Министерство науки и высшего образования Российской Федерации
федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Московский государственный технический университет имени
Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Робототехники и комплексной автоматизации»
КАФЕДРА «Системы автоматизированного проектирования (РК-6)»

НАУЧНО-ИССЛЕДОВАТЕЛЬСКИЕ ЗАМЕТКИ

по направлению «Разработка систем инженерного анализа и ресурсоемкого ПО (rndhpc)»

Авторы (исследователи):	Крехтунова Д., Ершов В., Муха В., Тришин И.
Научный(е) руководитель(и):	Соколов А.П.
Консультанты:	@Фамилия И.О.@

Москва, 2021–2021

RNDHPC
C59

Работа (документирование) над научным направлением начата 20 сентября 2021 г.

Руководители по направлению:

СОКОЛОВ, Александр Павлович	–	канд. физ.-мат. наук, доцент кафедры САПР, МГТУ им. Н.Э. Баумана
ПЕРШИН, Антон Юрьевич	–	PhD, ассистент кафедры САПР, МГТУ им. Н.Э. Баумана

Исследователи (студенты кафедры САПР, МГТУ им. Н.Э. Баумана):
Крехтунова Д., Ершов В., Муха В., Тришин И.

C59 **Крехтунова Д., Ершов В., Муха В., Тришин И.**. **Разработка систем инженерного анализа и ресурсоемкого ПО (rndhpc)**: Научно-исследовательские заметки. / Под редакцией Соколова А.П. [Электронный ресурс] — Москва: 2021. — 19 с. URL: https://arch.rk6.bmstu.ru (облачный сервис кафедры РК6)

Документ содержит краткие материалы, формируемые обучающимися и исследователями в процессе их работ по одному научному направлению.

Документ разработан для оценки результативности проведения научных исследований по направлению «Разработка систем инженерного анализа и ресурсоемкого ПО» в рамках реализации курсовых работ, курсовых проектов, выпускных квалификационных работ бакалавров и магистров, а также диссертационных исследований аспирантов кафедры «Системы автоматизированного проектирования» (РК6) МГТУ им. Н.Э. Баумана.

RNDHPC

Содержание

1 Теоретические основы графоориентированного программного каркаса

2022.01.01: Отличие сетевых моделей от сетевых графиков

> “*Сетевые модели отличаются от сетевых графиков тем, что в их вершинах могут реализовываться сложные логические и вероятностные функции, а также тем, что в них допускаются контуры*

Малинин Л.И.[1], 1970, [1].

В работе [1] проф. Л.И. Малинин использует термин *контуры*, что в современной литературе по теории графов часто называют *циклами*.

В работах [2, 3] д.т.н. В.И. Нечипоренко представляет обобщёный подход к графовому описанию сложных процессов и систем.

Подготовлено: *Соколов А.П. (РК-6), 2022.01.01*

2 Разработка web-ориентированного редактора графовых моделей

2021.10.05: Обзор языка описания графов DOT

Язык описания графов DOT предоставляется пакетом утилит Graphviz (Graph Visualization Software). Пакет состоит из набора утилит командной строки и программ с графическим интерфейсом, способных обрабатывать файлы на языке DOT, а также из виджетов и библиотек, облегчающих создание графов и программ для построения графов. Более подробно будет рассмотрена утилита dot.

Замечание 1
dot – программный инструмент для создания многоуровневого графа с возможностью вывода изображения полученного графа в различных форматах (PNG, PDF, PostScript, SVG и др.).

Установка graphviz:
Linux: `sudo apt install graphviz`
MacOS: `brew install graphviz`

Вызов всех программ Graphviz осуществляется через командную строку, в процессе ознакомления с языком использовалась следующая команда:

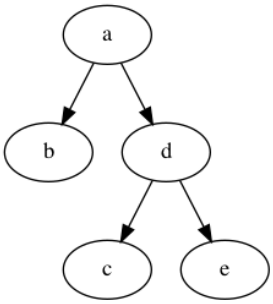
```
dot −Tpng <pathToDotFile> −o <imageName>
```

[1] Профессор Малинин Л.И. использовал псевдоним и публиковался как Ермилов Л.И.

В результате выполнения этой команды будет создано изображение графа в формате png.

Пример описания простого графа на языке DOT представлен далее.

```
digraph G {
    a -> b;
    a -> d -> c;
    d -> e;
}
```



Более подробная информация с примерами представлена в обзоре литературы, который размещён по следующему адресу:

01 - Курсовые проекты/2021-2022 - Разработка web-ориентированного редактора графовых моделей /0 - Обзор литературы/

Подготовлено: *Ершов В. (РК6-72Б), 2021.10.05*

2021.12.04: Краткое описание алгоритма визуализации графа

В ходе разработки web-ориентированного редактора графов, который позволяет импортировать и экспортировать файлы в формате aDOT [4], был разработан алгоритм визуализации графов. В этой заметке будет рассмотрен этот алгоритм и приведено несколько примеров aDOT файлов, которые корректно визуализируются, используя этот алгоритм.

Теоретические основы и назначение графоориентированной программной инженерии представлены в работе [5]. Принципы применения графоориентированного подхода зафиксированы в патенте [6].

Один из самых важных критериев построенного графа – это его читаемость. Граф должен быть построен корректно и недопускать неоднозначностей при его чтении. Например, вершины не должны налегать друг на друга, ребра не должны пересекаться, создавая сложные для восприятия связи.

