АиСД-2, 2023, КДЗ-3

Степанов А, БПИ212

Вводная

**Задание: /КДЗ3.pdf **

Мною реализованы алгоритм Дейкстры, Флойда-Уоршела, Форда-Беллмана, A*.

Запуск проекта

Через терминал из корневой директории проекта:

```
python generate.py # генерация входных данных

cmake . -B build # Создание папки для скомпилированного проекта

cd build && # Компиляция и запуск
cmake --build . --target all && ./main &&
cd ..

python make_graphs.py ./data/output # создание графиков
```

Для запуска таким образом на ПК должны быть установлен **cmake** и **python**.

Отчёт

> N - количество вершин в графе. E - количество ребер в графе. V - количество вершин / ребер в графе, над которым производились измерения.

Зависимости времени работы алгоритма от числа вершин и от числа ребер примерно одинаковы для всех алгоритмов (что логично -- ведь входные данные одни и те же, однако в задании сказано сгенерировать именно один набор данных: пункт 1).

- Алгоритм Дейкстры:

Фактические данные: Временная сложность: O((N + E) log N).

Полученные данные: Для полных, связных и разреженных графов алгоритм имеет квадратичную сложность O(V * log(V)) с примерно одинаковой константой.

- Алгоритм Флойда-Уоршела:

Фактические данные: Временная сложность: O(N^3).

Полученные данные: Для полных, связных и разреженных графов алгоритм имеет квадратичную сложность O(V^3). Однако константа сложности различается: для полных и разреженных графов она примерно одинакова (но в первом случае несколько больше), а для связных графов она заметно - примерно в 2 раза - меньше.

- Алгоритм Форда-Беллмана:

Фактические данные: Временная сложность: O(N * E).

Полученные данные: Для полных, связных и разреженных графов алгоритм имеет квадратичную сложность O(V^2) с практически одинаковой константой.

- Алгоритм A*:

Фактические данные: Временная сложность зависит от конкретной реализации. В наилучшем случае может быть O(1), а в худшем случае может быть экспоненциальной.

Полученные данные: Для полных и разреженных графов алгоритм имеет практически линейную сложность с примерно равными константами. Для связных графов алгоритм имеет сложность, близкую к O(N^2), однако полученные данные очень сильно колеблются, что может быть связано с тем, что входные данные не являются оптимальными для данного алгоритма.

> Соответственно, фактические данные и полученные данные в целом совпадают.

- Аггрегированные данные:

В среднем хуже всего себя проявили алгоритмы Флойда-Уоршела и Форда-Беллмана: они имеют экпоненциальную сложность и время выполнения на моих входнных данных, измеряемое в const*1e9 наносекунд.

Алгоритм Дейкстры и алгоритм A* имеют примерно одинаковую сложность близкую к линейной и время выполнения на моих входнных данных, измеряемое в const*1e7 наносекунд. Однако производительность A* на связных графах сильно упала (до const*1e8 наносекунд).

Выводы

Таким образом, судя по полученным данным, наилучшим алгоритмом для решения задачи поиска кратчайшего пути является алгоритм Дейкстры, а наихудшим - алгоритм Флойда-Уоршела. Алгоритм A* также показал себя достаточно хорошо, однако его производительность сильно зависит от входных данных. Алгоритм Форда-Беллмана показал себя несколько лучше алгоритма Флойда-Уоршела, однако все равно значительно хуже алгоритма Дейкстры и A*.

Карта проекта

> Main файл проекта, запускающий процесс анализа работы всех исходных алгоритмов: *main.cpp: /main.cpp*
>
> Python скрипт, генерирующий входные данные: *generate.py_graphs: /generate_graphs.py*
>
> Python скрипт, генерирующий графики на основе полученных выходных данных *make_diagrams.py: /make_diagrams.py*

-> data: /data/

> Входные файлы: *graphs: /data/graphs/*
>
> Выходные данные: *output: /data/output/*
>
> Графики, иллюстрирующие выходные данные: *results: /data/results/*
>
> > Графики отсортированы по папкам, соответствующим файлам со входными данными

-> for_time_measure: /for_time_measure/

> Содержит алгоритмы поиска подстрок без подсчёта производимых операций сравнений строк.

-> for_comp_count_measure: /for_comp_count_measure/

> Содержит алгоритмы поиска подстрок с подсчётом производимых операций сравнений строк.

-> checker: /checker/

> Содержит функции предназначенные для измерения времени работу функции / количества сравнений строк, производимых в ней.

-> task_execution: /task_execution/

> Содержит функции предназначенные для получения времени работы / количества сравнений строк, производимых в ней на некоторых входных данных и записи полученных данных в файл.