МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

УТВЕРЖДАЮ
Зав.кафедрой,
к. ф.-м. н.
_____ С. В. Миронов

ОТЧЕТ О ПРАКТИКЕ

студента 4 курса 411 группы факультета КНиИТ
Власова Андрея Александровича

вид практики: учебная
кафедра: математической кибернетики и компьютерных наук
курс: 4
семестр: 2
продолжительность: 2 нед., с 30.06.2017 г. по 13.07.2017 г.

Руководитель практики от университета,

доцент, к. ф.-м. н. _____ Ю. Н. Кондратова

Руководитель практики от организации (учреждения, предприятия),

доцент, к. ф.-м. н. _____ Ю. Н. Кондратова

Тема практики:«Создание приложения для анализа генетического алгоритма поиска центральных вершин»

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Введение

1 Использование приложения

2 Описание технологий и архитектуры приложения

В качестве языка программирования для решения поставленной задачи был выбран объектно-ориентированный язык C#. [1] Вместе с тем для создания клиент-серверного приложения был использован фреймворк `ASP.NET MVC 5`, который позволяет создавать веб-приложения с использованием архитектуры `MVC`. Кроме этого в качестве системы объектно-реляционного отображения используется технология `Entity Framework 6`. При этом приложение разделено на три слоя абстракции — уровень доступа к данным, уровень бизнес-логики и уровень визуального представления.

2.1 Структура базы данных

Хранение графов в базе данных были созданы следующие таблицы: `Graphs` и `Edges`. При этом таблица `Graphs` содержит следующие поля:

- поле `Id` (типа данных `INT`) — уникальный идентификатор, внутренний ключ,
- поле `N` (типа данных `INT`) — количество вершин в графе,
- поле `M` (типа данных `INT`) — количество ребер в графе,
- поле `Name` (типа данных `NVARCHAR`) — название графа,
- поле `R` (типа данных `INT`) — радиус графа.

Кроме этого таблица `Edges` состоит из следующих полей:

- поле `Id` (типа данных `INT`) — уникальный идентификатор, внутренний ключ,
- поле `V1` (типа данных `INT`) — одна из вершин, которые соединяет ребро,
- поле `V1` (типа данных `INT`) — вторая из вершин, которые соединяет ребро,
- поле `Graph_Id` (типа данных `INT`) — `Id` графа, которому принадлежит ребро, внешний ключ.

Кроме этого для хранения зарегестрированных пользователей существует таблица `Users`:

- поле `Id` (типа данных `INT`) — уникальный идентификатор, внутренний ключ,
- поле `Login` (типа данных `NVARCHAR`) — логин пользователя,
- поле `Password` (типа данных `NVARCHAR`) — зашифрованный пароль пользователя.

Полная диаграмма таблиц представлена на рисунке 1:

