
Руководство к выполнению расчетной работы по курсам ОИИ и ППВИС

Автор:
Д.А. Лазуркин

19 февраля 2013 г.

Оглавление

I	Курс ОИИ	3
	Введение	4
	Перечень теоретико-графовых задач	5
1	Разработка программы решения теоретико-графовой задачи	9
1.1	Задание	9
1.2	Литература	9
1.3	Выбор среды разработки	10
1.4	Программа волнового поиска минимального пути	11
1.4.1	Описание алгоритма	11
1.4.2	Тестовые примеры	11
1.4.3	Создания программы	11
2	Построение фрагмента онтологии	13
2.1	Задание	13
2.2	Формализация понятий «неориентированный» и «ориентированный граф»	14
2.3	Формализация понятия «минимальный путь»	18
2.4	Обобщение различных видов графов с использованием понятия графовой структуры	23
2.5	Пример выполнения	26
2.5.1	Список понятий	26
2.5.2	Тестовые примеры	38
3	Демонстрация работы программы решения теоретико-графовой задачи в семантической памяти	43
3.1	Задание	43
3.2	Волновой алгоритм поиска одного из минимальных путей в неориентированном графе	43
3.2.1	Описание алгоритма	43
3.2.2	Пример выполнения алгоритма в sc-памяти	44
II	Курс ППВИС	62
	Введение	63
4	Разработка программы решения теоретико-графовой задачи на языке программирования C++ с использованием семантической памяти	64
4.1	Задание	64
4.2	Установка и настройка рабочей среды	64
4.3	Назначение и структура модуля sc-core	69
4.4	Библиотека моделирования sc-памяти — libsc	70
4.4.1	Общие сведения	70

4.4.2	Сегментная модель sc-памяти	71
4.4.3	Начало работы	72
4.4.4	Тип sc-элемента	74
4.4.5	Основные обрабатываемые sc-конструкции	77
4.4.6	Генерация и удаление sc-конструкций в sc-памяти	77
4.4.7	Работа с содержимым sc-элементов	83
4.4.8	Поиск в sc-памяти	85
4.4.9	Высокоуровневая работа с основными абстракциями в sc-памяти	92
4.4.10	Вывод на консоль sc-конструкций	99
4.5	Создание программы wave_find_path с использованием sc-памяти	100
4.5.1	Инициализация и подготовка sc-памяти	100
4.5.2	Загрузка неориентированного графа	103
4.5.3	Вывод на консоль неориентированного графа	106
4.5.4	Запуск тестов алгоритма	109
4.5.5	Реализация алгоритма	111
4.5.6	Вывод маршрута на консоль	123

Литература	126
-------------------	------------

Часть I

Курс ОИИ

Введение

Расчетная работа по курсу «Основы искусственного интеллекта» состоит из следующих этапов:

- разработка программы решения теоретико-графовой задачи на языке программирования C++ (см. раздел [1](#))
- построение фрагмента онтологии теоретико-графовой задачи (см. раздел [2](#))
- демонстрация работы программы решения теоретико-графовой задачи в семантической памяти (см. раздел [3](#))

Все используемые в этой части материалы, исходные тексты, установочные файлы могут быть найдены на кафедральном информационном сервере по следующему пути:

\\Info\StudInfo\~Методическое обеспечение кафедры\~Учебные курсы\1 курс\ОИИ\Расчётная работа\

Перечень теоретико-графовых задач

1. Определить вид графа:

1. (1) Дерево (нг, ог)
2. (1) Ациклический граф (нг, ог)
3. (1) Полуэйлеров/эйлеров граф (нг)
4. (3) Гамильтонов граф (нг)
5. (0) Полный граф (нг)
6. (2) Связный граф (нг)
7. (3) Сильно-связный граф (ог)
8. (2) Двусвязный граф
9. (2) Двудольный неориентированный граф
10. (2) Регулярный, реберно-регулярный неориентированный граф
11. (2) Симметричный, антисимметричный, частично симметричный орграф
12. (2) Транзитивный, антитранзитивный, частично транзитивный орграф
13. (2) Рефлексивный, антирефлексивный, частично рефлексивный орграф
14. (2) Функциональный, контрафункциональный орграф
15. (1) Вычерчиваемый граф
16. (2) Односторонне связный орграф
17. (4) Кактус
18. (1) Турнир, транзитивный турнир (ог)
19. (1) Граф Бержа (ог)
20. (3) Граф Паппа (нг)
21. (4) Планарный граф (нг, ог)
22. (3) Транзитируемый граф (нг)

2. Определить числовую характеристику графа:

1. (2) Радиус (нг, ог, внг, вог)
2. (2) Диаметр (нг, ог, внг, вог)
3. (2) Средний диаметр (нг, ог, внг, вог)
4. (1) Полустепени захода/исхода и средние полустепени всех вершин в орграфе
5. (1) Минимальную степень/среднюю степень/максимальную степень ребра в неориентированном графе
6. (4) Число вершинной связности (нг, ог)
7. (4) Число рёберной связности (нг, ог)
8. (2) Среднее и максимальное расстояние между центральными вершинами неориентированного графа
9. (3) Число хорд неориентированного графа
10. (2) Минимальное и среднее расстояние между периферийными вершинами неориентированного графа

11. (0) Цикломатическое число неориентированного графа
12. (3) Окружение орграфа
13. (3) Обхват орграфа
14. (2) Число компонентов связности неориентированного графа
15. (3) Число Хадвигера для неориентированного графа
16. (5) Определить толщину неориентированного графа
17. (4) Индекс компонент относительно простой цепи в неориентированном графе

3. Построение графа:

1. (4) Сгенерировать клетку указанного обхвата
2. (?) Найти граф, являющийся пересечением множества всех диаметральных цепей неориентированного графа
3. (?) Найти граф, являющийся объединением всех радиусов графа
4. (?) Найти граф, являющийся объединением множества всех диаметров графа
5. (?) Найти граф, являющийся пересечением множества всех гамильтоновых циклов графа
6. (?) Найти граф, являющийся объединением множества всех гамильтоновых циклов графа

4. Операции над графами:

1. (2) Найти декартово произведение двух неориентированных графов
2. (2) Найти декартову сумму двух неориентированных графов
3. (2) Найти прямое (тензорное) произведение двух неориентированных графов
4. (2) Найти сильное произведение двух неориентированных графов
5. (2) Найти композицию двух неориентированных графов
6. (2) Найти модульное произведение двух неориентированных графов
7. (2) Найти большое модульное произведение двух неориентированных графов
8. (2) Найти объединение множества неориентированных графов
9. (2) Найти пересечение множества неориентированных графов
10. (2) Найти дополнение и фактор-дополнение неориентированного графа
11. (2) Найти граф инцидентий неориентированного графа
12. (2) Найти реберный граф для неориентированного графа
13. (2) Найти граф смежностей для неориентированного графа
14. (2) Найти тотальный граф для неориентированного графа
15. (2) Найти граф замыкания неориентированного графа
16. (4) Найти граф конденсации для орграфа
17. (4) Найти граф каркасов для неориентированного графа

5. Поиск в графе:

1. (2) Определить эксцентриситет каждой вершины в неориентированном графе

2. (3) Найти эйлеров цикл в графе (нг, ог)
 3. (3) Найти гамильтонов цикл (нг, ог)
 4. (3) Найти компоненты связности в неориентированном графе
 5. (3) Найти сильные компоненты связности в орграфе
 6. (0) Найти вершины с указанной степенью
 7. (3) Найти минимальный остов в неориентированном графа
 8. (3) Найти доли неориентированного графа
 9. (0) Найти тупики/антитупики в ориентированном графе
 10. (3) Найти максимальный простой разрез (нг, ог, внг, вог)
 11. (3) Найти минимальный простой разрез (нг, ог, внг, вог)
 12. (?) Найти все пары вершин с указанным расстоянием (нг, ог, внг, вог)
 13. (?) Найти множество рёбер, удаление которых приводит к увеличению числа компонентов связности ориентированного графа
 14. (3) Найти точки сочленения неориентированного графа
 15. (3) Найти мосты в неориентированном графе
 16. (5) Найти множество вершин, удаление которых приводит к увеличению числа компонентов связности ориентированного графа
 17. (3) Найти циклы указанной длины (нг, ог, внг, вог)
 18. (3) Найти максимальный путь между заданными вершинами (нг, ог, внг, вог)
 19. (?) Нахождение критических путей во взвешенном неориентированном графе
 20. (?) Найти множество вершин, удаление которых приводит к увеличению числа компонентов связности неориентированного графа
 21. (3) Найти простые цепи указанной длины (нг, ог, внг, вог)
 22. (?) Найти вершины с указанной полустепенью
 23. (1) Найти все доминирующие вершины (нг, ог, внг, вог)
 24. (?) Найти центры графа
 25. (?) Найти рёбра с указанной степенью
 26. (2) Найти все периферийные вершины (нг, ог, внг, вог)
 27. (?) Найти множество рёбер, удаление которых приводит к увеличению числа компонентов связности неориентированного графа
 28. (?) Найти звёзды с заданным числом листьев
 29. (4) Найти дерево кратчайших путей (нг, ог, внг, вог)
 30. (?) Определить рёбра и их степени в неориентированном мультиграфе
 31. (?) Найти n -фактор для указанного графа
 32. (5) Поиск подграфов в неориентированном графе, изоморфных графу-образцу
 33. (3) Сформировать множество различных суграфов неориентированного графа
 34. (3) Сформировать множество различных подграфов неориентированного графа
6. Приведение графа к указанному виду:

1. (3) Найти минимальное множество вершин неориентированного графа, удаление которых позволяет сделать его деревом
2. (3) Найти минимальное множество рёбер неориентированного графа, удаление которых позволяет сделать его деревом
3. (5) Найти минимальное множество вершин неориентированного графа, удаление которых позволяет сделать его планарным
4. (5) Найти минимальное множество рёбер графа, удаление которых позволяет сделать его планарным

Глава 1

Разработка программы решения теоретико-графовой задачи

1.1 Задание

При выполнении этого этапа расчетной работы студенту необходимо разработать и реализовать программу на C/C++, которая решает выданную преподавателем теоретико-графовую задачу и соответствует следующим требованиям:

- если преподаватель указал определенную структуру данных для представления графа, то в программе должна быть использована именно эта структура данных;
- при своем запуске программа должна в автоматическом режиме стартовать 5 разнообразных примеров решения задачи и выводить результаты работы на консоль;
- в программе должна оптимальным образом использоваться возможность языка программирования C/C++ по созданию функций;
- в программе нельзя использовать глобальные переменные.

1.2 Литература

Начиная знакомство с полученной теоретико-графовой задачей, необходимо поискать понятия, входящие в формулировку задачи, в [толковом словаре по теории графов](#). В определении понятия в этом словаре есть ссылки на соответствующую литературу. Также можно использовать словари по теории графов на [Wikipedia](#) и [mportal.narod.ru](#).

Основной литературой для выполнения расчетной работы является следующий перечень книг:

- Оре О. Теория графов. — 2-е изд.. — М.: Наука, 1980. — С. 336.
- Кормен Т. Х. и др. Часть VI. Алгоритмы для работы с графами // Алгоритмы: построение и анализ = Introduction to Algorithms. — 2-е изд.. — М.: Вильямс, 2006. — С. 1296.
- Харари, Ф. Теория графов / Ф. Харари / Пер. с англ. и предисл. В.П. Козырева. Под ред. Г.П. Гаврилова. Изд. 2-е. — М.: Едиториал УРСС, 2003. — 269 с.
- Нечипуренко, М. И. Алгоритмы и программы решения задач на графах и сетях / М.И. Нечипуренко, В.К. Попков, С.М. Майнагашев и др. — Новосибирск: Наука. Сиб. отд-ние, 1990. — 515 с. v

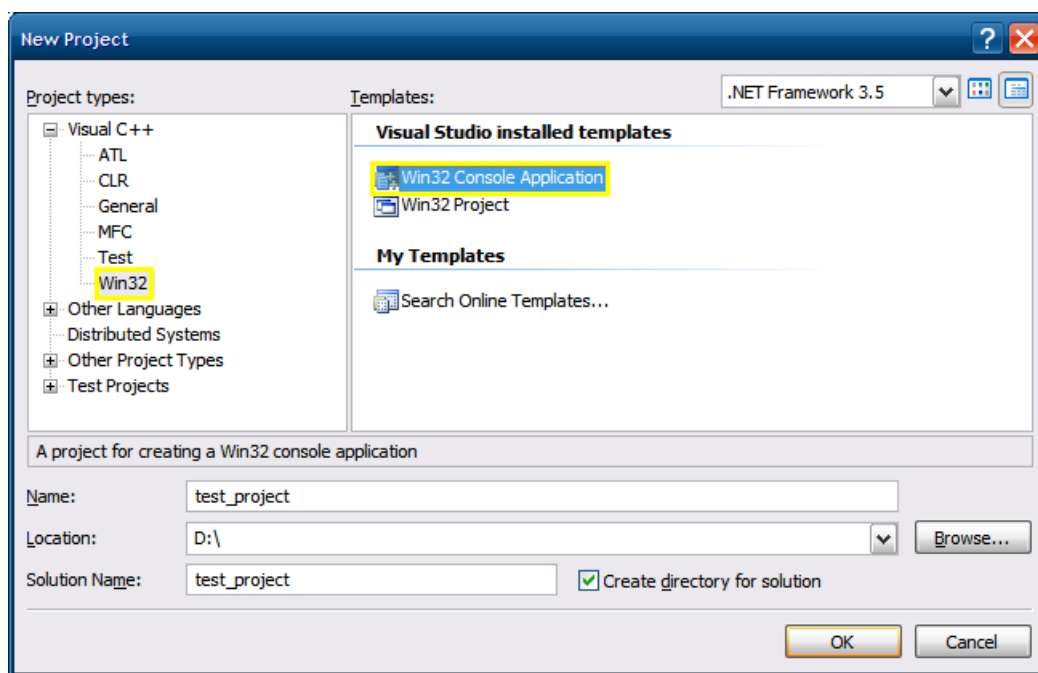
- Емеличев В. А., Мельников О. И., Сарванов В. И., Тышкевич Р. И. Лекции по теории графов. М.: Наука, 1990. 384с. (Изд.2, испр. М.: УРСС, 2009. 392 с.)
- Касьянов, В. Н. Графы в программировании: обработка, визуализация и применение / В. Н. Касьянов, В. А. Евстигнеева. – СПб. : БХВ-Петербург, 2003.

1.3 Выбор среды разработки

Рекомендую студентам в качестве среды разработки выбрать MS Visual Studio. На мой взгляд, самым оптимальным выбором здесь будет MS Visual Studio 2008 (или 9.0, что есть одно и то же). Образ диска-установщика находится в папке \\606srv\Install\~Development\~IDE\~MS Visual Studio и называется «MSVS_MSDN2008.iso». На кафедре в учебных лабораториях установлены MS Visual Studio 2003, 2008 и 2010.

Давайте рассмотрим процесс создания проекта в MS Visual Studio. Запустите эту среду разработки и выберите пункт меню File -> New -> Project. В диалоге «New Project» вас будет интересовать тип проекта «Win32 Console Application» (как показано на рисунке ниже). Выберите этот тип проекта и выведите название создаваемого проекта и директорию, в которой он будет создан. Нажимайте «ОК» и будет открыт мастер создания данного типа проекта. В нем можно просто нажать на кнопку «Finish». Теперь с функции `_tmain` (это аналог знакомой вам функции `main`) можно начинать писать программу.

Обратите внимание! Заголовочный файл `stdafx.h` является [предкомпилированным](#), поэтому во всех сpp-файлах инструкция `#include "stdafx.h"` должна быть в самом начале файла (раньше нее можно помещать только комментарии).



Чтобы собрать проект, выберите пункт меню Build -> Build Solution (Ctrl + Shift + B), затем выберите Debug -> Start Without Debugging (Ctrl + F5) для запуска. Если хотите запустить под отладкой, то выбирайте Debug -> Start Debugging (F5).

1.4 Программа волнового поиска минимального пути

1.4.1 Описание алгоритма

Алгоритм поиска одного из минимальных путей в неориентированном графе является волновым и основан на понятии волны. В рамках этого алгоритма волной называется множество вершин, каждая из которых является в обрабатываемом графе смежной хотя бы одной вершине из предыдущей волны. Волна, для которой нет предыдущей волны, называется начальной и состоит из вершины, от которой начинается поиск минимального пути. Волна, включающая конечную вершину пути, называется конечной. Таким образом, наш алгоритм можно задать следующим перечнем шагов:

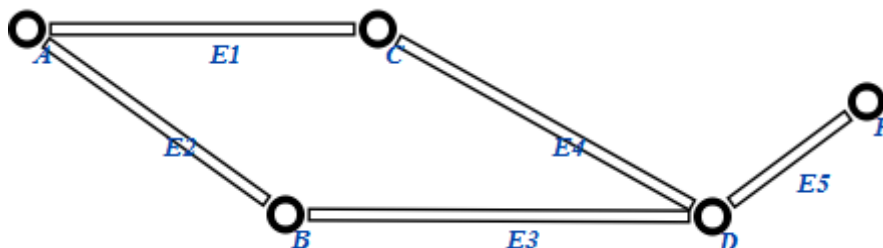
1. Добавить все вершины графа, кроме начальной вершины пути, во множество непроверенных вершин.
2. Создать новую волну и добавить в нее начальную вершину пути.
3. Начальная волна – это новая волна. Новой волной будем называть последнюю созданную волну.
4. Сформировать следующую волну для новой волны. В нее попадет та вершина, которая является смежной вершине из новой волны и присутствует во множестве непроверенных вершин. Если вершина попала в формируемую волну, то ее надо исключить из множества непроверенных вершин. Созданную волну установить как следующую для новой волны, и после этого созданную волну считать новой волной.
5. Если новая волна пуста, то между вершинами не существует пути. Завершить алгоритм.
6. Если в текущей волне есть конечная вершина, то перейти к пункту 7, иначе к пункту 4.
7. Сформировать один из минимальных путей, проходя в обратном порядке по списку волн. Завершить алгоритм.

1.4.2 Тестовые примеры

1. graph1.txt



2. graph2.txt

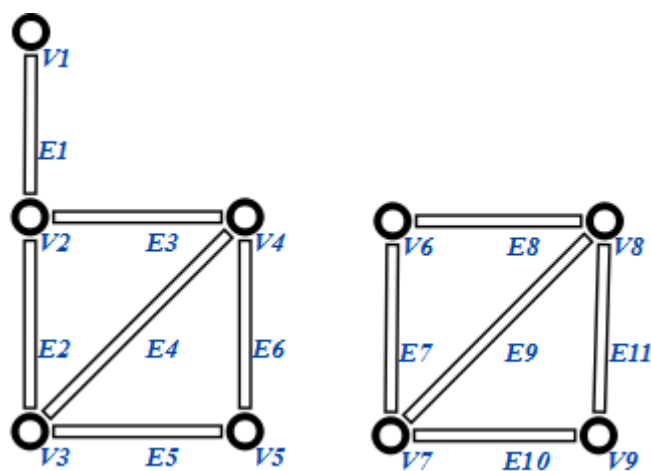
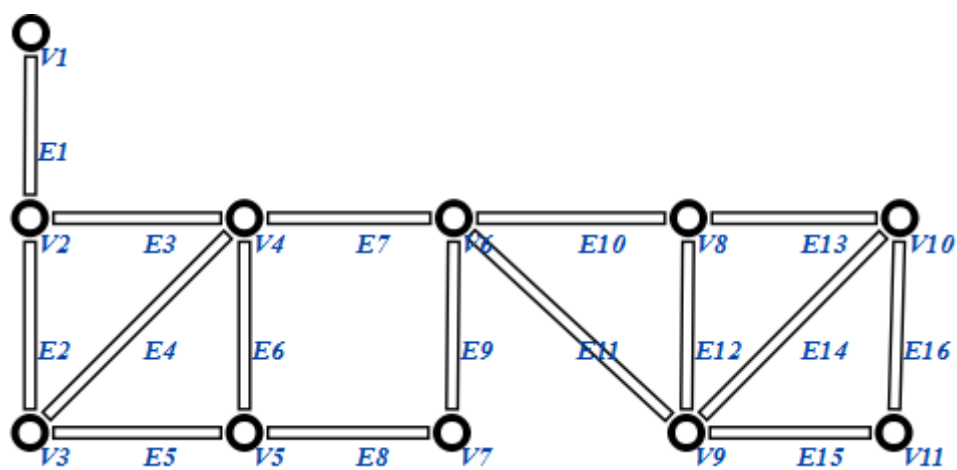
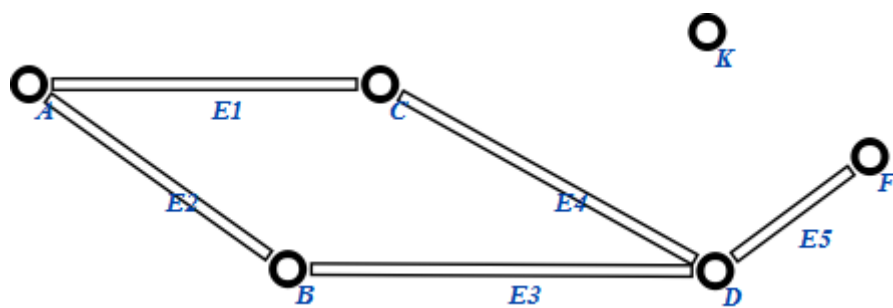


3. graph3.txt

4. graph4.txt

5. graph5.txt

1.4.3 Создания программы



Глава 2

Построение фрагмента онтологии

Для понимания этого раздела студент должен обладать следующими знаниями:

- базовые знания по теории графов
- SC-коде и SC-алфавите
- базовые знания о SCg-коде
- уметь различать абсолютные и относительные понятия
- знание правил формирования идентификаторов sc-элементов

2.1 Задание

Во-первых, что должны сделать вы, студент, при выполнении этого этапа расчетной работы, это выделить абсолютные и относительные понятия своей теоретико-графовой задачи. Список понятий должен включать не только непосредственно необходимые для решения вашей задачи, но и те понятия, на основе которых определены непосредственно необходимые. Таким образом, будет выделена целая иерархия понятий.

Во-вторых, для каждого понятия разработать способ представления его экземпляров в SC-коде. Подготовить отчет в электронном варианте о проделанной работе, который должен содержать следующее (пример отчета приведен в разделе 2.5):

1. перечень выделенных понятий со следующей информацией:
 - (a) название понятия;
 - (b) абсолютное или относительное;
 - (c) определение на естественном языке (его стоит брать из соответствующей литературы);
 - (d) пример представления экземпляра понятия в SC-коде на SCg (не маленький и не большой, среднего размера).
2. пять примеров на SCg входных и выходных данных для вашей программы (примеры нужно брать из предыдущего этапа расчетной работы и можно использовать сокращенную форму записи графов).

Для рисования SCg-текстов необходимо использовать редактор КБЕ (Knowledge Base Editor). Установщик и zip-архив собранной версии под Windows можно найти в той же папке на сервере Info, откуда вы взяли этот документ.

Мы плавно переходим к примеру выполнения 2-го этапа...

Выполненный мною пример отчета по этому этапу в dosx-формате можно найти в папке расчетной работы на сервере Info либо в конце этого документа. Напомню, что мы рассматриваем задачу поиска одного из минимальных путей в неориентированном графе. Поэтому первым, чем мы займемся, будет способ представления в SC-коде неориентированных и ориентированных графов.

2.2 Формализация понятий «неориентированный» и «ориентированный граф»

Возьмем для рассмотрения неориентированный граф G :

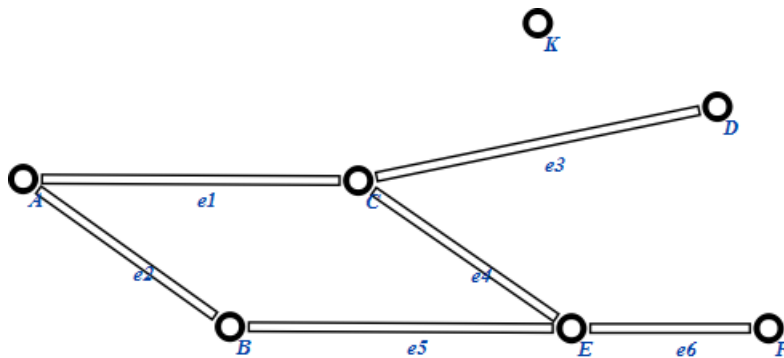


Рис. 2.1: Неориентированный граф G (это не SCg)

Распишем граф G классическим способом на языке теории множеств:

$$\begin{aligned} G &= \langle V_g, E_g \rangle; \\ V_g &= \{A, B, C, E, D, F, K\}; \\ E_g &= \{\{A, B\}, \{A, C\}, \{C, E\}, \{C, D\}, \{B, E\}, \{E, F\}\}. \end{aligned} \tag{2.1}$$

Теперь попробуем перевести запись, приведенную выше, в SC-код. Все sc-конструкции я буду приводить на SCg. Для представления неориентированного графа в SC-коде введем абсолютное понятие *неориентированный граф* (в дальнейшем я буду использовать именно такое форматирования для идентификаторов sc-элементов в тексте). Теперь мы можем преобразовать запись на языке теории множеств, задающую граф G , в SCg-конструкцию, которая изображена на рис. 2.2.

Приведенный выше способ задания графа на языке классической теории множеств является распространённым, но мы, с использованием SC-кода, можем найти другую форму, так как имеем возможность задавать ролевые отношения. Поэтому введем два относительных понятия (ролевых отношения): *вершина_* и *ребро_*. Тогда можно сформулировать, что неориентированный граф задается множеством объектов, в котором объект с ролью *вершина_* является вершиной графа, а объект с ролью *ребро_* - ребром графа. На языке теории множеств, расширенном возможностью задавать атрибуты (роли) у элементов кортежа, граф G будет задаваться выражением (2.2).

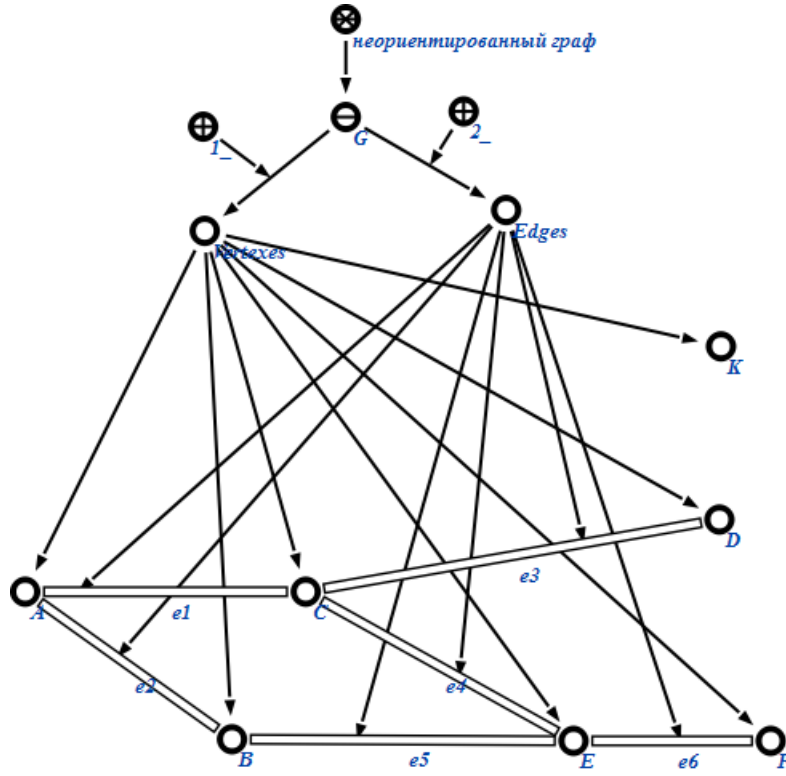


Рис. 2.2: Классический способ задания неориентированного графа G на SCg

$$\begin{aligned}
 G = \langle & \text{вершина_} : A, \text{вершина_} : B, \text{вершина_} : C, \\
 & \text{вершина_} : E, \text{вершина_} : D, \text{вершина_} : F, \text{вершина_} : K, \\
 & \text{ребро_} : \{A, B\}, \text{ребро_} : \{A, C\}, \text{ребро_} : \{C, E\}, \\
 & \text{ребро_} : \{C, D\}, \text{ребро_} : \{B, E\}, \text{ребро_} : \{E, F\} \rangle.
 \end{aligned} \tag{2.2}$$

Переведем выражение (2.2) в SCg, что показано на рис. 2.3.

Для студента, знакомого с теорией множеств и SCg-языком, в рис. 2.3 необходимо пояснить только тип узла с идентификатором G . Такой тип SCg-узла используется для обозначения sc-структуры, т.е. множества объектов, которые могут выступать как целое. Объекты, которые входят в sc-структуру, при совместном рассмотрении порождают некоторое новое качество. Попробуем изучить это через сравнение множества *неориентированный граф* с множеством G , которое является конкретным неориентированным графом.

Множество *неориентированный граф* – является sc-понятием (к слову, абсолютным), т.е. sc-множеством, все элементы которого обладают некоторым заданным свойством. В случае sc-понятия *неориентированный граф* его элементы должны обозначать неориентированные графы. Предположим, в множестве *неориентированный граф* у нас есть 5 элементов (неориентированных графов). Добавим в это множество еще 5 неориентированных графов. При добавлении элементов мощность множества изменилась, но качественно множество *неориентированный граф* осталось таким же. Этот факт лаконично можно выразить следующим образом: количество элементов множества, которое является sc-понятием, никогда не переходит в качество.

Совсем по-другому обстоят дела с множеством G . Это sc-множество, составлено из sc-элементов, которые вместе обладают некоторой целостностью, имеющей важные свойства (они образуют неориентированный граф только в том случае, если рассмотрены как единое целое). Если мы добавим в множество G новый элемент (вершину или ребро), то получим уже другой граф. Таким образом, можно заключить, что количество элементов для множество G переходит

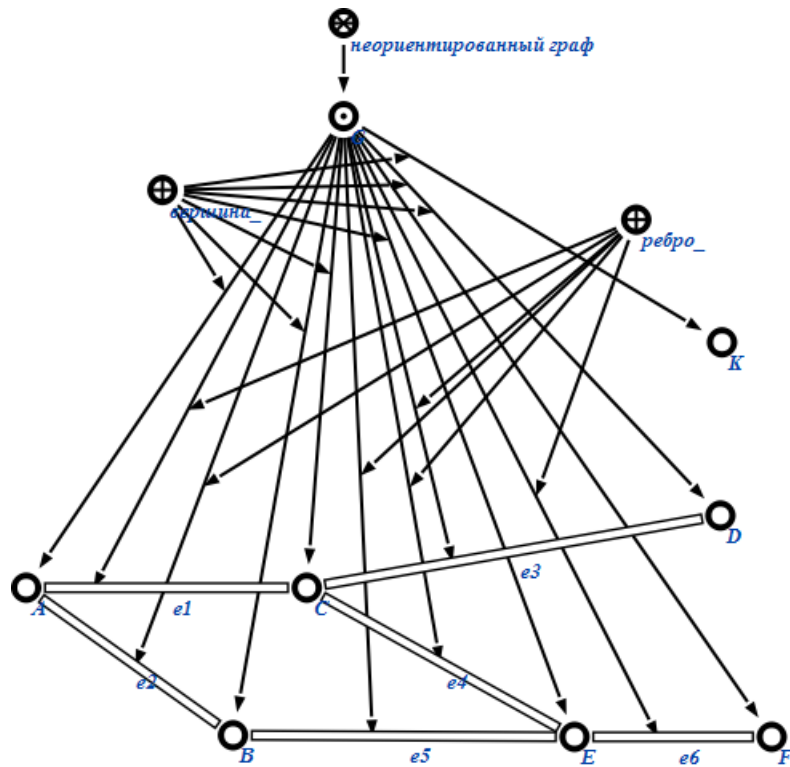


Рис. 2.3: Основной для нас способ задания графа G на SCg

в качестве. Множества, для которых выполняется это свойство, являются sc-структурами.

Продолжая разговор о структурах, стоит сказать, что связки так же являются структурами. Однако, очевидно, что неориентированный граф G не только обозначает сам факт существования связи между объектами, но и включает связи между своими собственными элементами. Такие структуры, как граф G не являются связками, и мы будем называть их одноуровневыми реляционными структурами. Способ представления графов в виде одноуровневых реляционных структур используется в проекте базы знаний по теории графов технологии OSTIS (OSTIS GT), поэтому ниже рассматривается только этот способ, и при выполнении расчетной работы должен использоваться только он.

Продолжим рассмотрение того, как кодировать графы на sc-языках. Давайте попробуем представить ориентированный граф на SCg. Преобразуем неориентированный граф G в ориентированный граф G_d (рис. 2.4).

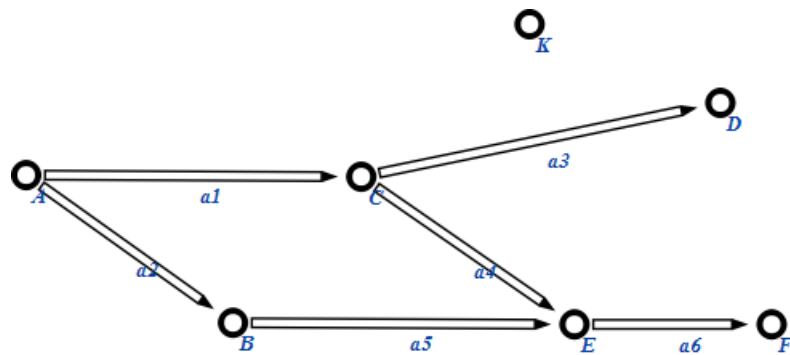


Рис. 2.4: Ориентированный граф G_d (это не SCg)

Для представления G_d на SCg введем абсолютное понятие *ориентированный граф* и относительное понятие (ролевое отношение) *дуга_*. С использованием этих понятий граф G_d на SCg

можно представить так, как показано на рис. 2.5.

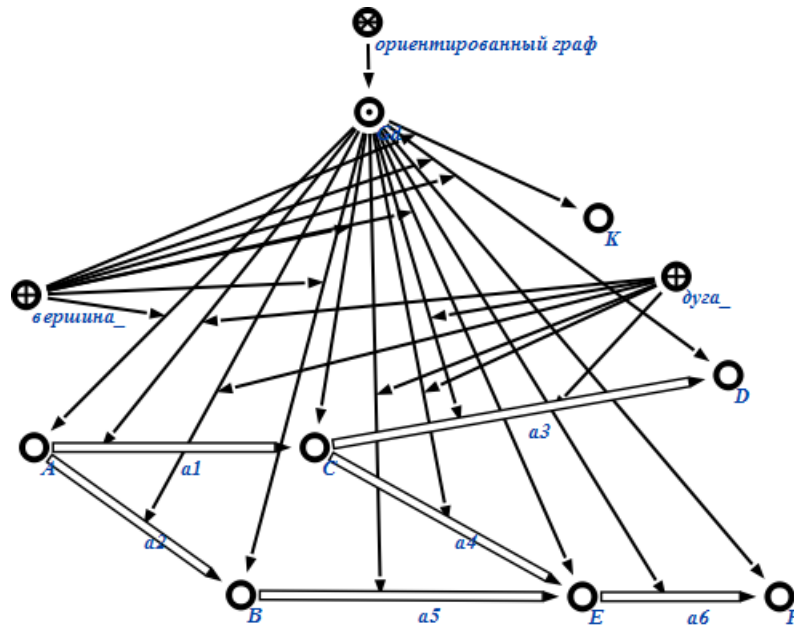


Рис. 2.5: Ориентированный граф G_d в SCg

Сравните запись графа G (рис. 2.3) на SCg и графа G_d (рис. 2.5). Разница только в использовании:

- узла *ориентированный граф* вместо *неориентированный граф*;
- узла *дуга_* вместо *ребро_*;
- ориентированных бинарных пар вместо неориентированных.

Можно заметить, что графы на SCg с использованием ролевых отношений *вершина_* и *ребро_* получаются очень громоздкими, поэтому в дальнейших объяснениях для представления неориентированных и ориентированных графов мы будем использовать сокращенную запись. С использованием сокращенной записи граф G , заданный на рис. 2.3, мы будем задавать так, как показано на рис. 2.6. Надеюсь: читатель понимает, что эта запись предназначена для восприятия человеком, а не машиной, потому что для машины является важным наличие опущенных атрибутов.

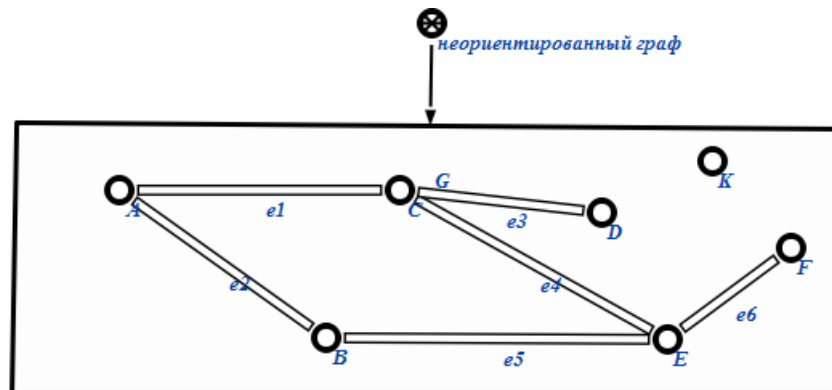


Рис. 2.6: Сокращенная форма задания графа G на SCg Мы определились,

Мы рассмотрели способ формализации с использованием семантических сетей неориентированных и ориентированных графов. Теперь перейдем к формализации минимального пути - второго понятия, без которого не обойтись при решении нашей задачи.

2.3 Формализация понятия «минимальный путь»

Если читатель заглянет в [3] на страницу 26, то в начале главы «Маршруты и связность» он может прочитать следующее:

Маршрутом в графе G называется чередующаяся последовательность вершин и ребер $v_0, x_1, v_1, \dots, v_{n-1}, x_n, v_n$; эта последовательность начинается и кончается вершиной, и каждое ребро последовательности инцидентно двум вершинам, одна из которых непосредственно предшествует ему, а другая непосредственно следует за ним. Указанный маршрут соединяет вершины v_0 и v_n , и его можно обозначить v_0, v_1, \dots, v_n (наличие ребер подразумевается). ... Маршрут называется **цепью**, если все его ребра различны, и **простой цепью** (путем), если все вершины (а, следовательно, и ребра) различны.

