Министерство науки и высшего образования Российской Федерации  
Пензенский государственный университет  
Кафедра «Вычислительная техника»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовой работе

по курсу «Логика и основы алгоритмизации в инженерных задачах»

на тему «Реализация алгоритма нахождения наибольшего паросочетания в двудольном графе»

Выполнил:

студент группы 22ВВВ2:  
 Хоссейни Нежад С. А. С. М.

Приняли:

Митрохин М. А.

Акифьев И. В.

Пенза 2023

**Содержание**

Реферат 5

[Введение](#_Toc135151524) 6

[1 Постановка задачи 7](#_Toc135151520)

[2 Теоритическая часть задания 7](#_Toc135151521)

[3 Описание алгоритма программы 9](#_Toc135151522)

[4 Описание программы 11](#_Toc135151524)

[5 Тестрирование 13](#_Toc135151524)

[6 Ручной расчет задачи 17](#_Toc135151524)

[Заключение 18](#_Toc135151526)

[Список литературы 19](#_Toc135151527)

[Приложение А Листинг программы 21](#_Toc135151528)

**Реферат**

Отчет 27 стр, 12 рисунков.

ДВУДОЛЬНЫЙ ГРАФ, ТЕОРИЯ ГРАФОВ, ПОИСК В ГЛУБИНУ, АЛГОРИТМ КУНА

Цель исследования – разработка программы, способная найти наибольшее паросочетание в двудольном графе.

В работе рассмотрены правила поиска в глубину и теорема Бержа, на основе которой работает алгоритм Куна. Установлено, что с помощью данного алгоритма можно найти наибольшее паросочетание в двудольном графе.

**Введение**

Двудольные графы являются важным классом графов, которые находят широкое применение в различных областях, таких как компьютерные науки, математика и экономика. Например задача о назначениях, где необходимо оптимально распределить ресурсы между задачами. Одной из ключевых задач, связанных с двудольными графами, является нахождение наибольшего паросочетания, то есть паросочетание, мощность которого максимальна среди всех возможных паросочетаний в данном графе.

Теорема Бержа, названная в честь знаменитого французского математика Клода Бержа, является основой для решения этой задачи. Она утверждает, что паросочетание без увеличивающих цепей является наибольшим. Это ключевое утверждение позволяет нам эффективно находить наибольшее паросочетание.

Алгоритм Куна - это популярный метод решения этой задачи. Он использует поиск в глубину для нахождения увеличивающих цепей, которые затем используются для увеличения текущего паросочетания. Этот алгоритм эффективен и прост в реализации, что делает его идеальным выбором для решения задачи о наибольшем паросочетании в двудольном графе.

В качестве среды разработки мною была выбрана среда

Visual Studio 2019, язык программирования – Си.

Целью данной курсовой работы является разработка программы на языке Си, который является широко используемым. Именно с его помощью в данном курсовом проекте реализуется алгоритм Куна, осуществляющий поиск наибольшего паросочетания в графе

**1 Постановка задачи**

Требуется разработать программу, которая найдет наибольшее паросочетание в двудольном графе.

Исходный граф в программе должен задаваться списком смежности.

Программа должна работать так, чтобы можно было указать размер доль двудольного графа и выбрать способ его заполнения: вручную или автоматически. После обработки этих

данных на экран должен выводится двудольный граф в виде списка смежности и наибольшее паросочетание.

Устройство ввода - клавиатура и мышь.

Программа должна иметь интуитивно понятный интерфейс для пользователя.

Многомодульность программы. Программа должна быть поделена на логические модули. Это упростит поиск ошибок при отладке и тестировании программы, а также позволит легко расширять функционал программы.

Режим работы видеосистемы - текстовый.

Устройство ввода-вывода - клавиатура и мышь. Необходимо различать и идентифицировать действия, произведенные с их помощью, это облегчит использование программы.

**2 Теоретическая часть задания**

Паросочетанием *М* называется набор попарно несмежных рёбер графа

(иными словами, любой вершине графа должно быть инцидентно не более одного ребра из *М*).