Проведя длительную аналитическую работу, было принято решение разбивать граф по уровням. Рассмотрим следующее aDOT-определение графовой модели G (листинг 1).

Листинг 1. Пример aDOT-определение простейшей графовой модели G

```
1 digraph G {
2 // Parallelism
3   s1 [parallelism=threading]
4 // Graph definition
5    __BEGIN__ ->s1
6   s4 ->s6 [morphism=edge_1]
7   s5 ->s6 [morphism=edge_1]
```

```
  s1 =>s2 [morphism=edge_1]
  s1 =>s3 [morphism=edge_1]
  s2 ->s4 [morphism=edge_1]
  s3 ->s5 [morphism=edge_1]
  s6 ->__END__
}
```

Если нарисовать такой граф на бумаге, то становится очевидно, что каждый набор вершин имеет одну координату по оси X, а связанные с ними вершины находятся правее по координате X, таким образом напрашивается разбиение графа по уровням. На рисунке (1) представлен этот граф, разбитый на уровни.

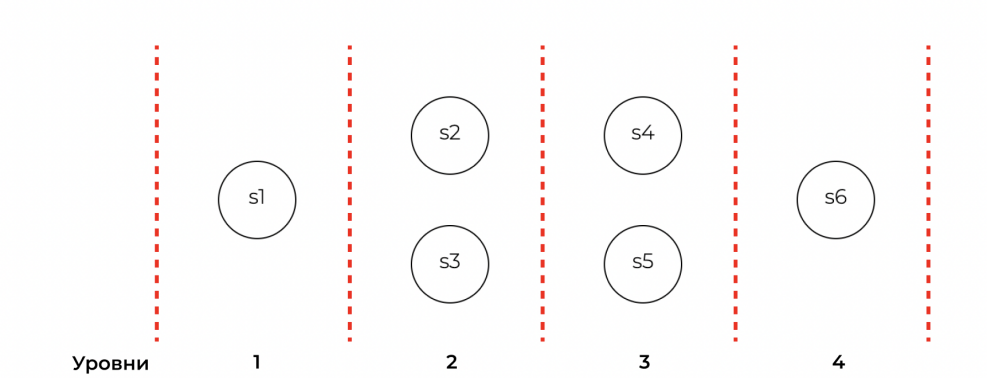


Рис. 1. Узлы графовой модели G, представленные на разных “уровнях”

Разбиение графа по уровням является основой алгоритма визуализации, далее на примере более сложного графа поэтапно разберем алгоритм.

Рассмотрим следующий файл на языке aDOT (листинг 2).

Листинг 2. Пример aDOT-определения графовой модели TEST

```
digraph TEST
{
// Parallelism
  s11 [parallelism=threading]
  s4 [parallelism=threading]
  s12 [parallelism=threading]
  s15 [parallelism=threading]
  s2 [parallelism=threading]
  s8 [parallelism=threading]
// Functions
  f1 [module=DEFAULT_VALUE, entry_func=DEFAULT_VALUE]
// Predicates
  p1 [module=DEFAULT_VALUE, entry_func=DEFAULT_VALUE]
// Edges
  edge_1 [predicate=p1, function=f1]
```

```
// Graph model description
  __BEGIN__ ->s1
  s6 ->s8 [morphism=edge_1]
  s7 ->s8 [morphism=edge_1]
  s10 ->s8 [morphism=edge_1]
  s11 =>s8 [morphism=edge_1]
  s11 =>s9 [morphism=edge_1]
  s14 ->s9 [morphism=edge_1]
  s3 ->s9 [morphism=edge_1]
  s4 =>s6 [morphism=edge_1]
  s4 =>s7 [morphism=edge_1]
  s4 =>s10 [morphism=edge_1]
  s4 =>s11 [morphism=edge_1]
  s12 =>s4 [morphism=edge_1]
  s12 =>s14 [morphism=edge_1]
  s13 ->s3 [morphism=edge_1]
  s15 =>s14 [morphism=edge_1]
  s15 =>s14 [morphism=edge_1]
  s2 =>s12 [morphism=edge_1]
  s2 =>s13 [morphism=edge_1]
  s2 =>s15 [morphism=edge_1]
  s1 ->s2 [morphism=edge_1]
  s8 =>s9 [morphism=edge_1]
  s8 =>s6 [morphism=edge_1]
  s9 ->__END__
}
```

В результате визуализации модели будет получен граф, представленный на рисунке (2).

Основная задача алгоритма – это отрисовать вершины графа в корректных позициях, а затем построить между ними ребра. Поскольку редактор графов – это web-ориентированное приложение, то вся бизнес-логика написана на языке JavaScript. В JavaScript имеется очень удобный инструмент – объекты. Аналогами объектов в других языках программирования являются хэш-карты, но в Javascript возможности использования объектов гораздо шире. Всю информацию об уровнях графа будем хранить в объекте levels. Ниже представлено как выглядит объект levels для рассматриваемого выше графа:

```
{
"1":[{"s1":[]}],
"2":[{"s2":["s1"]}],
"3":[{"s12":["s2"]}, {"s13":["s2"]}, {"s15":["s2"]}],
"4":[{"s4":["s12"]}],
"5":[{"s9":["s14","s3"]}, {"s6":["s4"]}, {"s7":["s4"]}, {"s10":["s4"]},
{"s11":["s4"]},{"s14":["s12","s15"]}, {"s3":["s13"]}],
```

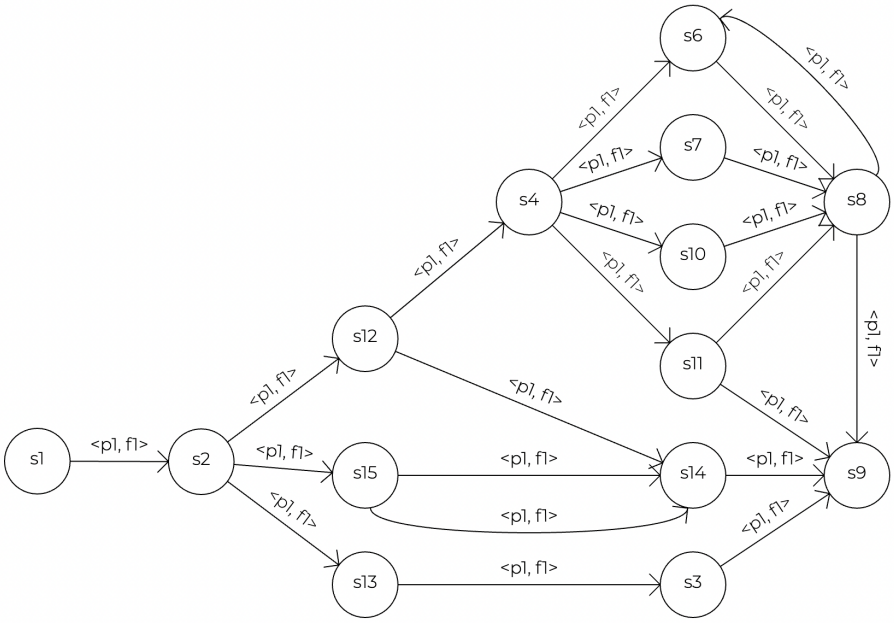


Рис. 2. Визуализация графовой модели TEST

```
"6":[{"s8":["s6","s7","s10","s11"]}, {"s9":["s11","s14","s3"]}]
}
```

Ключами объекта levels являются номера уровней, представленные в формате string. Свойсвами являются массивы, на один уровень - один массив. Каждый элемент массива содержит информацию об одной вершине на этом уровне, следовательно количество вершин на уровне - это размер массива. Далее заметим, что элемент массива это не просто string с названием вершины, а объект. Этот объект содержит один ключ - название вершины на этом уровне, а свойством является массив, который содержит список вершин из которых перешли в эту вершину (переход только по одному ребру).

В качестве примера рассмотрим уровень 5 объекта levels, который был представлен ранее.

```
{
  "5":[{"s9":["s14","s3"]}, {"s6":["s4"]}, {"s7":["s4"]},
  {"s10":["s4"]}, {"s11":["s4"]}, {"s14":["s12","s15"]}, {"s3":["s13"]}],
}
```

Уровень 5 в графе представлен 7-ю вершин: “s9”, “s6”, “s7”, “s10”, “s11”, “s14”, “s3”. Например, в вершину “s6” мы пришли из вершины “s4”, таким образом, по ключу “s6” находится массив с один элементом “s4”

```
{"s6": ["s4"]}
```

На рисунке (3) представлен граф с выделенным уровнем №5.

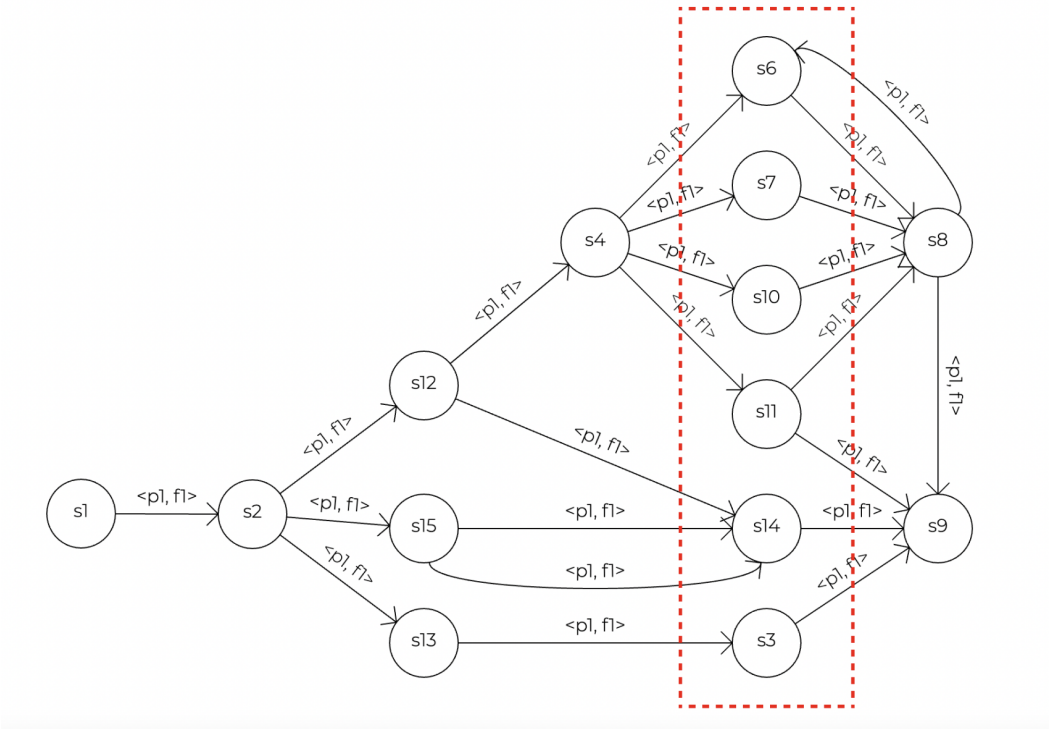


Рис. 3. Уровень №5 на графе

Обратим вниманием на то, что в объекте levels для уровня №5 содержится вершина s_9 которой нет на уровне №5, а она присутствует только на уровне №6. Заполнение объекта levels происходит слево-направо, то есть от меньшего уровня к большему, например, в графе представленном выше есть связь $s_{13} \to s_3$, таким образом пока в объект levels не будет записана вершина s_{13}, мы не сможем записать связанную с ней вершину s_3.

Обратным образом происходит дублирование вершин, если обратиться к описанию графовой модели предсталвенной выше, то можно заметить такую связь: $s_1 \to s_2 \to s_{13} \to s_3 \to s_9$. При начальной инициализации объекта levels s_1 будет находиться на уровне 1, s_2 будет находиться на уровне 2 и так далее. Таким образом, вершина s_3 будет находиться на уровне 4, что не совсем корректно. Обработка таких ситуаций является углублением в реализацию алгоритма и в этой заметке не будет подробно описываться.

Для разрешения подобных коллизий после начальной инициализации объекта levels “в лоб” предусмотрено множество дополнительных проверок. Таким образом, на выходе мы получаем корректно сформированный объект levels и дополнительно формируем объект, который будет хранить информацию о вершинах которые находятся не на своем уровне, эти вершины не будут отрисовываться, но при этом они будут учитываться при выравнивании связанных с ними вершин, следовательно просто удалить такие вершины из объекта levels нельзя.

Сама визуализация вершин после формирования объекта levels также является те-

мой отдельной заметки. На данном этапе работы над проектом сначала строится уровень содержащий наибольшее количество вершин, затем строятся оставшиеся уровни: сначала влево до уровня №1, затем вправо до самого последнего уровня.

Подготовлено: *Ершов В. (РК6-72Б), 2021.12.04*

2021.12.05: Описание текущего состояния проекта

В данной заметке описаны все основные пользовательские сценарии web-ориентированного редактора графов по состоянию на 2021.12.05. Для каждого сценария составлен следующий список:

1) реализованные возможности сценария;
2) сложности, возникшие при реализации сценария;
3) будущий функционал, который расширит user expirience.

Основые сценарии приложения:

1) создание графа в приложении путем добавления вершин и ребер между ними;
2) экспортирование графа в файл на языке описания графов aDOT;
3) импортирование графа из файла на языке описания графов aDOT.

Добавление вершины Для доблавения вершины необходимо выбрать соответствующий пункт в меню, а затем кликнуть на холст. После этого потребуется уточнить название вершины, после ввода названия вершины построение будет полностью завершено. Процесс построения вершины представлен на рисунке 4.

Дополнительный функционал сценария:

1) блокировка создания вершины если она находится в близости к другой вершине;
2) блокировка создания названия вершины, если такое название присвоено существующей вершине.

Сложности при реализации сценария:

1) валидировать все необходимые параметры в процессе создания вершины: положение вершины, название вершины;
2) выбор информации о вершине, которую нужно сохранить в оперативной памяти для дальнейшего использования в других сценариях.

Будущее расширение функционала сценария.

1) Возможность изменить положение вершины в процессе ее создания. На данный момент вершина создается в месте клика на холст (в том случае если местоположение вершины валидно) и это местоположение уже никак нельзя изменить.
2) Использование набора горячих клавиш для ускорения процесса создания вершины.
3) При названия подсказать пользователю название вершины, в том случае если прошлые названия имеют инкрементирующийся для каждой вершины постфикс, например, уже созданы вершины s_1, s_2, при создании новой вершины в поле ввода названия система предложит пользователю название s_3.

Рис. 4. Создание вершины

Добавление ребра Для добавления ребра необходимо поочередно выбрать две вершины, которые будут соединены ребром. Затем система потребует ввода предиката и функции, в том случае если введенный предикат или функция являются уникальными в рамках графа, то система потребует ввода информации о module и entry func. После этого построение ребра будет завершено. Процесс построения ребра представлен на рисунке 5.

Дополнительный функционал сценария.

– Блокировка создания ребра из вершины в нее же саму.

– Если между вершинами уже существуют ребра, то система построит ребро, используя кривые Безье.

– Если на момент построения ребра из вершины выходит только одно ребро, то система потребует учтонения типа параллелизма (формат aDOT).

– Если предикат или функция не являются уникальными в рамках графа, то не требуется уточнение информации о module и entry func.

Сложности при реализации сценария.

