МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

Новгородский государственный университет имени Ярослава Мудрого (НовГУ) Кафедра «Электронных и информационных систем»

**Методы поиска в пространстве состояний: изучение методов перебора и критериев оценки методов перебора на произвольных графах**

Лабораторная работа по дисциплине:

«Системы искусственного интеллекта»

Студент гр. 1095

Золотов Э. В.

(подпись)

«\_\_\_»\_ 2024 г.

Преподаватель

\_\_Михайлов Д.В

(подпись)

« » 2024 г.

г. Великий Новгород

-2024-

# ЦЕЛЬ РАБОТЫ

Изучение методов перебора и критериев оценки методов перебора на произвольных графах.

# ПОСТАНОВКА ЗАДАЧИ

Дан кувшин с водой емкостью N и пустой кувшин емкостью M. Требуется получить заданную емкость L. Воду можно либо выливать, либо переливать из одного кувшина в другой (Кувшины можно полностью наполнять водой из неограниченного резервуара)

# ОПИСАНИЕ ПРЕДСТАВЛЕНИЯ В ПРОСТРАНСТВЕ СОСТОЯНИЙ

Состояние представляет собой пару чисел, обозначающий текущий уровень воды в кувшинах. Первое число представляет уровень воды в кувшине емкостью N, второе – уровень воды в кувшине емкостью M.

Операторы: Вылить воду из кувшина N; Вылить воду из кувшина M; Наполнить кувшин N; Наполнить кувшин M; Перелить воду из кувшина N в кувшин M; Перелить воду из кувшина M в кувшин N

Критерием достижения цели является достижение заданного уровня воды L в одном из кувшинов.

# ПРЕДСТАВЛЕНИЕ ПРОСТРАНСТВА СОСТОЯНИЙ В ВИДЕ ГРАФА

Пускай кувшин с водой емкостью 5, пустой кувшин емкостью 3. Требуется получить заданную емкость 4.

Представим пространство состояний для задачи с кувшинами в виде ориентированного графа, где вершины будут представлять собой состояния кувшинов (количество воды в каждом кувшине), а ребра будут представлять возможные переходы между состояниями (операции наполнения, выливания и переливания воды).

Таким образом, можно создать вершины графа для каждой комбинации воды в кувшинах. К примеру, для первого и второго кувшина можно создать вершины для состояний (0, 0), (0, 3), (5, 0), (5, 3) и так далее, где первое число обозначает количество воды в первом кувшине, а второе - количество воды в втором кувшине.

Ребра между вершинами представляют возможные переходы между состояниями. Например, из состояния (0, 0) мы можем перейти к состояниям (5, 0), (0, 3) и (0, 0)

# КРИТЕРИИ ОЦЕНКИ КАЧЕСТВА МЕТОДОВ ПЕРЕБОРА

Необходимо оценить качество работы методов перебора, используемых для решения задачи, по критерию целенаправленности.

*Целенаправленность перебора* – позволяет узнать, в какой мере перебор идет в направлении цели, а не ведется по нежелательным направлениям.

Определяется соотношением: P= ,

где L – длина найденного пути до цели,

T – общее число построенных в течение перебора вершин (включая целевую, но исключая начальную).

# РЕЗУЛЬТАТЫ ИССЛЕДОВАНИЯ

Необходимо оценить качество работы методов перебора, используемых для решения задачи, по критерию целенаправленности.

, где

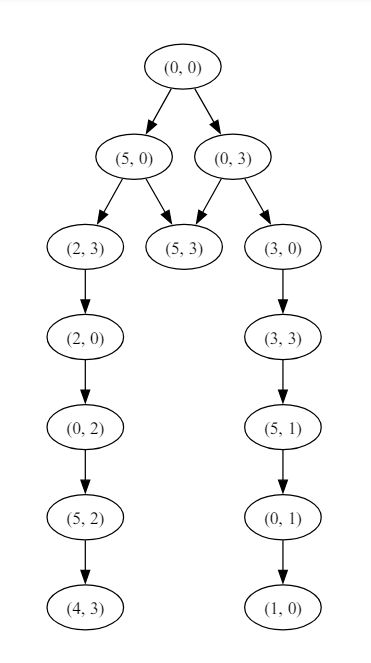
L – длина найденного пути до цели.

T – количество построенных вершин.