Все вершины, у которых есть смежное ребро из паросочетания (т.е. которые имеют степень ровно один в подграфе, образованном *М*), назовём насыщенными этим паросочетанием.

Мощностью паросочетания назовём количество рёбер в нём. Наибольшим паросочетанием назовём паросочетание, мощность которого максимальна среди всех возможных паросочетаний в данном графе.

**Цепью** длины *k* назовём некоторый простой путь (т.е. не содержащий повторяющихся вершин или рёбер), содержащий ровно *k* рёбер.

**Чередующейся цепью** относительно некоторого паросочетания назовём простой путь длины *k* в которой рёбра поочередно принадлежат/не принадлежат паросочетанию.

**Увеличивающей цепью** относительно некоторого паросочетания назовём чередующуюся цепь, у которой начальная и конечная вершины не принадлежат паросочетанию.

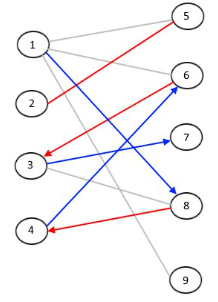


Рисунок 1 - пример

Здесь красными помечены вершины паросочетания, а в графе есть увеличивающая цепь: 1 -> 8 -> 4 -> 6 -> 3 -> 7.

С помощью увеличивающих цепей можно увеличивать паросочетание на единицу (отсюда и название). Можно взять такой путь и провести чередование - убрать из паросочетания все рёбра, принадлежащие цепи, и, наоборот, добавить все остальные. Всего в увеличивающей цепи нечетное число рёбер, а первое и последнее были не в паросочетании. Значит, мощность паросочетания увеличилась ровно на единицу.

В примере добавятся синие рёбра (1,8), (3,7) и (4,6), а удалятся красные (3,6) и (4,8). С ребром (2,5) ничего не случится - оно не в увеличивающей цепи. Таким образом, размер паросочетания увеличится на единицу.

**Алгоритм Куна** в этом и заключается - будем искать увеличивающую цепь, пока ищется, и проводить чередование в ней. Увеличивающие цепи удобны тем, что их легко искать: можно просто запустить поиск пути из произвольной свободной вершины из левой доли в какую-нибудь свободную вершину правой доли в том же графе, но в котором из правой доли можно идти только по рёбрам паросочетания (то есть у вершин правой доли будет либо одно ребро, либо ноль).

**3 Описание разработки программы**

Для программной реализации алгоритма понадобиться три массива и один список: pair (int) содержит в себе информацию о текущем паросочетании. pair[i] - это номер вершины первой доли, связанной ребром с вершиной i второй доли (или -1, если никакого ребра паросочетания из i не выходит), visited (int)- обычный массив "посещённостей" вершин в обходе в глубину (он нужен, просто чтобы обход в глубину не заходил в одну вершину дважды), used1 (int) – массив вершин первой доли, которых удалось связать паросочетанием со второй жадным алгоритмом, graph – список смежности, который показывает связь между первой долей и второй.

sizeFirstShare - число вершин в первой доле, sizeSecondShare - во второй доле.

До основного цикла алгоритма найдём произвольное паросочетание жадным алгоритмом (перебрать все вершины первой доли, и для каждой из них найти произвольное ребро, которое можно добавить в паросочетание, и добавить его.) и лишь затем будем выполнять цикл с вызовами функции dfsKuhn который будет улучшать это паросочетание. В результате алгоритм будет работать заметно быстрее на случайных графах — потому что в большинстве графов можно легко набрать паросочетание достаточно большого веса с помощью эвристики, а потом улучшить найденное паросочетание до наибольшего уже обычным алгоритмом Куна. Тем самым мы сэкономим на запусках обхода в глубину из тех вершин, которые мы уже включили с помощью эвристики в текущее паросочетание.