Таким образом, путь — это маршрут, поэтому сейчас мы будем рассматривать именно это более общее понятие. Если разберемся с представлением его в SC-коде, то разберемся и с более частным понятием.

Я думаю, что для читателя очевидно следующее: маршрут — это относительное понятие, так как конкретный маршрут существует в связи с конкретным графом. Поэтому для представления маршрутов введем бинарное ориентированное отношение *маршрут**. Первым компонентом связки этого отношения будет знак графа, а вторым знак структуры маршрута. Пример связки приведен на рис. 2.7.

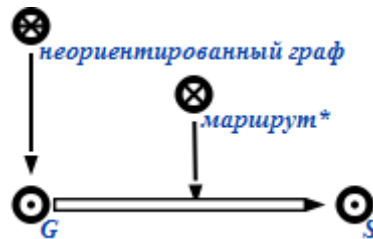


Рис. 2.7: Пример связки отношения *маршрут**

Теперь нам необходимо выяснить из чего же состоит структура S на рис. 2.7. Для этого рассмотрим в графе G (рис. 2.6) маршрут R между вершинами A и F , который задается последовательностью $A, e_2, B, e_5, E, e_6, F$ (это один из минимальных путей между A и F). Если маршрут состоит из вершин и ребер, то его можно представить как подграф графа, на котором задается маршрут. Посмотрите внимательно на рис. 2.8.

Наверное, вы обратили внимание, что на рис. 2.8 появилось бинарное ориентированное отношение *подграф**, суть которого есть связывание одного графа с другим графом, который является подграфом первого. А еще из рис. 2.8 вы можете заметить, что отношение *включение** включает отношение *подграф**, т.е. *подграф** - аналог *включения**, но только для графов. А наше отношение *маршрут** является подмножеством отношения *подграф**. Таким образом, граф структуры S является подграфом исходного графа G .

Возможно, вы посчитали, что на этом наш разговор о способе представления маршрутов можно закончить, но я попрошу вас не торопиться, а попробовать нарисовать SCg-конструкцию, которая задаст маршрут $A, e_2, B, e_5, E, e_4, C, e_1, A, e_2, B$. Ну как? Получилось? Хоть это и

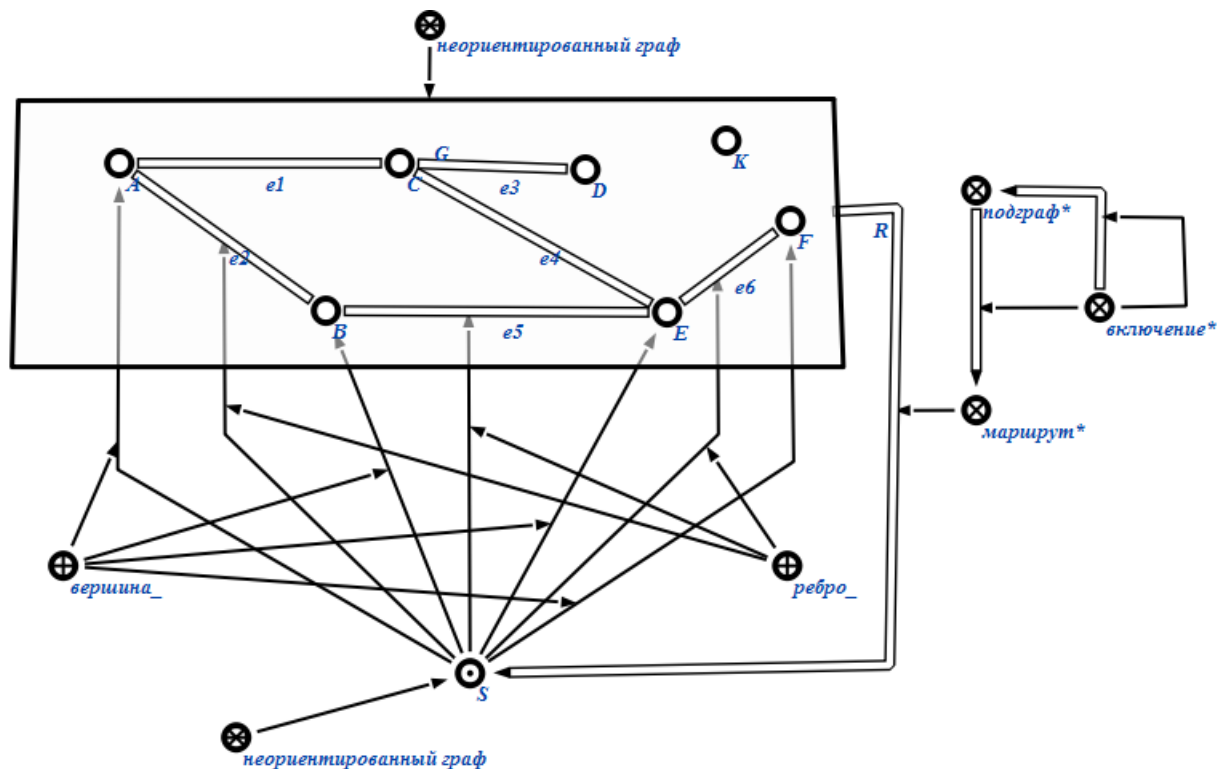


Рис. 2.8: Представление маршрута как подграфа графа G

не путь, потому что вершины и ребра повторяются, но это вполне допустимый маршрут. А все из-за повторяющихся вершин и ребер. На рис. 2.9 повторяющиеся вершины и ребра выделены красным цветом. Поэтому мы опять переходим к поиску способа задать структуру второго компонента связки отношения маршрут^* (структура S на рисунке 2.7).

Вернемся к тому, что мы решили рассматривать маршрут R в графе G . Тогда в основу нашего способа представления мы положили, что маршрут состоит из вершин и ребер и он есть подграф графа, на котором задается. Может быть, слабость этого подхода была в том, что мы не рассматривали маршрут как последовательность вершин и ребер, т.е. мы не задали порядок? Давайте попробуем задать последовательность элементов маршрута.

Как мы можем задать последовательность? Если без введения лишних понятий, то, например, при помощи ориентированного графа. Вершинами такого графа будут элементы последовательности, а дуги будут задавать отношение предыдущий/следующий. Первый элемент последовательности специально указывать не будем, потому что первым элементом является вершина, в которую нет входящих дуг. Последним элементом последовательности будет вершина, из которой нет исходящих дуг. Этот способ представления маршрута R показан на рис. 2.10 (синим цветом выделены элементы маршрута, а зеленым - связки следующий/предыдущий).

Но, даже используя новый способ представления, мы не сможем задать маршрут, который не является цепью или простой цепью (путем). Новый способ представления все равно ограничен. Предлагаю вам увидеть это самостоятельно, представив маршрут с повторяющимися вершинами и ребрами. Поэтому продолжим наш поиск.

Давайте еще раз взглянем на определение маршрута из [3]:

Маршрутом в графе G называется чередующаяся последовательность вершин и ребер $v_0, x_1, v_1, \dots, v_{n-1}, x_n, v_n$;

В нем сказано, что маршрут состоит из вершин и ребер. Но, быть может, это не совсем так? Мы нашли два ограниченных способа для представления маршрута, пользуясь буквально этим определением, а задачу еще не решили. Может быть, стоит сказать, что маршрут состоит не

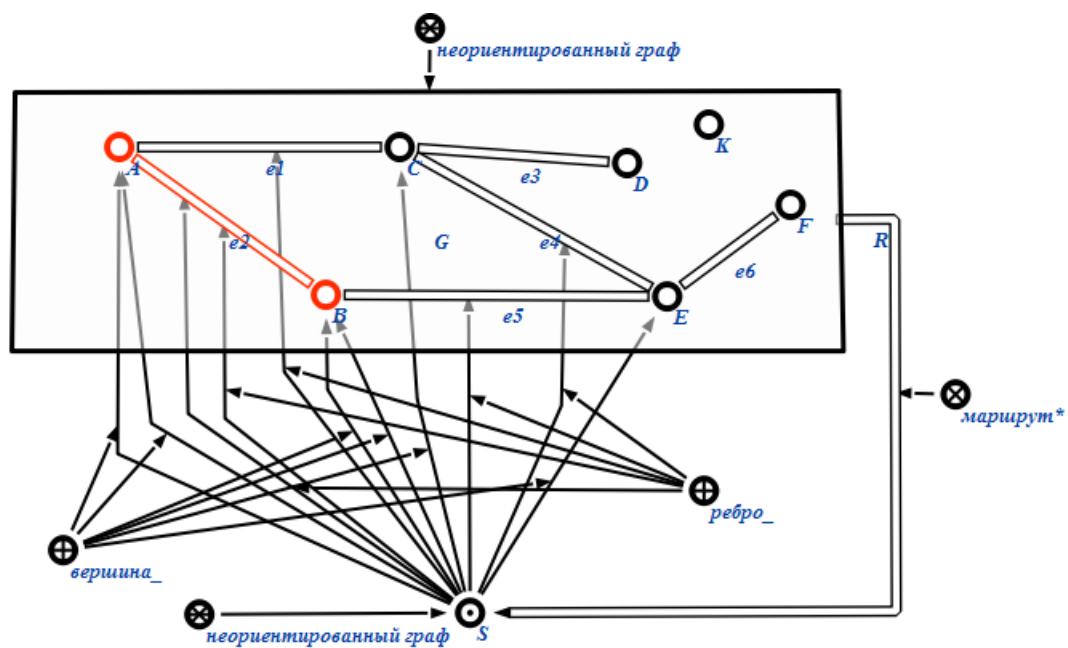


Рис. 2.9: Проблема представления маршрута как подграфа графа G

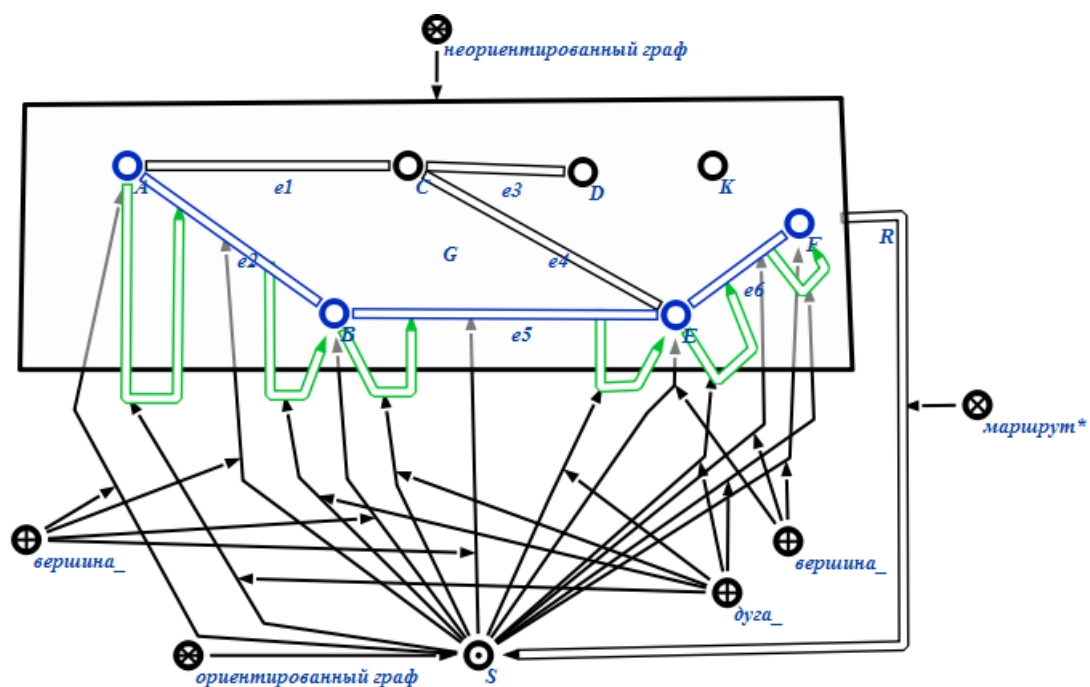


Рис. 2.10: Представление маршрута через ориентированный граф последовательности

из последовательности вершин и ребер, а из последовательности посещений вершин и ребер. Давайте попробуем использовать именно такую точку зрения. Тогда мы можем преобразовать ориентированный граф последовательности S из рис. 2.10 в ориентированный граф посещений S на рис. 2.11. Вершина графа S на рис. 2.11 обозначает посещение вершины графа G , а дуга графа S – посещение ребра графа G .

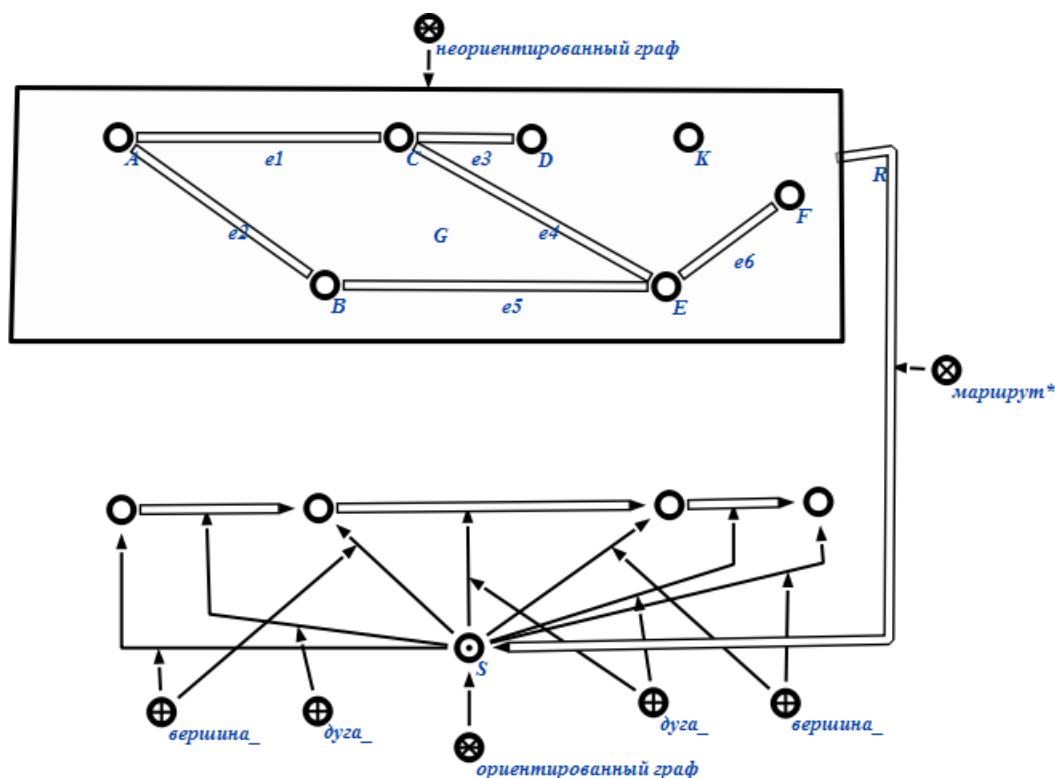


Рис. 2.11: Представление структуры S маршрута R как ориентированного графа посещений

На рис. 2.11 не хватает только связей между элементами посещения из графа S и посещенными элементами из графа G . Для того, чтобы мы могли закончить конструкцию рис. 2.11, вам надо вспомнить, что такое соответствие между двумя множествами, а, если вы этого не знаете, то устранить пробел в вашем образовании. А тем временем мы введем относительное понятие (тернарное отношение) *соответствие**, связка которого включает следующие три компонента (пример связки на рис. 2.12):

1. множество, которое является областью определения соответствия (множество X).
2. множество, которое является областью значений соответствия (множество Y).
3. бинарное ориентированное отношение, которое устанавливает соответствие между элементом из области определения и элементом из области значения (отношение Cr^*).

Мы можем рассматривать маршрут R как всюду определенное, функциональное соответствие между ориентированным графом (структурой маршрута) S и неориентированным исходным графом G . Тогда на основе рис. 2.11 и 2.12 построим окончательный вариант представления маршрута R (рис. 2.13). Рассмотрим отличия рис. 2.13 от рис. 2.11:

1. Для наглядности упрощен ориентированный граф S (скрыты узлы *вершина_* и *дуга_*).
2. Показано, что отношение *маршрут** является подмножеством отношения *соответствие**.

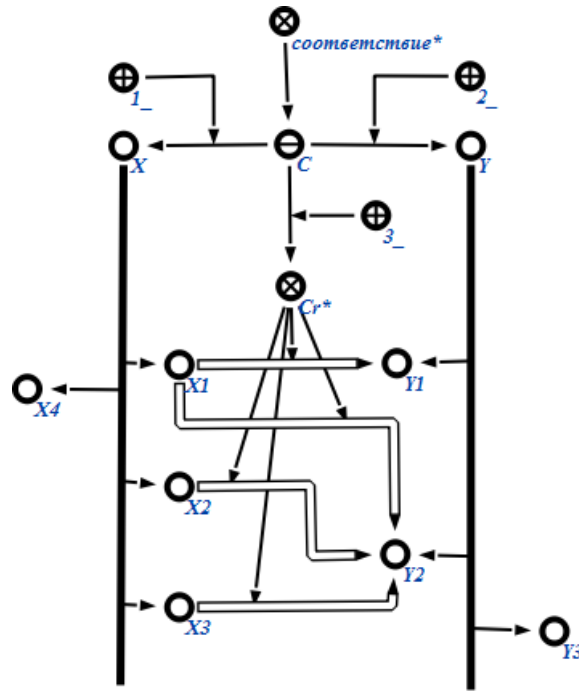


Рис. 2.12: Пример связки C отношения соответствие^* между множествами X и Y

- В связке R первым компонентом теперь является граф S , а вторым – граф G . На рис. 2.11 было наоборот. Чтобы понять, почему связка оказалась перевернутой, взгляните, пожалуйста, на рис. 2.12. На нем видно, что первым компонентом связки отношения соответствие^* должно быть множество, которое является областью определения, а вторым – множество, которое является областью значения. Именно поэтому произошла такая перестановка.
- Появилось отношения (третий компонент связки R), которое задает соответствие между посещением и посещенным элементом. Обратите внимание на направление бинарных ориентированных пар этого отношения!

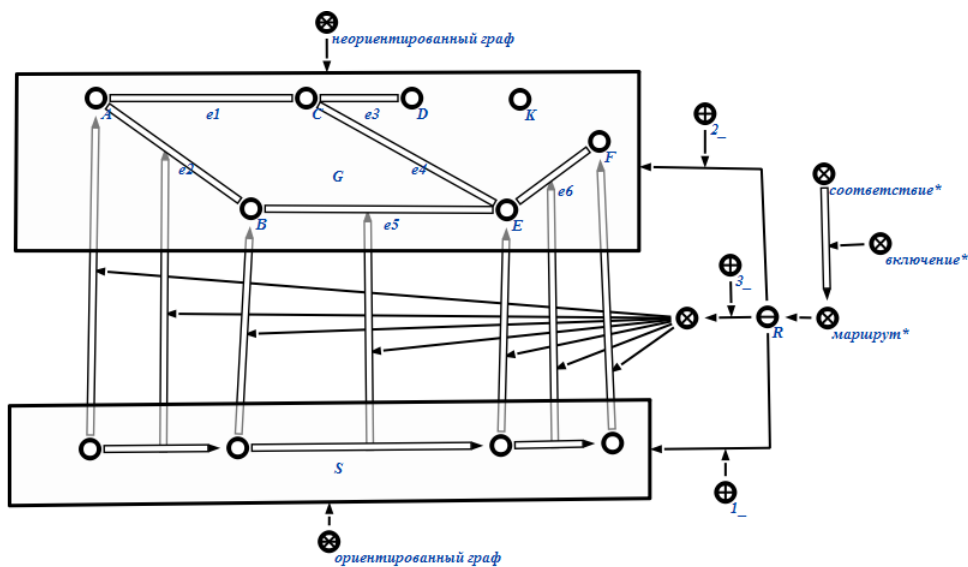


Рис. 2.13: Окончательный вариант представления маршрута R , как соответствия между графом S и графом G

путь, а именно – маршрута. Таким образом, определившись с кодированием понятия *маршрут**, мы определились и с кодированием понятия *путь**. А вот, когда мы разбирались с неориентированными и ориентированными графами, то рассматривали только то, что необходимо для решения нашей задачи, игнорируя более общие случаи. Теперь пришло время посмотреть на задачу представления графов в SC-коде более широко. Для этого мы обратимся к базе знаний по теории графов из проекта [OSTIS GT](#). Неполная иерархия различных типов графов из этой базы знаний изображена на рис. 2.16. Я думаю, что вы уже заметили узлы ориентированный граф и неориентированный граф. Однако они составляют только «низ» иерархии, так как определены на основе более общих понятий. А начнем рассматривать иерархию с «верха», а именно, с понятия *графовая структура*.

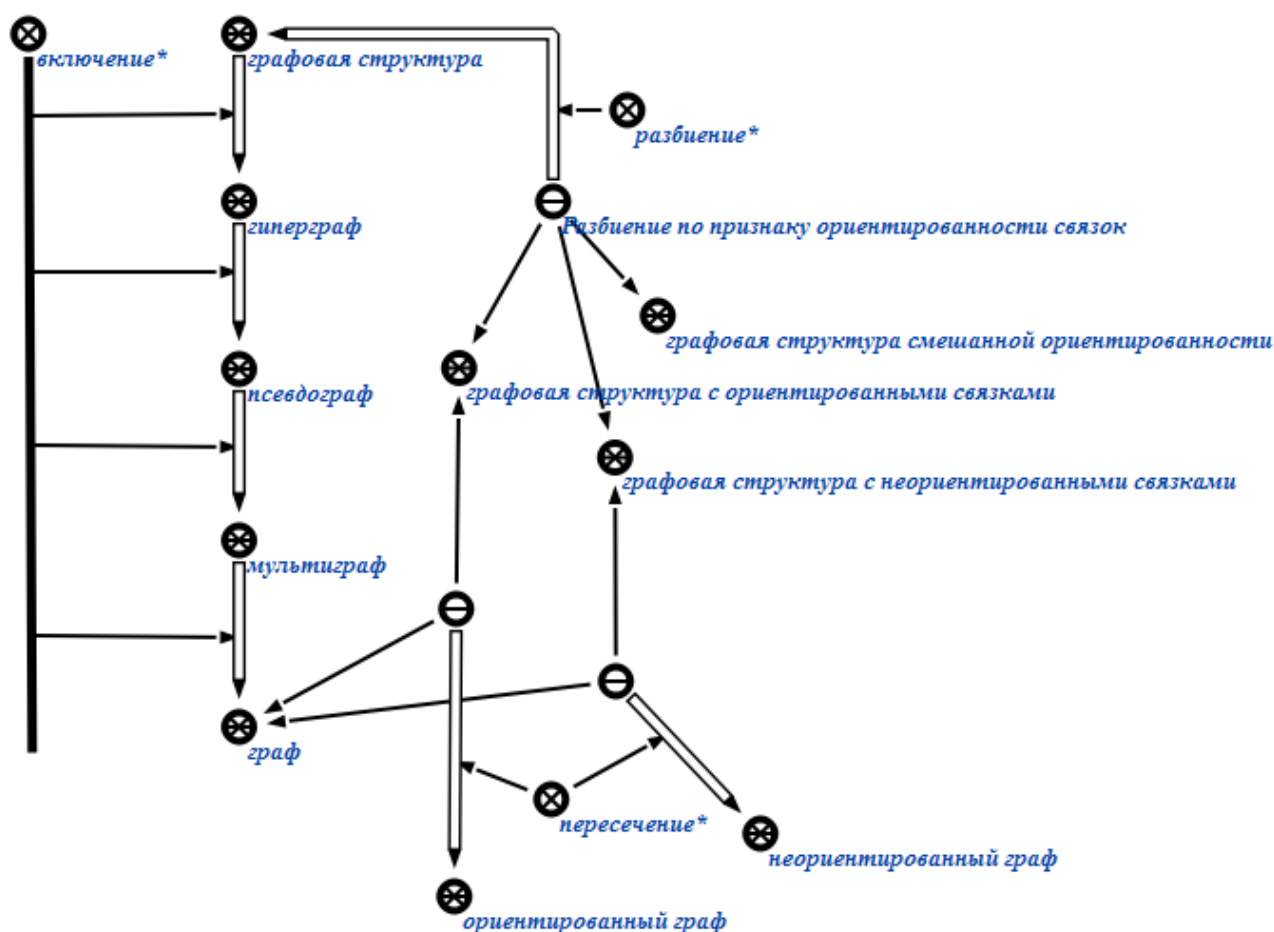


Рис. 2.16: Иерархия различных типов графовых структур (Это неполная иерархия из [OSTIS GT](#))

Экземпляр понятия *графовая структура* – это такая одноуровневая реляционная структура, которая содержит объекты с ролью *вершина_*, а связки между этими объектами - с ролью *связка_*. Вроде бы по определению *графовая структура* несильно отличается от *неориентированного графа*. Взяли просто и заменили ролевое отношение *ребро_* на *связка_*. И вот тут есть один нюанс. На элементы с ролью *связка_* *графовой структуры* не накладывается ограничений, как на элементы с ролью *ребро_* *неориентированного графа*, а именно:

- они могут быть любой арности, а не только бинарными;
- они могут быть как ориентированными, так и неориентированными;
- их компонентами могут быть не только вершины, но и другие связки (т.е. разрешены связки, «выходящие» из связок, и связки, входящие в другие связки).

На рис. 2.17 приведен пример графовой структуры G_s , в которой четыре вершины и три связки. Я хочу отметить, что графовая структура вполне может включать элементы с ролью *ребро_*. Просто *связка_* более общее понятие, чем *ребро_*. Чтобы лучше увидеть связь между этими ролевыми отношениями, рассмотрите рис. 2.18, на котором изображена иерархия ролевых отношений элементов графовой структуры из базы знаний по теории графов. А после того, как закончите, мы перейдем к краткому описанию абсолютных понятий *гиперграф*, *псевдограф*, *мультиграф* и *граф*.

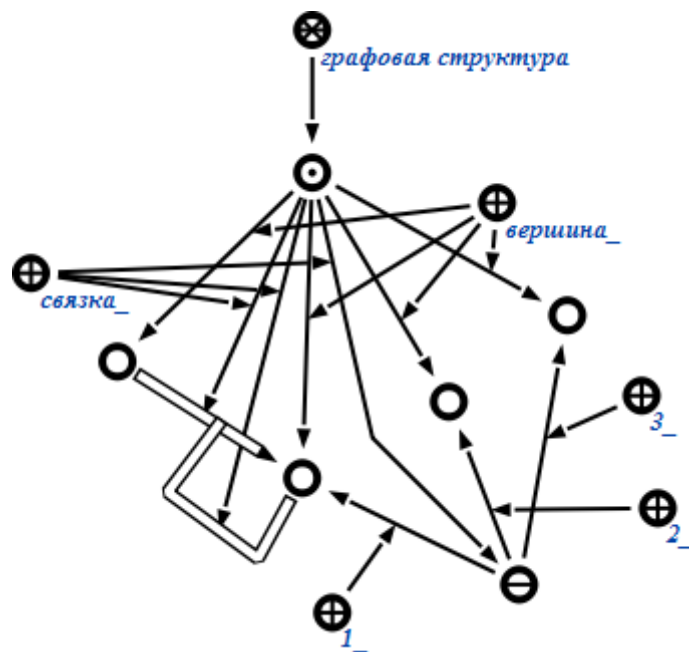


Рис. 2.17: Пример графовой структуры G_s

Экземпляр понятия *гиперграф* (см. рис. 2.16) – это такая графовая структура, в которой элемент с ролью *связка_* может иметь в качестве своих компонентов только элемент с ролью *вершина_* этой графовой структуры. На арность связок никакого ограничения не накладывается. Для указания связки введены ролевые отношения *гиперсвязка_*, *гипердуга_*, *гиперребро_* (рис. 2.18). Обратите внимание на то, что на рис. 2.18 *гипердуга_* определена как пересечение понятий *гиперсвязка_* и ориентированная связка. Аналогично дела обстоят с понятием *гиперребро_*. Пожалуйста, обдумайте самостоятельно такой способ введения новых понятий.

Экземпляр понятия *псевдограф* (см. рис. 2.16) – это такой гиперграф, в котором гиперсвязка может иметь только два компонента. Для указания связки введены ролевые отношения *бинарная связка_*, *петля_*, *дуга_*, *ребро_* (рис. 2.18). Петлей называется бинарная связка, у которой оба компонента одинаковы.

Экземпляр понятия *мультиграф* (см. рис. 2.16) – это такой псевдограф, в котором не может быть петель.

Экземпляр понятия *граф* (см. рис. 2.16) – это такой мультиграф, в котором не может быть кратных связок, т.е. связок у которых первый и второй компоненты совпадают.

А теперь самостоятельно на основе рис. 2.16 рассмотрите то, как определены уже знакомые вам понятия *ориентированный граф* и *неориентированный граф*.

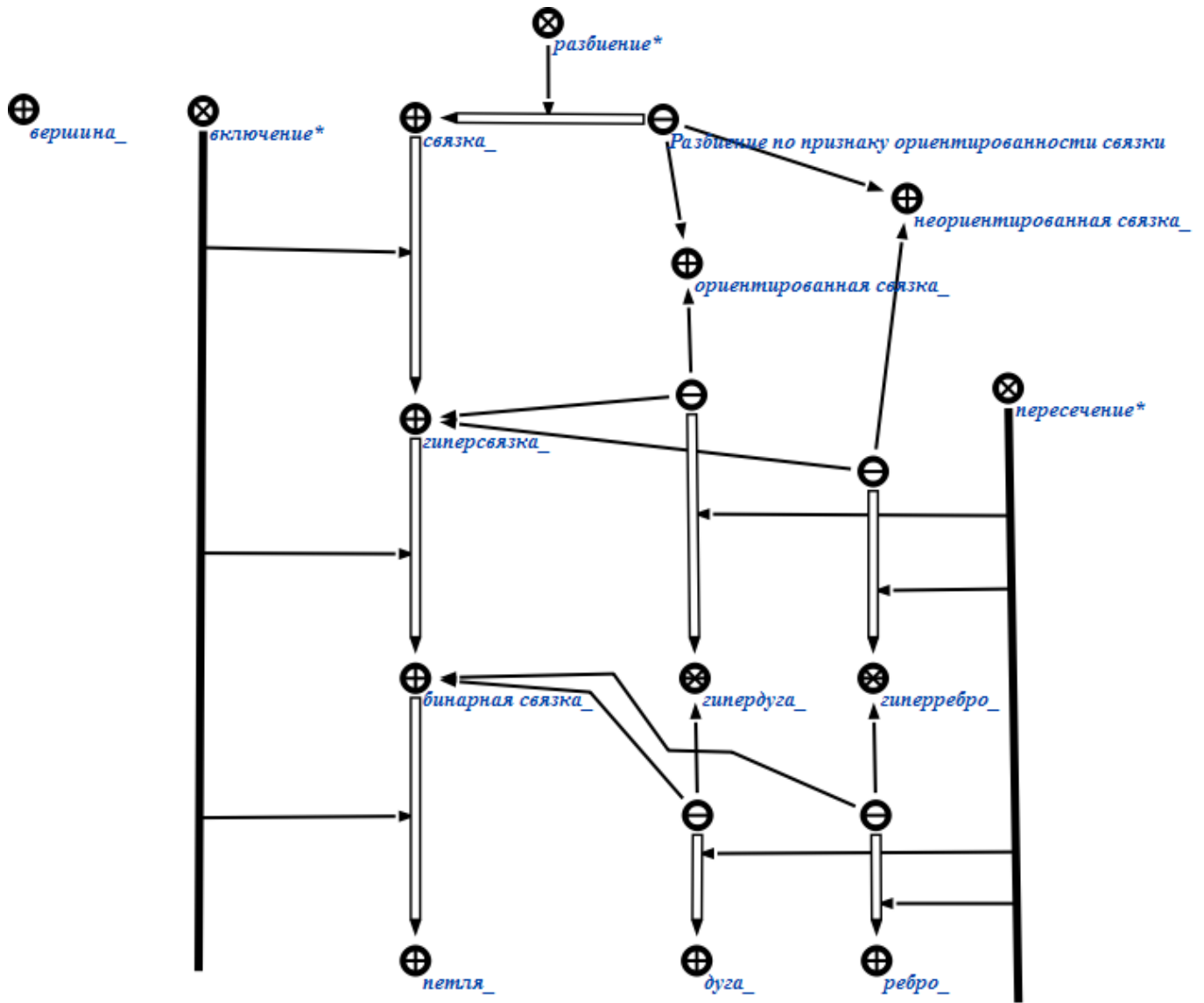
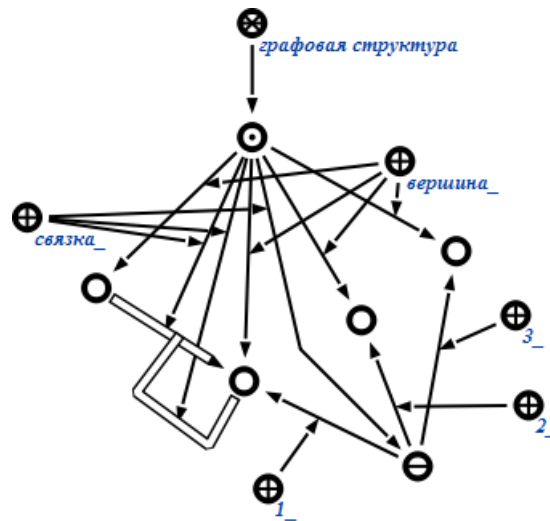


Рис. 2.18: Иерархия ролей элементов графовой структуры (Это неполная иерархия из [OSTIS GT](#))

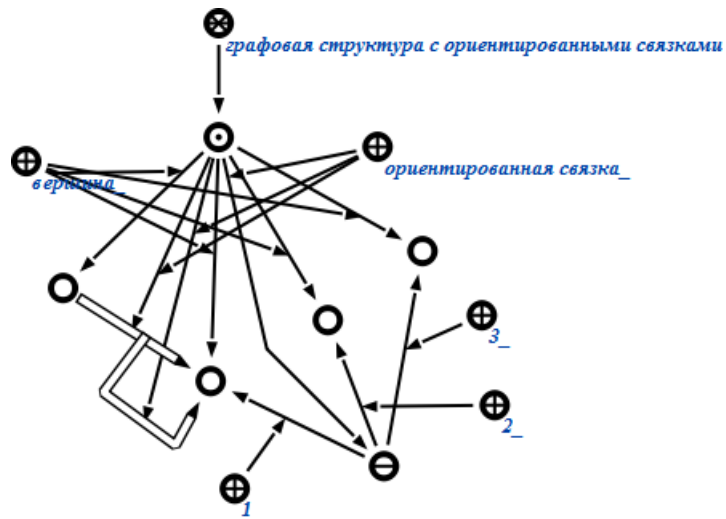
2.5 Пример выполнения

2.5.1 Список понятий

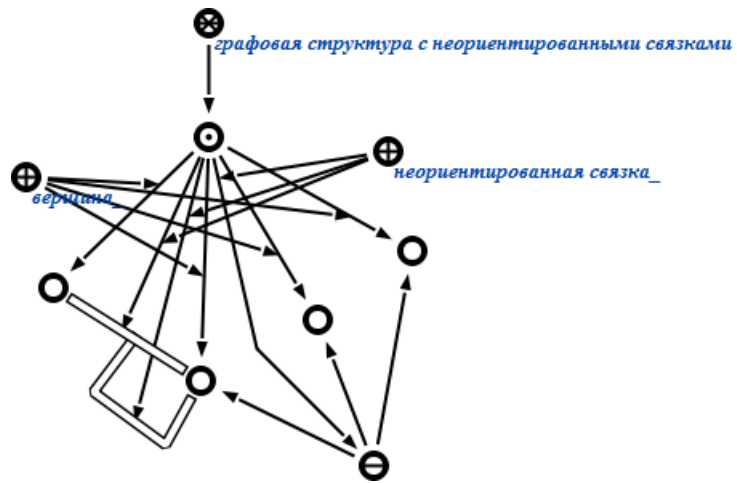
- Графовая структура (абсолютное понятие) - это такая одноуровневая реляционная структура, объекты которой могут играть роль либо вершины, либо связки:
 - Вершина (относительное понятие, ролевое отношение);
 - Связка (относительное понятие, ролевое отношение).



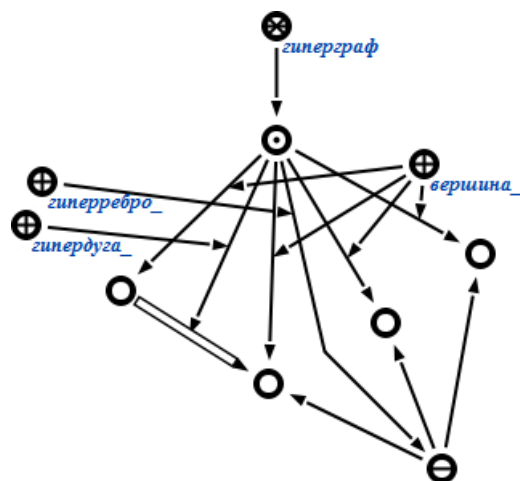
- Графовая структура с ориентированными связками (абсолютное понятие)
 - Ориентированная связка (относительное понятие, ролевое отношение) – связка, которая задается ориентированным множеством.



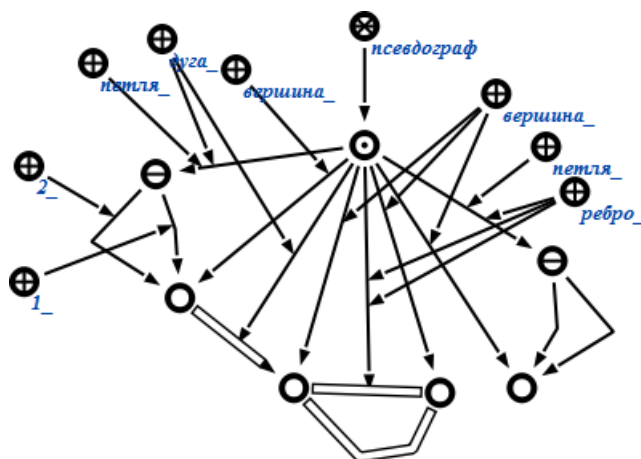
- Графовая структура с неориентированными связками (абсолютное понятие)
 - Неориентированная связка (относительное понятие, ролевое отношение) – связка, которая задается неориентированным множеством.



