]

chain = chain[:6]; chain
```

Out[11]:
```
'AEBCDA'
```

In [12]:
```
str_path = '$'

for symbol in chain:
    str_path += rf'{symbol} \rightarrow '

str_path += rf'{weight} $'
```

In [13]:
```
fprint(str_path)
```

$A \rightarrow E \rightarrow B \rightarrow C \rightarrow D \rightarrow A \rightarrow 12.0$