Міністерство освіти і науки України
Національний авіаційний університет
Факультет кібербезпеки, комп'ютерної та програмної інженерії
Кафедра комп'ютеризованих систем управління

Лабораторна робота № 1.4
з дисципліни «Системи штучного інтелекту»
на тему «Методи пошуку в просторі станів»

Виконав:
студент ФККПІ
групи СП-425
Клокун В. Д.
Перевірила:
Росінська Г. П.

Київ 2019

1. Завдання роботи

Ознайомитись з методами пошуку розв'язків інтелектуальних задач у просторі станів. Вивчити блок-схеми алгоритмів для всіх методів пошуку.

2. Хід роботи

Щоб виконати лабораторну роботу, необхідно розв'язати задачу комівояжера для графа, заданого за варіантом (рис. 1), тобто знайти такий найкоротший шлях, який відвідає усі точки графа і повернеться у початкову точку.

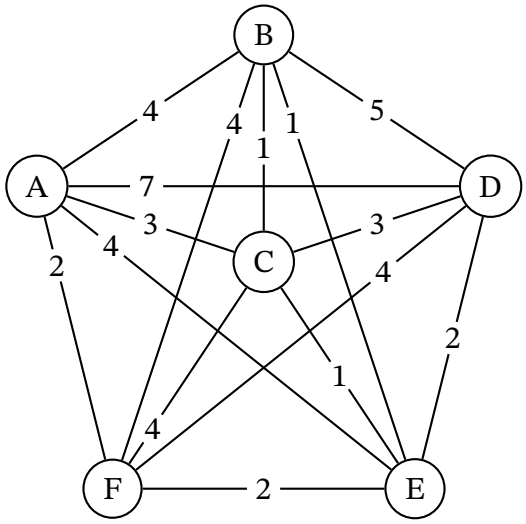


Рис. 1: Граф, який описує умову задачі комівояжера

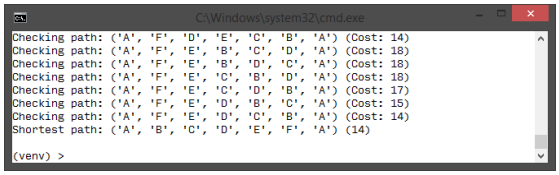
Щоб розв'язати задачу, розроблюємо програму, яка шукатиме розв'язок задачі за допомогою методу повного перебору можливих розв'язків у просторі станів. Необхідно розв'язати задачу двома способами: перебор в ширину та в глибину. Розробивши програму (лістинг А.1), запускаємо її і спостерігаємо за результатом (рис. 2). Як бачимо, програма знайшла такий найкоротший шлях:

$$A \xrightarrow{4} B \xrightarrow{1} C \xrightarrow{3} D \xrightarrow{2} E \xrightarrow{2} F \xrightarrow{2} A = 14,$$

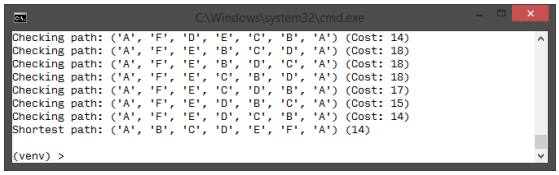
перебравши можливі розв'язки у просторі станів двома обраними способами (рис. 3, 4).

3. Висновок

Виконуючи дану лабораторну роботу, ми дослідили процес породження простору станів і ознайомились із прикладами задач представлення у просторі станів,



а) Пошук вшир



б) Пошук вглиб

Рис. 2: Результат розв’язання задачі програмою

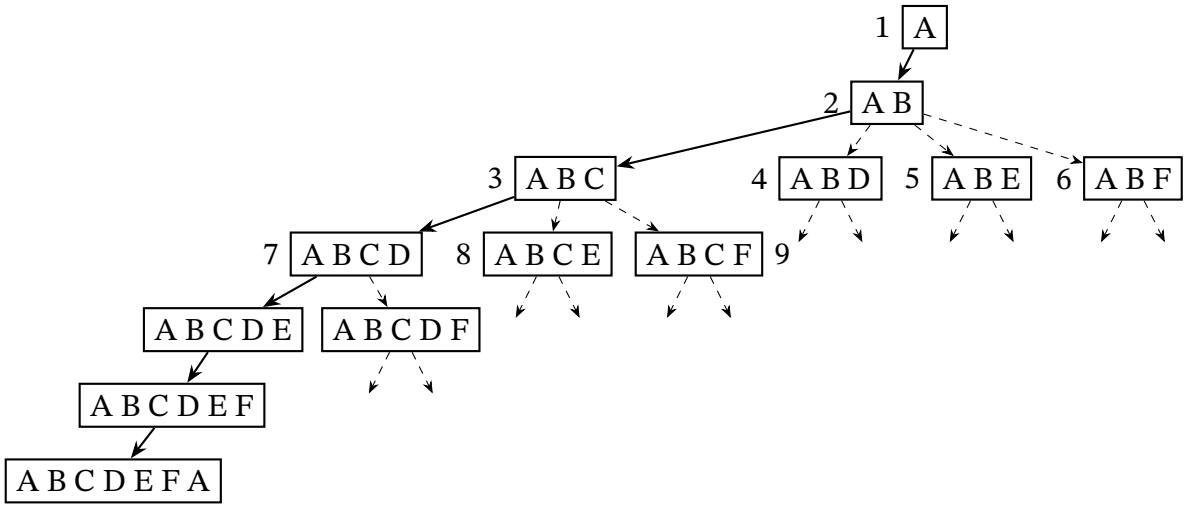


Рис. 3: Частина графа пошуку розв’язку вшир

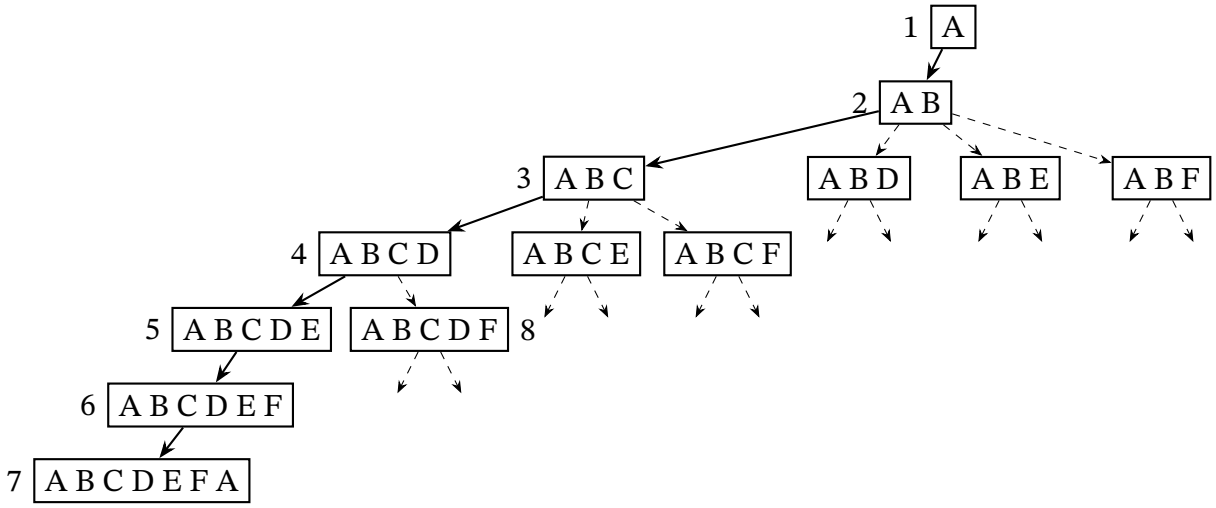


Рис. 4: Частина графа пошуку розв’язку вглиб

а також розробили програму для розв'язання задачі комівояжера методом повного перебору розв'язків у просторі станів.

А. Програма для розв'язку поставленої задачі

Лістинг А.1: Початковий код програмного модуля для розв'язання задачі комівояжера

```
import argparse
import itertools
import networkx as nx
import matplotlib.pyplot as plt

INF = 9999


class Path(object):

    def __init__(self, path):
        self.path = path

    def __str__(self):
        s = "".join(str(self.path))
        return s

    def __add__(self, other):
        if not isinstance(other, Path):
            raise TypeError(
                "Cannot add {} to Path"
                .format(type(other))
            )

        res = Path(
            self.path + other.path
        )
        return res

    def __hash__(self):
        return hash(self.path)

    def __eq__(self, other):
        return self.path == other.path

    def __len__(self):
        return len(self.path)