– Использование множества вспомогательных функций: подсчет количества ребер между вершинами, для определения типа построения ребра (прямое или кривые Безье), подсчет количества ребер выходящих из вершины для уточнения типа параллелизма, проверка полей ввода предиката и функции (предикат может быть

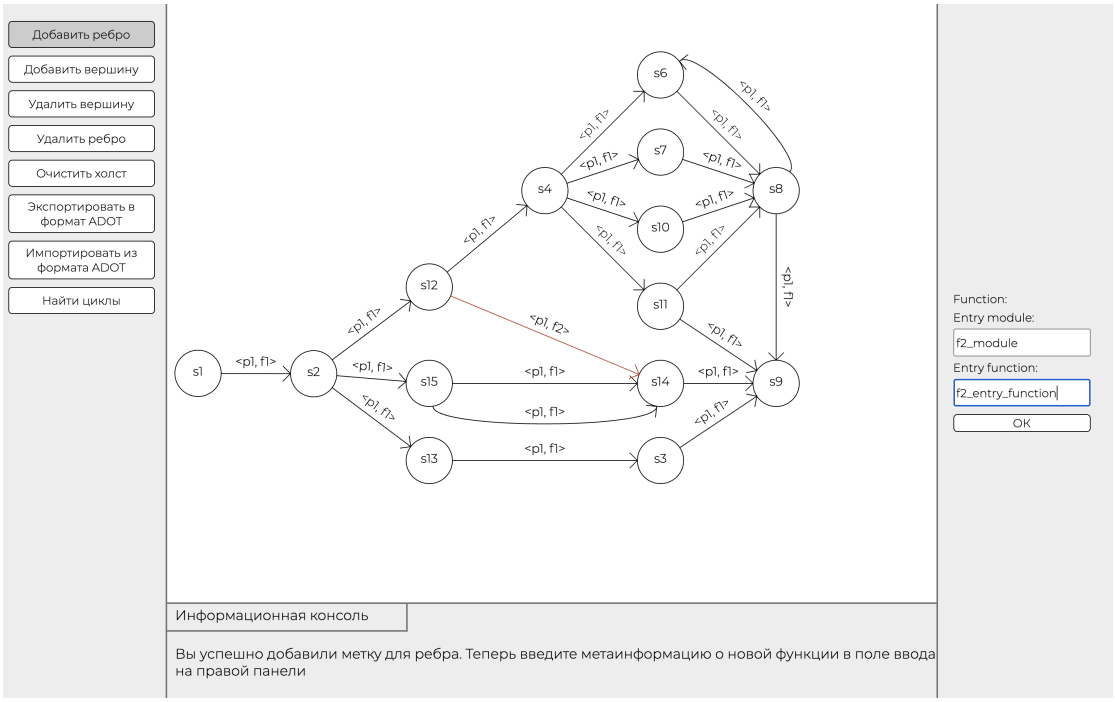


Рис. 5. Создание ребра

неопределен, а функция должна быть определена), проверка введенных предиката и функции на уникальность в рамках графа.

– Выбор типа построения ребра: прямое, кривые Безье. Обработка ситуации, когда пользователь создает цикл между этими вершинами.

Будущее расширение функционала сценария.

– Возможность перевыбора вершин для построения ребра между ними. На данный момент нельзя допустить ошибку при выборе вершин, необходимо закончить процесс построения ребра, удалить его и создать новое, выбрав другие вершины.

– Алгоритм для построения ребра с использованием кривых Безье. На данный момент ребро строится с помощью одной кривой Безье, что не позволяет строить ребра более сложного вида, тем самым разрешая коллизии, когда ребро проходит через другие вершины.

Удаление вершины Для удаления вершины необходимо выбрать соответствующий пункт в меню, а затем кликнуть на вершину для удаления - вершина удаляется с холста вместе со всеми связанными с ней ребрами.

Сложности при реализации сценария.

– Корректно удалить все связанные с вершиной ребра, а затем удалить информацию об этих ребрах и связанных вершин.

Будущее расширение функционала сценария.

– Возможность одновременного выбора нескольких вершин для удаления.

– Возможность откатить действия в том случае если была удалена лишняя вершина. В целом это касается всего приложения, на данный момент любое действие неотвратимо.

Удаление ребра Для удаления ребра необходимо выбрать соответствующий пункт в меню, а затем клкинуть на ребро для удаления - ребро удаляется с холста.

Сложности при реализации сценария.

– Удалить информацию о ребре из связанных этим ребром вершин.

Будущее расширение функционала сценария.

– Более корректный выбор ребра, на данный момент надо кликать ровно на ребро поскольку для перехвата события клика используется event.target.closest.

– Возможность одновременного выбора нескольких ребер для удаления.

Экспорт в формат aDOT Для экспорта в формат aDOT необходимо выбрать соответствующий пункт в меню, а затем поочередно выбрать две вершины, которые будут являться стартовой и конечной вершиной соответственно. Затем сформируется текстовый файл с описание графа в формате aDOT и автоматически начнется его загрузка.

Импорт из формата aDOT Для импорта из формата aDOT необходимо выбрать соответствующий пункт в меню, а затем выбрать файл для импорта. Далее система сама построит граф, сохранив всю необходимую информацию.

Сложности при реализации сценария.

– Разработка алгоритма визуализации...

Будущее расширение функционала сценария.

– Разработка более гибкого алгоритма визуализации, который сможет работать с более сложными графами.