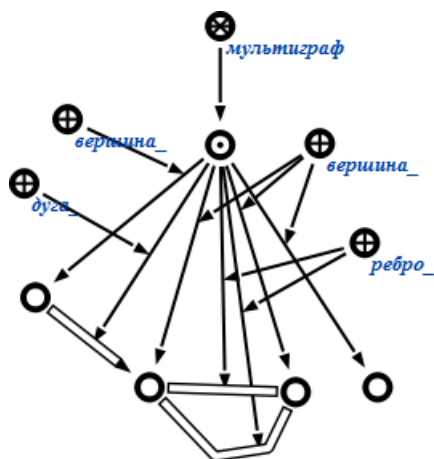
- Гиперграф (абсолютное понятие) – это такая графовая структура, в которой связки могут связывать только вершины:
 - Гиперсвязка (относительное понятие, ролевое отношение);
 - Гипердуга (относительное понятие, ролевое отношение) – ориентированная гиперсвязка;
 - Гиперребро (относительное понятие, ролевое отношение) – неориентированная гиперсвязка.



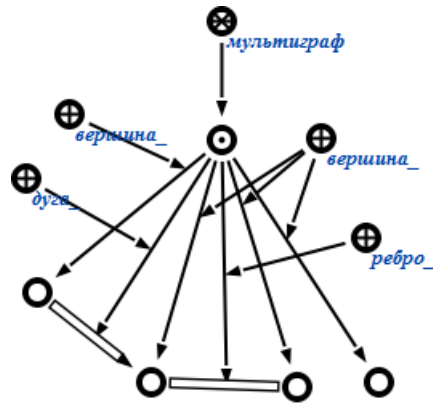
- Псевдограф (абсолютное понятие) – это такой гиперграф, в котором все связки должны быть бинарными.
 - Бинарная связка (относительное понятие, ролевое отношение) – гиперсвязка арности 2;
 - Ребро (относительное понятие, ролевое отношение) – неориентированная гиперсвязка;
 - Дуга (относительное понятие, ролевое отношение) – ориентированная гиперсвязка;
 - Петля (относительное понятие, ролевое отношение) – бинарная связка, у которой первый и второй компоненты совпадают.



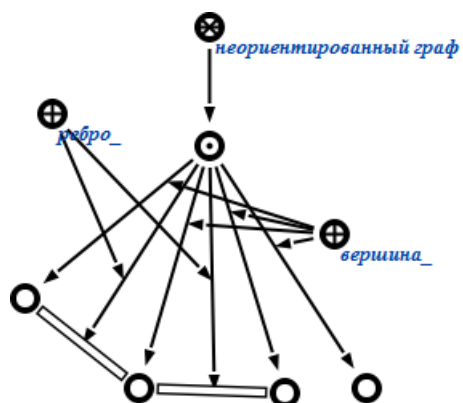
- Мультиграф (абсолютное понятие) – это такой псевдограф, в котором не может быть петель.



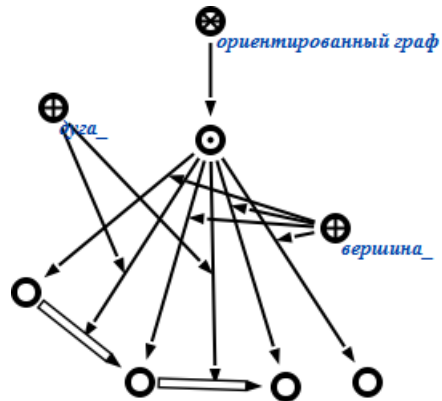
- Граф (абсолютное понятие) – это такой мультиграф, в котором не может быть кратных связок, т.е. связок у которых первый и второй компоненты совпадают.



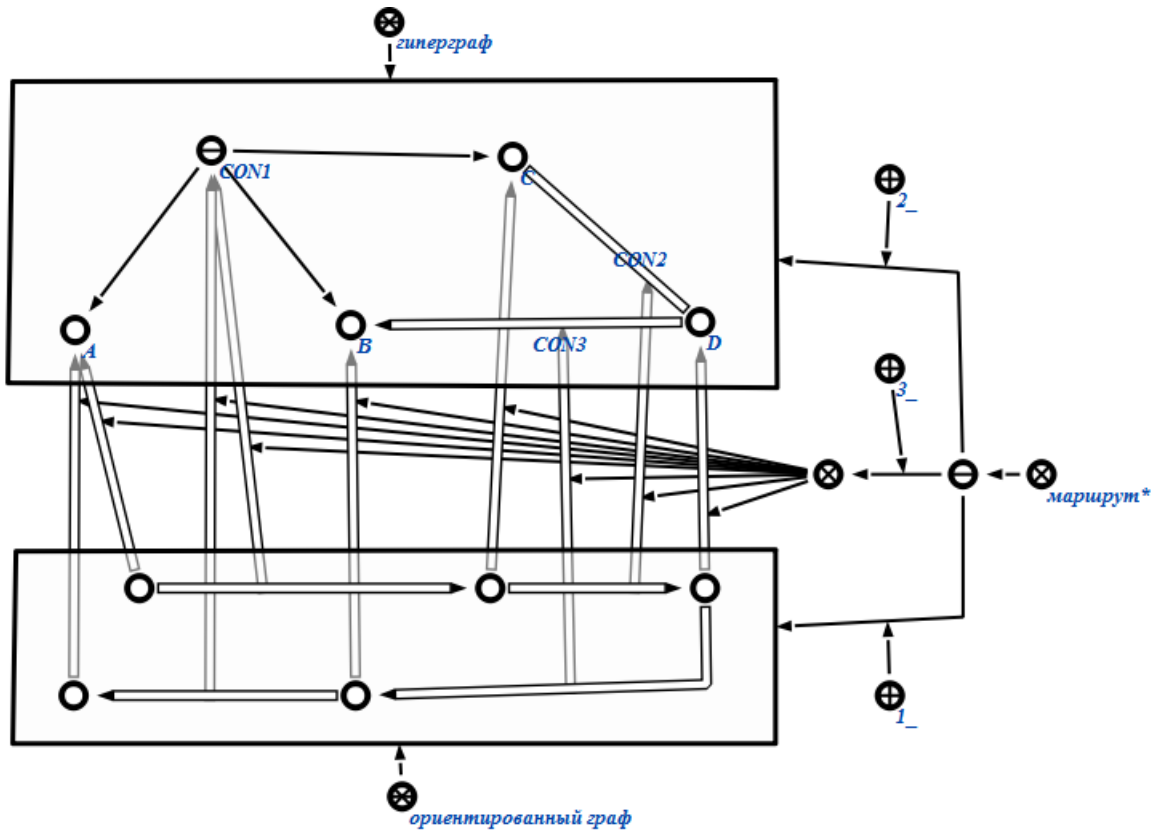
- Неориентированный граф (абсолютное понятие) – это такой граф, в котором все связи являются ребрами:



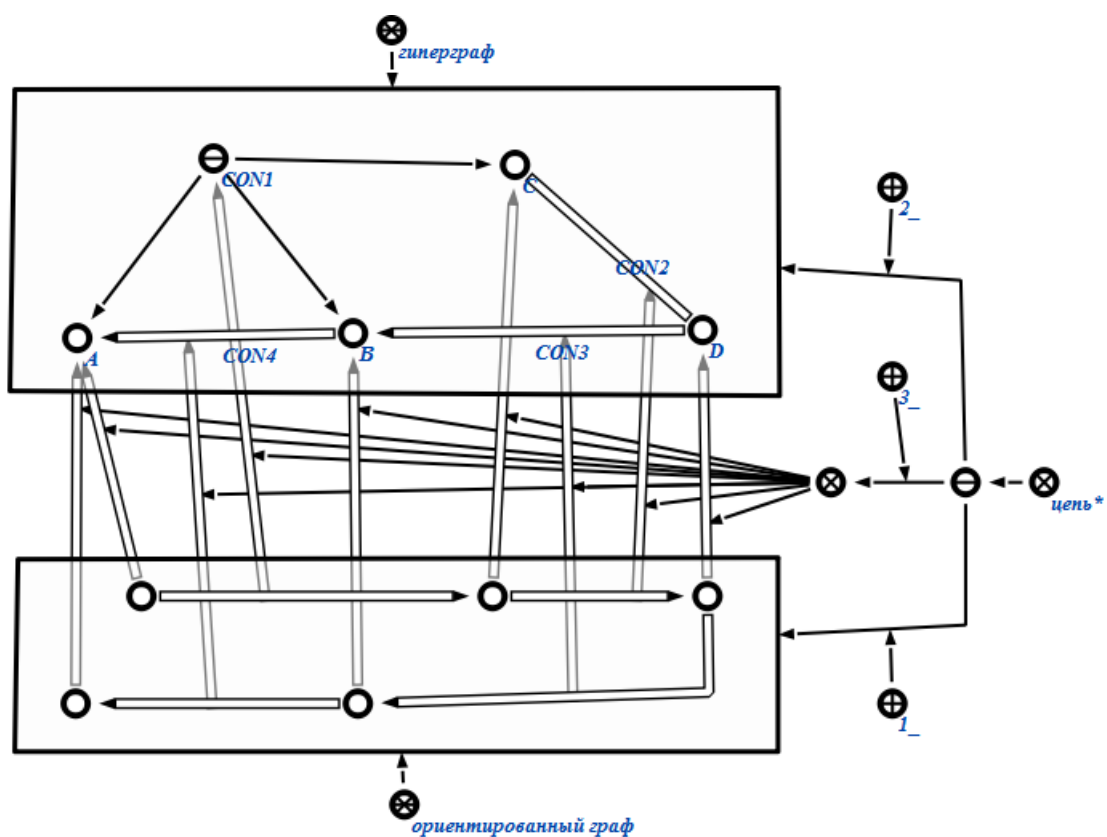
- Ориентированный граф (абсолютное понятие) - это такой граф, в котором все связи являются дугами:



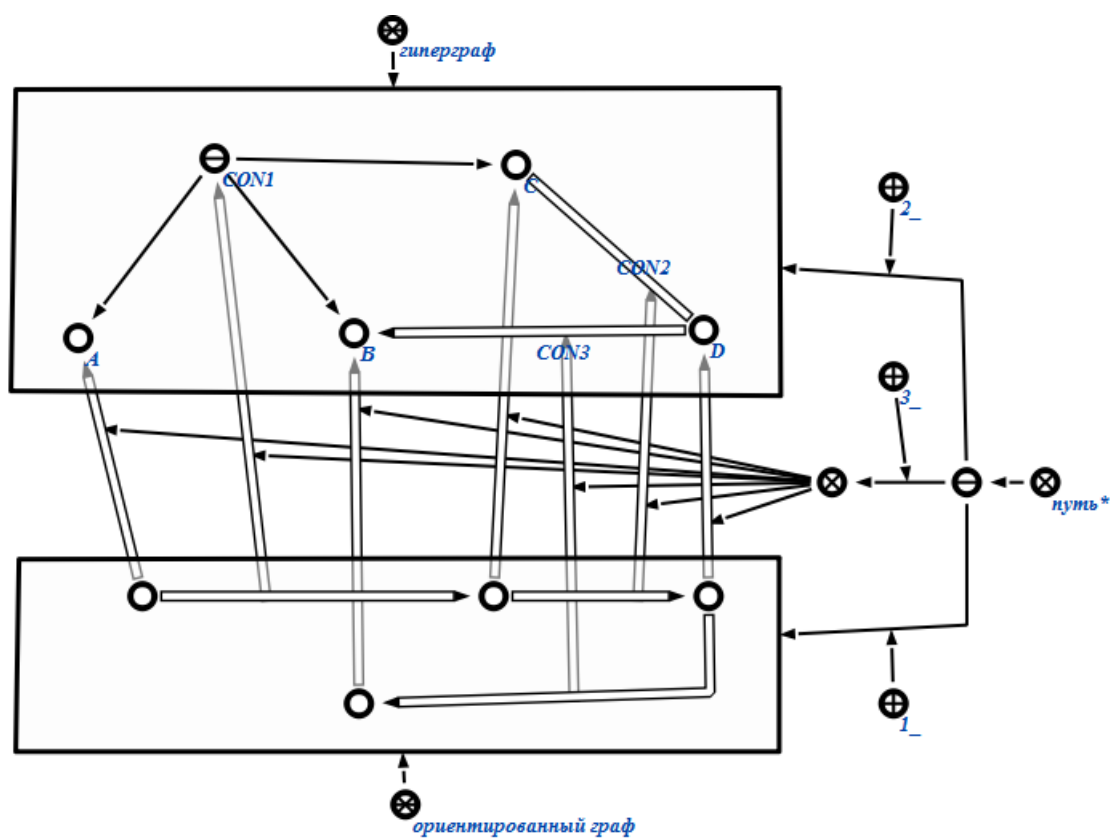
- Маршрут (относительное понятие, бинарное ориентированное отношение) – это чередующаяся последовательность вершин и гиперсвязок в гиперграфе, которая начинается и кончается вершиной, и каждая гиперсвязка последовательности инцидентна двум вершинам, одна из которых непосредственно предшествует ей, а другая непосредственно следует за ней. В примере ниже показан маршрут $A, CON1, C, CON2, D, CON3, B, CON1, A$ в гиперграфе. Обратите внимание, что графы в примере приведены в сокращенной форме, что $CON1$ – это тернарная неориентированная связка (гиперсвязка), а $CON2$ и $CON3$ – бинарные связки (гиперсвязки).



- Цепь (относительное понятие, бинарное ориентированное отношение) – это маршрут, все гиперсвязки которого различны. В примере ниже показана цепь $A, CON1, C, CON2, D, CON3, B, CON1, A$ в гиперграфе.



- Простая цепь, путь (относительное понятие, бинарное ориентированное отношение) – это цепь, в которой все вершины различны. В примере ниже показан путь $A, CON1, C, CON2, D, CON3, B$ в гиперграфе.

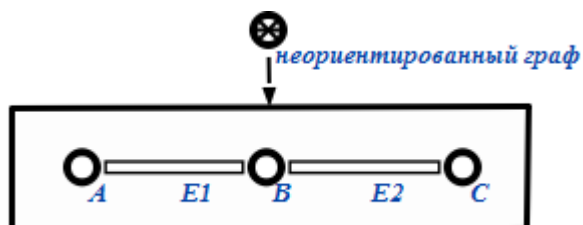


2.5.2 Тестовые примеры

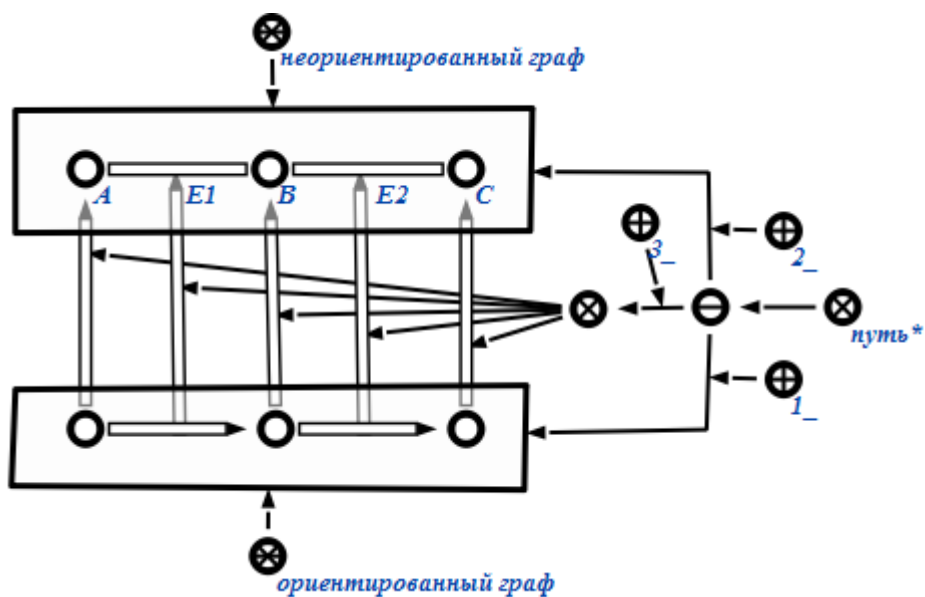
Во всех тестах графы будут приведены в сокращенной форме со скрытыми ролями элементов графа.

Тест 1

Вход: Необходимо найти минимальный путь между вершинами A и C .

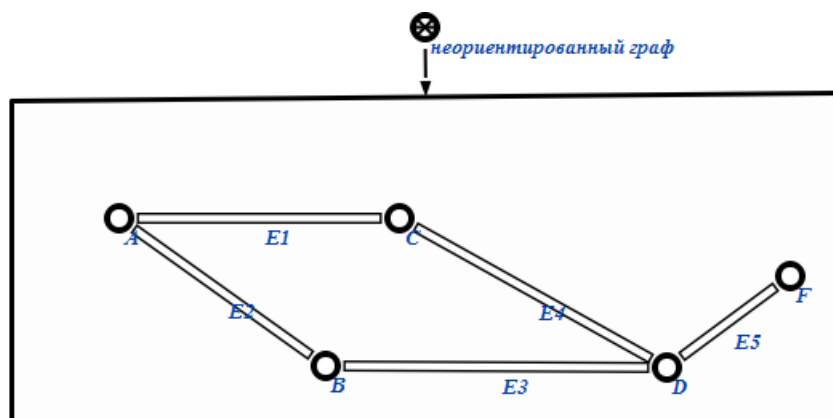


Выход: Будет найден единственный минимальный путь $A, E1, B, E2, C$:

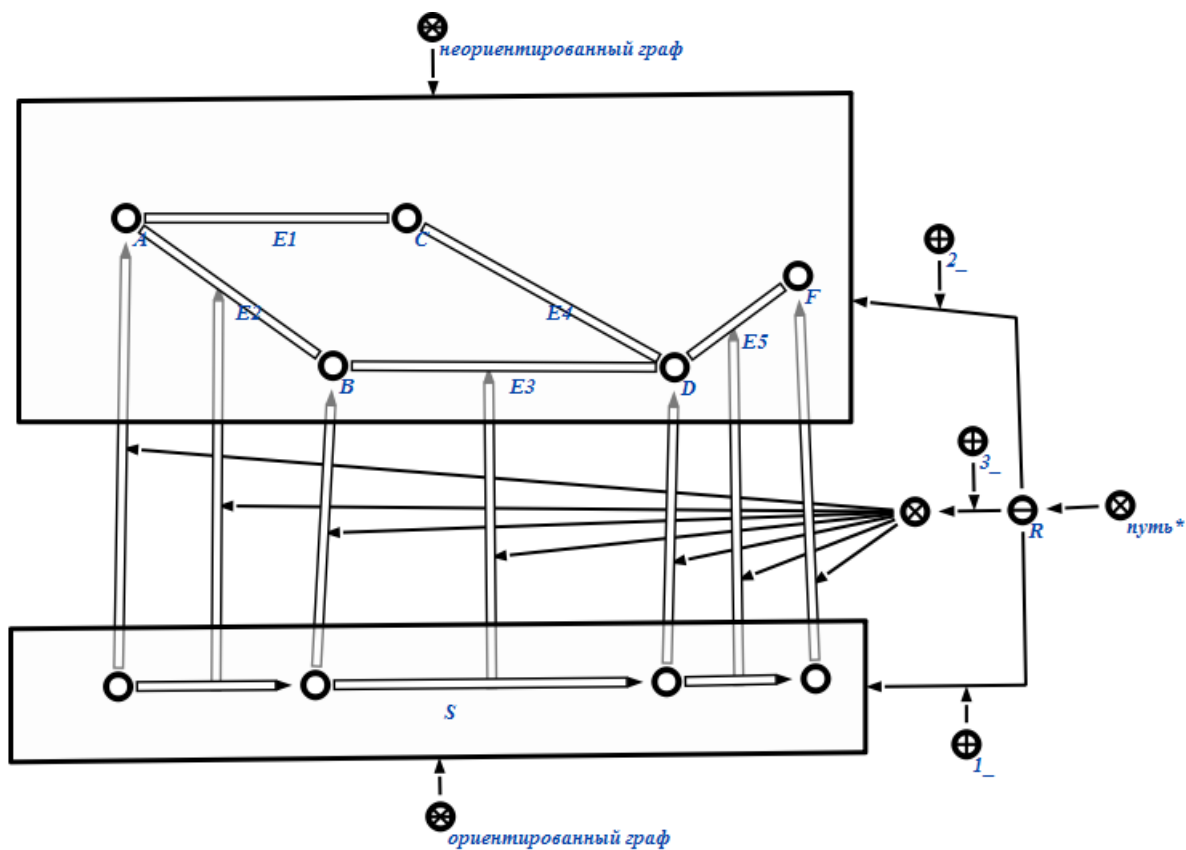


Тест 2

Вход: Необходимо найти минимальный путь между вершинами A и F .

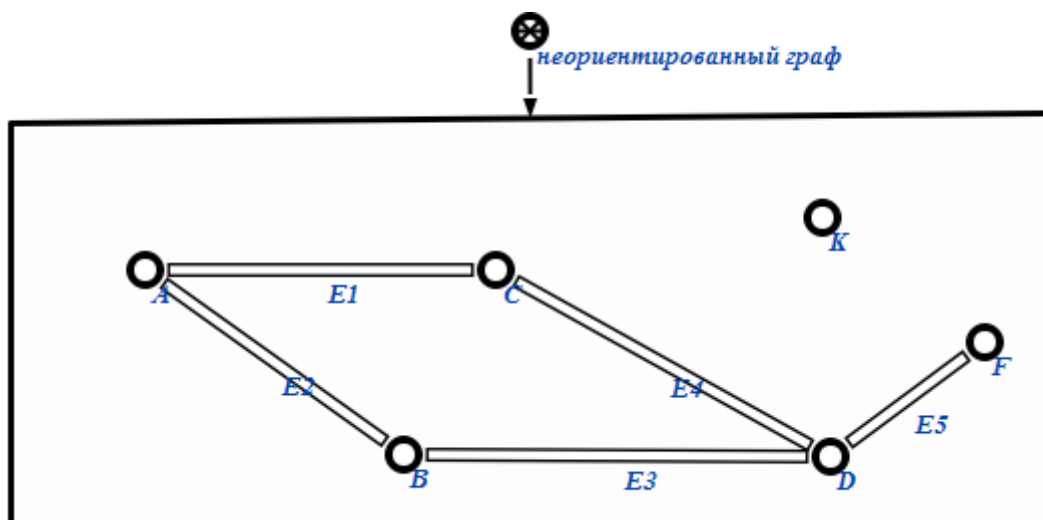


Выход: Будет найден один из двух минимальный путь $A, E2, B, E3, D, E5, F$:



Тест 3

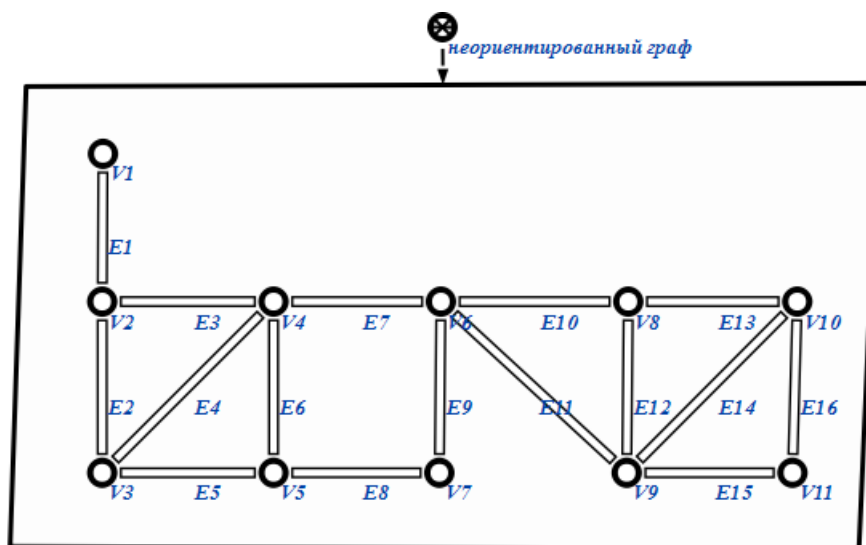
Вход: Необходимо найти минимальный путь между вершинами A и K .



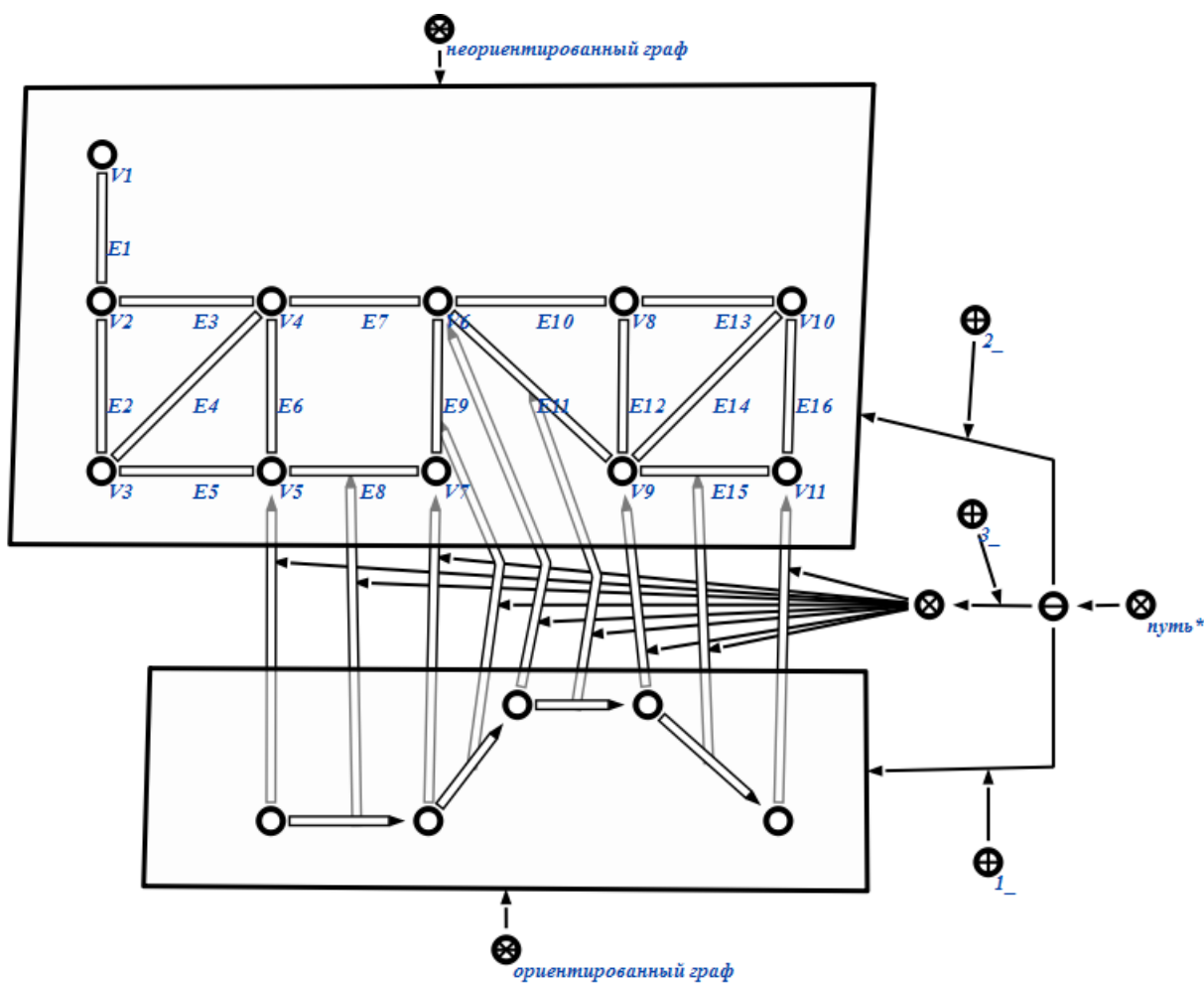
Выход: Минимального пути между вершинами A и K не существует. Программа должна вернуть ошибку вызывающему контексту.

Тест 4

Вход: Необходимо найти минимальный путь между вершинами $V5$ и $V11$.

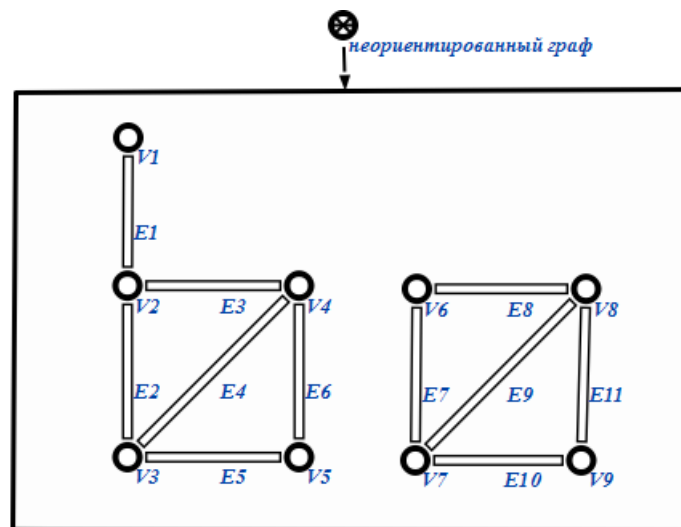


Выход: Будет найден один из двух минимальный путь $V5, E8, V7, E9, V6, E11, V9, E15, V11$:



Тест 5

Вход: Необходимо найти минимальный путь между вершинами $V1$ и $V9$.



Выход: Минимального пути между вершинами $V1$ и $V9$ не существует. Программа должна вернуть ошибку вызывающему контексту.

Глава 3

Демонстрация работы программы решения теоретико-графовой задачи в семантической памяти

3.1 Задание

На этом этапе выполнения расчетной работы вам необходимо будет продемонстрировать пошаговое выполнение алгоритма в sc-памяти. Сам алгоритм и структуры данных для него вы уже исследовали на предыдущих этапах расчетной работы, а сейчас надо будет продемонстрировать графодинамику выполнения алгоритма. Это значит, что вся информация, необходимая для работы вашего алгоритма, должна храниться в sc-памяти и там же и обрабатываться. В качестве примера «неудобства», которое вам может принести такое требование, я могу привести невозможность использования привычной матрицы смежности/инцидентности. Поэтому для прохождения этого этапа вам придется взглянуть на алгоритм, решающий выбранную задачу, под другим углом.

3.2 Волновой алгоритм поиска одного из минимальных путей в неориентированном графе

3.2.1 Описание алгоритма

Алгоритм поиска одного из минимальных путей в неориентированном графе является волновым и основан на понятии волны. В рамках этого алгоритма волной называется множество вершин, каждая из которых является в обрабатываемом графе смежной хотя бы одной вершине из предыдущей волны. Волна, для которой нет предыдущей волны, называется начальной и состоит из вершины, от которой начинается поиск минимального пути. Волна, включающая конечную вершину пути, называется конечной. Таким образом, наш алгоритм можно задать следующим перечнем шагов:

1. Добавить все вершины графа, кроме начальной вершины пути, во множество непроверенных вершин.
2. Создать новую волну и добавить в нее начальную вершину пути.
3. Начальная волна – это новая волна. Новой волной будем называть последнюю созданную волну.

4. Сформировать следующую волну для новой волны. В нее попадет та вершина, которая является смежной вершине из новой волны и присутствует во множестве непроверенных вершин. Если вершина попала в формируемую волну, то ее надо исключить из множества непроверенных вершин. Созданную волну установить как следующую для новой волны, и после этого созданную волну считать новой волной. v
5. Если новая волна пуста, то между вершинами не существует пути. Завершить алгоритм.
6. Если в текущей волне есть конечная вершина, то перейти к пункту 7, иначе к пункту 4.
7. Сформировать один из минимальных путей, проходя в обратном порядке по списку волн. Завершить алгоритм.

3.2.2 Пример выполнения алгоритма в sc-памяти

Соглашения по демонстрации

Перед демонстрацией выполнения алгоритма в sc-памяти нам необходимо установить некоторые соглашения по формированию SCg-рисунков.

При записи графов я буду использовать сокращенную форму - атрибуты для вершин и связей не приводятся, однако вы должны помнить, что сокращенная форма используется только для наглядности.

В качестве программных переменных, которые используются в ходе работы алгоритма, я буду использовать sc-переменные. Для указания значения sc-переменной используется отношение *значение**. На рисунке 3.1 задано значение для sc-переменной `_beg_vertex`.

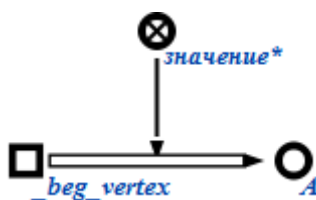


Рис. 3.1: Пример указания значения sc-переменной `_beg_vertex`

Однако я буду сокращать способ, показанный на рис. 3.1, опуская знак отношения *значение**. Таким образом, будет использована форма, показанная на рис. 3.2. Имейте в виду, что полученная sc-конструкция не является корректной в семантическом смысле, но в данном разделе использование именно такой формы позволит сделать рисунки более понятными.

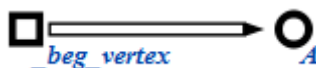


Рис. 3.2: Пример сокращенного указания значения sc-переменной `_beg_vertex`

Последнее, о чем мы договоримся, будет использование цветов с целью явного выделения изменений, произошедших с sc-конструкцией. Для этого я буду использовать следующий перечень цветов:

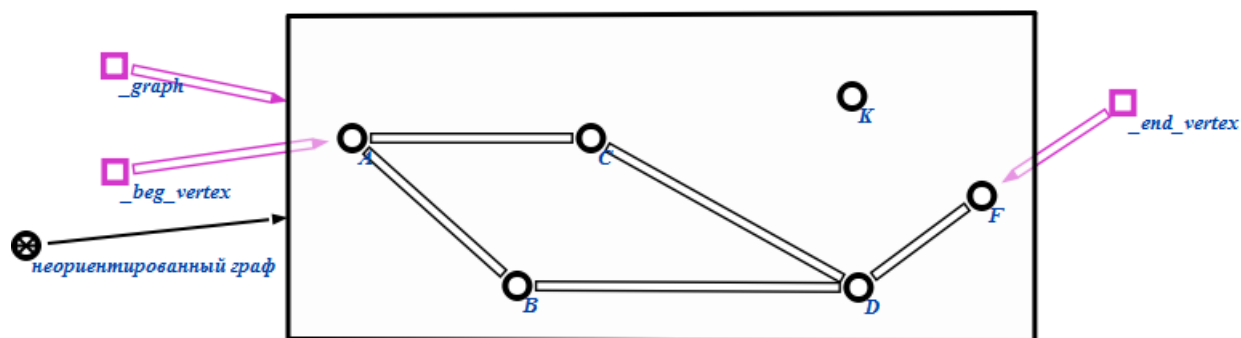
- **фиолетовым** цветом будут выделяться sc-переменные, значение которых изменилось;
- **зеленым** цветом будут выделяться созданные sc-элементы;
- **синим** цветом будут выделяться измененные sc-элементы (например, sc-множества, в которые был включен или из которых был исключен элемент).

Пришло время посмотреть, как же работает наш алгоритм в sc-памяти.

Демонстрация алгоритма

Выполнение алгоритма я продемонстрирую через состояния *sc*-памяти на каждом элементарном этапе решения задачи. Для всех элементарных этапов, для которых это возможно, в конце их краткого описания в скобках будет указан соответствующий номер шага алгоритма из раздела 3.2.1.

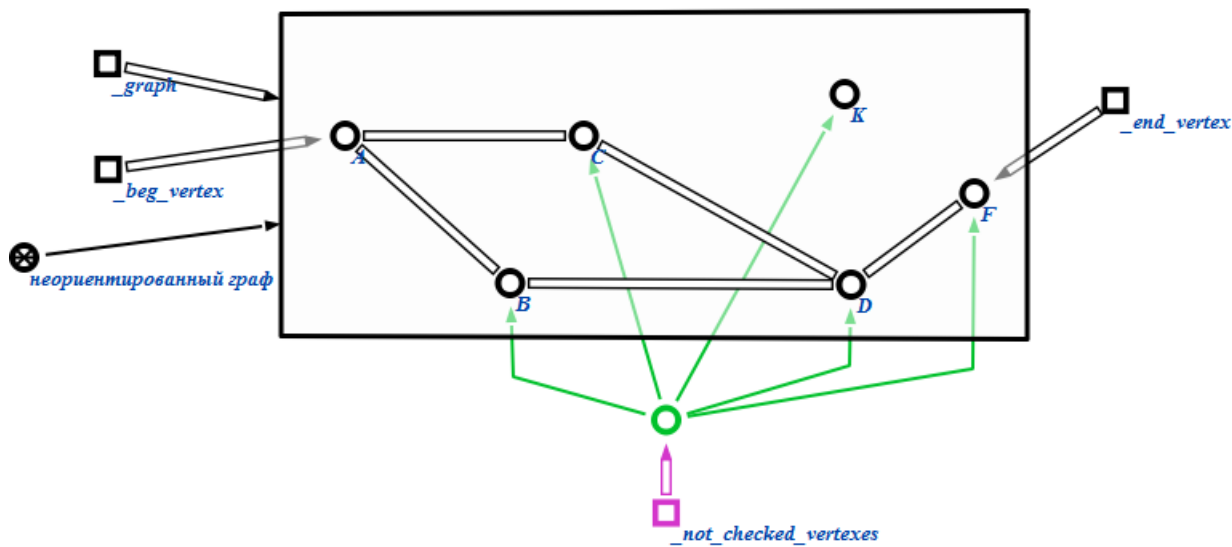
Задание входного графа, начальной и конечной вершины пути для работы алгоритма



Переменные изменятся следующим образом:

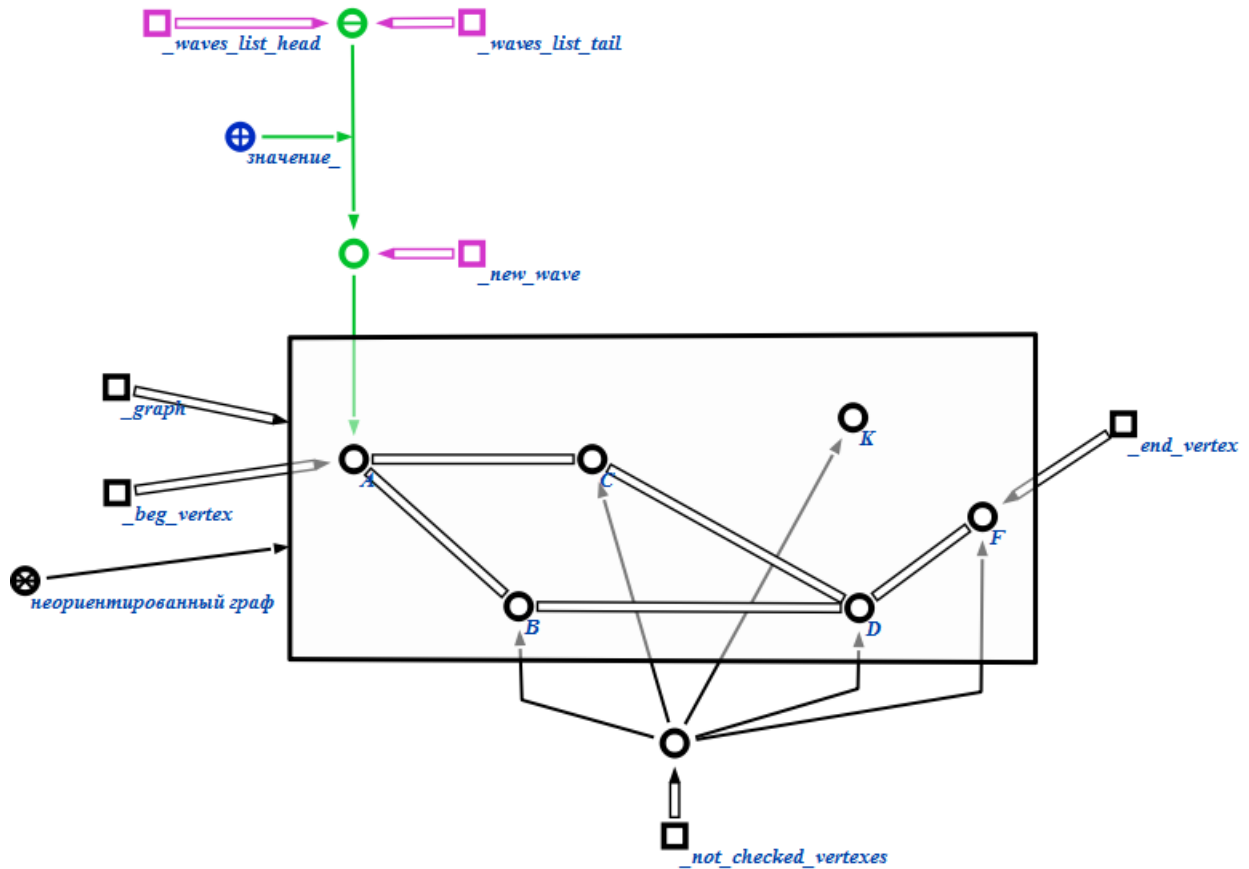
- `_graph` получит в качестве значения *sc*-узел неориентированного графа;
- `_beg_vertex` получит в качестве значения вершину A, которая будет начальной для поиска минимального пути;
- `_end_vertex` получит в качестве значения вершину F, которая будет конечной для поиска минимального пути. Таким образом, из состояния *sc*-памяти на этом шаге вам должно быть ясно, что будет производиться поиск минимального пути между вершинами A и F.

Создание множества непроверенных вершин (Шаг 1)



Переменная `_not_checked_vertices` получит в качестве значения множество непроверенных вершин обрабатываемого графа (в это множество не включена начальная вершина пути A).

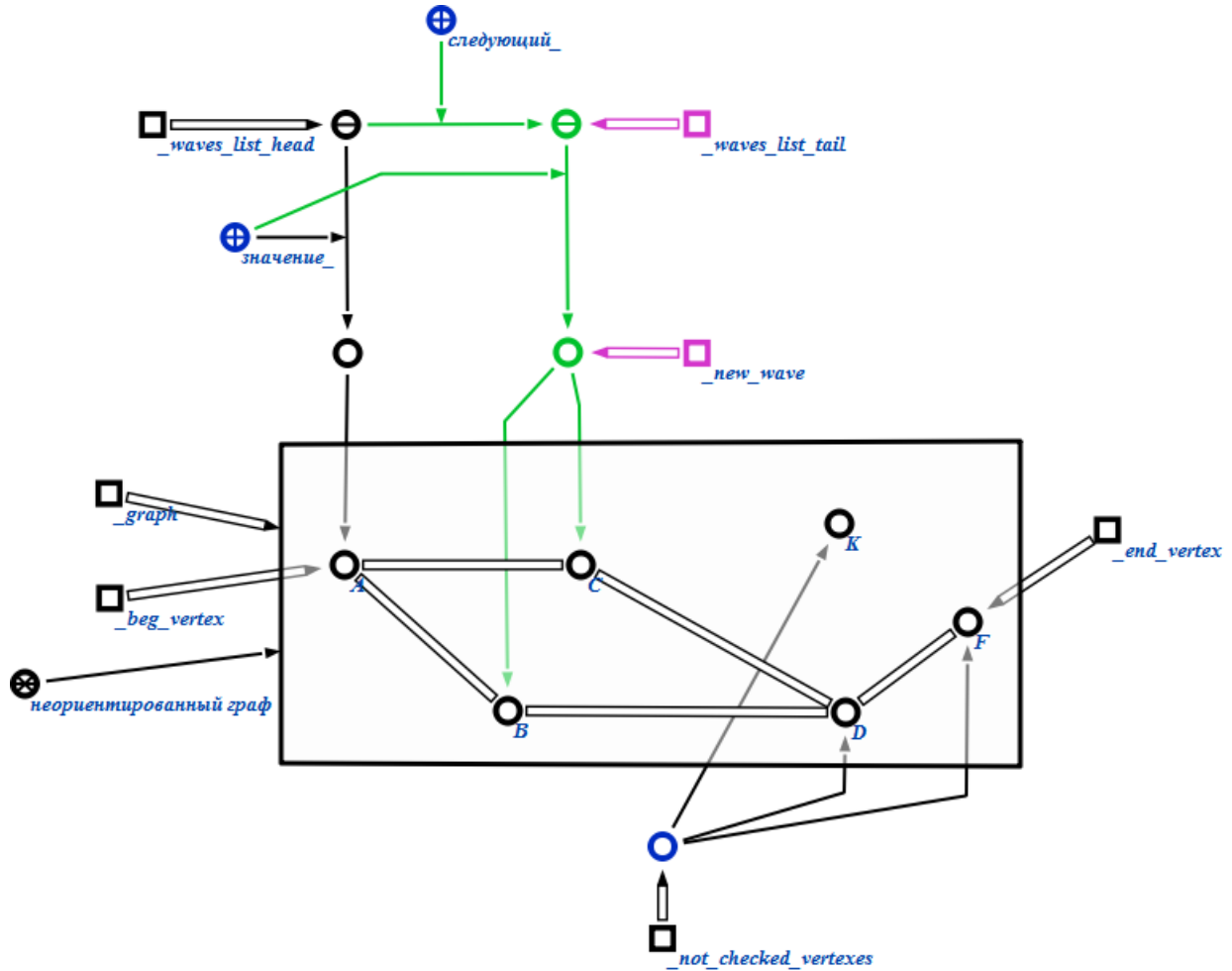
Создание волны, включающей начальную вершину *A* (Шаги 2, 3)



На этом этапе программа создает первую волну из списка волн. Первая волна содержит только начальную вершину пути *A*. Переменная *_new_wave* получает в качестве значения созданную волну, и в будущем будет всегда указывать на вновь созданную волну.

Переменная *_waves_list_head* указывает на начальный элемент списка волн, а переменная *_waves_list_tail* сейчас и в последующих шагах – на концевой элемент списка волн.

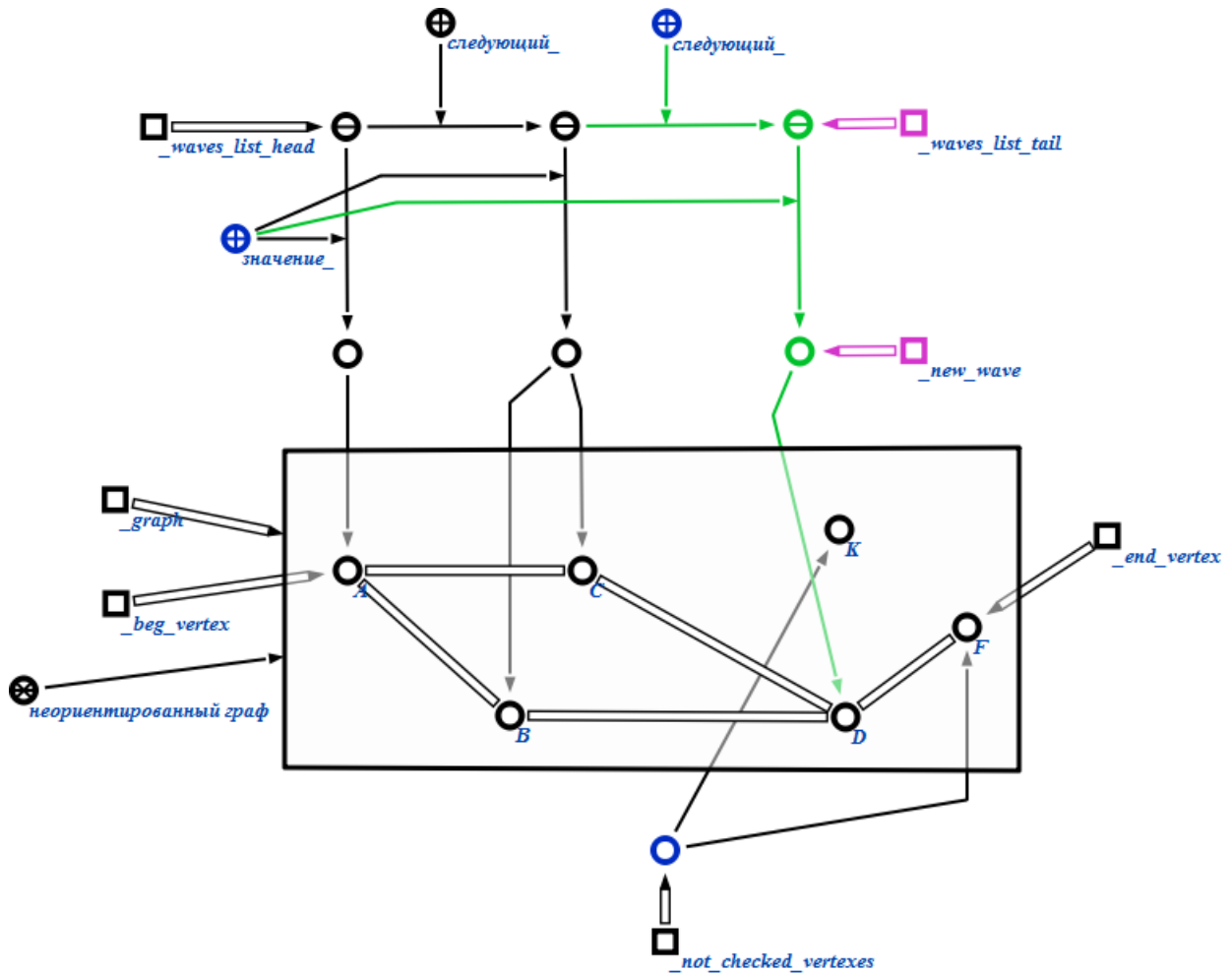
Создание волны, включающей вершины B и C (Шаг 4)



Для вершин из предыдущей волны являются смежными и входящими во множество проверенных вершин только вершины B и C . Из них формируем новую волну. Обратите внимание на то, что эти вершины исключаются из множества непроверенных вершин (см. значение переменной `_not_checked_vertexes`).

Переменная `_waves_list_tail` получает в качестве значения созданный элемент списка, а переменная `_new_wave` – созданную волну.

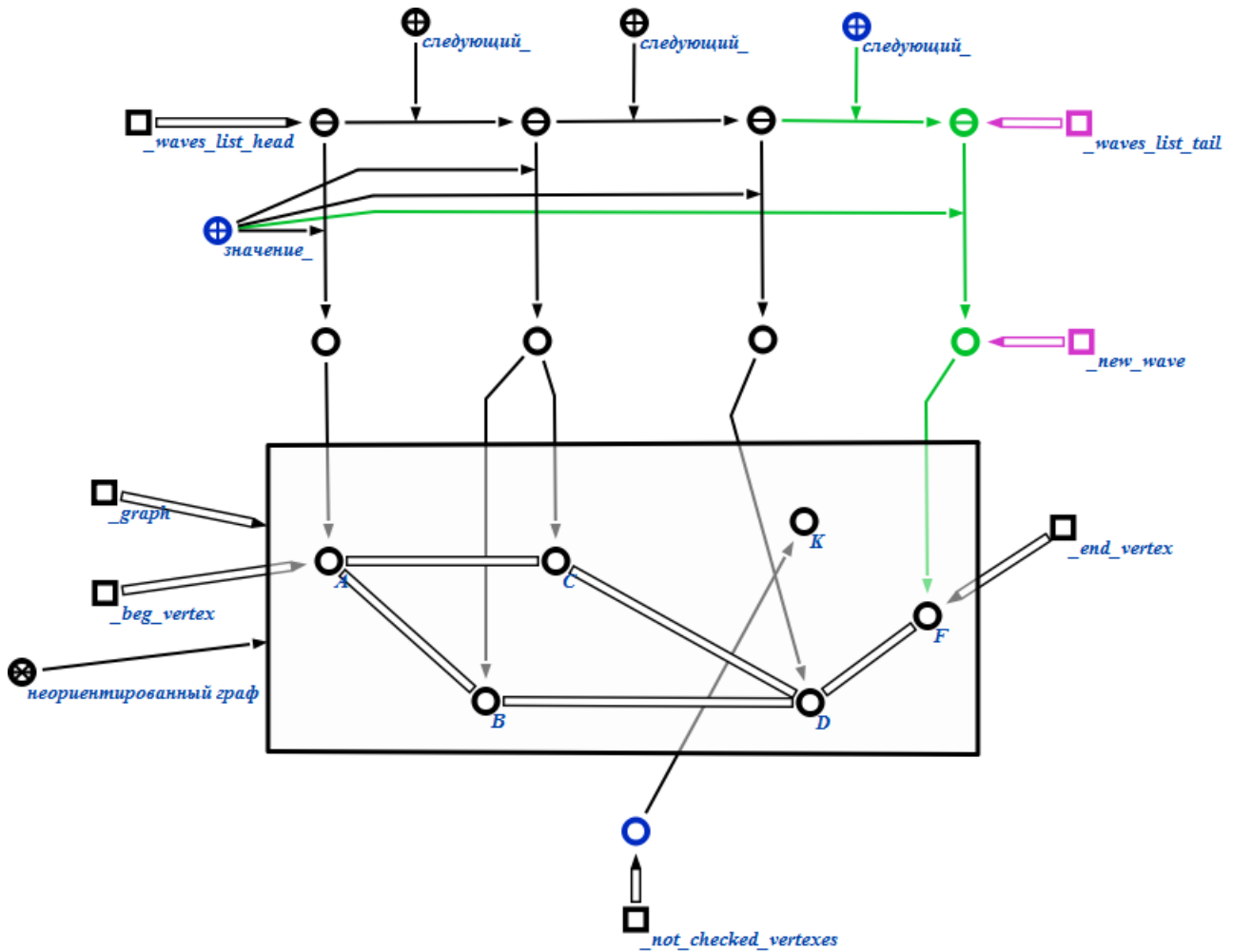
Создание волны, включающей вершину D (Шаг 4)



Для вершин из предыдущей волны является смежной и входящей во множество проверенных вершин только вершина D. Из нее формируем новую волну. Обратите внимание на то, что эта вершина исключается из множества непроверенных вершин (см. значение переменной `_not_checked_vertexes`).

Переменная `_waves_list_tail` получает в качестве значения созданный элемент списка, а переменная `_new_wave` – созданную волну.

Создание волны, включающей вершину F (Шаги 4, 5, 6)

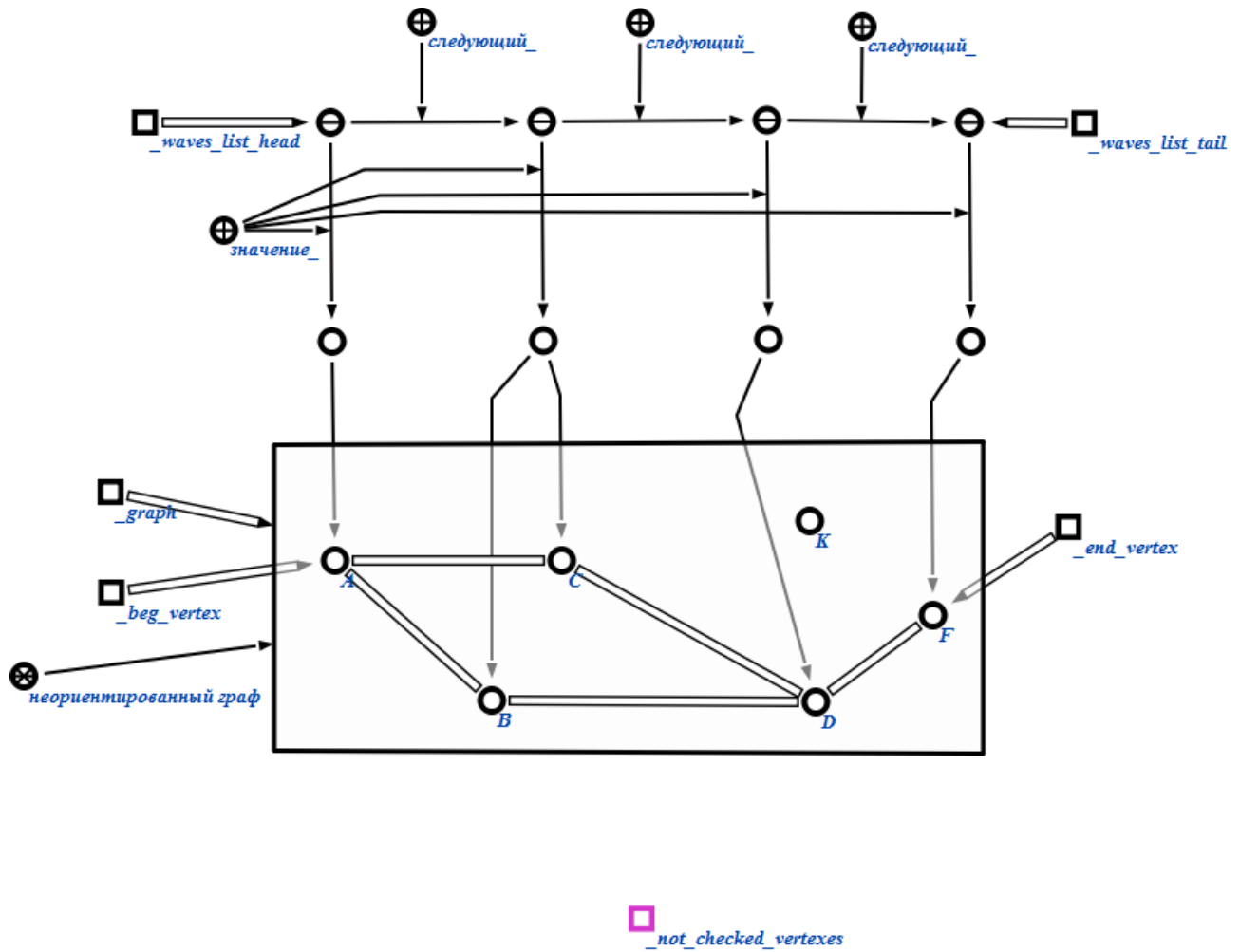


Для вершины из предыдущей волны является смежной и входящей во множество проверенных вершин только вершина F . Из нее формируем новую волну. Обратите внимание на то, что эта вершина исключается из множества непроверенных вершин (см. значение переменной `_not_checked_vertices`).

Переменная `_waves_list_tail` получает в качестве значения созданный элемент списка, а переменная `_new_wave` – созданную волну.

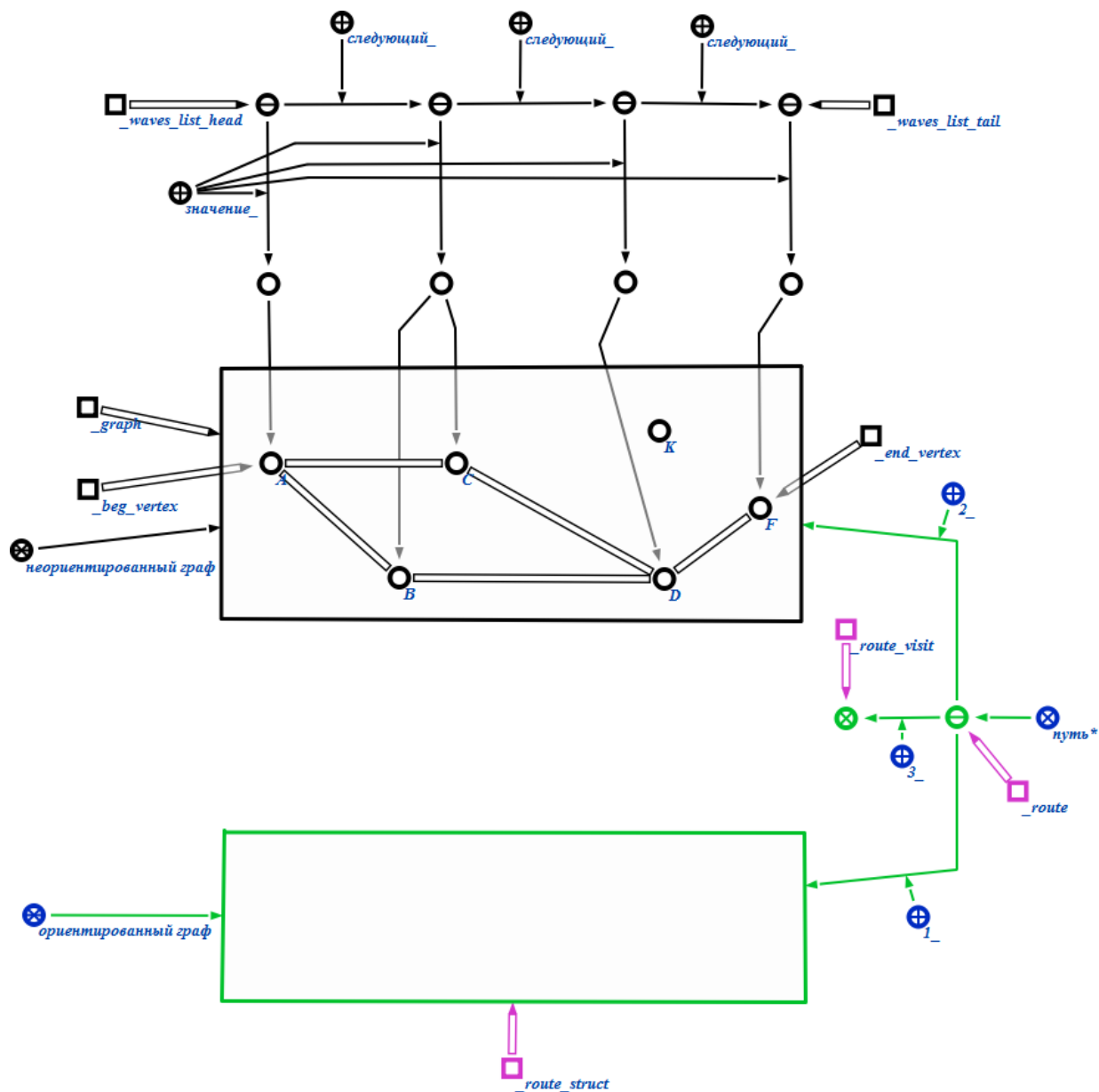
Так как эта волна содержит конечную вершину пути F , то на следующих этапах мы перейдем к генерации одного из найденных минимальных путей.

Удаление множества непроверенных вершин



Подчистим память, удалив значение переменной `_not_checked_vertexes`, так как оно нам уже не надо.

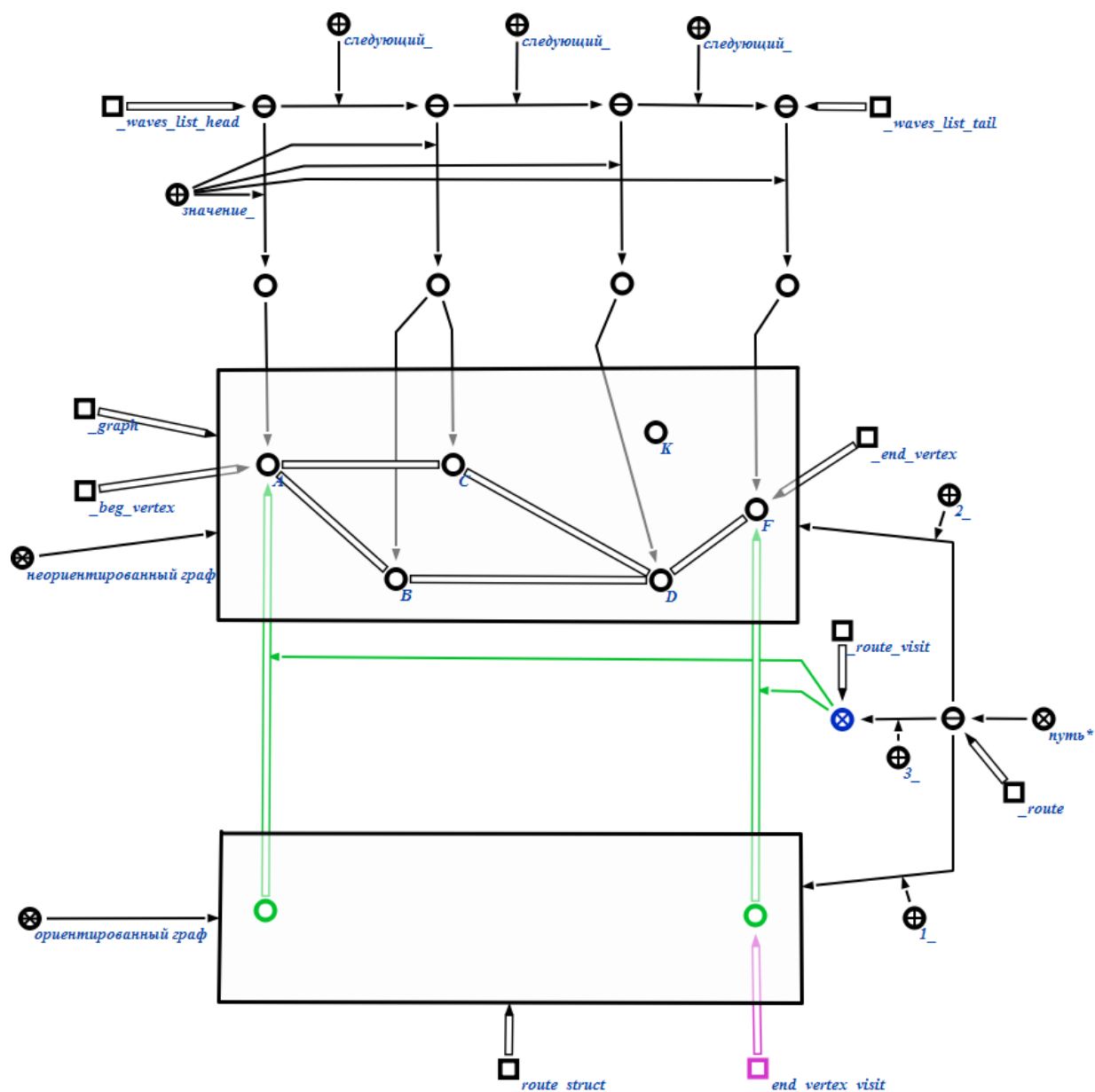
Создание связки отношения путь* (Шаг 8)



Начинаем генерацию одного из найденных минимальных путей.

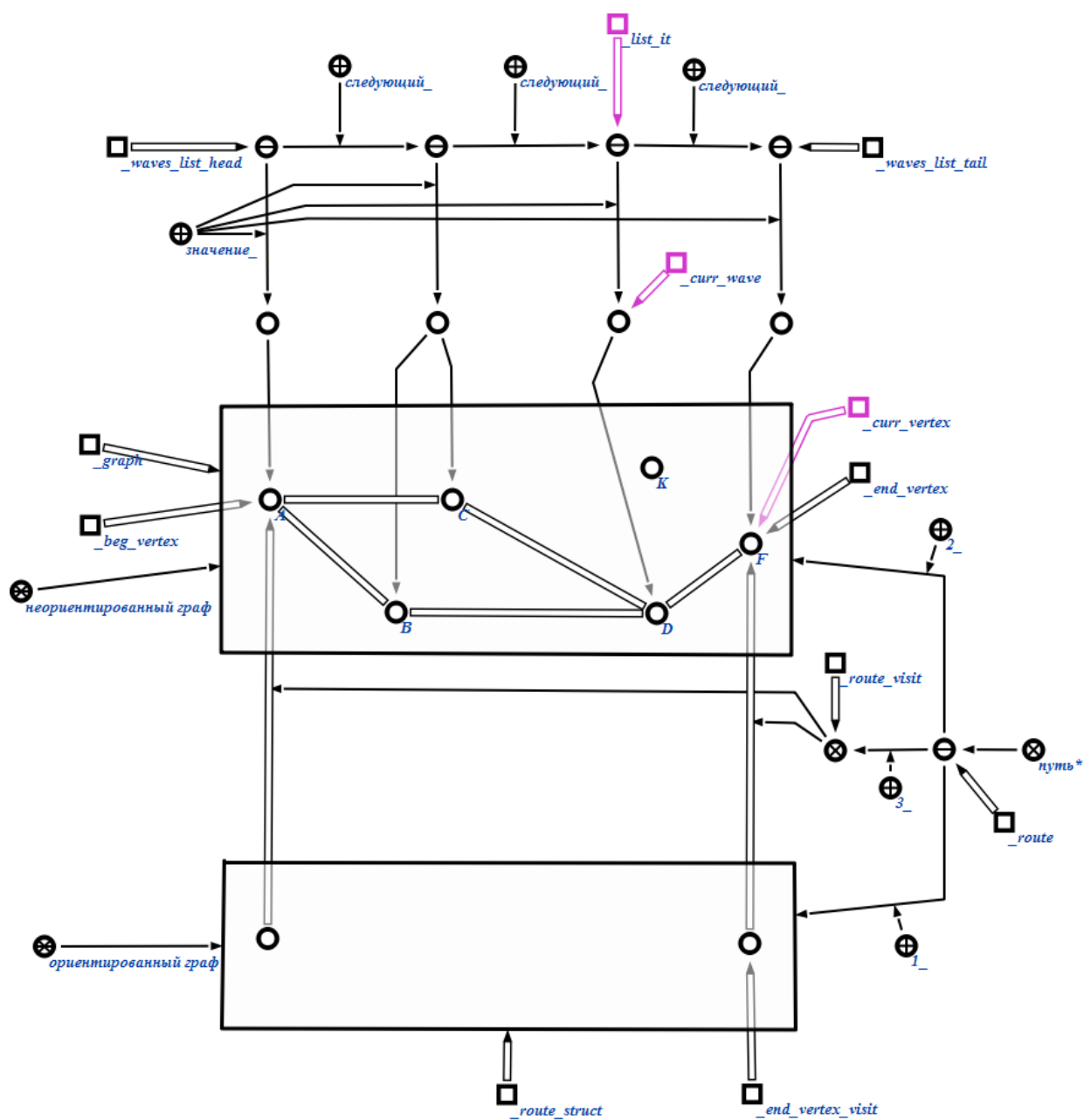
Создадим связку отношения *путь** и установим ее в качестве значения переменной *_route*. Переменная *_route_struct* получит в качестве значения ориентированный граф структуры пути, а переменная *_route_visit* – отношения посещения.

Добавление в структуру пути посещения конечной вершины генерируемого пути
(Шаг 8)



Добавим в структуру генерируемого пути посещение конечной вершины F . Созданное посещение получит в качестве значения переменная `_end_vertex_visit`.

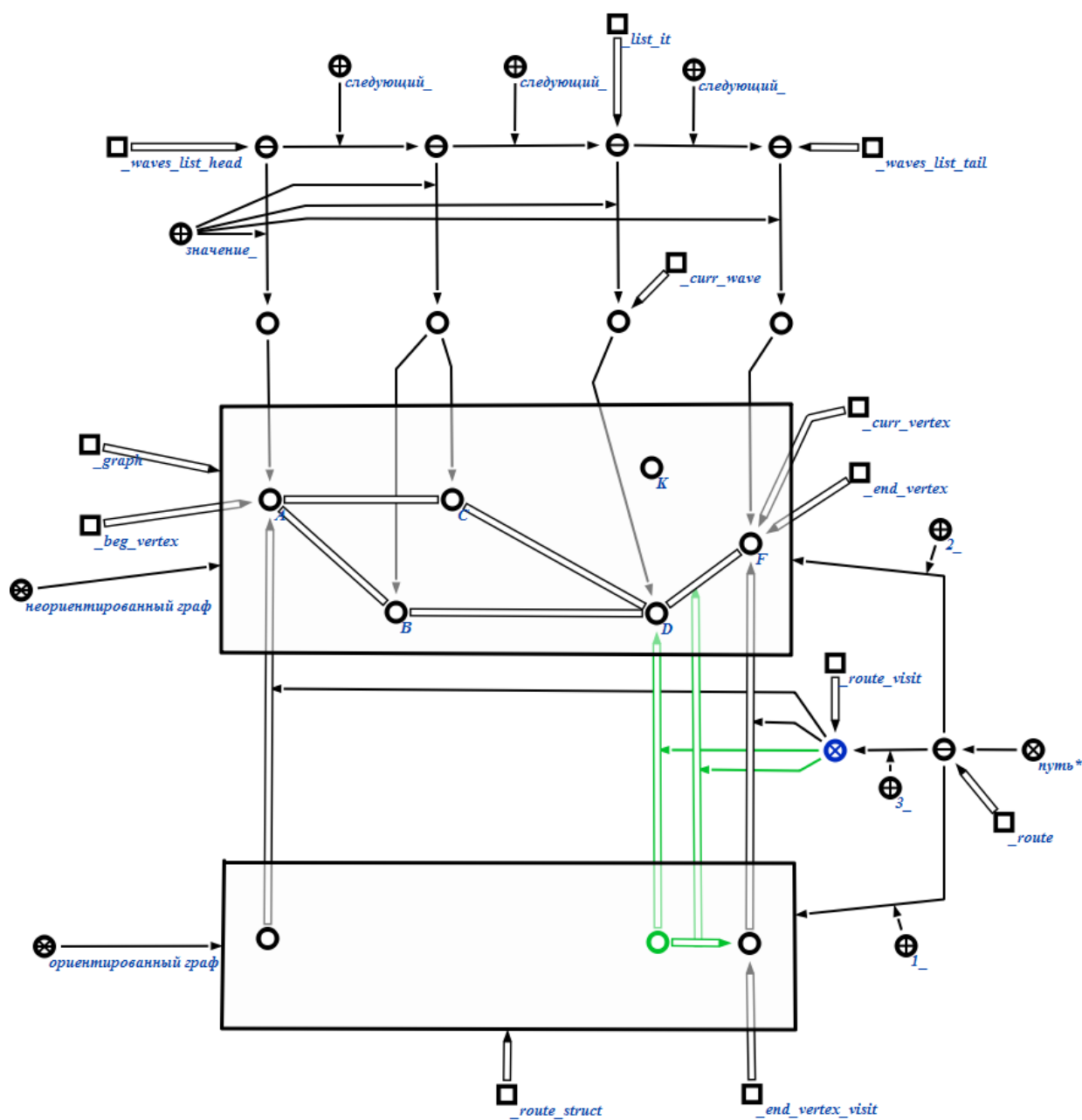
Установка переменных для 1-ой итерации цикла генерации структуры пути (Шаг 8)



Структура пути строится, начиная с конечной вершины пути F .