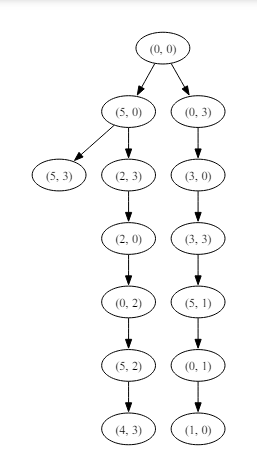
Набор текстовых данных: кувшин с водой 5, кувшин без воды 3. Требуемая емкость 4.

Оценка по целенаправленности

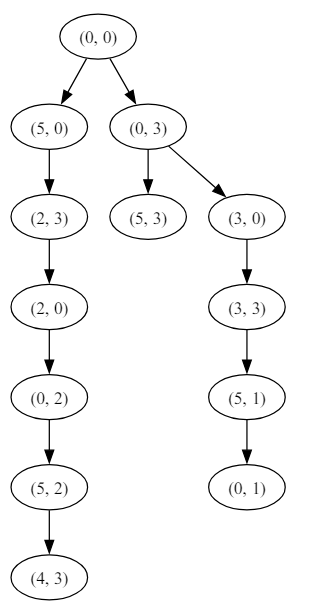
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Метод полного перебора** | **Метод равных цен** | **Метод перебора в глубину** | **Оптимальный алгоритм перебора A\*** |
| P | 0,4615 | 0,75 | 0,5 | 0,4615 |



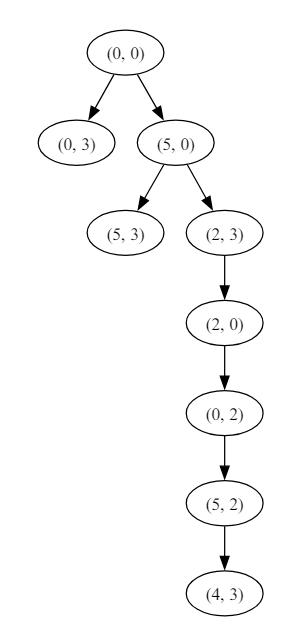
*Рисунок №1. Метод A\**



*Рисунок №2. Метод полного перебора*

**

*Рисунок №3. Метод перебора в глубину*

**

*Рисунок №4. Метод равных цен*

Набор текстовых данных: кувшин с водой 8, кувшин без воды 7. Требуемая емкость 2.

Оценка по целенаправленности

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Метод полного перебора** | **Метод равных цен** | **Метод перебора в глубину** | **Оптимальный алгоритм перебора A\*** |
| P | 0,46 | 0,75 | 0,6 | 0.5 |

***Таблица 1***

Набор текстовых данных: кувшин с водой 8, кувшин без воды 7. Требуемая емкость 5.

Оценка по целенаправленности

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Метод полного перебора** | **Метод равных цен** | **Метод перебора в глубину** | **Оптимальный алгоритм перебора A\*** |
| P | 0,44 | 0,9 | 0,8 | 0.53 |

# ВЫВОД

В результате выполнения лабораторной работы были изучены методы поиска по графу и сделаны следующие выводы: для данной задачи эффективными методами показали себя: Метод равных цен и метод перебора в глубину.

# ПРИЛОЖЕНИЕ A

# Метод полного перебора

**from** **queue** **import** Queue

**from** **graphviz** **import** Digraph

**def** **is\_goal**(state, target):

**return** state[**0**] == target **or** state[**1**] == target

**def** **find\_moves**(state, jug\_1, jug\_2):

moves = []

state\_jug\_1, state\_jug\_2 = state

**if** state\_jug\_1 > **0**:

moves.append((**0**, state\_jug\_2))

**if** state\_jug\_2 > **0**:

moves.append((state\_jug\_1, **0**))

**if** state\_jug\_1 < jug\_1:

moves.append((jug\_1, state\_jug\_2))

**if** state\_jug\_2 < jug\_2:

moves.append((state\_jug\_1, jug\_2))

**if** state\_jug\_1 != **0** **and** jug\_2-state\_jug\_2 >= state\_jug\_1:

moves.append((**0**, state\_jug\_2+state\_jug\_1))

**if** state\_jug\_2 != **0** **and** jug\_1-state\_jug\_1 >= state\_jug\_2:

moves.append((state\_jug\_1+state\_jug\_2, **0**))

**if** state\_jug\_2 != **0** **and** **0** < jug\_1-state\_jug\_1 < state\_jug\_2:

moves.append((jug\_1, state\_jug\_2 - (jug\_1 - state\_jug\_1)))

**if** state\_jug\_1 != **0** **and** **0** < jug\_2-state\_jug\_2 < state\_jug\_1:

moves.append((state\_jug\_1 - (jug\_2 - state\_jug\_2), jug\_2))

**return** moves

**def** **bfs**(jug\_1, jug\_2, target):

queue = Queue()

queue.put((**0**, **0**))

visited = [str((**0**, **0**))]

graph = Digraph()

graph.node(str((**0**, **0**)))

steps = **0**

**while** **not** queue.empty():

state = queue.get()

**if** is\_goal(state, target):

print(f"Требуется шагов: {steps}")

**return** graph

steps += **1**

**for** move **in** find\_moves(state, jug\_1, jug\_2):

**if** str(move) **not** **in** visited:

visited.append(str(move))

graph.node(str(move))

graph.edge(str(state), str(move))

queue.put(move)

**return** **None**

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

jug\_1, jug\_2, target =**8**, **7**, **5**

graph = bfs(jug\_1, jug\_2, target)

**if** graph **is** **not** **None**:

graph.render('bfs\_tree')

graph.view()

**else**:

print(f"Нельзя с такими кувшинами получить {target} л.")

# Метод равных цен

**from** **graphviz** **import** Digraph

**from** **queue** **import** LifoQueue

**def** **is\_goal**(state, target):

**return** state[**0**] == target **or** state[**1**] == target

**def** **find\_moves**(state, jug\_1, jug\_2):

moves = []

state\_jug\_1, state\_jug\_2 = state

**if** state\_jug\_2 > **0**:

moves.append((state\_jug\_1, **0**))

**if** state\_jug\_1 > **0**:

moves.append((**0**, state\_jug\_2))

**if** state\_jug\_2 < jug\_2:

moves.append((state\_jug\_1, jug\_2))

**if** state\_jug\_1 < jug\_1:

moves.append((jug\_1, state\_jug\_2))

**if** state\_jug\_2 != **0** **and** jug\_1-state\_jug\_1 >= state\_jug\_2:

moves.append((state\_jug\_1+state\_jug\_2, **0**))

**if** state\_jug\_1 != **0** **and** jug\_2-state\_jug\_2 >= state\_jug\_1:

moves.append((**0**, state\_jug\_2+state\_jug\_1))

**if** state\_jug\_1 != **0** **and** **0** < jug\_2-state\_jug\_2 < state\_jug\_1:

moves.append((state\_jug\_1 - (jug\_2 - state\_jug\_2), jug\_2))

**if** state\_jug\_2 != **0** **and** **0** < jug\_1-state\_jug\_1 < state\_jug\_2:

moves.append((jug\_1, state\_jug\_2 - (jug\_1 - state\_jug\_1)))

**return** moves

**def** **dfs**(jug\_1, jug\_2, target):

visited = [str((**0**, **0**))]

stack = LifoQueue()

stack.put((**0**, **0**))

steps = **0**

graph = Digraph()

graph.node(str((**0**, **0**)))

**while** **not** stack.empty():

state = stack.get()

**if** is\_goal(state, target):

print(f"Требуется шагов: {steps}")

**return** graph

steps += **1**

**for** move **in** find\_moves(state, jug\_1, jug\_2):

**if** str(move) **not** **in** visited:

visited.append(str(move))

graph.node(str(move))

graph.edge(str(state), str(move))

stack.put(move)

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

jug\_1, jug\_2, target =**8**, **7**, **5**

graph = dfs(jug\_1, jug\_2, target)

**if** graph **is** **not** **None**:

graph.render('dfs\_tree')

graph.view()

**else**:

print(f"Нельзя с такими кувшинами получить {target} л.")

# Метод поиска в глубину

**from** **queue** **import** PriorityQueue

**from** **graphviz** **import** Digraph

**def** **heuristic**(state, target):

**return** abs(state[**0**] - target) + abs(state[**1**] - target)

**def** **is\_goal**(state, target):

**return** state[**0**] == target **or** state[**1**] == target

**def** **find\_moves**(state, jug\_1, jug\_2):

moves = []

state\_jug\_1, state\_jug\_2 = state

**if** state\_jug\_1 > **0**:

moves.append((**0**, state\_jug\_2))

**if** state\_jug\_2 > **0**:

moves.append((state\_jug\_1, **0**))

**if** state\_jug\_1 < jug\_1:

moves.append((jug\_1, state\_jug\_2))

**if** state\_jug\_2 < jug\_2:

moves.append((state\_jug\_1, jug\_2))

**if** state\_jug\_1 != **0** **and** jug\_2-state\_jug\_2 >= state\_jug\_1:

moves.append((**0**, state\_jug\_2+state\_jug\_1))

**if** state\_jug\_2 != **0** **and** jug\_1-state\_jug\_1 >= state\_jug\_2:

moves.append((state\_jug\_1+state\_jug\_2, **0**))

**if** state\_jug\_2 != **0** **and** **0** < jug\_1-state\_jug\_1 < state\_jug\_2:

moves.append((jug\_1, state\_jug\_2 - (jug\_1 - state\_jug\_1)))

**if** state\_jug\_1 != **0** **and** **0** < jug\_2-state\_jug\_2 < state\_jug\_1:

moves.append((state\_jug\_1 - (jug\_2 - state\_jug\_2), jug\_2))

**return** moves

**def** **wide\_search**(jug\_1, jug\_2, target):

graph = Digraph()

graph.node(str((**0**, **0**)))

steps = **0**

queue = PriorityQueue()

queue.put((**0**, (**0**, **0**)))

visited = [str((**0**, **0**))]

**while** **not** queue.empty():

\_, state = queue.get()

**if** is\_goal(state, target):

print(f"Требуется шагов: {steps}")

**return** graph

steps += **1**

**for** move **in** find\_moves(state, jug\_1, jug\_2):

**if** str(move) **not** **in** visited:

visited.append(str(move))

graph.node(str(move))

graph.edge(str(state), str(move))

cost = heuristic(move, target) + **1**

queue.put((cost, move))

**return** **None**

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

jug\_1, jug\_2, target =**8**, **7**, **5**

graph = wide\_search(jug\_1, jug\_2, target)

**if** graph **is** **not** **None**:

graph.render('wide\_tree')

graph.view()

**else**:

print(f"Нельзя с такими кувшинами получить {target} л.")

# Метод A\*

**from** **queue** **import** PriorityQueue

**from** **graphviz** **import** Digraph

**def** **heuristic**(state, target):

**return** abs(state[**0**] - target) + abs(state[**1**] - target)

**def** **is\_goal**(state, target):

**return** state[**0**] == target **or** state[**1**] == target

**def** **find\_moves**(state, jug\_1, jug\_2):

moves = []

state\_jug\_1, state\_jug\_2 = state

**if** state\_jug\_1 > **0**:

moves.append((**0**, state\_jug\_2))

**if** state\_jug\_2 > **0**:

moves.append((state\_jug\_1, **0**))

**if** state\_jug\_1 < jug\_1:

moves.append((jug\_1, state\_jug\_2))

**if** state\_jug\_2 < jug\_2:

moves.append((state\_jug\_1, jug\_2))

**if** state\_jug\_1 > **0** **and** state\_jug\_2 < jug\_2:

amount\_to\_pour = min(state\_jug\_1, jug\_2 - state\_jug\_2)

moves.append((state\_jug\_1 - amount\_to\_pour, state\_jug\_2 + amount\_to\_pour))

**if** state\_jug\_2 > **0** **and** state\_jug\_1 < jug\_1:

amount\_to\_pour = min(state\_jug\_2, jug\_1 - state\_jug\_1)

moves.append((state\_jug\_1 + amount\_to\_pour, state\_jug\_2 - amount\_to\_pour))

**return** moves

**def** **a\_star\_search**(jug\_1, jug\_2, target):

graph = Digraph()

graph.node(str((**0**, **0**)))

steps = **0**

queue = PriorityQueue()

queue.put((**0**, [(**0**, **0**)]))

visited = []

**while** **not** queue.empty():

path\_cost, path = queue.get()

state = path[-**1**]

**if** is\_goal(state, target):

print(f"Требуется шагов: {steps}")

print(path)

**return** graph

steps += **1**

visited.append(str((state)))

**for** move **in** find\_moves(state, jug\_1, jug\_2):

**if** str(move) **not** **in** visited:

graph.node(str(move))

graph.edge(str(state), str(move))

new\_path\_cost = path\_cost + **1**

new\_heuristic = heuristic(move, target)

total\_cost = new\_path\_cost + new\_heuristic

new\_path = path + [move]

queue.put((total\_cost, new\_path))

**return** **None**

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

jug\_1, jug\_2, target =**8**, **7**, **5**

graph = a\_star\_search(jug\_1, jug\_2, target)

**if** graph **is** **not** **None**:

graph.render('a\_tree')

graph.view()

**else**:

print(f"Нельзя с такими кувшинами получить {target} л.")