DOI: 10.15514/ISPRAS-2019-31(4)-6

# Проектирование интерфейсов классов графовой модели нейронной сети

[1] Ю.Л. Карпов, ORCID: 0000-0001-6844-5530 <y.l.karpov@yandex.ru>
[2] И.А. Волкова, ORCID: 0000-0002-9869-7154 <irina.a.volkova@gmail.com>
[2] А.А. Вылиток, ORCID: 0000-0001-8643-873X <alexey.vylitok@gmail.com>
[3,2] Л.Е. Карпов, ORCID: 0000-0001-6400-8325 <mak@ispras.ru>
[4,5] Ю.Г. Сметанин, ORCID: 0000-0001-8828-0091 <ysmetanin@rambler.ru>
[1] ООО Люксофт Профешнл,
123060, Россия, г. Москва, 1-й Волоколамский проезд, 10
[2] Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1.
[3] Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.
[4] Федеральный исследовательский центр «Информатика и управление» РАН
119333 Москва, ул. Вавилова, д. 44, кор. 2.
[5] Московский физико-технический институт (технический университет)
141701 Московская область, г. Долгопрудный, Институтский пер., 9

**Аннотация**. Описывается подход к тестированию искусственных нейронных сетей, реализованный в программе на языке Си++ в виде набора структур данных и алгоритмов их обработки. В качестве структур данных используются классы языка Си++, реализующие работу с такими объектами, как вершина графа, ребро, ориентированный и неориентированный граф, остовное дерево, цикл. Приводятся интерфейсы важнейших перегруженных операций над используемыми объектами и тестирующих процедур. Дан пример реализации одной из тестирующих процедур, использующий перегруженные операции над используемыми объектами.

**Ключевые слова:** искусственная нейронная сеть; эвристика; тестирование нейронной сети; теория графов; граф; вершина; ребро; остовное дерево; цикл в графе; отрицательный цикл; устойчивость нейронной сети.

**Для цитирования:** Карпов Ю.Л., Волкова И.А., Вылиток А.А., Карпов Л.Е., Сметанин Ю.Г. Проектирование интерфейсов классов графовой модели нейронной сети. Труды ИСП РАН, том 31, вып. 4, 2019 г., стр. 97-112. DOI: 10.15514/ISPRAS-2019-31(4)-6

**Благодарности.** Авторы выражают благодарность Российскому фонду фундаментальных исследований, который поддерживает проекты № 18-07-0697-а, № 18-07-01211-а, № 19-07-00321-а, № 19-07-00493-а. Авторы также благодарны доценту кафедры системного программирования факультета ВМК МГУ им. М. В. Ломоносова Виктору Васильевичу Малышко, оказавшему помощь в проверке и уточнении диаграммы классов графовой модели.





# Designing classes' interfaces for neural network graph model

[1] Yu.L. Karpov, ORCID: 0000-0001-6844-5530 <y.l.karpov@yandex.ru>
[2] I.A. Volkova, ORCID: 0000-0002-9869-7154 <irina.a.volkova@gmail.com>
[2] A.A. Vylitok, ORCID: 0000-0001-8643-873X <alexey.vylitok@gmail.com>
[3,2] L.E. Karpov, ORCID: 0000-0001-6400-8325 <mak@ispras.ru>
[4,5] Yu.G. Smetanin, ORCID: 0000-0001-8828-0091 <ysmetanin@rambler.ru>
[1] Luxoft Professional LLC,
10, 1-st Volokolamsky proezd, Moscow, 123060, Russia
[2] Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.
[3] Ivannikov Institute for System Programming of RAS,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
[4] Federal Research Center "Informatics and Control" of RAS,
44, korp. 2, Vavilova st., Moscow, 119333, Russia
[5] Moscow Institute of Physics and Technology,
9, Institutskii per., Dolgoprudnyi, Moscow region, 141700, Russia

**Abstract.** An approach to testing artificial neural networks is described. The model of neural network is based on graph theory, and operations that are used in theoretical works devoted to graphs, trees, paths, cycles, and circuits. The methods are implemented in a C++ program in the form of a set of data structures and algorithms for their processing. C++ classes are used as data structures for implementing the processing of such objects as a graph vertex, edge, oriented and undirected graph, spanning tree, circuit. Lists of standard methods (constructors and destructors, different assigning operations) are given for all classes. Additional operations are represented in details, and among them – adding one graph to another graph, adding an edge to a graph, removing edges and vertices from graph, normalizing graph, and some more. Many different searching operations are offered. Variants of graph sorting operations are also included into graph model, some of them are similar to array sorting algorithms, and some are more specific. Above these low-level operations several more complex operations are considered as graph model components. These operations include building spanning tree for arbitrary graph, building cograph for spanning tree of a graph, discovering circuits in a graph, evaluating of circuit sign, and so on. Examples of the interfaces of the most important overloaded operations on the objects used are given. An example is given of implementation of one of testing procedures where overloaded operations of graph model objects are used.

**Keywords:** artificial neuron network; heuristic; neuron network testing; graph theory; graph; vertex; edge; spanning tree; circuit in graph; negative circuit.

**For citation:** Karpov Yu.L., Volkova I.A., Vylitok A.A., Karpov L.E., Smetanin Yu.G. Designing classes' interfaces for neuron network graph model. Trudy ISP RAN/Proc. ISP RAS, vol. 31, issue 4, 2019. pp. 97-112 (in Russian). DOI: 10.15514/ISPRAS-2019-31(4)-6

**Acknowledgments.** The authors are grateful to the Russian Foundation for Basic Research, which supports projects No. 18-07-0697-a, No. 18-07-01211-a, No. 19-07-00321-a, No. 19-07-00493-a. The authors are also grateful to the associate professor Viktor Vasilyevich Malyshko of the system programming department of the CMC faculty of Lomonosov Moscow State University, who assisted in the verification and refinement of the class diagram of the graph model.