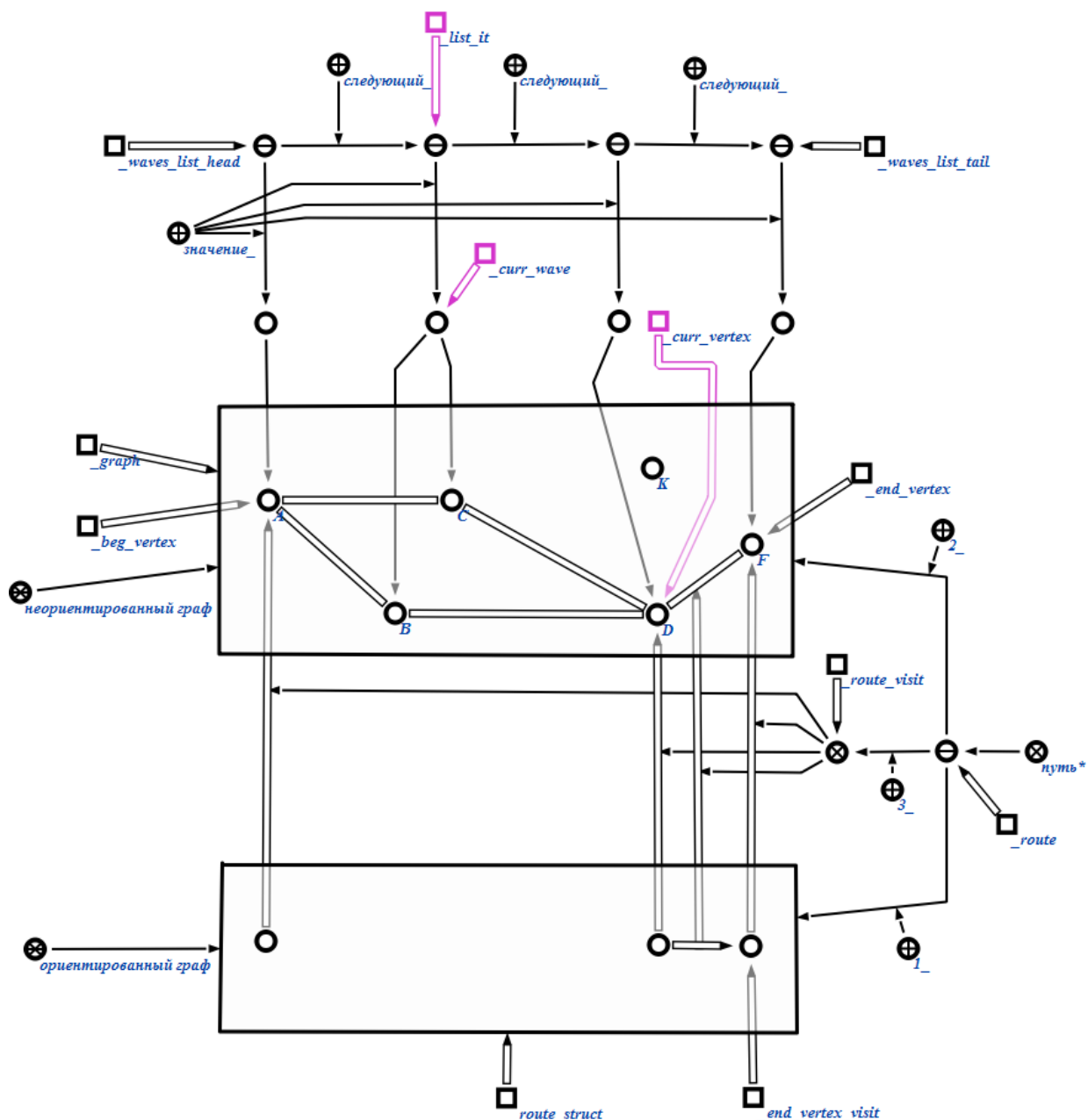
На этом этапе переменная `_curr_vertex` получит в качестве значения текущую обрабатываемую вершину, переменная `_list_it` – текущий обрабатываемый элемент списка волн, переменная `_curr_wave` – текущую обрабатываемую волну.

Создание посещения вершины на 1-ой итерации цикла генерации структуры пути (Шаг 8)



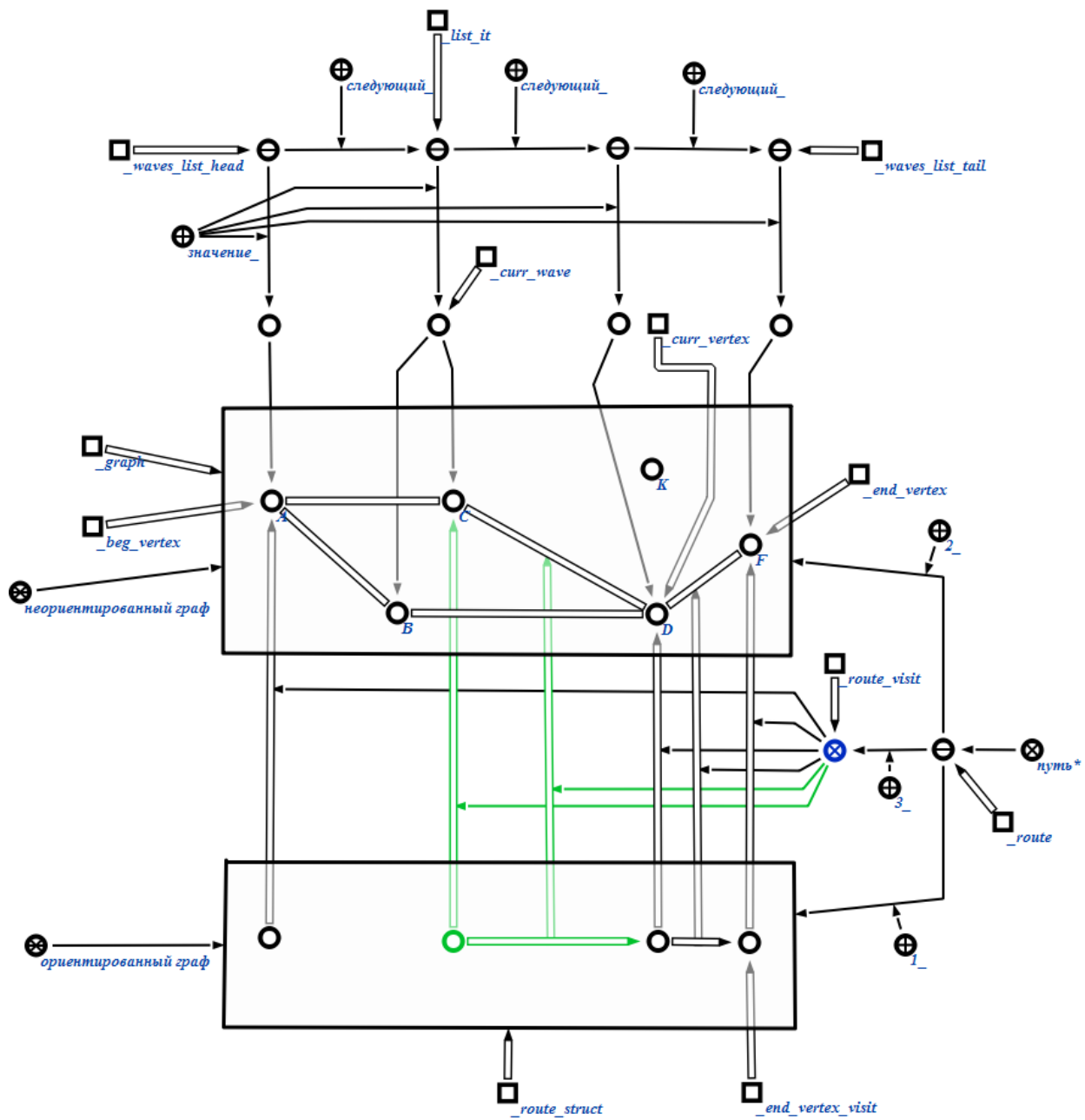
Создадим посещение для одной из вершин из волны `_curr_wave`, которая является смежной вершине из переменной `_curr_vertex`. Для связывающего их ребра тоже создадим посещение.

Переход ко 2-ой итерации цикла генерации структуры пути (Шаг 8)



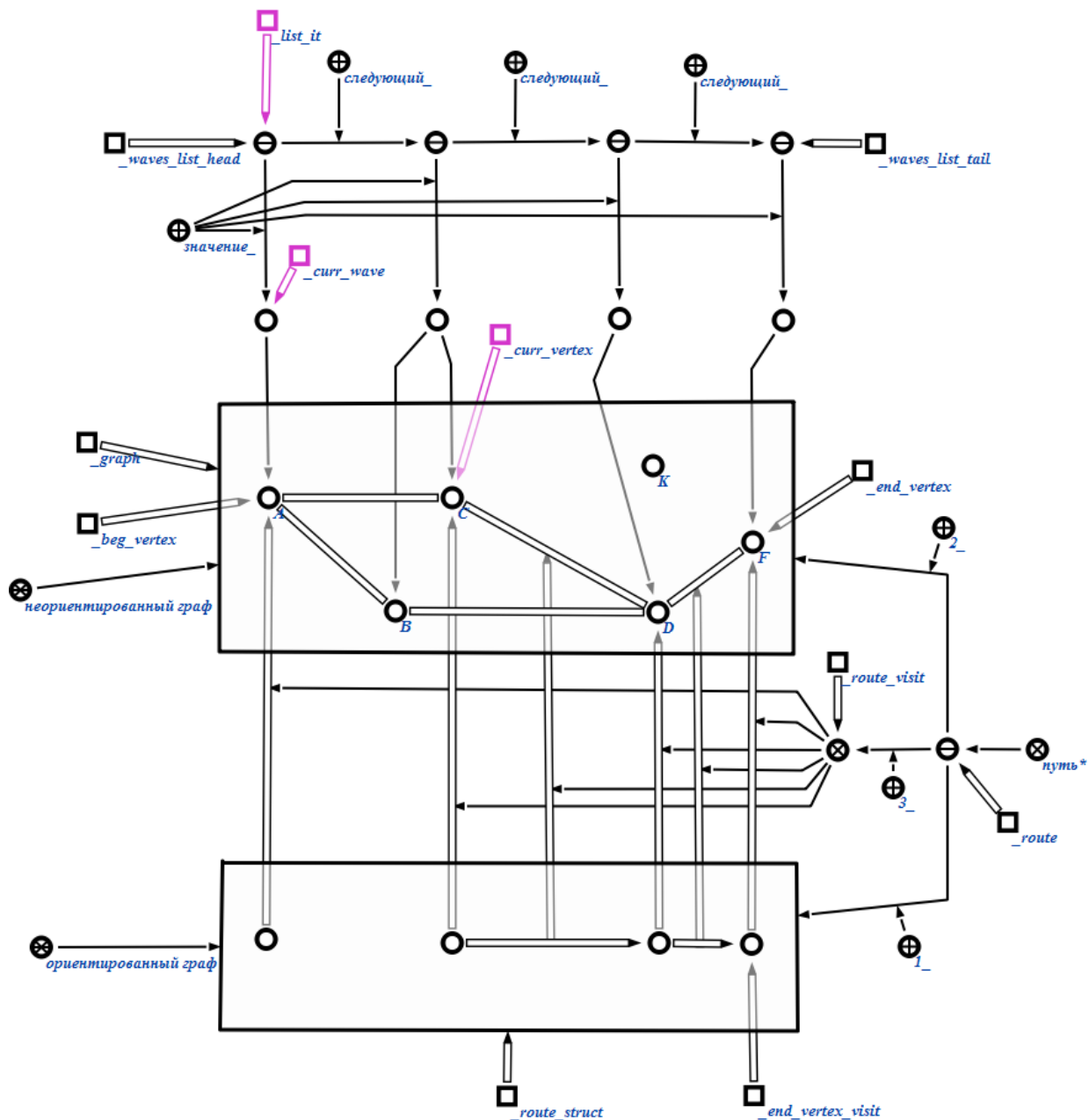
На этом этапе переменная `_curr_vertex` получит в качестве значения вершину, для которой было создано посещение на предыдущей итерации. Переменная `_list_it` – предшествующий элемент списка волн для значения переменной `_curr_wave` на предыдущем этапе. Переменная `_curr_wave` – обрабатываемую волну для элемента списка из установленного значения `_list_it`.

Создание посещения вершины на 2-ой итерации цикла генерации структуры пути (Шаг 8)



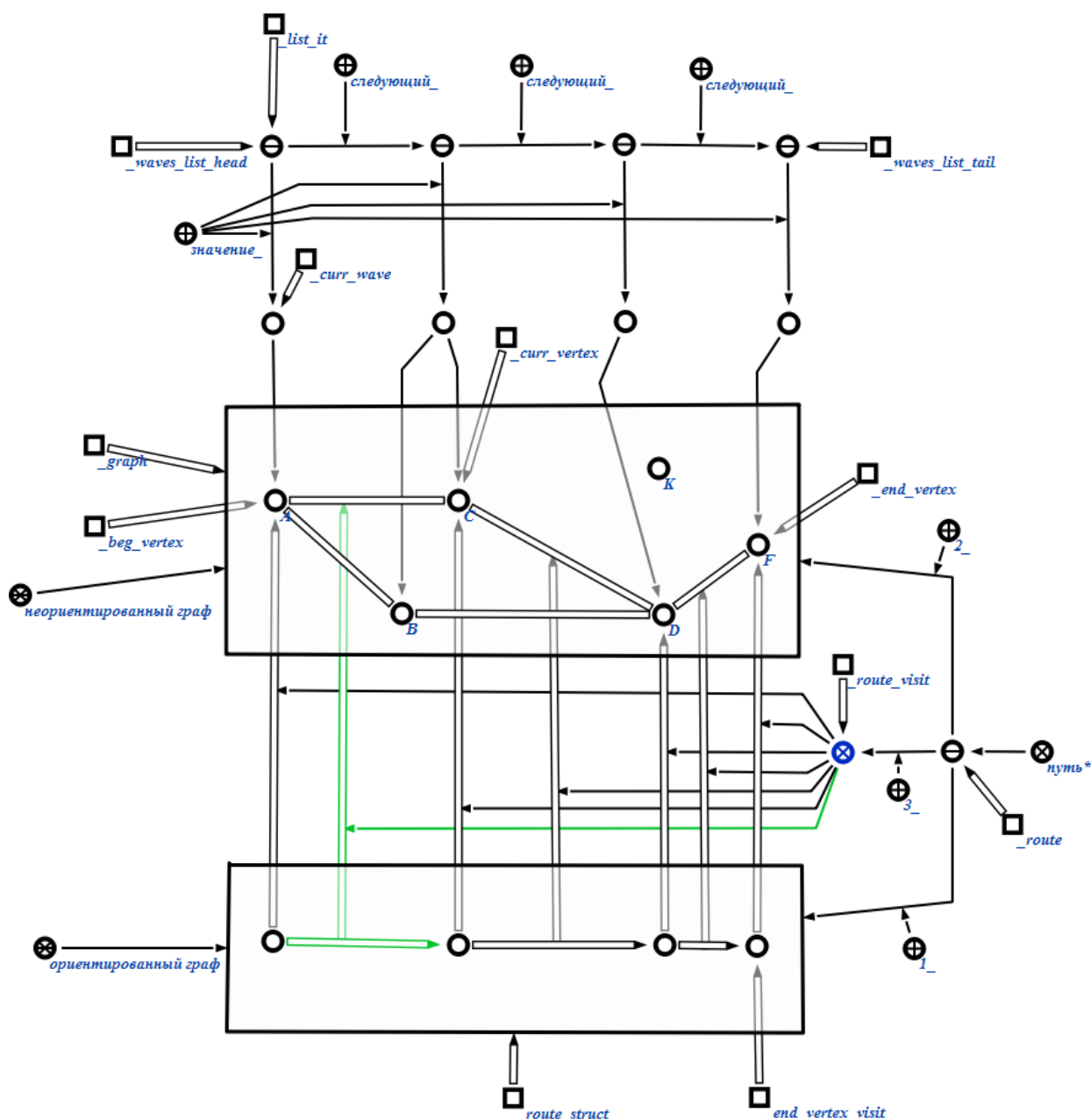
Создадим посещение для одной из вершин из волны `_curr_wave`, которая является смежной вершине из переменной `_curr_vertex`. На эту роль была выбрана вершина `C`. Для ребра, связывающего две вершины, тоже создадим посещение.

Переход к 3-ей итерации цикла генерации структуры пути (Шаг 8)



На этом этапе переменная `_curr_vertex` получит в качестве значения вершину, для которой было создано посещение на предыдущей итерации. Переменная `_list_it` – предшествующий элемент списка волн для значения переменной `_curr_wave` на предыдущем этапе. Переменная `_curr_wave` – обрабатываемую волну для элемента списка из установленного значения `_list_it`.

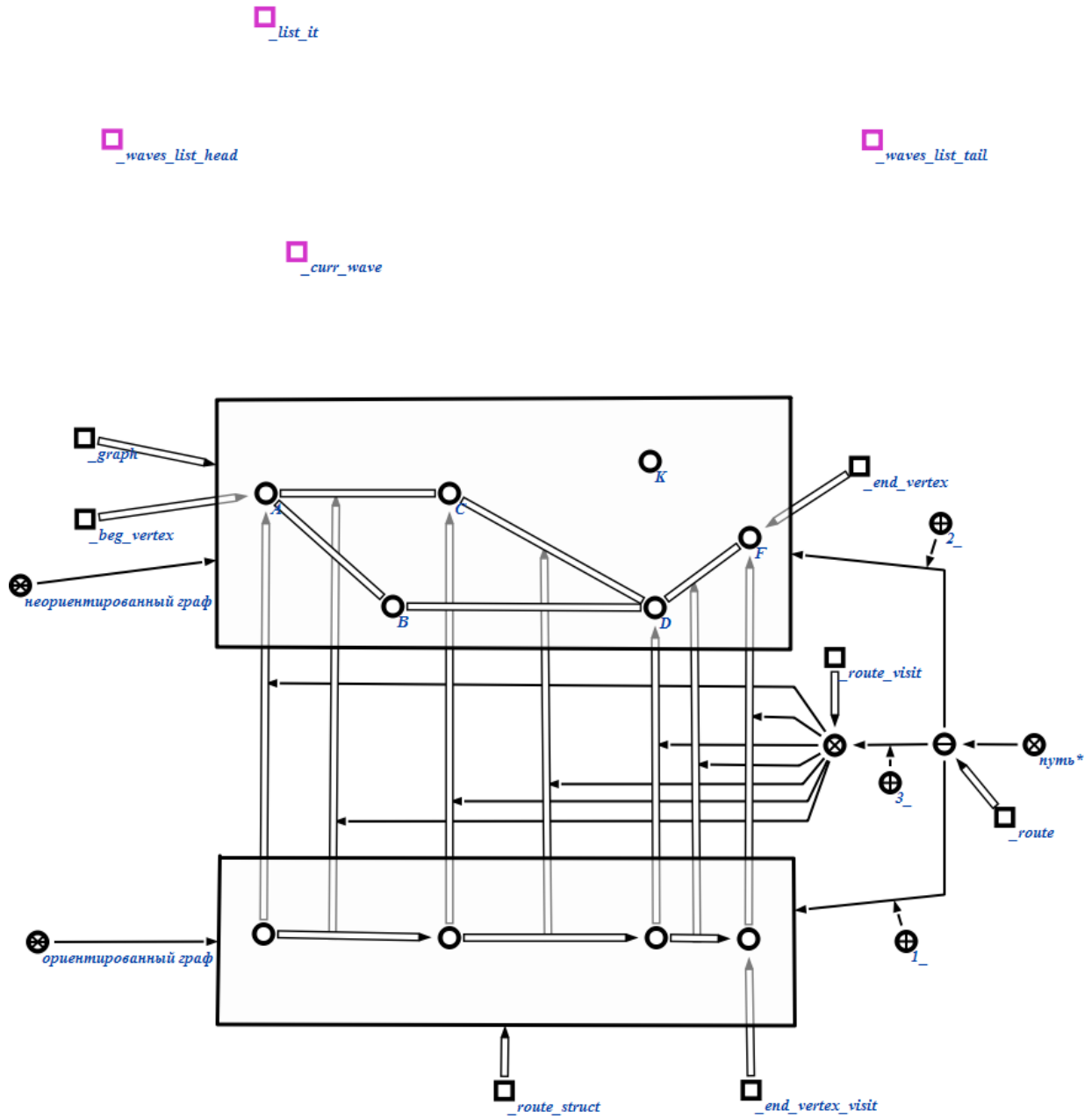
Создание посещения вершины на 3-ей итерации цикла генерации структуры пути
(Шаг 8)



Так как для вершины *A* уже создано посещение, то делать это повторно алгоритм не будет. А вот посещение ребра между вершинами *A* и *C* необходимо создать.

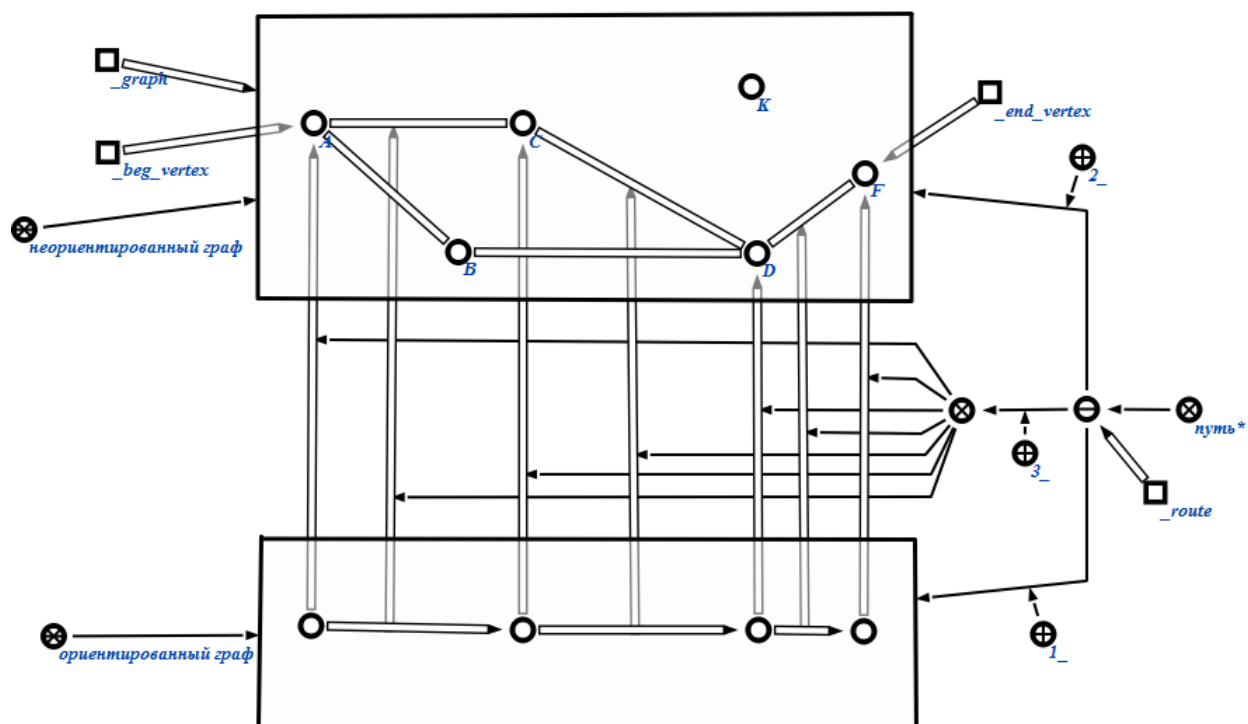
Структура пути создано, поэтому завершаем цикл и переходим к очистке sc-памяти от уже ненужных sc-конструкций.

Удаление списка волн



Удалим список волн. Переменные `_list_it`, `_waves_list_head`, `_curr_wave`, `_waves_list_tail` окажутся без значений.

Результат работы алгоритма



На данном этапе продемонстрирован результат работы алгоритма, значение переменной $_route$ будет возвращено в вызывающий контекст.

Часть II

Курс ППВИС

Введение

Расчетная работа по курсу ППВИС является продолжением расчетной работы по курсу ОИИ и основывается на предыдущих результатах. Теперь вам необходимо будет адаптировать разработанный алгоритм к решению теоретико-графовой задачи в семантической памяти, выполнив следующие этапы:

- разработка программы решения теоретико-графовой задачи в семантической памяти на языке программирования C++ (см. раздел 4)
- разработка программы решения теоретико-графовой задачи на языке программирования, предназначенном для обработки семантических сетей (см. раздел ??)

Все используемые в этой части материалы, исходные тексты, установочные файлы могут быть найдены на кафедральном информационном сервере по следующему пути:

\\Info\StudInfo\~Методическое обеспечение кафедры\~Учебные курсы\2 курс\ППВИС\1sem\Расчётная работа\

Глава 4

Разработка программы решения теоретико-графовой задачи на языке программирования C++ с использованием семантической памяти

Для понимания этой главы студент должен обладать следующими знаниями:

- базовыми знаниями по языку C++
- базовыми знаниями по работе с STL-контейнерами `std::set`, `std::map`, `std::list`, `std::pair`

4.1 Задание

На этом этапе выполнения расчетной работы вам необходимо будет разработать программу с использованием библиотеки моделирования sc-памяти, которая бы решала вашу теоретико-графовую задачу на основе формализации предметной области, проведенной на предыдущем этапе (см. главу 2). Сам алгоритм вы уже исследовали в прошлом семестре (см. главы 1 и 3), но сейчас надо будет не просто реализовать какой-то алгоритм, а адаптировать его к графодинамическому способу обработки информации. Это значит, что вся информация, необходимая для работы вашего алгоритма, должна храниться в sc-памяти и там же обрабатываться. В качестве примера «неудобства», которое вам может принести такое требование, я могу привести невозможность использования привычной матрицы смежности/инцидентности. Поэтому для прохождения этого этапа вам придется взглянуть на алгоритм, решающий выбранную задачу, под другим углом.

В качестве тестов для написанной программы необходимо использовать тестовые примеры, которые вы сделали в ходе предыдущих этапов расчетной работы.

4.2 Установка и настройка рабочей среды

Для начала мы установим и настроим рабочую среду для программирования с использованием программной модели sc-памяти и запустим программу-пример, которая использует эту библиотеку.

Во-первых, нам необходимо скачать и установить программный модуль `sc-core`, который представляет собой ядро для обработки sc-текстов:

- `sc-core-0.2.5-win32.exe` для MS Visual Studio 9

- `sc-core-0.2.5-vc10-bin-win32.exe` для MS Visual Studio 10

После того, как вы скачали инсталлятор, запустите его. Желательно устанавливать `sc-core` так, чтобы полный путь к установленной папке не содержал пробелов в именах директорий. При установке необходимо выбрать опцию, которая добавит директорию исполняемых файлов `sc-core` в переменную среды окружения `PATH` (см. рис. 4.1).

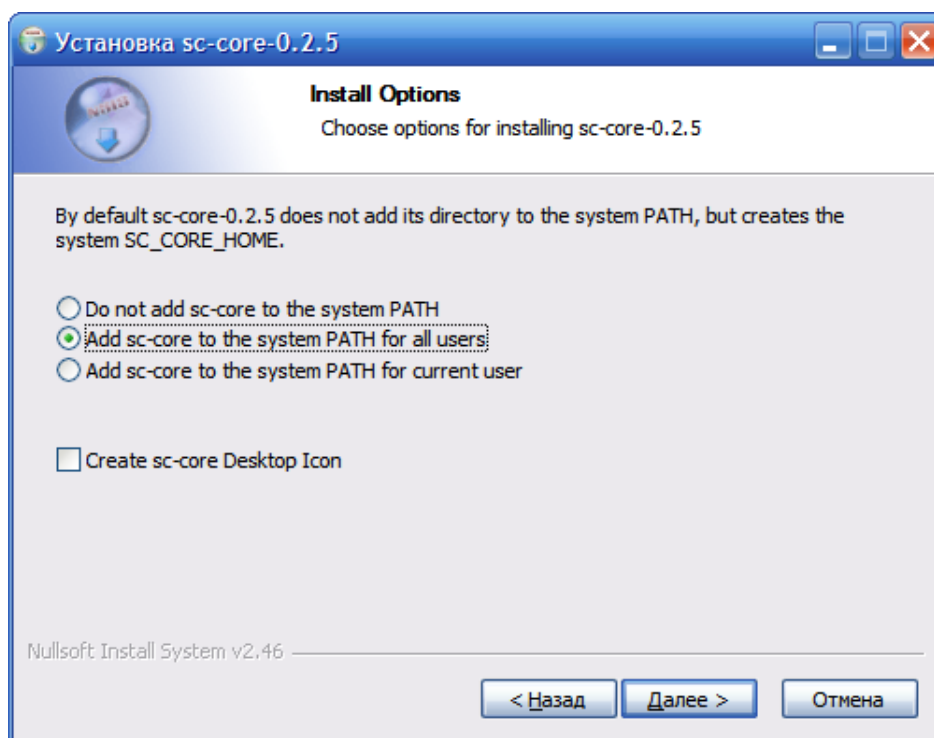


Рис. 4.1: Добавление директории исполняемых файлов `sc-core` в системную переменную `PATH` для всех пользователей

Мной была выбрана установка `sc-core` в папку `c:\sc-core`. После установки данного модуля были изменены следующие переменные среды окружения:

- `SC_CORE_HOME`, которая теперь имеет значение `c:\sc-core`
- `PATH`, к которой была добавлена директория `c:\sc-core\bin`

В дальнейшем для указания пути к директории `sc-core` я буду использовать значение переменной среды окружения `SC_CORE_HOME`.

Для сборки примера нам еще будет необходима программа `CMake`. Необходимо скачать установочный файл для версии не ниже 2.6.2. Как и при установке `sc-core`, при установке `CMake` необходимо выбрать опцию, которая добавит директорию исполняемых файлов в переменную среды окружения `PATH` (см. рис. 4.2).

С модулем `sc-core` версии 0.2.5 идет старая версия примера использования библиотеки моделирования `sc-памяти`, поэтому необходимо с сервера `Info` взять новую версию примера `wave_find_path`. Для этого необходимо скопировать всю папку `wave_find_path` в:

```
%SC_CORE_HOME%\examples\wave_find_path
```

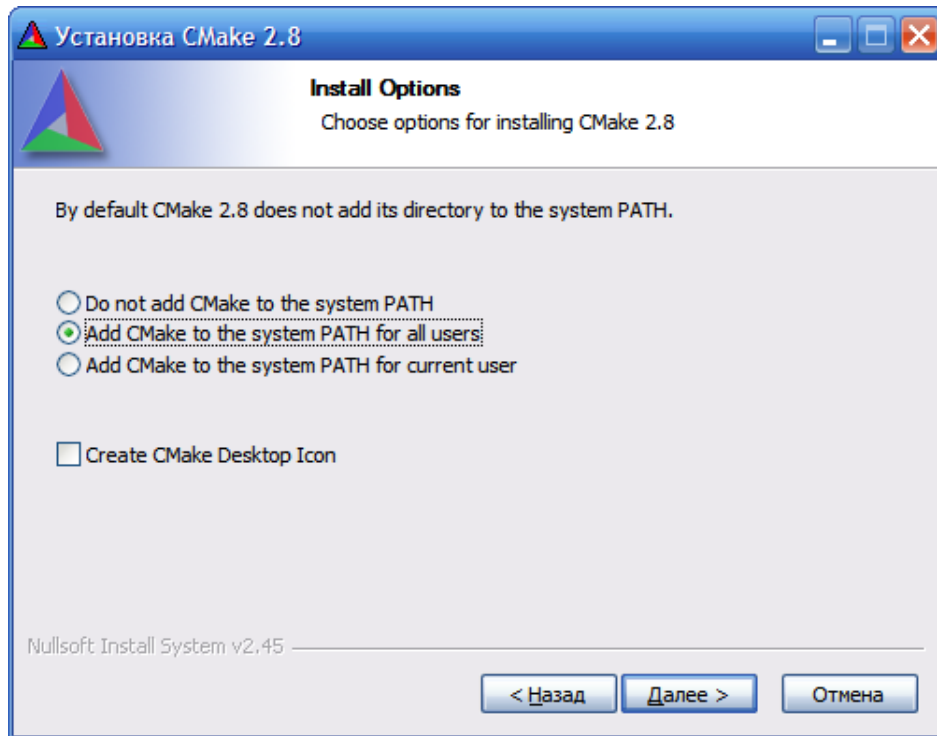


Рис. 4.2: Добавление CMake в системную переменную PATH для всех пользователей

Для генерации проекта примера воспользуемся консолью. Для запуска консоли нажимаем клавиши Win + R, в появившемся диалоге пишем cmd и нажимаем Enter. На экране должно появиться окно, как показано на рис. 4.3.

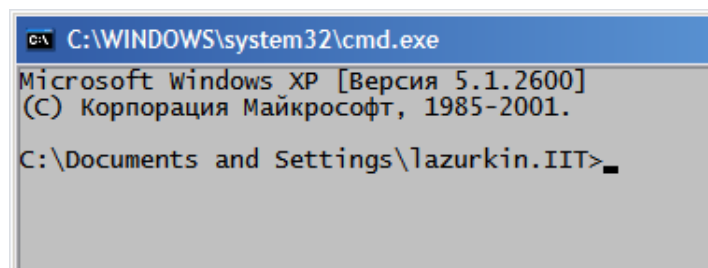


Рис. 4.3: Открытая консоль

Переходим в директорию примера, используя команды:

```
c:
cd %SC_CORE_HOME%\examples\wave_find_path
```

Если у вас Windows с русификацией, то могут возникнуть проблемы с запуском cmake. CMake будет сообщать о том, что он не может скопировать файлы CMakeVSMacros1.vsmacros и CMakeVSMacros2.vsmacros в (это путь на моей файловой системе с моим именем пользователя, для вас он будет отличаться названием директории вашего пользователя):

```
c:\Documents and Settings\lazarukin.IIT\Мои документы\Visual Studio 2008\Projects\VSMacros80\CMakeMacros
```

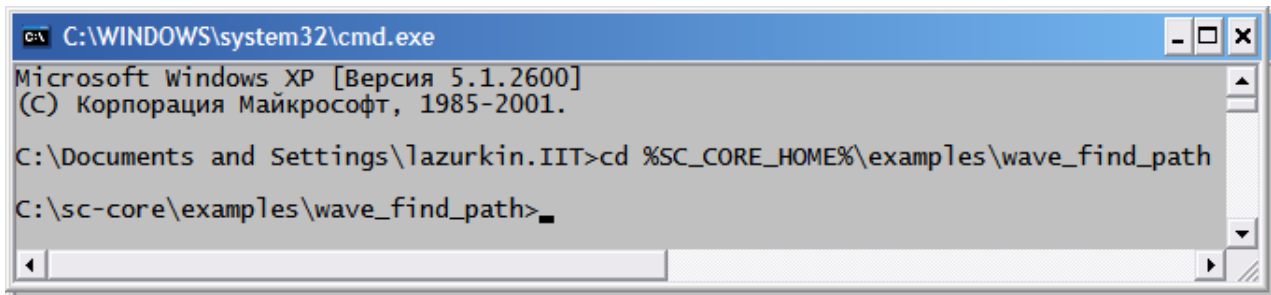


Рис. 4.4: Переход в директорию примера wave_find_path

Эти файлы необходимо скопировать вручную. Для этого просто скопируйте содержимое

```
c:\Program Files\cmake2.8\share\cmake-2.8\Templates\
```

в (путь для пользователя lazurkin.IIT)

```
c:\Documents and Settings\lazurkin.IIT\Мои документы\Visual Studio 2008\Projects\VSMacros80\CMacros
```

При помощи следующей команды сгенерируем проект MS Visual Studio 9.0 для сборки и запуска примера (в cmake есть генератор и для MS Visual Studio 10.0, просто введите команду cmake и на консоль будет выведен список всех доступных генераторов):

```
cmake -G "Visual Studio 9 2008" .
```

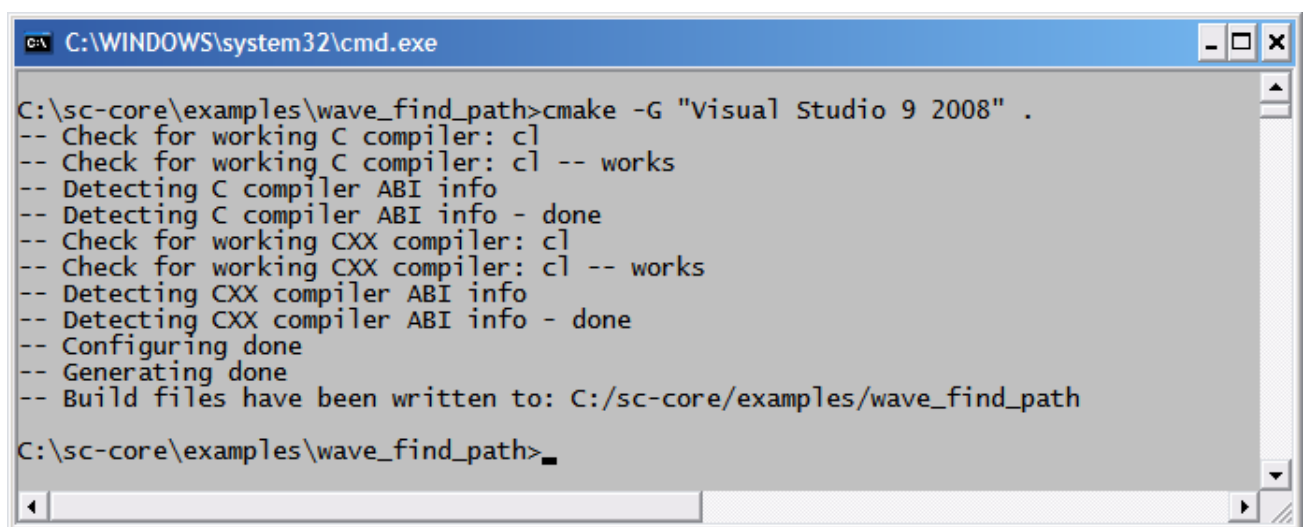


Рис. 4.5: Генерация проекта для примера wave_find_path

Если у вас MS Visual Studio 2010, то можете попробовать команду:

```
cmake .
```

После корректной работы CMake в директории будет создан проект `wave_find_path`. Открываем его при помощи MS Visual Studio 9.0. Нажимаем правой кнопкой мыши на проекте `wave_find_path` в Solution Explorer и выбираем пункт меню "Set as StartUp Project". После этого название данного проекта будет выделено жирным цветом. Теперь можно запустить на исполнение пример `wave_find_path`, который должен отработать со следующим выводом:

```
[Testcase 1]
Graph:
B -- A
C -- B
Find minimal path from 'A' to 'C'
Path:
A -> B -> C
```

```
[Testcase 2]
Graph:
B -- A
C -- A
D -- B
D -- C
F -- D
Find minimal path from 'A' to 'F'
Path:
A -> B -> D -> F
```

```
[Testcase 3]
Graph:
B -- A
C -- A
D -- B
D -- C
F -- D
K
Find minimal path from 'A' to 'K'
Path isn't exist.
```

```
[Testcase 4]
Graph:
V2 -- V1
V3 -- V2
V4 -- V2
V4 -- V3
V5 -- V3
V5 -- V4
```

```
V6 -- V4
V7 -- V5
V7 -- V6
V8 -- V6
V9 -- V6
V9 -- V8
V10 -- V8
V10 -- V9
V11 -- V9
V11 -- V10
Find minimal path from 'V5' to 'V11'
Path:
V5 -> V4 -> V6 -> V9 -> V11
```

```
[Testcase 5]
Graph:
V2 -- V1
V3 -- V2
V4 -- V2
V4 -- V3
V5 -- V3
V5 -- V4
V7 -- V6
V8 -- V6
V8 -- V7
V9 -- V7
V9 -- V8
Find minimal path from 'V1' to 'V9'
Path isn't exist.
```

4.3 Назначение и структура модуля `sc-core`

Модуль `sc-core` – это набор библиотек и программ, которые составляют ядро обработки `sc`-текстов. В него входят следующие динамические библиотеки:

- `libsys` — библиотека, обеспечивающая независимость от операционной системы
- `libtgf` — библиотека обработки формата TGF (Transfer Graph Format)
- `libsc` — библиотека моделирования `sc`-памяти
- `libpm` — библиотека процессорного модуля для обработки `sc`-текстов (например, она включает `scr`-интерпретатор и навигационно-поисковую машину)
- `librgp` — библиотека удаленного подключения к `sc`-памяти по протоколу RGP (Remote Graph Protocol)

Также модуль `sc-core` включает следующие программы:

- `start-pm` — средство запуска процессорного модуля из консоли

- **dumptgf** — утилита для просмотра TGF-файлов в человеко-читаемой форме
- **scs2tgf** — транслятор sc.s-текстов в формат TGF

Как вы уже знаете, при установке инсталлятор модуля **sc-core** создает переменную среды **SC_CORE_HOME**, которая в качестве своего значения будет иметь путь к корневой папке установленного модуля. Если вы заглянете в эту папку, то сможете увидеть следующую структуру подпапок и файлов (пути даются относительно значения переменной окружения **SC_CORE_HOME**):

- **bin/** : бинарные файлы библиотек и программ. В этой директории динамические библиотеки для отладочной версии приложения имеют в конце имени букву «d» (сравните, **libSCd.dll** и **libSC.dll**)
- **doc/** : doxygen-документация на английском языке по модулю **sc-core**
- **examples/** : примеры использования модуля:
 - **wave_find_path/** : C++-пример использования библиотеки sc-памяти **libsc** на основе алгоритма поиска одного из минимальных путей в неориентированном графе
 - **fs_repo_src/** : SCP-пример на основе алгоритма поиска одного из минимальных путей в неориентированном графе
- **include/** : содержит директории с заголовочными файлами для всех библиотек модуля
- **lib/** : библиотеки импорта для всех динамических библиотек модуля В этой директории библиотеки импорта для отладочной версии динамической библиотеки имеют в конце имени букву «d» (сравните, **libSCd.lib** и **libSC.lib**)
- **share/** : содержит дополнительные необходимые ресурсы для работы модуля, которые нельзя отнести ни в одну из уже описанных директорий

Из всего описанного выше в контексте данного этапа нас будет интересовать библиотека моделирования sc-памяти **libsc** и пример ее использования **wave_find_path**. В ходе дальнейшего разговора мы сначала будем рассматривать основные понятия, структуру и классы библиотеки **libsc**, а затем перейдем к более конкретному рассмотрению примера использования этой библиотеки.