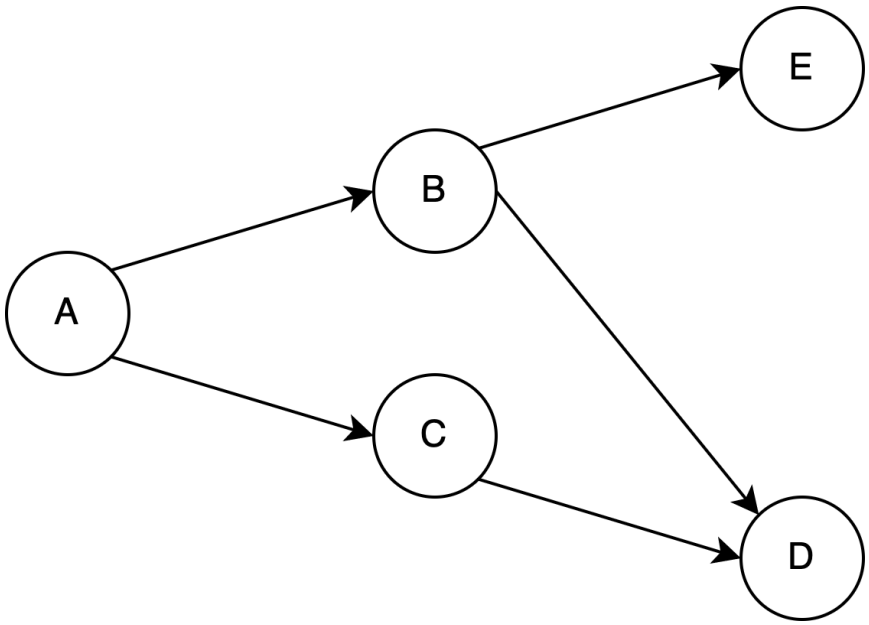
Подготовлено: *Ершов В. (РК6-72Б), 2021.12.05*

2022.04.25: Использование алгоритма DFS для решения задачи поиска циклов в ориентированном графе

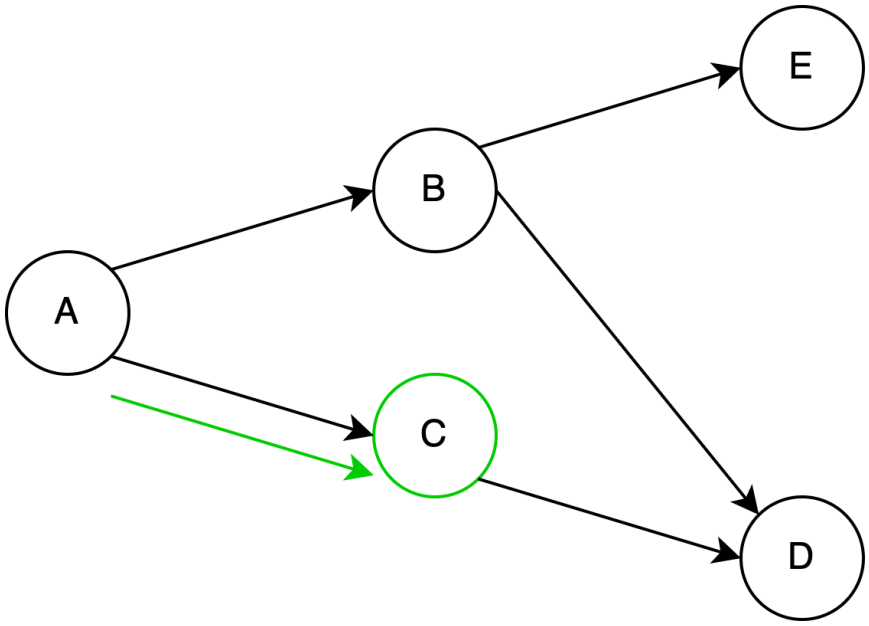
Для поиска циклов в ориентированном графе необходим алгоритм обхода графа. Обход графа - это переход от одной вершины графа к другой с целью поиска ребер или вершин, которые удовлетворяют некоторому условию.

Основными алгоритмами обхода графа являются поиск в ширину (Breadth-First Search, BFS) и поиск в глубину (Depth-First Search). Основное различие между DFS и BFS состоит в том, что DFS проходит путь от начальной вершины до конечной, а BFS двигается вперед уровень за уровнем. Из этого следует, что алгоритмы применяются для решения разных задач. BFS используется для более эффективного нахождения кратчайшего пути в графе, определения связанных компонент в графе, а также обнаружения двудольного графа. DFS применяется для проверки графа на ацикличность или для решения задачи поиска циклов в графе.

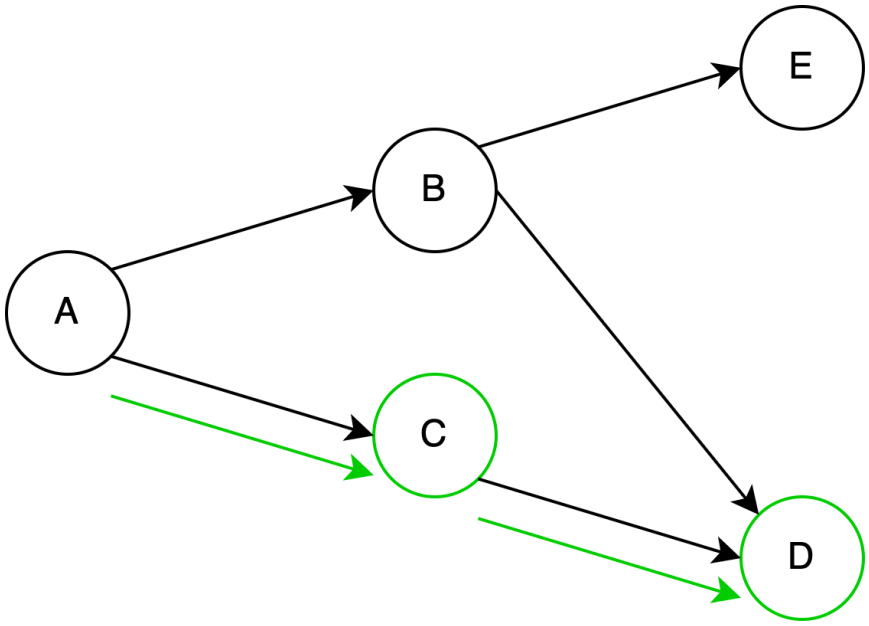
Как ранее было сказано, алгоритм DFS двигается от начальной вершины до тех пор пока не будет достигнута конечная вершина. Если был достигнут конец пути, но искомая вершина так и не была найдена, то необходимо вернуться назад (к точке разветвления) и пойти по другому маршруту. Рассмотрим работу алгоритма на примере:



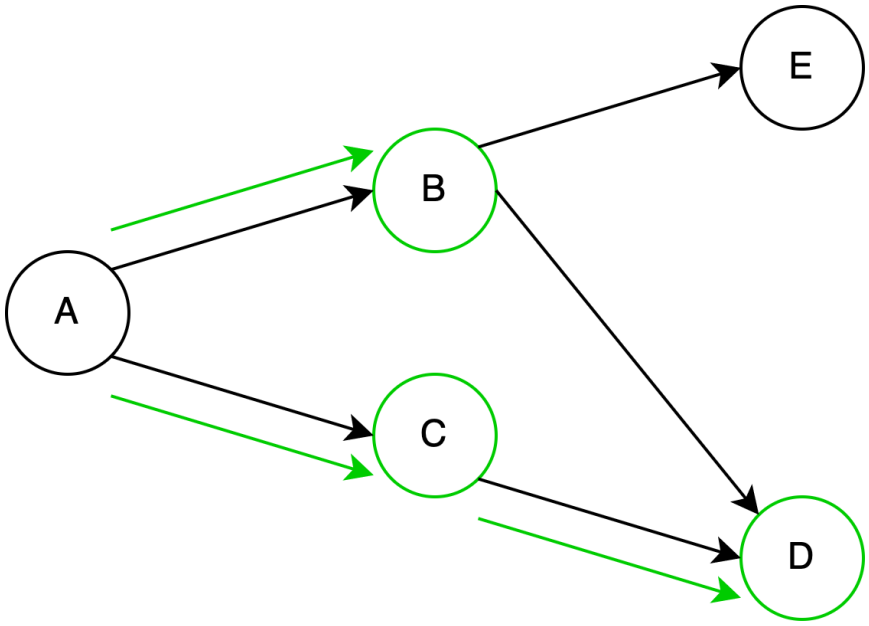
Мы находимся в точке “A” и хотим найти вершину “E”. Согласно принципу DFS, необходимо исследовать один из возможных маршрутов до конца, если не будет обнаружена вершина “E”, то возвращаемся и исследуем другой маршрут.



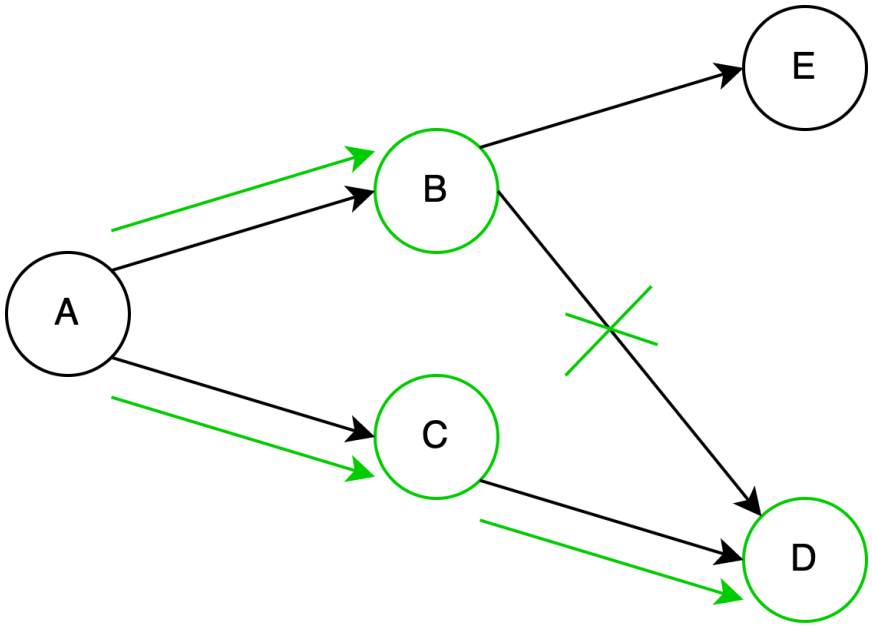
В данном случае мы двигаемся к ближайшей вершине “C”, поскольку это не конец пути, то переходим к следующей вершине.

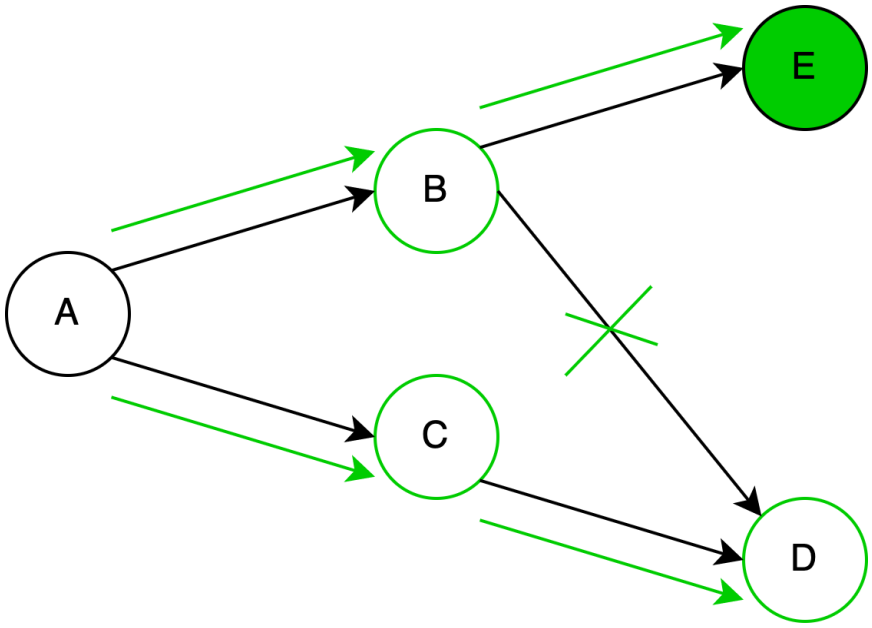


Мы достигли конца пути, но не нашли “E”, поэтому возвращаемся в начальную вершину “A” и двигаемся по другому пути.



Из вершины “B” существует два возможных дальнейших пути. Поскольку вершина “D” была ранее расмотрена двигаемся по другому пути.





Мы нашли искомую вершину “E”, следовательно можно завершать выполнение алгоритма.

В рассмотреном примере решалась достаточно тривиальная задача - поиск вершины в графе. Более интересной является задача поиска циклов в графе. Каждая вершина графа может находиться в трех различных состояниях: вершина не посещена, вершина посещена и вершина посещена, но мы не дошли до конца пути, который включает эту вершину. Для ясности введем следующие обозначения состояний:

– NOT_VISITED - вершина еще не посещена;

– IN_STACK - вершина посещена, но мы не дошли до конца пути;

– VISITED - вершина посещена.

Если в процессе обхода мы встречаем вершину, которая помечена как “IN_STACK”, то мы нашли цикл.

В программной реализации рационально разбить задачу на три функции:

– Функция, которая в цикле проходит по всем вершинам графа и если вершина не была ранее просмотрена, то запускает обход DFS из этой вершины

– Функция непосредственно реализующая обход DFS

– Функция для печати найденного цикла

Ниже приведен (листинг 3) кода данных функций на языке C++

Листинг 3. Решение задачи поиска циклов в ориентированном графе G

```
1 void FindCycles(const std::vector<int> &adjacency_list)
```

```
2  {
3      std::vector<std::string> visited(adjacency_list.size(), "NOT_VISITED");
4
5      for (int vertex = 0; vertex < adjacency_list.size(); ++vertex)
6      {
7          if (visited[vertex] == "NOT_VISITED")
8          {
9              std::stack<int> stack;
10             stack.push(vertex);
11             visited[vertex] = "IN_STACK";
12             processDFS(adjacency_list, visited, stack);
13         }
14     }
15 }
16
17 void processDFS(const std::vector<int> &adjacency_list,
18                 std::vector<std::string> &visited,
19                 std::stack<int> &stack)
20 {
21     for (const int &vertex : adjacency_list[stack.top()])
22     {
23         if (visited[vertex] == "IN_STACK")
24         {
25             printCycle(stack, vertex);
26         }
27         else if (visited[vertex] == "NOT_VISITED")
28         {
29             stack.push(vertex);
30             visited[vertex] = "IN_STACK";
31             processDFS(adjacency_list, visited, stack);
32         }
33     }
34
35     visited[stack.top()] = "DONE";
36     stack.pop();
37 }
38
39 void printCycle(std::stack<int>& stack, int vertex)
40 {
41     std::stack<int> stack_temp;
42     stack_temp.push(stack.top());
43     stack.pop();
44
45     while (stack_temp.top() != vertex)
46     {
47         stack_temp.push(stack.top());
```

```
48          stack.pop();
49      }
50
51      while (!stack_temp.empty())
52      {
53          std::cout << stack_temp.top() << ' ';
54          stack.push(stack_temp.top());
55          stack_temp.pop();
56      }
57
58      std::cout << '\n';
59  }
```

Подготовлено: *Ершов В. (РК6-82Б), 2022.04.25*

Список литературы

[1] Ермилов Л.И. Синтез сетевых моделей сложных процессов и систем. Москва: МО СССР, 1970. С. 100.

[2] Нечипоренко В.И. Структурный анализ и методы построения надёжных систем. Москва: Издательство «Советское радио», 1968. С. 256.

[3] Нечипоренко В.И. Структурный анализ систем (эффективность и надёжность). Москва: Издательство «Советское радио», 1977. С. 216.

[4] Соколов А.П. Описание формата данных aDOT (advanced DOT) [Электронный ресурс]. Облачный сервис SA2 Systems. [Офиц. сайт]. 2020. (дата обращения 05.03.2020)

[5] Соколов А.П., Першин А.Ю. Графоориентированный программный каркас для реализации сложных вычислительных методов // Программирование. 2019. Т. 47, № 5. С. 43–55.

[6] Соколов А.П., Першин А.Ю. Патент на изобретение RU 2681408. Способ и система графо-ориентированного создания масштабируемых и сопровождаемых программных реализаций сложных вычислительных методов. 2019. заявка № RU 2017 122 058 A, приоритет 22.07.2017, опубликовано 22.02.2019