

# Технически университет - София

курсов проект по дисциплина

## „Графи и Приложения“

изготвил

**Илия Георгиев Къркъмов**  
**991319006**

**София**  
**2020**

## Увод

### Проектиране на решение

Технологии за разработка

Език за програмиране

Използвани външни библиотеки

Файлова структура на проекта

Имплементация на граф

Клас „AdjacencyList“

Зареждане на граф от CSV файл

Премахване на ребра с ниско тегло

Намиране на максимално покриващо дърво

Записване на граф в GEXF формат

### Резултати

Получени графи

Подграф 1

Подграф 2

Максимално покриващо дърво на подграф 1

Максимално покриващо дърво на подграф 2

Скорост на изпълнение

### Използвана литература

# Увод

Курсовият проект по дисциплина „Графи и Приложения“ изисква да бъде разработено приложение, което да прочита файлове във формат CSV. Файловете представляват матрица на корелациите на логаритмичната възвръщаемост (log-returns) на 25 акции на компании с голяма пазарна капитализация и матрица на корелациите на волатилностите на същите 25 акции, пресметнати за периода 01.01.2019 до 14.12.2019. Данните прочетени от файловете трябва да бъдат обработени и съхранени в подходящи структури от данни описващи граф, след което върху получените графи трябва да бъдат извършени следните задачи:

- За всеки връх, да се запазят само ребрата съответстващи на трите теглови коефициента с най-голяма абсолютна стойност.
- Получените два подграфа трябва да бъдат записани в GEXF файлове [1].
- За двата под графа да бъдат намерени максимално покриващи дървета.
- Получените две максимално покриващи дървета да бъдат записани в GEXF файлове.
- Получените четири GEXF файла трябва да бъдат заредени успешно от софтуера Gephi [2] и да бъдат експортирани в png формат.

# Проектиране на решение

## Технологии за разработка

### Език за програмиране

За разработването на курсовият проект е избран език за програмиране C++. Приложението е независимо от конкретна среда на разработка, защото проектите са описани чрез CMake. Програмният код може да бъде компилиран на различни операционни системи, защото не използва директно функционалности от операционната система. Разчита се на стандартната библиотека на C++ за предоставяне на абстракция върху функционалностите на операционната система.

### Използвани външни библиотеки

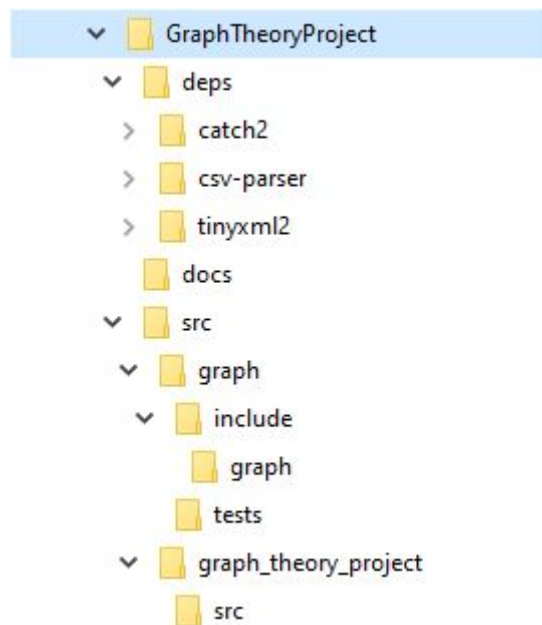
Използвани са няколко външни библиотеки, които да улеснят разработването на проекта. Нито една от тях не е свързана с реализирането на граф, намиране на максимално покриващо дърво или друга функционалност свързана с реализация на основните задачи по курсовия проект. Използваните библиотеки са следните:

1. **catch2** - библиотека за писане и изпълнение на unit тестове [3];
2. **csv-parser** - библиотека за прочитане на CSV файлове [4];
3. **tinysql2** - библиотека за четене и писане на XML [5].

### Файлова структура на проекта

Файловата структура на проекта, показана на фигура 1, подобно на други C++ проекти е разделена в множество папки както следва:

- Папка **deps** съдържа програмният код на всички външни библиотеки;
- Папка **docs** съдържа документи свързани със курсовият проект;
- Папка **src** съдържа подпапки всяка, от които представлява подпроект съдържащ програмен код.
  - Папка **graph** е header-only библиотека съдържаща програмния код реализиращ функционалността за работа с графи;
  - Папка **graph\_theory\_project** е програмният код на изпълнимия файл стартиращ проекта.