Дальше запускается основной алгоритм, в нем просматриваются все рёбра, исходящие из вершины v первой доли, если оно не насыщенно, и затем проверяется: если это ребро ведёт в ненасыщенную вершину to, либо если эта вершина to насыщена, но удаётся найти увеличивающую цепь рекурсивным запуском из pair[to], то мы говорим, что мы нашли увеличивающую цепь, и перед возвратом из функции с результатом 1 производим чередование в текущем ребре: перенаправляем ребро, смежное с to, в вершину v.

До основного цикла (алгоритма), а также после нахождения увеличивающей цепи (возвращение dfsKuhn 1), массив visited заполняем 0. В начале работы функции kuhnAlgorithm() массив used1 заполняем 0, а pair -1.

Ниже представлен псевдокод функции kuhnAlgorithm() и dfsKuhn().

**kuhnAlgorithm(struct Graph\* graph, int\* pair, int sizeFirstShare, int sizeSecondShare)**

1. Вызвать функцию fillArrey(used1, sizeFirstShare, 0)(заполняем нулями used1);
2. Вызвать функцию fillArrey(pair, sizeSecondShare, -1)(заполняем -1 pair);
3. Для i = 0 пока i < sizeFirstShare делать i++
   1. Пока есть следующая связь вершины i со второй долей в графе graph:
      1. v = temp->vertex (текущая вершина второй доли)
      2. если pair[v] == -1, т.е если v не связана паросочетанием, то
         1. pair[v] = i;
         2. used1[i] = 1;
         3. break

3.1. конец цикла

1. конец цикла
2. вызвать функцию fillArrey(visited, sizeFirstShare ,0) (заполняем нулями visited)
3. Для i = 0 пока i < sizeFirstShare делать i++
   1. Если used1[i], то
      1. Continue
   2. Если dfsKuhn(graph, i, visited, pair), то
      1. вызвать функцию fillArrey(visited, sizeFirstShare ,0) (заполняем нулями visited)
4. конец цикла

**dfsKuhn(**struct Graph\* graph, int vertex, int\* visited, int\* pair**)**

1. Если visited[vertex] == 1, то
   1. return 0;
2. visited[vertex] = 1;
3. temp = graph->adjLists[vertex];
4. пока temp:
   1. v = temp->vertex;
   2. Если pair[v] == -1 или (visited[pair[v]] == 0 и dfsKuhn(graph, pair[v], visited, pair) == 1, то
      1. pair[v] = vertex;
      2. return 1;
   3. temp = temp->next;
5. конец цикла

**4 Описание программы**

Файл Coursework.cpp является основным модулем программы. Работа программы начинается с запроса на ввод размеров доль графа, после чего пользователю нужно выбрать заполнение вручную или автоматически. Дальше программа выводит граф и выполняет алгоритмы. В конце программа выводит наибольшее паросочетание.

Ниже можно увидеть оформление начального запроса и дальнейшие

действия с ним.

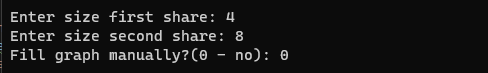


Рисунок 2 - ввод размеров доль графа и способа его построения

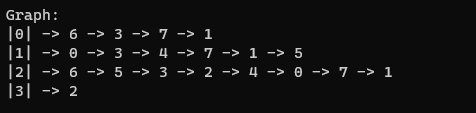


Рисунок 3 - вывод графа

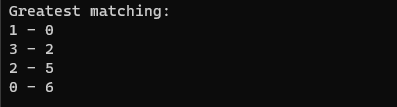


Рисунок 4 - вывод наибольшего паросочетания

Далее на рисунке 5 представлена схема взаимодействия программы.