```

```
    @property
    def name(self):
        name = "".join(self.path)
        return name

    @property
    def subpaths(self):
        for idx in range(1, len(self.path) + 1):
            yield Path(self.path[:idx])

    @property
    def edges(self):
        subpaths = list(self.subpaths)
        src_edge = subpaths[0]
        for dst_edge in subpaths[1:]:
            yield (src_edge, dst_edge)
            src_edge = dst_edge

    def calculate_cost(self, distance_matrix):
        distance = 0
        src_idx = self.path[0]
        for dst_idx in self.path[1:]:
            distance += distance_matrix[src_idx][dst_idx]
            src_idx = dst_idx

        return distance


class SearchTree(nx.Graph):

    SOLUTION_METHODS = {
        "bfs": nx.bfs_tree,
        "dfs": nx.dfs_tree,
    }

    def __init__(self, distance_matrix, *args, **kwargs):

        self.distance_matrix = distance_matrix
        self.shortest_path_length = len(distance_matrix) + 1

        super().__init__(*args, **kwargs)

        self._build()

    def _build(self):
        nodes = list(self.distance_matrix.keys())

```

```
86          starting_vertex = Path((nodes[0],))
87          nodes = nodes[1:]
88          for permutation in itertools.permutations(nodes):
89              cur_path = starting_vertex + Path(permutation) +
                ↪  starting_vertex
90              self.add_path(cur_path)
91
92      def add_path(self, path):
93          for sp in path.subpaths:
94              self.add_node(sp)
95
96          for src, dst in path.edges:
97              self.add_edge(src, dst)
98
99      def show(self, *args, **kwargs):
100         nx.draw(self, *args, **kwargs)
101         plt.show()
102
103     def solve_tsp(self, starting_v, method="bfs", print_search=False):
104         print("Solving using method: {}".format(method))
105         try:
106             method = self.SOLUTION_METHODS[method]
107         except KeyError:
108             raise ValueError(
109                 "I don't know the solution method <{}>"
110                 .format(method)
111             )
112
113         if isinstance(starting_v, str):
114             starting_v = Path((starting_v,))
115
116         cur_min = INF
117
118         # Traverse the tree using the selected method
119         for n in method(self, starting_v):
120             # If the current path length is less than the required
                ↪  minimum,
121             # continue the iteration. Used to skip short, unsuitable paths
122             if len(n) != self.shortest_path_length:
123                 continue
124
125             cur_cost = n.calculate_cost(self.distance_matrix)
126             if print_search:
127                 print(
128                     "Checking path: {} (Cost: {})"
129                     .format(
130                         n,
```

```
                        cur_cost
                    )
                )
            if cur_cost < cur_min:
                cur_min = cur_cost
                min_path = n

        return min_path


def main(method, print_search=False, *args, **kwargs):
    distances = {
        "A": {"A": INF, "B":   4, "C":   3, "D":   7, "E":   4, "F":   2},
        "B": {"A":   4, "B": INF, "C":   1, "D":   5, "E":   3, "F":   6},
        "C": {"A":   3, "B":   1, "C": INF, "D":   3, "E":   1, "F":   4},
        "D": {"A":   7, "B":   5, "C":   3, "D": INF, "E":   2, "F":   4},
        "E": {"A":   4, "B":   3, "C":   1, "D":   2, "E": INF, "F":   2},
        "F": {"A":   2, "B":   6, "C":   4, "D":   4, "E":   2, "F": INF},
    }

    search_tree = SearchTree(distances)

    minpath = search_tree.solve_tsp(
        starting_v="A",
        method=method,
        print_search=print_search,
    )
    minpathcost = minpath.calculate_cost(distances)

    print(
        "Shortest path: {} ({})"
        .format(
            minpath,
            minpathcost,
        )
    )


if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        "Solve a TSP problem using either DFS or BFS on solution tree."
    )
    parser.add_argument(
        "method",
        type=str,
        default="bfs"
    )
```

```
    parser.add_argument(
        "-p",
        "--print-search",
        dest="print_search",
        action="store_true",
        help=(
            "Print the search process."
        )
    )
    parser.set_defaults(print_search=False)

    args = parser.parse_args()
    method = args.method
    print_search = args.print_search
    main(
        method=method,
        print_search=print_search
    )
```