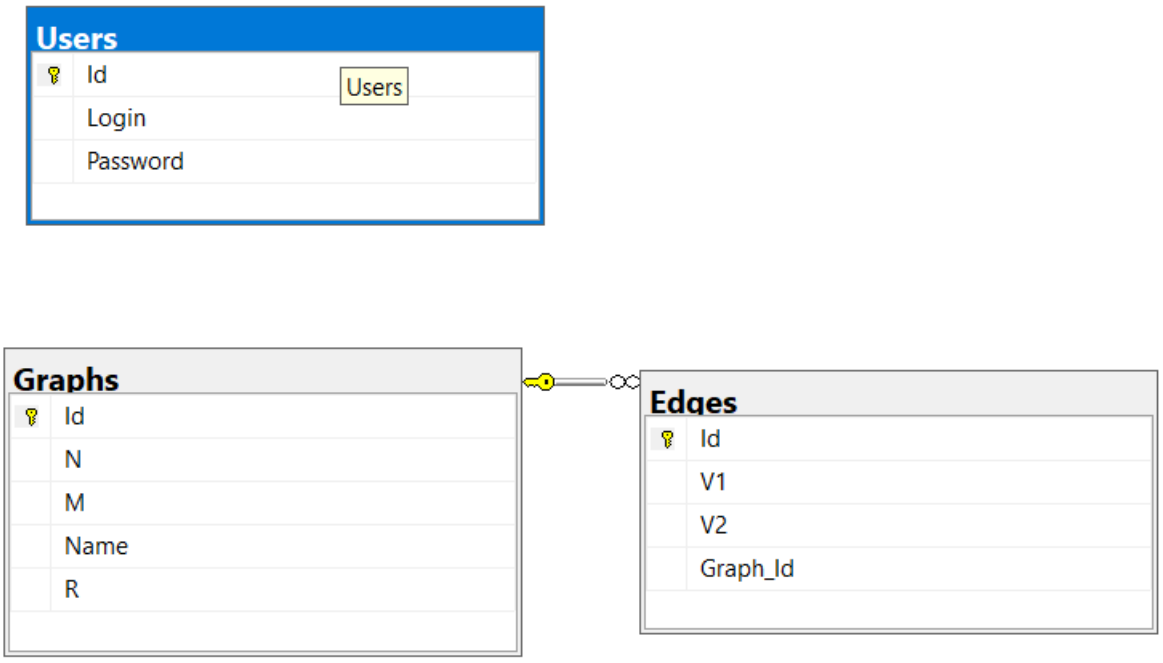

Рисунок 1 – Диаграмма базы данных

Для создания базы данных используется технология Entity Framework, вместе с чем использовался подход Code-first, согласно которому были созданы классы Graph, GraphInfo, Edge, User, описывающие модели данных:

```
public class GraphInfo
{
        public int Id { get; set; }
        public int N { get; set; }
        public int M { get; set; }
        public string Name { get; set; }
        public int R { get; set; }
}
```

```
public class Graph : GraphInfo
{
        public ICollection<Edge> Edges { get; set; }
        public Graph()
        {
                Edges = new List<Edge>();
        }
}
```

```
public class Edge
{
        public int Id { get; set; }
        public int V1 { get; set; }
        public int V2 { get; set; }
}
```

После этого фреймворк на основе созданных моделей сам генерирует таблицы в базе данных и ее структуру.

2.2 Уровень доступа к данным

Для гибкой и стандартизированной работы с базой данных был создан ряд интерфейсов, в которые были вынесены основные методы для доступа к данным и их изменениям. Классы, реализующие эти интерфейсы представляют собой уровень доступа к данным, при этом использование интерфейсов позволяет с легкостью изменять реализацию этих классов, а также упрощает процесс тестирования.

Далее приводится код основных интерфейсов, которые используются при работе с данными:

```
public interface IGraphDao
{
        IEnumerable<GraphInfo> GetAllGraphInfo();
        Graph GetById(int id);
        Graph Add(Graph graph);
}
```

```
public interface IUserDao
{
        User GetById(int id);
        User GetByName(string name);
        User Add(User user);
}
```

Вместе с тем для использования технологии `Entity Framework` созданы классы `GraphContext` и `UserContext`, которые наследуются от класса `System.Data.Entity.DbContext`, что позволяет получить возможность для легкого доступа к базе данных без написания SQL запросов. Классы `GraphContext` и `UserContext` содержат в себе поля типа `DbSet<Graph>` и `DbSet<User>`, через которые происходит добавление, чтение или изменение данных в базе

данных. Кроме этого в этих классах описан статический конструктор, внутри которого указан способ начальной инициализации базы данных, за счет классов `UserContextInitializer` и `GraphContextInitializer`. Эти классы наследуются от класса `CreateDatabaseIfNotExists`, что позволяет фреймворку выполнить начальное заполнение данными, если база данных еще не существует, за счет кода, который описан в переопределенном методе `Seed`. Внутри этого метода происходит чтение нескольких созданных графов из текстовых файлов, все это происходит внутри метода, описанного в классе `GraphContextInitializer`, в этом же методе в классе `UserContextInitializer` добавляется пользователь с логином `admin` и паролем `admin`. Полный код представлен в приложении [ссылка].