4.4 Библиотека моделирования sc-памяти — libsc

В данном разделе мы рассмотрим библиотеку моделирования sc-памяти в объеме, необходимом для выполнения этого этапа расчетной работы.

4.4.1 Общие сведения

Библиотека моделирования sc-памяти написана на языке C++, поэтому в этом разделе мы будем рассматривать в основном классы для моделирования sc-памяти и их методы, а также некоторые функции попадут в поле нашего зрения. Однако сейчас давайте рассмотрим файлы и директории из модуля **sc-core**, которые имеют отношение к библиотеке. И так, приступим (пути даются относительно значения переменной окружения **SC_CORE_HOME**):

- **bin/libSCd.dll** — debug-версия динамической библиотеки моделирования sc-памяти
- **bin/libSC.dll** — release-версия динамической библиотеки моделирования sc-памяти

- `lib/libSCd.lib` — библиотека импорта для динамической библиотеки `libSCd.dll`
- `lib/libSC.lib` — библиотека импорта для динамической библиотеки `libSC.dll`
- `doc/sc-core/` — общая `doxygen`-документация по модулю `sc-core`, в том числе и по рассматриваемой библиотеке `libsc`
- `include/libSC/` : заголовочные файлы рассматриваемой библиотеки
- `examples/wave_find_path` : пример использования рассматриваемой библиотеки

Библиотека `libsc` зависит от библиотеки обработки формата TGF `libtgf`, к которой относятся следующие папки (пути даются относительно значения переменной окружения `SC_CORE_HOME`):

- `bin/libTGFD.dll` — debug-версия динамической библиотеки `libtgf` (она необходима для работы с `libSCd.dll`)
- `bin/libTGF.dll` — release-версия динамической библиотеки `libtgf` (она необходима для работы с `libSC.dll`)
- `include/libTGF/` : заголовочные файлы библиотеки `libtgf`

Теперь перейдем к непосредственному описанию библиотеки `libsc`.

4.4.2 Сегментная модель `sc`-памяти

Когда вы занимались формализацией на `sc.g`, то считали, что `sc`-элементы с одинаковыми идентификаторами должны склеиться. Однако в реализации `sc`-памяти используется сегментная модель и поэтому всё несколько сложнее.

В сегментной модели вся память разбивается на `sc`-сегменты, где каждый `sc`-элемент принадлежит ровно одному сегменту. Уникальность идентификаторов `sc`-элементов имеет место только в рамках `sc`-сегмента. Сами `sc`-сегменты могут организовываться в `sc`-директории. Проиллюстрируем на конкретной структуре `sc`-памяти:

- `/`: корневая `sc`-директория, которая всегда существует
 - `graph_theory/`: `sc`-директория, которая содержит базу знаний по теории графов
 - * `keynode`: `sc`-сегмент, который содержит ключевые узлы базы знаний по теории графов
 - *графовая структура*
 - *вершина_*
 - *связка_*
 - *неориентированный граф*
 - *ребро_* и т.д.
 - `tmp/`: содержит `sc`-сегменты для временной обработки `sc`-конструкций
 - * `wave_find_path` : временный `sc`-сегмент для работы алгоритма поиска одного из минимальных путей в неориентированном графе
 - `proc/`: содержит системные `sc`-сегменты;
 - * `keynode`: `sc`-сегмент, который содержит системные ключевые узлы
 - *1_*

· 2_ и т.д.

Такая структура sc-директорий и sc-сегментов аналогична структуре папок и файлов на файловой системе. Аналогом файла является sc-сегмент, а аналогом содержимого файла — sc-конструкции. В сегментной модели любой объект (sc-директория, sc-сегмент, sc-элемент) может быть однозначно идентифицирован при помощи строки особого вида, которая по историческим причинам называется URI (Universal Resource Identifier). Формат URI аналогичен формату пути к папке или файлу на файловых системах Unix-подобных операционных систем. Вот некоторые примеры URI:

- / — URI sc-директории
- /graph_theory/keynode — URI sc-сегмента
- /graph_theory/keynode/вершина_ — URI sc-элемента

Адресация sc-элементов с использованием URI удобна для человека, но неудобна для компьютера, поэтому существует понятие sc-адреса. SC-адрес — это некоторое число, которое однозначно идентифицирует sc-элемент в сегментной модели sc-памяти. С использованием sc-адресов sc-элементов ведется их обработка в рамках sc-сессии. SC-сессия — это логически единая последовательность операций над некоторой областью sc-памяти (sc-директориями, sc-сегментами, sc-элементами). Для того, чтобы получить доступ к данным sc-сегмента, нужный sc-сегмент должен быть открыт в рамках используемой sc-сессии. Существует особый вид sc-сессии, который называется системная sc-сессия, в рамках которой все существующие sc-сегменты всегда открыты.

Я думаю, что читатель уже получил общие сведения о том, что такое сегментная модель, sc-директория, sc-сегмент, URI, sc-адрес, sc-сессия. Пусть он не волнуется, что из этой сухой теории ему ничего непонятно, потому что к этому разделу ему еще необходимо будет вернуться после изучения примеров кода из следующих разделов. Поэтому переходим к сугубо практическим вопросам использования библиотеки `libsc`.

4.4.3 Начало работы

В прошлом разделе мы рассмотрели логическое устройство сегментной модели sc-памяти, а теперь проведем аналогию с текущей реализацией на языке программирования C++.

Понятию sc-сегмента в библиотеке `libsc` соответствует класс `sc_segment`, который описан в файле `sc_segment.h`. Понятию sc-адреса соответствует `typedef sc_addr` из заголовочного файла `sc_types.h`, а sc-сессии — класс `sc_session` из `libsc.h`. Для представления URI и идентификаторов sc-элементов используется `typedef sc_string` (это просто другое имя для `std::string`), объявленный в `sc_types.h`. Чтобы начать работу со всеми этими типами, достаточно подключить только один заголовочный файл, а именно `libsc.h`, потому что в него включены и `sc_segment.h`, и `sc_types.h`.

Работа с объектами классов `sc_session` и `sc_segment` всегда ведется через указатель на объект, поэтому методы и функции, которые возвращают указатель на объекты этих классов, могут возвращать 0 или NULL, чтобы показать наличие ошибки в процессе выполнения. Если в тексте ниже я буду писать о работе с объектами классов `sc_session` и `sc_segment`, то это почти всегда надо понимать, как работу с такими объектами через указатель.

Рассмотрим, как обстоят дела в этом плане с типом `sc_addr`. В `sc_types.h` он объявлен следующим образом:

```
typedef sc_global_addr *sc_addr;
```

Как видно из приведенного выше объявления, `sc_addr` — это указатель на класс `sc_global_addr`. Поэтому методы и функции, которые возвращают `sc_addr`, могут возвращать 0, NULL или `SCADDR_NIL` (это `define` для 0), чтобы показать наличие ошибки в процессе выполнения. В отличие от `sc_session` и `sc_segment`, с типом `sc_addr` мы работаем без указателя, потому что `sc_addr` — это и так указатель (просто другое имя для указателя на `sc_global_addr`). Напоследок об `sc_addr` стоит сказать следующее:

- для переменных этого типа можно использовать операторы `==` и `!=`, чтобы проверить идет работа с одним и тем же `sc`-элементом или с разными
- у `sc_global_addr` есть открытое поле `sc_global_addr::seg`, которое позволяет получить `sc`-сегмент, в котором находится адресуемый `sc`-элемент
- если в дальнейшем будет идти речь о работе с `sc`-элементами, то это значит, что речь идет о работе с соответствующими им `sc`-адресами

Предыдущий текст должен был вас уже утомить, поэтому разбавим этот раздел примерами кода. Для начала работы с моделью `sc`-памяти ее надо инициализировать, это можно сделать вот так:

```
#include <libsc.h>

// ...

// Инициализируем sc-память при помощи функции libsc_init.
// Она вернет системную sc-сессию
sc_session *system = libsc_init();
```

Теперь необходимо инициализировать системные ключевые узлы (`1_`, `2_` и др.). Все системные ключевые узлы объявлены в заголовочном файле `pm_keynodes.h` и инициализируются функцией `pm_keynodes_init`:

```
#include <pm_keynodes.h>

// ...

// Ключевые узлы будут созданы в sc-сегменте /proc/keynode.
// Например:
// - для работы с ключевым узлом 1_ можно использовать имя N1_
// - для работы с ключевым узлом 2_ можно использовать имя N2_
// - закономерность очевидна ...
pm_keynodes_init(system);
```

В принципе уже можно работать с `sc`-памятью через системную `sc`-сессию `system`, но в идеале лучше работать через пользовательскую `sc`-сессию:

```
// ...

// Получим пользовательскую sc-сессию при помощи функции libsc_login
sc_session *session = libsc_login();

// При помощи метода sc_session::open_segment по URI "/proc/keynode"
// откроем в нашей пользовательской sc-сессии sc-сегмент системных
// ключевых узлов
session->open_segment("/proc/keynode");
```

Теперь создадим уникальный sc-сегмент для исследования работы sc-памяти:

```
// Этот заголовочный файл содержит функции, которые облегчают работу с
// sc-сегментами
#include <segment_utils.h>

// ...

// Создадим уникальный sc-сегмент с использованием
// create_unique_segment и URI этого sc-сегмента
// будет начинаться с /tmp/test
sc_segment *segment = create_unique_segment(session, "/tmp/test");

// Созданный при помощи sc-сессии session sc-сегмент будет автоматический
// открыт в ней
```

Пользовательская sc-сессия `session` и созданный sc-сегмент `segment` будут использоваться во всех дальнейших примерах без какого-то явного объявления.

После окончания необходимой работы sc-память нужно почистить:

```
// Удалим созданный sc-сегмент segment при помощи метода sc_session::unlink.
// Метод sc_segment::get_full_uri возвращает URI sc-сегмента.
session->unlink(segment->get_full_uri());

// Закроем пользовательскую sc-сессию.
session->close();

// Деинициализируем sc-память.
libsc_deinit();

// Обратите внимани на то, что явно при помощи оператора delete никакая
// память не освобождается. При работе с типами sc_session,
// sc_segment, sc_addr память явно освобождать не надо.
```

Перед тем, как мы будем рассматривать операции генерации и поиска в sc-памяти, вам еще нужно познакомиться со способом задания типа sc-элементов, поэтому переходим к следующему разделу.

4.4.4 Тип sc-элемента

Для задания типа sc-элемента в библиотеке `libsc` используется C++-тип `sc_type`, который объявлен в файле `sc_types.h`. Значение типа `sc_type` – это просто число, которое является результатом операции побитового ИЛИ некоторых числовых констант. Каждая из числовых констант задает значение свойства из заранее определенного диапазона. Для выполнения расчетной работы вам достаточно знать о следующих свойствах:

- Структурный тип sc-элемента. Используемые числовые константы C++:
 - `SC_UNDF` — sc-элемент неопределенного типа
 - `SC_ARC` — sc-дуга
 - `SC_NODE` — sc-узел
- Константность sc-элемента. Используемые числовые константы C++:






















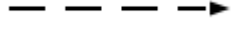
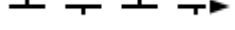




- SC_CONST — константность
- SC_VAR — переменность
- SC_METAVAR — метапеременность

- Нечеткость sc-дуги. Используемые числовые константы C++:

- SC_POS — позитивность
- SC_NEG — негативность
- SC_FUZ — нечеткость

С использованием описанных выше числовых констант, например, константный sc-узел можно задать как SC_NODE|SC_CONST, а позитивную константную sc-дугу как SC_ARC|SC_CONST|SC_POS. В таблице 4.1 приведены соответствующие значения типа `sc_type` для всех необходимых sc.g-элементов. Обратите внимание, что на уровне библиотеки `libsc` нет разницы в кодировании структурных типов sc.g-элементов (например, константный sc-атрибут и константное sc-отношение кодируются как константный sc-узел).

Таблица 4.1: Соответствие SCg-элемента значению типа `sc_type` из `libsc`

	SC_UNDF SC_CONST , SC_U_CONST
	SC_UNDF SC_VAR, SC_U_VAR
	SC_UNDF SC_METAVAR , SC_U_METAVAR
    	SC_NODE SC_CONST, SC_N_CONST
    	SC_NODE SC_VAR, SC_N_VAR
    	SC_NODE SC_METAVAR, SC_N_METAVAR
	SC_ARC SC_CONST SC_POS , SC_A_CONST SC_POS
	SC_ARC SC_CONST SC_FUZ , SC_A_CONST SC_FUZ
	SC_ARC SC_CONST SC_NEG , SC_A_CONST SC_NEG
	SC_ARC SC_VAR SC_POS , SC_A_VAR SC_POS
	SC_ARC SC_VAR SC_FUZ , SC_A_VAR SC_FUZ
	SC_ARC SC_VAR SC_NEG , SC_A_VAR SC_NEG
	SC_ARC SC_METAVAR SC_POS , SC_A_METAVAR SC_POS
	SC_ARC SC_METAVAR SC_FUZ , SC_A_METAVAR SC_FUZ
	SC_ARC SC_METAVAR SC_NEG , SC_A_METAVAR SC_NEG

Для получения и установки типа sc-элемента используются методы `sc_session::get_type` и `sc_session::change_type`. Более подробную информацию о них можно найти в `doxygen`-документации.

Бинарные неориентированные (рис. 4.6) и ориентированные пары (рис. 4.7) в текущей версии библиотеки `libsc` нельзя представить в виде атомарных элементов, а представляются они так, как показано на рис. 4.8 и 4.9 соответственно.

Как видно из приеденных рисунков бинарная неориентированная пара представляется при помощи трех, а не одного sc-элемента. Бинарная ориентированная пара представляется при помощи пяти sc-элементов, два из которых (sc-узлы `1_` и `2_`) находятся в sc-сегменте `/proc/keynode` и являются ключевыми узлами.

В следующем разделе мы рассмотрим виды основных обрабатываемых sc-конструкций.

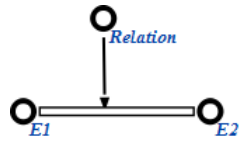


Рис. 4.6: Бинарная неориентированная пара

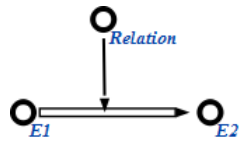


Рис. 4.7: Бинарная ориентированная пара

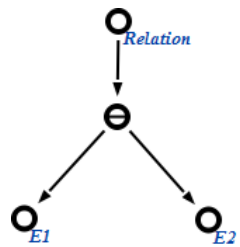


Рис. 4.8: Кодирование бинарной неориентированной пары

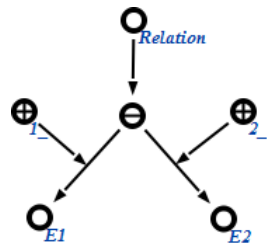


Рис. 4.9: Кодирование бинарной ориентированной пары

4.4.5 Основные обрабатываемые sc-конструкции

В sc-памяти sc-элементы могут обрабатываться как поэлементно, так и целыми группами. Можно выделить два вида таких групп sc-элементов.

Самой распространенной является трехэлементная sc-конструкция (по простому — «тройка»). Она состоит из sc-узла и sc-элемента произвольного типа, которые связаны sc-дугой. Пример частного случая «тройки» приведен на рис. 4.10 (обратите внимание на нумерацию элементов). Этот частный случай «тройки» включает в качестве первого элемента - константный sc-узел, второго элемента – константную позитивную sc-дугу, в качестве третьего элемента - константный sc-узел.

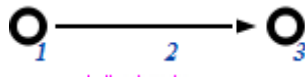


Рис. 4.10: Трехэлементная sc-конструкция

Не менее распространенной является пятиэлементная sc-конструкция (по простому — «пятёрка»), частный случай которой приведен на рис. 4.11. Этот частный случай «пятёрки» включает в качестве 1-го элемента – константный sc-узел, 2-го элемента – константную позитивную sc-дугу, 3-го элемента – константный sc-узел, 4-го элемента - константную позитивную sc-дугу, 5-го элемент – константный sc-узел.

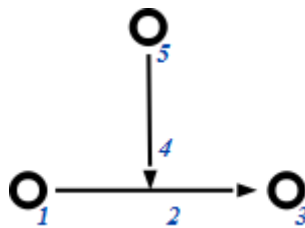


Рис. 4.11: Пятиэлементная sc-конструкция

Чаще всего «пятёрка» используется, когда второй элемент (sc-дуга) уточняется атрибутом, который является пятым элементом «пятёрки».

4.4.6 Генерация и удаление sc-конструкций в sc-памяти

Для генерации одноэлементной sc-конструкции предназначен метод `sc_session::create_el`. Пример генерации константного sc-узла (если sc-узел будет сгенерирован, то переменная получит в качестве значения sc-адрес этого узла, иначе она получит в качестве значения нулевой указатель):

```
sc_addr node = session->create_el(  
    segment, // sc-сегмент, в котором будет сгенерирован sc-элемент  
    SC_N_CONST // тип sc-элемента  
);
```

Давайте сгенерируем два константных sc-узла и присвоим им идентификаторы при помощи метода `sc_session::set_idtf` (к слову, для получения идентификатора служит метод `sc_session::get_idtf`):

```
sc_addr e1 = session->create_el(segment, SC_N_CONST);  
sc_addr e3 = session->create_el(segment, SC_N_CONST);
```



Рис. 4.12: Содержимое sc-сегмент segment после генерации двух sc-узлов

```
session->set_idtf(e1, "First");  
session->set_idtf(e3, "Third");
```

В данной версии модели sc-памяти все sc-элементы всегда имеют уникальный идентификатор, поэтому не удивляйтесь, если sc-элементы, для которых вы не устанавливали идентификатор, будут иметь в качестве него строки странного содержания. Это нормально. Вернемся к нашему примеру. Содержимое sc-сегмента `segment` после выполнения предыдущего кода показано на рис. 4.12.

В классе `sc_session` есть специальный метод `gen3_f_a_f`, который генерирует второй элемент «тройки», если известны ее первый и третий элементы. Его сигнатура выглядит следующим образом:

```
class sc_session  
{  
public:  
    // ...  
    virtual sc_retval gen3_f_a_f(  
        // sc-адрес 1-го элемента  
        sc_addr e1,  
  
        // по этому адресу поместить sc-адрес 2-го элемента  
        sc_addr *e2,  
  
        // sc-сегмент, в котором будет сгенерирован 2-ой элемента  
        sc_segment *seg2,  
  
        // тип 2-го элемента  
        sc_type t2,  
  
        // sc-адрес 3-го элемент  
        sc_addr e3  
    ) = 0;  
    // ...  
};
```

Генерацию «тройки» с sc-узлами *First* и *Third* можно провести следующим образом:

```
// переменная для sc-адреса 2-го элемента трехэлементной sc-конструкции.  
sc_addr e2 = 0;  
  
// Генерация sc-дуги между 1-ым и 3-им элементами.  
session->gen3_f_a_f(  
    // 1-ый элемент  
    e1,  
  
    // в e2 будет помещен sc-адрес 2-го элемента  
    &e2,  
  
    // sc-сегмент, в котором будет сгенерирован 2-ой элемент
```

```

segment,

// тип 2-ого элемента - константная позитивная sc-дуга
SC_A_CONST|SC_POS,

// 3-ий элемент
e3
);

```

После выполнения приведенного выше кода содержимое sc-сегмента `segment` изменится так, как показано на рис. 4.13.

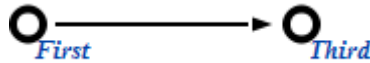


Рис. 4.13: Пример генерации трехэлементной sc-конструкции

Но в некоторых случаях нам не надо получать sc-адрес второго элемента при использовании метода `sc_session::gen3_f_a_f`. Тогда можно в качестве второго аргумента этого метода передавать нулевой указатель:

```

// Генерация sc-дуги между 1-ым и 3-им элементами.
session->gen3_f_a_f(
    // 1-ый элемент
    e1,

    // использован нулевой указатель,
    // а это значит что метод не будет возвращать sc-адрес
    // 2-го элемента трехэлементной sc-конструкции
    0,

    // sc-сегмент, в котором будет сгенерирован 2-ой элемент
    segment,

    // тип 2-ого элемента - константная позитивная sc-дуга
    SC_A_CONST|SC_POS,

    // 3-ий элемент
    e3
);

```

Запомните этот прием передачи нулевого указателя, потому что он используется во многих методах и функциях `libsc`, которые должны возвращать более одного аргумента.

Теперь обратим внимание на возвращаемое значение метода `sc_session::gen3_f_a_f`. Оно имеет тип `sc_retval` (sc-код возврата) и является просто числом, по которому можно определить код произошедшей ошибки. Тип `sc_retval` используется во многих методах и функциях библиотеки, и вам необходимо знать о следующих значениях этого типа:

- Константа `RV_OK` — функция или метод отработал успешно
- Константа `RV_ERR_GEN` — в процессе работы функции или метода произошла ошибка
- Константа `RV_THEN` — функция или метод сообщает о том, что необходим переход по then-ветке условного оператора

- Константа `RV_ELSE_GEN` — функция или метод сообщает о том, что необходим переход по `else`-ветке условного оператора

В случае успешного выполнения метод `sc_session::gen3_f_a_f` возвратит `RV_OK`, а в случае неуспешного — `RV_ERR_GEN`. Проверки можно организовать следующим образом:

```
if (session->gen3_f_a_f(...) == RV_OK) {
    // Генерация прошла успешно
} else {
    // Возникла ошибка в процессе генерации
}

// Зная о том, что константа RV_OK равна 0 можно
// организовать проверку следующим образом.
if (!session->gen3_f_a_f(...)) {
    // Генерация прошла успешно
} else {
    // Возникла ошибка в процессе генерации
}

// Зная о том, что код ошибки не равен 0 можно
// организовать проверку ошибочной ситуации следующим образом.
if (session->gen3_f_a_f(...)) {
    // Возникла ошибка в процессе генерации
}
```

Если вы не пишете какой-то специальный устойчивый код, то такие проверки в алгоритме генерации делать нет необходимости, потому что ошибочный код возврата метод `sc_session::gen3_f_a_f` возвращает в следующих случаях:

- не удалось выделить память под генерируемые элементы;
- `sc`-сегмент, в котором происходит генерация 2-го элемента «тройки» закрыт в рамках данной `sc`-сессии или не существует;
- указан неверный тип 2-го элемента «тройки» (2-ой элемент должен быть всегда `sc`-дугой);
- `sc`-адреса 1-го и 3-го элементов недействительны в рамках данной `sc`-сессии (эти элементы могут быть уже удалены или их `sc`-сегменты закрыты в рамках данной `sc`-сессии).

Напоследок давайте попробуем написать функцию `my_gen3_f_a_f`, которая делает то же самое, что и метод `sc_session::gen3_f_a_f`. Код этой функции будет следующим:

```
sc_retval my_gen3_f_a_f(
    sc_session *s, // sc-сессия, через которую будет идти работа
    sc_addr e1,
    sc_addr *e2,
    sc_segment *s2,
    sc_type t2,
    sc_addr e3)
{
    // Генерация 2-го элемента
    sc_addr ce2 = s->create_el(s2, t2);
    if (!ce2)
        return RV_ERR_GEN;
```

```

// Установка начала и конца для sc-дуги ce2 (2-го элемента "тройки")
if (s->set_beg(ce2, e1) || s->set_end(ce2, e3)) {
    // Произошла ошибка и необходимо удалить ce2
    s->erase_el(ce2);
    return RV_ERR_GEN;
}

if (e2)
    *e2 = ce2;

return RV_OK;
}

```

В приведенном выше коде для вас должны быть незнакомы только методы `sc_session::set_beg` и `sc_session::set_end`. Они используются для установки начала и конца sc-дуги (для получения начала и конца используются методы `sc_session::get_beg` и `sc_session::get_end` соответственно) и объявлены следующим образом:

```

class sc_session
{
public:
    // ...
    virtual sc_retval set_beg(sc_addr arc, sc_addr beg) = 0;
    virtual sc_retval set_end(sc_addr arc, sc_addr end) = 0;
    // ...
};

```

А теперь давайте вернем sc-сегмент `segment` из теперешнего состояния (рис. 4.13) в состояние, которое показано на рис. 4.12. Для этого нам надо удалить недавно сгенерированную константную позитивную sc-дугу. Удаление sc-элементов обеспечивается при помощи метода `sc_session::erase_el`:

```

class sc_session
{
public:
    // ...
    virtual sc_retval erase_el(sc_addr el) = 0;
    // ...
};

```

Следующий код осуществит удаление sc-дуги:

```
session->erase_el(e2);
```

Теперь добавим еще один константный sc-узел к уже существующим двум:

```

sc_addr e5 = session->create_el(segment, SC_N_CONST);
session->set_idtf(e5, "Fifth");

```

После выполнения приведенного выше куска кода состояние sc-сегмента будет таким, как показано на рис. 4.14.

В классе `sc_session` есть специальный метод `gen5_f_a_f_a_f`, который генерирует 2-ой и 4-ый элементы «пятёрки», если известны ее 1-ый, 3-ий и 5-ый элементы. Его сигнатура выглядит следующим образом:



Рис. 4.14: Содержимое `sc`-сегмента `segment` после генерации третьего `sc`-узла

```
class sc_session
{
public:
    // ...
    virtual sc_retval gen5_f_a_f_a_f(
        // sc-адрес 1-го элемент
        sc_addr e1,

        // по этому адресу поместить sc-адрес 2-го элемент
        sc_addr *e2,

        // sc-сегмент, в котором будет сгенерирован 2-ой элемент
        sc_segment *seg2,

        // тип 2-го элемента
        sc_type t2,

        // sc-адрес 3-го элемент
        sc_addr e3,

        // по этому адресу поместить sc-адрес 4-го элемент
        sc_addr *e2,

        // sc-сегмент, в котором будет сгенерирован 4-ой элемент
        sc_segment *seg4,

        // тип 4-го элемента
        sc_type t4,

        // sc-адрес 5-го элемент
        sc_addr e5
    ) = 0;
    // ...
};
```

Генерацию пятиэлементной `sc`-конструкции с `sc`-узлами *First*, *Third*, *Fifth* можно провести следующим образом:

```
// переменная для sc-адреса 2-го элемента пятиэлементной sc-конструкции.
sc_addr e2 = 0;

// переменная для sc-адреса 4-го элемента пятиэлементной sc-конструкции.
sc_addr e4 = 0;
```

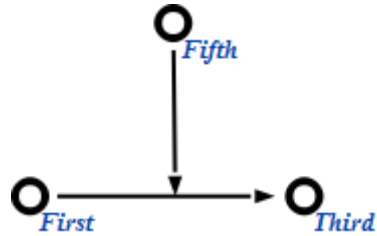


Рис. 4.15: Пример генерации «пятьёрки»

```
// Генерация sc-дуги между 1-ым и 3-им, 5-ым и 2-ым элементами.
session->gen5_f_a_f_a_f(
    // 1-ый элемент
    e1,

    // в e2 будет помещен sc-адрес 2-го элемента
    &e2,

    // sc-сегмент, в котором будет сгенерирован 2-ой элемент
    segment,

    // тип 2-ого элемента - константная позитивная sc-дуга
    SC_A_CONST|SC_POS,

    // 3-ий элемент
    e3,

    // в e4 будет помещен sc-адрес 4-го элемента
    &e4,

    // sc-сегмент, в котором будет сгенерирован 4-ой элемент
    segment,

    // тип 4-ого элемента - константная позитивная sc-дуга
    SC_A_CONST|SC_POS,

    // 5-ий элемент
    e5
);
```

После выполнения приведенного выше куска кода состояние sc-сегмента будет таким, как показано на рис. 4.15.

Метод `sc_session::gen5_f_a_f_a_f` работает аналогично методу `sc_session::gen3_f_a_f`, поэтому для него справедливо все сказанное ранее про `sc_session::gen3_f_a_f`.

4.4.7 Работа с содержимым sc-элементов

Каждому sc-элементу в программной модели sc-памяти может быть установлено содержимое следующих типов (`enum TCont` из заголовочного файла `sc_content.h`):

- строковое (`TCSTRING`);
- целочисленное (`TCINT`);

- вещественное (TCREAL);
- бинарное (TCDATA);
- отложенное (TCLAZY);
- пустое (TCEMPT).

Для представления содержимого используется класс `Content` (объявлен в файле `sc_content.h`), а для получения содержимого `sc`-элемента предназначен метод `sc_session::get_content`:

```
Content content = session->get_content(e1);

// Изначально у sc-элемента содержимое является пустым
if (content.type() == TCEMPT) {
    // У sc-элемента пустое содержимое
}
```

В приведенном выше примере используется метод `Content::type`, который возвращает тип содержимого через значение `TCont`. Для установки содержимого `sc`-элементу необходимо использовать метод `sc_session::set_content` и статические методы `Content::integer`, `Content::real`, `Content::string`, `Content::data`:

```
// Установка содержимого типа TCINT
session->set_content(first_el, Content::integer(1));

// Установка содержимого типа TCREAL
session->set_content(first_el, Content::real(0.05));

// Установка содержимого типа TCSTRING
session->set_content(first_el, Content::string("Hello_world!!!"));

int my_data[] = { 1, 2, 3, 4, 5 };
// Установка содержимого типа TCDATA
session->set_content(first_el, Content::data(sizeof(my_data), my_data));
```

Чтобы получить доступ к данным объекта класса `Content` необходимо использовать `union Cont`. Для пояснения я продемонстрирую реализацию функции вывода на консоль содержимого `sc`-элемента:

```
sc_retval print_content(const Content &content)
{
    // Класс Content имеет перегруженный оператор
    // приведения к типу Cont, поэтому возможна следующая запись:
    Cont c = content;

    switch (content.type()) {
    case TCSTRING:
        std::cout << c.d.ptr;
        break;
    case TCINT:
        std::cout << c.i;
        break;
    case TCREAL:
        std::cout << (double) c.r;
        break;
```

```

    case TCDATA:
        std::cout.write(c.d.ptr, c.d.size);
        break;
    case TCEMPTY:
        break;
    default:
        return RV_ERR_GEN;
}

return RV_OK;
}

```

Напоследок хочу сказать, что существует еще метод `sc_session::erase_content`, который уничтожает текущее содержимое `sc`-элемента. После применения метода `sc_session::erase_content` `sc`-элемент будет иметь пустое содержимое.

4.4.8 Поиск в `sc`-памяти

Базовые механизмы поиска по шаблону

Поиск в `sc`-памяти происходит по специальным шаблонам, которые называются ограничениями (constraints). Для перебора конкретных результатов поиска используется реализация идеи [итераторов](#) через интерфейс `sc_iterator`, объявленный в заголовочном файле `sc_iterator.h`:

```

/// Интерфейс итератора по различным видам sc-конструкций.
class sc_iterator
{
public:
    /// Переводит итератор на следующий результат поиска.
    virtual sc_retval next() = 0;

    /// @return true, если нет sc-конструкции, к которой можно перейти.
    /// @return false - в противном случае.
    virtual bool is_over() = 0;

    /// @return sc-адрес sc-элемента с порядковым номером @p num
    /// в найденной sc-конструкции.
    virtual sc_addr value(sc_uint num) = 0;

    virtual ~sc_iterator();
};

```

Для того, чтобы понять, как осуществлять поиск в `sc`-памяти, ниже мы будем рассматривать поиск в `sc`-памяти по различным видам ограничений. Первые ограничения, на которые мы обратим своё внимание, будут ограничения вида `CONSTR_3_*_*_*`, которые используются для поиска «троек». Ограничения `CONSTR_3_*_*_*` бывают следующие:

- `CONSTR_3_f_a_a`
- `CONSTR_3_f_a_f`
- `CONSTR_3_a_a_f`

Первое, что бросается в глаза в приведенных выше названиях, это странный набор букв «а» и «f». Возьмем `CONSTR_3_f_a_a` в качестве примера. Этот вид ограничения используется для

поиска «троек», а, как вы уже знаете, элементы таких sc-конструкций пронумерованы. Поэтому 1-ая буква «f» в названии вида ограничения расшифровывается как английское слово «fixed» и обозначает то, что 1-ый элемент «тройки» известен. 2-ая и 3-я буквы «a» в названии вида ограничения расшифровываются как английское слово «assign» и обозначают то, что 2-ой и 3-ий элементы неизвестны, и их необходимо найти.

Посмотрим, как будет производиться поиск по такому виду ограничения. Для этого допустим, что содержимое одного из sc-сегментов как показано на рис. 4.16. Так как поиск проходит в рамках sc-сессии, то этот sc-сегмент должен быть открыт в нашей sc-сессии.

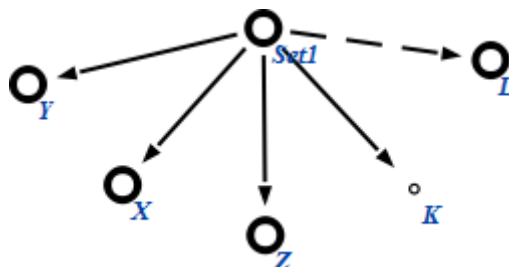


Рис. 4.16: Содержимое sc-сегмента для поиска по CONSTR_3_f_a_a

Нам надо найти все константные sc-узлы, связанные константными позитивными sc-дугами с известным sc-узлом *Set1*. В этом случае ограничения вида CONSTR_3_f_a_a параметризуется следующим образом:

1. sc-адрес sc-узла *Set1*
2. SC_A_CONST|SC_POS (тип константной позитивной sc-дуги)
3. SC_CONST|SC_NODE (тип константного sc-узла)

Как вы возможно уже догадались, если параметр для вида ограничения фиксирован («fixed»), то необходимо передавать sc-адрес, если параметр не фиксирован («assign»), то необходимо передавать шаблон-маску *sc_type* для поиска. Для формирования шаблона-маски *sc_type* нужно применять правило «Чем меньше признаков задано, тем больше будет найдено». Посмотрим на примерах, как реализуется это правило:

- SC_EMPTY или 0: под такую маску подойдет любой sc-элемент
- SC_NODE: под такую маску подойдет любой sc-узел
- SC_ARC|SC_POS: под такую маску подойдет любая позитивная sc-дуга

Вернемся к нашему примеру. Следующий код выведет на консоль идентификаторы всех константных sc-узлов, которые связаны с sc-узлом *Set1* константной позитивной sc-дугой:

```

// Создаем итератор при помощи метода sc_session::create_iterator
sc_iterator *it = session->create_iterator(
    // Создаем ограничение вида CONSTR_3_f_a_a
    sc_constraint_new(
        CONSTR_3_f_a_a, // вид ограничения
        Set1, // 1-ый параметр
        SC_A_CONST|SC_POS, // 2-ой параметр
        SC_CONST|SC_NODE // 3-ий параметр
    ),

```

```

    // Указываем, что итератор удалит ограничение
    // Здесь всегда указывайте true
    true
);

// Типичный цикл по результатам поиска
while (!it->is_over()) {

    // !!!! ОБРАТИТЕ ВНИМАНИЕ:
    // !!!! Индексация с использованием метода
    // !!!! sc_iterator::value работает как в массивах, т.е. с 0

    // Получаем 1-ый элемент найденной sc-конструкции
    // Им всегда будет sc-узел Set1
    sc_addr first = it->value(0);

    // Получаем 2-ой элемент найденной sc-конструкции
    sc_addr second = it->value(1);

    // Получаем 3-ий элемент найденной sc-конструкции
    sc_addr third = it->value(2);

    // Вывод строки на консоль
    std::cout << session->get_idtf(first) << "└─>"
        << session->get_idtf(third) << "\n";

    // Перевод итератора к следующей найденной sc-конструкции
    it->next();
}

// !!!! ОБРАТИТЕ ВНИМАНИЕ:
// Надо обязательно освободить память, выделенную под итератор
delete it;

```

Приведенный выше код в результате своего исполнения выведет на консоль (порядок может быть другим):

```

Set1 -> Y
Set1 -> X
Set1 -> Z

```

В этом коде я применил следующие незнакомые для вас функции и методы:

- метод `sc_session::create_iterator` для создания итератора по sc-конструкциям, подходящим под ограничение
- функцию `sc_constraint_new` для создания ограничения определенного вида. Эта функция принимает неопределенное число аргументов, но первым аргументом всегда должна быть константа, которая задает вид ограничения

На основе приведенного примера мы можем построить типовой код для работы с объектами класса `sc_iterator`. При помощи цикла `while` это можно сделать следующим образом:


```

// Создание итератора.

sc_iterator *it = session->create_iterator(
    sc_constraint_new(
        // Здесь должны быть параметры ограничения.
    ),
    true
);

while (!it->is_over()) {
    // Получение элементов найденной sc-конструкции
    // при помощи метода sc_iterator::value.

    // Обработка результатов поиска.

    // Перевод итератора на следующий результат поиска.
    it->next();
}

// Обязательно нужно освободить память.
delete it;

```

Приведенный выше код можно переписать при помощи цикла `for` (блок создания итератора не приведен):

```

// ...

for (; !it->is_over(); it->next()) {
    // Получение элементов найденной sc-конструкции
    // при помощи метода sc_iterator::value.

    // Обработка результатов поиска.
}

// Обязательно нужно освободить память.
delete it;

```

Однако два предыдущих варианта можно переписать в более простой форме с использованием специального макроса `sc_for_each` (блок создания итератора не приведен):

```

// ...

sc_for_each (it) {
    // Получение элементов найденной sc-конструкции
    // при помощи метода sc_iterator::value.

    // Обработка результатов поиска.
}

// Память освобождать не нужно, потому что она была освобождена
// при выходе из цикла sc_for_each.

