**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра ВТ**

отчет

**по лабораторной работе №1**

**по дисциплине «Искусственный интеллект»**

**Тема: «Неинформированный поиск (императивный ЯП)»**

|  |  |  |
| --- | --- | --- |
| Студенты гр. 9308 |  | Хамитов А.К.  Яловега Н.В.  Дементьев Д.П. |
| Преподаватель |  |  |

Санкт-Петербург

2022

# Цель работы

Практическое закрепление понимания общих идей поиска в пространстве состояний и стратегий слепого поиска.

# Постановка задачи

Рассматривается задача **«**Головоломка 8-ка**»**.

Задана доска с 8 пронумерованными фишками и с одним пустым участком.

Фишка, смежная с пустым участком, может быть передвинута на этот участок. Требуется достичь указанного целевого состояния.

Нашей задачей является написание на языке Python алгоритмов поиска пути из начального состояния в конечное, которые изображены на рисунках 1 и 2 соответственно.

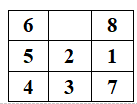


Рисунок 1. Начальное состояние

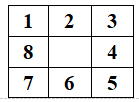


Рисунок 2. Конечное состояние

Написаны алгоритмы:

1. Поиск в глубину
2. Двунаправленный поиск

# **Описание выбранных структур данных**

|  |  |  |  |
| --- | --- | --- | --- |
| Структура | Поля | Описание | |
| class **Node** | **State: list** | Состояние в пространстве остояний, которому соответствует данный узел | |
| **Parent: Node** | Указатель на родительский узел | |
| **Previous\_action: Action** | Действие, которое было применено к родительскому узлу для формирования данного узла | |
| **Path\_cost: int** | Стоимость пути от начального состояния до данного узла | |
| **Depth: int** | Количество этапов пути от начального состояния (глубина) | |
| **Node\_id: int** | Уникальный идентификатор узла | |
| **\_\_Nodes\_count: int** | | Общая переменная среди всех объектов, содержащая количество узлов |
| class **Tree** | **Hashset: dict** | Словарь с хэшами (уникальные идентификаторы) каждого узла | |

# **Описание методов класса Tree**

|  |  |
| --- | --- |
| Метод | Описание |
| **get\_root(self)->list** | Получение корня дерева |
| **get\_node(self, node\_id: int)->int** | Получение узла по его идентификатору |
| **add\_node(self, node\_id: int)->void** | Добавление узла в дерево |
| **is\_int\_tree(self, new\_node)->bool** | Проверка: на наличие узла в дереве |
| **get\_path(self, node)->list** | Получение пути до узла, указанного в параметре |

# **Описание методов класса Node**

|  |  |
| --- | --- |
| Метод | Описание |
| **get\_nodes\_count()->int** | Статический метод класса, возвращающий количество узлов |

# Описание алгоритмов

## **Алгоритм обходя в глубину:**

Обход в глубину реализуется с помощью стека.

Инициализация:

Добавляем исходную вершину в стек.

Множество посещенных вершин visited.

Очередной шаг алгоритма:

Пока стек не пуст выполнять:

1. Достаем вершину из стека и добавляем ее в множество посещенных состояний.
2. Проверяем вершину на конечность, если конечна, то заканчиваем алгоритм, иначе продолжаем
3. Пробегаем по всем вершинам соседей, проверяем их на наличие во множестве посещенных состояний, если нет, то добавляем их в стек.

Иначе путь не найден.

## Двунаправленный поиск:

Инициализация:

Записываем начальное и конечное состояния в собственные стеки fringe1 и fringe 2 (поиск с двух направлений).

Два множества посещенных состояний visited1 и visited2 (поиск с двух направлений).

Очередной шаг алгоритма:

Пока очереди не пусты выполнять:

1. Добавляем вершину в очередь fringe 1.
2. Если вершина в множестве visited2, то:

возвращаем путь от начальной вершины к конечной

Иначе:

Пробегаемся по соседям вершины и, если их нет в visited 1, то добавляем их в visited1 и в fringe1.

1. Добавляем вершину в очередь fringe 2.
2. Если вершина в множестве visited1, то:

возвращаем путь от начальной вершины к конечной

Иначе:

Пробегаемся по соседям вершины и, если их нет в visited 2, то добавляем их в visited2 и в fringe2.

Иначе путь не найден.

# **Результаты работы программы:**

Программа на рисунке 3 говорит о том, что мы должны передавать аргументы в командной строке.



Рисунок 3. Попытка запустить программу

Выведем справку с описанием каждого аргумента. Результат представлен на рисунке 4.

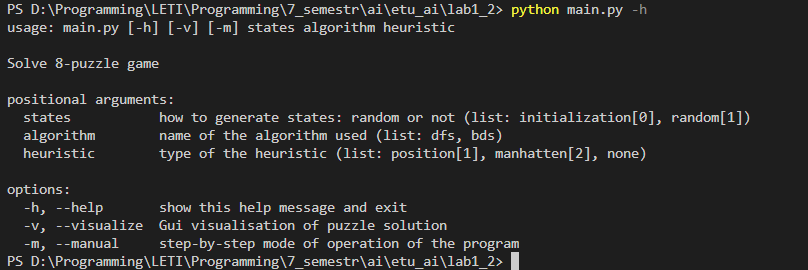


Рисунок 4. Справка аргументов

Запустим поиск в глубину с ключом визуализации. Запуститься окно с визуализацией смены каждого состояния с начального до конечного. Результат представлен на рисунках 5, 6 и 7.

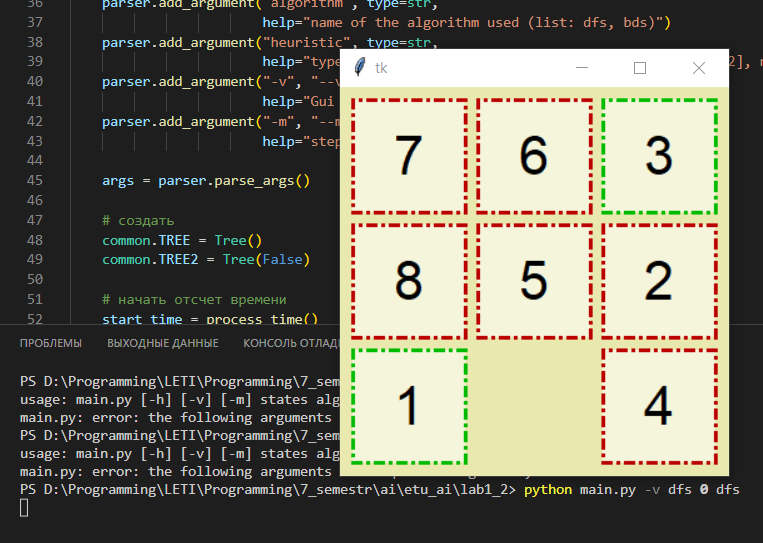
=

Рисунок 5. Поиск в глубину с ключом визуализации

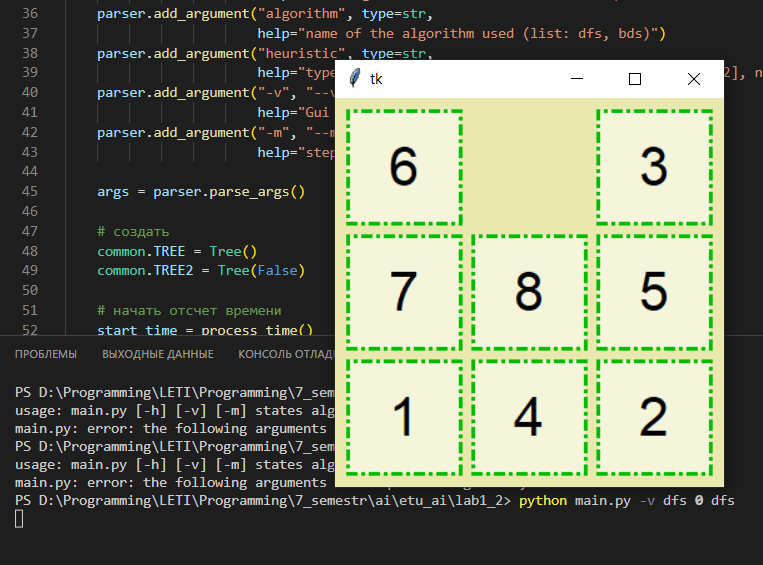


Рисунок 6. Результат поиска в глубину

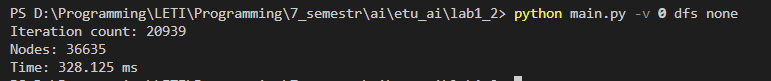


Рисунок 7. Количество итераций, созданных узлов и потраченного времени

Запустим двунаправленный поиск с ключом визуализации. Результат представлен на рисунках 8, 9 и 10.

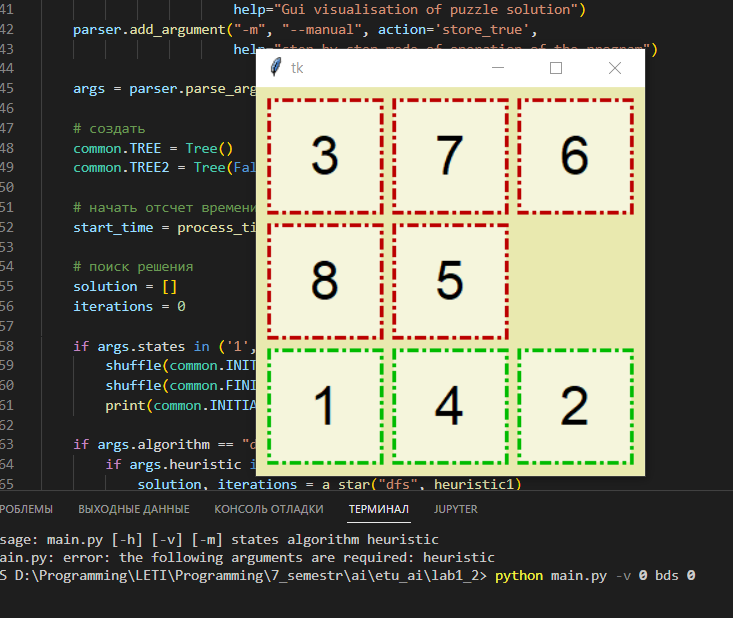


Рисунок 8. Двунаправленный поиск с ключом визуализации

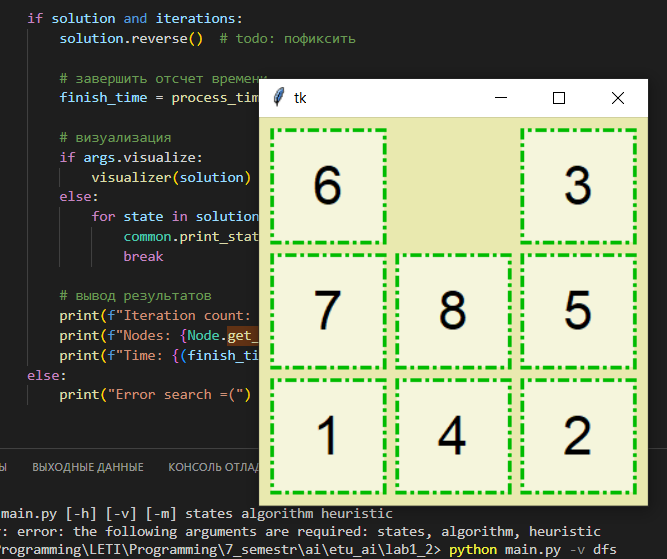


Рисунок 9. Результат двунаправленного поиска

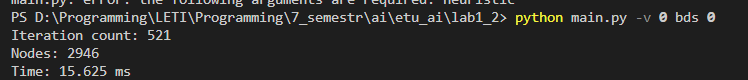


Рисунок 10. Количество итераций, созданных узлов и потраченного времени

Никита лох

Сравнительные оценки сложности алгоритмов поиска

|  |  |  |
| --- | --- | --- |
|  | Неинформированный  поиск | |
| Стратегия 1 | Стратегия 2 |
|
| Временная сложность (кол-во шагов) | 20939 | 521 |
| Емкостная сложность (кол-во вершин в дереве поиска) | 36635 | 2946 |

**Вывод**

Двунаправленный поиск (реализация через стек) сработал быстрее и был менее ресурсоёмким, чем поиск в глубину, так как поиск с одной из сторон может найти вершину, которую уже посетил второй и алгоритм на этом уже заканчивается, существенно ускоряя работу алгоритма. Все могло измениться для некоторых случаев, если бы мы использовали очередь в двунаправленном поиске.

**Листинг**