

МИСИИ

ЛАБ 1 – Python

1.6.1. Какие типы числовых данных поддерживает Python? Как определить тип переменной в Python?

Целые числа (int): представляют целочисленные значения без десятичной части, например, 5, -10.

Вещественные числа (float): представляют числа с плавающей точкой, содержащие десятичную часть, например, 3.14, -2.5.

Комплексные числа (complex): представляются в виде $a + bj$, где a и b - вещественные числа, $a j$ - мнимая единица, например, $2 + 3j$, $-1.5 + 0.5j$.

Для определения типа переменной в Python можно использовать функцию `type()`.

1.6.2. Как выполнить целочисленное деление и вычисление остатка от деления целых чисел?

Целочисленное деление и вычисление остатка от деления целых чисел можно выполнить с помощью операторов `//` и `%` соответственно.

1.6.3. Приведите примеры использования логических операций и операций отношений.

```
x = 5
```

```
y = 10
```

```
# Логические операции
```

```
print(x > 3 and y < 15) # Выводит True (оба условия выполняются)
```

```
print(x > 3 or y > 15) # Выводит True (хотя бы одно из условий выполняется)
```

```
print(not(x > 3)) # Выводит False (отрицание условия)
```

```
# Операции отношений
```

```
print(x == y) # Выводит False (x не равно y)
```

```
print(x != y) # Выводит True (x не равно y)
```

```
print(x < y) # Выводит True (x меньше y)
```

```
print(x >= y) # Выводит False (x не больше или равно y)
```

1.6.4. Как выполнить конкатенацию строк, заменить символы нижнего регистра на верхний и наоборот, определить длину строки?

Для конкатенации строк в Python используется оператор +. Для замены символов нижнего регистра на верхний и наоборот можно использовать методы upper() и lower() соответственно. Для определения длины строки используется функция len().

1.6.5. Приведите пример строки форматированного вывода в операторе print, объясните спецификации форматирования и назначение управляющих последовательностей.

```
name = "Alice"  
age = 25  
print("My name is %s and I am %d years old." % (name, age))  
# Выводит "My name is Alice and I am 25 years old."
```

1.6.6. Что такое f-стока? Приведите примеры использования.

F-строки (форматированные строки) в Python - это удобный способ вставки значений переменных в строку с использованием префикса f перед строкой.

```
name = "Alice"  
age = 25  
# Вставка значений переменных в строку с помощью f-строки  
message = f"My name is {name} and I am {age} years old."  
print(message) # Выводит "My name is Alice and I am 25 years old."  
# Выполнение арифметических вычислений в f-строке  
result = f"The result of 3 * 4 is {3 * 4}."  
print(result) # Выводит "The result of 3 * 4 is 12."
```

1.6.7. Объясните назначение методов для работы со строками: 'format', 'index', 'isalpha', 'isdigit', 'partition', 'replace', 'split', 'title', 'translate', 'upper', 'zfill'.

1.6.8. Как создать список? Как выполняется обращение к элементам списков, сегментов списков? Объясните назначение методов для работы со списками: pop, append, insert, sort, index.

```
my_list = [1, 2, 3, 4, 5]  
print(my_list[0]) # Выводит 1  
print(my_list[2]) # Выводит 3  
print(my_list[1:4]) # Выводит [2, 3, 4]
```

```
print(my_list[:3])    # Выводит [1, 2, 3]
print(my_list[2:])    # Выводит [3, 4, 5]
print(my_list[::-2])  # Выводит [1, 3, 5]
```

`pop(index)`: Удаляет элемент из списка по указанному индексу и возвращает его значение. Если индекс не указан, удаляется и возвращается последний элемент списка.

`append(item)`: Добавляет элемент в конец списка.

`insert(index, item)`: Вставляет элемент в список по указанному индексу.

`sort()`: Сортирует элементы списка по возрастанию. Если список содержит элементы разных типов данных, вызов `sort()` вызовет ошибку типа.

`index(item)`: Возвращает индекс первого вхождения элемента в списке. Если элемент не найден, вызов `index()` вызовет ошибку `ValueError`.

1.6.9. Как создать кортеж? Как его распаковать? Как обратиться к элементам кортежа? Как объединить список и кортеж?

Создание: `student_tuple = ()`

Для упаковки элементов в кортеж можно перечислить их, разделяя запятыми:

```
>>> student_tuple = 'John', 'Green', 3.3 #создание кортежа >>> student_tuple
('John', 'Green', 3.3)
>>> len(student_tuple)
>>> student_tuple[0] 'John'
```

Распаковка:

```
>>> pair = (3, 5) >>> x, y = pair
```

```
>>> x
```

```
3
```

```
>>> y
```

```
5
```

Объединение списка и кортежа:

Конструкция `+=` также может использоваться для присоединения кортежа к списку:

```
>>> numbers = [1, 2, 3, 4, 5] >>> numbers += (6, 7)
>>> numbers
```

[1, 2, 3, 4, 5, 6, 7]

1.6.10. Как создать множество? Как проверить принадлежность элемента множеству? Приведите примеры операций с множествами.

Создание:

```
>>> shapes = ['circle', 'square', 'triangle', 'circle']
```

```
>>> setOfShapes = set(shapes)
```

```
>>> 'circle' in setOfShapes
```

True

```
>>> favoriteShapes = ['circle', 'triangle', 'hexagon']
```

```
>>> setOfFavoriteShapes = set(favoriteShapes)
```

```
>>> setOfShapes - setOfFavoriteShapes
```

```
set(['square', 'polygon'])
```

```
>>> setOfShapes & setOfFavoriteShapes
```

```
set(['circle', 'triangle'])
```

```
>>> setOfShapes | setOfFavoriteShapes
```

```
set(['circle', 'square', 'triangle', 'polygon', 'hexagon'])
```

1.6.11. Что такое словарь? Как его создать? Как выполнить поиск в словаре?

Как заменить значение в словаре? Как удалить элемент словаря? Как получить список ключей, список значений, список кортежей ключ-значение?

Словарь – это структура, которая обеспечивает отображение одного объекта (ключ) на другой объект (значение).

```
>>> studentIds = {'knuth': 42.0, 'turing': 56.0, 'nash': 92.0}
```

```
>>> studentIds['turing']
```

56.0

```
>>> studentIds['nash'] = 'ninety-two'
```

```
>>> studentIds
```

```
{'knuth': 42.0, 'turing': 56.0, 'nash': 'ninety-two'}
```

```
>>> del studentIds['knuth']
```

#создание #поиск по ключу

#замена значения #удаление

```
>>> studentIds
```

```
{'turing': 56.0, 'nash': 'ninety-two'}  
>>> studentIds['knuth'] = [42.0, 'forty-two']  
>>> studentIds  
{'knuth': [42.0, 'forty-two'], 'turing': 56.0, 'nash': 'ninety-two'}  
>>> studentIds.keys()  
['knuth', 'turing', 'nash']  
>>> studentIds.values()  
[[42.0, 'forty-two'], 56.0, 'ninety-two']  
>>> studentIds.items()  
[('knuth', [42.0, 'forty-two']), ('turing', 56.0), ('nash', 'ninety-two')]  
>>> len(studentIds)  
3
```

Обратите внимание, что метод items возвращает пары – ключ-значение – в форме кортежей.

