Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

ОТЧЕТ ПО ПРАКТИКЕ — ТЕОРИЯ ГРАФОВ

РЕФЕРАТ

студента 3 курса 351 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Григорьева Алексея Александровича

Проверил
доцент, к. ф.-м. н. _____ М. С. Портенко

Саратов 2018

СОДЕРЖАНИЕ

1 Описание вспомогательных классов

Для решения задач были созданы вспомогательные классы: Node, EdgeTo, G_Graph

1.1 Описание класса Node

Класс Node моделирует вершину(узел) графа. Поля и методы:

- string m_id — имя вершины. Геттер — public string GetName().
- Color m_color — цвет вершины для обходов графа. Геттер — public Color GetColor() и сеттер — public void SetColor(Color color). Color — перечисляемый тип, принимающий значения WHITE, GREY, BLACK.

Определены 2 конструктора:

- public Node(string id) — создание узла с присвоением заданного имени
- public Node(Node node) — создание узла-копии заданного узла

1.2 Описание класса EdgeTo

Класс EdgetO моделирует ребро или дугу графа из какой-то вершины. Поля и методы:

- Node m_nodeTo — вершину, в которую проведено ребро или узел. Геттер — public Node GetNodeTo().
- int m_weight — вес дуги. Геттер — public int GetWeight() и сеттер — public void SetWeight(int weight).
- (Для графа-сети) public int capacity — пропускная способность дуги.
- (Для графа-сети) public int flow — поток в данной дуге.

Определены 4 конструктора:

- public EdgeTo(Node to) — создание ребра, ведущего в вершину to;
- public EdgeTo(Node to, int weight) — создание дуги, ведущей в вершину to с весом weight;
- public EdgeTo(Node to, int capacity, int flow) — создание дуги в графе-сети, ведущей в вершину to с заданной пропускной способностью и потоком.
- public EdgeTo(EdgeTo edgeTo, Node nodeTo) — создание ребра-копии, которое ведет в заданную вершину nodeTo.

1.3 Описание класса G_Graph

Класс G_Graph моделирует граф. Поля:

— private Dictionary<Node, List<EdgeTo» m_graph — список смежности, каждой вершине приписывается список ребер.
— private bool m_oriented — ориентированный ли граф
— private bool m_weighted — взвешенный ли граф
— private bool m_network — является ли граф сетью
— private InputMode m_inputMode — режим считывания при вводе графа, подробнее будет описано при рассмотрении конструкторов графа.
— public Node source get; set; — исток графа-сети
— public Node sink get; set; — сток графа-сети

Определены 3 конструктора:

— public G_Graph() — конструктор по умолчанию, создает пустой граф.
— public G_Graph(G_Graph graph, GraphCopyMode graphCopyMode) — конструктор копирования, GraphCopyMode — перечисляемое значение, может быть: STANDARD — граф-полная копия исходного, INVERSE — дополнение графа, VERTEX_ONLY — копирование только вершин.
— public G_Graph(string filename) — считывание графа из файла 1 по имени в папке с исполняемым файлом.

1.3.1 Считывание графа с файла

Для удобства пользования программой была разработана особая система ввода графов. Доступен следующий набор команд:

— SET OPTION позволяет указывать настройки графа: ориентированность, наличие весов у ребер, а также является ли граф сетью. По умолчанию граф неориентированный
— SET VERTEX добавляет перечисленные вершины в графе по их именам.
— SET EDGE проводит ребра между двумя вершинами, при этом от типа графа имеется возможность указать дополнительные значения, принадлежащие данному ребру (вес, поток, пропускная способность)
— SET ACTION позволяет проводить дополнительные действия над графом, такие как: добавить ребро или вершину, удалить ребро или вершину, указать источник и сток.
— SET END завершает работу с графом, останавливая считывание ввода

```
// Каждая команда вводится на отдельной строке
// Доступные опции: orient, weight, network, name <имя>
SET OPTION
network
// вершины могут идти как через пробел, так и каждая на своей строке
SET VERTEX
A B C D E F G
// вершина, а затем через пробел смежная вершина, 'w' + вес
// 1 5 w2
// 3 node1 w500
// 1 3 c5 f3 - для сетей
SET EDGE
A B c3 f0
A D c3 f0
B C c4 f0
C A c3 f0
C D c1 f0
C E c2 f0
D E c2 f0
D F c6 f0
E G c1 f0
F G c9 f0
// 1 1
// 5 5
// Добавление, удаление
// ADDV
// ADDE
// SETSOURCE -- не забыть!
// SETSINK -- не забыть!
SET ACTION

SETSOURCE A
SETSINK G

SET END
```

Рисунок 1 – Пример входного файла для графа

1.3.2 Методы для работы с графом

В ходе решения задач были разработаны следующие методы (краткое описание):

- **public List<Node> NodesByCondition(NodeSearchCondition nodeCondition, string conditionString)** – возвращает список вершин по заданному условию, которое принимает дополнительный аргумент с описанием условия. NodeSearchCondition принимает значения NOTADJACENT – несмежные вершины и HASLOOP – имеющие петли.
- **public List<Node> NodesByCondition(NodeSearchCondition nodeCondition)** – как предыдущий, только не принимает дополнительный аргумент.

- **public void WriteToConsole()** — вывод графа в консоль
- **public void DeleteNode(Node nodeToDelete)** — удаление узла и инцидентных ему ребер.
- **public void DeleteEdgeFromTo(Node nodeFrom, Node nodeTo)** — удалить ребро из вершины nodeFrom в nodeTo
- **private static bool NodeToNameSame(EdgeTo edgeTo, string name)** — условие для поиска узла с данным именем
- **public bool SearchNodeByName(string name, out Node toReturn)** — поиск узла с данным именем
- **private void AddEdge(Node node, EdgeTo edge)** — добавление ребра или дуги (в зависимости от типа графа)
- **private void InsertNode(Node node)** — добавление узла
- **public InputMode GetState()** — считать, в каком состоянии находится граф
- **public Dictionary<Node, List<EdgeTo» GetGraph()** — получение списка смежности для каждой вершины
- **public static G_Graph Boruv(G_Graph DFSgraph)** — алгоритм Борувки построения минимального остовного дерева.
- **private static void DFS_AddToList(Node visitNode, Dictionary<Node, List<EdgeTo» graph, ref List<Node> blackNodes)** — алгоритм поиска в глубину с добавлением посещенных вершин в список
- **private static bool DFS_Visit_Boruv(Node visitNode, G_Graph dfsGraph, out EdgeTo minEdgeTo)** — обход в глубину с возвращением минимального ребра в компоненте связности
- **public static bool DFS_HasAcycleRoot(G_Graph DFSgraph)** — обход в глубину с определением наличия корня ациклического графа
- **private static void DFS_Init(Dictionary<Node, List<EdgeTo».KeyCollection graph)** — инициализация поиска в глубину, обнуление вершин
- **private static void DFS_Visit(Node visitNode, G_Graph dfsGraph)** — посещение вершины (используется в алгоритме поиска в глубину)
- **public static Dictionary<Node, Node> BFS_Prev(G_Graph g_Graph, Node startFrom)** — поиск в ширину с возвращением отображения, где каждой вершине сопоставляется предок в порядке обхода.
- **public static void BFS_Visit_Route(G_Graph g_Graph, Node startFrom,**

ref Dictionary<Node, Node> toFrom, ref Dictionary<Node, bool> visited, ref Queue<Node> nodesQueue) — обход в ширину, возвращающий путь до вершины

- **public static Dictionary<Node, Dictionary<Node, KeyValuePair<Node, int> > > FloydShortestRoutes(G_Graph g_Graph)**— Нахождение кратчайших путей для всех пар вершин алгоритмом Флойда.
- **public static Dictionary<Node, KeyValuePair<Node, int» FordBellmanShortestF root, G_Graph g_Graph)** — нахождение всех кратчайших путей из данной вершины
- **public void Find_bridges()** — инициализация поиска мостов в графе
- **private void dfs_Bridges(int v, int p = -1)** — обход в глубину для поиска мостов в графе
- **private void CreateProjectionToMatrix()** — создание отображения множества вершин на матрицу чисел от 1 до N = кол-во вершин в графе
- **public void NetworkEdmondsKarp()** — нахождение максимального потока в графе
- **private static int FindLowestCapacity(G_Graph graph, Dictionary<Node, Node> preds)** — нахождение минимальной пропускной способности по найденному с помощью BFS пути от истока к стоку

2 Решение задач с использованием графов

Полный код каждой написанной функции можно увидеть в приложении А

2.1 Вывести на экран все вершины графа, не смежные с данной

Для того чтобы решить данную задачу необходимо проверить список смежности каждой другой вершины графа. Если отсутствует ребро или дуга в данную вершину, то необходимо добавить её в ответ.

Это можно сделать вызвав функцию NodesByCondition(NodeSearchCondition.NO "имя заданной вершины").

2.2 Вывести на экран все вершины графа, у которых есть петли.

Для решения поставленной задачи можно проверить наличие в списке смежности каждой вершины ребра или дуги в себя.

За это отвечает функция NodesByCondition(NodeSearchCondition.HASLOOP), возвращающая список вершин, удовлетворяющих условию

2.3 Вывести список смежности графа, являющегося дополнением данного графа.

Для выполнения поставленной задачи необходимо изначально скопировать только вершины, а затем запомнить все ребра в исходном графе, дополнение которого надо найти. При отсутствии потенциального ребра в списке запомненных, необходимо провести его в графе-копии.

Этот подход взят за основу в конструкторе копирования G_Graph(G_Graph graph, GraphCopyMode.INVERSE), на выходе которого получается граф-дополнение

2.4 Определить, имеет ли данный ацикличный орграф корень.

Корнем ацикличного орграфа называется такая вершина u, что из неё существуют пути в каждую из остальных вершин орграфа. Другими словами, обход из этой вершины пройдет по всем вершинам. Воспользуемся данной логикой, запуская обход в глубину от каждой вершины и считая количество пройденных, предварительно восстанавливая всем вершинам белый цвет.

Для решения поставленной задачи была разработана функция public

static bool DFS_HasAcycleRoot(G_Graph DFSgraph), запуск которой дает ответ на данный вопрос.

2.5 Распечатать самый короткий из путей от u до остальных вершин.

Самым коротким путем будет являться найденный с помощью обхода в ширину. Сохраним предков для каждой вершины до u. После обхода сделаем рекурсивный вывод путей от заданной вершины до u.

Решить поставленную задачу можно с помощью функции Dictionary<Node, Node> BFS_Prev(G_Graph g_Graph, Node startFrom), возвращающей предка в кратчайшем пути

2.6 Требуется найти в графе каркас минимального веса с помощью алгоритма Борувки.

Алгоритм Борувки заключается в следующем:

1. Из графа строится лес.
2. Выбираются компоненты связности с минимальным количеством узлов. К ним добавляется ребро исходного графа с минимальным весом, тем самым происходит слияние двух компонент связности. Запускаем обход в глубину от вершины компоненты связности, к которой добавили ребро.
3. Если остались необработанные компоненты связности, повторяем шаг 2.
4. Пока количество компонент связности больше 1, все вершины помечаются непройденными, переход к шагу 2.
5. На выходе получаем граф-каркас минимального веса.

Примечание: компоненты связности определяются с помощью обхода в глубину.

Данный алгоритм реализован в функции public static G_Graph Boruv(G_Graph DFSgraph).

ПРИЛОЖЕНИЕ А

Приложение

Отрывок программы, используемый для решения поставленной задачи.

IIIIIIIIIIII IIIIIIIIIIII IIIIIIIIIIII IIIIIIIIIIII IIIIIIIIIIII
IIIIIIIIIIII IIIIIIIIIIII IIIIIIIIIIII
IIIIIIIIIIII IIIIIIIIIIII