МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение

высшего образования

**«Дальневосточный федеральный университет»**

**ШКОЛА ЕСТЕСТВЕННЫХ НАУК**

**Кафедра прикладной математики, механики, управления и программного обеспечения**

**Отчет по лабораторной работе №1**

По дисциплине «Современные языки и системы программирования»

Специальность «Программная инженерия»

|  |
| --- |
| Выполнили студенты гр. М8119-09.04.04  \_\_\_\_\_\_\_ В. А. Титов |
| Проверил доцент кафедры ПММУиПО, к.т.н \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ С. Н. Остроухова  (зачтено/не зачтено) |

г. Владивосток

2020 г.

**Неформальная постановка задачи**

В зоопарке N есть M животных, которых необходимо распределить по клеткам для переезда. Некоторые представители зоопарка могут питаться друг другом. Необходимо распределить животных по клеткам так, чтобы никто из них никого не съел. При этом нескольких животных можно садить в одну клетку. Данные о пищевых цепочках предоставляет смотритель зоопарка. В них указано кто из животных кого может съесть.

Данный зоопарк может быть представлена графом, где дугами являются пищевые связи между животными, а вершины – животные. Необходимо при помощи объектно-ориентированной, функциональной и логической модели программирования:

1. Реализовать структуру графа, включающий функции:
2. конструктор и деструктор
3. добавление вершины и дуги
4. удаление вершины и дуги
5. раскраска вершин
6. Реализовать алгоритм неявного перебора
7. **Алгоритм неявного перебора**

Для определения хроматического числа графа может быть использован достаточно эффективно простой метод неявного перебора.

Предположим, что множество вершин как-то упорядочено и xi — i-я вершина этого множества. Тогда первоначальная допустимая раскраска может быть получена так:

* Окрасить xi в цвет 1.
* Каждую из оставшихся вершин окрашивать последовательно: вершина xi окрашивается в цвет с наименьшим возможным «номером» (т. е. выбираемый цвет должен быть первым в данном упорядочении цветом, не использованным при окраске какой-либо вершины, инцидентной xi). Далее улучшаем раскраску. Отыскиваем первую по номеру вершину с максимальным цветом. Из неё делаем шаг возвращения:
* Находим смежную вершину с максимальным номером, который меньше, чем у нее.
* Пытаемся перекрасить её в цвет больший собственного, но меньший, чем максимальный цвет в графе.
* Если это не удается, то делаешь шаг возвращения из этой вершины.
* Если это удаётся, то по правилу первой фазы перекрашиваем вершины с большим номером.
* Если при перекрашивании какая-то вершина затребовала максимальный цвет, от которого мы пытаемся избавиться, то делаем шаг возвращения из неё.
* Если достигнута первая вершина, то завершаем работу.

1. **Объектно-ориентированная парадигма**
   1. **Основные понятия**

Объектно-ориентированное программирование (ООП) позволяет разбить проблему на составные части, каждая из которых становится самостоятельным объектом. Каждый из объектов содержит свой собственный код и данные, которые относятся к этому объекту. Любая программа, написанная на языке ООП, отражает в своих данных состояние физических предметов либо абстрактных понятий – объектов программирования, для работы, с которыми она предназначена.

С объектом связывается набор действий, иначе называемых методами. С точки зрения языка программирования набор действий или методов – это функции, получающие в качестве обязательного параметра указатель на объект и выполняющие определенные действия с данными объекта программирования. Технология ООП запрещает работать с объектом иначе, чем через методы, таким образом, внутренняя структура объекта скрыта от внешнего пользователя.

Описание множества однотипных объектов называется классом.

Объект – это структурированная переменная, содержащая всю информацию о некотором физическом предмете или реализуемом в программе понятии.

Класс – это описание множества объектов программирования (объектов) и выполняемых над ними действий.

Любая функция в программе представляет собой метод для объекта некоторого класса. Класс должен формироваться в программе естественным образом, как только в ней возникает необходимость описания новых объектов программирования. Каждый новый шаг в разработке алгоритма должен представлять собой разработку нового класса на основе существующих.

Основными приципами ООП являются

* инкапсуляция;
* наследование;
* полиморфизм.

Инкапсуляция данных – это механизм, который объединяет данные и код, манипулирующий с этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования. В ООП код и данные могут быть объединены вместе (в так называемый «черный ящик») при создании объекта.

Наследование. Новый, или производный класс может быть определен на основе уже имеющегося, или базового класса. При этом новый класс сохраняет все свойства старого: данные объекта базового класса включаются в данные объекта производного, а методы базового класса могут быть вызваны для объекта производного класса, причем они будут выполняться над данными включенного в него объекта базового класса.

