

</ Математические  
алгоритмы поиска  
кратчайшего пути  
в программировании />

} /> [

Created by

Жаров Игорь  
Семенов Данил

# </ Введение

/> \*\*

## Поиск кратчайшего пути

Это важная задача в области математического программирования. Она заключается в нахождении самого короткого (или наиболее выгодного) маршрута в графе, который представляет собой набор объектов (вершин), соединённых линиями (рёбрами).

} /> [

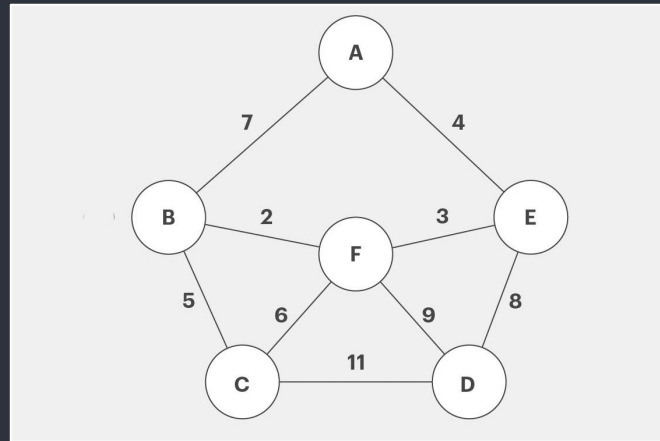
от маршрутизации грузов в логистике до построения маршрутов в GPS-навигаторах. В этом докладе мы рассмотрим два популярных алгоритма, которые решают эту задачу: алгоритм Дейкстры и алгоритм A-стар (A\*). Мы обсудим, как они работают, в каких ситуациях их лучше использовать и приведем примеры их применения.

# </ Что такое граф?

Также стоит отметить, что вообще такое граф и для чего он нужен.

**Граф** – это математическая структура, которая состоит из вершин (узлов) и рёбер (связей) между ними. Ребра могут иметь направление, а также веса – числа, которые обозначают силу связей с вершинами. Графы используются для представления объектов, их взаимосвязей и отношений между ними. **Например**, в виде графа можно представить социальную сеть, где вершины – это пользователи, дружеские связи между ними – ребра, а веса – это, частота обмена сообщениями.

# Граф>



&lt;/&gt;

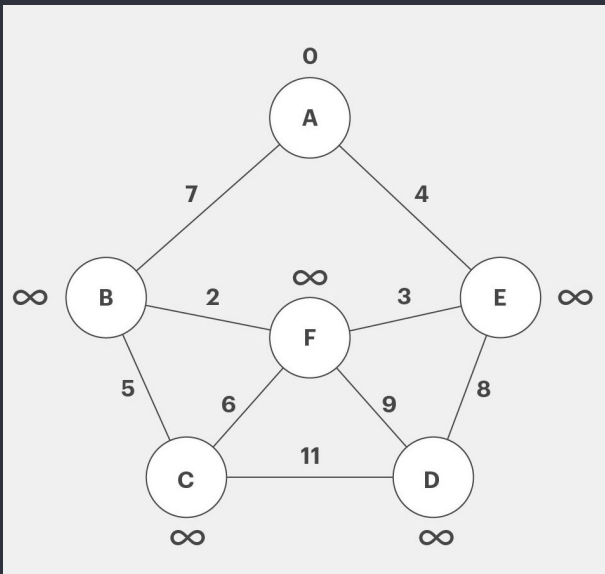
# Алгоритм Дейкстры

01

} /&gt; [

1 0 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 1 0 1

# </ Принципы алгоритма



## Стоит отметить

Это один из самых известных способов нахождения кратчайшего пути в графе с неотрицательными весами ребер.

## Начало работы

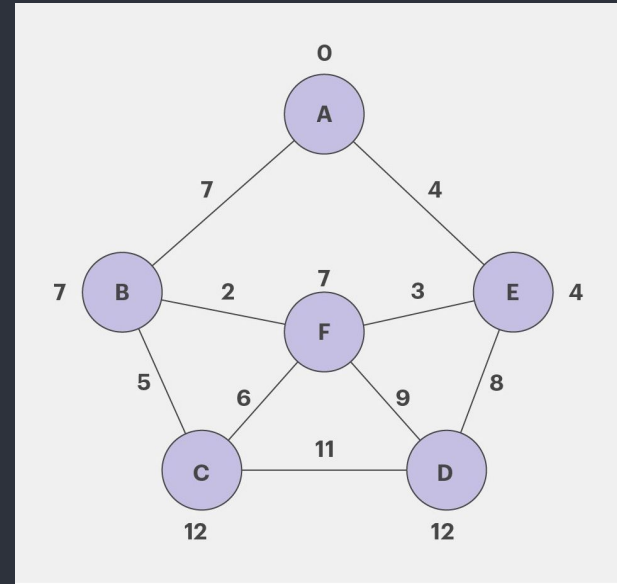
Сначала алгоритм выбирает начальную вершину и присваивает ей значение расстояния 0, так как путь до самой себя нулевой. Остальным вершинам назначается значение бесконечности, поскольку пока не известно, насколько далеко они находятся.

Затем мы начинаем проверять соседние вершины текущей вершины. Если новый найденный путь к соседу короче, чем известное расстояние, мы обновляем значение расстояния для этого соседа.

Мы продолжаем выбирать вершины с наименьшим расстоянием и обновлять соседей, пока не пройдем все вершины графа. В результате мы получим кратчайшие расстояния от начальной вершины до всех остальных.

Остается  
построить  
маршрут до точек

} /> [



# </ Особенности использования алгоритма Дейкстры

## Ограничения

Во-первых, он работает только со взвешенными графами — то есть такими, где веса между рёбрами известны заранее. А во-вторых, эти расстояния должны быть неотрицательными.

## Особенности

Алгоритм Дейкстры прост в реализации и очень эффективен для небольших графов. Однако алгоритм может работать медленно на больших графах, если не оптимизирован, например, с использованием приоритетной очереди.





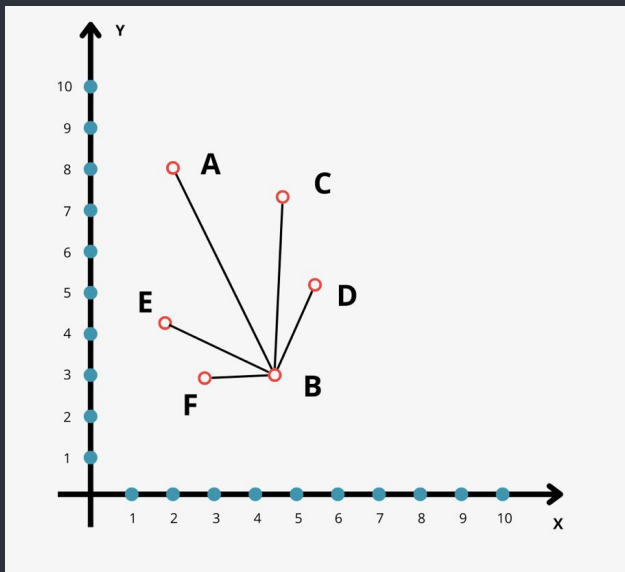
# Алгоритм А-стар

02





# </ Принципы алгоритма



Это более продвинутый алгоритм, который сочетает в себе идеи из алгоритма Дейкстры и эвристические подходы. Этот алгоритм не только ищет кратчайшие пути, но и использует оценки, чтобы ускорить процесс поиска.

## Начало работы

Так же как и алгоритм Дейкстры, A\* ищет расстояние от начальной точки до конечной. Но, в отличие от последнего, он учитывает не только расстояние от текущей точки до начальной, но и эвристическую оценку этого расстояния.

# </ Особенности использования алгоритма А-стар

## Особенности

Алгоритм А-стар обладает гибкостью благодаря использованию различных эвристик, которые могут значительно ускорить поиск. Однако выбор неправильной эвристики может привести к потере эффективности или неверным результатам. Важно понимать, что для достижения наилучших результатов эвристика должна быть как можно ближе к реальной стоимости пути.



# Сравнение алгоритмов



Алгоритм Дейкстры используется для нахождения кратчайшего пути от начальной вершины графа ко всем остальным. Алгоритм A\* используется для нахождения кратчайшего пути от начальной вершины к заданной, учитывая эвристическую оценку расстояния.

Алгоритм Дейкстры выбирает вершины с наименьшим расстоянием, а A\* использует эвристику, что повышает его эффективность для больших графов.

Алгоритм Дейкстры медленен для больших графов, так как исследует все вершины. A\* быстрее благодаря эвристике, сокращающей количество рассматриваемых вершин.



# Применение Алгоритмов

03



## </ На предприятиях

Оба алгоритма находят широкое применение в различных отраслях. Например, в логистике они используются для оптимизации маршрутов доставки товаров, что помогает сократить время и снизить затраты. В робототехнике алгоритмы помогают роботам планировать свои движения, избегая препятствий и находя наилучший маршрут к цели. Кроме того, алгоритмы используются в видеоиграх для управления перемещением персонажей, что позволяет создавать более реалистичное поведение.



# </ В повседневной жизни



В реальной жизни алгоритмы поиска кратчайшего пути применяются во множестве ситуаций. Например, в GPS-навигаторах они помогают пользователям находить самые быстрые маршруты до заданной точки. Алгоритмы также могут использоваться для оптимизации дорожного движения в городах, чтобы уменьшить пробки и сократить время в пути. В сетях передачи данных алгоритмы позволяют находить оптимальные маршруты для передачи информации, что улучшает скорость и эффективность.

# </ Примеры использования />

} /> [

Разработанные на python

# </ Задача 1

Нахождение кратчайших расстояний  
между городами

```
import heapq

def dijkstra(graph, start):
    distances = {vertex: float('inf') for vertex in graph}
    distances[start], priority_queue = 0, [(0, start)]

    while priority_queue:
        current_distance, current_vertex =
heapq.heappop(priority_queue)
        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances
```



# </ Задача 1

Алгоритм Дейкстры показал свою эффективность при работе с небольшими графами. Он успешно вычислил кратчайшие расстояния, и его реализация оказалась простой и понятной.

```
Кратчайшие расстояния до городов:  
До города Aksay: 0  
До города Birmingham: 4  
До города Cambridge: 7  
До города Dover: 6  
До города Edinburgh: 7  
До города Folkestone: 10
```

```
def main():  
    # Пример использования:  
    graph = {  
        'Aksay': {'Birmingham': 4, 'Cambridge': 7},  
        'Birmingham': {'Aksay': 4, 'Dover': 2, 'Edinburgh': 8},  
        'Cambridge': {'Aksay': 7, 'Dover': 2, 'Edinburgh': 5},  
        'Dover': {'Birmingham': 2, 'Cambridge': 2, 'Edinburgh': 1, 'Folkestone': 4},  
        'Edinburgh': {'Cambridge': 5, 'Dover': 1, 'Folkestone': 11},  
        'Folkestone': {'Birmingham': 8, 'Dover': 4, 'Edinburgh': 11}  
    }  
  
    result = dijkstra(graph, 'Aksay')  
    print("Кратчайшие расстояния до городов: ")  
    for vertex, distance in result.items():  
        print(f"До города {vertex}: {distance}")  
  
if __name__ == "__main__":  
    main()
```

# </ Задача 2

Вычислить наиболее короткий путь от одного здания, до другого.

**Манхэттенское расстояние** (также известное как расстояние городских кварталов) — это расстояние, которое определяется наименьшим числом кварталов, которые придётся пройти, двигаясь из одной точки (перекрёстка) в другую.

```
import heapq

# Функция для вычисления манхэттенского расстояния между двумя точками
def manhattan_distance(point1, point2):
    return abs(point1[0] - point2[0]) + abs(point1[1] - point2[1])

def astar(grid, start, goal):
    cost = {start: 0}
    came_from = {start: None}
    priority_queue = [(0, start)]

    n = len(grid)
    m = len(grid[0])

    while priority_queue:
        current_cost, current_point = heapq.heappop(priority_queue)

        if current_point == goal:
            break

        for neighbor in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
            x, y = current_point[0] + neighbor[0], current_point[1] + neighbor[1]
            grid_x = x
            grid_y = n - y - 1
```

# </ Задача 2

```
if 0 <= grid_x < m and 0 <= grid_y < n and grid[grid_y][grid_x] == 0:
    new_cost = cost[current_point] + 1
    if (x, y) not in cost or new_cost < cost[(x, y)]:
        cost[(x, y)] = new_cost
        priority = new_cost + manhattan_distance((x, y), goal)
        heapq.heappush(priority_queue, (priority, (x, y)))
        came_from[(x, y)] = current_point
```

```
path = []
current = goal
while current:
    path.append(current)
    current = came_from[current]
path.reverse()
return path
```

```
def main():
    grid = [
        [0, 0, 0, 0, 0],
        [0, 1, 1, 0, 0],
        [0, 0, 0, 1, 0],
        [0, 1, 0, 0, 0],
        [0, 0, 0, 0, 0]
    ]

    start, goal = (0, 0), (4, 4)
    result_path = astar(grid, start, goal)

    print("Путь от А до В:")
    for point in result_path:
        print(point)

if __name__ == '__main__':
    main()
```

Путь от А до В:

(0, 0)  
(0, 1)  
(0, 2)  
(0, 3)  
(0, 4)  
(1, 4)  
(2, 4)  
(3, 4)  
(4, 4)