Фигура 1 - Файлова структура на проекта

## Имплементация на граф

За имплементацията на граф в настоящия курсов проект са разгледани две възможни представяния:

- Списък на съседите (Adjacency list);
- Матрица на съседство (Adjacency matrix).

Данните предоставени в CSV файловете представляват матрици на съседство и лесно могат да бъдат представени, като имплементация чрез „матрица на съседство“. По този начин цялата матрица ще бъде заредена в паметта и достъпът до върховете би бил доста „евтин“. Но поради факта, че графите са ненасочени, матриците са симетрични, което означава, че връх  $(i, j)$  и връх  $(j, i)$  са еднакви. В такъв случай представянето чрез „матрица на съседство“ не би било подходящо, защото зареждането в паметта на цялата матрица е излишно. Вариантът за представяне на граф чрез „списък на съседите“ е по-подходящ, защото по-малко елементи от матрицата ще бъдат заредени в паметта.

### Клас „AdjacencyList“

Клас „AdjacencyList“ е съставен от списък съдържащ върховете на графа. Всеки връх съдържа списък от съседни върхове, както и текстов идентификатор. Всеки съседен връх съдържа теглови коефициент определящ теглото на реброто свързващо двата върха. Реализацията на класа използва C++ шаблони (templates), за да предостави възможност на потребителя да дефинира данните съхранявани за всеки връх и данните съхранявани за всеки съседен връх (данните за ребро). По този начин класът „Adjacency list“ се дефинира по абстрактен начин и може да бъде преизползван и за други данни. Фигура 2 показва интерфейса на класа.

```

15     template<typename TVertexData, typename TEdgeData>
16     class AdjacencyList
17     {
18     public:
19         explicit AdjacencyList() noexcept = default;
20         explicit AdjacencyList(size_t vertexCount) noexcept;
21
22         void reserve(size_t vertexCount);
23         void reserve(VertexDescriptor vertex, size_t edgeCount);
24
25         [[nodiscard]] size_t size() const noexcept;
26         [[nodiscard]] size_t size(VertexDescriptor vertex) const noexcept;
27         [[nodiscard]] size_t sizeOfEdges() const noexcept;
28
29         VertexDescriptor addVertex(TVertexData vertexData);
30         VertexDescriptor addVertex(TVertexData vertexData, size_t edgeCount);
31
32         EdgeDescriptor addEdge(VertexDescriptor src, VertexDescriptor dest, TEdgeData edgeData);
33
34         TVertexData& getVertex(VertexDescriptor vertex);
35         const TVertexData& getVertex(VertexDescriptor vertex) const;
36
37         TEdgeData& getEdge(VertexDescriptor src, VertexDescriptor dest);
38         const TEdgeData& getEdge(VertexDescriptor src, VertexDescriptor dest) const;
39
40         std::pair<VertexDescriptor, TEdgeData> getEdgeById(VertexDescriptor vertex, EdgeDescriptor edge) const;
41
42         std::vector<std::pair<std::pair<VertexDescriptor, VertexDescriptor>, TEdgeData>> getEdges() const;
43         std::vector<std::pair<VertexDescriptor, TEdgeData>> getEdges(VertexDescriptor vertex) const;
44
45         <UnaryPredicate>
46         void sortEdges(VertexDescriptor vertex, UnaryPredicate p);
47
48         void clear();
49         void clearEdges();
50
51     private:
52         struct Edge
53         {
54             TEdgeData edgeData;
55             VertexDescriptor vertex;
56
57             Edge(TEdgeData edgeData, VertexDescriptor vertex) : edgeData(std::move(edgeData)), vertex(vertex) {}
58         };
59
60         struct Vertex
61         {
62             TVertexData vertexData;
63             std::vector<Edge> edges;
64
65             Vertex(TVertexData vertexData, std::vector<Edge> edges)
66                 : vertexData(std::move(vertexData)), edges(std::move(edges)) {}
67         };
68
69         std::vector<Vertex> m_vertices;
70     };