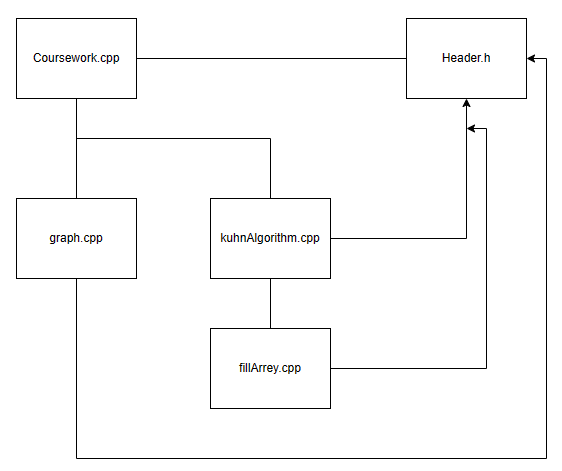


Рисунок 5-схема взаимодействия программы

Ниже, на рисунке 6 представлена схема данных.

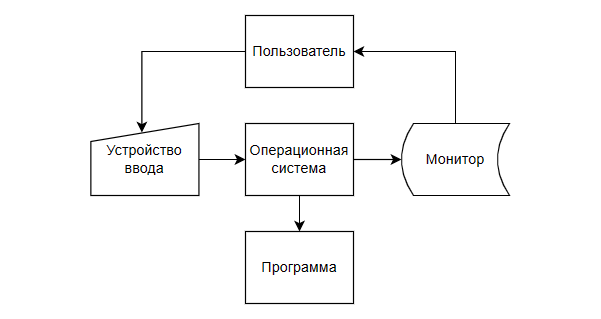


Рисунок 6 - схема данных

**5 Тестирование**

Среда разработки VisualStudio 2019 предоставляет все средства, необходимые при разработке и отладке многомодульной программы.

Тестирование проводилось в рабочем порядке, в процессе разработки, после завершения написания программы. В ходе тестирования было выявлено и исправлено множество проблем, связанных с вводом данных, изменением дизайна выводимых данных, алгоритмом программы, взаимодействием функций.

Ниже продемонстрирован результат тестирования программы при вводе пользователем различных количеств вершин и вывод наибольшего паросочетания.

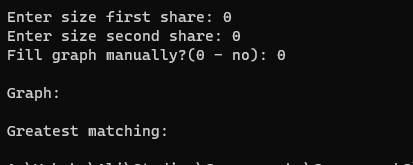


Рисунок 7 - тестирование при размерах доль 0

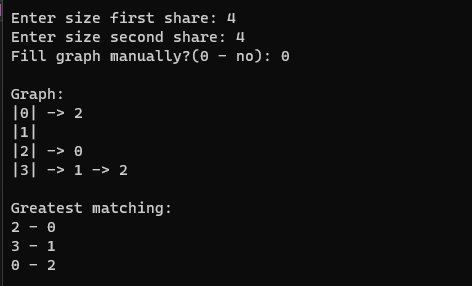


Рисунок 8 - тестирование при размерах доль 4 и 4

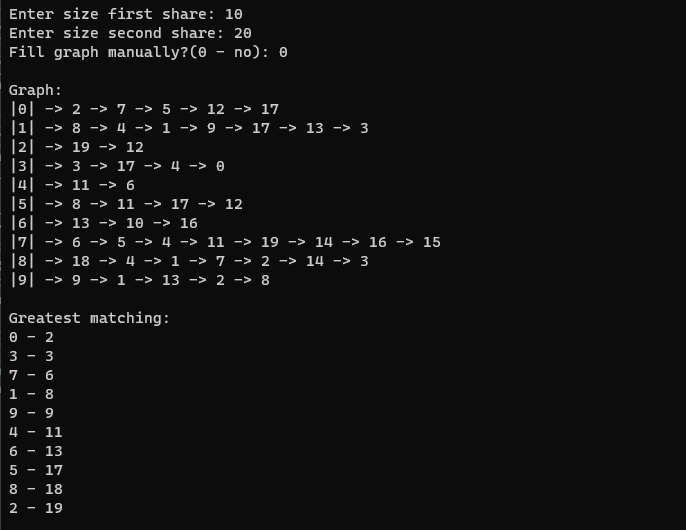


Рисунок 9 - тестирование при размерах доль 10 и 20

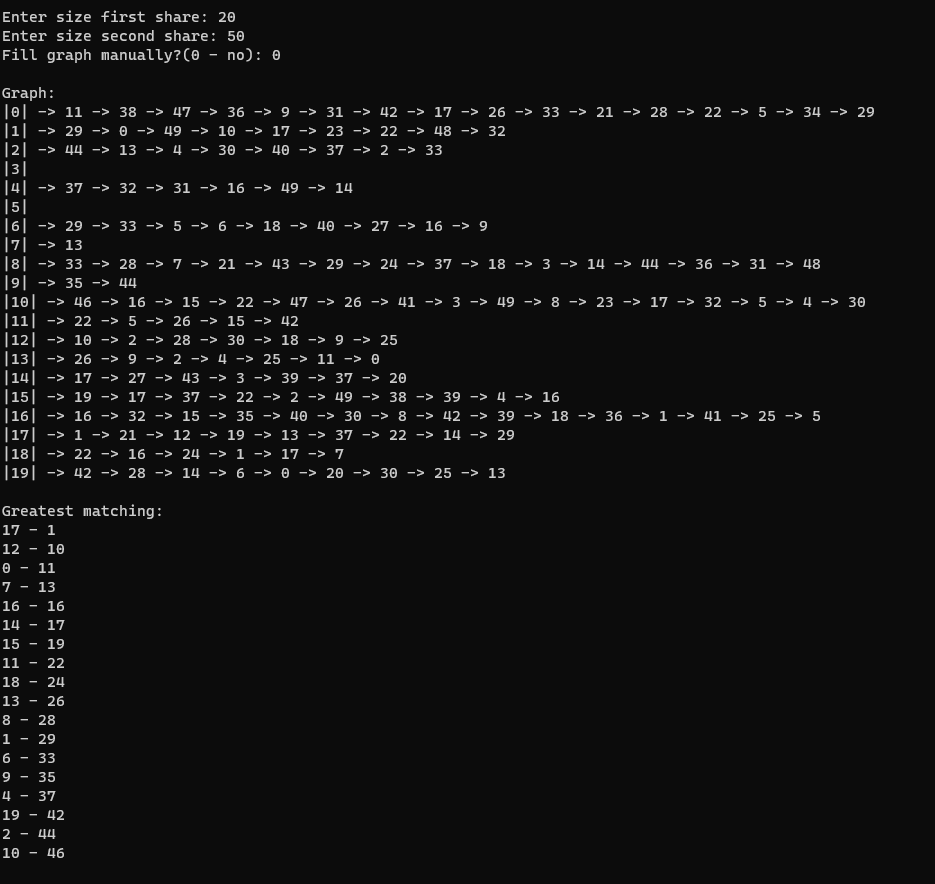


Рисунок 10 - тестирование при размерах доль 20 и 50

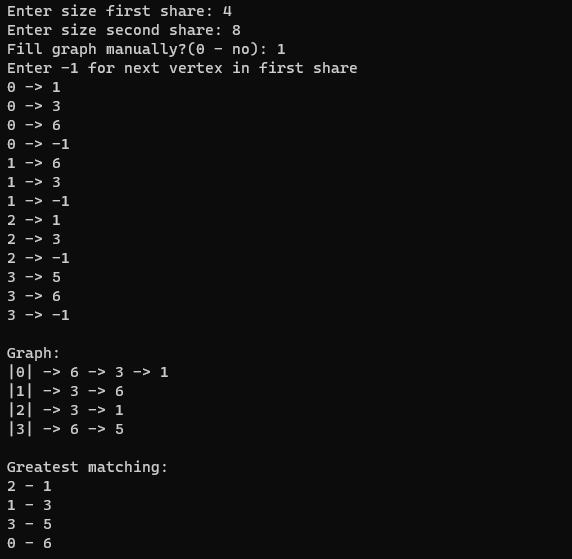


Рисунок 11 - тестирование ручного ввода

Таблица 1 - описание поведения программы при тестировании

|  |  |  |
| --- | --- | --- |
| Описание теста | Ожидаемый результат | Полученный результат |
| Запуск программы | Вывод сообщения о вводе размера доль | Верно |
| Выбор способа построения графа автоматически | Граф построится автоматически | Верно |
| Выбор способа построения графа вручную | Граф нужно будет построить вручную | верно |
| Вывод наибольшего паросочетания | Вывод верного результата | Верно |