## *1. Введение*

Одним из важнейших мероприятий при использовании искусственных нейронных сетей для решения задач из самых разных прикладных областей, будь то задачи классификации, оптимизации, машинного перевода, обработки и анализа изображений в режиме реального времени [1–3], обработки сигналов [4] или ещё какие-нибудь, являются работы по тестированию самих нейронных сетей. Процесс тестирования нейронных сетей существенно отличается от процесса тестирования программного обеспечения, что не в последнюю очередь связано с трудностями отслеживания движения данных в нейронных сетях. Авторы

уже сформировали и описали в общем виде предлагаемый ими эвристический подход к тестированию искусственных нейронных сетей (см. [5]). Однако понятно, что верность эвристического подхода не может быть подтверждена одними только теоретическими рассуждениями. Её надо демонстрировать на реальных примерах.

Прежде, чем переходить к работе с искусственными сетями, используемыми при решении практических задач, необходимо разработать программную модель, способную выявлять некоторые важные свойства таких сетей. В настоящее время используются самые разнообразные структуры искусственных нейронных сетей [6-9], из чего следует, что включение классов объектов в нейросетевые модели может производиться на основе учёта очень разных видов признаков. К наиболее содержательным основаниям выбора тех или иных систем классов можно отнести такие различия в структурах сетей:

- по наличию или отсутствию в сети обратных связей;
- по рекуррентному или нерекуррентному режиму работы;
- по области значений параметров (дискретные или аналоговые);
- по времени (дискретное и непрерывное);
- по режиму обновления параметров (с синхронизацией и без неё);
- по обучению с учителем и без учителя (есть также смешанные стратегии);
- по детерминированному или вероятностному режиму работы;
- наконец, просто по размеру – от малых сетей до глубоких, содержащих сотни слоёв нейронов.

Естественно в качестве модели, способной описать практически наблюдаемое многообразие нейронных сетей, выбирать модель, основанную на понятии графа, что даст возможность пользоваться методами теории графов [10-12].

## *2. Искусственные нейронные сети и графы*

Подход к построению модели тестирования на основе представления искусственной нейронной сети в виде графа в данном случае обладает дополнительным преимуществом. Он позволяет отвлечься от излишней детализации описания вычислительных узлов сети и сосредоточиться на исследовании структурных свойств и особенностей конкретных структур, встречающихся в большинстве практически используемых нейронных сетей.

Упомянутые особенности имеют критическое влияние на многие свойства, демонстрируемые нейронными сетями при решении задач с их использованием. Структуры одних (вполне успешно применяемых на практике) сетей радикально отличаются от структуры других (не менее успешных).

Такие факторы, как наличие или отсутствие промежуточных слоёв, их количество, могут серьёзно повлиять на способность строящейся нейронной сети решать те или иные классы задач, а иногда могут накладывать определённые ограничения на количественные параметры допустимых к решению задач. Столь же влиятельной является структура этих слоёв – наличие в них внутрислойных связей, интенсивность передач информации от слоя к слою, наличие обратных связей, как положительных, так и отрицательных.

Все вместе эти и другие важные структурные свойства сети могут влиять и на саму возможность получения решения конкретной задачи, и на устойчивость этого решения, то есть сохранение основных свойств решения при относительно малых искажениях входных данных.

Для всех искусственных нейронных сетей характерно наличие входного и выходного слоя нейронов, которые либо непосредственно взаимодействуют друг с другом, либо осуществляют это взаимодействие через промежуточные слои, количество которых определяется поставленной задачей. Наибольший интерес для исследования структурных





особенностей нейронных сетей представляют как раз промежуточные слои, что определяется и большим их количеством, и сложностью межнейронных связей.

Именно поэтому авторы в первую очередь решили начать строить свою модель для промежуточных слоёв искусственных нейронных сетей, и в прежде всего для сетей, в которых промежуточные слои образуют не простейшие линии передачи информации от входного слоя к выходному, что характерно для свёрточных сетей, а сложные переплетающиеся структуры с обратными связями, в которых практически невозможно отследить пути распространения данных.

## *3. Автоматизация тестирования через предварительное создание средств работы с графами*

Тестирование искусственной нейронной сети представляет собой сложный эмпирический процесс подбора или подтверждения правильности подбора многочисленных параметров. Такие важные тестируемые параметры есть у всех нейронных сетей, но у сетей с разной структурой промежуточных слоёв они разные. Например, при тестировании такой нейронной сети, как сеть Хопфилда [13], в которой все слои являются в некотором смысле промежуточными, являясь одновременно и входными, выходными, требуется исследовать начальные значения коэффициентов матрицы весов, векторы порогов срабатывания нейронов, метрики, позволяющие сравнивать выходы сети с обучающими выходными образцами, вид сигмоидальной функции, используемой при формировании выходных векторов на основе матрицы весов и значений входных векторов.

При тестировании машин Больцмана [14] необходимо обращать внимание на следующие параметры:

- начальное значение искусственной температуры (если выбранное значение оказывается неверным, алгоритм может отдавать предпочтение неверным решениям);
- подмножество обучающих векторов, которое будет использовано при настройке сети на решение поставленной задачи;
- метод изменения весов и целевых функций, влияющий на скорость сходимости процесса (количество шагов, которое выполняется сетью) и качество решения (близость найденного локального минимума целевой функции к её глобальному минимуму);
- способ определения необходимости коррекции веса исследуемого синапса, то есть разработки компаратора весов.