Полиморфизм – это свойство, которое позволяет один и тот же идентификатор (одно и то же имя) использовать для решения двух и более схожих, но технически разных задач.

* 1. **Обоснование выбора языка реализации**

Для реализации был выбран язык Java в первую очередь потому что, в Java нет средств, позволяющих писать не объектно-ориентированные программы. В Java реализованы все принципы объектно-ориентированного подхода:

* Все является объектом. Все данные программы хранятся в объектах. Каждый объект создается, существует какое-то время, потом уничтожается.
* Программа есть группа объектов, общающихся друг с другом. Кроме того, что объект хранит какие-то данные, он умеет выполнять различные операции над своими данными и возвращать результаты этих операций.
* Каждый объект имеет свою память, состоящую из других объектов и/или элементарных данных. Объект хранит некоторые данные.
* Каждый объект имеет свой тип (класс). В объектно-ориентированном подходе не рассматривается возможность создания произвольного объекта, состоящего из того, например, что мы укажем в момент его создания. Все объекты строго типизированы.
* Все объекты одного и того же типа могут получать одни и те же сообщения.
  1. **Описание класса**

**public class Graph** – класс, описывающий граф и содержащий методы, которые реализуют алгоритм минимальной покраски графа.

* private int maxC - максимальный цвет;
* private int remMaxC - дополнительная переменная, чтобы запомнить максимальный цвет;
* private int[] colors - массив цветов для вершин;
* private int[] remColors - для запоминания перед раскраской;
* private boolean[][] mat - матрица смежности графа;
* private int maxV - количество вершин;
* private int[] path – переменная, в которой хранится путь, пройденный методом shag.

**public Graph(boolean[][] mat)** – конструктор класса Graph. Принимает булевскую матрицу смежности и присваивает ее в переменную класса **mat**, также по количеству строк в матрице задается значение в переменную **maxV**, устанавливается размер массивов **colors** и **path**.

**public void paintMinVertex(int vertex, boolean repaint)** – метод, окрашивающий вершину в минимальный допустимый цвет. На вход подаются индекс вершины и булевская переменная **repaint**. При значении переменной true, начальный минимальный цвет при выборе из списка цветов, задается на один выше, чем есть у вершины в данный момент. Если значение false, то начальный минимальный цвет равен 1. При выполнении метода, новый цвет вершины заносится в переменную в массив **colors[vertex]**.

**public int getMaxC()** – метод проходит по списку цветов, находит максимальный цвет во всем графе и возвращает его номер.

**public void soutMat() –** вывод матрицы в консоль.

**private int shag(int cursor)** – рекурсивный метод, который реализует «шаг вверх по графу». На вход поступает положение «курсора» в массиве **path**, дальше алгоритм пытается сделать шаг назад из текущей вершины и если следующую вершину можно перекрасить, то метод перекрашивает и делает снова шаг, но теперь из нулевой вершины. Иначе, метод продолжает делать шаг назад и в определенных условиях рекурсия может прекратиться, например, когда вернулись в первую вершину, тогда алгоритм заканчивается.

**public void paint()** – основной метод, который использует методы **paintMinVertex()** и **shag()** для перекрашивания в минимальное количество цветов весь граф.

**public int[] getColors()** – метод, возвращающий переменную **colors.**

**public int getMaxV()** – метод, возвращающий переменную **maxV.**

**public int[] getPath()** – метод, возвращающий переменную **path.**

**public int[] getRemColors()** – метод, возвращающий переменную **remColors.**

**public int getRemMaxCol()** – метод, возвращающий переменную **remMaxC.**

**public void setMatrix(boolean[][] mat)** – метод принимает булевскую матрицу смежности и присваивает ее в переменную класса **mat**, также по количеству строк в матрице задается значение в переменную **maxV**, устанавливается размер массивов **colors** и **path**.

**public class Zoo –** класс, описывающий модель зоопарка с вольерами.

* **private String[] names** – массив имен животных по индексу;
* **private boolean[][] mat** – матрица связей животных. Если стоит **true**, то этих двух животных нельзя держать в одном вольере;
* **private int[M][N] avi** - матрица вольеров, где **M** означает количество вольеров, а **N** – количество животных, которых можно в нем держать;
* **private int cursor –** «курсор», который всегда стоит на свободном месте в **names** при обработке массива в методах сортировки животных по вольерам;
* **private int remIndex –** переменная, для запоминания индекса(во время поиска);
* **private int remAvi** – переменная, для запоминания вольера(во время поиска).

**public zoo() –** конструктор класса **Zoo**, создает матрицу **mat** размером N на N, где N – это количество животных, также создает **avi** и массив **names** с размерностью N, **cursor «**ставит» в значение 0.

**private void expand()** – метод, расширяющий список имен **names**, матрицу связей животных **mat** и матрицу вольеров **avi**.

**private String getNameAnim(int av, int index)** – метод, который возвращает имя животного, взятого по индексу из списка **names**.

**public void addInAvi(int av, int index)** – метод, добавляющий в первую свободную ячейку вольера номер **avi** животное по индексу **index** из списка имен **names**.

**private int searchIndex(String name, String[] arrayNames)** – поиск индекса животного в списке **names** по имени. Вторая параметр **arrayNames** позволяет передавать либо сохраненный список имен, который используется в алгоритме сортировки по вольерам, либо список имен класса **names**.

**private void delNameMat(int index)** – удаление животного по индексу только из **names**. Обнуление связей удаляемого животного из матрицы **mat**.

**public void searchAvi(String name)** – поиск животного по вольерам с помощью имени из списка **names**.

**public void delAvi(String name) –** удаление животного из вольера. Поиск в вольерах происходит по имени **name**.

**public int getFreeIndexAvi(int av) –** метод, который позволяет находить первую свободную ячейку в вольере номер **avi** в матрице вольеров.

**private void refresh(String[] remNames, boolean[][] remMat, int remCursor)** – восстановление списка имен **names**, позиции курсора **cursor** и матрицы связей животных **mat**, по сохраненным версиям **remNames**, **remMat**, **remCursor**.

**public void add(String name) –** метод, добавляющий животное **names** в список **names**, таким образом животное появляется в зоопарке.

**public void edge(String one, String two, boolean adOrDel)** – метод, который создает или удаляет связь между двумя животными в матрице связей. Если **adOrDel** = true, то создание связи у двух животных по именам **one**, **two** в матрице **mat**. Иначе значение в матрице связей между животными становится false, то есть связь удаляется.

**public void delAll(String name)** – метод, который убирает всю информацию о животным по имени **name** из списка **names,** матрицы **mat** и списка **avi.**

**public void distribution()** – метод, который создает по матрице связей **mat** граф в классе **graph**, где происходит минимальная раскраска графа. Дальше цвета каждой вершины графа становятся номерами вольера для каждого животного в матрице вольеров **avi**.

**public void soutNames()** – метод, выводящий удобный список всех животных **names** в консоль.

**public void soutMat()** – метод, выводящий удобное представление матрицы связей животных **mat** в консоль.

**public void soutAvi()** – метод, выводящий удобной представление матрицы вольеров **avi** в консоль.

1. **Тестирование**

| № | Описание тестовой ситуации | Исходное состояние графа | Входные данные | Выходные данные | |
| --- | --- | --- | --- | --- | --- |
| Раскраска | Сообщение |
| **Java** | | | | | |
| 1 | Выполнение алгоритма на пустом графе | - | z.add(**""**); | - |  |
| 2 | Выполнение алгоритма на графе с единственной вершиной (без дуг) | x1 | z.add(**"x1"**); | avi[0][0] = 0 |  |
| 3 | Выполнение алгоритма на графе с пустым списком вершин | x1- x4  x1- x6  x2- x3  x2- x4  x2- x5  x3- x5  x3- x6  x4- x6  x5- x6 | z.edge("x1","x4",true);  z.edge("x1","x6",true);  z.edge("x2","x3",true);  z.edge("x2","x4",true);  z.edge("x2","x5",true);  z.edge("x3","x5",true);  z.edge("x3","x6",true);  z.edge("x4","x6",true);  z.edge("x5","x6",true); | - | …  … |
| 4 | Успешное выполнение алгоритма на графе | x1-x3  x1-x8  x2-x4  x2-x7  x3-x4  x5-x6  x5-x8  x6-x7 | z.add("x1");  z.add("x2");  z.add("x3");  z.add("x4");  z.add("x5");  z.add("x6");  z.add("x7");  z.add("x8"); z.edge("x1","x3",true);  z.edge("x1","x8",true);  z.edge("x2","x4",true);  z.edge("x2","x7",true);  z.edge("x3","x4",true);  z.edge("x5","x6",true);  z.edge("x5","x8",true);  z.edge("x6","x7",true); | avi[0][0] = 0  avi[1][1] = 1  avi[1][2] = 2  avi[0][3] = 3  avi[0][4] = 4  avi[1][0] = 5  avi[0][0] = 6  avi[1][0] = 7 | …  …    …  …    …  … |

# **Вывод**

Были рассмотрены три парадигмы программирования для реализации алгоритма точной раскраски графа методом неявного перебора. Как видно из работы, данный алгоритм можно реализовать на всех трех парадигмах. Однако, мы считаем, что функциональная парадигма является более подходящей для реализации этого алгоритма. Так как алгоритм является рекурсивным, функциональная парадигма позволяет реализовать его более эффективно и элегантно, чем в других парадигмах.