1.6.12. Напишите скрипт, печатающий номера элементов списка и сами элементы.

```
fruits = ['apples', 'oranges', 'pears', 'bananas']  
for i, fruit in enumerate(fruits): #enumerate возвращает (i, fruit)  
print('#%d:%s' % (i, fruit))
```

1.6.13. Что такое списковое включение? Приведите примеры операций отображения и фильтрации с помощью спискового включения.

```
list1=[x for x in range(1,6)]  
#отображение nums на plusOneNums  
nums = [1, 2, 3, 4, 5, 6]  
plusOneNums = [x + 1 for x in nums]  
#фильтрация списков  
# создаем список из нечетных элементов  
oddNums = [x for x in nums if  
x % 2 == 1] print(oddNums)
```

1.6.14. Что такое выражение-генератор? Приведите пример применения в цикле for.

Выражение-генератор отчасти напоминает списковое включение, но оно создает итерируемый объект-генератор, производящий значения по требованию

[5]. Этот механизм называется отложенным вычислением. В списковом включении используется быстрое вычисление, позволяющее создавать списки в момент выполнения. При большом количестве элементов создание списка может потребовать значительных затрат памяти и времени. Таким образом, выражения-генераторы могут сократить потребление памяти программой и повысить быстродействие, если не все содержимое списка понадобится одновременно.

Выражения-генераторы обладают теми же возможностями, что и списковое включение, но они определяются в круглых скобках вместо квадратных. Ниже выражение-генератор возвращает квадраты только нечетных чисел из numbers:

```
numbers = [10, 3, 7, 1, 9, 4, 2, 8, 5, 6]
for value in (x ** 2 for x in numbers if x % 2 != 0):
    print(value, end=' ')
```

В результате выполнения этого кода будут выведены числа: 9 49 1 81 25.

Чтобы показать, что выражение-генератор не создает список, присвоим выражение-генератор из предыдущего фрагмента переменной и выведем значение этой переменной:

```
>>> squares_of_odds = (x ** 2 for x in numbers if x % 2 != 0)
>>> squares_of_odds
<generator object <genexpr> at 0x1085e84c0>
```

Текст "generator object <genexpr>" сообщает, что square_of_odds является объектом-генератором, который был создан на базе выражения-генератора (genexpr).

1.6.15. Как определить собственную функцию в Python? Определите функцию быстрой сортировки.

Для определения собственной функции в Python используется ключевое слово def, за которым следует имя функции, аргументы в скобках и блок кода с отступом. Пример определения функции быстрой сортировки (QuickSort):

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
```

```
left = [x for x in arr if x < pivot]
middle = [x for x in arr if x == pivot]
right = [x for x in arr if x > pivot]
return quicksort(left) + middle + quicksort(right)
```

1.6.16. Что такое функционал? Как определяется анонимная функция?

Приведите примеры использования функционалов filter и map.

Функционал (functional programming) - это стиль программирования, который подразумевает использование функций в качестве основных строительных блоков программы. В Python анонимные функции могут быть определены с использованием ключевого слова lambda. Они обычно используются вместе с функционалами filter() и map().

Примеры использования функционалов filter() и map():

```
# Фильтрация с использованием filter(): выбор только четных чисел
```

```
numbers = [1, 2, 3, 4, 5]
```

```
filtered_numbers = list(filter(lambda x: x % 2 == 0, numbers))
```

```
print(filtered_numbers) # Выводит [2, 4]
```

```
# Отображение с использованием map(): умножение каждого элемента списка на 2
```

```
numbers = [1, 2, 3, 4, 5]
```

```
mapped_numbers = list(map(lambda x: x * 2, numbers))
```

```
print(mapped_numbers) # Выводит [2, 4, 6, 8, 10]
```

1.6.17. Как определяется класс в языке Python? Приведите пример определения класса. Что такое статическая переменная класса? Чем она отличается от переменной экземпляра класса?

В Python класс определяется с использованием ключевого слова class, за которым следует имя класса. Внутри класса могут быть определены методы (функции, принадлежащие классу) и переменные.

Пример определения класса:

```
class Person:
```

```
    def __init__(self, name, age):
```

```
self.name = name  
self.age = age  
  
def say_hello(self):  
    print("Hello, my name is", self.name)  
person1 = Person("Alice", 25)  
person1.say_hello() # Выводит "Hello, my name is Alice"
```

В этом примере определен класс Person, который имеет два атрибута: name и age, и метод say_hello(), который выводит приветствие с именем объекта класса.

Статическая переменная класса (static class variable) - это переменная, которая принадлежит классу, а не конкретному экземпляру класса. Она разделяется между всеми экземплярами класса.

Пример статической переменной класса:

```
class Circle:  
    pi = 3.14159  
    def __init__(self, radius):  
        self.radius = radius  
    def calculate_area(self):  
        return Circle.pi * self.radius ** 2  
circle1 = Circle(5)  
print(circle1.calculate_area()) # Выводит 78.53975  
print(Circle.pi) # Выводит 3.14159
```

В этом примере pi - это статическая переменная класса Circle, которая используется для вычисления площади круга в методе calculate_area(). Статическая переменная доступна как через экземпляр класса (circle1.pi), так и через сам класс (Circle.pi).

Переменная экземпляра класса (instance variable) - это переменная, которая принадлежит конкретному экземпляру класса. Каждый экземпляр класса имеет свою копию переменной экземпляра.

Пример переменной экземпляра класса:

```
class Person:
```

```
def __init__(self, name):  
    self.name = name  
  
person1 = Person("Alice")  
person2 = Person("Bob")  
print(person1.name) # Выводит "Alice"  
print(person2.name) # Выводит "Bob"
```

ЛАБ 2 – Неинформированный

1. Назовите основные способы представления задач в ИИ?
 - представление задач в пространстве состояний;
 - представление, сводящее задачу к подзадачам;
 - представление задач в виде теорем.
2. Определите состояния, операторы преобразования состояний, функции стоимости для следующих задач:
 - а) задача о коммивояжере;
 - б) задача о миссионерах и каннибалах;
 - в) задача о раскраске карт.

а) Задача о коммивояжере:

Представление задачи о коммивояжере

S – мно-во состояний. **Состояние** представляется в виде пути Гамильтона (ни один город не повторяется) .

$$S = \{(x_1, x_2, \dots, x_n) \mid n=1, \dots, N+1, x_i \in X, x_i \neq x_j, \text{ кроме } i=1, j=N+1\}$$

Функция преобразования состояний – это расширение пути Гамильтона:

$$\Delta(x_1, x_2, \dots, x_n) = \{(x_1, x_2, \dots, x_n, x_{n+1}) \mid x_{n+1} \in X, x_{n+1} \neq x_i \text{ для всех } 1 \leq i \leq n\}$$

Множество целевых состояний включает все состояния (пути) длиной $N+1$. Среди этих путей надо найти кратчайший путь.

Состояние: Состояние в задаче о коммивояжере представляет собой перестановку городов, определяющую порядок, в котором коммивояжер должен посетить города.

Операторы преобразования состояний: Операторы преобразования состояний включают перестановку двух городов или обмен порядком нескольких городов.

Функция стоимости: Функция стоимости определяет общую стоимость пути коммивояжера, который проходит через все города. Цель состоит в минимизации этой стоимости, чтобы найти оптимальный маршрут.

б) Задача о миссионерах и каннибалах:

Представление задачи о каннибалах и миссионерах

Три миссионера и три каннибала находятся на левом берегу реки. Имеется лодка, вмещающая не более двух человек. Если на каком-то берегу каннибалов окажется больше, чем миссионеров, то они съедят миссионеров. Требуется найти безопасный план пересечения реки.

Состояние: $\{M, K, B\}$, где M - кол-во миссионеров на левом берегу, K - количество каннибалов на левом берегу, B - местонахождения лодки L или R , $M \geq K$.

Начальное состояние: $\{3, 3, L\}$.

Целевое состояние: $\{0, 0, R\}$.

Операторы: $\{M, K\}$, где M - кол-во миссионеров в лодке, K - кол-во каннибалов в лодке. Допустимые операторы: $\{1, 0\}; \{2, 0\}; \{1, 1\}; \{0, 1\}; \{0, 2\}$.

Состояние: Состояние в задаче о миссионерах и каннибалах обычно представляет собой количество миссионеров, каннибалов и лодок на каждом берегу реки.

Операторы преобразования состояний: Операторы преобразования состояний включают перемещение миссионеров, каннибалов и лодки с одного берега на другой. Однако, операторы должны соблюдать ограничения, чтобы не нарушать правила задачи, например, чтобы число каннибалов никогда не превышало число миссионеров на любом берегу.

Функция стоимости: Функция стоимости в этой задаче может быть просто количеством выполненных перемещений или более сложной, учитывающей другие факторы, такие как безопасность миссионеров и соблюдение правил.

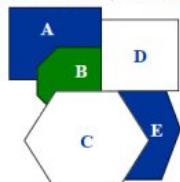
в) Задача о раскраске карт:

Представление задачи о раскраске карты

Введем множество имен вершин $\{v1, v2, \dots, vN\}$.

Множество цветов (красок) $\{c1, c2, c3, c4\}$.

Состояние представим в виде N элементной записи, состоящей из цветов вершин. Вершина имеет цвет x , если ей не был назначен цвет. Пример состояния: $\{c1, x, c3, x, x, x, \dots\}$.



Состояние задачи для рисунка:

$\{\text{blue}, \text{green}, \text{x}, \text{x}, \text{blue}\}$

Начальное состояние: $\{x, x, \dots, x\}$.

Целевая проверка : все вершины имеют цвет и для каждой пары вершин v_i и v_j , которые являются смежными, состояние должно удовлетворять $\text{color}(i) \neq \text{color}(j)$.

Множество целевых состояний: множество состояний, удовлетворяющих указанным условиям

Состояние: Состояние в задаче о раскраске карт представляет собой набор цветов, назначенных каждой области на карте.

Операторы преобразования состояний: Операторы преобразования состояний включают изменение цвета одной или нескольких областей на карте.

Функция стоимости: Функция стоимости может измерять степень неправильности раскраски, например, количество пар соседних областей с одинаковыми цветами. Цель состоит в минимизации этой функции стоимости, чтобы достичь правильной или оптимальной раскраски карты.

3. Сформулируйте для задачи поиска пути и задачи поедания всех пищевых гранул в AI Расман, что представляют собой состояния, действия, функция-приемник, функция проверки цели.

Нахождение пути:

- **состояние:** координаты (x, y) местоположения;
- **действия:** Север, Юг, Восток, Запад;
- **функция-преемник:** обновить только местоположение;
- **проверка цели:** $(x, y) = \text{END?}$

Поедание всех точек (гранул):

- **состояние:** координаты (x, y) местоположения, логические точки (true/false);
- **действия:** Север, Юг, Восток, Запад;
- **функция-преемник:** обновить местоположение и логические значения точек;
- **проверка цели:** все ли логические точки ложны?

4. Для задачи о двух кувшинах емкостью 5 литров и 2 литра, построить дерево поиска, если требуется налить во второй кувшин ровно 1 литр воды.

Представление задачи о двух кувшинах

Дан кувшин с водой емкостью 5 литров и пустой кувшин емкостью 2 литра. Требуется наполнить второй кувшин 1 литром воды. Воду можно либо выливать, либо переливать из одного кувшина в другой.

Состояние: $\{X,Y\}$, где X – объем воды в первом кувшине; Y – объем воды во втором кувшине.

Начальное состояние: $\{5,0\}$.

Целевое состояние: $\{*,1\}$, где * - означает любой объем.

Операторы:

$\{X,Y\} \rightarrow \{0,Y\}$ – вылить первый кувшин

$\{X,Y\} \rightarrow \{X,0\}$ – вылить второй кувшин

$\{X,2\} \rightarrow \{X+2,0\}$ – перелить 2-ой в 1-ый, $X \leq 3$.

$\{X,0\} \rightarrow \{X-2,2\}$ – перелить из 1-го во 2-ой, $X \geq 2$

$\{1,Y\} \rightarrow \{0, Y+1\}$ – перелить часть из 1-го во 2-ой, $Y < 2$

Решение: $\{5,0\} \{3,2\} \{3,0\} \{1,2\} \{1,0\} \{0,1\}$

5. Напишите на псевдоязыке процедуры поиска в ширину и глубину, объясните их различие с алгоритмической точки зрения.

Поиск в ширину

```
Procedure Breadth_First_Search; {BFS}
Begin
    Поместить начальную вершину в список OPEN;
    CLOSED:='пустой список';
    While OPEN<>'пустой список' Do
        Begin
            n:=first(OPEN);
            If n='целевая вершина' Then Выход(УСПЕХ);
            Переместить n из списка OPEN в CLOSED;
            Раскрыть вершину n и поместить все ее
            дочерние вершины [, отсутствующие в
            списках OPEN или CLOSED,] в конец списка
            OPEN, связав с каждой дочерней вершиной
            указатель на n;
        End;
        Выход(НЕУДАЧА);
    End.
```

В.Бондарев

Поиск в ширину (BFS) начинает с указанной стартовой вершины и постепенно расширяется на все ближайшие соседние вершины перед тем, как перейти к следующему уровню. Он исследует все вершины на текущем уровне, прежде чем перейти к вершинам следующего уровня. Поэтому BFS образует "волну" распространения от стартовой вершины.

Поиск в глубину (DFS) начинает с указанной стартовой вершины и исследует одну из соседних вершин до тех пор, пока не достигнет конечной точки или не окажется в тупике. Затем он возвращается на предыдущий уровень и исследует

следующую непосещенную соседнюю вершину. DFS продолжает идти вглубь, пока не достигнет конечной точки или не исследует все доступные пути в графе.

6. Напишите на псевдоязыке процедуру поиска в соответствии с алгоритмом равных цен.

Алгоритм равных цен

```
Procedure Uniform_Cost_Search;
Begin
    Поместить начальную вершину в список OPEN;
    CLOSED:='пустой список';
    While OPEN<>'пустой список' Do Begin
        n:=first(OPEN);
        If n='целевая вершина' Then Выход(УСПЕХ);
        Переместить вершину n из списка OPEN в
        CLOSED;
        Раскрыть вершину n, для каждой дочерней
        вершины i вычислить стоимость  $\hat{g}(n, n_i)$  ;
        Поместить дочерние вершины, которых нет в
        списках OPEN и CLOSED, в список OPEN, связав
        с каждой вершиной указатель на вершину n и
        положить  $\hat{g}(n_i) = \hat{g}(n, n_i)$  ;
        Для каждой из дочерних вершин, которые уже
        содержатся в списке OPEN, сравнить текущую
        стоимость  $\hat{g}(n, n_i)$  с ранее вычисленным
        значением стоимости  $\hat{g}(n_i)$  , хранящимся в
        списке OPEN, если  $\hat{g}(n, n_i) < \hat{g}(n_i)$  , то установить
         $\hat{g}(n_i) = \hat{g}(n, n_i)$ . Снабдить указанные дочерние
        вершины указателями на вершину n;
        Упорядочить список OPEN по возрастанию
        стоимости;
    End;
    Выход(НЕУДАЧА);
End.
```

7. Объясните назначение функций problem.getStartState(), problem.isGoalState(), problem.getSuccessors() среди AI Pacman и приведите примеры значений, возвращаемых этими функциями.

- problem.getStartState(): Эта функция возвращает начальное состояние, с которого начинается поиск. В контексте AI Pacman оно обычно представляет собой координаты пакмана и расположение привидений на игровом поле. Пример значений, возвращаемых этой функцией, может быть (2, 4), что означает, что пакман начинает движение из клетки с координатами (2, 4).
- problem.isGoalState(state): Эта функция проверяет, является ли текущее состояние state целевым состоянием, то есть состоянием, при котором задача считается решенной. В случае AI Pacman целевым состоянием может быть достижение определенной клетки, где находится пища или окончание игры.

Пример значений, возвращаемых этой функцией, может быть True, если пакман достиг клетки с пищей, и False в противном случае.

- `problem.getSuccessors(state)`: Эта функция возвращает список последователей (`successors`) для данного состояния `state`. Последователи представляют собой все возможные действия (например, движение вверх, вниз, влево, вправо), которые пакман может выполнить из текущего состояния. Каждый последователь представлен кортежем (следующее состояние, действие, стоимость). Пример значений, возвращаемых этой функцией, может быть `[((2, 3), 'влево', 1), ((3, 4), 'вниз', 1)]`, где `(2, 3)` и `(3, 4)` - следующие состояния, 'влево' и 'вниз' - действия, а `1` - стоимость или расстояние, связанное с выполнением действия.

8. Напишите на языке Python функцию, реализующую поиск в глубину применительно к среде AI Pacman.

Листинг 1 – Функция поиска в глубину

```
def depthFirstSearch(problem):  
    start_state = problem.getStartState()  
    if problem.isGoalState(start_state):  
        return []  
  
    visited = set() # Множество посещенных узлов  
  
    def dfs(state, path):  
        if problem.isGoalState(state):  
            return path  
  
        visited.add(state)  
  
        for next_state, action, _ in problem.getSuccessors(state):  
            if next_state not in visited:  
                new_path = path + [action]  
                result = dfs(next_state, new_path)  
                if result:  
                    return result  
  
    return [] # Если путь не найден  
return dfs(start_state, [])
```

9. Сформулируйте принципы поиска, используемые в алгоритмах поиска в глубину: с возвратом, с ограничением глубины и с итеративным углублением.

- Принцип поиска в глубину с возвратом (DFS с возвратом): Этот принцип поиска в глубину расширяет стандартный алгоритм DFS, позволяя возвращаться к предыдущим узлам, если текущий путь не приводит к решению. При достижении тупика или цикла алгоритм возвращается к предыдущему узлу и исследует другие возможные пути. Это позволяет исследовать все возможные пути в графе и найти оптимальное решение.

- Принцип поиска в глубину с ограничением глубины (DFS с ограничением глубины): В этом принципе поиска в глубину устанавливается ограничение на максимальную глубину поиска. Алгоритм останавливается, когда достигнута максимальная глубина, и продолжается только если решение не найдено. Это позволяет избежать бесконечного поиска в глубину и ограничить время выполнения.

- Принцип поиска в глубину с итеративным углублением (DFS с итеративным углублением): Этот принцип объединяет идеи поиска в глубину и поиска в ширину. Алгоритм выполняет итерации поиска в глубину с ограничением глубины, увеличивая эту глубину с каждой итерацией. На каждой итерации алгоритм исследует все возможные пути на текущей глубине, прежде чем переходить к следующей глубине. Это позволяет алгоритму достичь глубины поиска, аналогичной алгоритму поиска в глубину с возвратом, но с более эффективным использованием ресурсов.

10. Определите критерии полноты, оптимальности, минимальности, пространственной и временной сложности.

11. Сравните основные алгоритмы слепого поиска в пространстве состояний по критериям полноты, оптимальности, пространственной и временной сложности.

Сравнение методов слепого поиска

Критерий	BFS	DFS	DFID	Двунаправленный
Время	b^s	b^m	b^s	$b^{m/2}$
Память	b^s	bm	bs	$b^{m/2}$
Оптимальность	Да	Нет	Да	Да
Полнота	Да	Нет	Да	Да

ЛАБ 3 – Информированный

5.1. Что называют эвристикой?

Основная идея таких методов состоит в использовании дополнительной информации для ускорения процесса поиска. Эта дополнительная информация формируется на основе эмпирического опыта, догадок и интуиции исследователя, т.е. **эвристик**. Использование эвристик позволяет сократить количество просматриваемых вариантов при поиске решения задачи, что ведет к более быстрому достижению цели.

5.2. Объясните основной принцип построения процедур эвристического поиска. Запишите вид оценочной функции и объясните её составляющие.

Основной принцип построения процедур эвристического поиска заключается в использовании эвристической функции для оценки стоимости или потенциальной пользы каждого возможного шага или состояния во время поиска. Эвристическая функция предоставляет информацию о том, насколько близкое или оптимальное состояние относительно целевого состояния.

Вот основные шаги построения процедур эвристического поиска:

Определение состояний и операторов: Сначала необходимо определить, как представить состояния задачи и какие операторы или действия могут быть применены к этим состояниям для перехода к новым состояниям.

Определение целевого состояния: Определите, какое состояние считается целевым или оптимальным для вашей задачи. Это может быть, например, состояние, при котором задача считается решенной или которое обладает определенными качествами.

Определение эвристической функции: Разработайте эвристическую функцию, которая оценивает стоимость или потенциальную пользу каждого состояния относительно целевого состояния. Эта функция может использовать различные эвристические оценки, такие как эвристику расстояния или предположения о допустимости действий.

Выбор стратегии поиска: Определите стратегию поиска, которая будет использоваться с эвристической функцией. Некоторые из распространенных

стратегий поиска включают алгоритм A*, алгоритмы с ограничением времени или ресурсов, алгоритмы с итеративным углублением и т. д.

А-алгоритм похож на алгоритм равных цен, но в отличие от него учитывает при раскрытии вершин, как уже сделанные затраты, так и предстоящие затраты. В этом случае оценочная функция имеет вид:

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$$

где $\hat{g}(n)$ – оценка стоимости пути из начальной вершины в вершину n , которая вычисляется в соответствии с (2.2); $\hat{h}(n)$ – **эвристическая** оценка стоимости кратчайшего пути из вершины n в целевую вершину (предстоящие затраты). Чем меньше значение $\hat{h}(n)$, тем перспективнее путь, на котором находится вершина n . В ходе поиска раскрываются вершины с минимальным значением оценочной функции $\hat{f}(n)$.

5.3. Напишите на псевдоязыке процедуру поиска в соответствии с A*-алгоритмом.

```
def aStarSearch (problem):
    Определить стартовую вершину: start = problem.getStartState()
    Поместить стартовую вершину в список OPEN: OPEN.push(start)
    CLOSED = []
    Путь = []
    while not OPEN.isEmpty():
        node = OPEN.pop()
        If node == 'целевая вершина': return Путь
        CLOSED = CLOSED.append(node)
        Раскрыть node и для всех дочерних вершин  $n_i$  вычислить оценку
         $\hat{f}(n, n_i) = \hat{g}(n, n_i) + \hat{h}(n_i)$ 
        Поместить все дочерние вершины, отсутствующие в списке CLOSED
        или OPEN, в список OPEN, связав с каждой дочерней вершиной указатель на node
        Для дочерних вершин  $n_i$ , которые уже содержатся в OPEN, сравнить
        оценки  $\hat{f}(n, n_i)$  и  $\hat{f}(n_i)$ , если  $\hat{f}(n, n_i) < \hat{f}(n_i)$ , то связать с вершиной  $n_i$ 
        новую оценку  $\hat{f}(n, n_i)$  и указатель на вершину node
        Если вершина  $n_i$  содержится в списке CLOSED и  $\hat{f}(n, n_i) < \hat{f}(n_i)$ , то
        связать с вершиной  $n_i$  новую оценку  $\hat{f}(n, n_i)$ , переместить её в список OPEN
        и установить указатель на node;
        Упорядочить список OPEN по возрастанию  $\hat{f}(n_i)$ ;
    return 'НЕУДАЧА'
```

5.4. Сформулируйте алгоритм подъема в гору.

Алгоритм осуществляет целенаправленный поиск в направлении наибольшего убывания эвристической оценочной функции. Данная функция обеспечивает оценку (прогноз) стоимости кратчайшего пути от текущей вершины n до ближайшей целевой вершины, т.е. является мерой стоимости оставшегося

пути. Чем меньше значение этой функции, тем перспективнее путь, накотором находится вершина n .

Подобный алгоритм используется при поиске экстремумов функции. Поиск экстремума осуществляют в направлении наибольшего возрастания (убывания) градиента. Поиск максимума функции в этом случае напоминает восхождение к вершине по наиболее крутому маршруту.

Алгоритм "подъема на гору"

```
Procedure Hill_Climbing;
Begin
    n:='начальная вершина';
    While n<>'целевая вершина' Do
        Begin
            Раскрыть вершину  $n$  и для всех дочерних
            вершин  $n_i$  вычислить оценки  $h(n_i)$  ;
            Выбрать дочернюю вершину  $n_i$  с минимальным
            значением  $h(n_i)$  ;
            If  $h(n_i) \geq h(n)$  Then Выход(НЕУДАЧА);
            n:= $n_i$ ;
        End;
        Выход(УСПЕХ);
    End.
```

5.5. Сформулируйте алгоритм глобального выбора первой наилучшей вершины.

```
Procedure Best_First_Search;
Begin
    Поместить начальную вершину в OPEN;
    CLOSED:='пустой список';
    While OPEN<>'пустой список' Do Begin
        n:=first(OPEN);
        If n='целевая вершина' Then Выход(УСПЕХ);
        Переместить вершину  $n$  из списка OPEN в список
        CLOSED;
        Раскрыть вершину  $n$ , для каждой дочерней вершины
        вычислить  $h(n_i)$  ;
        Поместить дочерние вершины, которых нет в
        списках OPEN и CLOSED, в список OPEN, связав с
        каждой дочерней вершиной указатель на вершину  $n$ ;
        Упорядочить список OPEN по возрастанию значений  $h(n_i)$ 
    End;
    Выход(НЕУДАЧА);
End.
```

В.Бондарев

Алгоритм глобального выбора первой наилучшей вершины (Global Best-First Search) - это алгоритм поиска в графе, который выбирает следующую вершину для исследования на основе эвристической функции, предоставляющей оценку "качества" вершины. Алгоритм стремится найти наилучшую вершину, которая имеет наибольшую эвристическую оценку среди всех доступных вершин.

Вот формулировка алгоритма глобального выбора первой наилучшей вершины:

1. Создайте пустой список открытых вершин.
2. Поместите начальную вершину в список открытых вершин.
3. Пока список открытых вершин не пуст:

Выберите вершину V с наилучшей эвристической оценкой из списка открытых вершин.

Если V является целевой вершиной, то задача решена. Верните найденный путь или выполните требуемые действия. Иначе, перейдите к следующему шагу.

4. Расширьте вершину V путем применения операторов или действий, доступных из V , для создания новых вершин.

5. Для каждой новой вершины N :

Вычислите эвристическую оценку для N .

Если N уже присутствует в списке открытых вершин с меньшей эвристической оценкой, пропустите N . Иначе, добавьте N в список открытых вершин и установите V как его родителя.

6. Повторите шаги 3-5.

5.6. Почему в А-алгоритме возможен возврат вершин из списка закрытых вершин в список открытых вершин? Приведите примеры.

В алгоритме A^* возможен возврат вершин из списка закрытых вершин в список открытых вершин в случае, когда обнаруживается более оптимальный путь к этой вершине. Это происходит, когда новый путь к вершине предлагает меньшую стоимость или более оптимальное решение по сравнению с уже известным путем.

Пример 1:

Предположим, у нас есть граф с начальной вершиной A и целевой вершиной G . Путь от A до G может проходить через вершины B, C, D и E . При первом проходе алгоритм A^* может выбрать путь $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow G$, который имеет определенную стоимость. Однако, если в дальнейшем в процессе поиска будет обнаружен другой путь $A \rightarrow B \rightarrow X \rightarrow Y \rightarrow E \rightarrow G$, который имеет меньшую стоимость, вершина E может быть возвращена из списка закрытых вершин в список открытых вершин, чтобы рассмотреть новый путь.

5.7. Сформулируйте и объясните свойства А-алгоритма.

Свойства А-алгоритма существенно зависят от условий, которым удовлетворяет или не удовлетворяет эвристическая часть оценочной функции $\hat{f}(n)$ [1]:

- 1) А-алгоритм соответствует алгоритму равных цен, если $h(n)=0$;
- 2) А-алгоритм **гарантирует оптимальное решение**, если $\hat{h}(n) \leq h(n)$; в этом случае он называется **A* – алгоритмом**. A*-алгоритм недооценивает затраты на пути из промежуточной вершины в целевую вершину или оценивает их правильно, но никогда не переоценивает;

- 3) А-алгоритм обеспечивает однократное раскрытие вершин, если выполняется **условие монотонности** $\hat{h}(n_i) \leq \hat{h}(n_j) + c(n_i, n_j)$, где n_i – родительская вершина; n_j – дочерняя вершина; $c(n_i, n_j)$ – стоимость пути между вершинами n_i и n_j ;
- 4) алгоритм A_1^* эвристически более сильный, чем алгоритм A_2^* при условии $\hat{h}_1(n) > \hat{h}_2(n)$. Эвристически более сильный алгоритм A_1^* в большей степени сокращает пространство поиска;
- 5) A*-алгоритм полностью информирован, если $\hat{h}(n) = h(n)$. В этом случае никакого поиска не происходит и приближение к цели идет по оптимальному пути;
- 6) при $\hat{h}(n) > h(n)$ А-алгоритм не гарантирует получение оптимального решения, однако часто решение получается быстро.

5.8. Какой А-алгоритм называют гарантирующим (допустимым)?

3. А-алгоритм обеспечит поиск оптимального пути, если выполняется условие **допустимости (гарантированности)**
$$\hat{h}(n) \leq h(n).$$

Алгоритм, учитывающий это условие, называют **A*- алгоритмом или гарантирующим алгоритмом**. A*-алгоритм недооценивает затраты на пути из промежуточной вершины в целевую вершину или оценивает их правильно, но никогда не переоценивает.

5.9. Сформулируйте и объясните условие монотонности.

Чтобы A*-алгоритм не раскрывал несколько раз одну и ту же вершину необходимо, чтобы выполнялось **условие монотонности**

$$\hat{h}(n_i) \leq \hat{h}(n_j) + c(n_i, n_j),$$

где n_i – родительская вершина; n_j – дочерняя вершина; $c(n_i, n_j)$ – стоимость пути между вершинами n_i и n_j . Монотонность предполагает, что не только не переоценивается стоимость $\hat{h}(n)$ оставшегося пути из n до цели, но и не переоцениваются стоимости ребер между двумя соседними вершинами.

Условие монотонности поглощает условие гарантированности (допустимости). Если условие монотонности соблюдается для всех дочерних вершин, то можно доказать, что в тот момент, когда раскрывается некоторая вершина n , **оптимальный путь** к ней уже найден. Следовательно, оценочная функция для данной вершины в дальнейшем не меняет своих значений, и никакие вершины из списка CLOSED в список OPEN не возвращаются.

5.10. Сформулируйте эвристику манхэттенского расстояния.

Манхэттенское расстояние, также известное как расстояние городских кварталов, представляет собой метрику, используемую для измерения расстояния между двумя точками в прямоугольной системе координат. Эта метрика вычисляется как сумма абсолютных разностей между соответствующими координатами точек. Например, для двух точек с координатами (x_1, y_1) и (x_2, y_2) манхэттенское расстояние равно $|x_2 - x_1| + |y_2 - y_1|$. Это расстояние часто используется в компьютерной графике, такси-сервисах для определения расстояния между двумя точками на городских улицах и в задачах оптимизации.

Манхэттенское расстояние имеет несколько преимуществ и недостатков. Среди преимуществ можно отметить его простоту вычисления и интуитивную интерпретацию, особенно когда речь идет о перемещении вдоль прямоугольной сетки улиц. Однако недостатком является то, что это расстояние может недооценивать фактическое расстояние между двумя точками, особенно если они находятся на значительном расстоянии друг от друга в вертикальном и горизонтальном направлениях

5.11. Сравните алгоритмы слепого и эвристического поиска по критерию гарантированности получения результата и эффективности поиска.

Слепой поиск и эвристический поиск отличаются по гарантированности получения результата и эффективности поиска. Слепой поиск не использует эвристическую информацию и перебирает все возможные варианты, что гарантирует нахождение решения, но может быть неэффективным при большом

пространстве поиска. Эвристический поиск использует эвристическую информацию для оценки перспективности путей и может быть более эффективным, но не гарантирует нахождение оптимального решения.

ЛАБ 4 – Мультиагенты

1. Объясните понятия: детерминированные игры с нулевой суммой, состязательный поиск.

Первый класс игр, который мы рассмотрим, - это **детерминированные игры с нулевой суммой** (шашки, шахматы, го и др.). Такие игры характеризуются полной информацией о текущей игровой ситуации (имеются игры с неполной информацией), где два игрока-противника по очереди делают ходы. Успех одного игрока – такая же по величине потеря для другого игрока. Самый простой способ представить себе такие игры - это их определение с помощью единственной переменной, значение которой агент пытается максимизировать, а его противник пытается минимизировать. Например, в игре *Racman* такая переменная соответствует набранным баллам, которые *Racman* пытается максимизировать, поедая гранулы, в то время как призраки пытаются свести к минимуму эти баллы, съедая агента. Фактически игроки в этом случае соревнуются (состязаются), поэтому поиск в игровых программах называют **состязательным поиском**.

2. Объясните следующие понятия минимаксного поиска: дерево поиска, статические оценки, динамические оценки, функция полезности.

Дерево поиска: Дерево поиска представляет возможные ходы и состояния игры в виде древовидной структуры. Корневой узел представляет текущее состояние игры, а дочерние узлы - возможные ходы, доступные из этого состояния. Каждый уровень дерева соответствует ходу одного из игроков. В листовых узлах находятся конечные состояния игры или оценки, которые отражают выигрыш или проигрыш для игрока.

Статические оценки: Статические оценки используются для оценки состояний игры, которые не являются конечными. Они представляют собой числовые значения, отражающие полезность состояния для игрока. Чем выше значение, тем более выгодное состояние для игрока. Статические оценки могут быть основаны на различных эвристиках или правилах, которые оценивают позицию игры, например, на основе материального преимущества или позиционной оценки в шахматах.

Динамические оценки: Динамические оценки используются для оценки состояний игры, которые могут привести к разным последующим состояниям в результате различных ходов. Они представляют собой прогнозируемые значения, которые оценивают потенциальные выигрыши или проигрыши, которые могут возникнуть в результате ходов в дереве поиска. Динамические оценки могут быть получены с помощью алгоритмов просмотра в глубину или методов машинного обучения, которые аппроксимируют вероятности исходов игры.

Функция полезности: Функция полезности (или функция оценки) - это функция, которая используется для оценки конечных состояний игры. Она присваивает числовое значение каждому возможному исходу игры и отражает степень выигрыша или проигрыша для игрока. Функция полезности может быть определена вручную на основе знаний об игре или вычислена с помощью статических или динамических оценок. В минимаксном поиске, игрок, максимизирующий свою выгоду, стремится выбрать ход, который максимизирует значение функции полезности, в то время как игрок, минимизирующий выгоду, стремится выбрать ход, который минимизирует значение функции полезности.

3. Объясните на примере принцип минимаксного поиска и запишите формальные выражения, используемые для распространения оценок по дереву поиска.

Принцип минимаксного поиска:

Принцип минимаксного поиска является основой алгоритма состязательного поиска. Он предполагает, что игроки действуют оптимально и стремятся максимизировать свою выгоду или минимизировать выгоду противника.

Для объяснения принципа минимаксного поиска рассмотрим пример игры "Крестики-нолики". Допустим, у нас есть дерево игры, где каждый узел представляет состояние игры, а дочерние узлы соответствуют возможным ходам игроков. При решении этой игры мы хотим найти оптимальную стратегию для игрока, который играет крестиками.

Для каждого узла в дереве мы можем оценить его выгоду для игрока, который играет крестиками. Если мы находимся на ходе крестиков, то мы хотим выбрать

ход, который максимизирует нашу выгоду. Оценка выгоды для крестиков будет положительной, если крестики выиграют, отрицательной, если нолики выиграют, и нулевой, если игра закончится ничьей.

Аналогично, для каждого узла, где на ходе нолики, мы хотим выбрать ход, который минимизирует выгоду для крестиков. То есть, мы выбираем ход, который максимизирует выгоду для ноликов, которая будет отрицательной, если нолики выиграют, положительной, если крестики выиграют, и нулевой в случае ничьей.

Формальные выражения для распространения оценок по дереву поиска можно записать следующим образом:

Если узел является листом дерева (терминальным состоянием игры), то его оценка выгоды может быть вычислена напрямую.

Если узел является максимизирующим узлом (ход крестиков), то его оценка выгоды равна максимальной оценке выгоды среди его дочерних узлов.

Выражение: $\text{оценка_выгоды} = \max(\text{оценка_выгоды_дочерних_узлов})$

Если узел является минимизирующим узлом (ход ноликов), то его оценка выгоды равна минимальной оценке выгоды среди его дочерних узлов.

Выражение: $\text{оценка_выгоды} = \min(\text{оценка_выгоды_дочерних_узлов})$

Используя эти формальные выражения, мы можем рекурсивно распространять оценки выгоды по дереву игры, начиная с листьев и двигаясь вверх к корню дерева. В результате получится оптимальная стратегия для игрока, который играет крестиками, учитывая оптимальные ходы противника (ноликов).

4. Разработайте пример дерева игры в крестики-нолики и предложите способ вычисления статических оценок.

Давайте разработаем пример дерева игры в крестики-нолики для наглядности. Предположим, что на игровом поле 3x3 есть следующая расстановка крестиков (X) и ноликов (O):

X | O | X

O | X | O

X | |

Для удобства, пронумеруем клетки поля следующим образом:

1 | 2 | 3

4 | 5 | 6

7 | 8 | 9

Теперь давайте построим дерево игры, рассматривая возможные ходы для крестиков и ноликов:

1: O

/ | \

/ | \

X X X

/ \ | |

/ \ | |

2: X 3: O |

/ | | |

/ | | |

O X X X

| | | |

| | | |

X X X X

| | | |

| | | |

X X X X

Это пример простого дерева игры, где каждый узел представляет состояние игры после определенного хода. Для каждого узла можно вычислить статическую оценку, которая представляет собой оценку выгоды для крестиков или ноликов в данном состоянии.

Способ вычисления статических оценок может варьироваться, но обычно используются простые эвристические функции. Например, можно присвоить положительную оценку, если крестики выиграли, отрицательную, если нолики выиграли, и нулевую, если игра закончилась ничьей или еще не завершена.

В нашем примере, если крестики выигрывают, можно присвоить оценку +1, если нолики выигрывают, оценку -1, а в случае ничьей или незавершенной игры - оценку 0.

Таким образом, в данном примере статические оценки для последних узлов в дереве будут следующими:

2: X - оценка = 0

3: O - оценка = 0

4: O - оценка = 0

5: X - оценка = 0

6: X - оценка = 0

7: X - оценка = 0

8: X - оценка = 0

9: X - оценка = 0

Вычисление более сложных статических оценок может включать в себя проверку наличия выигрышных комбинаций на игровом поле, анализ позиций фигур, оценку контроля над полем и так далее.

5. Запишите псевдокод, определяющий основные функции минимаксного рекурсивного поиска, и объясните его на примере.

6. Объясните особенности игр с несколькими игроками, приведите пример соответствующего дерева игры и объясните механизм распространения оценок состояний.

Игры с несколькими игроками отличаются от игр с двумя игроками тем, что в них участвуют более двух игроков, каждый из которых принимает свои решения и стремится максимизировать свою выгоду. В таких играх возникает необходимость учитывать взаимодействие между всеми игроками и определять оптимальные стратегии для каждого из них.

Для объяснения особенностей игр с несколькими игроками рассмотрим пример игры "Морской бой" для трех игроков. В этой игре игровое поле представляет собой квадрат размером 3x3, и каждый игрок по очереди делает ходы, ставя свой корабль на свободную клетку поля. Цель игроков - потопить корабли других игроков и сохранить свои корабли нетронутыми.

Рассмотрим следующее состояние игры:

1: A | B |

2: A | |

3: | | B

В этом примере игрок А разместил свой корабль на клетке (1,1), игрок В на клетке (1,3), и игрок С на клетке (3,2).

Дерево игры для этого примера будет содержать различные возможные ходы каждого игрока в зависимости от текущего состояния игры. Допустим, игрок А должен сделать следующий ход. Возможные варианты для него будут:

1: A

/ | \

/ | \

2: B 3: B 4: B

/ / | \ \

/ / | \ \

5: B 6: B 7: B 8: B

В этом примере каждый узел представляет возможный ход игрока А, а дочерние узлы соответствуют возможным ходам других игроков (B и C).

Оценка состояний в играх с несколькими игроками может быть сложной задачей. В зависимости от конкретной игры и ее правил, могут использоваться различные методы вычисления оценок. Один из распространенных методов - это применение алгоритма Минимакс с альфа-бета отсечением, который позволяет оценивать состояния игры с учетом действий всех игроков.

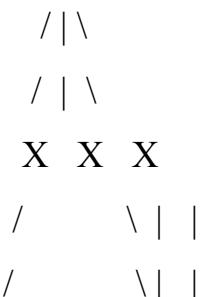
Механизм распространения оценок состояний в дереве игры с несколькими игроками основан на минимаксном принципе. Оценки состояний распространяются от листьев дерева вверх к корню, учитывая выигрышные и проигрышные сценарии для каждого игрока. В каждом узле дерева, игрок выбирает ход, который максимизирует его выгоду, при условии оптимальной игры остальных игроков. Затем, эти оценки передаются родительским узлам, учитывая роль каждого игрока.

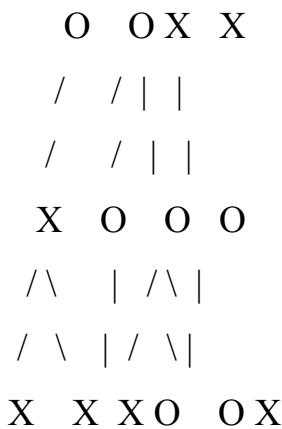
В итоге, после распространения оценок по дереву игры, можно определить оптимальные стратегии для каждого игрока, учитывая решения остальных игроков и стремясь максимизировать свою выгоду. Игры с несколькими игроками отличаются от игр с двумя игроками тем, что в них участвует более двух игроков, каждый из которых принимает свои решения и стремится максимизировать свою выгоду. В таких играх возникает необходимость учитывать взаимодействие между всеми игроками и определять оптимальные стратегии для каждого из них.

Давайте рассмотрим игру с тремя игроками в крестики-нолики. Игровое поле имеет размер 3x3, и игроки ходят по очереди, ставя свои фишечки (крестики или нолики) на свободные клетки поля. Цель каждого игрока - выстроить линию из трех своих фишечек по горизонтали, вертикали или диагонали.

Пример дерева игры с тремя игроками может выглядеть следующим образом:

O





В этом примере игроки обозначены буквами Х и О. Каждый узел дерева представляет состояние игры после определенного хода. Например, корень дерева представляет начальное состояние игры, где первым ходит игрок О. Дочерние узлы представляют возможные ходы игрока О. Затем следуют ходы игроков Х и О, и так далее.

Распространение оценок состояний в дереве игры с несколькими игроками может быть реализовано с использованием алгоритма Минимакс. Целью алгоритма является нахождение оптимальной стратегии для каждого игрока, учитывая решения остальных игроков.

Алгоритм Минимакс работает рекурсивно, начиная с листьев дерева игры. В листьях вычисляются статические оценки состояний игры. Затем оценки распространяются вверх по дереву, учитывая роль каждого игрока. Если игрок находится в своем ходе, он стремится максимизировать оценку, а если он находится в ходе противника, он стремится минимизировать оценку.

Таким образом, оценки состояний игры распространяются от листьев дерева к корню, учитывая стратегии и решения всех игроков. В результате алгоритма можно определить оптимальные стратегии для каждого игрока, учитывая решения остальных игроков и стремясь максимизировать свою выгоду.

7. Объясните на примере принцип альфа-бета поиска, сформулируйте правила альфа- и бета-отсечений.

Принцип альфа-бета поиска является эффективным алгоритмом для ускорения поиска в дереве игры, особенно в контексте алгоритма Минимакс. Он позволяет избежать проверки бесперспективных вариантов и сократить количество просмотренных узлов.

Для объяснения принципа альфа-бета поиска рассмотрим простой пример игры "Крестики-нолики" на дереве игры. Игра состоит из двух игроков, X и O, и игрового поля размером 3x3. Цель каждого игрока - выстроить линию из трех своих фишек по горизонтали, вертикали или диагонали.

Предположим, что мы рассматриваем дерево игры на глубине 3 ходов вперед. В начальном состоянии игры первым ходит игрок X. Примерно так может выглядеть часть дерева игры:

```
X  
/ \\  
/ | \\  
O X X  
/\ | |  
/ \ | |  
X O X O  
/ / | |  
/ / | |  
O O X X
```

gherkin

При использовании принципа альфа-бета поиска важно понимать два правила: альфа-отсечение и бета-отсечение.

Альфа-отсечение:

Альфа-значение (α) представляет наилучшую оценку для максимизирующего игрока (например, игрока X) на текущем пути.

Если на пути максимизирующего игрока найден ход, который приведет к оценке, меньшей или равной α , то остальные пути не требуют дальнейшего рассмотрения, их можно отсечь.

То есть, если игрок X уже нашел ход, который гарантирует ему оценку α или выше, ему не нужно рассматривать другие ходы, так как игрок O будет выбирать ходы, которые будут давать ему оценки меньше α .

Бета-отсечение:

Бета-значение (β) представляет наилучшую оценку для минимизирующего игрока (например, игрока O) на текущем пути.

Если на пути минимизирующего игрока найден ход, который приведет к оценке, большей или равной β , то остальные пути не требуют дальнейшего рассмотрения, их можно отсечь.

То есть, если игрок O уже нашел ход, который гарантирует ему оценку β или ниже, ему не нужно рассматривать другие ходы, так как игрок X будет выбирать ходы, которые будут давать ему оценки больше β .

Принцип альфа-бета поиска позволяет эффективно сократить количество просмотренных узлов в дереве игры, исключая неперспективные варианты. Алгоритм рекурсивно просматривает дерево игры, передавая альфа- и бета-значения вниз по дереву. Если в какой-то момент просмотра поддерева становится ясно, что альфа-бета условия отсечения выполняются, то остальные узлы в этом поддереве не рассматриваются, и поиск переходит к следующему поддереву.

9. Что такое функция оценки? В какой математической форме её обычно определяют?

В ходе минимаксного поиска процесс генерации ходов останавливают в узлах (состояниях), расположенных на некоторой выбранной глубине. Полезность

этих состояний определяют с помощью тщательно выбранной **функции оценки** (evaluation function), которая дает приближенное значение полезности этих состояний. Чаще всего функция оценки $eval(s)$ представляет собой линейную комбинацию функций $f_i(s)$:

$$eval(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s),$$

где каждая функция $f_i(s)$ вычисляет некоторую характеристику состояния s , и каждой характеристике назначается соответствующий вес w_i . Например, в игре в шашки мы могли бы построить оценочную функцию с 4-мя характеристиками: количество пешек у агента, количество королев у агента, количество пешек у противника и количество королев у противника. Структура оценочной функции может быть совершенно произвольной, и она не обязательно должна быть линейной.

10. Почему минимаксный принцип построения агентов, функционирующих в средах с элементами случайности, недостаточно эффективен?

Минимаксный принцип построения агентов в средах с элементами случайности может быть недостаточно эффективен по нескольким причинам:

Неопределенность: Среды с элементами случайности могут быть хаотичными и неопределенными. В таких средах предсказание будущих состояний игры или исходов может быть сложным или невозможным. Минимаксный подход не учитывает вероятностные распределения и не может эффективно адаптироваться к изменяющимся условиям.

Вычислительная сложность: Дерево поиска, используемое в минимаксном алгоритме, может быстро разрастаться в случаях с большим пространством состояний и возможных ходов. Это приводит к экспоненциальному росту вычислительной сложности алгоритма. В средах с элементами случайности, где каждый ход может иметь множество возможных исходов, минимаксный алгоритм может столкнуться с проблемой комбинаторного взрыва и стать непрактичным для применения.

Оптимальность в статическом смысле: Минимаксный алгоритм стремится выбрать оптимальный ход, исходя из текущего состояния игры. Однако, в средах с элементами случайности, оптимальный ход может зависеть не только от текущего состояния, но и от предыдущих ходов, текущих наблюдений и случайных событий. Минимаксный алгоритм не учитывает эту зависимость и может не достичь оптимальности в долгосрочной перспективе.

11. Сформулируйте метод поиска Expectimax, что такое узлы жеребьевки, как в этих узлах вычисляется ожидаемая оценка?

В отличие от минимаксного алгоритма, который предполагает, что противник всегда выбирает оптимальный ход, Expectimax учитывает случайные события и вычисляет ожидаемую оценку для каждого возможного хода.

Основные компоненты метода поиска Expectimax:

Узлы жеребьевки (Chance nodes): Узлы жеребьевки представляют случайные события в игре, которые могут изменить состояние игры и привести к различным исходам. В узлах жеребьевки происходит случайная выборка исходов с учетом вероятностей. Каждому исходу приписывается вероятность, и вычисляется взвешенная сумма оценок исходов.

Ожидаемая оценка (Expected value): В узлах жеребьевки вычисляется ожидаемая оценка, которая представляет собой средневзвешенную оценку исходов, учитывая их вероятности. Ожидаемую оценку можно рассчитать, умножив каждую оценку исхода на его вероятность и суммируя полученные значения.

12. Запишите псевдокод, определяющий основные функции Expectimax поиска и объясните их работу на примере.

```
def value(state):
    if state является терминальным: return utility(state)
    if агент MAX: return max_value(state)
    if агент EXP: return exp_value(state)

def max_value(state):          # максимальная оценка
    v = -∞
    for succ in successor(state):
        v=max(v, value(succ))

    return v

def exp_value(state):          # ожидаемая оценка
    v = 0
    for succ in successor(state):
        p=probability(succ)
        v+=p* value(succ)
    return v
```