2.3 Уровень бизнес-логики

Уровень бизнес-логики представляет собой похожую структуру, как и уровень доступа данных — так же созданы ряд интерфейсов и классы, которые их реализуют. При этом многие из этих интерфейсов похожи на те, которые описаны в уровне доступа к данным, однако именно на этом уровне происходит запуск генетического алгоритма с различными параметрами и анализ загруженных графов. С определением интерфейсов и классов реализующих бизнес-логику приложения можно ознакомиться в приложении [ссылка]. Классы `GraphBL` и `UserBL`, реализуют интерфейсы `IGraphBL` и `IUserBL`. Они содержат ссылки на объекты, реализующие интерфейсы `IGraphDao` и `IUserBL`, при этом эти объекты передаются в качестве параметров в соответствующие конструкторы. При добавлении нового пользователя происходит проверка на существование записи с таким же логином, а кроме этого происходит шифровка пароля.

Также при добавлении нового графа в базу данных в классе, который отвечает за работу с моделью графа, происходит проверка графа на связность и его размеры. При неудачном прохождении проверки выбрасывается исключение, которое отлавливается на уровне представления.

Кроме этого для работы с генетическим алгоритмом создан интерфейс `IAlgorithm`:

```
1 public interface IAlgorithm
2     {
3         FindingVertexResponse FindCentralVertex(Graph graph);
```

```
4           ResearchAlgorithmResponse ResearchAlgorithm(ResearchRequest param);
5       }
```

Интерфейс определяет набор методов, в которых будет реализована логика для работы с генетическим алгоритмом. Вместе с тем класс `Algorithm` реализует данный интерфейс и в нем содержится вся логика работы с алгоритмом — получение результатов поиска центральных вершин, замеры времени работы и процента неверно найденных решений.

Результаты измерений возвращаются из методов при помощи классов `FindingVertexResponse` и `ResearchAlgorithmResponse`:

```
1  public class FindingVertexResponse
2  {
3          public int[] Center { get; set; }
4          public int R { get; set; }
5          public double Time { get; set; }
6  }
```

```
1  public class ResearchAlgorithmResponse
2      {
3          public double AvgTime { get; set; }
4          public double Error { get; set; }
5      }
```

2.4 Уровень представления

Так как проект представляет собой веб-приложение, то уровень, отвечающий за пользовательский интерфейс реализован при помощи технологии `ASP .NET MVC 5`. В связи с чем весь код этого уровня разделен на:

— контроллеры, которые отвечают на HTTP запросы клиента с помощью представлений,

— представления, которые написаны с использованием технологии Razor, позволяющей внедрять серверный C# код,

— модели данных, внутри которых происходит передача данных от клиента серверу и обратно.

2.4.1 Контроллеры

Для взаимодействия с клиентской частью приложения и обработки пользовательских данных было создано несколько контроллеров: `HomeController`, `GraphController`, `LoginController`, `ResearchController`.

Класс `HomeController` содержит один метод `Index`, который отвечает на GET-запрос и возвращает домашнюю страницу.

Класс `GraphController` включает в себя методы, определяющие URL-адреса при взаимодействии с которыми клиентской части приложения предоставляется возможность запускать генетический алгоритм для поиска центральных вершин или добавлять новый граф в базу данных. В целом данный контроллер ответственен за обработку следующих запросов:

— `/Graph` — обрабатывает GET-запрос возвращает загрузочную страницу для поиска центральных вершин,

— `/Graph/FindCentralVertex` — обрабатывает POST-запрос, в котором передается файл с графом, после чего запускается генетический алгоритм. В качестве результата возвращается страница с сообщением об ошибке, которая могла произойти при обработке запроса из-за неверного формата данных в файле с графом, или слишком большого размера загружаемого графа, либо же при успешном запуске возвращает страницу с результатами работы — радиус графа и возможные центральные вершины,

— `/Graph/Add` — обрабатывает GET-запрос, который возвращает форму для загрузки графа в базу данных,

— `/Graph/Add` — обрабатывает POST-запрос, в котором пользователь предает на сервер для сохранения файл с графом и название графа. При успешном добавлении перенаправляет пользователя на домашнюю страницу, при возможной ошибке — страницу с описанием ошибки.

