



МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт искусственного интеллекта

Кафедра высшей математики

КУРСОВАЯ РАБОТА
по дисциплине
«Объектно-ориентированное программирование»

Тема курсовой работы
«Применение обучения с подкреплением в игре на плоском дискретном поле»

Студент группы КМБО-01-22

Трудолюбов Н.А.

Руководитель курсовой работы
доцент кафедры Высшей математики
к.ф.-м.н

Петрусевич Д.А.


Работа представлена к
защите

«28» *июл* 20 *23* г.


(подпись студента)

«Допущен к защите»

«28» *июл* 20 *23* г.


(подпись руководителя)



МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт искусственного интеллекта

Кафедра высшей математики

Утверждаю

Исполняющий обязанности заведующего
кафедрой *М.В. Шатина* А.В. Шатина

«22» сентября 2023 г.

ЗАДАНИЕ
на выполнение курсовой работы
по дисциплине «Объектно-ориентированное программирование»

Студент *Трудолюбов Н.А.*

Группа *КМБО-01-22*

1. Тема: «Применение обучения с подкреплением в игре на плоском дискретном поле»

2. Исходные данные:



Построить класс для модели реализации обучения с подкреплением (метод временных разностей и UCS, как минимум)

На игровом поле есть пустые ячейки, ямы, ячейки с препятствием заданной высоты. Агент может переходить в соседнюю ячейку или строить лестницу через стену (число действий равно высоте стены), если это не препятствие. За попадание в яму даётся максимальный штраф, малое поощрение за переход в ячейку без ямы (с приближением к конечной точке), большое поощрение за переход через препятствие, максимальное – за попадание в конечный пункт.

Реализовать переход между обучением и стационарным состоянием агента в виде эпсилон-жадной стратегии

3. Перечень вопросов, подлежащих разработке, и обязательного графического материала:
Продemonстрировать изменение распределения вероятностей выбора действия (строить переход через препятствие или выбрать ход в нужном направлении)
Продemonстрировать изменение выигрыша агента со временем

4. Срок представления к защите курсовой работы: до «22» декабря 2023 г.

Задание на курсовую работу выдал	«22» сентября 2023 г.		(Петрусеvич Д.А.)
Задание на курсовую работу получил	«22» сентября 2023 г.		(Трудолюбoв Н.А.)

Оглавление

Введение.....	3
Глава 1	4
UCB-обучение	4
TD-обучение	5
Вывод к главе 1	5
Глава 2	7
Вывод к главе 2	16
Заключение	18
Список литературы	19
Приложение	20

Введение

В главе 1 будет реализована теоретическая часть курсовой работы. Рассмотрим алгоритмы обучения с подкреплением на основе верхнего доверительного интервала (UCB) и метод временных разниц (TD), а также ε -жадную стратегию, которая будет применяться в данных алгоритмах для увеличения активности обучения агентом поля. Проведем анализ и сравнение этих методов, на основе чего выявим наилучший способ обучения для динамического окружения.

В главе 2 будет представлена практическая реализация алгоритмов обучения с подкреплением. На языке программирования C++ будут созданы классы для моделирования игрового поля, агента и самой игры. Будут разработаны функции и методы, необходимые для работы алгоритмов. Затем сравним эффективности для получившихся функций-методов при различных параметрах. В конце измерим скорости работы для каждого из методов и выверим количественно эффективность алгоритмов.

Глава 1. Теоретическая часть

Обучение с подкреплением (reinforcement learning) – это метод машинного обучения (ML), который обучает программное обеспечение принимать решения для достижения наиболее оптимальных результатов. Такое обучение основано на имитации процесса обучения методом проб и ошибок, который люди используют для достижения своих целей.

К настоящему моменту существует большое количество методов подобного типа. Поэтому, для целей курсовой работы ограничимся на двух из них, а именно на UCB и TD обучениях.

Метод UCB (upper confidence bound) или, в переводе на русский, метод верхнего доверительного интервала связан с так называемым принципом оптимизма в условиях неопределенности – статистическим принципом, основанным на законе больших чисел. UCB строит оптимистическую гипотезу, основанную на выборочном среднем вознаграждении и на оценке верхней доверительной границы вознаграждения. Оптимистическая гипотеза определяет ожидаемую выплату при каждом действии с учетом неопределенности действий. Таким образом, UCB всегда выбирает действие с более высоким потенциальным вознаграждением, пытаясь найти баланс между риском и наградой. Если оказывается, что у других действий оптимистическая оценка меньше, чем у текущего, то алгоритм переключается на другое действие. Точнее, UCB хранит ожидаемое вознаграждение каждого действия $Q(S_t, a)$ и верхнюю доверительную границу $c \sqrt{\frac{\ln(t)}{N(S_t)}}$ во всех состояниях. Затем алгоритм выбирает то действие, для которого принимает максимум их сумма. Таким образом, общая формула выглядит следующим образом:

$$A_t = \underset{a}{argmax} [Q(S_t, a) + c \sqrt{\frac{\ln(t)}{N(S_t)}}]. \quad (1)$$

В этой формуле:

t - дискретный временной шаг,

a – одно из возможных действий,

$Q(S_t, a)$ - оценка истинного значения (ожидаемого вознаграждения) действия a в состоянии S_t ,

c – константа, которая регулирует верхнюю доверительную границу (выбирается в зависимости от типа решаемой задачи),

$N(S_t)$ - сколько раз агент был в состоянии S_t ,

$\underset{a}{argmax} f(a)$ - действие a , при котором $f(a)$ является максимальной.

$Q(S_t, a)$ вычисляется следующим образом:

$$Q(S_t, a) = \frac{\sum_{t=1}^{T-1} R(S_t)}{N(S_t)}. \quad (2)$$

Здесь $R(S_t)$ - вознаграждение в текущем состоянии .

TD-обучение (temporal difference learning), что на русском означает обучение на основе временных разностей, можно рассматривать как сочетание методов Монте-Карло и динамического программирования, поскольку в них используется идея выборки, заимствованная у первых, и идея бутстрэппинга, заимствованная у вторых. TD-обучение широко применяется во всех алгоритмах обучения с подкреплением и составляет ядро многих из них. Одна из вариаций TD-метода выполняет обновление по формуле:

$$Q(S_t, a) \leftarrow Q(S_t, a) + \alpha [R_{t+1} + \gamma * \max_{a'} Q(S_{t+1}, a') - Q(S_t, a)] \quad (3)$$

сразу после перехода в состояние S_{t+1} и получения вознаграждения R_{t+1} . TD-обучение при таком способе обновления называется Q-обучением.

При этом, в формуле (3):

S_t - состояние в момент t ,

$Q(S_t, a)$ – ожидаемая ценность в состоянии S_t при выборе действия a ,

$\max_{a'} Q(S_{t+1}, a')$ – ожидаемая ценность при выборе максимизирующего действия в состоянии S_{t+1} ,

α – постоянный параметр размера шага (влияет на скорость обучения, может находиться в диапазоне $(0, 1]$),

γ - коэффициент обесценивания.

Для обоих методов действия будем выбирать через ε -жадную стратегию, которая работает следующим образом:

Введем параметр $\varepsilon \in (0, 1)$.

На каждом шаге t :

- Получим значение v - случайной величины, равномерно распределенной на отрезке $(0,1)$;
- Если $v \in (0, \varepsilon]$, то выберем действие $A_t \in A$ случайно и равновероятно, иначе выберем действие в соответствии с политикой принятия решения для каждого обучения;
- Обновляем оценку математического ожидания для действия A_t .

Оценка сложности для обновления UCB методом равна $O(1)$. Для TD получается тоже $O(1)$. В то же время, сравнивая оба подхода, можно сказать, что обучение на основе временных разностей должно работать лучше, несмотря на оценку сложности функций обновления. Это связано с тем, что TD-обучение использует информацию об обновлении оценок ценности из полученных образцов опыта, что позволяет агенту адаптироваться к изменяющейся динамической среде. В то время как UCB, хотя и использует верхние доверительные интервалы, но не обновляет оценки ценности на основе фактического

опыта, что может привести к менее гибкому поведению в динамически изменяющейся среде.

Вывод:

Таким образом мы рассмотрели работы алгоритмов обучения UCB и TD при ϵ -жадной стратегии и в конце вывели, что обучение на основе временных разностей справляется лучше обучения на основе верхнего доверительного интервала для динамического окружения

Глава 2. Практическая часть

Напишем код на языке программирования C++.

Для начала реализуем общий класс Field, который будет представлять из себя поле, по которому будет двигаться агент. В нем будут храниться двумерный вектор и значения его “высоты” и “ширины”, а также матрица вероятностного распределения для каждого цвета. Также в нем будут содержаться функции и перегруженные операторы для возможности управления внутренними элементами Field. По заданию, каждая клетка в поле может иметь один из заданных цветов, для простоты вывода данных в консоль заменим их на числа. Кроме того, вместо выхода за границу в противоположной части поля, будем останавливать обучение агента уже при достижении конечной ячейки и выдавать ему максимально возможную награду.

Введем класс Agent, хранящий в себе координаты клеток, где агент появился изначально, где он находится на текущий момент и где он был на предыдущем шаге. Последний пункт будет использоваться в TD-методе обучения. У Agent будут свои функции и перегруженные методы для манипуляции внутренними данными.

Создадим главный класс Game, в котором будут храниться агент и изучаемое поле, а также функции методов обучения и перегруженные операторы для вывода игры. Пользователю будут доступны следующие параметры:

- epsilonTD и epsilonUCB – параметры, отвечающие за границы в эпсилон-жадной стратегии, для TD и UCB обучений соответственно;
- alpha и gamma – константы из TD-обучения;
- c – константа из UCB-обучения;
- iteration_count – количество раз, сколько агент будет обучаться через выбранный метод;
- print_agent – вывод всех ходов агента;
- last_iteration – вывод ходов агента только на последней итерации обучения.

И следующие основные функции:

- MethodTD – обучение агента на основе временных разностей;
- MethodUCB – обучение агента на основе верхней доверительной границы;
- getAgent – получение копии агента;
- getField – получение копии поля;
- ClearGame – функция сброса результатов обучения;
- print – функция для вывода результатов обучения агента.

Для начала в классе Game реализуем ε -жадную стратегию:

```
//Эпсилон жадная стратегия
bool Epsilon(double epsilon) {
    double probability = GetRandomNumber(1, 999) * 0.001;
    if (probability < epsilon) { return 1; }
    return 0;
}
```

Листинг 1. Функция ε -жадной стратегии

Данная функция будет принимать значение epsilon, которая будет указывать на процент случайных действий и возвращать булеву переменную 1, если случайно сгенерированное число меньше epsilon, и 0 в противном случае. Здесь GetRandomNumber представляет из себя простейшую функцию генерации случайного числа в указанном промежутке.

Реализуем UCB метод. Для этого создадим трехмерную матрицу Qt_a, которая будет хранить оценку награды для каждой возможной комбинации состояние-действие. Также введем такую же матрицу Nt_a, но при этом она будет показывать, сколько раз мы выполняли возможные действия в каждом из состояний.

Выбор последующего жадного шага для агента будет осуществляться в соответствии со стратегией текущего метода в функции argmaxUCB. В нем будет проверяться, находятся ли соседние к агенту ячейки внутри поля, а затем произойдет отбор наилучшего состояния на основе верхней доверительной границы для следующего хода.

Таким образом, на выходе будет получаться действие, которое нужно сделать, чтобы получить максимальную ожидаемую награду.

Значения в введенных полях будут обновляться через функцию UpdateUCB:

```
void UpdateUCB(int action) {
    int x = agent.getx();
    int y = agent.gety();

    t += 1;
    Rt += field.getFieldElement(x, y);
    view_t += 1;
    view_Rt += field.getFieldElement(agent.getx(), agent.gety());

    double Qt_new = view_Rt / Nt[x][y];
    Qt_a[x][y][action] = Qt_new;
}
```

Листинг 2. Функция обновления ожидаемой ценности при действии action для UCB

В этой функции t – общее количество совершенных шагов, view_t – количество шагов в текущей итерации, view_Rt – сумма наград, собранных в текущей итерации обучения, Rt – общая сумма собранных наград за весь промежуток обучения агента, Nt – матрица, хранящая количество посещений каждой ячейки поля.

С учетом листинга 2, один шаг агента будет выполняться в функции UCB, которая будет вызывать функцию argmaxUCB для нахождения максимизирующего действия и UpdateUCB для обновления весов в Qt_a и шагов в Nt_a.

Один проход по полю реализуем через UCBLearning с учетом эпсилон-жадной стратегии. При этом, когда будет выбираться случайное действие, данная функция будет вызывать RandomStepUpdateUCB, специально для этого сделанная. Иначе будет вызвана функция UCB.

И полное обучение в несколько итераций будет уже проходить в MethodUCB:

```
void MethodUCB() {
    Qt_a = CreatVector(width, height, 4); //right, left, down, up
    Nt_a = CreatVector(width, height, 4);
    Nt = CreatVector(width, height);
}
```

```

for (int i = 0; i < iteration_count; i++) {
    vector<int> getres = UCBLearning(i);
}
cout << "\nUCB learning was successfull\n";
}

```

Листинг 3. Функция обучения по методу UCB

Обучение через TD-метод не будет сильно отличаться своей структурой от предыдущего способа обучения. На этот раз, вместо параметра c введем переменные α (α), определяющую, насколько сильно новые значения весов будут учитывать предыдущие, и γ (γ), показывающую, насколько будущие вознаграждения будут учитываться в обучении.

В этот раз функция `argmaxTD` будет выбирать действие с наибольшей оценкой. При этом, не посещённые ячейки также будут считаться максимизирующими, чтобы агент активнее исследовал пространство.

Значительное изменение претерпит функция обновления, где и будет реализована значительная часть метода обучения через временные разности:

```

void UpdateTD(int action) {
    int x = agent.getx();
    int y = agent.gety();
    int predx = agent.getpredx();
    int predy = agent.getpredy();
    double R_next = field.getFieldElement(x, y);
    t += 1;
    view_t += 1;
    view_Rt += field.getFieldElement(x, y);
    double V_St_pred = Qt_a[predx][predy][action];
    double V_St_next = Qt_a[x][y][argmaxTD(x, y)];
    double V_St_new = V_St_pred + alpha * (R_next + gamma * V_St_next -
V_St_pred);
    Qt_a[predx][predy][action] = V_St_new;
    Nt[x][y] += 1; }

```

Листинг 4. Функция обновления ожидаемой ценности при действии `action` для TD

Остальные функции будут практически идентичными. По итогу в `public` секции будет доступна функция `MethodTD` для запуска цикла обучения в несколько итераций:

```

void MethodTD() {
    Qt_a = CreatVector(width, height, 4); //right, left, down, up
    Nt = CreatVector(width, height);

    for (int i = 0; i < iteration_count; i++) {
        vector<int> getres = TDLearning(i);
    }
    cout << "\nTD learning was successfull\n";
}

```

Листинг 5. Функция обучения по методу TD

Попробуем подобрать такие параметры для методов, чтобы скорость работы была наиболее быстрой, а также чтобы в итоге агент совершал минимальное количество шагов и получал максимально возможную награду.

Так как у UCS скорость обучения становится слишком долгой для полей 5*5 и выше, возьмем в качестве тестового полигона поле 4*4. Награды за посещения клеток определенного цвета будут следующими:

Таблица 1. Таблица наград, содержащихся в клетках каждого из цветов

Белая	Синяя	Красная	Желтая	Зеленая	Черная
-1	-2	-3	-4	-5	-15

Для каждой уникальной клетки выберем распределения вероятностей, указанных в таблице 2:

Таблица 2. Таблица распределения вероятности выбора следующей ячейки в зависимости от того, какого цвета текущее состояние. По горизонтали цвет текущей ячейки, по вертикали цвет клетки при следующем шаге

	Белая	Синяя	Красная	Желтая	Зеленая	Черная
Белая	0.25	0.1	0.3	0.25	0.05	0.05
Синяя	0.2	0.15	0.2	0.3	0.05	0.1
Красная	0.1	0.1	0.1	0.55	0.05	0.1
Желтая	0.25	0.1	0.3	0.25	0.05	0.05
Зеленая	0.25	0.15	0.25	0.15	0.1	0.1
Черная	0.15	0.1	0.4	0.15	0.1	0.1

За переход в конечную клетку агент будет получать максимальную награду, значение которой выставим в 25, чтобы агент охотнее стремился к окончанию итерации.

Далее попробуем поэкспериментировать с параметром c в UCS обучении. Для начала определим $\epsilon_{UCS} = 0.01$. Это будет означать, что с вероятностью 1 % агент будет выбирать случайное действие на каждом шаге. Количество итераций будет равно 200. Получаем следующие графики:

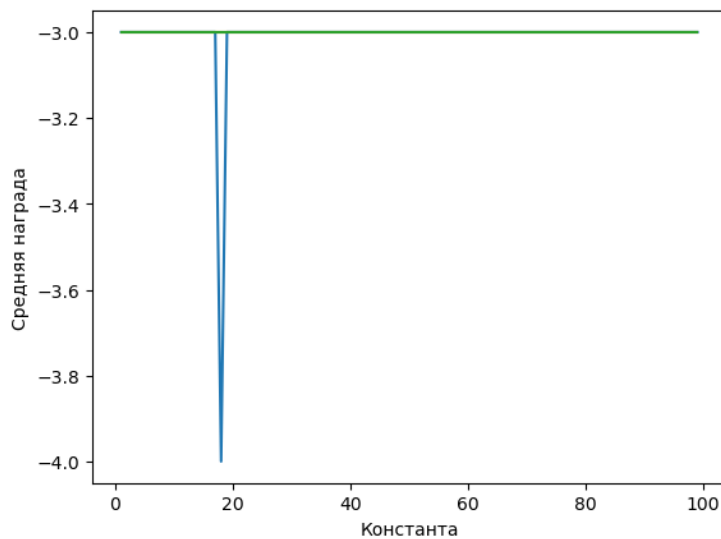


Рисунок 1. Зависимость средней награды от параметра c (зеленый график – зависимость, полученная при первом тестировании, синий – зависимость, полученная при тех же параметрах при втором тестировании)

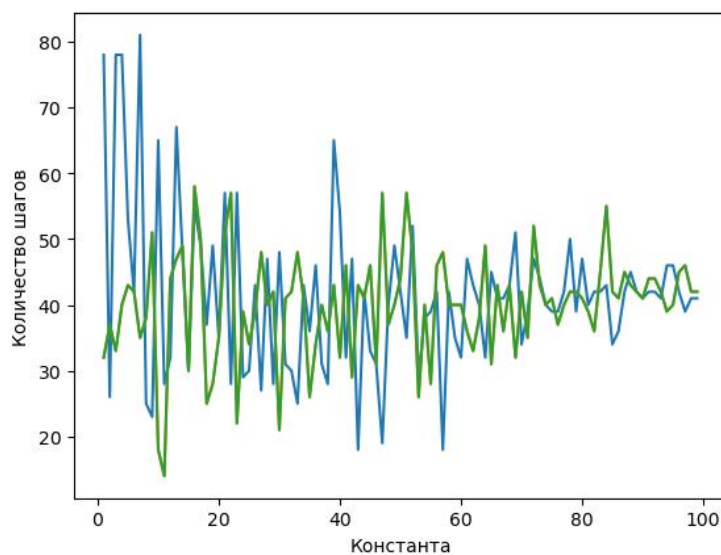


Рисунок 2. Зависимость количества шагов от параметра c (зеленый график – зависимость, полученная при первом тестировании, синий – зависимость, полученная при тех же параметрах при втором тестировании)

Отсюда можно сделать вывод, что при c меньших 30, модель начинает более случайно исследовать среду. При константе, большем 30, количество шагов увеличивается. Таким образом, удачным коэффициентом можно считать $c = 30$.

В итоге, значения награды и шага от итерации выглядят так:

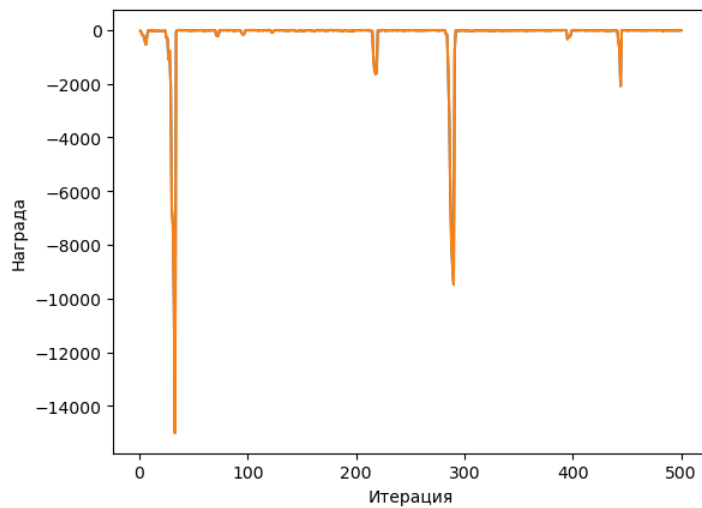


Рисунок 3. Награда, получаемая на каждой итерации

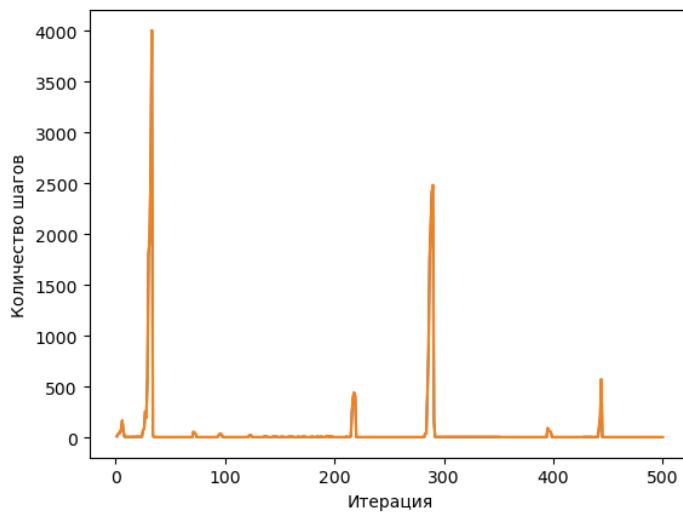


Рисунок 4. Количество шагов, которые агент совершает на каждой итерации

После обучения таблица весов для агента будет выглядеть следующим образом:

Таблица 3. Матрица весов для UCS-обучения, с помощью которой агент будет предпринимать действия при хождении по полю. Здесь учитывается как каждое состояние, так и каждой действие

Правый ход			
-0.000563698	-0.00474383	-0.439689	0
-0.524115	-26.3129	-0.0505145	0
-1.32543	-145.727	-0.0105871	0
-9.47059	-4720.4	-4720.6	0
Левый ход			
0	-0.360644	-0.298938	-0.206667
0	-1.15583	-6.24091	-0.0236601

0	-0.257919	-45.7143	-0.0184805
0	-1.05078	-11.0417	0
Нижний ход			
-0.000765111	-1.49299	-0.0245614	-0.286928
-0.00543884	-7.98246	-0.0987654	-0.00118819
-1.32129	-28.3247	-10.1154	-0.00262626
0	0	0	0
Верхний ход			
0	0	0	0
-0.00212258	-0.610868	-5.77866	-0.0137221
-0.00268513	-14.916	-66.4545	-0.0123703
-0.0786026	-45.9091	-3040.5	0

На основе полученных данных, можно увидеть, что агент будет делать следующие ходы: вправо-вправо-вниз-вправо-вниз-вниз.

Теперь попробуем поиграться с параметрами в обучении на основе временных разностей. Выставим $\epsilon_{TD} = 0.01$ и создадим, как и в UCSB, поле размером 4×4 . Для начала будем менять параметр γ :

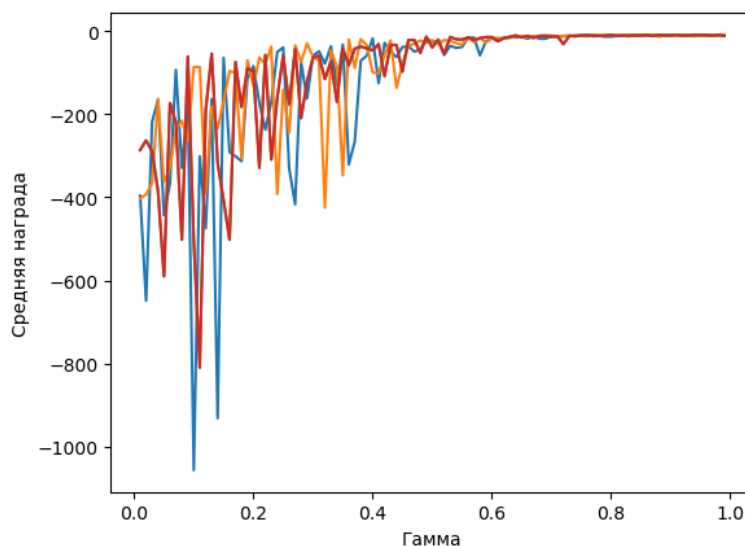


Рисунок 5. Зависимость средней награды от параметра гамма (красный график – первое тестирование, желтый – второе тестирование, синий – третье тестирование)

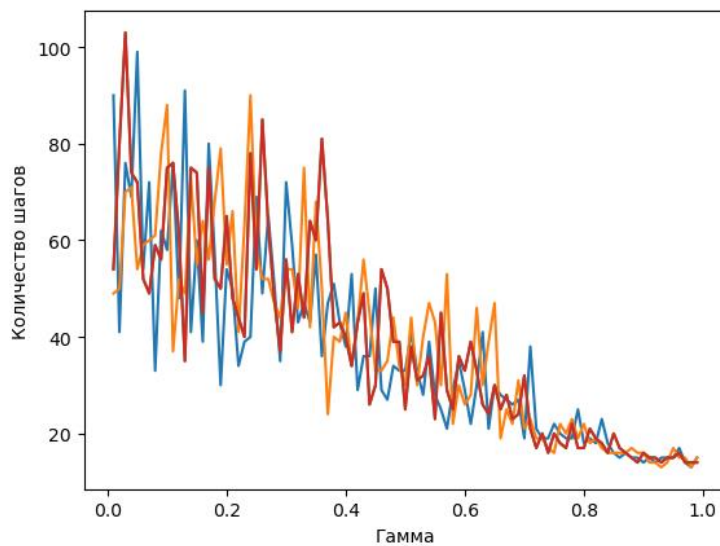


Рисунок 6. Зависимость количества шагов от параметра гамма (красный график – первое тестирование, желтый – второе тестирование, синий – третье тестирование)

Как мы видим, с приближением гаммы к единице суммарная награда растет экспоненциально, а количество шагов убывает практически линейно. Таким образом, гамму можно приравнять к 0.99.

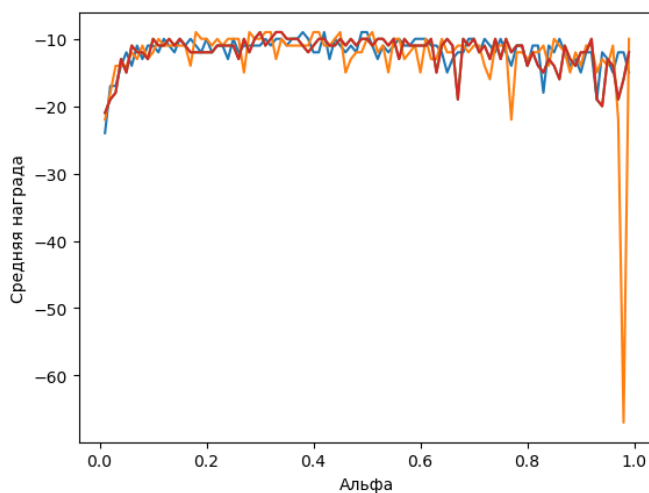


Рисунок 7. Зависимость средней награды от параметра альфа (красный график – первое тестирование, желтый – второе тестирование, синий – третье тестирование)

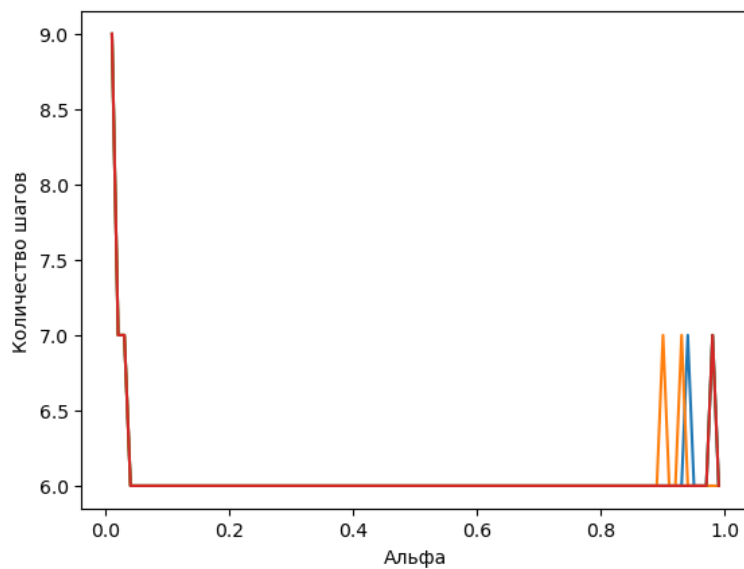


Рисунок 8. Зависимость количества шагов от параметра альфа (красный график – первое тестирование, желтый – второе тестирование, синий – третье тестирование)

В этот раз видно, что первый график вначале растёт, а после с некоторого момента начинает убывать. Количество шагов перестаёт меняться почти сразу же. Поэтому, можно сделать вывод, что наиболее оптимальным α будет 0.4.

Таким образом, график обучения после подбора параметров будет следующим:

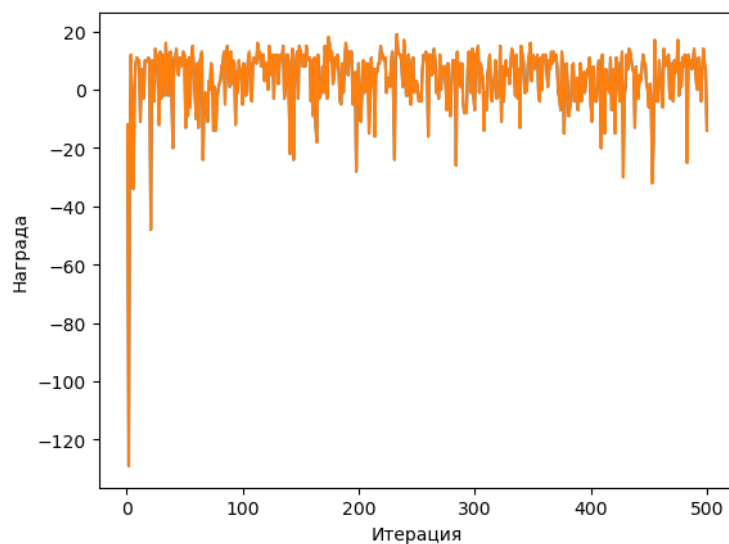


Рисунок 9. Награда, получаемая на каждой итерации

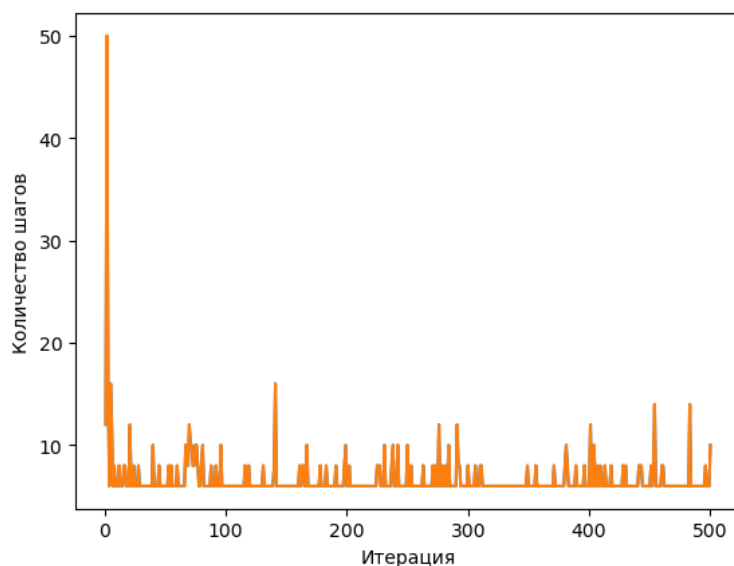


Рисунок 10. Количество шагов, которые агент совершает на каждой итерации

Также получаем матрицу весов для каждого хода:

Таблица 4. Матрица весов для TD-обучения, с помощью которой агент будет предпринимать действия при хождении по полю. Как и в UCSB учитываются как каждое состояние, так и каждой действие

Правый ход			
-12.9625	-3.83183	-3.5242	0
-2.03302	1.21797	7.70567	0
-1.39337	-5.06752	15.1265	0
-7.04184	14.2877	24.8488	0
Левый ход			
0	-9.17424	-5.05997	-2.88
0	-8.41564	-4.59853	-2
0	-11.1946	-0.736	5.45884
0	-1.31776	-1.6	0
Нижний ход			
-9.07392	-8.97785	-0.0382737	5.90971
-11.2422	-6.35808	-0.723613	14.9131
-9.99248	4.48008	-0.4	25
0	0	0	0
Верхний ход			
0	0	0	0
-12.901	-8.87611	-4.63288	-0.198933
-7.98298	-8.62164	-1.2	0
-9.07371	-1.952	-0.512	0

В этот раз агент предпочел идти вниз-вправо-вправо-вправо-вниз-вниз.

Сравним результаты времени работы алгоритмов для поля 4*4 при количестве итераций обучения 500 и при полученных оптимальных параметрах. UCS в таком случае отрабатывает за 0.730 секунд, а TD за 0.207. Видим, что второй метод обучения справляется лучше первого примерно в 3.5 раза быстрее.

Вывод:

По итогу удалось реализовать алгоритмы обучений с подкреплением на основе верхней доверительной границы (UCB) и методом временных разностей (TD) на языке C++. В конечном счете получилось выявить наилучшие параметры ($c = 30$, $\gamma = 0.99$ и $\alpha = 0.4$) для нашей задачи, а также сравнить эффективности двух подходов на примере. Как и ожидалось, UCS хуже справляется с нестатическим полем, по которому перемещается агент.

Заключение

1. В ходе работы были изучены алгоритмы обучения с подкреплением UCB и TD. Было освоено понятие подкрепления, моделирование среды и выбор оптимальных стратегий на основе полученного опыта. Это знание позволило лучше понять принципы работы и эффективность данных алгоритмов.
2. Была проведена практическая реализация алгоритмов обучения с подкреплением на языке программирования C++. Были созданы необходимые классы для представления игрового поля, агента и самой игры. Были также разработаны функции и методы, обеспечивающие корректную работу алгоритмов. Это позволило проверить работоспособность алгоритмов и провести исследование их эффективности.
3. Был проведен анализ эффективности алгоритмов UCB и TD. Были рассмотрены влияющие параметры, такие как c в обучении на основе верхней доверительной границы, гамма и альфа в методе временных разностей, а также проведены сравнения результатов работы алгоритмов. В итоге на практике был сделан вывод, что TD-обучение лучше подходит для поставленной задачи, нежели UCB.

Код решения задачи приведен в [Приложении](#).

Список используемой литературы можно посмотреть [здесь](#).

Список литературы

1. Саттон Р. С., Обучение с подкреплением: Введение. 2-е изд. / Саттон Р. С., Барто Э. Дж. [пер. с англ. А. А. Слинкина.] – Москва: ДМК Пресс, 2020 – 552 с. ISBN 978-5-97060-097-9
2. Лонца А., Алгоритмы обучения с подкреплением на Python [пер. с англ. А. А. Слинкина.] – Москва: ДМК Пресс, 2020 – 286 с. ISBN 978-5-97060-855-5
3. Лю Ю., Обучение с подкреплением на PyTorch: сборник рецептов [пер. с англ. А. А. Слинкина.] – Москва: ДМК Пресс, 2020 – 282 с. ISBN 978-5-97060-853-1
4. Судхарсан Р., Глубокое обучение с подкреплением на Python. OpenAI Gym и TensorFlow для профи. [пер. с англ. Е. Матвеев] – СПб.: Питер, 2019 – 250 с. ISBN 978-5-4461-1251-7
5. Грессер Л., Глубокое обучение с подкреплением: теория и практика на языке Python / Грессер Л., Кенг В. Л. [пер. с англ. Панов А. И.] – СПб.: Питер, 2022 – 416 с. ISBN 978-5-4461-1699-7

Приложение

```
#include <iostream>
#include <vector>
#include <string>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
using namespace std;

vector<vector<vector<double>>> CreatVector(int width, int height, int
action_count)
{
    vector<vector<vector<double>>> field;
    for (int x = 0; x < width; x++)
    {
        vector<vector<double>> fieldy;
        for (int y = 0; y < height; y++)
        {
            vector<double> fielda;
            for (int a = 0; a < action_count; a++)
            {
                fielda.push_back(0);
            }
            fieldy.push_back(fielda);
        }
        field.push_back(fieldy);
    }
    return field;
}

vector<vector<double>> CreatVector(int width, int height, double value = 0)
{
    vector<vector<double>> field;
    for (int x = 0; x < width; x++)
    {
        vector<double> fieldy;
        for (int y = 0; y < height; y++)
        {
            fieldy.push_back(value);
        }
        field.push_back(fieldy);
    }
    return field;
}

//Функция для получения случайного натурального числа от min до max
int GetRandomNumber(int min, int max)
{
    int num = min + rand() % (max - min + 1);
    return num;
}

ostream& operator<< (ostream& stream, vector<double> v)
{
    stream << "\nVector:\n( ";
    for (int i = 0; i < v.size() - 1; i++) {
        stream << v[i] << ", ";
    }
    stream << v[v.size() - 1];
    stream << " )\n";

    return stream;
}
```

```

ostream& operator<< (ostream& stream, vector<vector<double>>> v)
{
    stream << "\nVector:\n";
    for (int y = 0; y < v[0].size(); y++) {
        for (int x = 0; x < v.size(); x++) {
            stream << v[x][y] << "\t";
        }
        stream << "\n";
    }
    return stream;
}

ostream& operator<< (ostream& stream, vector<vector<vector<double>>>> v)
{
    stream << "\nVector:\n";
    for (int a = 0; a < v[0][0].size(); a++) {
        if (a == 0) { cout << "Right:\n"; }
        else if (a == 1) { cout << "Left:\n"; }
        else if (a == 2) { cout << "Down:\n"; }
        else if (a == 3) { cout << "Up:\n"; }

        for (int y = 0; y < v[0].size(); y++) {
            for (int x = 0; x < v.size(); x++) {
                stream << v[x][y][a] << "\t";
            }
            stream << "\n";
        }
        stream << "-----\n";
    }
    return stream;
}

//Дискретное поле
class Field {
protected:
    int width;
    int height;

    vector<vector<double>>> mas;
    vector<vector<double>>> getField() const { return mas; }

    // Награда за клетку:   ||| Вероятность, что следующая клетка будет
    // указанного цвета
    // 0 - белая клетка (б)   - б: 0.25, с: 0.1, к: 0.3, ж: 0.25, з: 0.05, ч:
    0.05
    // 1 - синяя клетка (с)   - б: 0.25, с: 0.1, к: 0.3, ж: 0.25, з: 0.05, ч:
    0.05
    // 2 - красная клетка (к) - б: 0.25, с: 0.1, к: 0.3, ж: 0.25, з: 0.05, ч:
    0.05
    // 3 - желтая клетка (ж)  - б: 0.25, с: 0.1, к: 0.3, ж: 0.25, з: 0.05, ч:
    0.05
    // 4 - зеленая клетка (з) - б: 0.25, с: 0.1, к: 0.3, ж: 0.25, з: 0.05, ч:
    0.05
    // -2 - черная клетка (ч) - б: 0.25, с: 0.1, к: 0.3, ж: 0.25, з: 0.05, ч:
    0.05
    // 25 - выход
    vector<vector<double>>> propabilities = { { 0.25, 0.1, 0.3, 0.25, 0.05, 0.05 },
                                                { 0.2, 0.15, 0.2, 0.3, 0.05, 0.1 },
                                                { 0.1, 0.1, 0.1, 0.55, 0.05, 0.1 },
                                                { 0.25, 0.1, 0.3, 0.25, 0.05, 0.05 },
                                                { 0.25, 0.15, 0.25, 0.15, 0.1, 0.1 },
                                                { 0.15, 0.1, 0.4, 0.15, 0.1, 0.1 } };

    vector<int> rewards = { -1, -2, -3, -4, -5, -15 };
    int max_reward = 25;
}

```

```

public:
    Field() { width = 0; height = 0; }
    Field(int w, int h) { width = w; height = h; mas = CreatVector(w, h); }
    Field(int w, int h, double value) { width = w; height = h; mas =
CreatVector(w, h, value); }
    Field(vector<vector<double>> f, int w, int h) { mas = f; width = w; height =
h; }
    Field(const Field& f) { width = f.getWidth(); height = f.getHeight(); mas =
f.getField(); }

    int getWidth() const { return width; }
    int getHeight() const { return height; }
    double getFieldElement(int x, int y) const { return mas[x][y]; }

    void setValue(int x, int y, double value) {
        if (x >= width or x < 0 or y >= height or y < 0) { throw "Value isn't
correct"; }
        mas[x][y] = value;
    }

    int getReward(int this_reward) {
        double pr = GetRandomNumber(0, 99)*0.01;
        if (this_reward == rewards[5]) {
            this_reward = 5;
        }
        double sumpr = 0;
        for (int i = 0; i < propabilities.size(); i++) {
            sumpr += propabilities[this_reward][i];
            if (sumpr >= pr) {
                return rewards[i];
            }
        }
    }
    void setNextReward(vector<int> thispos, vector<int> nextpos) {
        if (nextpos[0] == width - 1 and
            nextpos[1] == height - 1)
        {
            mas[nextpos[0]][nextpos[1]] = max_reward;
        }
        else {
            mas[nextpos[0]][nextpos[1]] = getReward(mas[thispos[0]][thispos[1]]);
        }
    }

    Field& operator= (const Field& other) {
        width = other.getWidth(); height = other.getHeight();
        this->mas = other.getField(); return *this; }

    friend ostream& operator<< (ostream& stream, const Field& f);
};

ostream& operator<< (ostream& stream, const Field& f)
{
    stream << "\nField:\n";
    for (int y = 0; y < f.height; y++) {
        for (int x = 0; x < f.width; x++) {
            stream << f.mas[x][y] << "\t";
        }
        stream << "\n";
    }
    return stream;
}

//Агент, исследующий поле
class Agent {

```



```

protected:
    int start_pos_x;
    int start_pos_y;

    int pred_pos_x;
    int pred_pos_y;

    int pos_x;
    int pos_y;
public:
    Agent() { start_pos_x = start_pos_y =
              pos_x = pos_y =
              pred_pos_x = pred_pos_y = 0; }
    Agent(int x, int y) {
        if (x < 0 or y < 0)
            throw "Coordinates of the agent aren't correct!";
        start_pos_x = pos_x = pred_pos_x = x; start_pos_y = pos_y = pred_pos_y =
y; }
    Agent(const Agent& a) {
        start_pos_x = a.getStartPosition()[0]; start_pos_y =
a.getStartPosition()[1];
        pos_x = a.getPosition()[0]; pos_y = a.getPosition()[1];
        pred_pos_x = a.getPredPosition()[0]; pred_pos_y = a.getPredPosition()[1];
    }
    ~Agent() {}

    vector<int> getPosition() const { vector<int> res = { pos_x, pos_y}; return
res; }
    void setPosition(vector<int> pos) { pos_x = pos[0]; pos_y = pos[1]; }
    void setPosition(int x, int y) { pos_x = x; pos_y = y; }

    vector<int> getPredPosition() const { vector<int> res = { pred_pos_x,
pred_pos_y }; return res; }
    void setPredPosition(vector<int> predpos) { pred_pos_x = predpos[0];
pred_pos_y = predpos[1]; }
    void setPredPosition(int x, int y) { this->pred_pos_x = x; this->pred_pos_y =
y; }

    vector<int> getStartPosition() const { vector<int> res = { start_pos_x ,
start_pos_y }; return res; }
    void RestartPosition() { pos_x = pred_pos_x = start_pos_x; pos_y = pred_pos_y
= start_pos_y; }
    int getx() { return pos_x; }
    int gety() { return pos_y; }

    int getpredx() { return pred_pos_x; }
    int getpredy() { return pred_pos_y; }

    vector<int> RandomStep()
    {
        //выбираем, по какой оси будем делать шаг
        int xory = GetRandomNumber(0, 1); //0 - x, 1 - y
        //размер шага
        int step = GetRandomNumber(0, 1); //шаг -1 либо 1
        if (step == 0) { step = -1; }

        vector<int> result;
        if (xory)
        {
            result.push_back(pos_x);
            result.push_back(pos_y + step);
        }
        else {
            result.push_back(pos_x + step);
            result.push_back(pos_y);
        }
    }

```

```

    }
    return result;
}

Agent& operator= (const Agent& a) {
    this->start_pos_x = a.getStartPosition()[0]; this->start_pos_y =
a.getStartPosition()[1];
    this->pos_x = a.getPosition()[0]; this->pos_y = a.getPosition()[1];
    this->pred_pos_x = a.getPredPosition()[0]; this->pred_pos_y =
a.getPredPosition()[1];
    return *this; }
};

class Game {
protected:
    Agent agent;
    Field field;

    // Определяет динамичность поля
    bool propabilities_color = true;

    int t = 0; // общее кол-во шагов
    double Rt = 0; //общее полученное вознаграждение к текущему моменту
    int view_t = 0; //кол-во шагов в текущей итерации
    double view_Rt = 0; //набранное вознаграждение в текущей итерации
    vector<double> result_Rt;
    vector<double> result_t;

    int width = 0;
    int height = 0;
    vector<vector<vector<double>>> Nt_a; // сколько раз действие а выбиралось для
каждого состояния до момента t (для UCSB)
    vector<vector<vector<double>>> Qt_a; //ожидаемое вознаграждение при действии а
для каждого состояния
    vector<vector<double>> Nt; // кол-во шагов в текущую клетку

protected:
    int Action(vector<int> pos, vector<int> pred_pos) {
        int action = 0;
        if (pos[0] - pred_pos[0] == -1) {
            action = 1;
        }
        else if (pos[1] - pred_pos[1] == 1) {
            action = 2;
        }
        else if (pos[1] - pred_pos[1] == -1) {
            action = 3;
        }
        return action;
    }
    vector<int> Action(vector<int> pos, int act) {
        vector<int> res = pos;
        if (act == 0) {
            res[0] += 1;
        }
        else if (act == 1) {
            res[0] -= 1;
        }
        else if (act == 2) {
            res[1] += 1;
        }
        else if (act == 3) {
            res[1] -= 1;
        }
    }
};

```

```

        return res;
    }
    bool BorderPosition() {
        int x = agent.getx();
        int y = agent.gety();
        if (x >= 0 and x < field.getWidth() and
            y >= 0 and y < field.getHeight())
        {
            return 1;
        }
        return 0;
    }
    bool BorderPosition(int x, int y) {
        if (x >= 0 and x < field.getWidth() and y >= 0 and y < field.getHeight())
        {
            return 1;
        }
        return 0;
    }
    //Эпсилон жадная стратегия
    bool Epsilon(double epsilon) {
        double probability = GetRandomNumber(1, 999) * 0.001;
        if (probability < epsilon) { return 1; }
        return 0;
    }
    void Clear() { view_t = 0; view_Rt = 0; }
    //=====
    //Обучение через метод UCB
    vector<int> UCBLearning(int iteration) {
1)))    if (print_agent or (last_iteration and iteration == (iteration_count -
        {
            cout << *this << "\n===== \n";
        }
        while (1) {
            if (Epsilon(epsilon_ucb)) { RandomStepUpdateUCB(iteration); }
            else { UCB(iteration); }

            if (agent.getx() == (field.getWidth() - 1) and agent.gety() ==
(field.getHeight() - 1))
            {
                agent.RestartPosition();
                vector<int> res;
                res.push_back(view_t);
                res.push_back(view_Rt);
                result_t.push_back(view_t);
                result_Rt.push_back(view_Rt);

                Clear();
                return res;
            }
        }
    }
    void UCB(int iteration) {
        int x = agent.getx();
        int y = agent.gety();
        vector<int> oldpos = agent.getPosition();

        int action = argmaxUCB(x, y);
        Nt_a[x][y][action] += 1;
        vector<int> newpos = Action(agent.getPosition(), action);

        // Задаем случайный цвет, в этом случае поле нестатично
        if (propabilities_color) {
            field.setNextReward(oldpos, newpos);

```

```

    }

    UpdateUCB(action);
    Nt[newpos[0]][newpos[1]] += 1;
    agent.setPredPosition(x, y);
    agent.setPosition(newpos[0], newPos[1]);

    if (print_agent or (last_iteration and iteration == (iteration_count -
1)))
    {
        cout << *this << "\n=====\\n";
    }
}

void RandomStepUpdateUCB(int iteration) {
    vector<int> newPos = agent.RandomStep();

    if (BorderPosition(newpos[0], newPos[1])) {
        vector<int> pos = agent.getPosition();
        int action = Action(newpos, pos);
        // Задаем случайный цвет, в этом случае поле нестатично

        if (propabilities_color) {
            field.setNextReward(pos, newPos);
        }

        UpdateUCB(action);
        Nt_a[pos[0]][pos[1]][action] += 1;
        Nt[newpos[0]][newpos[1]] += 1;
        agent.setPredPosition(pos[0], pos[1]);
        agent.setPosition(newpos[0], newPos[1]);

        if (print_agent or (last_iteration and iteration == (iteration_count -
1)))
        {
            cout << *this << "\n=====\\n";
        }
    }
    else
        RandomStepUpdateUCB(iteration);
}

int argmaxUCB(int x, int y) {
    vector<int> pos = { x, y };
    vector<int> actions;

    if (BorderPosition(x + 1, y)) {
        actions.push_back(0);
    }
    if (BorderPosition(x - 1, y)) {
        actions.push_back(1);
    }
    if (BorderPosition(x, y + 1)) {
        actions.push_back(2);
    }
    if (BorderPosition(x, y - 1)) {
        actions.push_back(3);
    }

    int act_max = actions[GetRandomNumber(0, actions.size() - 1)];
    for (int i = 0; i < actions.size(); i++) {

```

```

        double value1 = Qt_a[x][y][actions[i]] + c * sqrt(log(t) /
Nt_a[x][y][actions[i]]);
        double value2 = Qt_a[x][y][act_max] + c * sqrt(log(t) /
Nt_a[x][y][act_max]);
        if (value1 > value2)
            act_max = actions[i];
    }

    vector<int> max_actions;
    max_actions.push_back(act_max);
    for (int i = 0; i < actions.size(); i++) {
        vector<int> nextpos = Action(pos, actions[i]);

        double value1 = Qt_a[x][y][actions[i]] + c * sqrt(log(t) /
Nt_a[x][y][actions[i]]);
        double value2 = Qt_a[x][y][act_max] + c * sqrt(log(t) /
Nt_a[x][y][act_max]);
        if (value1 == value2 or Nt[nextpos[0]][nextpos[1]] == 0)
            max_actions.push_back(actions[i]);
    }

    act_max = max_actions[GetRandomNumber(0, max_actions.size() - 1)];
    return act_max;
}

void UpdateUCB(int action) {
    int x = agent.getx();
    int y = agent.gety();

    t += 1;
    Rt += field.getFieldElement(x, y);
    view_t += 1;
    view_Rt += field.getFieldElement(agent.getx(), agent.gety());

    double Qt_new = view_Rt / Nt[x][y];
    Qt_a[x][y][action] = Qt_new;
} //=====
//TD - обучение
vector<int> TDLearning(int iteration) {
    if (print_agent or (last_iteration and iteration == (iteration_count -
1)))
    {
        cout << *this << "\n===== \n";
    }
    while (1) {
        if (Epsilon(epsilon_td)) { RandomStepUpdateTD(iteration); }
        else { TD(iteration); }

        if (agent.getx() == (field.getWidth() - 1) and agent.gety() ==
(field.getHeight() - 1)) {
            agent.RestartPosition();
            vector<int> res;
            res.push_back(view_t);
            res.push_back(view_Rt);
            result_t.push_back(view_t);
            result_Rt.push_back(view_Rt);

            Clear();
            return res;
        }
    }
}

void TD(int iteration) {
    int x = agent.getx();
    int y = agent.gety();

```

```

        vector<int> pos = agent.getPosition();

        int action = argmaxTD(x, y);
        vector<int> newpos = Action(pos, action);
        // Задаем случайный цвет, в этом случае поле нестатично
        if (propabilities_color) {
            field.setNextReward(pos, newpos);
        }

        agent.setPredPosition(x, y);
        agent.setPosition(newpos[0], newpos[1]);

        UpdateTD(action);

        if (print_agent or (last_iteration and iteration == (iteration_count -
1)))
        {
            cout << *this << "\n=====\\n";
        }
    }

    void RandomStepUpdateTD(int iteration) {
        vector<int> newpos = agent.RandomStep();

        if (BorderPosition(newpos[0], newpos[1])) {
            int action = Action(newpos, agent.getPosition());
            // Задаем случайный цвет, в этом случае поле нестатично
            vector<int> pos = agent.getPosition();
            if (propabilities_color) {
                field.setNextReward(pos, newpos);
            }

            agent.setPredPosition(agent.getx(), agent.gety());
            agent.setPosition(newpos[0], newpos[1]);
            UpdateTD(action);

            if (print_agent or (last_iteration and iteration == (iteration_count -
1)))
            { cout << *this << "\n=====\\n"; }
        }
        else
            RandomStepUpdateTD(iteration);
    }

    int argmaxTD(int x, int y) {
        vector<int> pos = { x, y };
        vector<int> actions;

        if (BorderPosition(x + 1, y)) {
            actions.push_back(0);
        }
        if (BorderPosition(x - 1, y)) {
            actions.push_back(1);
        }
        if (BorderPosition(x, y + 1)) {
            actions.push_back(2);
        }
        if (BorderPosition(x, y - 1)) {
            actions.push_back(3);
        }

        int act_max = actions[GetRandomNumber(0, actions.size()-1)];
        for (int i = 0; i < actions.size(); i++) {
            if (Qt_a[x][y][actions[i]] > Qt_a[x][y][act_max])
                act_max = actions[i];
        }
    }

```

```

    }

    vector<int> max_actions;
    for (int i = 0; i < actions.size(); i++) {
        vector<int> nextpos = Action(pos, actions[i]);

        if (Qt_a[x][y][actions[i]] == Qt_a[x][y][act_max] or
Nt[nextpos[0]][nextpos[1]] == 0)
            max_actions.push_back(actions[i]);
    }

    act_max = max_actions[GetRandomNumber(0, max_actions.size() - 1)];
    return act_max;
}

void UpdateTD(int action) {
    int x = agent.getx();
    int y = agent.gety();

    int predx = agent.getpredx();
    int predy = agent.getpredy();

    double R_next = field.getFieldElement(x, y);

    t += 1;
    view_t += 1;
    view_Rt += field.getFieldElement(x, y);
    Rt += view_Rt;

    double V_St_pred = Qt_a[predx][predy][action];
    double V_St_next = Qt_a[x][y][argmaxTD(x, y)];

    double V_St_new = V_St_pred + alpha * (R_next + gamma * V_St_next -
V_St_pred);
    Qt_a[predx][predy][action] = V_St_new;

    Nt[x][y] += 1;
}

//=====

friend ostream& operator<< (ostream& stream, Game& g);
public:
    //из TD
    double alpha = 0.9; //параметр, определяющий скорость обучения
    double gamma = 0.85; //
    double epsilon_td = 0.05; //для TD обучения

    // Из UCB
    double c = 2.4; //константа из UCB
    double epsilon_ucb = 0.005; // параметр из эpsilon жадной стратегии (для UCB)

    // Общие настройки
    int iteration_count = 100; // сколько раз модель будет обучаться
    bool print_agent = 0; //вывод результата ходьбы агента на каждом шаге (0/1
соответственно)
    bool last_iteration = 0; //вывод ходов агента на последнем обучении

    Game() {}
    Game(Field f) { field = f; }
    Game(Agent a, int width0, int height0) {
        if (width0 <= 1 and height0 <= 1) { throw "The size of field is very
little"; }
        agent = a; Field newf(width0, height0); field = newf;
        width = width0; height = height0;
    }

```

```

        propabilities_color = true;
    }
    Game(Field f, Agent a) {
        if ((a.getPosition())[0] >= f.getWidth() or (a.getPosition())[0] < 0 or
            (a.getPosition())[1] >= f.getHeight() or (a.getPosition())[1] < 0)
        {
            throw "Coordinates of the agent aren't correct!";
        }
        field = f;
        agent = a;
        propabilities_color = false;

        width = f.getWidth();
        height = f.getHeight();
    }
    Game(const Game& g) { agent = g.getAgent(); field = g.getField(); }

    Agent getAgent() const { return agent; }
    Field getField() const { return field; }

    vector<double> get_Rt() { return result_Rt; }
    vector<double> get_t() { return result_t; }
    int getTotal_t() { return t; }
    int getTotal_r() { return Rt; }

    void ClearGame() {
        t = 0; Rt = 0;
        Qt_a = CreatVector(field.getWidth(), field.getHeight(), 4);
        Nt = CreatVector(field.getWidth(), field.getHeight());
        Nt_a = CreatVector(field.getWidth(), field.getHeight(), 4);
        result_Rt.clear(); result_t.clear();
    }
    void MethodUCB() {
        Qt_a = CreatVector(width, height, 4); //right, left, down, up
        Nt_a = CreatVector(width, height, 4);
        Nt = CreatVector(width, height);

        for (int i = 0; i < iteration_count; i++) {
            vector<int> getres = UCBLearning(i);
        }
        cout << "\nUCB learning was successfull\n";
    }
    void MethodTD() {
        Qt_a = CreatVector(width, height, 4); //right, left, down, up
        Nt = CreatVector(width, height);

        for (int i = 0; i < iteration_count; i++) {
            vector<int> getres = TDLearning(i);
        }
        cout << "\nTD learning was successfull\n";
    }
    void print() {
        if (t == 0) { throw "You need do learning to see results"; }

        cout << "\n\nQt_a:\n" << Qt_a;
        cout << "\n\nNt:\n" << Nt;

        cout << "\nRewards:" << get_Rt();
        cout << "\nSteps:" << get_t();
    }
};
ostream& operator<< (ostream& stream, Game& g)
{

```



```

        stream << "\\Game:\\n";
        for (int y = 0; y < (g.field).getHeight(); y++) {
            for (int x = 0; x < (g.field).getWidth(); x++) {
                if (x == ((g.agent).getPosition())[0] and y ==
((g.agent).getPosition())[1])
                    stream << "*\\t";
                else
                    stream << (g.field).getFieldElement(x, y) << "\\t";
            }
            stream << "\\n";
        }
        return stream;
    }
}
int main()
{
    try
    {
        // Для генератора случайных чисел
        srand(time(NULL));

        Agent a(0, 0);

        Game newgame(a, 4, 4);
        cout << newgame << "\\n=====\\n";

        newgame.MethodUCB();
        newgame.print();

        newgame.ClearGame();
        newgame.MethodTD();
        newgame.print();

        return 0;
    }
    catch (const char* error) { cout << error; }
}

```