Пытаясь разобраться с другими видами нейронных сетей, кроме уже упомянутых, можно встретить такие подлежащие тестированию и настройке параметры, как количество промежуточных слоёв и алгоритмы уменьшения размерности (вычисления максимального, минимального, среднего, медианного значений), а также множество других параметров, каждый из которых характерен для конкретных архитектур нейронной сети. Задача, которую решают при выборе нужной архитектуры, связана с получением конкретных рекомендаций по выбору комбинации свойств сети, способной содействовать поискам решения прикладной проблемы, стоящей перед пользователями нейронной сети. Разработка методов автоматизированного тестирования сетей позволит вплотную приблизиться к пониманию методики выработки подобных рекомендаций.

### 3.1 Графовая модель искусственной нейронной сети, графы ориентированные и неориентированные

Учитывая огромное разнообразие видов нейронных сетей, наиболее приемлемым выбором модели, которую можно было бы использовать при тестировании нейронных сетей, можно считать модель, основанную на теории графов, которая могла бы оперировать такими объектами:

- вершина графа;
- ребро графа;
- набор вершин графа;
- набор рёбер графа;
- граф, как совокупность вершин и рёбер;
- остовное дерево графа;
- цикл (простой замкнутый путь) в графе: все рёбра и вершины в цикле различны;
- матрица смежности вершин графа;
- матрица инциденции;
- матрица весов.

Фактическая объектно-ориентированная декомпозиция прикладной области уже была проделана усилиями многих математиков всего мира, и для программной реализации было естественно выбрать объектно-ориентированный язык программирования. В настоящее время объектно-ориентированный подход к программированию стал довлеющей концепцией при построении моделей реального мира, а также абстрактных построений, примером которых являются объекты, встречающиеся при решении задач теории графов. Существует огромное множество объектно-ориентированных языков программирования, из которых авторами был выбран язык Си++. Причин для такого выбора было множество – от объективных, связанных с наличием эффективных систем программирования и возможностью работать на разном оборудовании и в разном операционном окружении, до субъективных, возникших в связи личными преференциями авторов.

При оценке методов реализации графовой модели нейронной сети их эффективность рассматривалась авторами с учётом того, что эта модель должна использоваться для построения инструментальных программных средств. На практике при работе с графами часто возникают задачи с разными уровнями вычислительной сложности – от задач, разрешимых за линейное время, до NP-полных задач [12]. Это приводит к постоянной борьбе за снижение требований к вычислительным ресурсам, которые используются нейронными сетями. Проведение подготовительных мероприятий (анализ архитектуры сети, определение свойств отдельных компонентов сети, имеющихся в ней циклов, их весов и знаков) тоже приводят к необходимости построения вычислительно сложных, часто переборных алгоритмов. Однако тот факт, что подготовка и улучшение структуры сети проводятся на модельных данных на предварительном этапе, то есть ещё до начала решения основной задачи, позволяет менее требовательно относиться к сложности используемых алгоритмов, то есть в некоторых случаях выбор между сложностью алгоритма при исполнении и временем его реализации делался в последнего.

Дополнительным аргументом в пользу Си++ послужила та естественность, с которой в этом языке допускается использование концепции абстрактных типов данных.

## 3.2 Структура данных – классы и их взаимодействие (агрегация и наследование)

При выборе структур данных, которые должны использоваться в разрабатываемой графовой модели нейронной сети, за основу можно принимать одно из трёх эквивалентных представлений графа:

- перечисление рёбер и вершин графа;
- задание матрицы смежности вершин графа;
- задание матрицы инциденции.

Практические соображения заставили остановиться на первом способе, обладающем преимуществами наглядности и позволяющем одновременно с той информацией, которая





присутствует в матрицах смежности (количество рёбер, соединяющих некоторые вершины) и инциденции (признаки инциденции вершин рёбрам), задавать также веса рёбер, причём со знаками весов (рис. 1). По перечню рёбер и вершин легко восстанавливаются и матрица смежности, и матрица инциденции.

Выбранное представление исходной информации и представление об используемых в модели объектах использовались для формирования классов объектов (рис. 2). Для работы с достаточно простыми объектами или их свойствами были введены синонимы стандартных (встроенных) типов. Это помогло избежать неприятных ситуаций, когда при программировании некоторые объекты (или свойства объектов) модели могли случайно оказаться перепутанными со вспомогательными переменными, имеющими те же стандартные типы.

```
6  11 11
1  2  2.1
1  6  8.4
11 1  2
2  5  3.6
5  2  -3.2
2  2  8.4
```

*Рис. 1. Пример задания рёбер и вершин графа*
*Fig. 1. Graph edges and vertices list example*

В частности, для работы с номерами и именами вершин и рёбер был введён синоним `Node` стандартного беззнакового интегрального типа `size_t`, с той же целью для работы с весами рёбер вместо встроенного плавающего типа `double` было отдано предпочтение его синониму `Weight`.

```
class Vertex;           typedef vector <Vertex> Vertices;
class Edge;             typedef vector <Edge>   Edges;
class Graph;
class SpanT;
typedef size_t Node;    typedef vector <Node>   vNodes;
                        typedef set    <Node>   sNodes;
```

*Рис. 2. Классы, использовавшиеся в графовой модели нейронной сети*
*Fig. 2. Classes used in graph model of neuron network*

Объектам, соответствующим вершинам и рёбрам моделируемых графов, были сопоставлены классы `Vertex` и `Edge`, атрибутика которых выбрана предельно простой:

- вершина – это поле имени и три поля, предназначенных для хранения количества входящих, выходящих и полного числа входящих и выходящих рёбер;
- ребро – это две инцидентные вершины и вес, приписанный ребру.

Коллекции вершин и рёбер было принято решение реализовать, применяя один из библиотечных контейнеров-последовательностей, так как в определённых ситуациях, например, при построении циклов на графах, отношение следования становится весьма важным. В конечном итоге было решено использовать библиотечные контейнеры `vector`, которые могут эффективно использоваться при сортировке рёбер и вершин графов, при поиске циклов в графах, при модификациях графов.

Несмотря на стремление максимально пользоваться возможностями языка реализации и стандартной библиотеки, для объектов некоторых невстроенных типов потребовалось

определить новые классы, построить агрегации классов, а также, хотя и очень простую, но необходимую в данном случае наследственную иерархию (рис. 3).

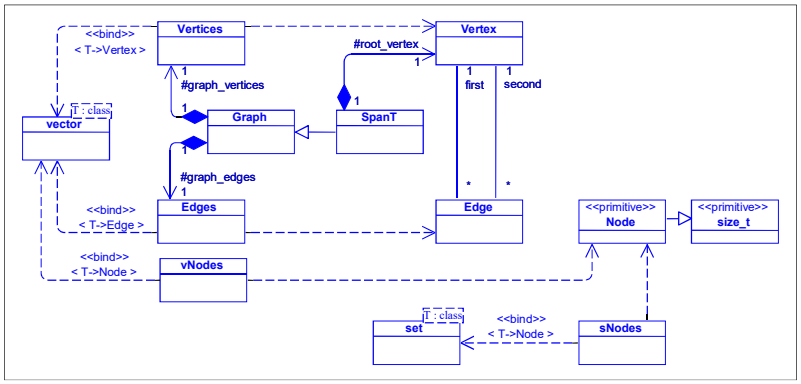


*Рис. 3. Взаимоотношения классов графовой модели нейронной сети*
*Fig. 3. Neuron network graph model classes' relations*

```
enum Graph_direction { Directed, NonDirected };
class    Vertex { Node name, degree_from, degree_twrd,
                      degree_both; Weight weight;
 public: Vertex ( Node vN = 0, Node vDF = 0, Node  vDT = 0,
                      Node vDB = 0, Weight vW = 0.0);
                                                 /* ... */
};
class    Edge   { Node name;   Vertex vertex_from;
                      Vertex vertex_twrd;
                  Node degree; Weight weight;
 public: Edge   ( Node eN, Vertex eF, Vertex eT,
                  Node degree = 0, Weight eW = 0.0); /* ... */
};
class   Graph { protected:  Vertices * Graph_Vertices;
                      Edges * Graph_Edges;
                Graph_direction   Graph_Orientation;
public: Graph (const Graph_direction D = Directed);
        Graph (const Graph & G);
        Graph (      Graph &&G);
        Graph (const Graph_direction d = Directed);
        Graph (const Edges & E);
        virtual    ~ Graph   ();                    /* ... */
};
class SpanT: public Graph { protected: Vertex  Root_Vertex;
public: SpanT ():    Graph (Directed) {};
        SpanT (const SpanT & S);
        SpanT (      SpanT &&S);
        SpanT (const Edges & E);
        SpanT (const Edges & E, const Node nF, const Node nT);
        SpanT (const Edges & E, const Vertex & V);   /* ... */
};
```

*Рис. 4. Интерфейсы конструкторов и деструктора основных классов графовой модели нейронной сети*
*Fig. 4. Constructors and destructor interfaces of neuron network graph model main classes*





В этой иерархии класс `Graph` описания графов общего вида (как ориентированных, так и неориентированных) является базовым классом, производным от которого был организован класс `SpanT`, предназначенный для работы с графами специального вида – остовными деревьями. Построенная иерархия действительно очень проста, хотя в ней есть виртуальные функции, при этом многие методы, необходимые для работы с остовными деревьями, наследуются и используются в производном классе без изменения. Связана такая простота с тем, что практически вся нагрузка ложится на классы, агрегированные в графы и остовные деревья, то есть на коллекции рёбер и вершин.

## 3.3 Конструкторы и деструкторы

Конструкторы и деструкторы классов графовой модели имеют строго индивидуальный характер (рис. 4).

В самых простых классах (`Vertex` и `Edge`) вводятся только по одному специальному конструктору, а деструкторов вообще нет. В большинстве ситуаций эти классы могут обходиться теми возможностями, которые предоставляются автоматически генерируемыми специальными методами классов. Библиотечные коллекции (`Vertices` и `Edges`) имеют много разных конструкторов, все они предоставляются библиотекой языка Си++, не требуя никаких дополнительных построений.

Нетривиальный набор конструкторов есть у базового (`Graph`) и производного (`SpanT`) классов наследственной иерархии. Кроме обычного набора конструкторов копирования, конструктора умолчания и совмещённого с ним конструктора преобразования, определяющего ориентированные или неориентированные графы, в базовом классе присутствует ещё один конструктор преобразования, позволяющий создать объект по перечню рёбер. Такого конструктора оказывается достаточным для реализации содержательной работы по преобразованию произвольного списка рёбер в граф, с полностью определённой структурой рёбер и вершин. Аналогичный набор конструкторов имеется и в классе `SpanT` остовных деревьев. Единственным отличием этого класса от его базового класса является присутствие в производном классе делегирующего конструктора и конструкторов, позволяющих определить не только рёбра и вершины дерева, но также его корневую вершину.

Содержательный виртуальный деструктор достаточно иметь только у базового класса, поскольку этот класс неплоский и уничтожение соответствующих объектов должно предваряться освобождением памяти, ранее занятой составными частями объекта.

## 3.4 Операции над вершинами, рёбрами и графами

В теории графов вопрос об операциях над графами проработан в достаточной степени. Важность самого понятия графа и сложность работы с такими объектами давно привели к выработке большого количества разнообразных операций над графами и их составными частями. Некоторые операции довольно просты, другие сложнее, для некоторых в языке математики введены специальные знаки операций ($\cup$, $\cap$, $x$), другие выражаются словами («*построим матрицу смежности ...*»).

Прямого соответствия операций над графами операциям языка Си++ нет, но можно попытаться перегрузить некоторые операции так, чтобы некоторая семантическая близость между теоретическими и языковыми операциями всё же оставалась.

Введение некоторых операций определялось самим языком. Например, совершенно очевидно, что в каждом классе должна присутствовать операция присваивания, позволяющая создать ещё одну копию исследуемого объекта. Также очевидно, в классах, не являющихся плоскими, должна определяться и операция присваивания по правой ссылке (перенос значения).

Другие операции оказалось необходимым наполнять некоторым изначально неочевидным смыслом. Например, в классе *Edge* описания рёбер графа была определена унарная операция инверсии последовательности рёбер ('~'), а также бинарные операции '%' и '%=', выполняющие такую сортировку последовательности рёбер, что в отсортированной последовательности два ребра следуют друг за другом, если одно из них является входящим в некоторую вершину, а второе – выходящим из той же вершины (рис. 5). В отсортированную последовательность при этом включаются только те рёбра исходной последовательности, которые могут быть выстроены в указанном порядке, начиная с заданной вторым операндом вершины. Эти операции оказались очень удобными при работе с циклами в графах, так как для циклов указанные операции не приводят к исключению рёбер из отсортированных последовательностей.

```
class Edge  {                                              /* ... */
friend       Edges&operator+=(      Edges&eL, const Edges&eR);
friend const Edges operator- (const Edges& E);
friend const Edges operator% (      Edges& E, const Node   n);
friend       Edges&operator%=(      Edges& E, const Node   n);
                                                           /* ... */
};
```

*Рис. 5. Интерфейсы операций класса описания рёбер графа*
*Fig. 5. Interfaces of class Edge operators*

```
class Graph {                                              /* ... */
public:
virtual       Graph&   operator-=(const Node     & n);
virtual       Graph&   operator-=(const Edge     & e);
virtual       Graph&   operator-=(const Edges    & E);
virtual       Graph&   operator-=(const Vertex   & v);
virtual       Graph&   operator-=(const Vertices & V);
virtual       Graph&   operator+=(const Edge     & e);
virtual       Graph&   operator+=(const Edges    & E);
virtual       Graph&   operator+=(const Graph    & G);
virtual       Graph&   operator= (const Graph    & G);
virtual       Graph&   operator= (      Graph   && G);
virtual       Graph    operator% (const Node       n) const;
virtual       Graph&   operator%=(const Node       n);
virtual const Graph    operator+ () const;
        const Graph    operator~ ();
virtual       Graph&   operator>>(Graph    & G) const;
virtual       Edges&   operator>>(Edges    & E) const;
virtual       Vertices&operator>>(Vertices & V) const;
virtual const Graph    operator- () const;
        const Graph&   operator--();
        const Graph    operator--(int i);
 friend const Graph operator+(const Edges& E,const Graph& G);
 friend const Graph operator+(const Graph& G,const Edges& E);
 friend const Graph operator+(const Graph&gL,const Graph&gR);
                                                           /* ... */
};
```

*Рис. 6. Интерфейсы операций класса описания графов*
*Fig. 6. Interfaces of class Graph operators*

Аналогичные операции '~' (реверса), '%' и '%=' (специальной сортировки) удобно ввести и в класс *Graph* описания графов, так как в каждом графе имеется собственная





последовательность рёбер (рис. 6). Естественно, что эти операции наследуются классом *SpanT* описания остовных деревьев (рис. 7), над которыми также приходится строить циклы (добавлением некоторых рёбер) и, следовательно, проводить их сортировку.

```
class SpanT: public Graph {                                /* ... */
public:       SpanT&   operator= (const SpanT  & S);
              SpanT&   operator= (      SpanT && S);
              SpanT&   operator= (const Graph  & S) override;
              SpanT&   operator= (      Graph && S) override;
        const SpanT&   operator--();
        const SpanT    operator--(int i);                  /* ... */
};
```

*Рис. 7. Интерфейсы операций класса описания остовных деревьев*
*Fig. 7. Interfaces of class SpanT operators*

В состав операций над объектами этих типов также было признано удобным вставить унарную операцию '+', которая нормализует граф, то есть, беря за основу последовательность рёбер графа, составляет соответствующую ей последовательность вершин.

Оказалось, что это не единственный вид операции нормализации графа, которая должна применяться к графам при моделировании нейронных сетей. Другая унарная операция нормализации '~' при применении к графу оставляет в нём только вершины промежуточных слоёв, которые имеют и входящие, и исходящие рёбра, попутно удаляя из графа рёбра с нулевыми весами.

## 3.5 Сортировка и поиск, реверс графа

Функции сортировки и поиска играют важную роль при обработке любых коллекций данных. Важны они для реализации графовых моделей, которые так или иначе связаны с коллекциями объектов, предназначенных для моделирования вершин, рёбер, графов, деревьев.

```
const Node   Find   (const Vertices & V, const Node   & n);
const Node   Find   (const Vertices & V, const Vertex& v);
const Weight Find   (const Edges    & E, const Edge   & e);
const Weight Find   (const Edges    & E, const Edge   & e,
                     const Node_position nP);
const Node   Find   (const Edges    & E, const Node     n,
                     const Node_position nP);
const Node   Find   (const Edges    & E, const Node     F,
                                         const Node    vT);
const Node   Find   (const Graph    & G, const Node    nF,
                                         const Node    nT);
const Node   Find   (const Graph    & G, const Node     n,
                     const Node_position nP);
const Node   Find   (const Graph    & G, const Node    sE,
                     const Node nF,      const Node    nT);
const bool   Find   (const Graph    & G,       Edge   & e,
                     const Node nF,      const Node    nT);
const Weight Find   (const Graph    & G, const Edge   & e,
                     const Node_position nP);
const bool   Check  (const Graph    & G, const Edge   & e);
```

*Рис. 8. Интерфейсы функций поиска и проверки вхождения и реверсирования*
*Fig. 8. Interfaces of Find and Check functions*

Обычно таких функций требуется много, так как и для сортировки, и для поиска данных в коллекциях часто требуется реализовать множество алгоритмов и их вариантов.

При реализации графовой модели нейронной сети понадобилось более 10 вариантов для функций поиска и семантически примыкающей к ним функции проверки вхождения (рис. 8). Несколько вариантов оказалось необходимо реализовать и для сортировки (рис. 9), так как разные объекты должны сортироваться на основе совершенно разных критериев, а иногда и алгоритмов. В частности, если рёбра и вершины обладают некоторыми численными характеристиками (*степенями*), позволяющими, использовав наличие у них свойства быть операндами операций отношения, легко сортировать их по возрастанию или убыванию некоторых численных величин, то, например, для такого объекта, как цикл в графе, сортировка имеет совершенно другой смысл.

```
Edges Sort (Edges    & G, const Node n);
Graph Sort (Graph    & G, const Node n);
Node  Sort (Vertices & V, const Node_position Np);
void  Sort (Edges    & E, const Node_position Np);
Node  Sort (Graph    & G,
            const Node_position Vertex_degree_direction,
            const Node position  Edges weight direction);
```

*Рис. 9. Интерфейсы функций сортировки графов, рёбер и вершин*
*Fig. 9. Interfaces of Sort functions*

Сортировка и поиск – это операции, часто встречающиеся в самых разных программных системах, разрабатываемых для самых разных прикладных областей. Они используются не только при работе с графами, но и с другими структурами данных. Однако при работе с графами необходимо выполнять и более специфические операции, в частности, в разрабатываемой модели в состав операций классов была включена функция реверса и реализованная на её основе операция '–' (унарного минуса) (рис. 10). С помощью этой операции и лежащей в её основе функции можно выполнить перестановку элементов (рёбер) в контейнере типа `Edges` или в графе.

```
class Edge  {                                          /* ... */
public:       Edges & Revert     (      Edges & E);
friend  const Edges   operator - (const Edges & E);  /* ... */
};
class Graph {                                          /* ... */
public:       Edges & Revert     (      Edges & E);
virtual const Graph   operator - () const;
```

*Рис. 10. Интерфейсы функций реверса перечня рёбер и графа*
*Fig. 10. Interfaces of graph and edges set Revert functions*

## 3.6 Сложные тестирующие операции – поиск циклов, подсчёт весов и знаков циклов, вставка вспомогательных вершин и рёбер

Несмотря на исключительную полезность уже описанных структур и операций, суть графовой модели составляют другие её компоненты. Разработка модели производилась с целью исследования циклов, которые возникают в графах, соответствующих нейронным сетям, в которых между различными слоями нейронов имеются не только прямые, но и обратные связи. И те, и другие связи обладают весами, то есть численными характеристиками, от которых зависит степень взаимного влияния нейронов друг на друга. Структуры связей нейронов в разных нейронных сетях могут быть разными. Если обратных связей нет, сети называются сетями прямого распространения. Для таких сетей соответствующей графовой моделью будет дерево. Такие структуры встречаются достаточно часто (например, в свёрточных сетях), но авторов больше интересовали сети с обратными связями, имеющие более сложные структуры, а среди таких сетей те из них, которые имеют





отрицательные веса. Обратные связи могут приводить к образованию циклов, эти циклы могут иметь разные свойства, для изучения которых и строилась графовая модель. В работе [15] предлагается вычислять знак цикла, выявленного в графе нейронной сети, и доказывается, что наличие в графе циклов, имеющих отрицательные знаки, может приводить к возникновению ложных и неустойчивых решений. Знак цикла вычисляется как знак произведения весов входящих в цикл рёбер.

Авторы строили графовую модель и программную систему на её основе для проведения экспериментальной проверки и иллюстрации работы предложенного ими алгоритма модификации циклов, позволяющего избавиться от отрицательных циклов в структуре нейронной сети [16]. Доказанная в цитированной работе теорема о знаках циклов утверждает, что, если все циклы, образующиеся при присоединении к остовному дереву графа любого одного ребра из кографа этого остовного дерева, положительны, то все циклы данного графа будут также иметь положительный знак.

Для демонстрации разработанного авторами алгоритма были написаны несколько тестирующих процедур, реализующих отдельные шаги алгоритма (рис. 11).

```
Weight Build_Spanning_Tree
    (const Graph & G, const Node Root_node, SpanT & S);
void Build_CoGraph (const Graph & G, SpanT & S, Graph & CoG);
Node Count_sign
    (const Graph & G, const Edge & e, vNodes * C []);
Node Discover_Span_Circuits
    (const Graph & G, SpanT & S, vNodes * C []);
Node Discover_All_Circuits (const Graph & G, vNodes * C []);
Node Evaluate_Circuits_Signs
    (const Graph & G, vNodes * C [], Signs & Ss);
Node Add_Vertex_and_Edges (Graph & G,
      Node From_Vertex_Name,    Node Twrd_Vertex_Name,
    Weight First_part_weight, Weight Second_part_weight);
```

*Рис. 11. Интерфейсы функций, реализующих основные шаги алгоритма модификации циклов графа*
*Fig. 11. Interfaces of main functions of graph modification algorithm*

Основные процедуры, реализующие шаги алгоритма модификации графа, в котором присутствуют отрицательные циклы, были следующими:

- процедуры `Build_Spanning_Tree()` и `Build_CoGraph()` построения остовного дерева графа и кографа этого дерева;
- процедура `Discover_Span_Circuits()` выявления цикла, возникающего при добавлении к остовному дереву графа одного из рёбер, входящих в соответствующий этому дереву кограф;
- процедура `Discover_All_Circuits()` выявления всех циклов в графе;
- процедура `Count_sign()` вычисления знака цикла, возникающего при добавлении к остовному дереву графа одного из рёбер, входящих в соответствующий этому дереву кограф;
- процедура `Evaluate_Circuits_Signs()` вычисления знаков циклов;
- процедура `Add_Vertex_and_Edges()` модификации цикла путём замены одного из рёбер парой новых рёбер, имеющих одну общую вершину, также добавляемую к графу.

Приведённый перечень процедур в совокупности с ранее описанными операциями над графами, остовными деревьями, рёбрами и вершинами, а также со вспомогательными процедурами построения внешнего представления графов и циклов в них позволил

полностью и наглядно проиллюстрировать работу предложенного алгоритма модификации графов, детали которой можно найти в уже цитировавшейся работе [16].

Все эти процедуры запрограммированы с активным применением представленных в предыдущих разделах операциях над графами и остовными деревьями, а также над входящими в них вершинами и рёбрами.

Включение в созданную графовую модель большого числа перегруженных операций позволило записывать алгоритмы сложных тестирующих процедур очень наглядно, что в свою очередь способствовало снижению числа потенциальных ошибок в этих процедурах и снижению времени отладки всего комплекса программ. На рис. 12 приведён пример реализации одной из таких процедур.

Перечень операций над графами, используемых в текущей версии графовой модели, является открытым. При изменении каких-либо требований к модели или при их расширении или возникновении новых сценариев тестирования сетевых структур, этот перечень может дополнен дополнительными операциями. Единственным ограничением здесь могут быть возможности самого языка программирования Си++, который не позволяет перегружать некоторые операции и не позволяет вводить новые знаки операций. Однако, наталкиваясь на подобные ограничения, модель не теряет своей гибкости, хотя до определённой степени в случае роста числа моделируемых операций над графами может начать терять в наглядности представления этих операций.

```
Node Discover_Span_Circuits (const Graph  & G, SpanT & S,
                             vNodes * C [])
{ Node Nc = 0;
  Graph CoG (G.get_Orientation());
  Build_CoGraph (G, S, CoG);
  for (auto & edge: CoG.get_Edges())
      { S  += edge;
        Nc += Discover_All_Circuits(S, C + Nc);
        -- S;
      }
  S = + S;
  return Nc;
}
```

*Рис. 12. Реализация процедуры `Discover_Span_Circuits()` с помощью перегруженных операций и других процедур графовой модели*
*Fig. 12. Implementation of `Discover_Span_Circuits()` procedure using overloaded operations and other procedures of graph model*

## *4. Заключение*

В работе описан подход к тестированию искусственных нейронных сетей, реализованный на языке Си++ в программном комплексе в виде набора классов и алгоритмов обработки объектов этих классов. Представлены интерфейсы важнейших операций над используемыми объектами и тестирующих процедур. Продемонстрированный авторами подход даёт возможность унифицировать процесс решения прикладных задач с помощью нейронных сетей. При появлении конкретной задачи, которую требуется решить, несложно выбрать нейросетевую модель, которая лучше всего подходит для этой задачи, определить архитектуру сети, а также организовать процедуры и сценарии тестирования. Ядром используемых методов являются операции над графами, описывающими структуры связей между нейронами. Эти операции могут быть достаточно сложными, однако они не сложнее, чем операции, выполняемые самой нейронной сетью. Если используемое аппаратное обеспечение даёт возможность реализовать нейронную сеть, оно также может дать





возможность и протестировать её с помощью предлагаемых алгоритмов. Приведённые эксперименты, некоторые из которых описаны в другой работе авторов (см. [16]) показывают, что предложенный подход позволяет в значительной степени устранить как использование чрезмерно сложных моделей для решения конкретных прикладных задач, так и риск получения неверных решений.