```

Фигура 2 - Интерфейс на клас „AdjacencyList“

Както се вижда на фигура 2 имплементацията на класа използва **std::vector** за структура от данни за съхранение на върховете и съседните върхове. Друг вариант за структура от данни е **std::list**. Съдейки по алгоритмичната сложност за добавяне и изтриване на елемент, съответно  $O(1)$  за **std::list** и  $O(N)$  за **std::vector**, по-добър кандидат изглежда **std::list**. От друга страна **std::list** е имплементиран чрез „doubly-linked lists“, което означава, че данните не са последователни в паметта за разлика от **std::vector**. Това би довело до множество cache misses, а тъй като реализацията на курсовият проект изпълнява множество обхождания по върховете на графа то това би се отразило на скоростта на изпълнение на програмата.

Класът показан на фигура 2 се намира във файл „adjacency\_list.h“.

## Зареждане на граф от CSV файл

Зареждането на данните от двата CSV файла в паметта се извършва използвайки библиотеката „csv-parser“. Преобразуването на данните и зареждането им в обект от тип „AdjacencyList“ се извършва от допълнителна функция, която дефинира оператор „right shift“ за клас „AdjacencyList“ както се вижда от прототипа показан на фигура 3.

```
25     template<typename TVertexData, typename TEdgeData>
26     std::istream& operator>>(std::istream& is, AdjacencyList<TVertexData, TEdgeData>& graph)
27     {
```

Фигура 3 - Оператор „right shift“ за клас „AdjacencyList“

Функцията показана на фигура 3 се намира във файл „csv\_io.h“.

## Премахване на ребра с ниско тегло

Задачата по премахване на тегловите коефициенти съответстващи на слаба корелация е реализирана от функция „sieveEdgesIf“. Прототипът на функцията е показан на фигура 4.

```
11     template<typename TVertexData, typename TEdgeData, typename UnaryPredicate>
12     void sieveEdgesIf(AdjacencyList<TVertexData, TEdgeData>& graph, size_t keep, UnaryPredicate p)
13     {
```

Фигура 4 - Прототип на функция „sieveEdgesIf“

Горната функция е дефинирана по абстрактен начин, така че да може да бъде преизползвана. Тя приема като параметри графа, за когото трябва да бъде извършена операцията, броят на ребрата, които да бъдат запазени и предикат, който дефинира условието за сортиране на ребрата.

Имплементацията на функцията създава временна структура от данни (whitelist), която представлява списък от ребрата, които трябва да бъдат запазени, след което обхожда графа сортирайки ребрата на всеки връх спрямо условието дефинирано от предиката

р. Във временната структура се запазват „**keep**“ на брой ребрата за всеки връх. Накрая ребрата на графа се премахват, след което в графа се добавят само тези ребра, които са запазени във временната структура.

Функцията показана на фигура 4 се намира във файл „**algorithms.h**“.

## Намиране на максимално покриващо дърво

Задачата за намиране на максимално покриващо дърво на граф е реализирана от функцията „**kruskalSpanningTree**“. Прототипът на функцията е показан на фигура 5.

```
38     template<typename TVertexData, typename TEdgeData, typename UnaryPredicate>
39     void kruskalSpanningTree(AdjacencyList<TVertexData, TEdgeData>& graph, UnaryPredicate p)
40     {
```

Фигура 5 - Прототип на функцията „kruskalSpanningTree“

Подобно на функцията показана на фигура 5, тази също е дефинирана по абстрактен начин, така че да може да бъде преизползвана. Тя приема като параметри графа, за когото трябва да бъде намерено максимално покриващо дърво и предикат, който дефинира условието за сортиране на ребрата на графа. В зависимост от предиката функцията намира минимално покриващо дърво или максимално покриващо дърво.

Имплементацията на функцията извлича списък с всички ребра от подадения граф, след което ги сортира спрямо условието дефинирано от предиката р. Функцията дефинира и резултатен списък, където ще бъдат съхранени ребрата, които ще формират покриващо дърво. Обхожда се списъкът от ребрата на графа и с помощта на структура „Disjoint sets“ (Union-Find) се определя кои ребра трябва да бъдат добавени в резултатния списък. Обхождането се прекратява ако в резултатния списък броят на елементите достигне  $N - 1$ , където  $N$  е броят на върховете в началния граф. Накрая ребрата на графа се премахват, след което в графа се добавят само тези ребра, които са запазени в резултатния списък.

За реализацията на тази задача са разгледани няколко възможни алгоритъма:

- Алгоритъм на Kruskal [6];
- Алгоритъм на Prim [7];
- Алгоритъм на Borůvka [8].

Алгоритъмът на Prim се препоръчва, когато броят на ребрата в графа е по-голям от броя на върховете, също така алгоритъмът е по-добър от този на Kruskal от гледна точка алгоритмична сложност. Алгоритъмът на Prim би бил добър избор за решаването на задачата дефинирана в текущият курсов проект. Алгоритъмът на Borůvka се препоръчва, когато всички ребра в графа имат различен (уникален) теглови коефициент, също така той може да бъде лесно паралелизиран. За реализацията на задачата е избран алгоритъмът на Kruskal, защото броят на ребрата не е много голям и също така той е най-лесен за имплементация.



Функцията показана на фигура 4 се намира във файл „**algorithms.h**“.

## Записване на граф в GEXF формат

Записването на получените четири графа в XML формат отговарящ на схемата GEXF XSD [1] се извършва чрез библиотеката „**tinyxml2**“. Сериализирането се извършва от допълнителна функция, която дефинира оператор „left shift“ за клас „AdjacencyList“, както се вижда от прототипа показан на фигура 6.

```
10 template<typename TVertexData, typename TEdgeData>
11 std::ostream& operator<<(std::ostream& os, const AdjacencyList<TVertexData, TEdgeData>& graph)
12 {
```

Фигура 6 - Оператор „left shift“ за клас „AdjacencyList“

Функцията показана на фигура 6 приема като параметри output stream, където да бъдат записани данните за графа и графът, който да бъде сериализиран. Имплементацията на функцията създава основните елементи на описани в XSD схемата GEXF, след което обхожда графа два пъти, веднъж за да бъдат записани върховете на графа и втори път, за да бъдат записани ребрата му.

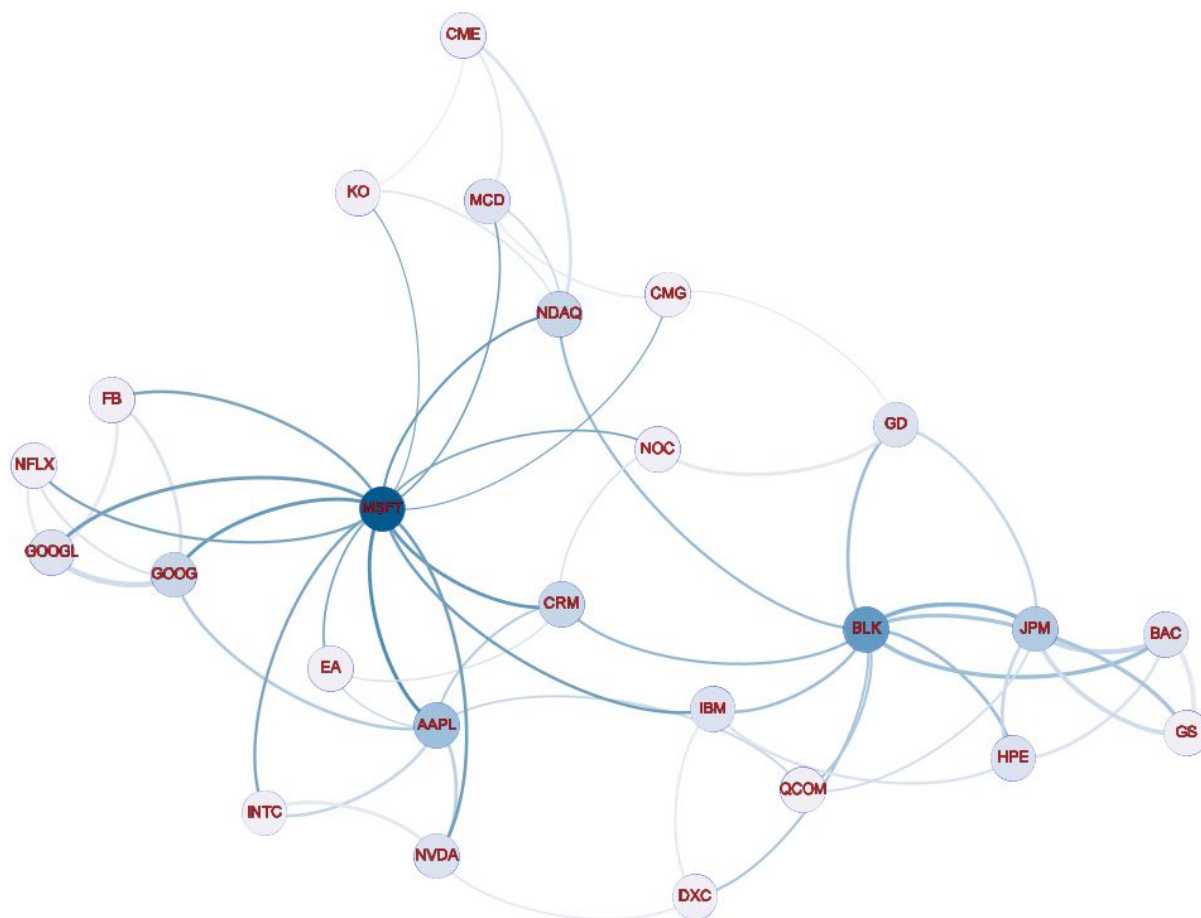
Функцията показана на фигура 6 се намира във файл „**xml\_io.h**“.

## Резултати

### Получени графи

#### Подграф 1

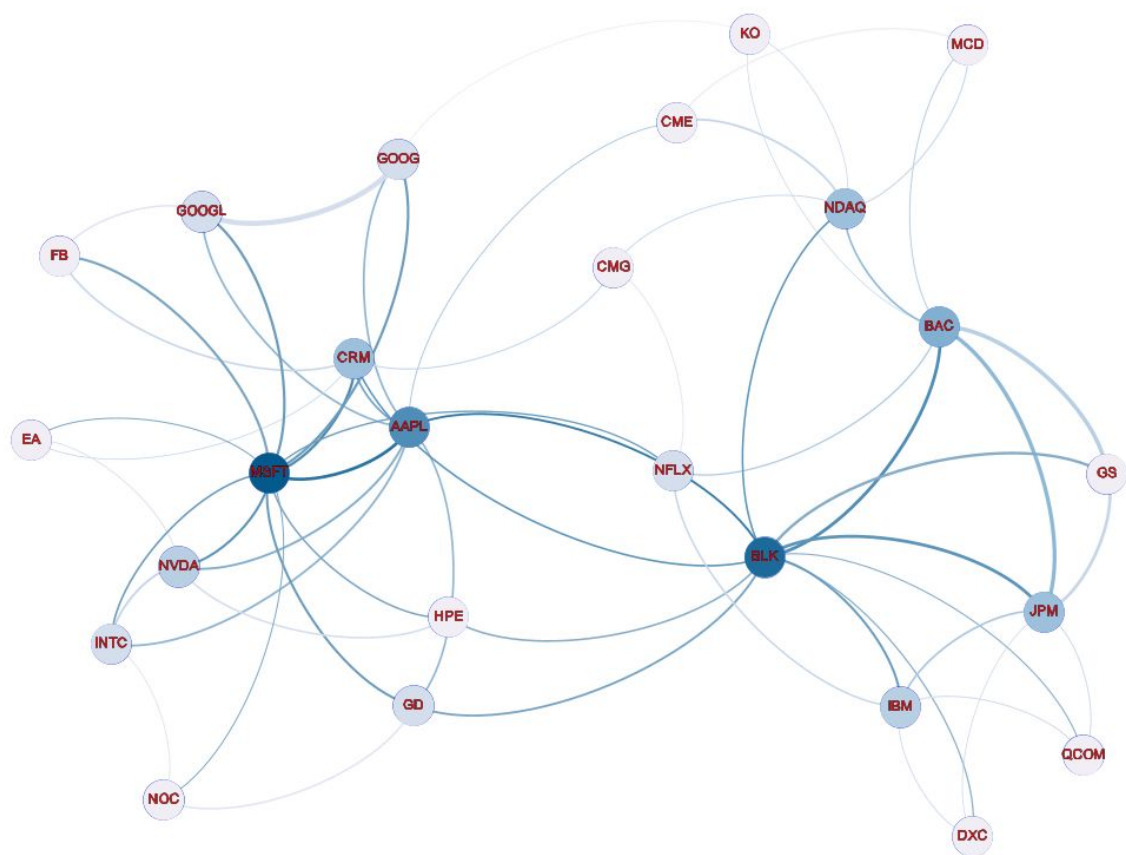
„Подграф 1“ е полученият подграф в резултатът от премахването на ребрата с теглови коефициенти съответстващи на слаба корелация на графа представляващ матрица на корелациите на логаритмичната възвръщаемост (log-returns) на 25 акции на компании с голяма пазарна капитализация. Подграфът е показан на фигура 7.



Фигура 7 - Подграф 1

## Подграф 2

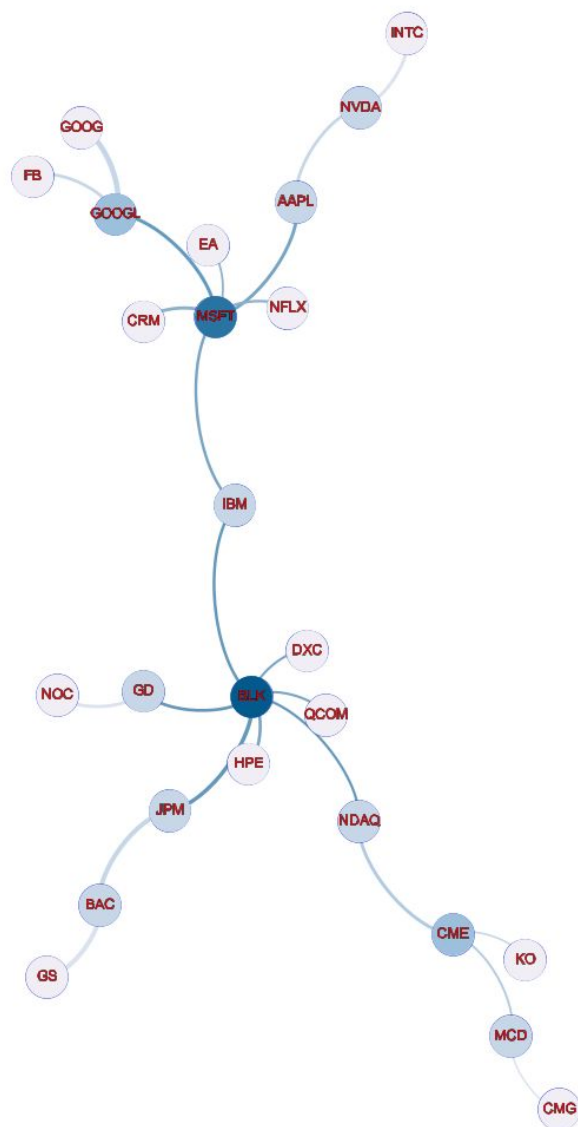
„Подграф 2“ е полученият подграф в резултатът от премахването на ребрата с теглови коефициенти съответстващи на слаба корелация на графа представляващ матрицата на корелациите на волатилностите на 25 акции на компании с голяма пазарна капитализация. Подграфът е показан на фигура 8.



Фигура 8 - Подграф 2

## Максимално покриващо дърво на подграф 1

На фигура 9 е показано максимално покриващо дърво на подграф 1.

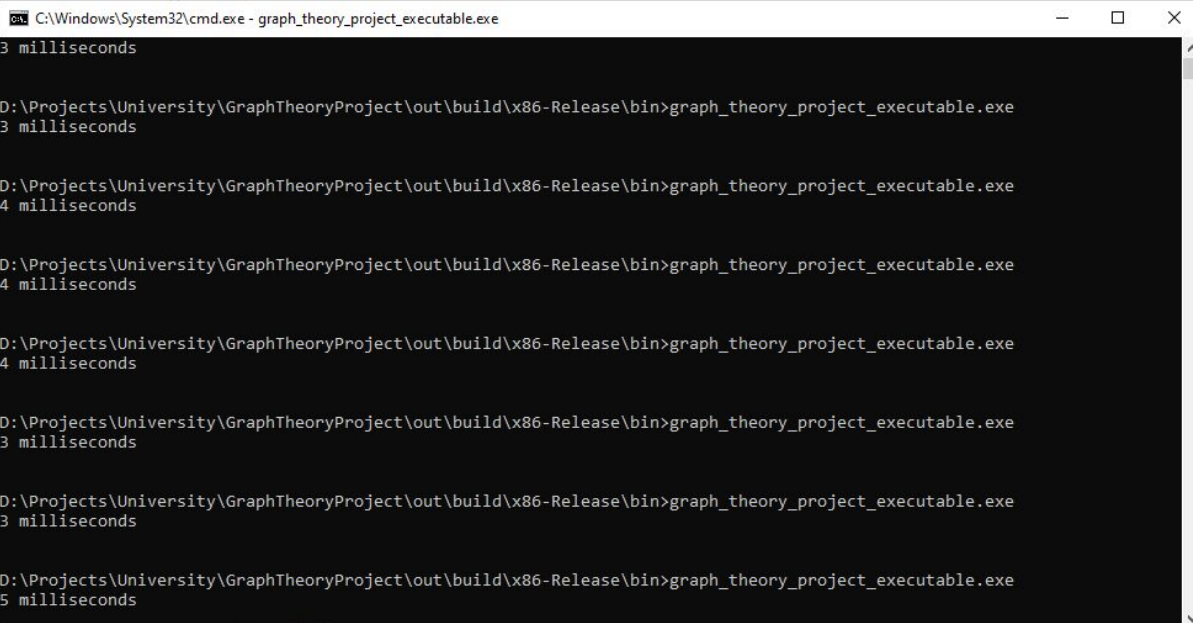


Фигура 9 - Максимално покриващо дърво на подграф 1

## Максимално покриващо дърво на подграф 2

На фигура 10 е показано максимално покриващо дърво на подграф 2.





```
C:\Windows\System32\cmd.exe - graph_theory_project_executable.exe
3 milliseconds

D:\Projects\University\GraphTheoryProject\out\build\x86-Release\bin>graph_theory_project_executable.exe
3 milliseconds

D:\Projects\University\GraphTheoryProject\out\build\x86-Release\bin>graph_theory_project_executable.exe
4 milliseconds

D:\Projects\University\GraphTheoryProject\out\build\x86-Release\bin>graph_theory_project_executable.exe
4 milliseconds

D:\Projects\University\GraphTheoryProject\out\build\x86-Release\bin>graph_theory_project_executable.exe
4 milliseconds

D:\Projects\University\GraphTheoryProject\out\build\x86-Release\bin>graph_theory_project_executable.exe
3 milliseconds

D:\Projects\University\GraphTheoryProject\out\build\x86-Release\bin>graph_theory_project_executable.exe
3 milliseconds

D:\Projects\University\GraphTheoryProject\out\build\x86-Release\bin>graph_theory_project_executable.exe
5 milliseconds
```

*Фигура 11 - Груб тест показващ времето за изпълнение за няколко опита*

Заснемането на времеви точки се прави в началото на стартиране на програмата и в края на изпълнението и. Това включва и операциите по писане и четене от файловата система. Изследвайки изпълнението на програмата чрез profiler се вижда, че в доста голям процент от времето на изпълнение, процесорът изпълнява инструкции свързани с input/output операции. Истински тест на скоростта на изпълнение не би включвал времето за извършване на операции по писане и четене, както и би изпълнил разписаните алгоритми по-голям брой пъти върху по-обемни данни.

# Използвана литература

- [1] GEXF формат - <https://gephi.org/gexf/format/schema.html>
- [2] Gephi - <https://gephi.org/>
- [3] Catch2 библиотека за писане и изпълнение на unit tests - <https://github.com/catchorg/Catch2>
- [4] csv-parser библиотека за четене на csv файлове - <https://github.com/AriaFallah/csv-parser>
- [5] tinysql2, библиотека за писане и четене на XML файлове - <https://github.com/leethomason/tinysql2>
- [6] Алгоритъм на Kruskal - [https://en.wikipedia.org/wiki/Kruskal%27s\\_algorithm](https://en.wikipedia.org/wiki/Kruskal%27s_algorithm)
- [7] Алгоритъм на Prim - [https://en.wikipedia.org/wiki/Prim%27s\\_algorithm](https://en.wikipedia.org/wiki/Prim%27s_algorithm)
- [8] Алгоритъм на Borůvka - [https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s\\_algorithm](https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm)