Отчет по производительности алгоритмов поиска кратчайшего пути в графе.

1. Постановка задачи

Сравнение следующих алгоритмов поиска кратчайшего пути в графе по производительности и использованию памяти.

1) Алгоритм Дейкстры
2) Алгоритм Форда-Беллмана
3) Алгортим Левита
4) Алгоритм поиска кратчайшего пути проходящего через заданные вершины

2. Параметры вычислительного узла.

Процессор - Intel® Core™ i7-7820HQ CPU @ 2.90GHz × 8
ОС - Ubuntu 18.04.4 LTS 64-bit
Оперативная память - 15,5 GiB

3. Краткое описание тестируемых алгоритмов.

Алгоритм Дейкстры - представляет собой n (где n — количество вершин) итераций, на каждой из которых выбирается непомеченная вершина с наименьшим расстоянием до помеченных, эта вершина помечается, и затем просматриваются все рёбра, исходящие из данной вершины, и вдоль каждого ребра делается попытка улучшить значение на другом конце ребра.
Алгоритм был реализован с помощью очереди с приоритетом.

Алгоритм Форда-Беллмана представляет из себя не болле n фаз (где n — количество вершин). На каждой фазе просматриваются все рёбра графа, и алгоритм пытается произвести релаксацию вдоль каждого ребра (a, b) стоимости c. После каждой фазы будет найдено кратчайшие расстояния состоящие не более чем из i ребер, где i — номер фазы.

Алгоритм Левита Пусть di— текущая длина кратчайшего пути до вершины i. Изначально, все элементы d, кроме s-го равны бесконечности; d[s]=0.

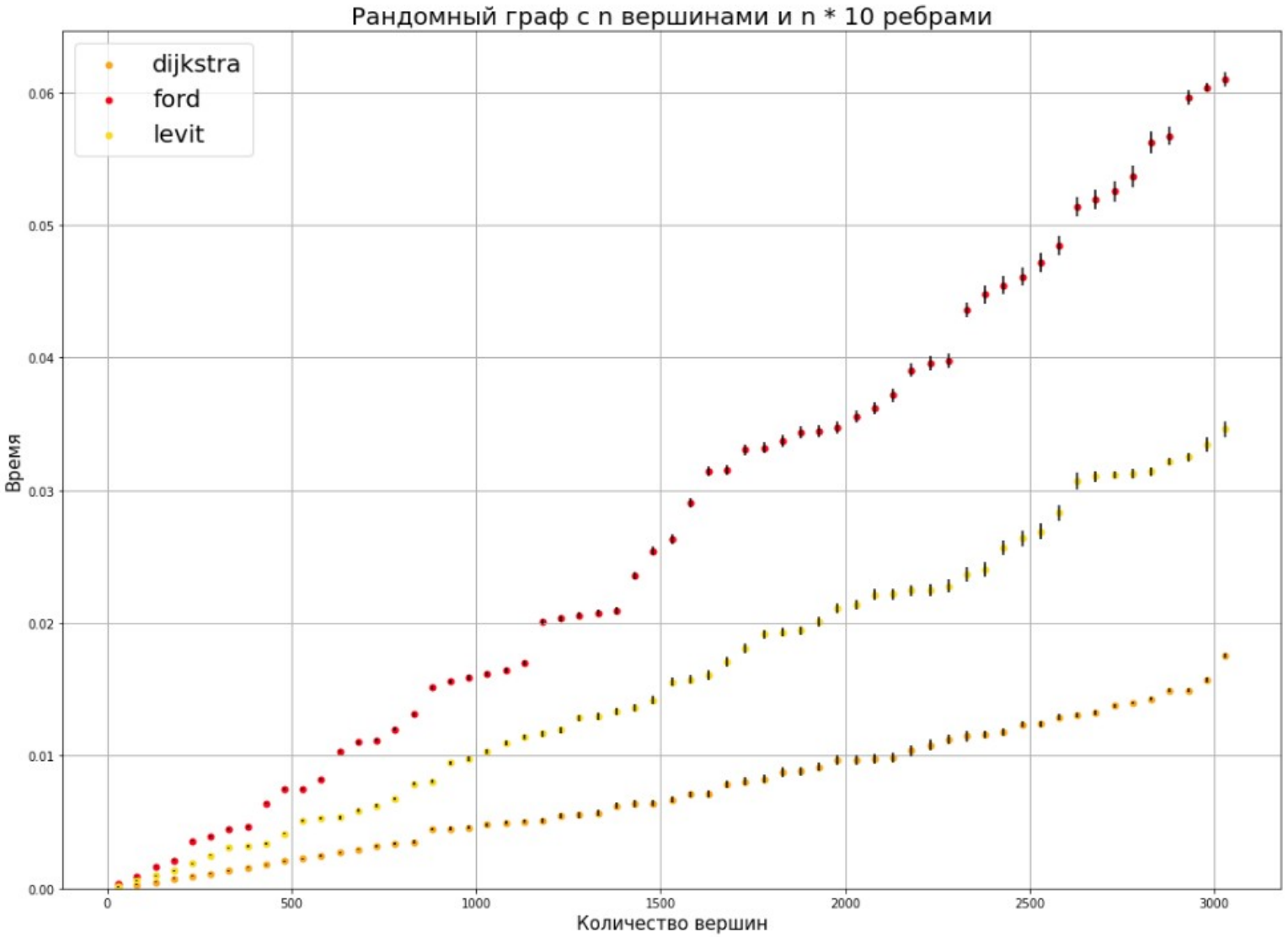
Разделим вершины на три множества:

● M0 — вершины, расстояние до которых уже вычислено (возможно, не окончательно),
● M1— вершины, расстояние до которых вычисляется. Это множество в свою очередь делится на две очереди:
1. M′1— основная очередь,
2. M′′1— срочная очередь;
● M2— вершины, расстояние до которых еще не вычислено.
Изначально все вершины, кроме s помещаются в множество M2. Вершина s помещается в множество M1 (в любую из очередей).
Шаг алгоритма**:** выбирается вершина u из M1. Если очередь M′′1 не пуста, то вершина берется из нее, иначе из M′1. Для каждого ребра uv∈E возможны три случая:
●v∈M2, то v переводится в конец очереди M′1. При этом dv←du+wuv (производится релаксация ребра uv),
●v∈M1, то происходит релаксация ребра uv,
●v∈M0. Если при этом dv>du+wuv, то происходит релаксация ребра uv и вершина v помещается в M′′1; иначе ничего не делаем.
В конце шага помещаем вершину u в множество M0.
Алгоритм заканчивает работу, когда множество M1 становится пустым.

Алгоритм поиска кратчайшего пути проходящего через заданные вершины
1) С помощью алгоритма Флойда-Уоршелла находим все попарные расстояние между вершинами.
2) Строим новый граф — где вершины это вершины через которые должен пролегать путь, а ребра — расстояние между данными вершинами.
3) С помощью алгоритма Прима находим минимальный остов, сумма ребер которого будет являться ответом.

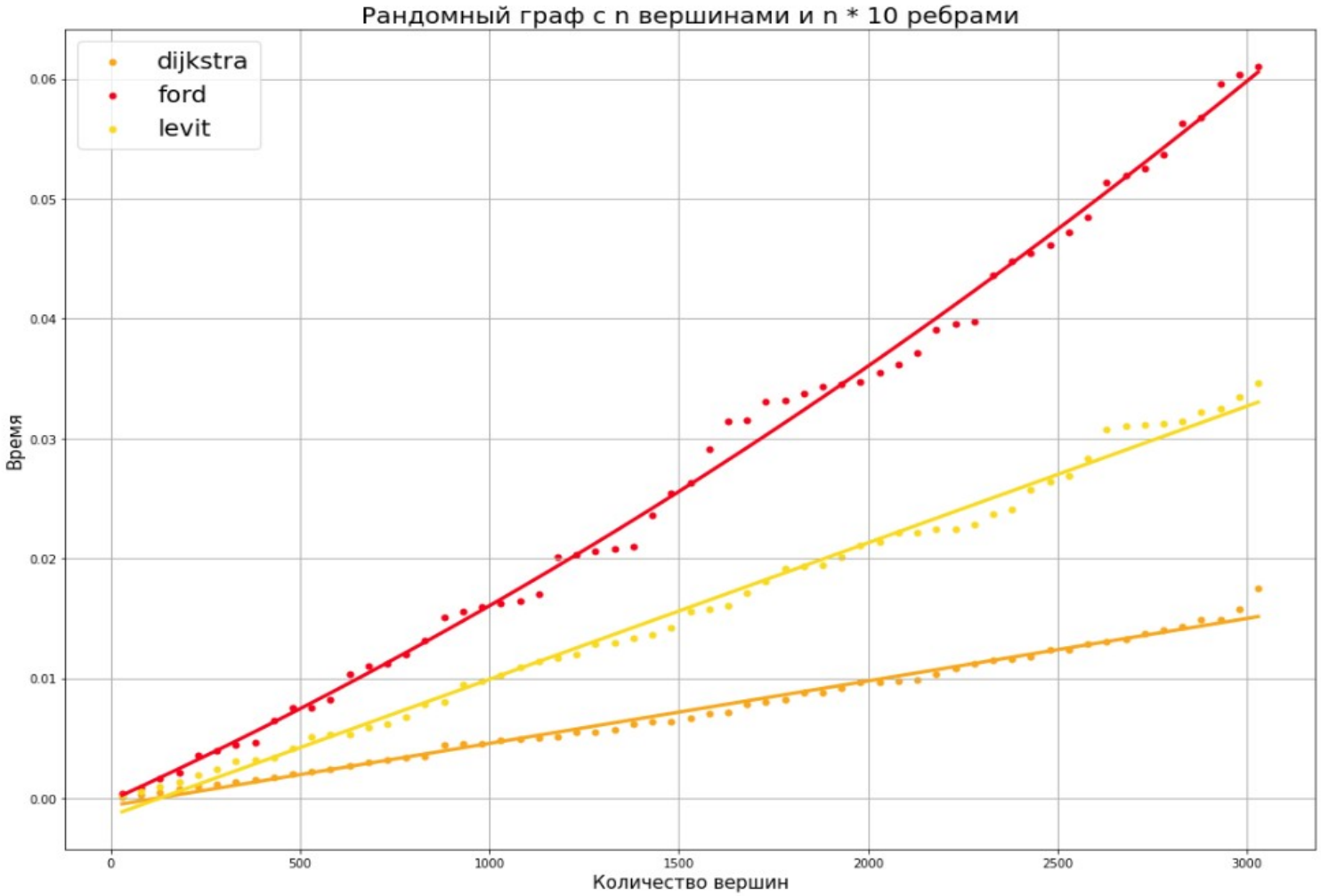
4. Результаты измерений в виде графиков.

1) Рандомный граф с n вершинами и n * 10 ребрами.

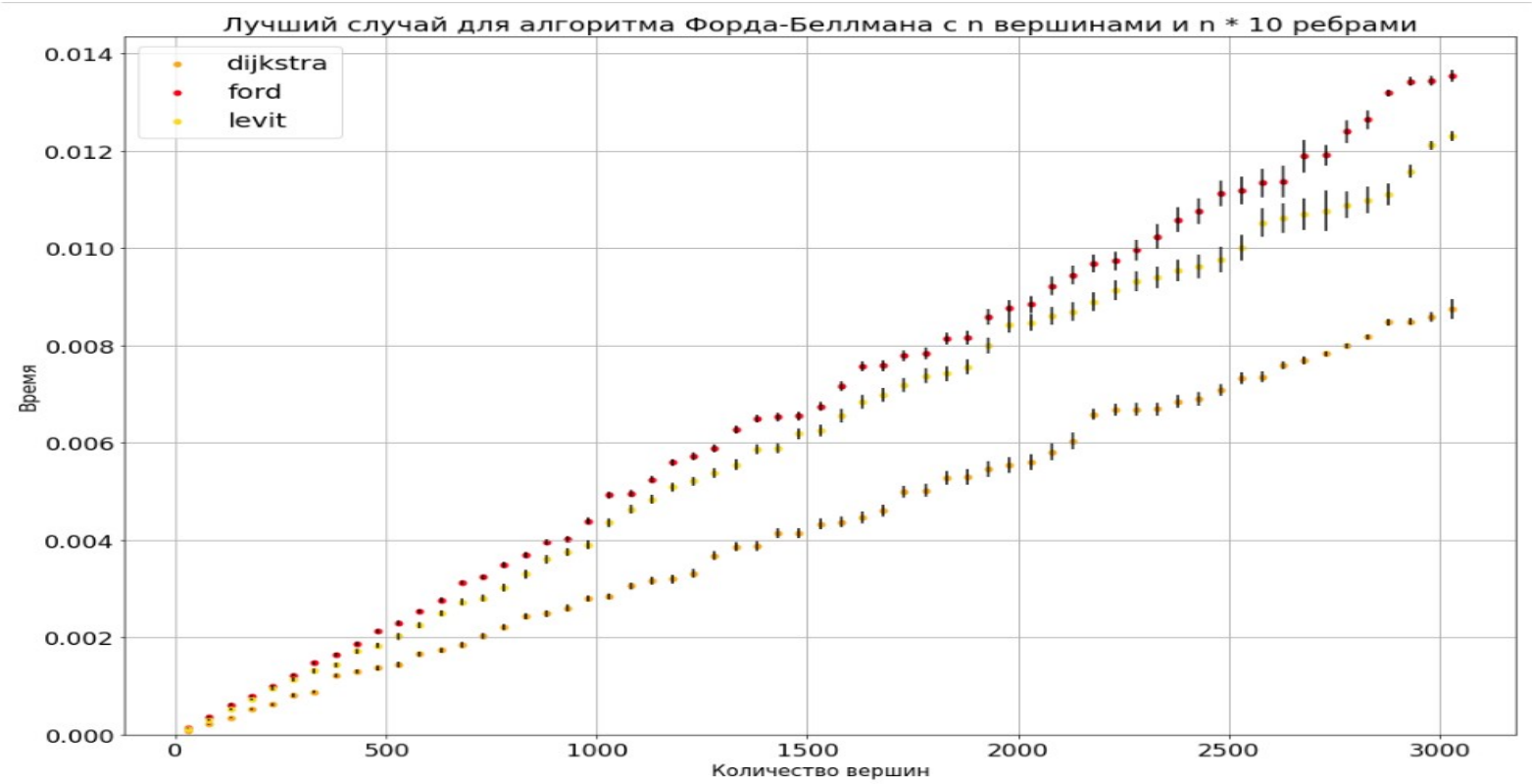


Выполняем приближение полученных данных следующими функциями:
Алгоритмы Дейкстры и Левита — линейной функцией

Алгоритм Форда — квадратичной функцией



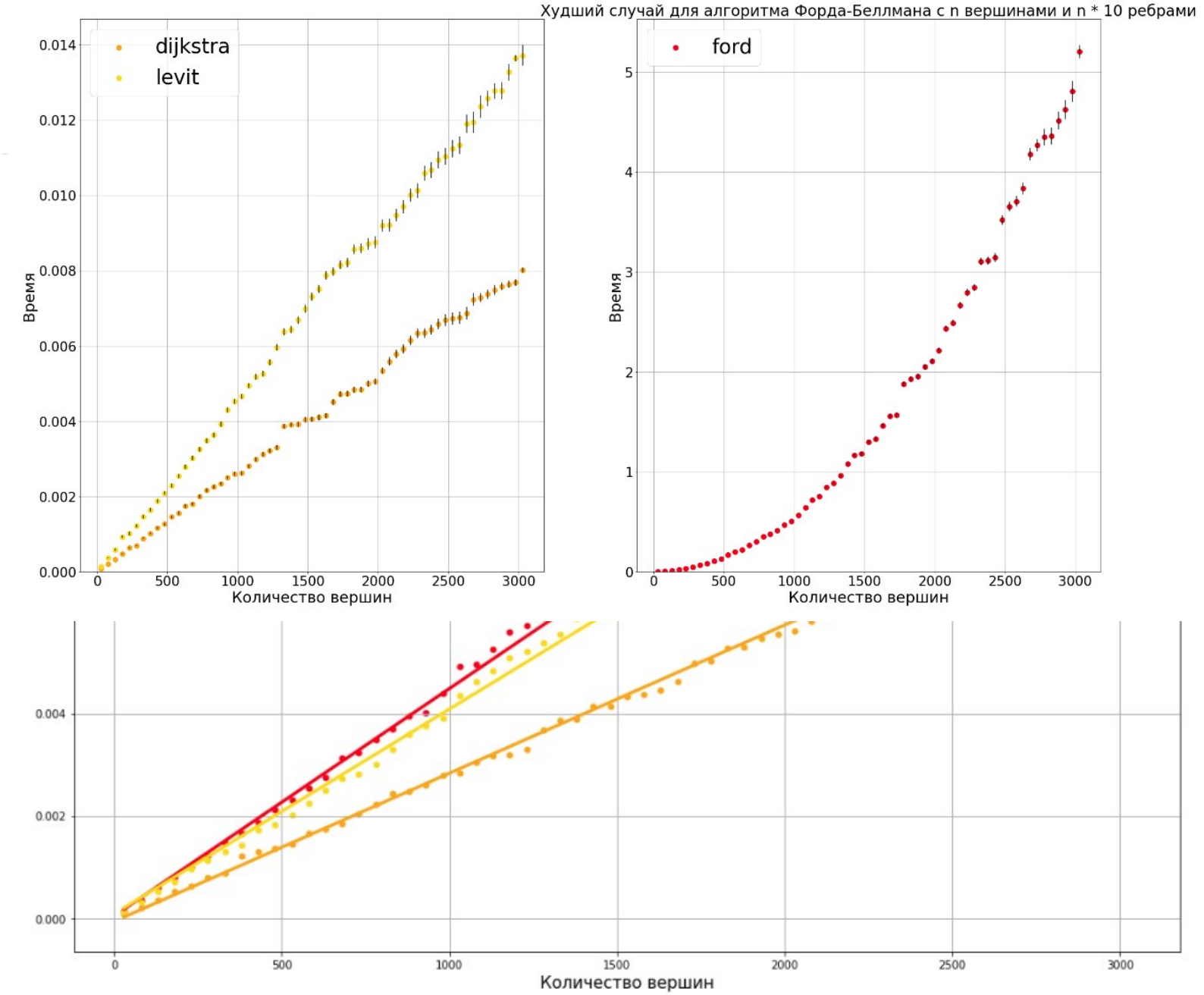
2) Лучшие данные для алгоритмов Форда-Беллмана и Левита с n вершинами и n * 10 ребрами



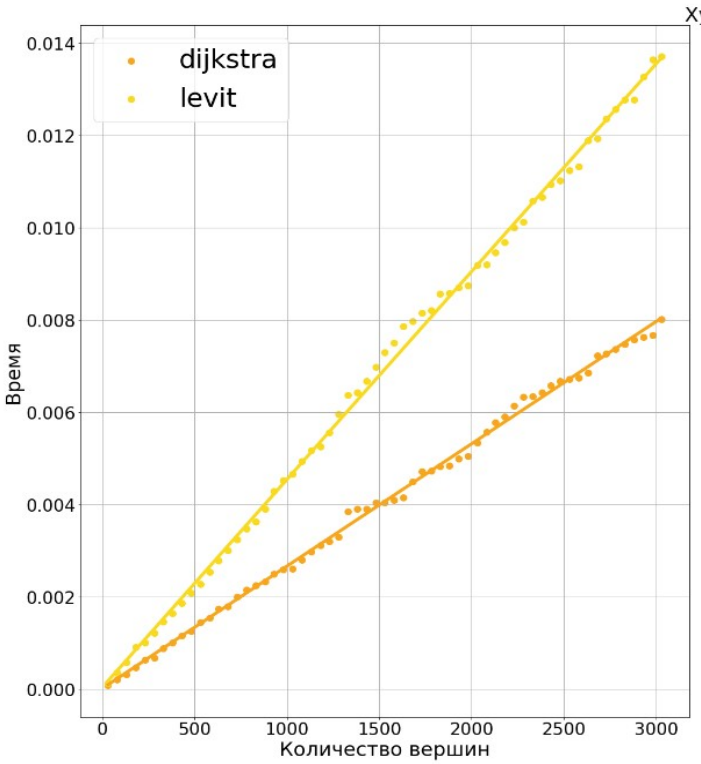
Выполняем приближение полученных данных следующими функциями:

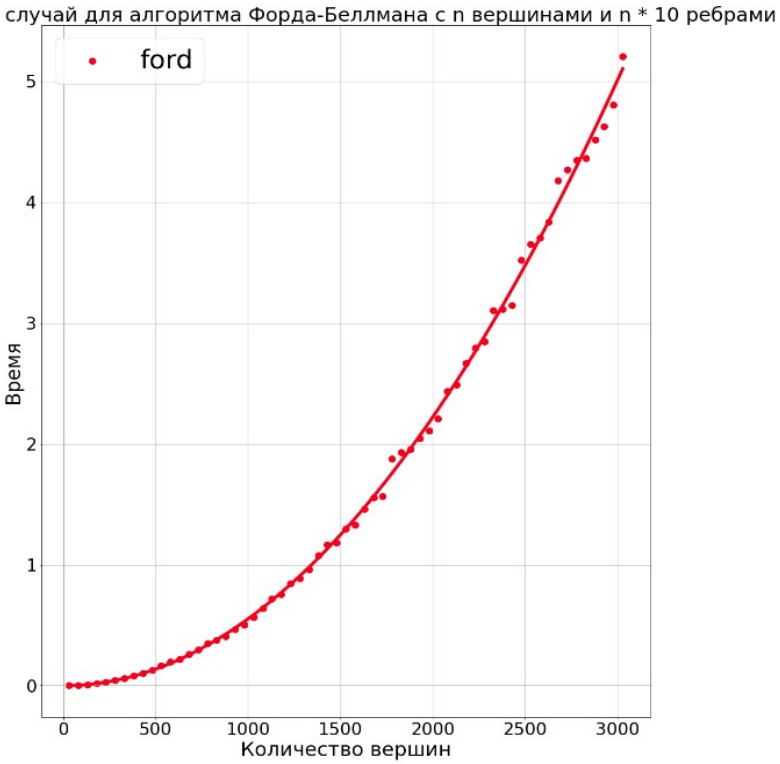
Все три линейными функциями

3) Худшие данные для алгоритма Форда-Беллмана с n вершинами и n * 10 ребрами

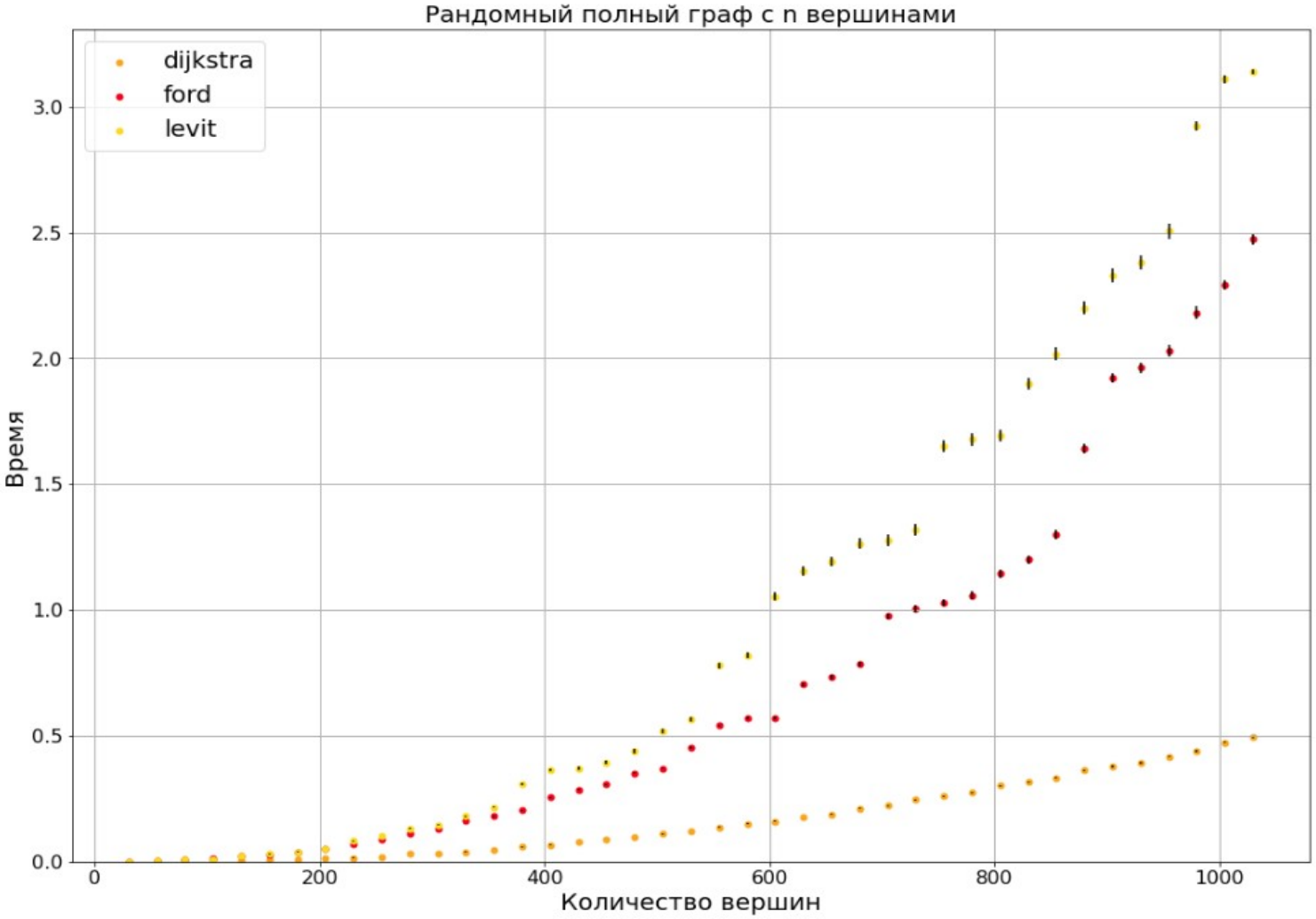


Выполняем приближение полученных данных следующими функциями:
Данные полученные с помощью алгоритмов Дейкстры и Левита — линейной функцией.
Данные полученные с помощью алгоритма Форда-Беллмана — квадратичной функцией.

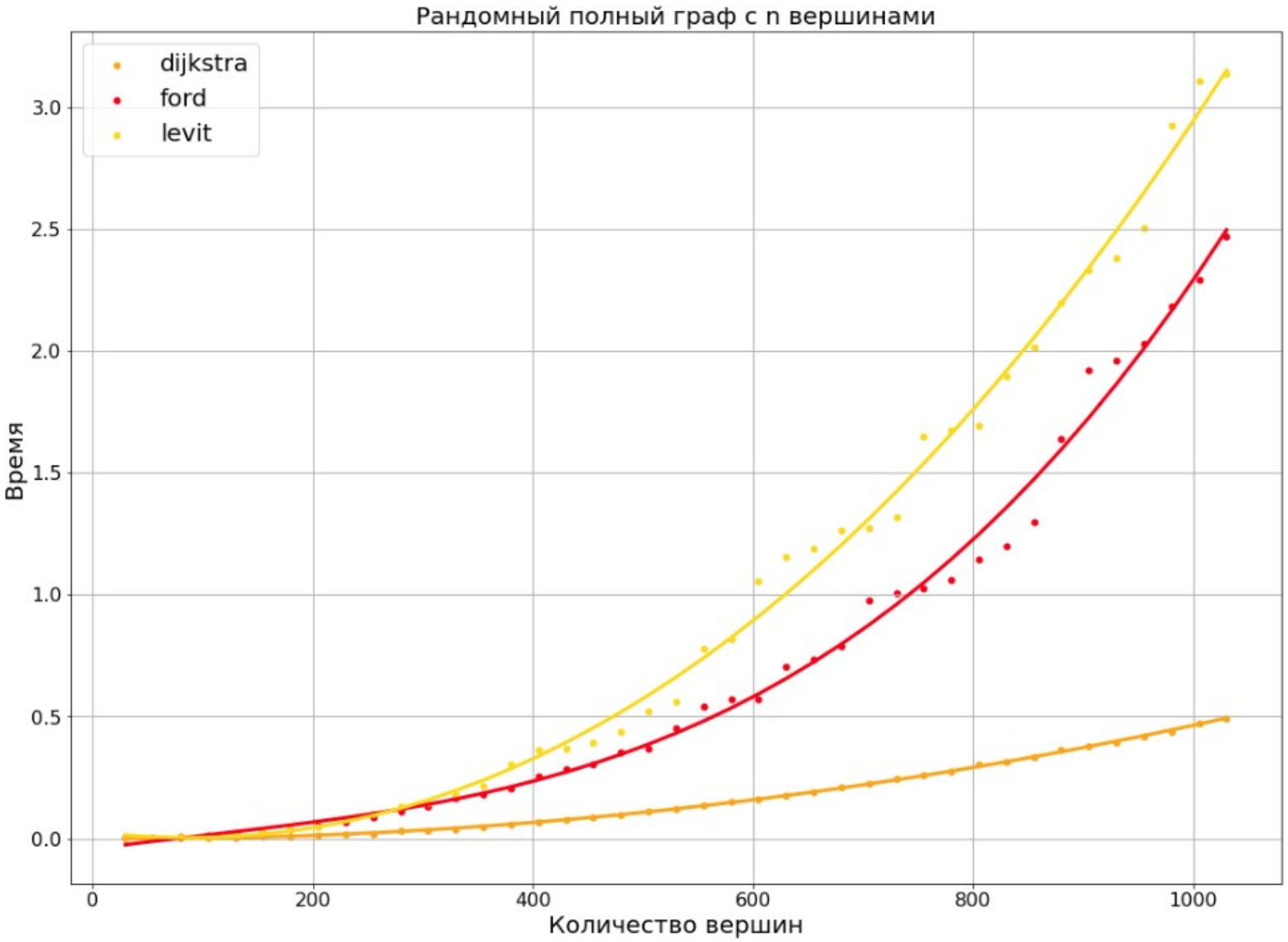




4) Рандомный граф с n вершинами и n * (n + 1) / 2 ребрами(полный граф)

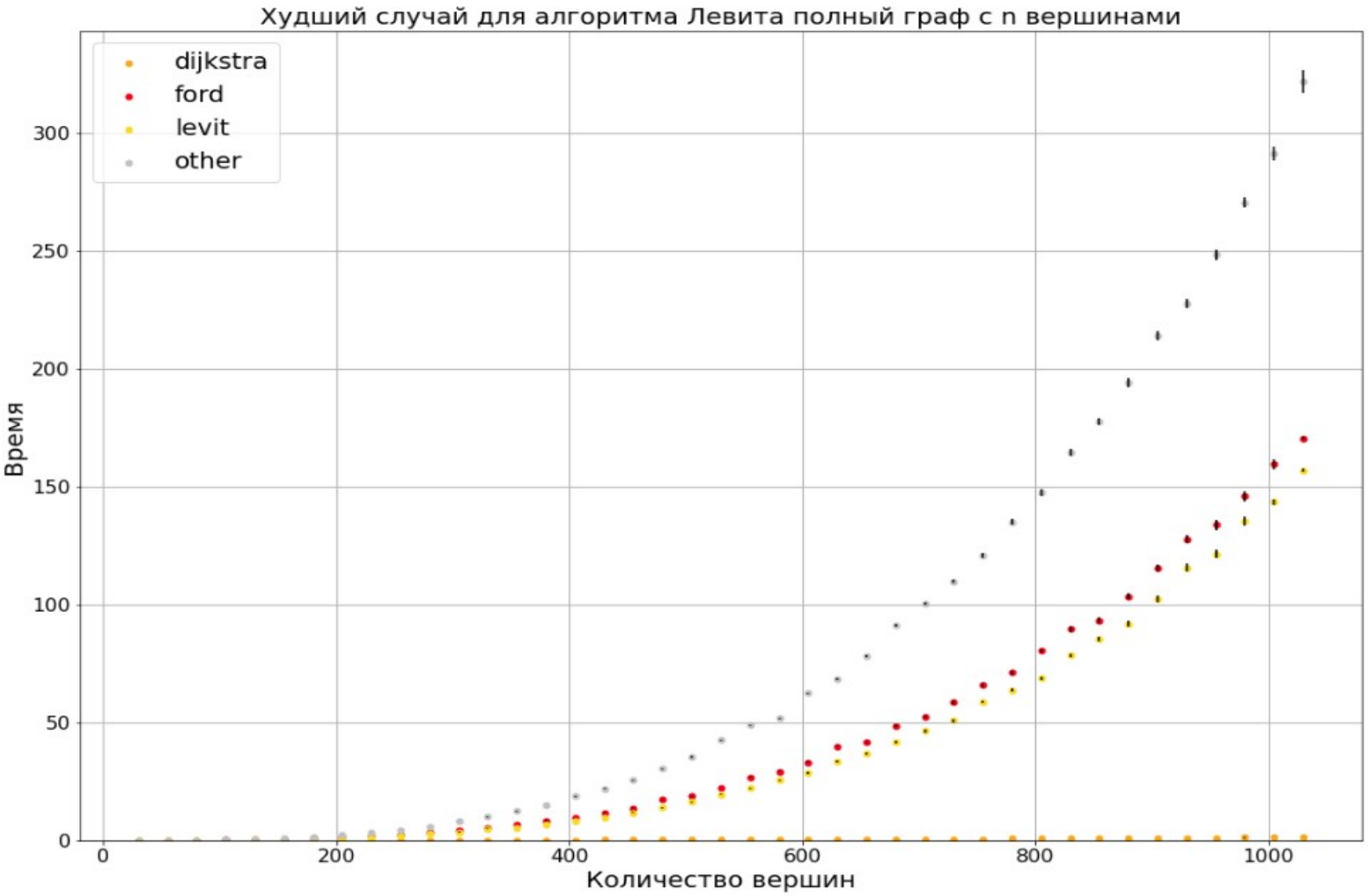


Выполняем приближение полученных данных следующими функциями:
Данные полученные с помощью алгоритмов Форда-Беллмана и Левита — полиномом 3 степени.
Данные полученные с помощью алгоритма Дейкстры — полиномом 2 степени.

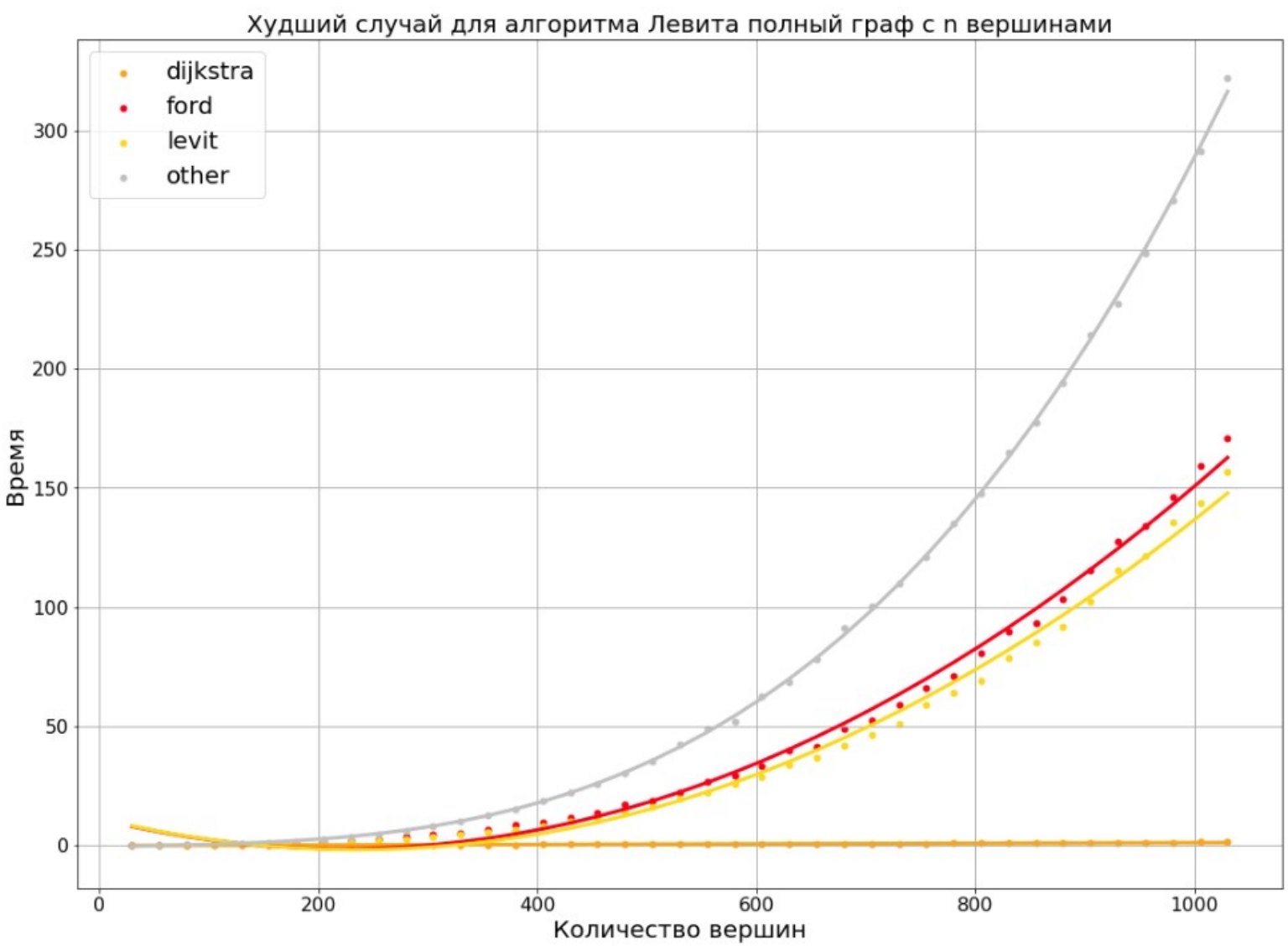


5) Худшие данные для алгоритма Левита в полном графе.

other - Алгоритм поиска кратчайшего пути проходящего через заданные вершины



Выполняем приближение полученных данных следующими функциями:
Данные полученные с помощью алгоритмов Форда-Беллмана и Левита — полиномом 3 степени.
Данные полученные с помощью алгоритма Дейкстры — полиномом 2 степени.
Данный полученные с помощью алгоритма поиска кратчайшего пути проходящего через заданные вершины — полиномом 3 степени



5. Обоснование и анализ результатов.

2) Худшие данные для алгоритма Форда-Беллмана

Алгоритм должен проработать как можно больше фаз. Для этого должна быть как минимум 1 вершина кратчайший путь до которой из стартовой будет проходить через максимально возможное количество вершин.

При работе алгоритм Форда-Беллмана на вышеописанных входных данных отработает n — 1 фазу, где каждая фаза будет состоять из m действий. Следовательно алгоритм отработает около n * m действий(где n — количество вершин, m = n * 10). Сгенерированные входные данные можно аппроксимировать квадратичной функцией.

Условие на то, что кратчайший путь между некоторой парой вершин должен состоять из максимального количества ребер никак не влияет на время работы алгоритмов Дейкстры и Левита.

Действительно, графикам видно, что алгоритмы Дейкстры и Левита на этих данных ведут себя так же, как и на рандомной графе.

3) Лучшие данные для алгоритма Форда-Беллмана и Левита.

Алгоритм должен найти все кратчайшие расстояния за минимальное количество фаз. Давайте считать, что стартовая вершина — 0. Тогда построим такой граф, что все кратчайшие расстояния от стартовой вершины это минимальное количество ребер от нее до всех остальных вершин. Следовательно алгоритм совершит небольшое количество фаз, а когда станет понятно, что не произошло ни 1 релаксации, алгоритм завершится.

В алгоритме Левита все вершины на этих данных максимально быстро попадут в очередь M`1, откуда они так же стремительно должны попасть в очередь M0 и алгоритм завершиться.

На графиках видно, что алгоритмы Форда-Беллмана и Левита на таких данных работают быстрее чем на рандомных. И их (как и алгоритм Дейкстры) можно максимально хорошо приблизить линейной функцией.

Время работы алгоритма Форда-Беллмана на лучших и худших данных отличается примерно в n (количество вершин) раз, что подтверждается функциями с помощью которых производилось приближение данных. (линейной — на лучших данный, квадратичной — на худших).

5) Худшие данные для алгоритма Левита в полном графе.

Рассмотрим полный граф K с n вершинами и такими m рёбрами, идущими в лексикографическом порядке:

- для всех вершин $1<i<j\leqslant n$ вес ребра (i, j) = j − i− 1, т.е. количество вершин между i и j; $w_{i,i+1}=0$,
- ребро (1,n) веса 0,
- для всех вершин 1<i<n вес ребра (1, I) = w1, i + 1 + i −1; от1 до I вершины расстояние равно $\sum_{k=i-1}^{n-2k}$.

Ясно, что кратчайший путь до каждой вершины равен 0, но в плохом случае алгоритм при подсчёте вершины I будет пересчитывать все вершины до неё (кроме первой). На 1 шаге в очередь положат вершины от 2 до n, причём вершину 1 из M0 больше не достанут. На следующем шаге

добавлений не произойдёт, так как вершины больше 2 уже в очереди. На 3 шаге алгоритм улучшит расстояние до вершины 2 на 1 (что видно из веса рёбер (1,2) и (1,3), равных $\sum k=1n-2k$ и $\sum k=2n-2k$ соответственно), так что её добавят в M1″ и обработают на 4 шаге (релаксаций не происходит). На следующем шаге из обычной очереди достанут вершину 4, расстояние до неё, равное $\sum k=3n-2k$, на 2 меньше, чем расстояние до 2 и 3 вершин. Их добавят в срочную очередь, но так как $w24-1=w34$, то после подсчёта вершины 3 вершину 2 снова добавят в M1″. Затем дойдёт очередь до вершины 5, что вызовет релаксацию предыдущих вершин 2,3,4, затем прорелаксируют вершины 2,3, и после вершина 2. Аналогично будут происходить релаксации всех вершин при обработке вершины i из очереди M0. Таким образом, вершину i будут добавлять в срочную очередь n−i раз (добавление вершин из очереди M2 с номером больше i) + количество добавлений "старшей" вершины i+1. Количество добавлений вершины i составит $1+\sum k=1n-ik$, а сумма всех добавлений примерно составит O(n*m).

Таким образом алгоритм Левита на графе построенном таким образом будет работать за O(n*m) действий.

Для алгоритма Форда-Беллмана эти данные так же будут являться худшими при заданном количестве вершин и ребер, т.к. в таком графе существуют минимальные по стоимости пути проходящие через n - 1 ребро.

Алгоритм Дейкстры же как и везде отработает за O(m * log(n)) действий.

В основе алгоритма поиска кратчайшего пути проходящего через заданные вершины лежит алгоритм Флойда-Уоршелла, асимптотической сложностью которого является O(n^3)

На графике видно, что время работы алгоритмов Левита и Форда-Беллмана сопоставимо. Результаты полученные от этих алгоритмов отлично приближаются функцией f(x) = ax^3.

Алгоритм поиска кратчайшего пути проходящего через заданные вершины так же приближается функцией f(x) = ax^3, но она растет примерно в 2 раза быстрее. Это достигается из-за того что количество ребер в полном графе = n * (n + 1) / 2.

В различных статьях про алгоритм Левита говорится, что на рандомных разряженных графах он работает быстрее, чем алгоритм Форда-Беллмана. Полученные в ходе эксперимента результаты подтверждают это высказывание.
Так же хочется отметить, что на рандомном полном графе алгоритм Левита проявил себя немного хуже, чем алгоритм Форда-Беллмана.

6. Приложение.

Реализация алгоритма Дейкстры.

```
def pathfinder(self, start=0, end=None):
    distances = [INF for _ in range(self.graph.max_vertex() + 1)]
    distances[start] = 0
    q = []
    heapq.heappush(q, (0, start))

    while q:
        dist, v = heapq.heappop(q)
        if dist > distances[v]:
            continue

        for u, len_edge in self.graph.adjacency_list[v]:
            if distances[v] + len_edge < distances[u]:
                distances[u] = distances[v] + len_edge
                heapq.heappush(q, (distances[u], u))

    return distances if end is None else distances[end]
```

Реализация алгоритма Форда-Беллмана.

```
def pathfinder(self, start=0, end=None):
    distances = [INF for _ in range(self.graph.max_vertex() + 1)]
    distances[start] = 0

    for _ in range(len(self.graph.adjacency_list)):
        check = True
        for edge in self.graph.edge_list:
            if distances[edge.s] < INF and distances[edge.s]\
                    + edge.weight < distances[edge.f]:
                distances[edge.f] = distances[edge.s] + edge.weight
                check = False
        if check:
            break
    return distances if end is None else distances[end]
```

Реализация алгоритма Левита.

```
def pathfinder(self, start=0, end=None):
    distances = [INF for _ in range(self.graph.max_vertex() + 1)]
    distances[start] = 0
    q1 = deque()
    q2 = deque()
    q1.append(start)
    unused = set()
    used = set()
    for v in range(len(self.graph.adjacency_list)):
        if v != start:
            unused.add(v)
    while q1 or q2:
        if q2:
            u = q2.popleft()
        else:
            u = q1.popleft()
        for v, dist in self.graph.adjacency_list[u]:
            if v in unused:
                q1.append(v)
                unused.remove(v)
                distances[v] = min(distances[v], distances[u] + dist)
            elif v in used and distances[v] > distances[u] + dist:
                q2.append(v)
                used.remove(v)
                distances[v] = distances[u] + dist
            else:
                distances[v] = min(distances[v], distances[u] + dist)

        used.add(u)
    return distances if end is None else distances[end]
```

Реализация алгоритма поиска кратчайшего пути проходящего через заданные вершины

```
def floyd_algorithm(self):
    size = self.graph.max_vertex()
    print(self.graph.max_vertex())
    distances = [[INF for _ in range(size + 1)] for _ in range(size + 1)]
    for edge in self.graph.edge_list:
        distances[edge.s][edge.f] = edge.weight
    for i in range(size):
        for j in range(size):
            for k in range(size):
                if distances[i][k] < INF and distances[k][j] < INF:
                    distances[i][j] = min(
                        distances[i][j],
                        distances[i][k] + distances[k][j]
                    )
    return distances
```

```
def pathfinder(self):
    graph = self.get_graph_from_specified_vertexes()
    size = len(self.specified_vertexes)
    used = [False for _ in range(size)]
    min_edges_weight = [INF for _ in range(size)]
    min_edges_weight[0] = 0

    for i in range(size):
        v = -1
        for j in range(size):
            if not used[j] and (v == -1 or min_edges_weight[j]
                                < min_edges_weight[v]):
                v = j
        used[v] = True

        for j in range(size):
            min_edges_weight[j] = min(min_edges_weight[j], graph[v][j])
    return min_edges_weight
```

```
def get_graph_from_specified_vertexes(self):
    distances = self.floyd_algorithm()
    size = len(self.specified_vertexes)
    new_graph = [[INF for _ in range(size)] for _ in range(size)]
    for i in range(size):
        for j in range(size):
            if j != i:
                new_graph[i][j] = distances[
                    self.specified_vertexes[i]
                ][self.specified_vertexes[j]]

    return new_graph
```

Ссылка на гитхаб репозиторий с сгенерированными данными, результатами измерений.
https://github.com/asdminerpro/search_in_graph