МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Інститут Комп'ютерних наук та інформаційних технологій
Кафедра Програмної інженерії та інтелектуальних технологій управління
Спеціальність 122 Комп'ютерні науки
Освітня програма Комп'ютерні науки та інтелектуальні системи

ЛАБОРАТОРНА РОБОТА

з дисципліни

«Моделі штучного інтелекту»

Виконав студент 5 курсу, групи КН-М422

Захар Геннадійович ПАРАХІН

(підпис, прізвище та ініціали)

Перевірила Ольга Юріївна ЧЕРЕДНІЧЕНКО

(підпис, прізвище та ініціали)

Харків 2023

ЗМІСТ

1 Хід виконання роботи

1.1 Розгляд поставленої задачі

Задача про фермера, вовка, козу і капусту — логічна гра, яка ілюструє класичну проблему в області штучного інтелекту з пошуку рішення, яке задоволення системі обмежень. З одного берега на інший фермер має перевезти човном вовка, козу і капусту. Човен витримує лише човняра і одного «пасажира». Одночасно не можна залишати разом на березі вовка і козу, козу і капусту. Можна робити скільки завгодно рейсів.

Розглядаючи проблему, можно сформулювати, що кожена переправа (відправлення човна з берега річки А на берег річки Б з одним або декількома) призводить до нового стану, і ми можемо досягти цільового стану, вирішуючи задачу пошуку в стані простір. Коли додаємо перспективу графа до проблеми, кожен стан стає вузлом, а зміна стану стає ребром.

Отже, проблему можна вирішити, розширюючи вузли ходами (переправами згідно з правилами задачі), доки ми не досягнемо цільового вузла.

Якщо ми зробимо це в стилі «Спочатку в ширину» (розгорнувшись вшир перед тим, як спуститися на дерево), ми побачимо, що перший вузол рішення, який ми отримуємо, має найкоротший шлях Якщо інше рішення є на тому ж рівні, вартість буде такою ж.

Якщо інше рішення існує нижче поточного рівня, вартість буде вищою за поточну вартість

Отже, оскільки ми зацікавлені в пошуку рішення з найменшою вартістю, а не в пошуку всіх можливих рішень, ми можемо завершити проблему до пошуку шляху до цільового вузла від кореневого вузла в DAG (спрямований ациклічний граф). Інакше, якщо нас цікавлять інші рішення, ми можемо побачити, що графік стає циклічним.

1.2 Виділення основних обмежень

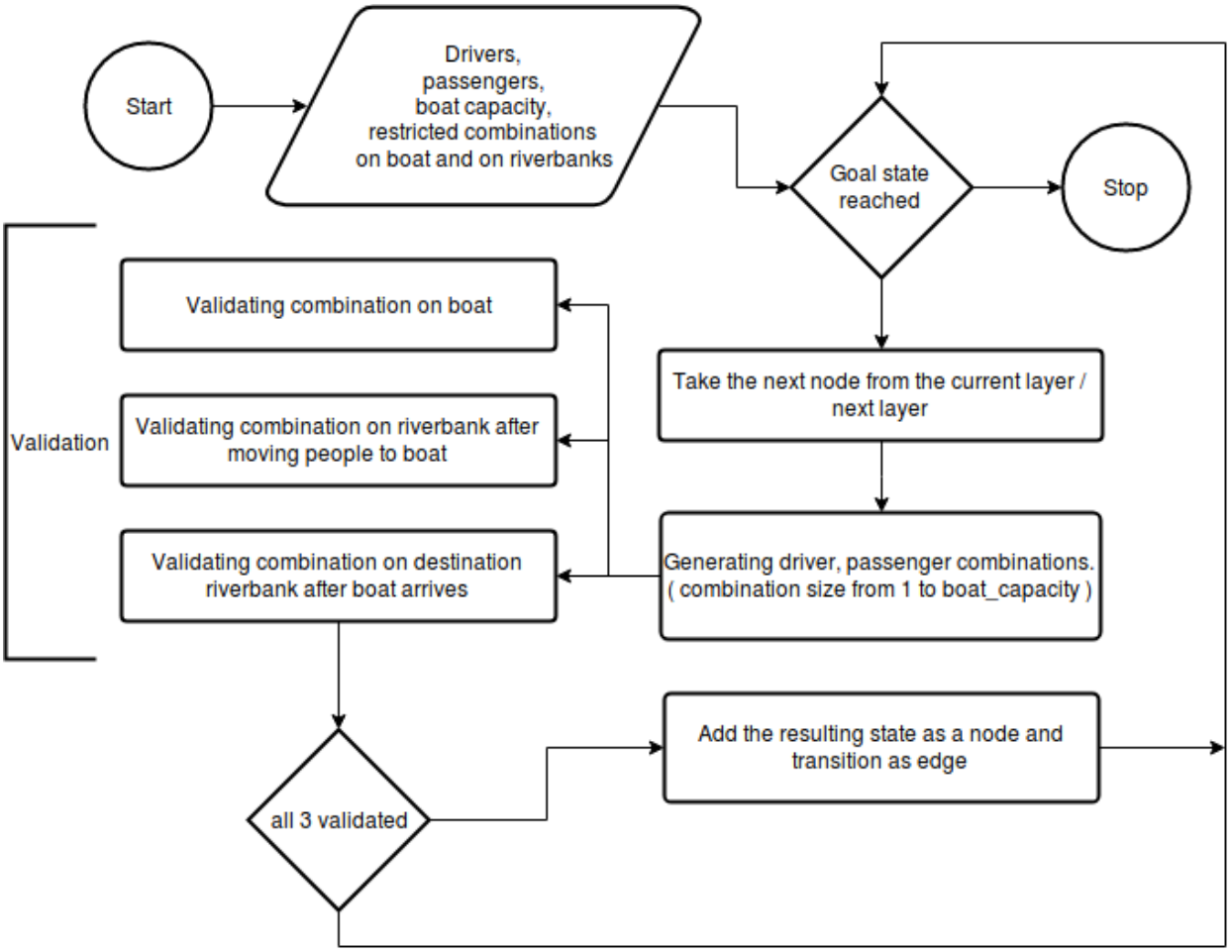

Рисунок 1 - Блок схема типового алгоритму вирішення задачі.

Існує багато проблем з переправою через річку, які залежать від комбінації обмежень на берегах річки, човна та обмежень місткості човна. Отже, сформулювавши 3 валідатори, як на блок-схемі, ми можемо максимізувати простір проблем, який ми можемо вирішити.

1.3 Вибір методу для вирішення задачі

Пошук в шиирину (BFS) - це проста стратегія, у якій спочатку розгортається кореневий вузол, потім - усі наступники кореневого вузла, після цього розгортаються наступники цих наступників тощо. Взагалі кажучи, при пошуку в ширину, насамперед чим відбувається розгортання будь-яких вузлів наступному рівні, розгортаються все вузли даної конкретної глибині у дереві пошуку.

Алгори́тм по́шуку в глибину́ (DFS) — алгоритм для обходу дерева, структури подібної до дерева, або графу. Робота алгоритму починається з кореня дерева (або іншої обраної вершини в графі) і здійснюється обхід в максимально можливу глибину до переходу на наступну вершину.

При реалізації використаємо ці два методи для їхнього порівняння.

2 Реалізація рішення поставленої задачі

Замість лівого і правого берега, будемо називати їх в коді захід і схід, а також розділемо код агента (wolfgoatcabbage.py) від кода задіяного у формуванні дерева і запуску дій (main.py). І обрана мова програмування Python. Для скорочення вовк замінено на “w”, козу на “g”, а капусту на “c”

Перш ніж агент зможе приступити до пошуку рішень, він має формулювати ціль, а потім використовувати цю мету для формулювання задачі. Завдання складається з чотирьох частин: початковий стан, безліч дій, функція перевірки мети та функція вартості шляху. Середовище завдання представлене простором станів, а шлях через простір станів від початкового стану до цільового стану є рішенням. Для вирішення будь-якого завдання може використовуватися єдиний, загальний алгоритм Tree-Search; конкретні варіанти цього алгоритму втілюють різні стратегії.

Лістинг агента:

```
from search import *
# A* WolfGoatCabbage
class WolfGoatCabbage(Problem):
        def __init__(self, initial=frozenset({'F', 'W', 'G', 'C'}),
goal=set()):
            super().__init__(initial, goal)

    def goal_test(self, state):
        return state == self.goal

    def result(self, state, action):
        next_state = state + action
        return frozenset(next_state)
```

```
    def actions(self, state):

        if state == {'F', 'W', 'G', 'C'}:
            return [{'G', 'F'}]
        if state == {'W', 'C'}:
            return [{'F'}]
        if state == {'F', 'W', 'C'}:
            return [{'C', 'F'}, {'F', 'W'}]
        if state == {'C'}:
            return [{'F', 'G'}]
        if state == {'W'}:
            return [{'F', 'G'}]
        if state == {'F', 'C', 'G'}:
            return [{'F', 'C'}]
        if state == {'F', 'W', 'G'}:
            return [{'W', 'F'}]
        if state == {'G'}:
            return [{'F'}]
        if state == {'G', 'F'}:
            return [{'G', 'F'}]

    def result(self, state, actions):
        new_state = set()
        for a in state:
            new_state.add(a)

        for b in actions:
            if b not in state:
                new_state.add(b)
            else:
                new_state.remove(b)

        return frozenset(new_state)

if __name__ == '__main__':
    wgc = WolfGoatCabbage()

    solution = depth_first_graph_search(wgc).solution()
    print(solution)
    solution = breadth_first_graph_search(wgc).solution()
    print(solution)
```

Лістинг main.py

```
class WGC_Node:
    incompatibilities = [
        ["c", "g", "w"],
        ["g", "w"],
        ["c", "g"]
    ]

    def __init__(self, west=["w", "g", "c"], east=[], boat_side=False, children=[]):
        self.west = west
        self.east = east
        self.boat_side = boat_side
        self.children = children

    def __str__(self):
        return str(self.west) + str(self.east) + ("Left" if not self.boat_side else "Right")

    def generate_children(self, previous_states, parent_map):
        children = []
        if not self.boat_side:
            for i in self.west:
                new_west = self.west[:]
                new_west.remove(i)
                new_east = self.east[:]
                new_east.append(i)
                if sorted(new_west) not in WGC_Node.incompatibilities and not WGC_Node.state_in_previous(previous_states, new_west, new_east, not self.boat_side):
                    child = WGC_Node(new_west, new_east, not self.boat_side, [])
                    children.append(child)
                    parent_map[child] = self
            if sorted(self.west) not in WGC_Node.incompatibilities and not WGC_Node.state_in_previous(previous_states, self.west[:], self.east[:], not self.boat_side):
                child = WGC_Node(self.west[:], self.east[:], not self.boat_side, [])
                children.append(child)
                parent_map[child] = self
```

```
            else:
                for i in self.east:
                    new_west = self.west[:]
                    new_west.append(i)
                    new_east = self.east[:]
                    new_east.remove(i)
                      if sorted(new_east) not in WGC_Node.incompatibilities and
not   WGC_Node.state_in_previous(previous_states,   new_west,   new_east,   not
self.boat_side):
                                        child  =  WGC_Node(new_west,  new_east,  not
self.boat_side, [])
                            children.append(child)
                            parent_map[child] = self
                 if sorted(self.east) not in WGC_Node.incompatibilities and not
WGC_Node.state_in_previous(previous_states,   self.west[:],   self.east[:],   not
self.boat_side):
                                child  =  WGC_Node(self.west[:],  self.east[:],  not
self.boat_side, [])
                    children.append(child)
                    parent_map[child] = self
            self.children = children

        @staticmethod
        def state_in_previous(previous_states, west, east, boat_side):
            return any(
                sorted(west) == sorted(i.west) and
                sorted(east) == sorted(i.east) and
                boat_side == i.boat_side
                for i in previous_states
            )


    def find_solution(root_node, use_bfs=False):
        '''
        Find a solution to the WGC Problem.
        use_bfs: False for DFS, True for BFS
        '''
        to_visit = [root_node]
        node = root_node
        previous_states = []
        parent_map = {root_node: None}
        while to_visit:
```

```
            node = to_visit.pop()
                if not WGC_Node.state_in_previous(previous_states, node.west,
node.east, node.boat_side):
                    previous_states.append(node)
                node.generate_children(previous_states, parent_map)
                if use_bfs:
                    to_visit = node.children + to_visit
                else:
                    to_visit = to_visit + node.children
                if sorted(node.east) == ["c", "g", "w"]:
                    solution = []
                    while node is not None:
                        solution = [node] + solution
                        node = parent_map[node]
                    return solution
            return None

        if __name__ == "__main__":
            root = WGC_Node()
            solution = find_solution(root, use_bfs=False)
            if isinstance(solution, type(None)):
                print("No solvution is found")
            else:
                print("DFS solution = [", end='')
                for i in solution:
                    print(i, '\b, ', end='')
                print("\b\b]")

                solution = find_solution(root, use_bfs=True)
                print("BFS solution = [", end='')
                for i in solution:
                    print(i, '\b, ', end='')
                print("\b\b]")
```

Результати виводу при різних варіянтах (рис.2-4):

```
C:\Users\User\AppData\Local\Programs\Python\Python310\python.exe D:\wolfgoatcabbage-main\main.py
No solvution is found

Process finished with exit code 0
```

Рисунок 2 - При не знаходженні результатів. При даних

incompatibilities = [["c", "g", "w"], ["g", "w"],["c", "g"]]

```
DFS solution = [['w', 'g', 'c'][]Left,
['w', 'c']['g']Right,
['w', 'c']['g']Left,
['w']['g', 'c']Right,
['w', 'g']['c']Left,
['g']['c', 'w']Right,
['g']['c', 'w']Left,
[]['c', 'w', 'g']Right,
]
BFS solution = [['w', 'g', 'c'][]Left,
 ['w', 'c']['g']Right,
 ['w', 'c']['g']Left,
 ['w']['g', 'c']Right,
 ['w', 'g']['c']Left,
 ['g']['c', 'w']Right,
 ['g']['c', 'w']Left,
 []['c', 'w', 'g']Right,
]
```

Рисунок 3 - Результат перевезень однаковий знайдений алгоритмами пошуку. При даних `west=["w", "g", "c"], east=[]`

```
DFS solution = [['w', 'g']['c']Left,
['w']['c', 'g']Right,
['w', 'c']['g']Left,
['c']['g', 'w']Right,
['c', 'g']['w']Left,
['g']['w', 'c']Right,
['g']['w', 'c']Left,
[]['w', 'c', 'g']Right,
]
BFS solution = [['w', 'g']['c']Left,
 ['g']['c', 'w']Right,
 ['g']['c', 'w']Left,
 []['c', 'w', 'g']Right,
]
```

Рисунок 4 - Результати, що зображують більшу ефективність пошуку в ширину. При значеннях `west=["w", "g"], east=["c"]`.

Висновки

Було вивчено використання методів пошуку при рішенні задач та ознайомлено з використанням інтелектуальних агентів, під терміном інтелектуальний агент розуміють розумні сутності, що спостерігають за навколишнім середовищем і діють у ньому, при цьому їхня поведінка раціональна в тому розумінні, що вони здатні до розуміння і їхні дії завжди спрямовані на досягнення якої-небудь мети. Такий агент може бути як роботом, так і вбудованою програмною системою.

Також було вирішено типову задачу для штучного інтелекту перетину річки “Вовк, коза і капуста”.

Список джерел інформації

1 Artificial Intelligence A Modern Approach Second edition Stuart J. Russel and Peter Norvig p.110

2 Стаття в Вікіпедії «Задача про вовка, козу і капусту». URL: https://uk.wikipedia.org/wiki/%D0%97%D0%B0%D0%B4%D0%B0%D1%87%D0%B0_%D0%BF%D1%80%D0%BE_%D0%B2%D0%BE%D0%B2%D0%BA%D0%B0,_%D0%BA%D0%BE%D0%B7%D1%83_%D1%96_%D0%BA%D0%B0%D0%BF%D1%83%D1%81%D1%82%D1%83