ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 6

«Динамическое программирование (задача 773.Sliding Puzzle)»

Выполнил работу

Кузнецова Таисия

Академическая группа №J3113

Принято

Ходненко И.В

Санкт-Петербург

2024

**Введение**

**Цель -** обучится подходу решения задач с помощью динамического програмирования

**Задачи**:

1. Изучить принцип работы дп
2. Найти подходящую задачу.
3. Обдумать как применить динамическое программирование в этой задаче
4. Написать код
5. Протестировать
6. **Теоретическая подготовка**

Для этой лабораторной работы я искала информацию о жадных алгоритмах на википедии, ютубе и на habr и других тематических платформах

**Динамическое программирование** (Dynamic Programming, DP) — это метод решения задач, который заключается в разбиении задачи на более простые подзадачи, запоминаниирешений этих подзадач и использовании этих решений для вычисления ответа на исходную задачу. Основная идея заключается в том, чтобы избежать повторных вычислений, сохраняя результаты промежуточных вычислений.

**Особенности:**

Используется для задач, которые можно разбить на перекрывающиеся подзадачи и имеют оптимальную подструктуру.Перекрывающиеся подзадачи означают, что подзадачи решаются многократно.Оптимальная подструктура предполагает, что оптимальное решение задачи можно получить из оптимальных решений её подзадач.Применяется в задачах на поиск оптимального решения, таких как максимизация/минимизация, подсчёт комбинаций и т.д.

**Особенности на C++**

В C++ динамическое программирование реализуется с использованием:

Массивов или векторов:Одномерные или многомерные массивы (например, dp[i] или dp[i][j]) используются для хранения результатов подзадач.Это обеспечивает доступ к результатам за O(1).

Рекурсии с мемоизацией: Вместо итеративного подхода можно использовать рекурсию, запоминая уже решённые подзадачи с помощью, например, std::unordered\_map.

Табуляции:Итеративный метод заполнения таблицы снизу вверх. Более популярный и эффективный способ, так как он не требует затрат памяти на стек вызовов.

Преимущества на C++:

Высокая производительность: статические массивы и точное управление памятью дают преимущество над интерпретируемыми языками.

Гибкость: использование STL (векторы, хэши) упрощает реализацию сложных задач.

1. **Реализация**
2. Нашла задачу, которую нужно реализовать ( задача 773.Sliding Puzzle).В ней нужно решить головоломку.На доске размером 2 х 3 расположены пять плиток, помеченных от 1 до 5, и пустой квадрат, обозначенный 0. Ход состоит в выборе 0 и числа, расположенного рядом в 4 направлениях, и замене их местами.Состояние доски разгадывается тогда и только тогда, когда на доске находится доска для головоломок. [[1,2,3],[4,5,0]].

Выполните наименьшее количество ходов, необходимых для разгадки состояния доски. Если невозможно определить состояние платы, верните значение -1.

1. Преобразование состояния в строку.

Первым шагом я преобразовала двухмерное состояние доски в строку. Это сделано для того, чтобы было удобно хранить и обрабатывать различные состояния. Каждое состояние в программе представляется строкой, где элементы доски представлены цифрами от 0 до 5 (где 0 — это пустая плитка). Строки позволили эффективно работать с состоянием в алгоритмах поиска.

1. Проверка на разрешимость

Для проверки, является ли начальная конфигурация разрешимой, была использована концепция инверсий. Инверсия — это ситуация, когда плитка стоит перед другой, но по порядку должна быть после неё. Задача решаема, если количество инверсий в строке чётное.

1. Выбор алгоритма

Для поиска минимального пути к решению я выбрала алгоритм A\*. Этот алгоритм сочетает два аспекта: количество шагов, которые были сделаны (расстояние) и эвристическую функцию, которая оценивает, насколько близко текущее состояние к целевому.Эвристическая функция использует манхэттенское расстояние для каждой плитки, которое измеряет, сколько шагов нужно, чтобы переместить плитку в её правильное место. Это эффективная мера, так как она позволяет быстро оценивать стоимость перемещения плитки.

1. Основной цикл

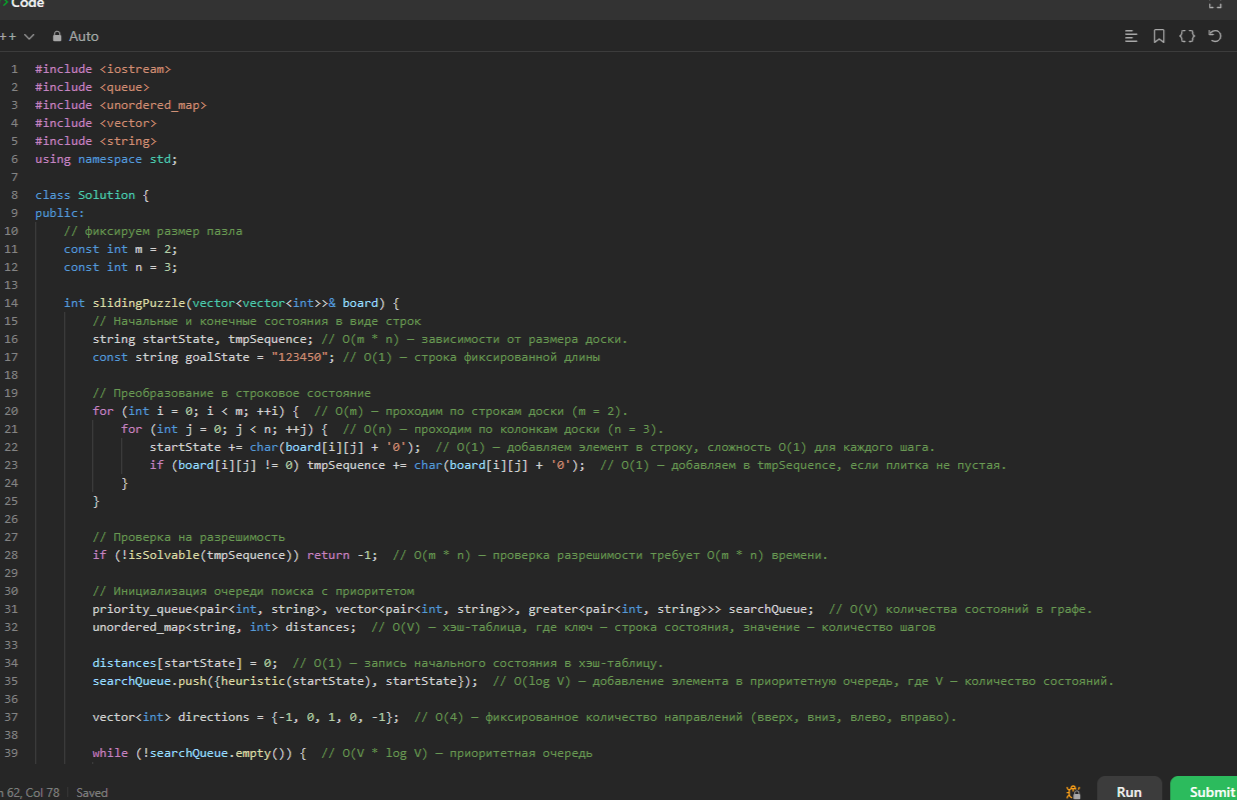
Алгоритм A\* использует приоритетную очередь, где состояния с меньшей стоимостью (сумма текущих шагов и эвристической оценки) обрабатываются в первую очередь. В каждом шаге программа пытается переместить пустую плитку в одно из соседних положений (вверх, вниз, влево или вправо) и проверяет, может ли полученное состояние привести к решению.

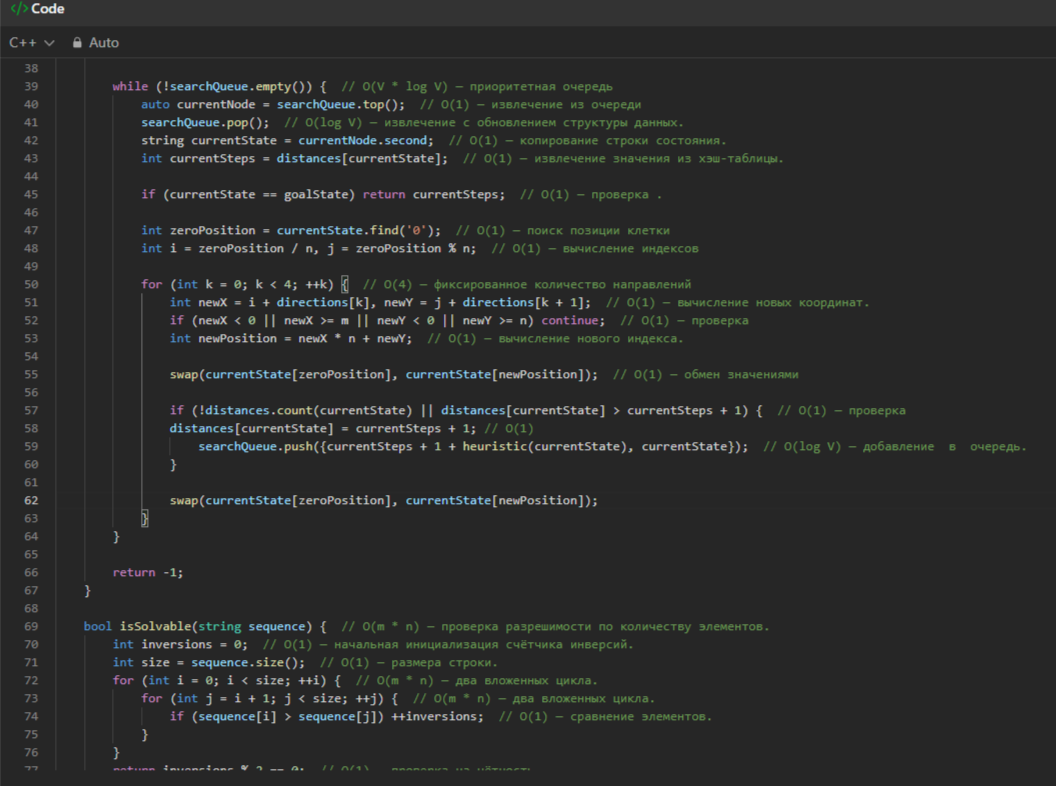
1. Возвращение результата

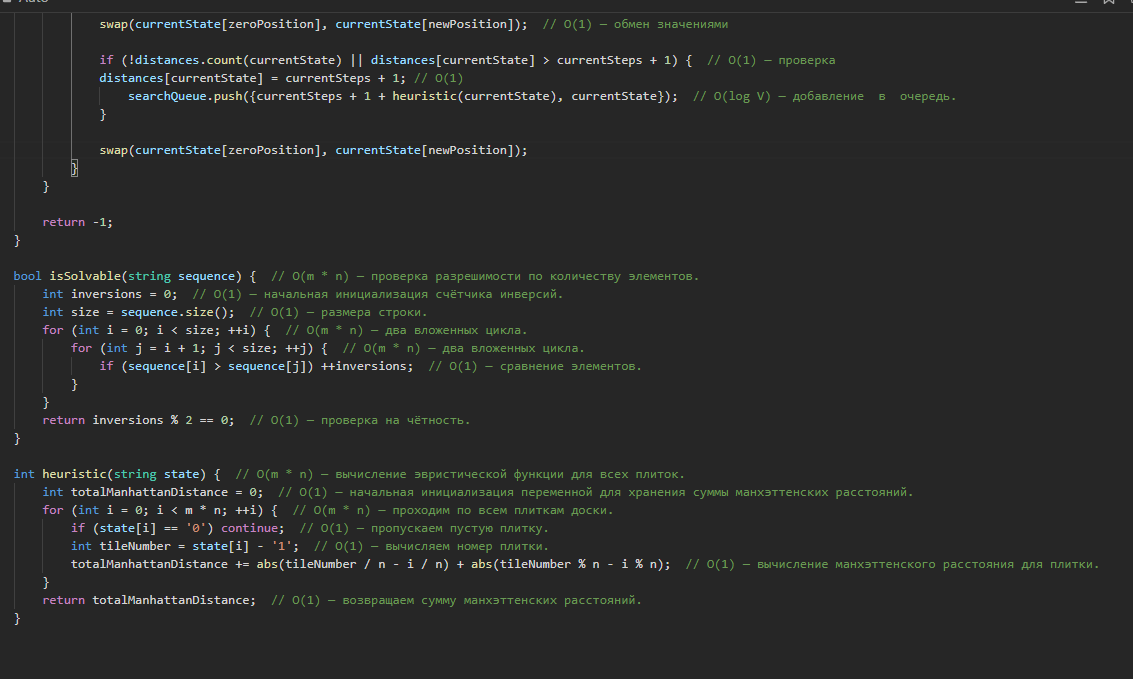
Если алгоритм находит целевое состояние, то возвращается количество шагов, необходимых для его достижения. Если путь не найден, то программа возвращает -1, что означает, что решение невозможно.

1. **Экспериментальная часть**

Подсчёт по памяти и асимптотики структур, а так же готовый код:







Массивы и строки (например, *startState, tmpSequence, goalState*):

Память: Эти структуры данных занимают память O(m \* n), так как хранят строку, длина которой зависит от количества элементов на доске (в данном случае 6 элементов для 2x3).Асимптотика: О(m \* n), так как мы работаем с каждым элементом доски.

Очередь с приоритетом (*priority\_queue*):Память: Очередь хранит состояния, и в худшем случае она будет хранить все возможные состояния доски. Это количество будет O(V), где V — количество возможных состояний.Асимптотика: О(log V) для каждой операции вставки и извлечения, где V — количество состояний.

Хэш-таблица (*unordered\_map*)Память: Хэш-таблица хранит состояние в качестве ключа и количество шагов в качестве значения. Это также требует O(V) памяти.Асимптотика: О(1) для операций вставки, извлечения и поиска.

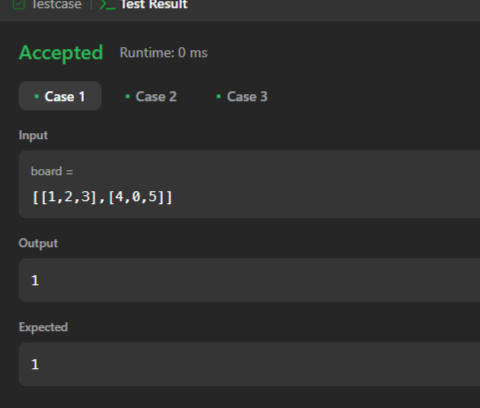
Вектор направлений (vector<int>)Память: Вектор содержит фиксированное количество направлений, что требует O(1) памяти. Асимптотика: О(1), так как операции над вектором выполняются с постоянной сложностью.

**Итог:**

Основные затраты по памяти происходят из-за хранения состояний в очереди с приоритетом и хэш-таблице. Это делает память O(V), где V — количество возможных состояний.

Основные операции, такие как добавление в очередь и проверка на наличие состояния в хэш-таблице, имеют асимптотику O(log V) для очереди и O(1) для хэш-таблицы.

**Пройденные тесты:**

****



**Вывод**

В этой задаче я использовала элементы динамического программирования, поскольку задача поиска кратчайшего пути в пазле 2x3 подразумевает работу с множеством промежуточных состояний. Каждое состояние можно рассматривать как подзадачу, решение которой может быть использовано для дальнейших шагов.

Суть динамического программирования заключается в том, чтобы избежать повторных вычислений, и это именно то, что я сделала, используя хэш-таблицу для хранения уже посещённых состояний. Когда мы встречаем уже обработанное состояние, мы не пересчитываем его заново, а просто используем сохранённый результат, что экономит время и ресурсы.

Вместо того, чтобы решать задачу «с нуля» на каждом шаге, я использую идею запоминания результатов — ключевая характеристика динамического программирования. Это позволяет мне оптимизировать процесс поиска кратчайшего пути и избежать лишних вычислений, что особенно важно в задачах с большими графами состояний.

Использование динамического подхода в сочетании с приоритетной очередью и эвристикой помогает найти решение более эффективно и минимизировать количество шагов до достижения целевого состояния.