Класс `ResearchController` содержит методы через которые пользователю предоставляется возможность запускать генетический алгоритм с различными параметрами (p_c, p_m, N), а также граф на котором будет тестироваться алгоритм. Контроллер содержит следующие методы:

— `/Research/Index` — обрабатывает GET-запрос и возвращает форму с выбором параметров для генетического алгоритма, при этом в результат подгружаются описания графов, сохраненных в базе данных,

— `/Research/ResearchAlgorithm` — обрабатывает POST-запрос в который передаются выбранные параметры алгоритма и выбранный граф.

В классе `LoginController` определены методы, за счет которых происходит регистрация и аутентификация пользователей. Пользователь, который вошел в систему получает возможность для добавления новых графов и добав-

ление новых пользователей, по сути залогиненному пользователю предоставляются права администратора. В контроллере `LoginController` реализованы следующие методы:

— `/Login/Index` — обрабатывает GET-запрос и возвращает форму заполнения для входа в систему,

— `/Research/SignIn` — обрабатывает POST-запрос в который передается логин и пароль. В этом методе происходит валидация введенных данных, проверка принадлежности пароля введенному пользователю и при успешном прохождении пользователю отправляется набор cookie-данных, вследствие чего происходит аутентификация пользователя. При неверном логине или пароле пользователю возвращается сообщение об ошибке, при успешном прохождении аутентификации — метод возвращает переадресацию на домашнюю страницу,

— `/Login/SignUp` — обрабатывает GET-запрос и возвращает форму для регистрации нового пользователя,

— `/Login/SignUp` — обрабатывает POST-запрос и добавляет нового пользователя при успешном прохождении валидации и отсутствии пользователя с таким же логином.

2.4.2 Модели данных и валидация

Основными классами, с помощью объектов которых происходит передача данным в контроллеры, `AddGraphRequest`, `CreateUserRequest`, `LoginUserRequest`, `ResearchRequest`, при этом результаты вычислений возвращаются из контроллеров в виде представлений, которые представляют собой HTML разметку с внедрением данных переданных через классы `AlgorithmResultResponse`, `FindingVertexResponse`, `ResearchAlgorithmResponse`.

При этом очевидно, что при этом введенные пользовательские данные должны удовлетворять некоторым условиям. Для того, чтобы передаваемые данные можно было проверить из любого участка кода для некоторых свойств были использованы атрибуты валидации [ссылка] `Required`, `Compare`, `StringLength`:

```
1 public class CreateUserRequest
2     {
3         [Required(ErrorMessage = "Введите логин")]
```

```
        [StringLength(50, MinimumLength = 3,
        ErrorMessage = "Длина логина должна быть от 3 до 50 символов")]
        public string Login { get; set; }

        [Required(ErrorMessage = "Введите пароль")]
        [StringLength(50, MinimumLength = 3,
        ErrorMessage = "Длина логина должна быть от 3 до 50 символов")]
        public string Password { get; set; }

        [Required(ErrorMessage = "Повторите пароль")]
        [Compare("Password", ErrorMessage = "Пароли не совпадают")]
        public string ConfirmPassword { get; set; }
    }
```

```
public class LoginUserRequest
    {
        [Required(ErrorMessage = "Введите логин")]
        public string Login { get; set; }

        [Required(ErrorMessage = "Введите пароль")]
        public string Password { get; set; }
    }
```

При таком использовании атрибутов валидации проверить модель на соответствие выдвинутым требованиям можно при помощи следующего кода:

```
if (ModelState.IsValid) {
        ...
}
```

внутри любого из контроллеров, где `ModelState` — свойство класса `Controller`, которое инкапсулирует состояние модели, переданной в качестве параметра запроса. В случае неудачного прохождения валидации в свойство `ModelState` при помощи метода `AddModelError` добавляется сообщение об ошибке, которое затем будет вставлено в HTML разметку. Атрибут `Requred` установлен для логина и пароля, вводимого пользователем, что гарантирует тот факт, что в базу данных не будет помещена запись с пустыми полями. В добавок к этому у свойства `ConfirmPassword` утановлен атрибут `Compare`, который требует, чтобы свойство, отвечающее за хранение пароля, совпадало с свойством, отвечающим за хранение повтора пароля. Также используется атрибут

StringLength, в котором устанавливаются минимальная и максимальная длина логина и пароля.

2.4.3 Аутентификация

Как уже отмечалось ранее доступ к возможности добавлять графы в базу данных и добавлять туда же новых пользователей имеют доступ только пользователи, которые вошли в систему. В связи с этим в качестве технологии аутентификации в созданном приложении используется аутентификация с помощью форм [ссылка]. Для ее включения в файл Web.config были добавлены следующие строки:

```
<authentication mode="Forms">
      <forms loginUrl="~/login" timeout="60" />
</authentication>
```

в которых указывается по какому адресу будет отправлен пользователь в случае, если он не имеет прав доступа к запрошенным ресурсам и время действия cookieфайлов. При успешном прохождении проверки на принадлежность пользователю введенного им пароля при помощи следующей строки кода клиентская часть получает cookie-файлы, которые затем будут присоединяться ко всем остальным запросам:

```
FormsAuthentication.SetAuthCookie(user.Login, true);
```

Для того, чтобы к определенным методам был доступ только аутентифицированным пользователям к каждому методу применяется атрибут Authorize, который гарантирует проверку на доступность для пользователя этих методов. При этом для пользователя вошедшего в систему несколько изменяется HTML разметка, что достигается при помощи использования свойства User.Identity.IsAuthenticated.

2.5 Шифровка паролей

Очевидно, что хранение паролей пользователей в открытом виде представляет собой подход нарушающий основы требования к безопасности приложения. В связи с чем каждый пароль при регистрации пользователя хешируется и полученный хеш сохраняется в базе данных. Хеширование паролей происходит в классе Encryption (см. приложение [ссылка]), где определены

публичные методы CreatePassword и CheckPassword. Создание хеша пароля происходит при помощи объекта класса Rfc2898DeriveBytes [ссылка]:

```
var pbkdf2 = new Rfc2898DeriveBytes(password, salt, iterations);
byte[] hash = pbkdf2.GetBytes(hashSize);
```

В конструктор класса передается строка с паролем, «соль» — псевдослучайная последовательность байт, которая используется для повышения криптоустойчивости хеша и параметр, отвечающий за искусственную временную задержку, которая позволяет избежать попытки грубого перебора. В базу данных сохраняется полученный хеш и сгенерированная «соль». При проверки подлинности пароля из базы данных извлекается хеш с «солью», после чего введенный пароль хешируется с сохраненной «солью» и результат сравнивается с тем, что было сохранено в базе данных.

2.6 Внедрение зависимостей

Ранее описывались независимые уровни, на которые разделено приложение, при этом гибкость и заменяемость каждого из уровней гарантируется описанием интерфейсов. Каждый из уровней содержит ссылки на объекты, которые реализуют тот или иной интерфейс при этом эти объекты передаются в качестве параметров в конструкторы. Для того, чтобы гарантировать тот факт, что во все конструкторы будут переданы одни и те же реализации интерфейсов и избежать дублирования кода в созданном приложении используется IoC-контейнер Ninject, который связывает интерфейсы с объектами, которые их реализуют и предоставлять их при необходимости. Связывание интерфейсов и реализации происходит в методе класса NinjectRegistrations:

```
public class NinjectRegistrations : NinjectModule
    {
        public override void Load()
        {
            Bind<IUserDao>().To<UserDao>();
            Bind<IGraphDao>().To<GraphDao>();
            Bind<IAlgorithm>().To<Algorithm>();
            Bind<IGraphBL>().To<GraphBL>();
            Bind<IUserBL>().To<UserBL>();
```

```
        }
    }
```

При этом в глобальном файле запуска приложения происходит регистрация этого класса в качестве основного способа разрешения зависимостей (см. приложение [ссылка]).