```

Обратите внимание, что в приведенном выше цикле макрос `sc_for_each` берет на себя следующую работу:

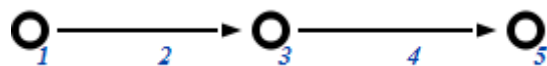


Рис. 4.17: Пример sc-конструкции для ограничения вида `CONSTR_3l2_f_a_a_a_f`

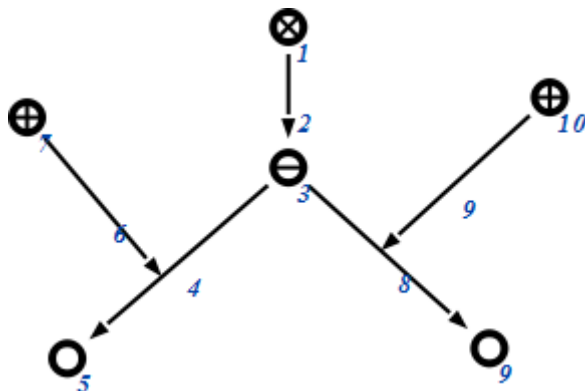


Рис. 4.18: Пример sc-конструкции для ограничения вида `CONSTR_ORD_BIN_CONN1`

- проверка действительности итератора (вызов метода `sc_iterator::is_over`)
- перевод итератора на следующий результат поиска (вызов метода `sc_iterator::next`)
- освобождение памяти, на которую указывает `it`, при выходе из цикла

Цикл `sc_for_each` работает так же, как и циклы `while` и `for`, и поддерживает инструкции `break` и `continue`. А теперь перейдем к рассмотрению других видов ограничений поиска, которые предоставляет библиотека `libsc`.

Аналогично ограничениям вида `CONSTR_3_*_*_*` существуют ограничения вида `CONSTR_5_*_*_*_*_*` для «пятёрок». Ограничения `CONSTR_5_*_*_*_*_*` бывают следующих видов:

- `CONSTR_5_f_a_a_a_a`
- `CONSTR_5_f_a_a_a_f`
- `CONSTR_5_f_a_f_a_a`
- `CONSTR_5_f_a_f_a_f`
- `CONSTR_5_a_a_a_a_f`
- `CONSTR_5_a_a_f_a_a`
- `CONSTR_5_a_a_f_a_f`

Работать с такими ограничениями надо так же, как и с ограничениями вида `CONSTR_3_*_*_*`. Существует еще вид ограничения `CONSTR_3l2_f_a_a_a_f` для поиска sc-конструкций, одна из которых показана на рис. 4.17. На рис. 4.17 элементы специально занумерованы.

Хочу сказать еще про один вид ограничения, который называется `CONSTR_ORD_BIN_CONN1` и используется для поиска связки бинарного отношения с фиксированным компонентом. На рисунке 4.18 показана типовая sc-конструкция для поиска по этому виду ограничения. Элементы с порядковыми номерами 1, 7 и 10 фиксированы, а для всех остальных задается маска типа.

Oneshot-поиск по шаблону

Oneshot-поиск - это поиск при помощи ограничения, при котором находятся не все возможные sc-конструкции, а только одна (первая найденная). При двух последовательных oneshot-поисках не гарантируется, что будет найдена одна и та же sc-конструкция. Для oneshot-поиска применяется специальная функция `search_oneshot`. Следующий код найдет sc-узел *L* и входящую в него из sc-узла *Set1* переменную позитивную sc-дугу (см. рис. 4.16):

```
sc_addr first = 0, second = 0, third = 0;
if (search_oneshot(session,
    sc_constraint_new(
        CONSTR_3_f_a_a,
        Set1,
        SC_ARC|SC_VAR|SC_POS,
        SC_CONST|SC_NODE
    ),
    // Указываем, что экземпляр ограничения необходимо удалить
    true,

    // Количество получаемых значений
    3,

    // Порядковый номер в ограничении 1-го результата
    0,

    // 1-ый элемент найденной конструкции - узел Set1
    &first,

    // Порядковый номер в ограничении 2-го результата
    1,
    // 2-ой элемент найденной конструкции --
    // константная позитивная sc-дуга
    &second,

    // Порядковый номер в ограничении 3-го результата
    2,

    // 3-ий элемент найденной конструкции --
    // константный sc-узел L
    &third
) == RV_OK) {
    // Результат найден
} else {
    // Ничего не найдено
}
```

Синтаксис `search_oneshot` громоздок, поэтому для часто используемых oneshot-поисков созданы специальные функции:

- `search_3_f_cpa_f` — поиск при помощи `CONSTR_3_f_a_f`, когда для второго элемента задана маска типа `SC_A_CONST|SC_POS`

```
// В эту переменную будет занесен
// sc-адрес найденного 2-го элемента
sc_addr e2 = 0;
```

```

if (search_3_f_cpa_f(session,
    // 1-ый элемент известен
    e1,

    // Если что-то будет найдено, то
    // переменная second получит в качестве значения sc-адрес
    // найденного sc-элемента
    &e2,

    // 3-ий элемент известен
    third
) == RV_OK) {
    // Поиск успешен
} else {
    // Поиск неуспешен
}

```

- search_3_f_cpa_cna — поиск при помощи CONSTR_3_f_a_a, когда для второго элемента задана маска типа SC_A_CONST|SC_POS, а для третьего — SC_CONST|SC_NODE

```

// В эту переменную будет занесен sc-адрес найденного 2-го элемента
sc_addr e2 = 0;

// В эту переменную будет занесен sc-адрес найденного 3-го элемента
sc_addr e3 = 0;

if (search_3_f_cpa_cna(session,
    // 1-ый элемент известен
    e1,

    // Если что-то будет найдено в качестве 2-го элемента, то
    // переменная second получит в качестве значения sc-адрес
    // найденного sc-элемента
    &e2,

    // Если что-то будет найдено в качестве 3-го элемента, то
    // переменная third получит в качестве значения sc-адрес
    // найденного sc-элемента
    &e3
) == RV_OK) {
    // Поиск успешен
} else {
    // Поиск неуспешен
}

```

- search_5_f_cpa_a_cpa_f — поиск при помощи CONSTR_5_f_a_a_f, когда для 2-го элемента задана маска типа SC_A_CONST|SC_POS, для 3-го — SC_EMPTY (нет ограничения на тип), для 4-го — SC_A_CONST|SC_POS

```

// В эту переменную будет занесен sc-адрес найденного 2-го элемента
sc_addr e2 = 0;

```

```

// В эту переменную будет занесен sc-адрес найденного 3-го элемента
sc_addr e3 = 0;

// В эту переменную будет занесен sc-адрес найденного 4-го элемента
sc_addr e4 = 0;

if (search_5_f_cpa_a_cpa_f(
    session,
    e1, // sc-адрес известен
    &e2,
    &e3,
    &e4,
    e5 // sc-адрес известен
) == RV_OK) {
    // Поиск успешен
} else {
    // Поиск неуспешен
}

```

- Функция `search_5_f_cpa_sna_cpa_f` — аналогично функции `search_5_f_cpa_a_cpa_f`, только для 3-го элемента задана маска типа `SC_NODE|SC_CONST`
- Функция `search_3l2_f_a_a_a_f` — поиск при помощи `CONSTR_3l2_f_a_a_a_f` (функция используется так же, как и все описанные выше oneshot-функции)

4.4.9 Высокоуровневая работа с основными абстракциями в sc-памяти

В данном разделе будут рассмотрены функционал, который упрощает и делает более читабельными создание и обработку основных более высокоуровневых конструкций, который встречаются в sc-памяти.

Работа с sc-множествами

Под sc-множеством в этом разделе подразумевается sc-узел, т.е. sc-элемент, из которого могут выходить sc-дуги принадлежности/непринадлежности (позитивные, негативные и константные sc-дуги). Если sc-элемент входит в sc-множество, значит от sc-множества к нему проведена позитивная sc-дуга.

В заголовочном файле `sc_set.h` объявлен класс `sc_set`, который предоставляет статические методы обработки sc-множеств. Все статические методы собраны в класс `sc_set`, а не объявлены как функции, потому что в языке C++ класс предоставляет возможности пространства имен, когда не подходит использование явного введения пространства имен с использованием объявления `namespace`. Поэтому класс `sc_set` является классом только с формальной точки зрения.

В классе `sc_set` собраны основные часто используемые операции по обработке sc-множеств и они все в качестве первого аргумента принимают sc-сессию, через которую будет осуществляться работа. Например, если вам надо проверить пусто ли sc-множество, то при помощи метода `sc_set::is_empty` это можно сделать вот так:

```

#include <sc_set.h>

// ...

```

```

if (sc_set::is_empty(session, set1)) {
    // Множество пусто
} else {
    // Множество непусто
}

```

Первым мы рассмотрим набор статических методов, которые занимаются проверкой вхождения sc-элемента в sc-множество. Таких методов существует три вида:

```

/// Если sc-элемент @p el входит в sc-множество @p set,
/// то метод вернет позитивную sc-дугу, связывающую
/// sc-множество с этим sc-элементом. В противном случае
/// метод вернет 0.
static sc_addr is_in(sc_session *s, sc_addr el, sc_addr set);

/// Если sc-элемент @p el входит в sc-множество @p set1
/// или в sc-множество @p set2, то метод вернет позитивную sc-дугу,
/// связывающую sc-множество с этим sc-элементом, которое в списке аргументов
/// стоит раньше. В противном случае метод вернет 0.
static inline sc_addr is_in_any(sc_session *s, sc_addr el, sc_addr set1,
    sc_addr set2);

/// Если sc-элемент @p el входит в sc-множество @p set1
/// и в sc-множество @p set2, то метод вернет true.
/// В противном случае метод вернет false.
static inline bool is_in_all(sc_session *s, sc_addr el, sc_addr set1,
    sc_addr set2);

```

Существуют варианты методов `sc_set::is_in_any` и `sc_set::is_in_all`, которые перегружены для поддержки большего количества sc-множеств (до 9 включительно). Я думаю, что читателю должно быть понятно, как изменяется поведение методов при использовании более двух sc-множеств. Самым часто используемым является метод `sc_set::is_in`.

Следующий набор статических методов предназначен для включения sc-элемента в sc-множество:

```

/// Метод включает sc-элемент @p el в sc-множество @p set1.
/// Позитивная sc-дуга создается в sc-сегменте @p seg.
/// Признак константности этой sc-дуги определяется исходя из константности sc-множества.
///
/// Есть варианты этого метода, которые перегружены для работы с более
/// чем одним sc-множеством (до 9 sc-множеств).
static inline sc_addr include_in(sc_session *s, sc_segment *seg,
    sc_addr el, sc_addr set1);

/// Метод работает аналогично sc_set::include_in, только
/// позитивная sc-дуга создается в sc-сегменте включаемого sc-элемента.
///
/// Есть варианты этого метода, которые перегружены для работы с более
/// чем одним sc-множеством (до 9 sc-множеств).
static inline sc_addr include_in(sc_session *s, sc_addr el,
    sc_addr set1);

/// Метод включает в sc-множество @p set1 sc-элемент @p el1.
/// Позитивная sc-дуга создается в sc-сегменте @p seg.
/// Признак константности этой sc-дуги определяется исходя из константности sc-множества.
///

```

```

/// Есть варианты этого метода, которые перегружены для работы с более
/// чем одним включаемым sc-элементом (до 9 sc-множеств).
static inline void include(sc_session *s, sc_segment *seg, sc_addr set,
    sc_addr el1);

/// Метод работает аналогично sc_set::include, только
/// позитивная sc-дуга создается в sc-сегменте sc-множества.
///
/// Есть варианты этого метода, которые перегружены для работы с более
/// чем одним включаемым sc-элементом (до 9 sc-множеств).
static inline void include(sc_session *s, sc_addr set, sc_addr el1);

```

И последний рассматриваемый набор статических методов предназначен для исключения sc-элемента из sc-множества:

```

/// Исключает sc-элемент @p el из sc-множества @p set1.
/// Этот метод удаляет позитивную sc-дугу, связывающую sc-множество с его элементом.
/// Если удаление sc-дуги произошло, то метод вернет true, иначе false.
///
/// Метод работает с sc-множествами без кратных вхождений элементов.
///
/// Есть варианты этого метода, которые перегружены для исключения
/// с более чем из одного sc-множества (до 9 sc-множеств). В этом случае
/// метод вернет true, если исключение произошло хотя бы из одного
/// sc-множества.
static inline bool exclude_from(sc_session *s, sc_addr el,
    sc_addr set1);

/// Этот метод почти аналогичен методу sc_set::exclude_from, только
/// он учитывает кратные вхождения элемента в sc-множество.
static inline bool exclude_multi_from(sc_session *s, sc_addr el,
    sc_addr set1);

/// Удаляет элементы sc-множества с sc-адресом @p set.
static void erase_from(sc_session *s, sc_addr set);

```

Работа с sc-связками

Класс `sc_tup` (объявлен в файле `sc_tuple.h`) предоставляет статические методы для работы с sc-связками. Самой распространенной операцией над sc-связкой является получение компонент с указанным атрибутом. Для этого можно использовать oneshot-поиск, однако удобнее пользоваться статическим методом `sc_tup::at`. Например, вот так:

```

#include <sc_tuple.h>

// ...

sc_addr first_component = sc_tup::at(
    session,
    tuple, // sc-адрес связки
    N1_ // sc-адрес атрибута 1_
);

// first_component будет равно 0, если

```

```
// компонент с атрибутом 1_ не был найден
if (first_component) {
    // Компонент найден
} else {
    // Компонент не найден
}
```

Также распространенной операцией является добавление в sc-связку компонента с заданным атрибутом. Для этого можно использовать низкоуровневые методы типа `sc_session::gen5_f_a_f_a_f`, но, на мой взгляд, гораздо удобнее использовать статический метод `sc_tup::add`. Вот два способа добавить компонент `component` с атрибутом `1_` в связку `tuple`:

```
sc_tup::add(
    session,
    segment, // все sc-дуги будут сгенерированы в этом sc-сегменте
    tuple, // sc-адрес связки
    N1_, // sc-адрес атрибута
    component // sc-адрес добавляемого компонента
);

// Все sc-дуги будут сгенерированы в sc-сегменте,
// в котором находится sc-элемент tuple
sc_tup::add(
    session,
    tuple, // sc-адрес связки
    N1_, // sc-адрес атрибута
    component // sc-адрес добавляемого компонента
);
```

Работа с sc-отношениями

Аналогично классам `sc_set` и `sc_tup` для работы с sc-отношениями есть класс `sc_rel`, объявленный в файле `sc_relation.h`. Часто используемыми операциями для работы с sc-отношениями являются добавление бинарной ориентированной связки в sc-отношение и поиск бинарной ориентированной связки с указанными компонентами.

Для добавления бинарной ориентированной связки в sc-отношение используется статический метод `sc_rel::add_ord_tuple`:

```
/// Метод создает бинарную ориентированную sc-связку, в которую
/// включает под атрибутом 1_ sc-элемент @p v1, а
/// под атрибутом 2_ - sc-элемент @p v2.
/// Созданная sc-связка включается в sc-отношение @p rel.
/// Созданная sc-связки возвращается в качестве результата.
///
/// Признак константности sc-связки определяется по константности
/// sc-отношения.
///
/// Если необходимы sc-дуги, связывающие sc-связку с ее компонентами,
/// то можно передать в качестве аргументов @p av1 и @p av2 адреса
/// переменных, в которые будут записаны соответствующие
/// sc-дуги. По-умолчанию в качестве этих аргументов передается 0.
static sc_addr add_ord_tuple(sc_session *s, sc_addr rel, sc_addr v1,
    sc_addr v2, sc_addr *av1=0, sc_addr *av2=0);
```

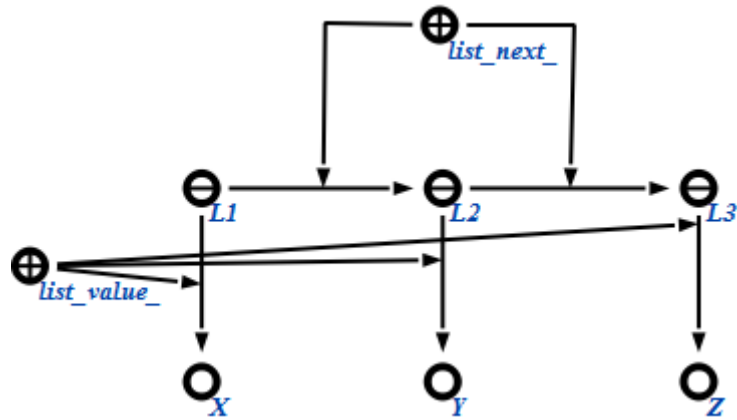



Рис. 4.19: Кодирование списка из элементов X, Y, Z

Для поиск бинарных ориентированных sc-связок в sc-отношении могут быть использованы следующие статические методы класса `sc_rel`:

```

/// В бинарном ориентированном sc-отношении с sc-адресом @p rel
/// находит связку, в которой компонентом с атрибутом 1_ является
/// sc-элемент с sc-адресом @p val1.
///
/// Метод возвращает sc-адрес компонента с атрибутом 2_ в данной связке.
/// Если такой связки не найдено, то метод вернет 0.
///
/// Если необходимо получить sc-адрес найденной связки, то в качестве
/// @p tuple можно передать адрес переменной, куда будет записан искомый
/// sc-адрес.
static inline sc_addr bin_ord_at_2(sc_session *s, sc_addr rel,
    sc_addr val1, sc_addr *tuple=0);

/// Этот метод аналогичен методу sc_rel::bin_ord_at_2, только теперь
/// мы ищем связку, зная sc-адрес компонента с атрибутом 2_, а
/// возвращаем компонент с атрибутом 1_.
static sc_addr bin_ord_at_1(sc_session *s, sc_addr rel, sc_addr val2,
    sc_addr *tuple=0);

```

Двусвязный список в sc-памяти

Библиотека `libsc` предоставляет функционал для представления в sc-памяти двусвязных списков. Для этого используются ключевые узлы `list_next_` и `list_value_`, которые находятся в sc-сегменте `/proc/keynode`. На рис. 4.19 показан пример кодирования списка из константных sc-узлов X, Y, Z.

Элементы `L1`, `L2`, `L3` называются элементами списка и имеют идентификаторы исключительно для наглядности объяснения. Элемент `L1` — это начальный элемент списка, а `L3` — конечный элемент списка. Каждый элемент списка является связкой из двух компонентов:

- под атрибутом `list_value_` в элемент списка входит его значение. Например, на рис. 4.19 значением элемента списка `L1` является sc-узел X
- под атрибутом `list_next_` в элемент списка входит следующий элемент списка. Этот компонент связки может отсутствовать. Например, на рис. 4.19 следующим для элемента

списка $L1$ является элемент списка $L2$, а для элемент списка $L3$ нет следующего элемента, потому он является концевым

По способу кодирования список, представленный на рис. 4.19, является односвязным, но по интерфейсу работы с ним является двусвязным. В заголовочном файле `sc_list.h` объявлен класс `sc_list`, который предоставляет статические методы для работы со списками в `sc`-памяти и STL-совместимые однонаправленные итераторы для перебора их элементов. Все методы класса `sc_list` в качестве своего первого аргумента принимают `sc`-сессию, через которую будет производиться работа с `sc`-памятью.

Первым методом, который мы рассмотрим, будет `sc_list::create`. Он объявлен как:

```
class sc_list
{
public:
    /// Создать элемент списка в sc-сегменте @p seg со значением @p val и
    /// установить его следующим для элемента @p prev
    ///
    /// @note Не используйте @p prev, так как в методе есть баг.
    static sc_addr create(sc_session *s, sc_segment *seg, sc_addr val,
        sc_addr prev=0);

    // ...
};
```

Метод `sc_list::create` нужно использовать для создания первого элемента списка следующим образом:

```
// Создание головы списка.
sc_addr list_head = sc_list::create(
    session,
    segment,
    value // sc-адрес значения головы списка
);

// Список состоит из одного элемента, поэтому
// голова списка является и его концом.
sc_addr list_tail = list_head;
```

Для работы со списками существует набор статических `get`-методов, которые позволяют получить различные характеристики для элемента списка:

```
class sc_list
{
public:
    // ...

    /// Возвращает следующий элемент списка
    /// для элемента списка @p list.
    /// Вернет 0, если элемент списка @p list является хвостовым.
    static sc_addr get_next(sc_session *s, sc_addr list);

    /// Возвращает предыдущего элемент списка
    /// для элемента списка @p list.
    /// Вернет 0, если элемент списка @p list является головным.
    static sc_addr get_prev(sc_session *s, sc_addr list);
```

```

    /// Возвращает значения элемента списка @p list.
    /// Вернет 0, если для элемента списка @p list значение не установлено.
    static sc_addr get_value(sc_session *s, sc_addr list);

    // ...
};

```

Так же в классе `sc_list` существует набор статических `set`-методов, которые позволяют установить различные характеристики для элемента списка:

```

class sc_list
{
public:
    // ...

    /// Установить следующим элементом для элемента списка @p list1
    /// sc-элемент @p list2.
    /// Необходимые sc-элементы будут созданы в sc-сегменте @p seg.
    static sc_retval set_next(s2c_session *s, sc_segment *seg, sc_addr list1,
        sc_addr list2);

    /// Аналогично #sc_list::set_next, только необходимые sc-элементы
    /// будут созданы в sc-сегменте, в котором находится sc-элемент с
    /// sc-адресом @p l1.
    static inline sc_retval set_next(sc_session *s, sc_addr l1, sc_addr l2);

    /// Установить значением для элемента списка @p list1
    /// sc-элемент @p list2.
    /// Необходимые sc-элементы будут созданы в sc-сегменте @p seg.
    static sc_retval set_value(sc_session *s, sc_segment *seg, sc_addr list,
        sc_addr val);

    /// Аналогично #sc_list::set_value, только необходимые sc-элементы
    /// будут созданы в sc-сегменте, в котором находится sc-элемент @p l1.
    static inline sc_retval set_value(sc_session *s, sc_addr l, sc_addr v);

    // ...
};

```

Самый простой способ перебора элементов таких списков от головы к хвосту можно организовать следующим образом:

```

// Текущим обрабатываемым элементом
// является голова списка.
sc_addr list_current = list_head;
while (list_current) {
    // Получаем значения текущего элемента списка.
    sc_addr value = sc_list::get_value(session, list_current);

    //
    // Производим обработку значения.
    //

    // Получаем следующий элемент списка для текущего.
}

```

```

// Если текущим является хвостовой элемент, то
// sc_list::get_next вернет 0.
list_current = sc_list::get_next(session, list_current);
}

```

Однако при помощи однонаправленных итераторов такой перебор элементов списка можно организовать более лаконично:

```

// Создаем итератор, указывающий на голову списка.
sc_list::iterator list_it(session, list_head);

// Создаем итератор, который покажет, что
// достигнут конец списка.
sc_list::iterator list_it_end;

for (; list_it != list_it_end; ++list_it) {
    // Получаем значения текущего элемента списка.
    sc_addr value = *list_it;

    //
    // Производим обработку значения.
    //
}

```

Кроме `sc_list::iterator` есть еще `sc_list::reverse_iterator`, который осуществляет перебор списка в обратном направлении, т.е. оператор `++` переводит итератор с текущего элемента списка на предыдущий. Для определения начала списка все равно необходимо использовать объект класса `sc_list::reverse_iterator`, созданный без аргументов.

И напоследок, если список вам уже не нужен, то его можно удалить при помощи метода `sc_list::erase`:

```

class sc_list
{
public:
    // ...

    /// Удалит элемент списка @p list и все следующие за ним элементы списка.
    /// Удаляет только элементы списка, а не их значения.
    ///
    /// Как правило в качестве @p list надо передавать головной элемент списка.
    static void erase(sc_session *s, sc_addr list);
};

```

4.4.10 Вывод на консоль sc-конструкций

Функции и классы для печати на консоль sc-конструкций объявлены в файле `sc_print_utils.h`:

```

/// Выводит на консоль содержимое @p cont.
sc_retval print_content(const Content &cont);

/// Выводит на консоль sc-элемент @p el и все выходящие из него sc-дуги,
/// работая с sc-памятью при помощи @p session.
sc_retval print_el(sc_session *session, sc_addr el);

```

```

/// Аналогично предыдущей функции, но для работы с sc-памятью
/// используется системная sc-сессия.
void print_el(sc_addr el);

/// Выводит на консоль sc-элемент @p el, все выходящие из него sc-дуги и их атрибуты,
/// работая с sc-памятью при помощи @p session.
void print_struct(sc_session *session, sc_addr el);

/// Аналогично предыдущей функции, но для работы с sc-памятью
/// используется системная sc-сессия.
void print_struct(sc_addr el);

```

4.5 Создание программы wave_find_path с использованием sc-памяти

В этом разделе мы будем создавать программу поиска минимального пути с использованием библиотеки libsc. По примененному алгоритму поиска минимального пути и своей структуре программа из этого раздела будет похожа на программу из раздела 1.4.3. Самым важным отличием будет то, что обработка информации происходит в sc-памяти.

Начнем мы с подготовки sc-памяти для работы алгоритма.

4.5.1 Инициализация и подготовка sc-памяти

Создадим набросок функции main для начала работы с sc-памятью:

```

// Необходимые заголовочные файлы
#include <libsc.h>
#include <pm_keynodes.h>

// ...

int main(int argc, char **argv)
{
    // 1. Инициализация libsc.
    sc_session *system = libsc_init();

    // Теперь мы получили пустую sc-память.

    // 2. Создание sc-сегмента /proc/keynode и
    // системных ключевых узлов в нем (например, 1_, 2_ и т.д.)
    // Эту функцию нужно вызывать обязательно.
    pm_keynodes_init(system);

    // ...

    // 7. Закроем пользовательскую сессию.
    session->close();

    // 8. Деинициализируем libsc.
    libsc_deinit();
}

```

```
    return 0;
}
```

Я думаю, что комментарии к приведенному выше листингу излишне. Если же у вас возникли трудности, то обратитесь к разделу 4.4.3, в котором были описаны все использованные функции.

Мы получили sc-память с системными ключевыми узлами, однако для работы алгоритма необходимы еще и такие ключевые узлы, как *неориентированный граф*, *вершина_*, *ребро_* и др., т.е. ключевые узлы базы знаний по теории графов (см. раздел 2.5.1). Однако, по данному пункту есть два отличия от раздела 2.5.1.

Во-первых, реализация sc-памяти имеет сегментную организацию, поэтому необходимо выбрать сегмент, в котором будут храниться ключевые узлы базы знаний по теории графов. Для этого мы будем использовать сегмент `/graphs_theory/keynode`.

Во-вторых, реализация sc-памяти работает только с кодировкой CP1251, а исходники примера подготовлены в кодировке UTF-8, поэтому при решении нашей задачи не очень удобно использовать русские идентификаторы ключевых узлов. Таким образом, мы будем использовать в исходном тексте английские идентификаторы ключевых узлов, например:

- *вершина_* \Leftrightarrow *vertex_*
- *связка_* \Leftrightarrow *connective_*
- *ребро_* \Leftrightarrow *edge_*
- *дуга_* \Leftrightarrow *arc_*
- *неориентированный граф* \Leftrightarrow *undirected graph*
- *ориентированный граф* \Leftrightarrow *directed graph*
- *простая цепь** \Leftrightarrow *simple trail**

Теперь можно переходить к организации создания и работы с ключевыми узлами. Для этого нам понадобится функция создания сегмента по полному URI, которая определена в файле `segment_utils.h` следующим образом:

```
/// Функция при помощи sc-сессии @p s создает sc-сегмент по полному URI @p uri
/// с созданием всех промежуточных директорий.
///
/// Если создание прошло успешно, то sc-сегмент, иначе 0.
sc_segment *create_segment_full_path(sc_session *s, const sc_string &uri);
```

Напишем код, который обеспечивает работу с ключевыми узлами. В листинге ниже представлено пространство имен `graph_theory`, которое содержит объявление ключевых узлов и функцию для их инициализации:

```
// Для использования create_segment_full_path
#include <segment_utils.h>

// ...

/// Пространство имен ключевых узлов по теории графов.
///
/// Для начала работы необходимо вызвать функцию graph_theory::init,
/// передав в качестве параметра системную сессию.
/// После этого, например, чтобы обратиться к узлу вершина_,
```

```

/// достаточно написать graph_keynodes::vertex_.
namespace graph_theory
{
    /// URI сегмента ключевых узлов.
    const char *segment_uri = "/graph_theory/keynode";

    /// Созданный сегмент ключевых узлов.
    sc_segment *segment;

    /// Массив идентификаторов ключевых узлов.
    const char *idtfs[] = {
        "vertex_", // вершина_
        "connective_", // связка_
        "edge_", // ребро_
        "arc_", // дуга_
        "undirected_graph", // неориентированный граф
        "directed_graph", // ориентированный граф
        "simple_trail*" // простая цепь*
    };

    /// Массив sc-адресов созданных ключевых узлов.
    sc_addr keynodes[sizeof(idtfs) / sizeof(const char*)];

    /// Ссылки на ключевые узлы для более удобной работы.
    /// @{
    const sc_addr &vertex = keynodes[0];
    const sc_addr &connective_ = keynodes[1];
    const sc_addr &edge_ = keynodes[2];
    const sc_addr &arc_ = keynodes[3];
    const sc_addr &undirected_graph = keynodes[4];
    const sc_addr &directed_graph = keynodes[5];
    const sc_addr &simple_trail = keynodes[6];
    /// @}

    /// Производит создание ключевых узлов при помощи сессии @p s
    /// и готовит их к работе.
    void init(sc_session *s)
    {
        // Сперва создадим сегмент «/graph_theory/keynode» для
        // ключевых узлов.
        segment = create_segment_full_path(s, segment_uri);

        // Пробежимся по массиву идентификаторов ключевых узлов idtfs
        // и создадим каждый ключевой узел.
        // Создаваемые узлы будем заносить в массив keynodes.
        for (int i = 0; i < sizeof(keynodes) / sizeof(sc_addr); ++i) {
            keynodes[i] = s->create_el(segment, SC_N_CONST);
            s->set_idtf(keynodes[i], idtfs[i]);
        }
    }
}

```

С учетом написанного кода для работы с ключевыми узлами, продолжим работу над функцией main. Во-первых, до начала работы алгоритма нам необходимо в этой функции инициа-

лизировать ключевые узлы теории графов. Во-вторых, создать пользовательскую сессию для работы алгоритма. В-третьих, так как в рамках только что созданной пользовательской сессии нет открытых сегментов, открыть сегменты ключевых узлов. Окончательный вариант разрабатываемой функции будет следующим:

```
int main(int argc, char **argv)
{
    // 1. Инициализация libsc.
    sc_session *system = libsc_init();

    // Теперь мы получили пустую sc-память.

    // 2. Создание sc-сегмента /proc/keynode и
    // системных ключевых узлов в нем (например, 1_, 2_ и т.д.)
    // Эту функцию нужно вызывать обязательно.
    pm_keynodes_init(system);

    // 3. Инициализируем ключевые узлы базы знаний по теории графов.
    graph_theory::init(system);

    // 4. Создадим пользовательскую сессию для работы алгоритма.
    sc_session *session = libsc_login();

    // 5. Откроем сегменты системных ключевых узлов и ключевых узлов
    // базы знаний по теории графов в рамках пользовательской сессии.
    session->open_segment("/proc/keynode");
    session->open_segment(graph_theory::segment_uri);

    // 6. Запуск алгоритма.
    // ...

    // 7. Закроем пользовательскую сессию.
    session->close();

    // 8. Деинициализируем libsc.
    libsc_deinit();

    return 0;
}
```

Все готово для написания и запуска алгоритма, но сейчас мы будем рассматривать вспомогательные функции для загрузки из файла и вывода на консоль неориентированных графов.

4.5.2 Загрузка неориентированного графа

Объявим функцию загрузки графа следующим образом:

```
/// Генерирует в sc-памяти неориентированный граф.
///
/// @param s сессия для работы с sc-памятью.
/// @param seg сегмент, в котором будет сгенерирован неориентированный граф.
/// @param graph_file входной файл с матрицей смежности неориентированного графа.
///
/// @return сгенерированный неориентированный граф.
```



```
sc_addr load_graph(sc_session *s, sc_segment *seg, const std::string &graph_file);
```

Обратите внимание, что 1-ым параметром функции `load_graph` является `sc-сессия`, при помощи которой будет происходить работа с `sc-памятью`. Все функции из этого раздела будут иметь аналогичный 1-ый параметр. Теперь перейдем к созданию тела функции.

Во-первых, мы должны создать узел графа и включить его во множество *неориентированный граф*:

```
#include <assert.h>

// ...

sc_addr load_graph(sc_session *s, sc_segment *seg, const std::string &graph_file)
{
    assert(s);
    assert(seg);

    // 1. Создадим узел графа.
    sc_addr graph = s->create_el(seg, SC_N_CONST);
    s->gen3_f_a_f(graph_theory::undirected_graph, 0, seg, SC_A_CONST|SC_POS, graph);

    // ...
}
```

Макрос `assert` в дальнейшем будет использоваться часто для проверки параметров функций, поэтому прочтите его назначение на [странице в Wikipedia](#).

Загружать неориентированный граф мы будем из файла, формат которого аналогичен использованному в части 1. Откроем файл с графом и считаем количество вершин:

```
#include <fstream>

// ...

sc_addr load_graph(sc_session *s, sc_segment *seg, const std::string &graph_file)
{
    // ...

    // 2. Откроем файл с входным графом и считаем количество вершин.
    size_t vcount; // количество вершин
    std::ifstream in(graph_file.c_str());
    in >> vcount;

    // ...
}
```

Теперь необходимо считать имена вершин и создать вершины в `sc-памяти`. Так как при работе с матрицей смежности мы будем иметь дело с индексами вершин, то необходим способ для быстрого перехода от индекса вершины к ее `sc-адресу`. Для этого мы воспользуемся `std::vector`. Для самых распространенных STL-контейнеров, которые хранят данные типа `sc_addr`, в файле `sc_types.h` объявлены `typedef`'ы. Нас будет интересовать `addr_vector` - `typedef` для `std::vector<sc_addr>`. Итак, код создания вершин будет следующим:

```
sc_addr load_graph(sc_session *s, sc_segment *seg, const std::string &graph_file)
{
    // ...
}
```

```

// 3. Считаем имена вершин и создадим каждую из вершин.
std::string name;
// Этот массив позволит перейти от индекса вершины к ее sc-адресу.
addr_vector vertexes;
for (size_t i = 0; i < vcount; ++i) {
    in >> name;

    // Создадим вершину, установим идентификатор, добавим ее в граф.
    sc_addr vertex = s->create_el(seg, SC_N_CONST);
    s->set_idtf(vertex, name);
    s->gen5_f_a_f_a_f(graph, 0, seg, SC_A_CONST|SC_POS, vertex, 0, seg,
        SC_A_CONST|SC_POS, graph_theory::vertex_);

    vertexes.push_back(vertex);
}

// ...
}

```

Пришло время считывать матрицу смежности. Матрица смежности обладает симметрией, поэтому нам необходимо создавать ребра только для одной из ее половин и главной диагонали. Будем создавать ребра при считывании нижней половины и главной диагонали (номер столбца меньше либо равен номеру строки), а при считывании верхней половины не выполнять никаких действий:

```

sc_addr load_graph(sc_session *s, sc_segment *seg, const std::string &graph_file)
{
    // ...

    // 4. Считаем нижнюю половину матрицы смежности и создадим ребра графа.
    for (size_t i = 0; i < vcount; ++i) {
        for (size_t j = 0; j < vcount; ++j) {
            unsigned int k;
            in >> k;

            if (j <= i && k) {
                // Создадим ребро.
                sc_addr edge = s->create_el(seg, SC_N_CONST);
                s->gen3_f_a_f(edge, 0, seg, SC_A_CONST|SC_POS, vertexes[i]);
                s->gen3_f_a_f(edge, 0, seg, SC_A_CONST|SC_POS, vertexes[j]);

                // Добавим ребро в граф.
                s->gen5_f_a_f_a_f(graph, 0, seg, SC_A_CONST|SC_POS, edge, 0, seg,
                    SC_A_CONST|SC_POS, graph_theory::edge_);
            }
        }
    }

    return graph;
}

```

Неориентированный граф создан в sc-памяти, значит можно вернуть его sc-адрес в качестве результата выполнения функции:

```

sc_addr load_graph(sc_session *s, sc_segment *seg, const std::string &graph_file)
{
    // ...

    return graph;
}

```

Теперь напомним функцию, которая будет выводить на консоль неориентированный граф, загруженный при помощи `load_graph`.

4.5.3 Вывод на консоль неориентированного графа

Для вывода неориентированного графа на консоль нам надо будет предпринять следующие шаги:

1. вывести на консоль каждое ребро графа вместе с инцидентными вершинами и запомнить выведенные вершины;
2. вывести на консоль те вершины, которые еще не были выведены.

Приступим к написанию функции `print_graph`, объявление которой выглядит следующим образом:

```

/// Выводит на консоль неориентированный граф @p graph.
void print_graph(sc_session *s, sc_addr graph);

```

Для начала определимся со структурой данных, в которой будем хранить уже выведенные на консоль вершины. Можно было бы, сохраняя чистоту подхода, использовать для хранения такой информации множество в `sc`-памяти. Однако, на мой взгляд, в данном случае будет достаточно использовать STL-контейнер `std::set`, который сохранит `sc`-адреса вершин. Нам будет интересно использовать `addr_set` из `sc_types.h` - `typedef` для `std::set<sc_addr>`. Таким образом, начало функции будет следующим:

```

void print_graph(sc_session *s, sc_addr graph)
{
    assert(s);
    assert(graph);

    addr_set printed_vertexes; // множество выведенных вершин

    // ...
}

```

Следующим шагом будет вывод на консоль ребер и инцидентных им вершин. Так как граф хранится в `sc`-памяти, то для этого нам необходимо использовать ее базовые поисковые механизмы (раздел 4.4.8).

Ребра входят в граф с атрибутом `ребро_`, поэтому для их перебора мы будем использовать одно из ограничений поиска пятиэлементных `sc`-конструкций (см. рис 4.11). В нашем случае фиксированными являются 1-ый (узел графа, т.е. параметр `graph`) и 5-ый (атрибут `ребро_`) элементы пятиэлементной `sc`-конструкции. Следовательно, мы будем использовать ограничение вида `CONSTR_5_f_a_a_a_f`. Этот вид ограничения является одним из самых часто используемых, когда 2-ой и 4-ый элементы являются константными позитивными `sc`-дугами, т.е. подходят под маску типа `SC_A_CONST|SC_POS`. В нашем случае это так и есть, а для 3-го элемента будем использовать маску типа 0, т.е. он может быть любого типа. Ниже приведен код создания итератора по ребрам:

```

void print_graph(sc_session *s, sc_addr graph)
{
    // ...

    // 1. Создание итератора по ребрам.
    sc_iterator* edges_it = s->create_iterator(
        sc_constraint_new(
            CONSTR_5_f_a_a_a_f,
            graph,
            SC_A_CONST|SC_POS,
            0,
            SC_A_CONST|SC_POS,
            graph_theory::edge_
        ), true);

    // ...
}

```

Для цикла по ребрам логично использовать макрос `sc_for_each`:

```

#include <iostream> // для std::cout и std::endl

// ...

void print_graph(sc_session *s, sc_addr graph)
{
    // ...

    // 2. Вывод ребер.
    sc_for_each (edges_it) {
        sc_addr edge = edges_it->value(2);

        // Получим вершины, инцидентные ребру edge.
        sc_addr v1 = 0, v2 = 0;
        get_edge_vertexes(s, edge, v1, v2);

        // Выведем ребро вместе с инцидентными вершинами.
        std::cout << s->get_idtf(v1) << "└─┬─" << s->get_idtf(v2) << std::endl;

        // Запомним вершины, как выведенные.
        printed_vertexes.insert(v1);
        printed_vertexes.insert(v2);
    }

    // ...
}

```

В приведенном выше листинге используется еще ненаписанная функция `get_edge_vertexes`, которая возвращает вершины, инцидентные переданному ребру. Объявим эту функцию следующим образом (**обратите внимание**, что функция возвращает результаты работы через параметры `v1` и `v2`, которые имеют ссылочный тип):

```

/// Возвращает в @p v1 и @p v2 вершины,
/// инцидентные ребру @p edge.

```

```
void get_edge_vertexes(sc_session *s, sc_addr edge, sc_addr &v1, sc_addr &v2);
```

Так как ребро неориентированного графа является двумощным множеством, то для перебора инцидентных вершин логично использовать одно из ограничений для поиска трехэлементных sc-конструкций (см. рис 4.10). Фиксированным в данном случае является 1-ый элемент (ребро графа, т.е. параметр `edge`), значит мы будем использовать для поиска ограничение вида `CONSTR_3_f_a_a`. Теперь можно написать следующее тело функции (обратите внимание, что в коде не используется цикл `sc_for_each`, поэтому после работы надо освободить итератор):

```
void get_edge_vertexes(sc_session *s, sc_addr edge, sc_addr &v1, sc_addr &v2)
{
    assert(s);
    assert(edge);

    sc_iterator *it = s->create_iterator(
        sc_constraint_new(
            CONSTR_3_f_a_a,
            edge,
            SC_A_CONST|SC_POS,
            0
        ), true);

    v1 = it->value(2);
    it->next();
    v2 = it->value(2);

    delete it;
}
```

Вернемся к `print_graph`. Последним этапом этой функции является вывод тех вершин, которые еще не были выведены на консоль, т.е. не входят во множество `printed_vertexes`. Для этого необходимо организовать цикл по всем вершинам. Вершины входят в граф с атрибутом *вершина*_, поэтому нам подходит уже знакомое ограничение вида `CONSTR_5_f_a_a_a_f`. Ограничению в качестве атрибута мы передадим не `graph_theory::edge_`, как в случае с перебором ребер, а `graph_theory::vertex_`. Разрабатываемый цикл будет следующим:

```
void print_graph(sc_session *s, sc_addr graph)
{
    // ...

    // 3. Вывод вершин, которые не имеют инцидентных ребер.
    sc_iterator *vertexes_it = s->create_iterator(
        sc_constraint_new(
            CONSTR_5_f_a_a_a_f,
            graph,
            SC_A_CONST|SC_POS,
            0,
            SC_A_CONST|SC_POS,
            graph_theory::vertex_
        ), true);
    sc_for_each (vertexes_it) {
        sc_addr vertex = vertexes_it->value(2);

        // Проверим, входит ли вершина в множество printed_vertexes
        if(printed_vertexes.find(vertex) == printed_vertexes.end())
```

```

        std::cout << s->get_idtf(vertex) << '\n';
    }
}

```

Мы рассмотрели две простые функции для работы с неориентированными графами, представленными в *sc*-памяти. Научились генерировать граф в *sc*-памяти, перебирать вершины и ребра. Пришло время перейти к реализации алгоритма поиска минимального пути.

4.5.4 Запуск тестов алгоритма

Мы будем разрабатывать функцию `run_testcase`, которая запустит алгоритм с конкретными входными данными. Варианты входных данных будем брать из раздела 2.5.2. Эта функция очень похожа на тезку из раздела 1.4.3.

Объявим функцию `run_testcase` вот так:

```

/// Подготавливает запуск и запускает тестовый пример для алгоритма поиска минимального пути.
///
/// @param number порядковый номер тестового примера.
/// @param graph_file путь к файлу с тестовым графом для функции #load_graph.
/// @param beg_name имя начальной вершина для поиска минимального пути.
/// @param end_name имя конечной вершина для поиска минимального пути.
///
/// @see load_graph
/// @see find_min_path
void run_testcase(int number, const char *graph_file, const char *beg_name,
                  const char *end_name);

```

Тогда в функции `main` ее вызовем следующим образом с тестовыми файлами из раздела 1.4.2:

```

int main(int argc, char **argv)
{
    // ...

    // 6. Запуск тестов алгоритма.
    run_testcase(1, "graph1.txt", "A", "C");
    run_testcase(2, "graph2.txt", "A", "F");
    run_testcase(3, "graph3.txt", "A", "K");
    run_testcase(4, "graph4.txt", "V5", "V11");
    run_testcase(5, "graph5.txt", "V1", "V9");

    // ...
}

```

Саму функцию начнем с того, что создадим сегмент для работы алгоритма. В него будем загружать граф из файла, в нем алгоритм будет создавать временные структуры и результат. Для этого воспользуемся функцией `create_unique_segment`, которая объявлена в файле `segment_utils.h` как показано ниже:

```

/// Функция для создания уникального сегмента.
///
/// @param s sc-сессия, при помощи которой будет происходить работа с sc-памятью.
/// @param base базовый uri, на основе которого будет происходить создание сегмента.
/// Например, "/tmp/myseg".
/// @return sc-сегмент, если создание прошло успешно.
/// @return 0, если возникла ошибка.

```

```
sc_segment *create_unique_segment(sc_session *s, const sc_string &base);
```

Создание рабочего сегмента показано в листинге ниже:

```
void run_testcase(int number, const char *graph_file, const char *beg_name,
const char *end_name)
{
    std::cout << "[Testcase_] << number << "]\n";

    // 1. Для работы создаем рабочий сегмент /tmp/wave_find_path.
    sc_segment *tmp_seg = create_unique_segment(s, "/tmp/wave_find_path");

    // ...
}
```

Теперь загрузим в рабочий сегмент при помощи функции load_graph (из раздела 4.5.2) граф:

```
void run_testcase(int number, const char *graph_file, const char *beg_name,
const char *end_name)
{
    // ...

    // 2. Загрузим тестовый граф в sc-памяти и распечатаем его.
    sc_addr graph = load_graph(s, tmp_seg, graph_file);

    std::cout << "Graph:_" << std::endl;
    print_graph(s, graph);

    // ...
}
```

Найдем вершины по идентификаторам при помощи метода sc_session::find_by_idtf:

```
void run_testcase(int number, const char *graph_file, const char *beg_name,
const char *end_name)
{
    // ...

    // 3. Найдем вершины по именам в sc-памяти.
    sc_addr beg = s->find_by_idtf(beg_name, tmp_seg);
    assert(beg);

    sc_addr end = s->find_by_idtf(end_name, tmp_seg);
    assert(end);

    std::cout << "Find_minimal_path_from_" << beg_name << " to_"
        << end_name << " " << std::endl;

    // ...
}
```

Всё подготовлено для запуска алгоритма. Он запускается в функции find_min_path, которая будет описана ниже в разделе 4.5.5, а после этого выведем маршрут при помощи функции print_route (см. раздел 4.5.6):

```
void run_testcase(int number, const char *graph_file, const char *beg_name,
```

```

    const char *end_name)
{
    // ...

    // 4. Найдем минимальный пути между начальной и конечной вершинами,
    // распечатаем его на консоль.
    sc_addr result = find_min_path(s, tmp_seg, graph, beg, end);

    std::cout << "Path";

    if (result) {
        std::cout << ":" << std::endl;
        print_route(s, result);
    } else {
        std::cout << "isn't exist." << std::endl;
    }

    std::cout << std::endl;

    // ...
}

```

Тестовый пример выполнен. Необходимо удалить созданные sc-конструкции. Это можно сделать, удалив целиком весь рабочий сегмент при помощи метода `sc_session::unlink`, который удаляет сегмент по полному URI:

```

void run_testcase(int number, const char *graph_file, const char *beg_name,
    const char *end_name)
{
    // ...

    // 5. Удалим рабочий сегмент.
    s->unlink(tmp_seg->get_full_uri());
}

```

Функция запуска тестового примера закончена, а значит перейдем к реализации «сердца» программы — функции `find_min_path`.

4.5.5 Реализация алгоритма

Объявление функции будет выглядеть вот так:

```

/// Находит один из минимальных путей в графе @p graph
/// от вершины @p beg_vertex до вершины @p end_vertex.
///
/// @param s сессия для работы с sc-памятью.
/// @param seg сегмент, в котором происходит работа алгоритма.
/// @param graph неориентированный граф, в котором будет находится минимальный путь.
/// @param beg_vertex начальная вершина пути.
/// @param end_vertex конечная вершина пути.
///
/// @return связка отношения простая цель* или 0, если минимальный путь не найден.
sc_addr find_min_path(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr beg_vertex, sc_addr end_vertex);

```


Каждый кусок кода в дальнейшем мы будем связывать с шагами работы алгоритма в sc-памяти из раздела 3.2.2. Названия переменных на рисунках в этом разделе будет соответствовать локальным переменным функции `find_min_path`. Обратите внимание, что вся обработка информацией `find_min_path` будет происходить в sc-памяти. Это соответствует требованию, указанному в задании. В вашей программе должно соблюдаться это же требование — обработка всей информации происходит в sc-памяти.

В функции `find_min_path` во всю будут использоваться классы `sc_set`, `sc_tup`, `sc_rel`, которые описаны в разделе 4.4.9.

Подготовка к созданию списка волн

Начнем мы с формирования множества непроверенных вершин, что соответствует шагу на странице 46. Для этого нам необходимо создать множество непроверенных вершин и, перебрав вершины входного графа, добавить все, кроме начальной, в созданное множество:

```
sc_addr find_min_path(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr beg_vertex, sc_addr end_vertex)
{
    // 1. Добавим все вершины графа (кроме начальной вершины пути)
    // в множество непроверенных вершин.

    // множество непроверенных вершин
    sc_addr not_checked_vertexes = s->create_el(seg, SC_N_CONST);

    // Перебор всех вершин.
    sc_iterator *it = s->create_iterator(
        sc_constraint_new(
            CONSTR_5_f_a_a_a_f,
            graph,
            SC_A_CONST|SC_POS,
            0,
            SC_A_CONST|SC_POS,
            graph_theory::vertex_
        ), true);
    sc_for_each(it) {
        sc_addr vertex = it->value(2);

        // Не добавляем вершину начала пути в множество непросмотренных вершин.
        if (vertex != beg_vertex)
            sc_set::include_in(s, it->value(2), not_checked_vertexes);
    }

    // ...
}
```

Теперь, согласно шагу на странице 47, необходимо создать первую волну списка волн. Для работы со списком волн мы будем использовать класс `sc_list` (см. раздел 4.4.9). Таким образом, код для создания начальной волны будет следующим:

```
sc_addr find_min_path(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr beg_vertex, sc_addr end_vertex)
{
    // ...
}
```

```

// 2. Создадим начальную волну и добавим в нее начальную вершину пути.
// Включим в список волн.
sc_addr new_wave = s->create_el(seg, SC_N_CONST);
sc_set::include_in(s, beg_vertex, new_wave);

// Создадим начало списка волн.
sc_addr waves_list_head = sc_list::create(s, seg, new_wave);
sc_addr waves_list_tail = waves_list_head;

// ...
}

```

Перейдем к написанию кода для формирования оставшегося списка волн.

Формирование списка волн

Код в этом разделе будет соответствовать шагам алгоритма на страницах 48, 49, 50. Учитывая то, что в переменной `new_wave` сейчас находится первая волна, а на каждой итерации она в качестве значения будет получать новую созданную волну, можно составить основу цикла формирования волн. В конце каждой итерации нужно проверять, не входит ли в новую волну конечная вершина пути:

```

sc_addr find_min_path(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr beg_vertex, sc_addr end_vertex)
{
    // ...

    // 3. Сформируем список волн.
    do {

        // ...

        // Если в новой волне есть конечная вершина, то перейдем в начало цикла.
    } while (!sc_set::is_in(s, end_vertex, new_wave));

    // ...
}

```

Теперь обратим внимание на тело цикла. Начнем мы его с создания новой волны на основе текущей, записанной в переменную `new_wave`. Для этого будем использовать функцию создания новой волны `create_wave`:

```

/// Создает следующую волну из непроверенных вершин,
/// смежных с вершинами из волны @p wave неориентированного графа @p graph.
///
/// @param s сессия для работы с sc-памятью.
/// @param seg сегмент, в котором происходит генерация новой волны.
/// @param graph обрабатываемый неориентированный граф.
/// @param wave текущая волна.
/// @param not_checked_vertexes множество непроверенных вершин.
///
/// @return созданная новая волна.
sc_addr create_wave(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr wave, sc_addr not_checked_vertexes);

```

Разберем функцию `create_wave` чуть позже, а сейчас посмотрим как она применяется в цикле формирования списка волн:

```
sc_addr find_min_path(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr beg_vertex, sc_addr end_vertex)
{
    // ...

    // 3. Сформируем список волн.
    do {
        // Создадим новую волну на основе предыдущей
        new_wave = create_wave(s, seg, graph, new_wave, not_checked_vertexes);

        // ...
    } while (!sc_set::is_in(s, end_vertex, new_wave));

    // ...
}
```

Оставим на время цикл создания списка волн и рассмотрим функцию `create_wave`. Для создания новой волны нам необходимо перебрать вершины из предыдущей волны `wave`:

```
sc_addr create_wave(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr wave, sc_addr not_checked_vertexes)
{
    // Создадим узел новой волны.
    sc_addr new_wave = s->create_el(seg, SC_N_CONST);

    // 1. Перебор всех вершин из волны wave.
    sc_iterator *it_vertex = s->create_iterator(
        sc_constraint_new(
            CONSTR_3_f_a_a,
            wave,
            SC_A_CONST|SC_POS,
            0
        ), true);
    sc_for_each (it_vertex) {
        sc_addr vertex = it_vertex->value(2);

        // ...
    }

    return new_wave;
}
```

Для каждой вершины `vertex` переберем смежные с ней вершины. Для этого воспользуемся ограничением вида `CONSTR_3l2_f_a_a_a_f` (см. рис. 4.17). 1-ым элементом будет `graph`, 5-ым — `vertex`, все остальные — нефиксированные.

```
sc_addr create_wave(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr wave, sc_addr not_checked_vertexes)
{
    // ...

    sc_for_each (it_vertex) {
```

```

sc_addr vertex = it_vertex->value(2);

// 2. Перебор всех ребер, которые инцидентны vertex.
sc_iterator *it_edge = s->create_iterator(
    sc_constraint_new(
        CONSTR_312_f_a_a_a_f,
        graph,
        SC_A_CONST|SC_POS,
        0,
        SC_A_CONST|SC_POS,
        vertex
    ), true);
sc_for_each (it_edge) {
    sc_addr edge = it_edge->value(2); // ребро, инцидентное vertex
    sc_addr other_vertex = get_other_vertex_incidence_edge(
        s, edge, vertex); // вершина, смежная vertex и инцидентная edge

    // ...
}
}

return new_wave;
}

```

На рис. 4.20 показана работа поиска по ограничению CONSTR_312_f_a_a_a_f. Фиксированными элементами являются граф G и вершина C . Фиксированные элементы отмечены **зеленым** цветом, а найденные нефиксированные — **синим**. Обратите внимание, что в этом поиске не учитывается то, что ребро должно входить в граф с атрибутом *ребро_*.

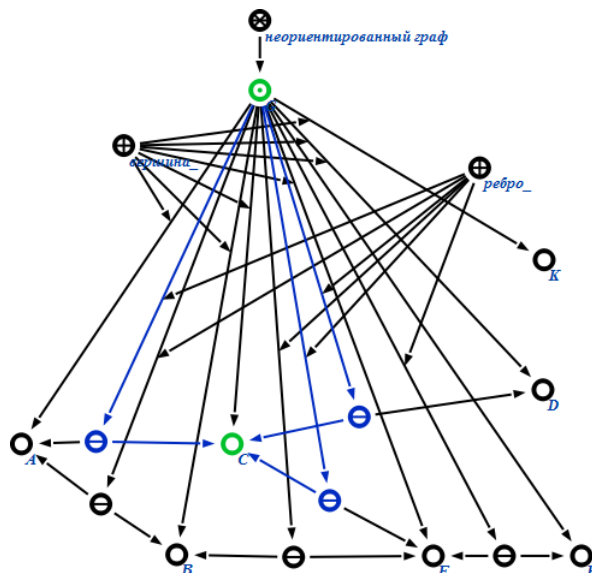


Рис. 4.20: Поиск в указанном неориентированном графе инцидентных указанной вершине ребер

Теперь в новую волну *new_wave* необходимо добавить ту вершину *other_vertex*, которая входит в *not_checked_vertexes*, и исключить ее из *not_checked_vertexes*. Эти действия продемонстрированы в листинге ниже:

```

sc_addr create_wave(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr wave, sc_addr not_checked_vertexes)

```

```

{
    // ...

    sc_for_each (it_vertex) {
        // ...

        sc_for_each (it_edge) {
            sc_addr edge = it_edge->value(2); // ребро, инцидентное vertex
            sc_addr other_vertex = get_other_vertex_incidence_edge(
                s, edge, vertex); // вершина, смежная vertex и инцидентная edge

            // Исключим вершину other_vertex из множества непроверенных вершин и
            // включим в создаваемую волну.
            if (sc_set::exclude_from(s, other_vertex, not_checked_vertexes))
                sc_set::include_in(s, other_vertex, new_wave);
        }
    }

    return new_wave;
}

```

На этом написание функции `create_wave` закончено, поэтому вернемся к месту в цикле формирования списка волн, на котором мы остановились. Новая волна создана, но она может быть пустой. Это значит, что пути между вершинами не существует. Если волна пустая, то необходимо выйти из функции `find_min_path`:

```

sc_addr find_min_path(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr beg_vertex, sc_addr end_vertex)
{
    // ...

    // 3. Сформируем список волн.
    do {
        // ...

        // Если новая волна пуста, то значит между вершинами не существует пути.
        if (sc_set::is_empty(s, new_wave)) {
            // Очищаем память и завершаем алгоритм.
            erase_waves_list(s, waves_list_head);
            s->erase_el(new_wave);
            s->erase_el(not_checked_vertexes);
            return 0;
        }

        // ...
    } while (!sc_set::is_in(s, end_vertex, new_wave));

    // ...
}

```

Обратите внимание, что при выходе из функции удалили все созданные `sc`-конструкции. Функция `erase_waves_list` имеет следующее объявление (эту функцию вы рассмотрите самостоятельно):

```

/// Удаляет все волны из списка и список волн, начиная

```

```

/// с элемента списка @p waves_list_head.
void erase_waves_list(sc_session *s, sc_addr waves_list_head);

```

Новая волна создана и непуста, значит пришла пора добавить ее в конец списка волн:

```

sc_addr find_min_path(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr beg_vertex, sc_addr end_vertex)
{
    // ...

    // 3. Сформируем список волн.
    do {
        // ...

        // Добавим новую волну в конец списка.
        sc_addr waves_list_curr = sc_list::create(s, seg, new_wave);
        sc_list::set_next(s, waves_list_tail, waves_list_curr);

        waves_list_tail = waves_list_curr;

        // Если в новой волне есть конечная вершина, то перейдем в начало цикла.
    } while (!sc_set::is_in(s, end_vertex, new_wave));

    // ...
}

```

После цикла не забудем удалить уже ненужные sc-конструкции:

```

sc_addr find_min_path(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr beg_vertex, sc_addr end_vertex)
{
    // ...

    // Подчистим память.
    s->erase_el(not_checked_vertexes);

    // ...
}

```

Итак... Мы закончили цикл формирования списка волн и переходим к построению маршрута.

Подготовка к формированию структуры маршрута

Прежде, чем формировать структуру маршрута, нам необходимо создать связку отношения *простая цепь* *. Это соответствует шагу алгоритма на странице 52.

Сгенерируем в sc-памяти связку отношения *простая цепь* *:

```

sc_addr find_min_path(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr beg_vertex, sc_addr end_vertex)
{
    // ...

    // 4. Создадим связку отношения простая цепь *.
    sc_addr route = s->create_el(seg, SC_N_CONST); // связка отношения
    sc_set::include_in(s, route, graph_theory::simple_trail);
}

```

```

    // ...
}

```

Обратите внимание, что в листинге выше для включения связки в отношение используется метод `sc_set::include_in`, потому что необходима генерация `sc`-дуги в сегменте связки `route`, а не ключевого узла `graph_theory::simple_trail`.

Теперь создадим компоненты связки `route`:

```

sc_addr find_min_path(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr beg_vertex, sc_addr end_vertex)
{
    // ...

    // 5. Создадим компоненты связки route.

    // Ориентированный граф структуры маршрута
    sc_addr route_struct = s->create_el(seg, SC_N_CONST);
    sc_set::include_in(s, route_struct, graph_theory::directed_graph);

    // Бинарное отношение посещения
    sc_addr route_visit = s->create_el(seg, SC_N_CONST);

    // ...
}

```

Созданные компоненты включим в связку `route`:

```

sc_addr find_min_path(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr beg_vertex, sc_addr end_vertex)
{
    // ...

    // 6. Добавим все компоненты в связку route.
    sc_tup::add(s, route, N1_, route_struct);
    sc_tup::add(s, route, N2_, graph);
    sc_tup::add(s, route, N3_, route_visit);

    // ...
}

```

В соответствии со следующим шагом алгоритма (см. стр. ??), добавим посещение конечной вершины пути в структуру маршрута:

```

sc_addr find_min_path(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr beg_vertex, sc_addr end_vertex)
{
    // ...

    // 7. Добавим в простую цепь посещение конечной вершины.
    sc_addr end_vertex_visit = add_vertex_visit_to_route(s, route, end_vertex);

    // ...
}

```

Функцию `add_vertex_visit_to_route` объявим вот так:

```

/// Создает в структуре маршрута @p route посещение вершины @p vertex
/// и возвращает это посещение.
///
/// @note Все элементы генерируются в сегменте связки @p route.
sc_addr add_vertex_visit_to_route(sc_session *s, sc_addr route, sc_addr vertex);

```

Прежде, чем рассматривать код этой функции, напомним две простые вспомогательные функции для работы с маршрутом:

```

/// Возвращает структуру маршрута для связки отношения маршрут*.
/// @note Структура маршрута - это компонент с атрибутом 1_.
inline sc_addr get_route_struct(sc_session *s, sc_addr route)
{
    return sc_tup::at(s, route, N1_);
}

/// Возвращает отношение посещения для связки отношения маршрут*.
/// @note Отношение посещения - это компонент с атрибутом 3_.
inline sc_addr get_route_visit(sc_session *s, sc_addr route)
{
    return sc_tup::at(s, route, N3_);
}

```

Теперь перейдем к `add_vertex_visit_to_route`. В начале каждой функции для работы с маршрутом мы будем выполнять одно и то же действие, а именно:

```

sc_addr add_vertex_visit_to_route(sc_session *s, sc_addr route, sc_addr vertex)
{
    // 1. Получим компоненты маршрута: структуру маршрута и отношение посещения.
    sc_addr route_struct = get_route_struct(s, route); // структура маршрута
    sc_addr route_visit = get_route_visit(s, route); // отношение посещения

    // ...
}

```

Зная структуру маршрута и отношение посещения, можно создать посещение вершины в ориентированном графе структуры маршрута:

```

sc_addr add_vertex_visit_to_route(sc_session *s, sc_addr route, sc_addr vertex)
{
    // ...

    // 2. Создадим посещение вершины.
    sc_addr vertex_visit = s->create_el(route->seg, SC_N_CONST);
    sc_tup::add(s, route->seg, route_struct, graph_theory::vertex_, vertex_visit);
    sc_rel::add_ord_tuple(s, route_visit, vertex_visit, vertex);

    return vertex_visit;
}

```

В будущем нам понадобится функция аналогичная `add_vertex_visit_to_route`, но для добавления посещения ребра. Начало ее будет такое же, как и у `add_vertex_visit_to_route`:

```

/// Создает в структуре маршрута @p route посещение ребра/дуги @p connective
/// из вершины-посещения @p from_visit в вершину-посещение @p to_visit
/// и возвращает это посещение.

```



```

///
/// @note Все элементы генерируются в сегменте связки @p route.
sc_addr add_connective_visit_to_route(sc_session *s, sc_addr route, sc_addr connective,
                                     sc_addr from_visit, sc_addr to_visit)
{
    // 1. Получим компоненты маршрута: структуру маршрута и отношение посещения.
    sc_addr route_struct = get_route_struct(s, route); // структура маршрута
    sc_addr route_visit = get_route_visit(s, route); // отношение посещения

    // ...
}

```

Создадим узел посещения ребра в структуре маршрута:

```

sc_addr add_connective_visit_to_route(sc_session *s, sc_addr route, sc_addr connective,
                                     sc_addr from_visit, sc_addr to_visit)
{
    // ...

    // 2. Создадим узел посещения ребра/дуги в структуре маршрута,
    // укажем чьим посещением он является.
    sc_addr edge_visit = s->create_el(route->seg, SC_N_CONST);
    sc_tup::add(s, route_struct, graph_theory::arc_, edge_visit);
    sc_rel::add_ord_tuple(s, route_visit, edge_visit, connective);

    // ...
}

```

Узел посещения создали, а теперь надо указать, что посещение выходит из from_visit, а входит в to_visit:

```

sc_addr add_connective_visit_to_route(sc_session *s, sc_addr route, sc_addr connective,
                                     sc_addr from_visit, sc_addr to_visit)
{
    // ...

    // 3. Укажем, что посещение ребра/дуги выходит из вершины from_visit и
    // входит в вершину to_visit.
    sc_tup::add(s, edge_visit, N1_, from_visit);
    sc_tup::add(s, edge_visit, N2_, to_visit);

    return edge_visit;
}

```

Вернемся к построению пути. Нам нужно пройтись по списку волн от конца в начало и построить найденный путь, что соответствует шагам алгоритма на страницах [54](#), [55](#), [56](#), [57](#), [58](#), [59](#). Просмотрите, пожалуйста, эти шаги, чтобы знать, как будут меняться значения переменных. А цикл мы организуем следующим образом:

```

sc_addr find_min_path(sc_session *s, sc_segment *seg, sc_addr graph,
                     sc_addr beg_vertex, sc_addr end_vertex)
{
    // ...

    // 8. Пройдем в обратном направлении по списку волн
    // и сформируем структуру маршрута.
}

```

```

sc_addr curr_vertex = end_vertex;
sc_addr curr_visit = end_vertex_visit;

sc_list::reverse_iterator list_it(s, waves_list_tail), list_end;
for (++list_it; list_it != list_end; ++list_it) {
    sc_addr curr_wave = *list_it;

    // ...
}

// ...
}

```

В начале цикла найдем ребро, которое связывает `curr_vertex` с предыдущей вершиной в пути при помощи функции `find_any_edge`:

```

sc_addr find_min_path(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr beg_vertex, sc_addr end_vertex)
{
    // ...

    for (++list_it; list_it != list_end; ++list_it) {
        sc_addr curr_wave = *list_it;

        // 9. Находим ребро, которое связывает curr_vertex с предыдущей вершиной пути.
        sc_addr edge = find_any_edge(s, graph, curr_vertex, curr_wave);

        // ...
    }

    // ...
}

```

Функцию `find_any_edge` определим следующим образом (похожий цикл мы уже использовали в листинге 4.5.5):

```

/// Ищет в графе @p graph любое ребро, которое инцидентно вершине @p vertex
/// и любой вершине из волны @p prev_wave.
sc_addr find_any_edge(sc_session *s, sc_addr graph, sc_addr vertex, sc_addr prev_wave)
{
    sc_addr edge = 0;

    // 1. Перебор всех ребер, которые инцидентны vertex.
    sc_iterator *it = s->create_iterator(
        sc_constraint_new(
            CONSTR_3l2_f_a_a_a_f,
            graph,
            SC_A_CONST|SC_POS,
            SC_N_CONST,
            SC_A_CONST|SC_POS,
            vertex
        ), true);
    sc_for_each (it) {
        edge = it->value(2); // ребро, инцидентное vertex
    }
}

```

```

    sc_addr other_vertex = get_other_vertex_incidence_edge(
        s, edge, vertex); // вершина, смежная vertex и инцидентная ребру

    if(sc_set::is_in(s, other_vertex, prev_wave))
        break;
}

return edge;
}

```

Вернемся к циклу построения структуры пути. На основе найденного ребра `edge` мы можем найти предыдущую вершину в пути и добавить ее посещение в структуру формируемого пути:

```

sc_addr find_min_path(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr beg_vertex, sc_addr end_vertex)
{
    // ...

    for (++list_it; list_it != list_end; ++list_it) {
        // ...

        // 10. Получаем предыдущую вершину в пути и добавляем ее посещение в структуру пути.
        sc_addr prev_vertex = get_other_vertex_incidence_edge(s, edge, curr_vertex);
        sc_addr prev_visit = add_vertex_visit_to_route(s, route, prev_vertex);

        // ...
    }

    // ...
}

```

Не забудем при помощи функции `add_connective_visit_to_route` добавить в структуру пути посещение ребра `edge`:

```

sc_addr find_min_path(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr beg_vertex, sc_addr end_vertex)
{
    // ...

    for (++list_it; list_it != list_end; ++list_it) {
        // ...

        // 11. Добавляем посещение ребра edge в структуру пути.
        add_connective_visit_to_route(s, route, edge, prev_visit, curr_visit);

        // ...
    }

    // ...
}

```

Оканчиваем одну итерацию цикла, приняв предыдущую вершину и ее посещение за текущие:

```

sc_addr find_min_path(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr beg_vertex, sc_addr end_vertex)
{

```

```

// ...

for (++list_it; list_it != list_end; ++list_it) {
    // ...

    // 12. Принимаем предыдущую вершину и ее посещение за текущие.
    curr_vertex = prev_vertex;
    curr_visit = prev_visit;
}

// ...
}

```

Итак, цикл закончился, и структура пути готова. Подчистим память, что соответствует шагу алгоритма на странице [60](#):

```

sc_addr find_min_path(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr beg_vertex, sc_addr end_vertex)
{
    // ...

    // 13. Удалим список волн.
    erase_waves_list(s, waves_list_head);

    // ...
}

```

Вернем из функции `find_min_path` связку `route`, что соответствует шагу алгоритма на странице [61](#):

```

sc_addr find_min_path(sc_session *s, sc_segment *seg, sc_addr graph,
    sc_addr beg_vertex, sc_addr end_vertex)
{
    // ...

    return route;
}

```

Мы завершили функцию `find_min_path` и у нас еще осталась неописанной функция `print_route` (см. раздел [4.5.4](#)).

4.5.6 Вывод маршрута на консоль

Для вывода маршрута на консоль нам необходимо найти начальное посещение в структуре маршрута и от него уже выводить маршрут. Для этого напомним функцию `get_route_struct_begin`:

```

/// Находит начальное посещение в структуре маршрута @p route_struct.
sc_addr get_route_struct_begin(sc_session *s, sc_addr route_struct)
{
    // 1. Начальной считается вершина, в которую нет входящих связей.
    // Переберем все вершины и проверим их на это свойство.
    sc_iterator *it = s->create_iterator(
        sc_constraint_new(
            CONSTR_5_f_a_a_a_f,
            route_struct,

```

```

        SC_A_CONST|SC_POS,
        0,
        SC_A_CONST|SC_POS,
        graph_theory::vertex_
    ), true);
sc_for_each (it) {
    sc_addr vertex = it->value(2);

    // ...
}

// ...
}

```

И найдем вершину, в которую не входит ни одна дуга структуры маршрута:

```

sc_addr get_route_struct_begin(sc_session *s, sc_addr route_struct)
{
    // ...

    sc_for_each (it) {
        sc_addr vertex = it->value(2);

        // 2. Ориентированный граф структуры маршрута можно рассматривать как
        // бинарное ориентированное отношение, которое связывает вершины.
        // Это позволяет нам использовать для проверки метод sc_rel::bin_ord_at_1.
        if (!sc_rel::bin_ord_at_1(s, route_struct, vertex))
            return vertex;
    }

    return 0;
}

```

Теперь приступим к написанию функции вывода маршрута на консоль — `print_route` (начало стандартное для всех функций работы с маршрутами):

```

/// Выводит на консоль маршрут, получив связку отношения @p route.
void print_route(sc_session *s, sc_addr route)
{
    // 1. Получим компоненты маршрута: структуру маршрута и отношение посещения.
    sc_addr route_struct = get_route_struct(s, route); // структура маршрута
    sc_addr route_visit = get_route_visit(s, route); // отношение посещения

    // 2. Найдем начальное посещение и посещенную вершину в оригинальном графе.
    sc_addr curr_visit = get_route_struct_begin(s, route_struct);
    sc_addr visited_vertex = sc_rel::bin_ord_at_2(s, route_visit, curr_visit);

    // ...
}

```

Переберем от начального посещения к конечному посещению и выведем посещённые вершины на консоль:

```

/// Выводит на консоль маршрут, получив связку отношения @p route.
void print_route(sc_session *s, sc_addr route)
{

```

```

// ...

// 3. Пройдем от начального посещения к конечному посещению
// и выведем посещенные вершины на консоль.
// Ориентированный граф структуры маршрута можно рассматривать как
// бинарное ориентированное отношение, которое связывает вершины.
// Это позволяет нам использовать sc_rel::bin_ord_at_2
// для поиска конца дуги орграфа.
std::cout << s->get_idtf(visited_vertex);
while (true) {
    curr_visit = sc_rel::bin_ord_at_2(s, route_struct, curr_visit);

    if (curr_visit) {
        visited_vertex = sc_rel::bin_ord_at_2(s, route_visit, curr_visit);
        std::cout << "□->□" << s->get_idtf(visited_vertex);
    } else {
        break;
    }
}

std::cout << '\n';
}

```

На этом завершим рассмотрение программы волнового поиска одного из минимальных путей в неориентированном графе и данного этапа расчетной работы.

Литература

- [1] Оре О. Теория графов. – 2-е изд.. – М.: Наука, 1980. – С. 336.
- [2] Кормен Т. Х. и др. Часть VI. Алгоритмы для работы с графами // Алгоритмы: построение и анализ = Introduction to Algorithms. – 2-е изд.. – М.: Вильямс, 2006. – С. 1296.
- [3] Харари, Ф. Теория графов / Ф. Харари / Пер. с англ. и предисл. В.П. Козырева. Под ред. Г.П. Гаврилова. Изд. 2-е. – М.: Едиториал УРСС, 2003. – 269 с.
- [4] Нечипуренко, М. И. Алгоритмы и программы решения задач на графах и сетях / М.И. Нечипуренко, В.К. Попков, С.М. Майнагашев и др. – Новосибирск: Наука. Сиб. отд-ние, 1990. – 515 с.
- [5] Емеличев В. А., Мельников О. И., Сарванов В. И., Тышкевич Р. И. Лекции по теории графов. М.: Наука, 1990. 384с. (Изд.2, испр. М.: УРСС, 2009. 392 с.)
- [6] Касьянов, В. Н. Графы в программировании: обработка, визуализация и применение / В. Н. Касьянов, В. А. Евстигнеева. – СПб. : БХВ-Петербург, 2003.
- [7] База знаний по теории графов OSTIS GT [Электронный ресурс] / проект OSTIS, 2012. – Режим доступа: <http://ostisgraphstheo.sourceforge.net>. — Дата доступа : 11.09.2012.