**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №2**

**по дисциплине «Построение и анализ алгоритмов»**

Тема: Нахождение стягивающего дерева методом поиска в глубину

|  |  |  |
| --- | --- | --- |
| Студент гр. 6304 |  | Пискунов Я.А. |
| Преподаватель |  | Балтрашевич В.Э. |

Санкт-Петербург

2018

**Цель работы.**

Изучить реализацию алгоритма поиска стягивающего дерева в графе на языке программирования C++.

**Основные теоретические положения.**

Существует много алгоритмов на графах, в основе которых лежит систематический перебор вершин графа, такой что каждая вершина просматривается в точности один раз. Поэтому важной задачей является нахождение хороших методов поиска в графе. Вообще говоря, метод поиска «хорош», если

* Он позволяет алгоритму решения интересующей нас задачи легко «погрузиться в этот метод
* Каждое ребро графа анализируется не более одного раза (или, что существенно не меняет ситуации, число раз, ограниченное константой).

Общая идея метода поиска в глубину состоит в следующем. Мы начинаем поиск с некоторой фиксированной вершины, затем выбираем другую произвольную вершину, смежную с исходной и повторяем процесс от нее. Если существует новая, еще не просмотренная вершина, то мы рассматриваем ее, и она перестает быть новой. Если же не существует ни одной новой вершины, смежной с текущей, то текущая вершина становится использованной.

**Ход работы**

За основу алгоритма поиска в глубину возьмем алгоритм на языке Pascal, представленный в книге В. Липского «Комбинаторика для программистов». Этот алгоритм представлен на рис. 0.

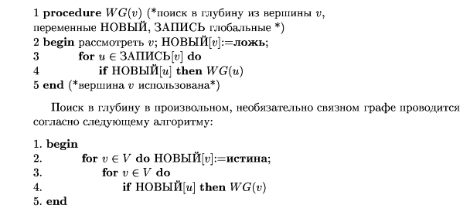


Рисунок 0 – алгоритм из книги

Работать будем в Qt Creator, так как данная программа предоставляет широкий функционал, который значительно упростит создание программы. В дальнейшем будут рассмотрены файлы, созданные непосредственно «руками». Все автоматически созданные файлы, а также файлы, которые необходимы для графического отображения результата, но не для обработки алгоритма, в данный отчет не включены ввиду их громоздкости и отсутствия критической необходимости в их описании.

Так как сам алгоритм прост в реализации и не требует особых усилий по реализации, то включим его в программу для поиска гамильтоновых путей как дополнительную функцию. Для этого сначала модифицируем интерфейс, добавив кнопку для старта выполнения алгоритма. Также модифицируем сам вывод графа, вставив его непосредственно в окно программы. Это позволит пошаговому алгоритму поиска гамильтоновых путей выглядеть более красиво. Новый интерфейс представлен на рис. 1.

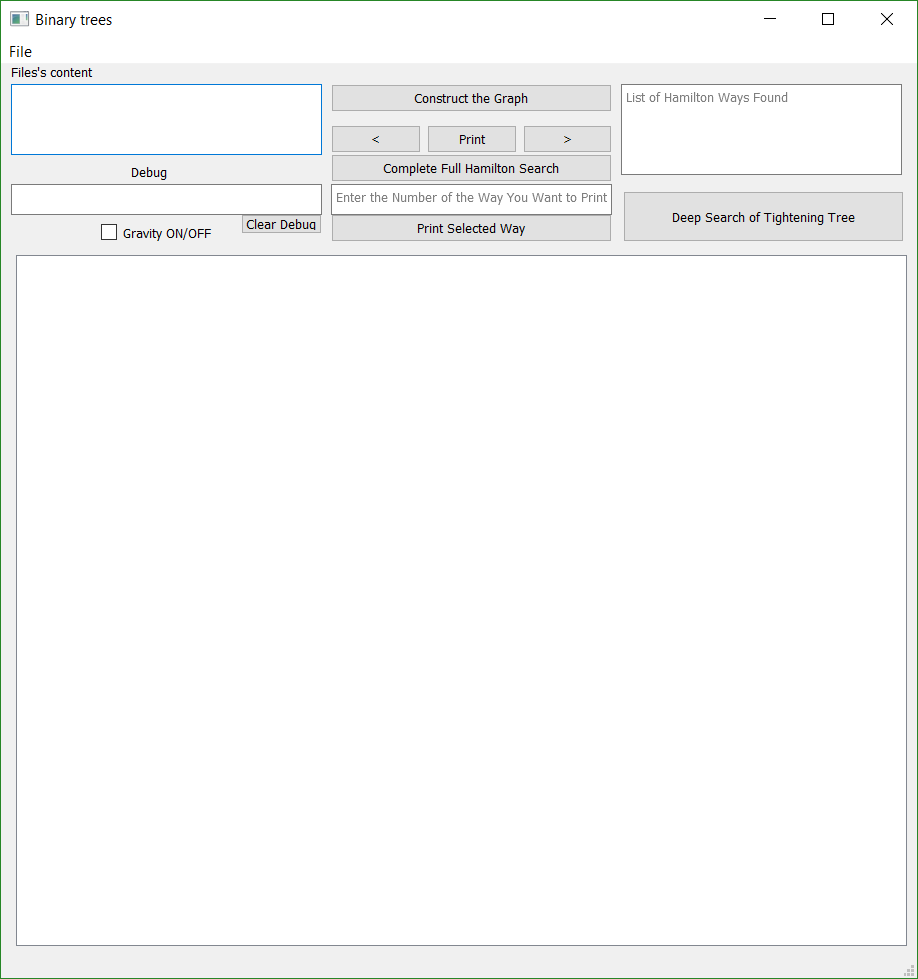


Рисунок 1 – новый интерфейс программы

Как мы можем видеть, запуск необходимого алгоритма происходит по нажатию кнопки «Deep Search of Tightening Tree».

Все функции по интерпретации данных из файла в список инцидентности не подверглись изменениям, так что не будем заострять на них внимание.

По нажатию кнопки осуществляется инициализация алгоритма. Для его работы в код были добавлены новые глобальные переменные:

* tTree – переменная типа INCTR\*, хранящая указатель на массив структур, которые представляют собой список инцидентности стягивающего дерева.
* filename – переменная типа QString, которая, как и раньше, хранит путь к файлу. Она сделана глобальной, так как во время работы алгоритма, данные о дереве заменяются, что приводит к ошибкам при попытке повторного построения графа. При таком подходе данные из файла перепроверяются каждый раз, когда необходимо построить граф.

В структуру INCTR добавлено поле label типа boolean, которое является индикатором того, была ли вершина уже обработана.

Перейдем непосредственно к инициализации алгоритма. После нажатия на кнопку происходит выделение памяти под tTree. После этого в tTree копируется информация из Tree и все указатели на инцидентные вершины отсекаются. Таким образом мы получаем массив вершин, который в будущем станет списком инцидентности. После происходит непосредственно запуск самого алгоритма и, после его завершения, вывод полученного дерева на экран. Реализация инициализатора представлена в приложении А.

Сам алгоритм реализован рекурсивно. На вход подается номер текущей вершины в массиве инцидентности. По нему можно получить доступ ко всей информации о вершине. Используя его, мы меняем значение поля label в нужной структуре на true. После этого создаем копию графа (переменная Copy), чтобы не испортить граф во время обработки. Далее алгоритм проходит по всем инцидентным вершинам текущей и проверяет поля label в их структурах. В случае, если вершина еще не была обработана, то рассматриваемое ребро добавляется к дереву и алгоритм начинается от каждой такой инцидентной вершины рекурсивно. В конце в copy восстанавливается указатели, как в первоначальном графе и указатель на текущий элемент. Реализация алгоритма представлена в приложении Б.

Перейдем к тестированию программы. Для начала попробуем запустить алгоритм без входных данных. Результат запуска представлен на рис. 2.

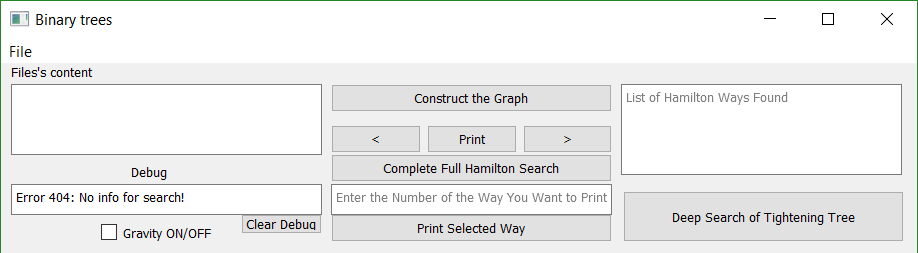


Рисунок 2 – запуск алгоритма без входных данных.

Как мы видим, алгоритм не запустился, а в окне «Debug» появилось сообщение об ошибке.

Теперь откроем файл со входными данными. Граф, который получен путем их обработки представлен на рис. 3.

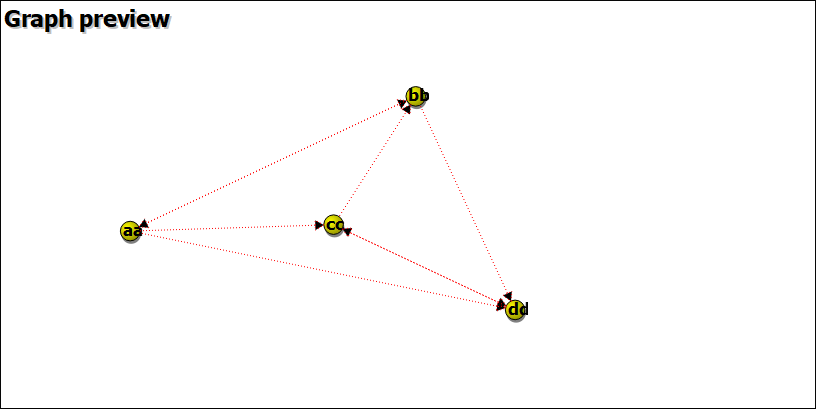


Рисунок 3 – исходный граф для алгоритма

Теперь запустим алгоритм и проверим его работоспособность. Результат представлен на рис. 4.

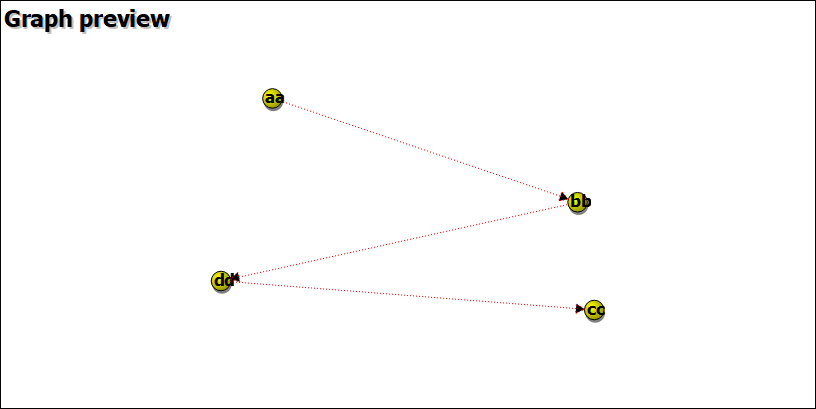


Рисунок 4 – полученное стягивающее дерево

Данный результат соответствует ожидаемому. Алгоритм работает правильно.

**Выводы.**

В ходе данной работы была изучена реализация алгоритма поиска стягивающего дерева методом поиска в глубину в графе.

Приложение А

Инициализация алгоритма

void MainWindow::**on\_pushButton\_8\_clicked**()

{

tree = ReadFile(fileName, ui->textEdit);

tTree = new INCTR[peaks\_i];

for (int i=0; i<peaks\_i; i++)

{

tTree[i] = tree[i];

tTree[i].deg = 0;

tTree[i].next = NULL;

}

if (tree!=NULL)

{

dfst(0);

globalWidget = new GraphWidget(this, tTree, &gravity, peaks\_i, NULL);

ui->graphicsView->setViewport(globalWidget);

}

else ui->textEdit\_3->append("Error 404: No info for search!");

}

Приложение Б

Реализация алгоритма

void **dfst** (int nodeTurn)

{

tree[nodeTurn].label = true;

INCTR\* copy = new INCTR[peaks\_i];

for (int i=0; i<peaks\_i; i++)

copy[i] = tree[i];

INCTR\* temp = copy[nodeTurn].next;

while (temp!=NULL)

{

if (tree[(temp->turn)-1].label == false)

{

INCTR\* moreTemp = tTree[nodeTurn].next;

if (moreTemp==NULL)

{

tTree[nodeTurn].next = new INCTR;

tTree[nodeTurn].next->name = temp->name;

tTree[nodeTurn].next->turn = temp->turn;

tTree[nodeTurn].next->deg = 1;

}

else

{

while (moreTemp->deg != 1)

moreTemp = moreTemp->next;

moreTemp->deg = 0;

moreTemp->next = new INCTR;

moreTemp->next->name = temp->name;

moreTemp->next->turn = temp->turn;

moreTemp->next->deg = 1;

}

dfst((temp->turn)-1);

}

for (int i=0; i<peaks\_i; i++)

copy[i] = tree[i];

QString stopName = temp->name;

temp = copy[nodeTurn].next;

while (temp->name!=stopName)

temp = temp->next;

temp = temp->next;

}

}