3 Работа с генетическим алгоритмом

Для изолированного доступа к методам генетического алгоритма и для корректного измерения временных затрат, а так же для корректного чтения графов из текстовых файлов были созданы следующие классы: `ExactAlgorithmCore`, `GeneticAlgorithmCore`, `GraphContext`, `GraphParser`.

3.0.1 Чтение графа из файла

Ранее говорилось о том, что графы передаются на сервер в текстовых файлах, при этом формат данных этого файла должен быть следующим: на каждой строке указывается ровно два числа — начало и конец ребра, при этом нумерация вершин должна начинаться с 0 и идти по порядку, т. е. если в графе N вершин, то максимальным числом в файле должно быть число $N - 1$ и встречаться все числа от 0 до $N - 1$. Кроме этого в файле не должно встречаться отрицательных чисел. С целью гарантированно принимать только файлы с корректными графами был создан статический класс `GraphParser`, в котором реализован статический метод `ParseTxtFormat`. Метод принимает массив строк, читает их и создает объект класса `Graph`, который возвращается в качестве результата. При этом если формат файла не удовлетворяет одному из описанных ранее требований метод выбрасывает исключение с сообщением о неверном формате данных, исключение отлавливается на уровне пользовательского интерфейса.

3.0.2 Запуск алгоритма на графах

При работе генетического алгоритма после запуска обхода в ширину такие результаты, как расстояние между вершинами и найденные кротчайшие пути сохраняются до окончания работы алгоритма, в связи с чем был создан класс `GraphContext`, который инкапсулирует структуры данных, хранящие описанные сведения, появляющиеся в процессе работы. Класс позволяет найти эксцентриситет вершин при помощи алгоритма поиска в ширину, а также позволяет проверять граф на связность с помощью этого же алгоритма. Расстояния, найденные в ходе процесса определения эксцентриситета хранятся в матрице $N \times N$, а кроме этого для хранения кротчайших путей используется такая же матрица, где каждым элементом является список вершин в пути. С полным кодом описанного класса можно ознакомиться в приложении [ссылка].

3.0.3 Генетический алгоритм

Сам генетический алгоритм реализован в классе `GeneticAlgorithmCore`. При создании объекта этого класса в конструктор передаются такие параметры, как объект типа `Graph`, значение параметра для уровня вероятности мутации, параметр для уровня вероятности скрещивания и размер популяции. После того, как объект этого класса создан у него можно вызвать метод `StartAlgorithm`:

```
public FindingVertexResponse StartAlgorithm() {
        Init();
        _watch.Start();
        for (int i = 0; i < _step; i++) {
                EvolutionStep();
        }
        _watch.Stop();

        FindingVertexResponse res = new FindingVertexResponse() {
                Time = _watch.ElapsedMilliseconds / (double)1000,
        };
        GetBestResult(res);
        return res;
}
```

в котором происходит вызов метода ответственного за начальную инициализацию популяции генетического алгоритма, а также за создание нового объекта класса `GraphContext`. После чего итерационно запускаются процессы мутации, скрещивания и естественного отбора. В качестве результата возвращается время работы алгоритма и найденные центральные вершины.

Кроме этого при добавлении нового графа необходимо выяснить его реальный радиус и с этой целью создан класс `ExactAlgorithmCore`, который запускает алгоритм обхода в ширину, из каждой вершины, что позволяет гарантированно найти точный радиус графа.

ЗАКЛЮЧЕНИЕ

Заключение

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Создание приложения Windows Forms с помощью .NET Framework (C++) [Электронный ресурс]. — URL: http://msdn.microsoft.com/ru-ru/library/vstudio/ms235634(v=vs.100).aspx (Дата обращения 13.07.2017). Загл. с экр. Яз. рус.

ПРИЛОЖЕНИЕ А

CD-диск с отчетом о выполненной работе

На приложенном диске можно ознакомиться со следующими файлами: