Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра автоматизованих систем обробки інформації та управління

Лабораторна робота № 3

З дисципліни «Основи штучного інтелекту»

на тему: “Локальний пошук”

Виконав студент гр. ІС-72

Кривохижа Р.А.

Перевірила ст. вик. каф. АСОІУ

Мажара О. О

Київ

2020

**Примітки**

Лабораторну роботу я здавав на парі 09.04.2020.

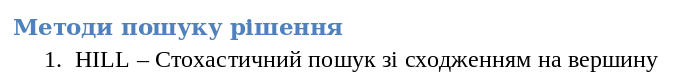
GitHub: https://github.com/kryvokhyzha/Courses/tree/master/AI/Lab3

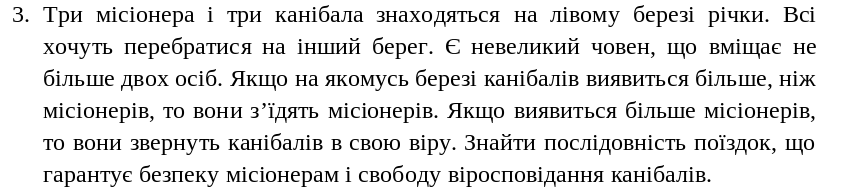
**Мета роботи**

Дослідити роботу алгоритму методу локального пошуку: стохастичного пошук зі сходженням на вершину.

**Завдання до роботи**







1. **Текст розробленого програмного забезпечення з коментарями.**
2. Файл main.py:

import sys

from state import State

from computation import expand

from math import inf

from copy import deepcopy

import random

def stochastic\_hill\_climbing(layer):

state\_number = 0

global\_best\_h = -1 \* inf

visited = []

while True:

state\_number = state\_number + 1

print("Visited states: ", state\_number)

if not layer:

return None

# best\_idx = get\_idx\_of\_best\_state()

# if best\_idx is None:

# return None

# best\_state = layer.pop(best\_idx)

best\_state = get\_randomly\_best(layer, global\_best\_h)

if best\_state is None:

return None

if best\_state.h < global\_best\_h:

return None

else:

global\_best\_h = best\_state.h

if sum(best\_state.shore) == 6:

return best\_state

print('Depth:', best\_state.depth)

print('current', best\_state.shore)

layer = expand(best\_state, layer, visited)

visited.append(best\_state)

print()

def get\_idx\_of\_best\_state(layer):

best\_h = -1 \* inf

best\_idx = None

for idx, state in enumerate(layer):

if state.h >= best\_h:

best\_h = state.h

best\_idx = idx

if best\_idx is not None:

return best\_idx

else:

return None

def get\_randomly\_best(x, global\_best):

while True:

if len(x) == 0:

break

elem = random.choice(x)

if elem.h < global\_best:

x.remove(elem)

else:

return elem

return None

def get\_res(state, buffer):

if state.prev\_state is None:

return buffer

return get\_res(state.prev\_state, buffer + str(state.shore) + ' <- ')

if \_\_name\_\_ == '\_\_main\_\_':

couples\_number = 3

boat\_capacity = 2

initial = State([], 0)

init\_layer = [] # open list

initial.shore.extend([0 for \_ in range(couples\_number \* 2)])

print('Initial shore:', initial.shore)

init\_layer.append(initial)

best\_goal = None

best\_depth = inf

for i in range(1000):

goal = stochastic\_hill\_climbing(init\_layer)

if goal is None:

continue

if best\_depth > goal.depth:

best\_goal = deepcopy(goal)

best\_depth = best\_goal.depth

if best\_goal is None:

print('\nCan`t find path!')

sys.exit(-1)

print('---------------------------------------------------------')

print("\nSuccess: ", best\_goal.shore)

print("Depth: ", best\_goal.depth)

path = get\_res(best\_goal, '') + str(initial.shore)

print("Path: ", path)

Файл computation.py:

from copy import deepcopy

def is\_not\_valid(current, couples\_number=3):

"""

Check condition for current state.

Args:

current:

couples\_number:

Returns:

"""

cp = [change\_position(s) for s in current.shore]

missionary\_left = sum(cp[:3])

missionary\_right = sum(current.shore[:3])

cannibal\_left = sum(cp[3:])

cannibal\_right = sum(current.shore[3:])

if missionary\_left >= 0 and missionary\_right >= 0 \

and cannibal\_left >= 0 and cannibal\_right >= 0 \

and (missionary\_left == 0 or missionary\_left >= cannibal\_left) \

and (missionary\_right == 0 or missionary\_right >= cannibal\_right):

print(current.shore, '-', 'X')

return False

else:

print(current.shore, '-', 'is not good way')

return True

# for i in range(0, couples\_number):

# if current.shore[i] != current.shore[couples\_number + i]:

# for j in range(couples\_number, couples\_number \* 2):

# if current.shore[j] == current.shore[i]:

# print(current.shore, '-', 'X')

# return True

# print(current.shore, '-', 'is not good way')

# return False

# if sum(current.shore[:3]) == sum(current.shore[3:]) or \

# (sum(current.shore[:3]) >= 0 and sum(current.shore[3:]) == 0) or \

# (sum(current.shore[:3]) == 0 and sum(current.shore[3:]) >= 0) or \

# (sum(cp[:3]) >= 0 and sum(cp[3:]) == 0) or \

# (sum(cp[:3]) == 0 and sum(cp[3:]) >= 0) or \

# sum(cp[:3]) == sum(cp[3:]):

# print(current.shore, '-', 'X')

# return False

# print(current.shore, '-', 'is not good way')

# return True

def change\_position(bit):

"""

Used to change position of people or the boat.

Args:

bit:

Returns:

"""

return abs(bit - 1)

def people\_near\_boat(state):

"""

Get people on the same side as the boat.

Args:

state:

Returns:

"""

people = deepcopy(state.shore)

for i in range(0, len(state.shore)):

if state.shore[i] == state.boat:

people[i] = True

else:

people[i] = False

return people

def is\_visited(state, visited):

"""

Determines whether a State has already been visited.

Args:

state:

visited:

Returns:

"""

for visited\_state in visited:

if state.shore == visited\_state.shore and state.boat == visited\_state.boat:

return True

return False

def move(cap, state, movement, result, start):

"""

Computes all possible moves from a current State with a certain boat capacity.

Args:

cap:

state:

movement:

result:

start:

Returns:

"""

for i in range(start, len(state.shore)):

if people\_near\_boat(state)[i]:

movement.append(i)

if cap > 1:

# without duplicates (permutations)

move(cap - 1, state, movement, result, i)

if cap == 1:

result.append(deepcopy(movement))

movement.pop()

return result

def expand(state, layer, visited, couples\_number=3, boat\_capacity=2):

"""

Expand and add new states to layer.

Args:

state:

layer:

visited:

couples\_number:

boat\_capacity:

Returns:

"""

result = []

new\_layer = []

# get all possible moves for the current State and capacity

possible\_moves = move(boat\_capacity, state, [], result, 0)

for pos\_move in possible\_moves:

# create next state

following\_state = deepcopy(state)

# move one or two person

if len(set(pos\_move)) == 1:

following\_state.shore[pos\_move[0]] = change\_position(state.shore[pos\_move[0]])

else:

for person\_idx in pos\_move:

following\_state.shore[person\_idx] = change\_position(state.shore[person\_idx])

# move boat

following\_state.boat = change\_position(state.boat)

if is\_visited(following\_state, visited):

continue

elif is\_not\_valid(following\_state):

visited.append(following\_state)

else:

following\_state.depth += + 1

following\_state.prev\_state = state

new\_layer.append(following\_state)

return new\_layer

Файл state.py:

class State:

shore = None

boat = 0

depth = 0

prev\_state = None

def \_\_init\_\_(self, shore=None, boat=0):

if shore is None:

shore = []

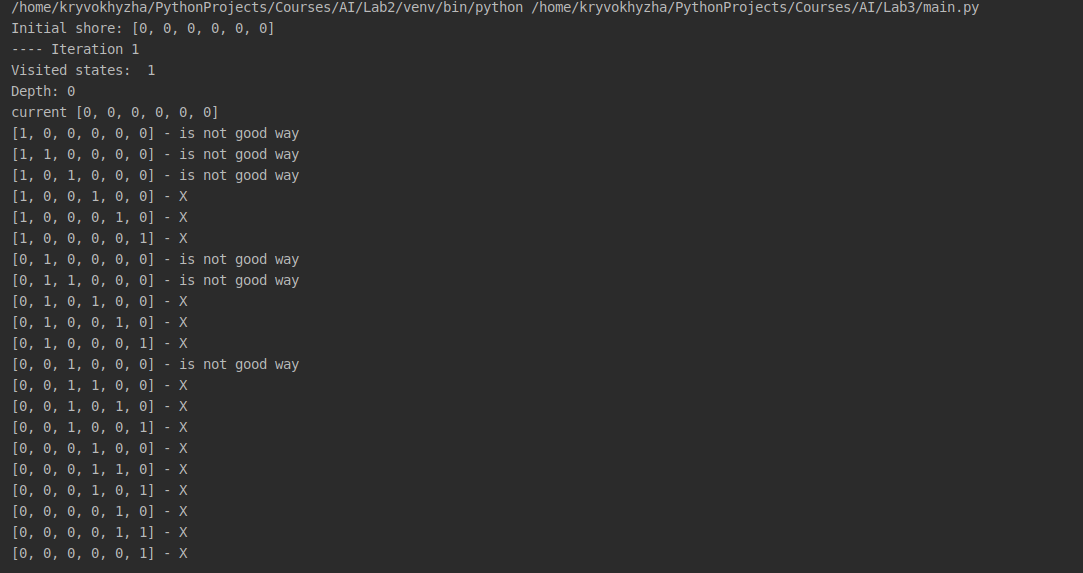
self.shore = shore

self.boat = boat

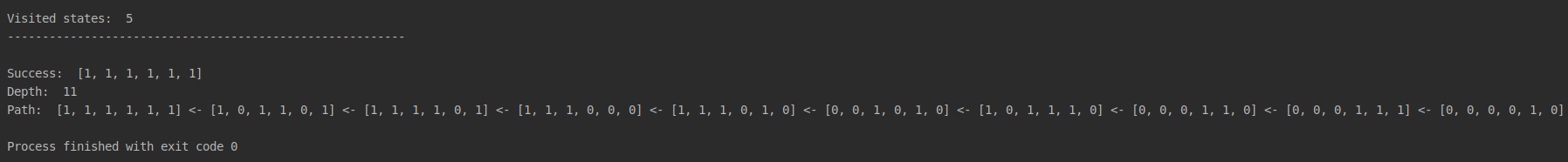
self.depth = 0

self.prev\_state = None

self.h = sum(shore) / 2

1. **Результати роботи програмного забезпечення, що включають результати тестування та копії екранних форм.**
2. ****

Мал.1 перша ітерація алгоритму



Мал. 2 остання ітерація алгоритму

1. **Висновки, що відображають особисто отримані результати виконання роботи, їх критичний аналіз.**

Для знаходження допустимого розв’язку даної задачі я використовував алгоритм Stochastic Hill Climbing. Спочатку, для певного стану визначались всі дочірні, тоді серед них випадковим чином обирався один. Для обраного стану я перевіряв чи має він кращу евристичну функцію. Якщо так, то процес повторювався вже для нового стану, якщо ні, то продовжував випадковим чином обирати стани з сусідніх.

Даний алгоритм я застосовував близько 1000 разів для заданого початкового стану. Найкращий отриманий результат мав глубину 11.

**Контрольні запитання**

1. Яка відмінність між алгоритмами звичайного пошуку та локального пошуку?

*Основною метою пошуку є не знаходження оптимального шляху до цільової точки, а оптимізація деякої цільової функції. Задача зводиться до пошуку стану глобального максимуму (або мінімуму) на даному просторі станів.*

1. Поясніть в чому полягає складність роботи алгоритмів локального пошуку (пов’язана з ландшафтом станів)

*Складність полягає в тому, щоб серед деякої кількості локальних максимумів (мінімумів) обрати найкращий, тобто глобальний. Використовуючи неоптимізовані алгоритми локального пошуку, із високою ймовірністю можна не потрапити у точку глобального максимуму (мінімуму).*

1. Що таке задачі з обмеженнями (CSP)? Як вони сформульовані?

*Задача виконання обмежень (Constraint satisfaction problem) — це математичні проблеми, визначені як сукупність об'єктів, стан яких має задовольняти ряду* [*обмежень*](https://uk.wikipedia.org/wiki/Обмеження_(математика))*. ЗВО представляє сутності проблеми як однорідний набір обмежень, що накладаються на* [*змінні*](https://uk.wikipedia.org/wiki/Змінна)*, які розв'язуються методами* [*виконання обмежень*](https://uk.wikipedia.org/w/index.php?title=Виконання_обмежень&action=edit&redlink=1)*.*

*ЗВО формулюється множиною змінних, областю значень і множиною обмежень. Змінні представляються кортежами, кількість яких відповідає кількості обмежень. Розв’язком є таке значення, при я кому усі змінні задовольняють усім обмеженням.*

1. Які складності виникають при розв’язанні задач у неперервних просторах станів?

*Проблеми дослідження виникають, якщо агент не має жодного уявлення про те, які полягання і дії в його середовищі, тому варто скласти карту і знайти мету, якщо вона існує. Якщо у дискретному просторі можна легко представити, як зробити такий ландшафт, то у неперервному треба спочатку оцінити найбільш правильні інтервали для переміщення зі стану на стан за допомогою статистичних методів.*

1. Поясніть принцип роботи алгоритмів, які реалізовані у лабораторній роботі.

*Стратегія алгоритму Stochastic HILL climb полягає в тому, щоб повторити процес випадкового вибору сусіда для рішення кандидата і прийняти його лише в тому випадку, якщо це призведе до поліпшення.*