**6 Ручной расчет задачи**

Проведем проверку программы посредством ручных вычислений на

примере графа с размером первой доли 4 и второй доли 4

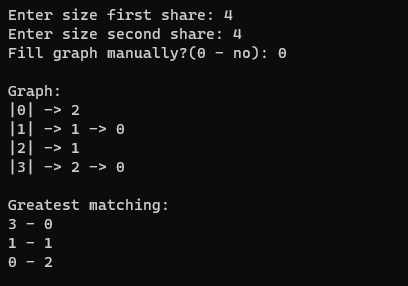


Рисунок 12 - граф и результат для ручной проверки

Если соединить последовательно каждое левое ребро с правым, что и делает жадный алгоритм до основного, то мы как раз и получим этот результат, что как раз и показывает, что таким способом до основного алгоритма мы экономим очень много времени.

**Заключение**

Таким образом, в процессе создания данного проекта разработана программа, реализующая алгоритм поиска наибольшего паросочетания в Visual Studio 2019.

При выполнении данной курсовой работы были получены навыки разработки программ и освоены приемы создания списка смежности для двудольных графов. Приобретены навыки по осуществлению алгоритма Куна. Углублены знания языка программирования Cи.

Недостатком разработанной программы является примитивный пользовательский интерфейс. Потому что программа работает в консольном режиме, не добавляющем к сложности языка сложность программного оконного интерфейса.

Программа имеет небольшой, но достаточный для использования функционал возможностей.

**Список используемых источников**

1. Мартынов, Н. Н. Программирование для Windows на C/C++. В 2 томах. Том 1 / Н.Н. Мартынов. - М.: Бином, 2013. - 528 c.

2. Керниган, Б.У. Язык программирования С / Б.У. Керниган, Д.М. Ритчи; Пер. с англ. В.Л. Бродовой. — М.: Вильямс, 2016. — 304 c.

3. Культин, Н. C/C++ в задачах и примерах / Н. Культин. - М.: БХВ-Петербург, 2022. - 368 c.

4. Дорогов, В.Г. Основы программирования на языке С: Учебное пособие / В.Г. Дорогов, Е.Г. Дорогова; Под общ. ред. проф. Л.Г. Гагарина. — М.: ИД ФОРУМ, НИЦ ИНФРА-М, 2017. — 224 c.

5. Головин, И.Г. Языки и методы программирования: Учебник для студентов учреждений высшего профессионального образования / И.Г. Головин, И.А. Волкова. — М.: ИЦ Академия, 2017. — 304 c.

6. Дейтел, П. Как программировать на С / П. Дж. Дейтел , Х. М. Дейтел. - М.: Бином, 2022. - 858 c.

7. Культин, Н. C/C++ в задачах и примерах / Н. Культин. - М.: БХВ-Петербург, 2022. - 368 c.

8. Березин, Б. И. Начальный курс С и С++ / Б. И. Березин, С. Б. Березин. – М. : Диалог– МРТИ, 1999.

9. Громов Ю.Ю., Татаренко С.И. Программирование на языке Си: учебное пособие / Ю.Ю.Громов, С.И.Татаренко – Тамбов, 1995. – 169 с.

10. Демидович, Е. М. Основы алгоритмизации и программирования. Язык Си / Е. М. Демидович. – Минск : Бест­принт, 2001.

11. Шилдт, Г. Полный справочник по C / Г. Шилдт – М.: Вильямс, 2009

12.  Аблязов, Р. Программирование на ассемблере на платформе x86-64 / Р. Аблязов. - М.: Книга по Требованию, 2011. - 302 c.

13. Тюгашев, А.А. Основы программирования. Учебное пособие. Часть I. / А.А. Тюгашев - М.: СПб: Университет ИТМО, 2016. – 160 с.

14. Калашников, Олег Ассемблер? Это просто! Учимся программировать / Олег Калашников. - М.: БХВ-Петербург, 2005. - 384 c.

15. Пирогов, Владислав Ассемблер для Windows / Владислав Пирогов. - М.: БХВ-Петербург, 2007. - 896 c.

16. Пирогов, Владислав Ассемблер на примерах / Владислав Пирогов. - М.: БХВ-Петербург, 2013. - 416 c.

17. Ирвин, К. Язык ассемблера для процессоров Intel, 4-е издание. / К. Р. Ирвин. Пер. с англ. – М.: Издательский дом «Вильямс», 2005. – 912 с.: ил.

18. Гриффитс, Д. Изучаем программирование на C / Д. Гриффитс, Д. Гриффитс. - М.: Эксмо, 2015. - 624 c.

19. Гукин, Д. Для "чайников". Программирование на C / Д. Гукин. - М.: Вильямс, 2016. - 384 c.

20. Прата, С. Язык программирования С. Лекции и упражнения, 6-е изд. / С. Прата. Пер. с англ. Ю.Н. Артеменко. – М.: ООО “И.Д. Вильямс”, 2015. — 928 с.

**Приложение А**

**(Рекомендуемое)**

**Листинги программы**

**Файл Coursework.cpp**

#include "Header.h"

int main(int argc, char\* argv[])

{

srand(time(NULL));

int sizeFirstShare = 4;

int sizeSecondShare = 8;

int manually = 0;

printf("Enter size first share: ");

scanf("%d", &sizeFirstShare);

printf("Enter size second share: ");

scanf("%d", &sizeSecondShare);

int\* pair = (int\*)malloc(sizeSecondShare \* sizeof(int));

struct Graph\* graph = createGraph(sizeFirstShare);

printf("Fill graph manually?(0 - no): ");

scanf("%d", &manually);

if (manually == 0)

fillGraphAuto(graph, sizeFirstShare, sizeSecondShare);

else

fillGraphManually(graph, sizeFirstShare, sizeSecondShare);

printGraph(graph);

kuhnAlgorithm(graph, pair, sizeFirstShare, sizeSecondShare);

printKun(pair, sizeSecondShare);

free(pair);

freeGraph(graph, sizeFirstShare);

return 0;

}

**Файл kuhnAlgorithm.cpp**

#include "Header.h"

void kuhnAlgorithm(struct Graph\* graph, int\* pair, int sizeFirstShare, int sizeSecondShare)

{

int\* used1 = (int\*)malloc(sizeFirstShare \* sizeof(int));

used1 = fillArrey(used1, sizeFirstShare, 0);

pair = fillArrey(pair, sizeSecondShare, -1);

for (int i = 0; i < sizeFirstShare; i++)

{

struct node\* temp = graph->adjLists[i];

while (temp)

{

int v = temp->vertex;

if (pair[v] == -1)

{

pair[v] = i;

used1[i] = 1;

break;

}

temp = temp->next;

}

}

int\* visited = (int\*)malloc(sizeFirstShare \* sizeof(int));

visited = fillArrey(visited, sizeFirstShare ,0);

for (int i = 0; i < sizeFirstShare; i++)

{

if (used1[i])

continue;

if (dfsKuhn(graph, i, visited, pair))

visited = fillArrey(visited, sizeFirstShare, 0);

}

free(visited);

free(used1);

}

int dfsKuhn(struct Graph\* graph, int vertex, int\* visited, int\* pair)

{

if (visited[vertex])

return 0;

visited[vertex] = 1;

struct node\* temp = graph->adjLists[vertex];

while (temp)

{

int v = temp->vertex;

if (pair[v] == -1 || (visited[pair[v]] == 0 and dfsKuhn(graph, pair[v], visited, pair)))

{

pair[v] = vertex;

return 1;

}

temp = temp->next;

}

free(temp);

return 0;

}

void printKun(int\* arr, int sizeSecondShare)

{

printf("\nGreatest matching:\n");

for (int i = 0; i < sizeSecondShare; i++)

{

if (arr[i] == -1)

continue;

printf("%d - %d\n", arr[i], i);

}

}

**Файл graph.cpp**

#include "Header.h"

void freeGraph(struct Graph\* graph, int size)

{

for (int i = 0; i < size; i++)

{

struct node\* temp = graph->adjLists[i];

while (temp)

{

struct node\* p = temp;

temp = temp->next;

free(p);

}

free(temp);

}

free(graph->adjLists);

free(graph);

}

void fillGraphAuto(struct Graph\* graph, int sizeFirstShare, int sizeSecondShare)

{

for (int i = 0; i < sizeFirstShare; i++)

{

for (int j = 0; j < rand() % sizeSecondShare; j++)

{

int v;

do

{

v = rand() % sizeSecondShare;

} while (total(graph, i, v) == 1);

addEdge(graph, i, v);

}

}

}

void fillGraphManually(struct Graph\* graph, int sizeFirstShare, int sizeSecondShare)

{

int v;

printf("Enter -1 for next vertex in first share\n");

for (int i = 0; i < sizeFirstShare; i++)

{

while (1)

{

do

{

printf("%d -> ", i);

scanf("%d", &v);

if (total(graph, i, v) == 1)

printf("such a connection already exists\n");

} while (total(graph, i, v) == 1);

if (v == -1)

break;

if (v >= sizeSecondShare)

{

printf("such vertex in second share no\n");

continue;

}

addEdge(graph, i, v);

}

}

}

struct node\* createNode(int v)

{

struct node\* newNode = (struct node\*)malloc(sizeof(struct node));

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

struct Graph\* createGraph(int vertices)

{

struct Graph\* graph = (struct Graph\*)malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = (struct node\*\*)malloc(vertices \* sizeof(struct node\*));

for (int i = 0; i < vertices; i++)

graph->adjLists[i] = NULL;

return graph;

}

void addEdge(struct Graph\* graph, int src, int dest)

{

struct node\* newNode = createNode(dest);

newNode->next = graph->adjLists[src];

graph->adjLists[src] = newNode;

}

void printGraph(struct Graph\* graph)

{

printf("\nGraph:");

for (int v = 0; v < graph->numVertices; v++)

{

struct node\* temp = graph->adjLists[v];

printf("\n|%d|", v);

while (temp)

{

printf(" -> %d", temp->vertex);

temp = temp->next;

}

}

printf("\n");

}

int total(struct Graph\* graph, int src, int vertex)

{

int count = 0;

struct node\* temp = graph->adjLists[src];

while (temp)

{

if (temp->vertex == vertex)

return 1;

temp = temp->next;

}

return 0;

}

**Файл fillArrey.cpp**

#include "Header.h"

int\* fillArrey(int\* arr, int size, int x)

{

for (int i = 0; i < size; i++)

{

arr[i] = x;

}

return arr;

}

**Файл Header.h**

#pragma once

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <string.h>

#include <string>

struct node\* createNode(int v);

struct Graph\* createGraph(int vertices);

void addEdge(struct Graph\* graph, int src, int dest);

void printGraph(struct Graph\* graph);

int dfsKuhn(struct Graph\* graph, int vertex, int\* visited, int\* pair);

void fillGraphAuto(struct Graph\* graph, int sizeFirstShare, int sizeSecondShare);

void fillGraphManually(struct Graph\* graph, int sizeFirstShare, int sizeSecondShare);

int total(struct Graph\* graph, int src, int vertex);

void printKun(int\* arr, int size);

int\* fillArrey(int\* arr, int size, int x);

void freeGraph(struct Graph\* graph, int size);

void kuhnAlgorithm(struct Graph\* graph, int\* pair, int sizeFirstShare, int sizeSecondShare);

struct node

{

int vertex;

struct node\* next;

};

struct Graph

{

int numVertices;

struct node\*\* adjLists;

};