Федеральное государственное бюджетное образовательное учреждение высшего образования «Нижегородский государственный архитектурно-строительный университет» (ННГАСУ)

*Факультет инженерно-экологических систем и сооружений*

*Кафедра информационных систем и технологий*

КУРСОВАЯ РАБОТА

по дисциплине: «Язык программирования Python»

На тему: «Алгоритмы поиска пути и структурное программирование»

Выполнил студент 1 курса гр. ИС-34 Миронов Е.С

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc137403601)

[Задачи 3](#_Toc137403602)

[Теоретическая часть. 4](#_Toc137403603)

[Особенности 4](#_Toc137403604)

[Свойство жадного выбора 4](#_Toc137403605)

[Случаи сбоя 5](#_Toc137403606)

[Типы 6](#_Toc137403607)

[Теория 6](#_Toc137403608)

[Алгоритм А\* 6](#_Toc137403609)

[Описание алгоритма А\* 7](#_Toc137403610)

[Свойства 8](#_Toc137403611)

[Реализация алгоритма 9](#_Toc137403612)

[Пример работы 10](#_Toc137403613)

[Заключение 11](#_Toc137403614)

[Список литературы 12](#_Toc137403615)

[Листинг программы 13](#_Toc137403616)

# Введение

Алгоритмы поиска пути и структурное программирование являются одной из важнейших задач в программировании. Жадный алгоритм – это метод решения задач, основанный на принципе выбора на каждом шаге локального оптимальному решению. Достоинства жадного алгоритма заключаются в быстром времени работы и способности работать с большими значениями.

**Цель работы**: исследование алгоритмов поиска пути и их применения в задачах проходящего пути в графе.

# Задачи

1. В папке \_3 запустить скрипт gen\_lab\_origin.py, подождать его работу несколько минут;
2. Полученный файл maze-for-u.txt переместить в свою папку;
3. Найти любой маршрут от начальной координаты аватара до ключа, используя жадный алгоритм.
4. Используя А\* со следующими параметрами (вес g(x), вес h(x), максимальная длина хранимого списка возможных шагов), найти оптимальный путь до ближайшего выхода;
5. Сохранить в файл 'maze-for-me-done.txt', в котором точка ключа будет указана как '\*', а сам маршрут построен точками к ключу и запятыми от него к выходу.

# Теоретическая часть.

**Жадный алгоритм** - это любой [алгоритм](https://translated.turbopages.org/proxy_u/en-ru.ru.092578d0-647b538a-4e922166-74722d776562/https/en.wikipedia.org/wiki/Algorithm), который следует [эвристике](https://translated.turbopages.org/proxy_u/en-ru.ru.092578d0-647b538a-4e922166-74722d776562/https/en.wikipedia.org/wiki/Heuristic_(computer_science)) решения проблемы, заключающейся в локально оптимальном выборе на каждом этапе. Во многих задачах жадная стратегия не дает оптимального решения, но жадная эвристика может дать локально оптимальные решения, которые приближаются к глобально оптимальному решению за разумный промежуток времени.

Например, жадной стратегией для [задачи коммивояжера](https://translated.turbopages.org/proxy_u/en-ru.ru.092578d0-647b538a-4e922166-74722d776562/https/en.wikipedia.org/wiki/Travelling_salesman_problem) (которая отличается высокой вычислительной сложностью) является следующая эвристика: "На каждом этапе путешествия посещайте ближайший не посещаемый город". Эта эвристика не направлена на поиск наилучшего решения, но она завершается за разумное количество шагов; поиск оптимального решения такой сложной задачи обычно требует неоправданно большого количества шагов. В математической оптимизации жадные алгоритмы оптимально решают комбинаторные задачи, обладающие свойствами [матроидов](https://translated.turbopages.org/proxy_u/en-ru.ru.092578d0-647b538a-4e922166-74722d776562/https/en.wikipedia.org/wiki/Matroid" \o "Матроид), и дают аппроксимации с постоянным коэффициентом к задачам оптимизации с субмодулярной структурой.

## Особенности

Жадные алгоритмы дают хорошие решения для некоторых [математических задач](https://translated.turbopages.org/proxy_u/en-ru.ru.092578d0-647b538a-4e922166-74722d776562/https/en.wikipedia.org/wiki/Mathematical_problem), но не для других. Большинство задач, для решения которых они работают, будут обладать двумя свойствами:

## Свойство жадного выбора

Мы можем сделать любой выбор, который кажется наилучшим в данный момент, а затем решить подзадачи, которые возникнут позже. Выбор, сделанный жадным алгоритмом, может зависеть от выборов, сделанных на данный момент, но не от будущих выборов или всех решений подзадачи. Он итеративно делает один жадный выбор за другим, сводя каждую заданную проблему к более мелкой. Другими словами, жадный алгоритм никогда не пересматривает свой выбор. В этом основное отличие от [динамического программирования](https://translated.turbopages.org/proxy_u/en-ru.ru.092578d0-647b538a-4e922166-74722d776562/https/en.wikipedia.org/wiki/Dynamic_programming), которое является исчерпывающим и гарантированно находит решение. После каждого этапа динамическое программирование принимает решения на основе всех решений, принятых на предыдущем этапе, и может пересмотреть алгоритмический путь предыдущего этапа к решению.

## Случаи сбоя

Жадные алгоритмы не могут найти оптимального решения для многих других задач и могут даже выдать *единственное наихудшее из возможных* решений. Одним из примеров является [проблема коммивояжера](https://translated.turbopages.org/proxy_u/en-ru.ru.092578d0-647b538a-4e922166-74722d776562/https/en.wikipedia.org/wiki/Travelling_salesman_problem), упомянутая выше: для каждого количества городов существует присвоение расстояний между городами, для которых эвристика ближайшего соседа создает уникальный наихудший возможный тур. Другие возможные примеры см. в разделе [Эффект горизонта](https://translated.turbopages.org/proxy_u/en-ru.ru.092578d0-647b538a-4e922166-74722d776562/https/en.wikipedia.org/wiki/Horizon_effect).

**Примеры того, как жадный алгоритм может не привести к достижению оптимального решения.**

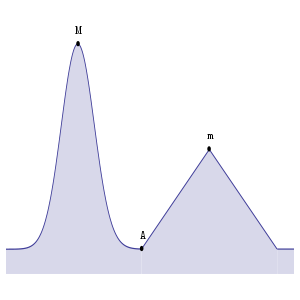
[](https://translated.turbopages.org/proxy_u/en-ru.ru.092578d0-647b538a-4e922166-74722d776562/https/en.wikipedia.org/wiki/File:Greedy_Glouton.svg)

Рисунок 1. Пример 1

Начиная с A, жадный алгоритм, который пытается найти максимум, следуя наибольшему наклону, найдет локальный максимум в точке "m", не обращая внимания на глобальный максимум в точке "M".

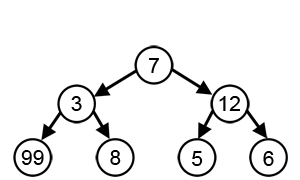
[](https://translated.turbopages.org/proxy_u/en-ru.ru.092578d0-647b538a-4e922166-74722d776562/https/en.wikipedia.org/wiki/File:Greedy-search-path-example.gif)

Рисунок 2. Пример 2

Чтобы получить наибольшую сумму, на каждом шаге жадный алгоритм будет выбирать то, что представляется оптимальным немедленным выбором, поэтому на втором шаге он выберет 12 вместо 3 и не достигнет наилучшего решения, которое содержит 99.

## Типы

Жадные алгоритмы можно охарактеризовать как "недальновидные", а также как "не подлежащие восстановлению". Они идеальны только для задач, имеющих "оптимальную подструктуру". Несмотря на это, для многих простых задач наиболее подходящие алгоритмы являются жадными. Важно, однако, отметить, что жадный алгоритм может использоваться в качестве алгоритма выбора для приоритизации вариантов в рамках поиска или алгоритма ветвления и привязки. Существует несколько вариаций жадного алгоритма:

* Чистые жадные алгоритмы
* Ортогональные жадные алгоритмы
* Смягченные жадные алгоритмы

## Теория

Жадные алгоритмы имеют долгую историю изучения в [комбинаторной оптимизации](https://translated.turbopages.org/proxy_u/en-ru.ru.8254f525-6480b160-3a4c5c45-74722d776562/https/en.wikipedia.org/wiki/Combinatorial_optimization) и [теоретической информатике](https://translated.turbopages.org/proxy_u/en-ru.ru.8254f525-6480b160-3a4c5c45-74722d776562/https/en.wikipedia.org/wiki/Theoretical_computer_science). Известно, что жадные эвристики приводят к неоптимальным результатам во многих задачах, и поэтому естественными вопросами являются:

* Для решения каких задач жадные алгоритмы работают оптимально?
* Для каких задач жадные алгоритмы гарантируют приблизительно оптимальное решение?
* Для каких задач гарантированно, что жадный алгоритм *не* приведет к оптимальному решению?

Существует большое количество литературы, отвечающей на эти вопросы для общих классов задач, таких как [матроиды](https://translated.turbopages.org/proxy_u/en-ru.ru.8254f525-6480b160-3a4c5c45-74722d776562/https/en.wikipedia.org/wiki/Matroid" \o "Матроид), а также для конкретных задач, таких как [покрытие множеств](https://translated.turbopages.org/proxy_u/en-ru.ru.8254f525-6480b160-3a4c5c45-74722d776562/https/en.wikipedia.org/wiki/Set_cover).

## Алгоритм А\*

**Алгоритм A\*** — в [информатике](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0) и [математике](https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0), [алгоритм](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC) поиска [по первому наилучшему совпадению](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B8%D1%81%D0%BA_%D0%BF%D0%BE_%D0%BF%D0%B5%D1%80%D0%B2%D0%BE%D0%BC%D1%83_%D0%BD%D0%B0%D0%B8%D0%BB%D1%83%D1%87%D1%88%D0%B5%D0%BC%D1%83_%D1%81%D0%BE%D0%B2%D0%BF%D0%B0%D0%B4%D0%B5%D0%BD%D0%B8%D1%8E) на [графе](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D1%84_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), который находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной).

Порядок обхода вершин определяется [**эвристической функцией**](https://ru.wikipedia.org/wiki/%D0%AD%D0%B2%D1%80%D0%B8%D1%81%D1%82%D0%B8%D0%BA%D0%B0) «расстояние + стоимость» (обычно обозначаемой как *f(x)*). Эта функция — сумма двух других: функции стоимости достижения рассматриваемой вершины (*x*) из начальной (обычно обозначается как *g(x)* и может быть как эвристической, так и нет), и функции эвристической оценки расстояния от рассматриваемой вершины к конечной (обозначается как *h(x)*).

Функция *h(x)* должна быть **допустимой эвристической оценкой**, то есть не должна переоценивать расстояния к целевой вершине. Например, для задачи маршрутизации *h(x)* может представлять собой расстояние до цели по прямой линии, так как это физически наименьшее возможное расстояние между двумя точками.

Этот алгоритм был впервые описан в [1968 году](https://ru.wikipedia.org/wiki/1968_%D0%B3%D0%BE%D0%B4) [Питером Хартом](https://ru.wikipedia.org/w/index.php?title=%D0%A5%D0%B0%D1%80%D1%82,_%D0%9F%D0%B8%D1%82%D0%B5%D1%80&action=edit&redlink=1), [Нильсом Нильсоном](https://ru.wikipedia.org/w/index.php?title=%D0%9D%D0%B8%D0%BB%D1%8C%D1%81%D0%BE%D0%BD,_%D0%9D%D0%B8%D0%BB%D1%8C%D1%81&action=edit&redlink=1) и [Бертрамом Рафаэлем](https://ru.wikipedia.org/w/index.php?title=%D0%A0%D0%B0%D1%84%D0%B0%D1%8D%D0%BB%D1%8C,_%D0%91%D0%B5%D1%80%D1%82%D1%80%D0%B0%D0%BC&action=edit&redlink=1). Это по сути было расширение [алгоритма Дейкстры](https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D1%8B), созданного в 1959 году. Новый алгоритм достигал более высокой производительности (по времени) с помощью эвристики. В их работе он упоминается как «алгоритм A». Но так как он вычисляет лучший маршрут для заданной эвристики, он был назван A\*.

## Описание алгоритма А\*

A\* пошагово просматривает все пути, ведущие от начальной вершины в конечную, пока не найдёт минимальный. Как и все [информированные алгоритмы поиска](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA), он просматривает сначала те маршруты, которые «кажутся» ведущими к цели. От [жадного алгоритма](https://ru.wikipedia.org/wiki/%D0%96%D0%B0%D0%B4%D0%BD%D1%8B%D0%B9_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC), который тоже является алгоритмом поиска по первому лучшему совпадению, его отличает то, что при выборе вершины он учитывает, помимо прочего, *весь* пройденный до неё путь. Составляющая *g(x)* — это стоимость пути от начальной вершины, а не от предыдущей, как в жадном алгоритме.

В начале работы просматриваются узлы, смежные с начальным; выбирается тот из них, который имеет минимальное значение *f(x)*, после чего этот узел раскрывается. На каждом этапе алгоритм оперирует с множеством путей из начальной точки до всех ещё не раскрытых (листовых) вершин графа — множеством частных решений, — которое размещается в [очереди с приоритетом](https://ru.wikipedia.org/wiki/%D0%9E%D1%87%D0%B5%D1%80%D0%B5%D0%B4%D1%8C_%D1%81_%D0%BF%D1%80%D0%B8%D0%BE%D1%80%D0%B8%D1%82%D0%B5%D1%82%D0%BE%D0%BC_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). Приоритет пути определяется по значению *f(x) = g(x) + h(x)*. Алгоритм продолжает свою работу до тех пор, пока значение *f(x)* целевой вершины не окажется меньшим, чем любое значение в очереди, либо пока всё дерево не будет просмотрено. Из множества решений выбирается решение с наименьшей стоимостью.

Чем меньше эвристика *h(x)*, тем больше приоритет, поэтому для реализации очереди можно использовать [сортирующие деревья](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%80%D1%82%D0%B8%D1%80%D1%83%D1%8E%D1%89%D0%B5%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE).

## Свойства

Как и алгоритм [поиска в ширину](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B8%D1%81%D0%BA_%D0%B2_%D1%88%D0%B8%D1%80%D0%B8%D0%BD%D1%83), A\* является **полным** в том смысле, что он всегда находит решение, если таковое существует.

Если эвристическая функция *h* допустима, то есть никогда *не переоценивает* действительную минимальную стоимость достижения цели, то A\* сам является допустимым (или **оптимальным**), также при условии, что мы не отсекаем пройденные вершины. Если же мы это делаем, то для оптимальности алгоритма требуется, чтобы *h(x)* была ещё и **монотонной**, или **преемственной** эвристикой. Свойство монотонности означает, что если существуют пути **A—B—C** и **A—C** (не обязательно через **B**), то оценка стоимости пути от **A** до **C** должна быть меньше либо равна сумме оценок путей **A—B** и **B—C**. (Монотонность также известна как [неравенство треугольника](https://ru.wikipedia.org/wiki/%D0%9D%D0%B5%D1%80%D0%B0%D0%B2%D0%B5%D0%BD%D1%81%D1%82%D0%B2%D0%BE_%D1%82%D1%80%D0%B5%D1%83%D0%B3%D0%BE%D0%BB%D1%8C%D0%BD%D0%B8%D0%BA%D0%B0): одна сторона [треугольника](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B5%D1%83%D0%B3%D0%BE%D0%BB%D1%8C%D0%BD%D0%B8%D0%BA) не может быть длиннее, чем сумма двух других сторон.)

Математически, для всех путей *x*, *y* (где *y* — потомок *x*) выполняется:

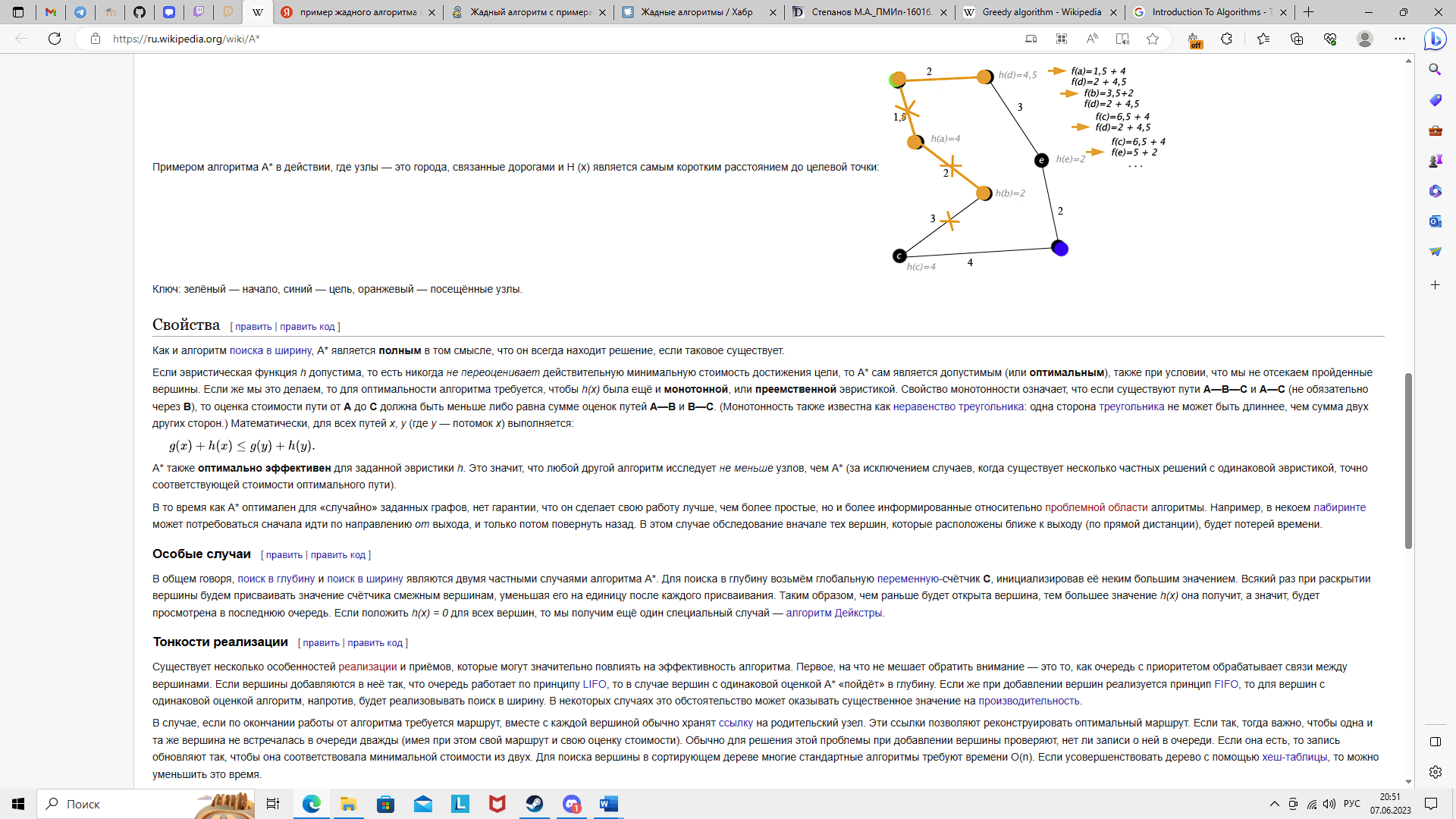


Рисунок 3. Математическая формула путей

�(�)+ℎ(�)≤�(�)+ℎ(�).

A\* также **оптимально эффективен** для заданной эвристики *h*. Это значит, что любой другой алгоритм исследует *не меньше* узлов, чем A\* (за исключением случаев, когда существует несколько частных решений с одинаковой эвристикой, точно соответствующей стоимости оптимального пути).

В то время как A\* оптимален для «случайно» заданных графов, нет гарантии, что он сделает свою работу лучше, чем более простые, но и более информированные относительно [проблемной области](https://ru.wikipedia.org/w/index.php?title=%D0%9F%D1%80%D0%BE%D0%B1%D0%BB%D0%B5%D0%BC%D0%BD%D0%B0%D1%8F_%D0%BE%D0%B1%D0%BB%D0%B0%D1%81%D1%82%D1%8C&action=edit&redlink=1) алгоритмы. Например, в некоем [лабиринте](https://ru.wikipedia.org/wiki/%D0%9B%D0%B0%D0%B1%D0%B8%D1%80%D0%B8%D0%BD%D1%82) может потребоваться сначала идти по направлению *от* выхода, и только потом повернуть назад. В этом случае обследование вначале тех вершин, которые расположены ближе к выходу (по прямой дистанции), будет потерей времени.

# Реализация алгоритма

Жадный алгоритм используется для нахождения наилучшего варианта на момент выбора.

**Алгоритм:**

def Gadniy\_algoritm(maze, start, end):  
 #Создаем список соседних клеток  
 neighbors=[(0,1),(0,-1),(1,0),(-1,0)]  
 #Создаем список посещенных клеток  
 visited=set()  
 #создадим очередь для хранения пути  
 queue = [[start]]  
 #пока очередь не пуста  
 while queue:  
 path = queue.pop(0) #достаем 1 путь из очереди  
 current = path[-1] #берем последнюю точку пути  
 if current not in visited:  
 visited.add(current)  
 for neighbor in neighbors: #для каждого соседа текущей точки  
 #находим координаты точки  
 x = current[0] + neighbor[0]  
 y = current[1] + neighbor[1]  
 if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] != "#":#Если сосед находится в пределах лабиринта и не стена  
 new\_path = path+[(x,y)] #создаем новый путь до соседа  
 # если сосед равен конечной точку то возвращаем путь  
 if(x, y) == end:  
 return new\_path  
 else: # иначе добавляем новый путь в очередь  
 queue.append(new\_path)  
 return[]# Если не удалось найти путь до конечной точки, возвращаем пустой список

В программе жадный алгоритм был использован дважды:

1. Для нахождения кротчайшего пути к ключу.
2. Для нахождения от ключа до выхода.

# Пример работы

1. Используем программу «gen\_lab\_origin» для создания файла «maze-for-u» и получаем лабиринт:

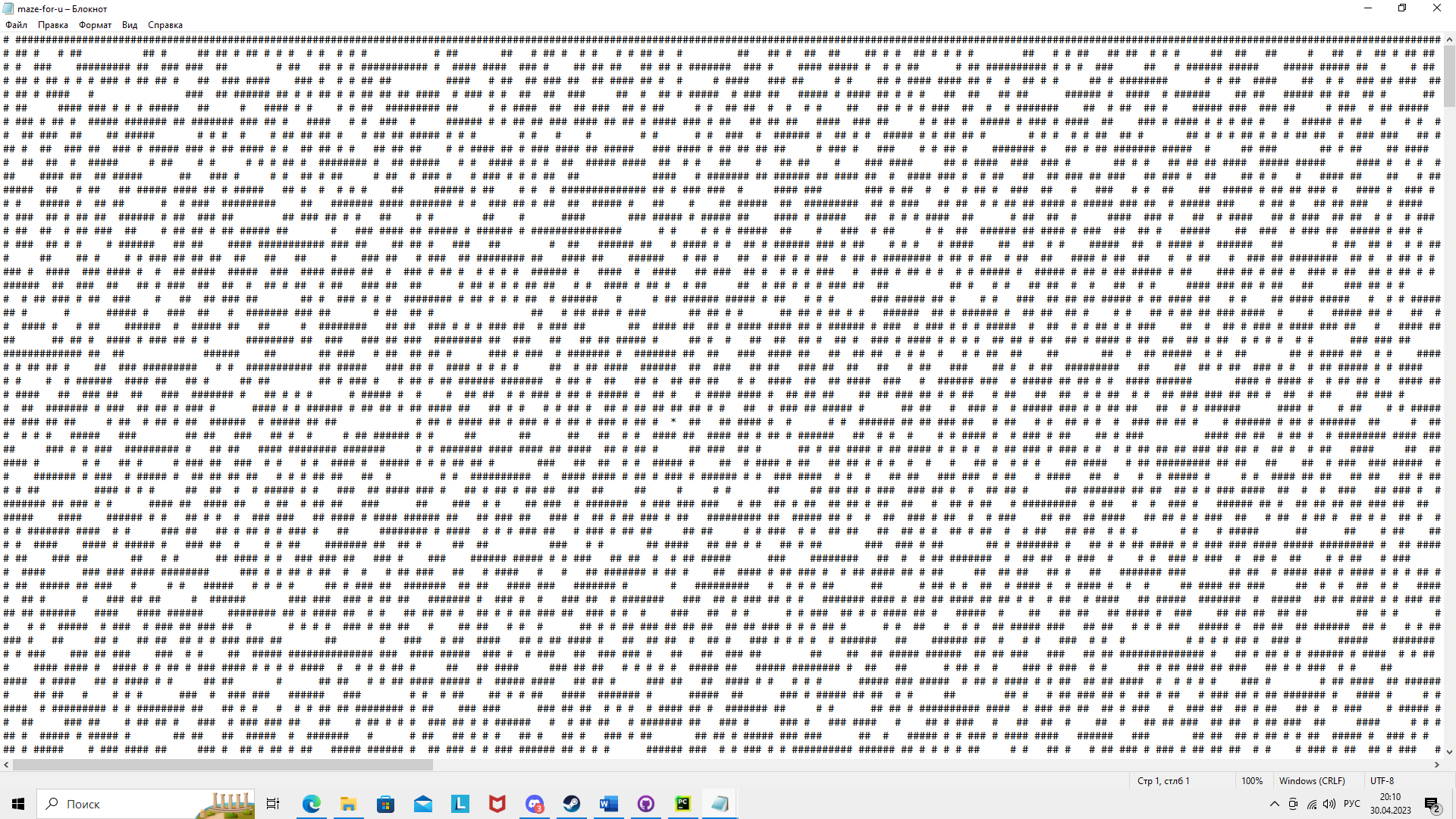


Рисунок 4. Лабиринт

1. Используем программу под названием «laba 3» после ее запуска появляется текстовый файл «maze-for-me-done»
2. Открываем текстовый файл и получаем результат:

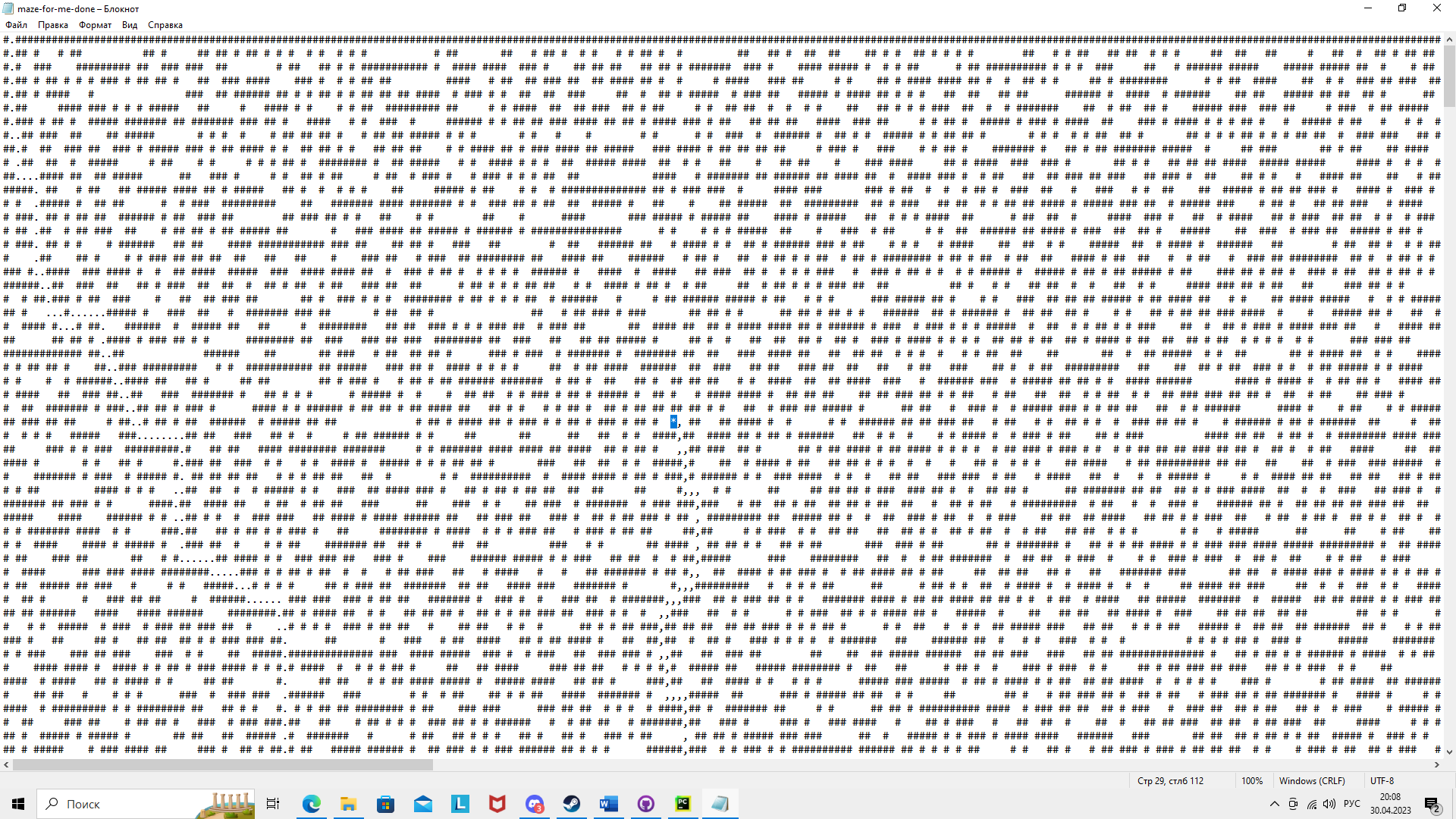


Рисунок 5. Путь в лабиринте

# Заключение

В ходе проделанной работы была создана программа для нахождения ключа в лабиринте и нахождения выхода. Жадные алгоритмы оказались эффективным методом решения задач на графах, но требуют тщательного выбора стратегии выбора локально оптимального решения на каждом шаге. Кроме того, не всегда жадный алгоритм приводит к глобально оптимальному решению, поэтому необходимо учитывать особенности конкретной задачи при выборе метода ее решения.

# Список литературы

1. Скиена С. Стивен Алгоритмы. Руководство по разработке. 3-е изд / Скиена С. Стивен. – СПб.: Springer, 2022. – 848 с. – Текст:  
   непосредственный.
2. Станковец А.В. АЛГОРИТМЫ НА ГРАФАХ / Станковец А.В. – Текст: электронный // M[ODERN SCIENCE](https://www.elibrary.ru/contents.asp?id=44150018). — 2020. — №10-2. — С.532-536. — EDN: [pdmxpx](https://www.elibrary.ru/pdmxpx)
3. Книга «Парадигмы алгоритмического проектирования(жадные алгоритмы, разделяй и властвуй и динамическое программирование)- [Бесплатная электронная книга о жадных алгоритмах, «разделяй и властвуй» и динамическом программировании - DEV Community](https://dev.to/brandonskerritt/a-free-ebook-on-greedy-algorithms-divide-conquer-and-dynamic-programming-1712)
4. Жадный алгоритм [Электронный ресурс]. URL: [Greedy algorithm - Wikipedia (turbopages.org)](https://translated.turbopages.org/proxy_u/en-ru.ru.be023311-6480bef0-1fee2eda-74722d776562/https/en.wikipedia.org/wiki/Greedy_algorithm)
5. Алгоритм А\* [Электронный ресурс]. URL: [A\* — Википедия (wikipedia.org)](https://ru.wikipedia.org/wiki/A*)
6. Преимущества и недостатки алгоритмов [Электронный ресурс]. URL: [Степанов М.А.\_ПМИп-1601б.pdf (tltsu.ru)](https://dspace.tltsu.ru/bitstream/123456789/12342/1/%D0%A1%D1%82%D0%B5%D0%BF%D0%B0%D0%BD%D0%BE%D0%B2%20%D0%9C.%D0%90._%D0%9F%D0%9C%D0%98%D0%BF-1601%D0%B1.pdf)
7. Пример жадного алгоритма [Электронный ресурс]. URL: [Жадный алгоритм с примерами на Python (pythonist.ru)](https://pythonist.ru/zhadnyj-algoritm-s-primerami-na-python/?ysclid=lilztibavb69493190)

**Приложение 1**

# Листинг программы

#функция для преобразования лабиринта

def readmaze(file\_name):

with open(file\_name) as f:

maze = [list(line.strip()) for line in f]#создаем список который содержит строки текстового файла

return maze

#поиск точек начало и конца

def tochky\_start\_end(maze):

start = (0, maze[0].index(" ")) #Ищем вход в первой линии

end = (len(maze)-1, maze[-1].index(" ")) #Ищем выход в последней

return start, end

#поиск ключа

def key\_tochka(maze):

for i, g in enumerate(maze): #мы ищем координаты ключа(i,g.index("\*"))

if "\*" in g:

print(i,g.index("\*"))

return(i, g.index("\*"))

#Жадный алгоритм для поиска пути в лабиринте

def Gadniy\_algoritm(maze, start, end):

#Создаем список соседних клеток

neighbors=[(0,1),(0,-1),(1,0),(-1,0)]

#Создаем список посещенных клеток

visited=set()

#создадим очередь для хранения пути

queue = [[start]]

#пока очередь не пуста

while queue:

path = queue.pop(0) #достаем 1 путь из очереди

current = path[-1] #берем последнюю точку пути

if current not in visited:

visited.add(current)

for neighbor in neighbors: #для каждого соседа текущей точки

#находим координаты точки

x = current[0] + neighbor[0]

y = current[1] + neighbor[1]

if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] != "#":#Если сосед находится в пределах лабиринта и не стена

new\_path = path+[(x,y)] #создаем новый путь до соседа

# если сосед равен конечной точку то возвращаем путь

if(x, y) == end:

return new\_path

else: # иначе добавляем новый путь в очередь

queue.append(new\_path)

return[]# Если не удалось найти путь до конечной точки, возвращаем пустой список

def new\_maze(maze, path, mark): #обьявляем лабиринт и помечаем путь

for cord in path:

x, y = cord

maze[x][y] = mark

return maze

def new\_file(maze, file\_name): #запись лабиринта в отдельный файл

with open(file\_name, "w") as f:

for g in maze:

f.write("".join(g)+"\n")

maze = readmaze("maze-for-u.txt")

start, end = tochky\_start\_end(maze)

key = key\_tochka(maze)

path1 = Gadniy\_algoritm(maze, start, key)

path2 = Gadniy\_algoritm(maze, key, end)

maze = new\_maze(maze, path1, ".")

maze = new\_maze(maze, path2, ",")

x, y = key

maze[x][y] = "\*"

new\_file(maze, "maze-for-me-done.txt")