## Список литературы / References

[1]. Ciresan D., Meier U., Masci J., and Schmidhuber J. Multi-column deep neural network for traffic sign classification. Neural Networks, vol. 12, 2012, pp. 333-338.
[2]. David Talbot. CES 2015: Nvidia Demos a Car Computer Trained with "Deep Learning". MIT Technology Review, January 6, 2015, available at: https://www.technologyreview.com/s/533936/ces-2015-nvidia-demos-a-car-computer-trained-with-deep-learning/.
[3]. Roth S. Shrinkage Fields for Effective Image Restoration. In Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2014, pp. 2774-2781.
[4]. Deng L. and Yu D. Deep Learning: Methods and Applications, Foundations and Trends in Signal Processing, vol. 7, no. 3–4, 2014, pp. 1–19.
[5]. Ю.Л. Карпов, Л.Е. Карпов, Ю.Г. Сметанин. Адаптация общих концепций тестирования программного обеспечения к нейронным сетям. Программирование, т. 44, № 5, 2018, стр. 43-56. DOI: 10.31857/S013234740001214-0 / Yu.L. Karpov, L.E. Karpov, Yu.G. Smetanin. Adaptation of General Concepts of Software Testing to Neural Networks. Programming and Computer Software, vol. 44, № 5, 2018, pp. 324-334. DOI: 10.1134/S0361768818050031.
[6]. Rosenblatt Frank. Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms. Spartan Books, Washington DC, 1961, 616 p.
[7]. Kohonen T. Self-Organization and Associative Memory. New York: Springer, 1984, 332 p.
[8]. Grossberg S. Nonlinear Neural Networks: Principles, Mechanisms, and Architectures. Neural Networks, vol. 1, issue 1, 1988, pp. 17-61.
[9]. Hebb D.O. The Organization of Behavior. Wiley, New York, 1948, 335 p.
[10]. Harary F. Graph theory, Addison Wesley, 1969, 273 p.
[11]. Ore O. Theory of graphs. American Mathematical Society, Providence, RI, 1962, 269 p.
[12]. Иорданский М.А. Конструктивная теория графов и её приложения. Из-во Кириллица, 2016, 172 стр. / Iordanski M.A. Constructive graph theory and its applications. Cyrillic, 2016, 172 p. (in Russian).
[13]. J.J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. Proceedings of the National Academy of Sciences of the USA, vol. 79 no. 8, 1982, pp. 2554–2558.
[14]. Hinton D.E. and Seinowski T. Optimal Perceptual Inference. In Proc. of the IEEE Conference on Computer Vision and Pattern Recognition, 1983, pp. 448–453.
[15]. Julio Aracena, Jacques Demongeot, and Eric Goles. Positive and Negative Circuits in Discrete Neural Networks. IEEE Transactions on Neural Networks, vol. 15, No. 1, Jan. 2004, pp. 77-83.
[16]. Ю.Л. Карпов, Л.Е. Карпов, Ю.Г. Сметанин. Устранение отрицательных циклов в некоторых структурах нейронных сетей с целью достижения стационарных решений. Программирование, т. 45, № 5, стр. 25-35, 2019. DOI: 10.1134/S0132347419050029 / Yu.L. Karpov, L.E. Karpov, Yu.G. Smetanin. Elimination of Negative Circuits in Certain Neural Network Structures to Achieve Stable Solutions. Programming and Computer Software, vol. 45, № 5, pp. 241-250, 2019, DOI: 10.1134/S0361768819050025.

## Информация об авторе / Information about the author

Юрий Леонидович КАРПОВ – кандидат технических наук, начальник отдела. Научные интересы: программирование, технологии тестирования программного обеспечения.

Yuri Leonidovich KARPOV – Candidate of Technical Sciences, Head of Department. Research interests: computer programming, software testing and engineering.

Ирина Анатольевна ВОЛКОВА – кандидат физико-математических наук, доцент по кафедре алгоритмических языков ф-та ВМК. Научные интересы: алгоритмические языки, методы трансляции.

Irina Anatolievna VOLKOVA – Candidate of Physics and Mathematics, Associate Professor in the Department of Algorithmic Languages, Faculty of CMC. Research interests: algorithmic languages, compilation techniques.

Алексей Александрович ВЫЛИТОК – кандидат физико-математических наук, доцент по кафедре алгоритмических языков ф-та ВМК. Научные интересы: языки программирования, формальные грамматики, объектно-ориентированная декомпозиция.

Alexey Alexandrovich VYLITOK – Candidate of Physics and Mathematics, Associate Professor in the Department of Algorithmic Languages, Faculty of CMC. Research interests: programming languages, formal grammars, object-oriented decomposition.

Леонид Евгеньевич КАРПОВ – доктор технических наук, ведущий научный сотрудник ИСП РАН, доцент кафедры системного программирования ф-та ВМК. Научные интересы: системное программирование, методы компиляции, системы программирования.

Leonid Evgenievich KARPOV – Doctor of Technical Sciences, Leading Researcher at ISP RAS, Associate Professor of the System Programming Department of the VMK Faculty. Research interests: system programming, compilation techniques, programming systems.

Юрий Геннадьевич СМЕТАНИН – доктор физико-математических наук, главный научный сотрудник ВЦ РАН им А.А. Дородницына ФИЦ «Информатика и управление» РАН, старший научный сотрудник кафедры интеллектуальных систем ф-та ФУПМ МФТИ. Научные интересы: комбинаторика слов, символическая динамика, нейронные сети.

Yuri Gennadievich SMETANIN – Doctor of Physical and Mathematical Sciences, Chief Researcher of the Dorodnicyn Computing Centre of FRC «Informatics and Control» RAS, senior researcher at the Department of Intelligent Systems, Faculty of Physics and Technology of MIPT. Scientific interests: combinatorics of words, symbolic dynamics, neural networks.