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. — 67 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

Список сокращений и условных обозначений

aDOT Расширенный формат DOT (описание представлено в [1]). 50, 54, 55
aINI Расширенный формат INI (описание представлено в [2]). 51

ГПИ графо-ориентированная программная инженерия (англ., graph-based software engineering (GBSE)), ориентированная для создания программных реализаций СВМ (патент на изобретение RU 2681408 [3]). 36, 37, 56, 58
ПО программное обеспечение. 34
СВМ сложный вычислительный метод. 4
ТО технический объект, в т.ч. сложный процесс, система. 47, 50

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, [4].

В работе [4] проф. Л.И. Малинин использует термин *контуры*, что в современной литературе по теории графов часто называют *циклами*.

В работах [5, 6] д.т.н. В.И. Нечипоренко представляет обобщённый подход к графовому описанию сложных процессов и систем.

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

2 Разработка графоориентированного дебаггера

2021.11.10: Автоматизированные и автоматические методы отладки, применяемые при реализации сложных вычислительных методов (СВМ), отладка наукоёмкого кода, science code debugging, graph based programming и пр. (первичный обзор литературы)

Анализ взвешенных графов вызовов для локализации ошибок программного обеспечения

Далее представлен материал, являющийся результатом анализа работы [7].

Проблема Обнаружение сбоев, которые приводят к ошибочным результатам с некоторыми, но не со всеми входными данными.

Предложенное решение Метод анализа графов вызовов функций с последующим составлением рейтинга методов, которые вероятнее всего содержат ошибку.

Описание решения *Граф вызовов*

Метод основан на анализе графа вызовов. Такой граф отражает структуру вызовов при выполнении конкретной программы. Без какой-либо дополнительной обработки граф вызовов представляет собой упорядоченное дерево с корнем.

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

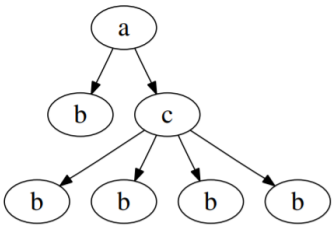


Рис. 1. Граф вызовов

Узлами являются сами методы, а гранями – их вызовы. Метод main() программы обычно является ее корнем, а все методы, вызываемые напрямую, являются ее дочерними элементами.

Редукция графа

Трассировка представляется в виде графов вызовов, затем повторяющиеся вызовы методов, вызванные итерациями, удаляются, вместо этого вводятся веса ребер, представляющие частоту вызовов.

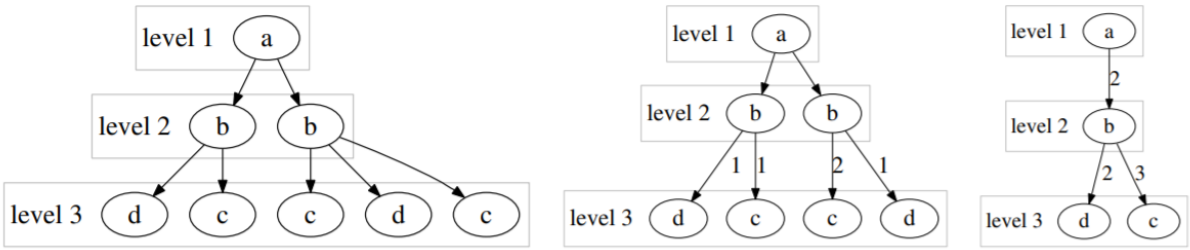


Рис. 2. Редукция графа вызовов

Поиск подграфов

Для поиска подграфов используется фреймворк для ранжирования потенциально ошибочных методов.

После сокращения графов вызовов, полученных от правильного и неудачного выполнения программ, применяется поиск часто встречающихся замкнутых подграфов SG в наборе данных графа G, используя алгоритм CloseGraph. Полученный набор подграфов разделяется на те, которые встречаются при правильном и неудачном выполнении (SGcf), и те, которые возникают только при неудачном выполнении (SGf).

Анализ

Два набора подграфов рассматриваются отдельно.

SGcf используется для построения рейтинга на основе различий в весах ребер при правильном и неудачном выполнении. Графы анализируются: применяется алгоритм выбора характеристик на основе энтропии к весам различных ребер для вычисления вероятности вызова метода и вероятности содержания в нем ошибки.

Алгоритм на основе энтропии не может обнаружить ошибки, которые не влияют на частоту вызовов и не учитывает подграфы, которые появляются только в ошибочной

версии (SGf).

Поэтому отдельно рассчитывается оценка для методов, содержащихся только в ошибочной версии. Эта оценка - еще одна вероятность наличия ошибки, основанная на частоте вызовов методов при неудачных выполнениях.

Затем вычисляется общая вероятность наличия ошибки для каждого метода. Этот рейтинг дается разработчику программного обеспечения, который выполняет анализ кода подозрительных методов.

Интегрированная отладка моделей Modelica

Далее представлен материал, являющийся результатом анализа работы [8].

Modelica – объектно-ориентированный, декларативный язык моделирования сложных систем (в частности, систем, содержащих механические, электрические, электронные, гидравлические, тепловые, энергетические компоненты).

Проблема Из за высокого уровня абстракции и оптимизации компиляторов, обеспечивающих простоту использования, ошибки программирования и моделирования часто трудно обнаружить.

Предложенное решение В статье представлена интегрированная среда отладки, сочетающая классическую отладку и специальные техники (для языков основанных на уравнениях) частично основанные на визуализации графов зависимостей.

Описание решения На этапе моделирования пользователь обнаруживает ошибку в нанесенных на график результатах, или код моделирования во время выполнения вызывает ошибку.

Отладчик строит интерактивный граф зависимостей (IDG) по отношению к переменной или выражению с неправильным значением.

Узлы в графе состоят из всех уравнений, функций, определений значений параметров и входных данных, которые использовались для вычисления неправильного значения переменной, начиная с известных значений состояний, параметров и времени.

Переменная с ошибочным значением (или которая вообще не может быть вычислена) отображается в корне графа.

Ребра могут быть следующих двух типов.

- Ребра зависимости данных: направленные ребра, помеченные переменными или параметрами, которые являются входными данными (используются для вычислений в этом уравнении) или выходными данными (вычисляются из этого уравнения) уравнения, отображаемого в узле.

- Исходные ребра: неориентированные ребра, которые связывают узел уравнения с реальной моделью, к которой принадлежит это уравнение. ребра, указывающие

из сгенерированного исполняемого кода моделирования на исходные уравнения или части уравнений, участвующие в этом коде

Пользователь может:

- Отобразить результаты симуляции, выбрав имя переменной или параметра(названия ребер). График переменной покажется в дополнительном окне. Пользователь может быстро увидеть, имеет ли переменная ошибочное значение.
- Отобразить код модели следуя по исходным ребрам.
- Вызвать подсистему отладки алгоритмического кода, если пользователь подозревает, что результат переменной, вычисленной в уравнении, содержащем вызов функции, неверен, но уравнение кажется правильным.

Используя эти средства интерактивного графа зависимостей, пользователь может проследить за ошибкой от ее проявления до ее источника.

Систематические методы отладки для масштабных вычислительных фреймворков высокопроизводительных вычислений

Далее представлен материал, являющийся результатом анализа работы [9].

Параллельные вычислительные фреймворки для высокопроизводительных вычислений играют центральную роль в развитии исследований, основанных на моделировании, в науке и технике.

Фреймворк Uintah был создан для решения сложных задач взаимодействия жидких структур с использованием параллельных вычислительных систем.

Проблема Поиск и исправление ошибок в параллельных фреймворках, возникающих из-за параллельного характера кода.

Предложенное решение В статье описывается подход к отладке крупномасштабных параллельных систем, основанный на различиях в исполнении между рабочими и нерабочими версиями. Подход основывается на трассировке стека вызовов.

Исследование проводилось для вычислительной платформы Uintah Computational Framework.

Описание решения Так как количество трассировок стека, которые можно получить при выполнении программы, может быть большим, для лучшего понимания используются графы, которые могут сжать несколько миллионов трассировок стека в одну управляемую фигуру. Метод основан на получении объединенных графов трассировки стека (CSTG).

Хотя сбор и анализ трассировки стека ранее изучались в контексте инструментов и подходов, их внимание не было сосредоточено на кросс-версии (дельта) отладке.

```
void A()                {
       cstg.addStackTrace();
}
void B()                {

     A();
}
int   main() {
    int     x = random();
    if       (x > 0)  B();
    A();
}
```

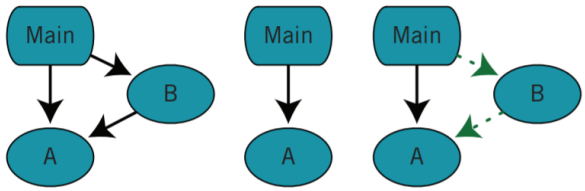


Рис. 3. Пример построения CSTG

CSTG не записывают каждую активацию функции, а только те, что в трассировках стека ведут к интересующей функции (функциям), выбранной пользователем. Каждый узел CSTG представляет все активации конкретного вызова функции. Помимо имен функций, узлы CSTG также помечаются уникальными идентификаторами вызова. Грани представляют собой вызовы между функциями.

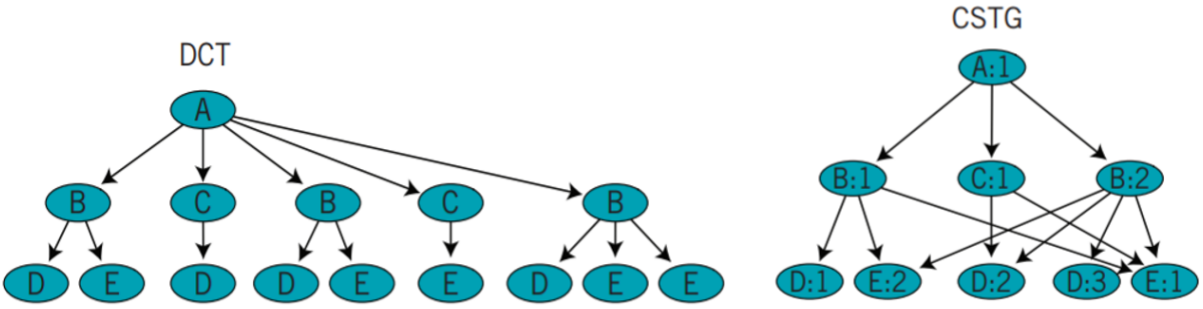


Рис. 4. CSTG

Пользователю необходимо вставить в интересующие функции вызовы cstg.addStackTrace(). Инструмент CSTG автоматически запускает тестируемый пример с использованием различных сценариев, создает графы и помогает пользователям увидеть существенные различия между сценариями. Сама ошибка обычно обнаруживается и подтверждается с помощью традиционного отладчика, при этом дельта-группы CSTG наводят на место возникновения ошибки.

Ориентированный на данные фреймворк для отладки параллельных приложений

Далее представлен материал, являющийся результатом анализа работы [10].

Проблема Обнаружение ошибок в крупномасштабных научных приложениях, работающих на сотнях тысяч вычислительных ядер.

Неэффективность параллельных отладчиков для отладки пета-масштабных приложений.

Предложенное решение В этом исследовании представлена реализация фреймворка для отладки, ориентированного на данные, в котором в качестве основы используются утверждения. Подход, ориентированный на данные можно использовать для повышения производительности параллельных отладчиков.

Метод, представленный в статье основан на параллельном отладчике Guard, который поддерживает тип утверждения во время отладки, называемые сравнительные утверждения.

Описание решения Утверждение – это утверждение о предполагаемом поведении компонента системы, которое должно быть проверено во время выполнения. В программировании программист определяет утверждение, чтобы гарантировать определенное состояние программы во время выполнения.

Пользователь делает заявления о содержимом структур данных, затем отладчик проверяет достоверность этих утверждений.

В следующем списке показаны примеры утверждений, которые можно использовать для обнаружения ошибок во время выполнения:

- «Содержимое этого массива всегда должно быть положительным».
- «Сумма содержимого этого массива всегда должна быть меньше постоянной границы».
- «Значение в этом скаляре всегда должно быть больше, чем значение в другом скаляре».
- «Содержимое этого массива всегда должно быть таким же, как содержимое другого массива».

В работе используются два новых шаблона утверждений помимо сравнительных: общие специальные утверждения и статистические утверждения.

Общие специальные утверждения позволяют использовать простые арифметические и булевы операции.

```
for (j = 0; j < n; j++) {
  for (i = 0; i < m; i++) {
    pold[j][i] = 50000.;
    pressure[j][i] = 50000.; }}
```

```
assert $a::pressure@"init.c":180 = 50000
```

Например, учитывая следующий фрагмент кода из исходного файла init.c и предполагая, что код вызывается с набором процессов $a, можно рассмотреть следующее утверждение:

Это утверждение гарантирует, что каждый элемент в структуре данных набора процессов $a равен 50 000 в строке 180 исходного файла init.c.

Соответственно, *сравнительные утверждения* позволяют сравнивать две отдельные структуры данных во время выполнения.

Следующее утверждение сравнивает данные из big_var в $a в строке 4300 исходного файла ref.c с large_var в $b в строке 4300 исходного файла sus.c.

```
assert $a::big_var@"ref.c":4300=$b::large_var@"sus.c":4300
```

Статистические утверждения - это определяемый пользователем предикаты, состоящие из двух моделей данных в форме либо статистических примитивов (средние значения, значения стандартного отклонения), либо функциональных моделей (гистограммы, функции плотности)

Статистические утверждения позволяют пользователю сравнивать информацию о шаблонах данных между двумя структурами данных, тогда как более ранние утверждения требовали сравнения точных значений.

Например, можно утверждать, что среднее значение большого набора данных находится между определенными границами до или после вызова функции или что количество элементов в массиве должно находиться в определенном диапазоне.

Определение степени и источников недетерминизма в приложениях MPI с помощью ядер графов

Далее представлен материал, являющийся результатом анализа работы [11].

Проблема Выявление причин сбоев воспроизводимости в приложениях на экзафлопсных платформах.

Крупномасштабные приложения MPI обычно принимают гибкие решения во время выполнения том порядке, в котором процессы обмениваются данными, чтобы улучшить свою производительность. Следовательно, недетерминированные коммуникативные модели стали особенностью этих научных приложений в системах высокопроизводительных вычислений.

Недетерминизм мешает разработчикам отслеживать вариации выполнения программы для отладки. Также сложно воспроизвести результаты при повторных запусках, что затрудняет доверие к научным результатам.

Предложенное решение Фреймворк для определения источников недетерминизма с использованием графов событий.

Описание решения Параллельное выполнение программы моделируется в виде ориентированных графов событий, ядра графов используются, чтобы охарактеризовать изменения межпроцессного взаимодействия от запуска к запуску. Ядра могут количественно определять тип и степень недетерминизма, присутствующего в моделях коммуникации MPI.

Фреймворк позволяет найти первопричины недетерминизма в исходном коде без знания коммуникативных паттернов приложения.

Платформа моделирует недетерминизм в следующие три этапа.

Этап первый. Сбор трассировки выполнения. Фреймворк фиксирует трассировку нескольких выполнений недетерминированного приложения с помощью двух модулей трассировки: CSMPI (фиксирует стек вызововов, связанных с вызовами функций MPI) и DUMPI (фиксирует порядок отправляемых и получаемых сообщений для каждого процесса MPI).

Для каждого выполнения приложения фреймворк генерирует один файл трассировки CSMPI и один файл трассировки DUMPI для каждого процесса MPI (или ранга). Файлы трассировки впоследствии загружаются конструктором графа событий для восстановления порядка сообщений выполнения.

Этап второй. Построение модели графа событий. На втором этапе фреймворк моделирует выполнение недетерминированного приложения в виде ориентированного ациклического графа (DAG), используя файлы трассировки, созданные на первом этапе.

Здесь вершины представляют собой связи point-to-point, такие как отправка и получение сообщения, а направленные ребра представляют отношения между этими событиями. Модели межпроцессного взаимодействия этой формы обычно называются графами событий.

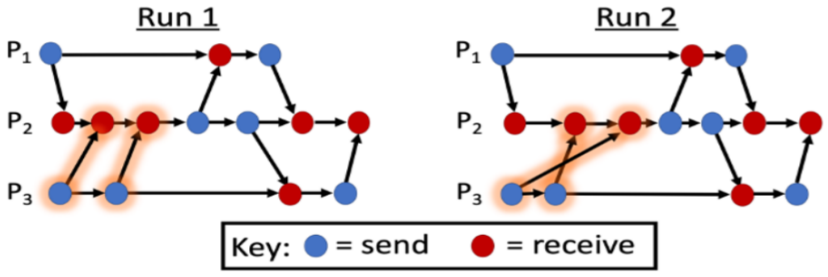


Рис. 5. DGA

Затем используются ядра графов для количественной оценки (несходства) графов событий, тем самым количественно оценивая степень проявления недетерминированности в приложении.

Ядра графов представляют собой семейство методов для измерения структурного сходства графов.

Простыми словами, ядро графа можно рассматривать как функцию, которая подсчитывает совпадающие подструктуры (например, поддеревья) двух входных графов, как показано на рис. 6, сопоставляя пары графов со скалярами, которые количественно определяют, насколько они похожи.

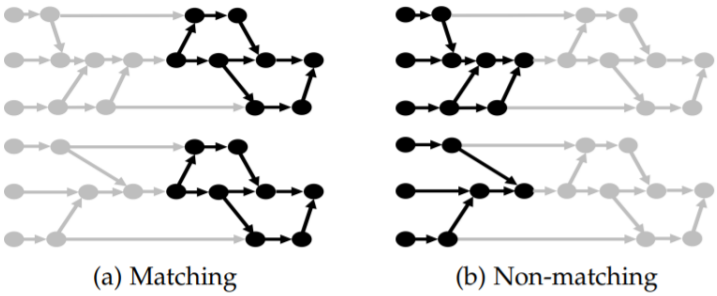


Рис. 6. Сравнение двух графов

Этап третий. Анализ графа событий. На последнем этапе анализ графа событий позволяет ученым количественно оценить недетерминированность многократного выполнения приложения MPI без каких-либо знаний о коммуникативных моделях приложения MPI.

Подготовлено: *Крехтунова Д.Д. (РК6-73Б), 2021.11.10*

2021.11.22: Web-инструменты для древовидного представления ассоциативных массивов, визуализация ассоциативных массивов (первичный обзор литературы)

Ассоциативные массивы

Ассоциативный массив – это массив, в котором обращение к значению осуществляется по ключу (того или иного типа).

Часто в качестве ключа используется не индекс, а строка, задаваемая программистом. Таким образом представить ассоциативный массив можно как набор пар “ключ-значение”. При этом каждое значение связано с определённым ключом.

Поддержка ассоциативных массивов есть во многих интерпретируемых языках программирования высокого уровня, таких, как Perl, PHP, Python, Ruby, Tcl, JavaScript и других.

Реализации ассоциативного массива Простейший способ хранения ассоциативных массивов – список пар (ключ, значение), где *ключ* определяет “индекс” элемента, а *значение* – значение элемента с этим индексом.

Наиболее популярными являются реализации ассоциативных массивов, основанные на различных деревьях поиска. Так, например, в стандартной библиотеке STL языка C++ контейнер *map*<*K*,*T*> реализован на основе красно-чёрного дерева [**?**]. В языках Java, Ruby, Tcl, Python используется один из вариантов хеш-таблицы [**?**]. Известны и другие реализации (ааа, ммм, ...).

Операции с деревом работают быстрее. При реализации на основе списков все функции требуют $O(n)$ операций, где n – количество элементов в рассматриваемой структуре. Операции над деревьями же требуют $O(h)$, где h – максимальная глубина дерева.

Данные в дереве хранятся в его вершинах. В программах вершины дерева обычно представляют структурой, хранящей данные и две ссылки на левого и правого сына.

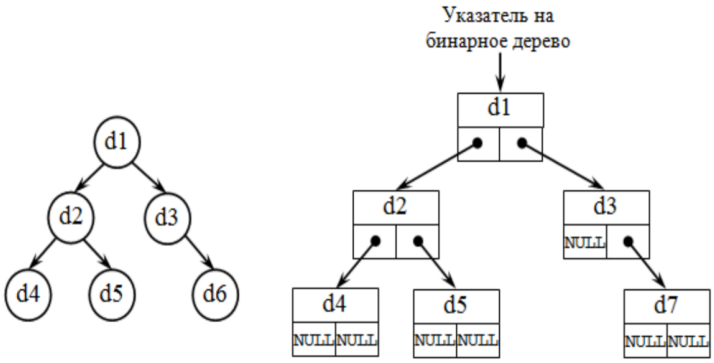


Рис. 7. Пример представления бинарного дерева

Набор библиотек TnT для web-визуализации деревьев и аннотаций на основе трассировок

В статье [12] представлен набор библиотек, предназначенных для web-визуализации деревьев и трассировок.

В статье говорится в первую очередь о визуализации биологических данных в веб-приложениях. Но представленные библиотеки не имеют узкой направленности и могут использоваться для разных целей.

Библиотеки TnT (англ. Trees and Tracks) предназначены для создания настраиваемых, динамических и интерактивных визуализаций деревьев и аннотаций на основе треков.

Библиотеки написаны на Javascript с использованием библиотеки D3, которая сама по себе является мощным инструментом визуализации данных. Она использует стандарты масштабируемой векторной графики (SVG), HTML и CSS.

Ниже приведено краткое описание библиотек TnT, непосредственно связанных с визуализацией деревьев.

– *TnT Tree*

Эта библиотека построена на основе кластерного расположения D3 (D3 cluster layout) и позволяет создавать динамические и интерактивные деревья. Он состоит из нескольких настраиваемых элементов: макета, определяющего общую форму дерева, узлов дерева в которых можно изменять форму, размер и цвет, ярлыков, состоящих из текста или изображений и данных для загрузки объектов Javascript или строк newick / nhx.2.2 TnT Tree Node.

PhyloCanvas - аналогичный проект, предлагающий средства для визуализации деревьев. В качестве основной технологии в нем используется Canvas. Но библиотеки TnT более универсальны и поддерживают интеграцию с другими библиотеками. Документацию и примеры для TnT Tree можно найти по ссылке http://tntvis.github.io/tnt.tree/.

– *TnT Tree Node*

Эта библиотека предоставляет методы для управления деревом на уровне данных и используется TnT Tree, хотя также может использоваться и независимо. Методы, включенные в TnT Tree Node, варьируются от вычисления наименьшего общего предка набора узлов до извлечения поддеревьев. Документацию для этой библиотеки можно найти как часть документации библиотеки TnT Tree.

Программное обеспечение Empress интерактивного и исследовательского анализа многомерных наборов данных на основе деревьев

В работе [13] представлен интерактивный web-инструмент EMPress для визуализации деревьев в контексте микробиома, метаболома и других областей. EMPress предоставляет широкие функциональные возможности, такие как анимация объектов, наряду со стандартными функциями визуализации дерева.

EMPress реализован в виде подключаемого модуля QIIME 2 (или отдельной программы Python, которую можно использовать вне QIIME 2), способной создавать HTML-документы с автономным пользовательским интерфейсом визуализации. Кодовая база состоит из компонента Python и компонента JavaScript. Кодовая база Python отвечает за проверку данных, предварительную обработку, фильтрацию и форматирование. Взаимодействие с пользователем, рендеринг и генерация рисунков обрабатываются базой кода JavaScript.

Кодовая база Python EMPress использует такие модули, как NumPy, SciPy, Pandas, Click, Jinja2, scikit-bio, формат BIOM и EMPeror.

В кодовой базе JavaScript используются Chroma.js, FileSaver.js, glMatrix, jQuery, Require.js, Spectrum, и Underscore.js.

Программный инструмент Treemap визуализации иерархических структур

В статье [14] представлена интерактивная версия Treemap.

В теории графов дерево – это особый тип графа, который связан и ацикличен [?]. Обычно деревья представляются визуально с помощью диаграмм узлов и связей

(древовидных диаграмм). В деревьях ребра присутствуют только между соседними слоями, и, следовательно, деревья наилучшим образом подходят для представления иерархических данных, для которых характерно расположение элементов на различных уровнях относительно друг друга: “ниже”, “выше” или “на одном уровне”.

Treemap - это метод визуализации иерархических структур. Используя этот метод, можно отобразить дерево с миллионами узлов в ограниченном пространстве. Основная идея, лежащая в основе этого метода визуализации, состоит в том, чтобы выделять прямоугольники для родительских узлов, а для дочерних узлов блоки (прямоугольники) в соответствующем родительском прямоугольнике.

Интерфейс создавался с использованием html / css с jQuery для обработки событий, связанных с кликами.Входными данными для интерфейса является дерево родительских указателей (parent pointer tree - структура данных N-арного дерева, в которой каждый узел имеет указатель на свой родительский узел, но не указывает на дочерние узлы).

Web-инструмент DoubleRecViz для визуализации согласования деревьев транскриптов и генов

В статье [15] представлен веб-инструмент для визуализации согласований между филогенетическими деревьями (деревья, отражающее эволюционные взаимосвязи между различными видами, имеющими общего предка).

В статье описан формат DoubleRecViz, основанный на формате phyloXML (XML, предназначенный для описания филогенетических деревьев и связанных с ними данных).

DoubleRecViz написан на Python и использует библиотеку Dash, которая предоставляет функции динамической визуализации веб-данных. Dash является связкой Flask, React.Js, HTML и CSS.

Приложения на Dash — веб-серверы, которые запускают Flask и связывают пакеты JSON через HTTP-запросы. Интерфейс Dash формирует компоненты, используя React.js.

Компоненты Dash — это классы Python, которые кодируют свойства и значения конкретного компонента React и упорядочиваются как JSON.

Полный набор HTML-тегов также обрабатывается с помощью React, а их классы Python доступны через библиотеку. CSS и стили по умолчанию хранятся вне базовой библиотеки, чтобы сохранить принцип модульности и независимого управления версиями.

Подготовлено: *Крехтунова Д.Д. (РК6-73Б), 2021.11.22*

2021.12.19: Концептуальная постановка задачи

Требуется реализовать web-ориентированный программный инструмент (далее *GBSE-отладчик*), обеспечивающий проведение отладки графовых реализации некоторых сложных вычислительных методов. GBSE-отладчик должен обеспечивать отслеживание текущих значений каждого отдельного элемента данных в состояниях, соответствующих узлам связанной графовой модели.

Цель разработки. Создать программный инструментарий (web-ориентированный), который бы позволил визуализировать значения отдельных элементов общих данных[16], остановив обработку на произвольном, выбираемом(ых) заранее, состоянии(ях) данных.

Назначение. Реализация задачи разработки GBSE-отладчика позволит отслеживать текущие значения отдельных элементов данных в произвольном состоянии данных (узле) при проведении текущего расчета, что упростит процесс разработки графоориентированных решателей.

Поставленные задачи (частичный перечень).

1. Провести обзор литературы по темам:

(a) “Автоматические методы отладки наукоемкого кода” (автоматизированные и автоматические методы отладки, приме- няемые при реализации сложных вычислительных методов (СВМ), отладка наукоёмкого кода, science code debugging, graph based programming);

(b) “Методы визуализации ассоциативных массивов” (web-инструменты для древовидного представления ассоциативных массивов, визуализация ассоциативных массивов).

2. Определить перечень поддерживаемых типов элементов данных СВМ, предложить и реализовать методы визуализации для каждого из них.

3. Разработать функцию на языке Python, позволяющую определять тип данных, хранящихся в каждом отдельном элементе ассоциативного массива (хранение данных осуществляется в объекте типа dict).

4. С использованием программного каркаса Django разработать программное обеспечение для визуализации объектов типа *dict* в древовидном виде.

Подготовлено: *Крехтунова Д.Д. (РК6-73Б), Соколов А.П. (РК6), 2021.12.19*

3 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 графиче-

ским 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

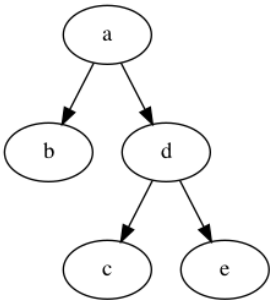
Вызов всех программ Graphviz осуществляется через командную строку, в процессе ознакомления с языком использовалась следующая команда:

dot −Tpng <pathToDotFile> −o <imageName>

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 [1], 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 [16]. Принципы применения графоориентированного подхода зафиксированы в патенте [3].

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

```
digraph G {
// Parallelism
  s1 [parallelism=threading]
// Graph definition
  __BEGIN__ ->s1
  s4 ->s6 [morphism=edge_1]
  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 (8) представлен этот граф, разбитый на уровни.

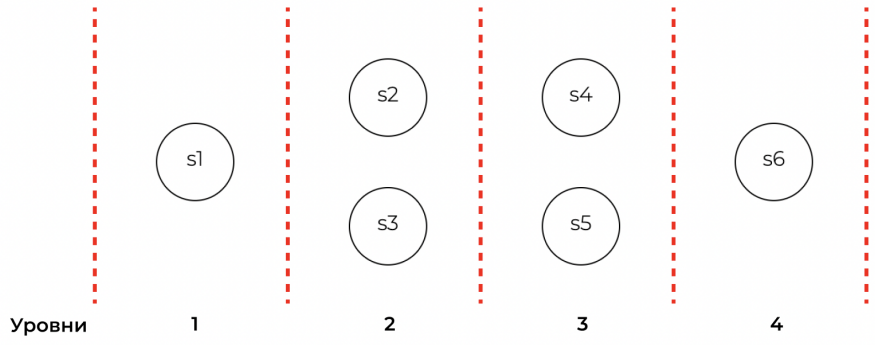


Рис. 8. Узлы графовой модели G, представленные на разных “уровнях”

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

```
1 digraph TEST
2 {
3 // Parallelism
4   s11 [parallelism=threading]
5   s4 [parallelism=threading]
6   s12 [parallelism=threading]
7   s15 [parallelism=threading]
8   s2 [parallelism=threading]
9   s8 [parallelism=threading]
10 // Functions
11   f1 [module=DEFAULT_VALUE, entry_func=DEFAULT_VALUE]
12 // Predicates
13   p1 [module=DEFAULT_VALUE, entry_func=DEFAULT_VALUE]
14 // Edges
15   edge_1 [predicate=p1, function=f1]
16 // Graph model description
17   __BEGIN__ ->s1
18   s6 ->s8 [morphism=edge_1]
19   s7 ->s8 [morphism=edge_1]
20   s10 ->s8 [morphism=edge_1]
21   s11 =>s8 [morphism=edge_1]
22   s11 =>s9 [morphism=edge_1]
23   s14 ->s9 [morphism=edge_1]
24   s3 ->s9 [morphism=edge_1]
25   s4 =>s6 [morphism=edge_1]
26   s4 =>s7 [morphism=edge_1]
27   s4 =>s10 [morphism=edge_1]
28   s4 =>s11 [morphism=edge_1]
29   s12 =>s4 [morphism=edge_1]
30   s12 =>s14 [morphism=edge_1]
31   s13 ->s3 [morphism=edge_1]
32   s15 =>s14 [morphism=edge_1]
33   s15 =>s14 [morphism=edge_1]
34   s2 =>s12 [morphism=edge_1]
35   s2 =>s13 [morphism=edge_1]
36   s2 =>s15 [morphism=edge_1]
37   s1 ->s2 [morphism=edge_1]
38   s8 =>s9 [morphism=edge_1]
39   s8 =>s6 [morphism=edge_1]
40   s9 -> __END__
41 }
```

В результате визуализации модели будет получен граф, представленный на рисунке (9).

Основная задача алгоритма – это отрисовать вершины графа в корректных пози-

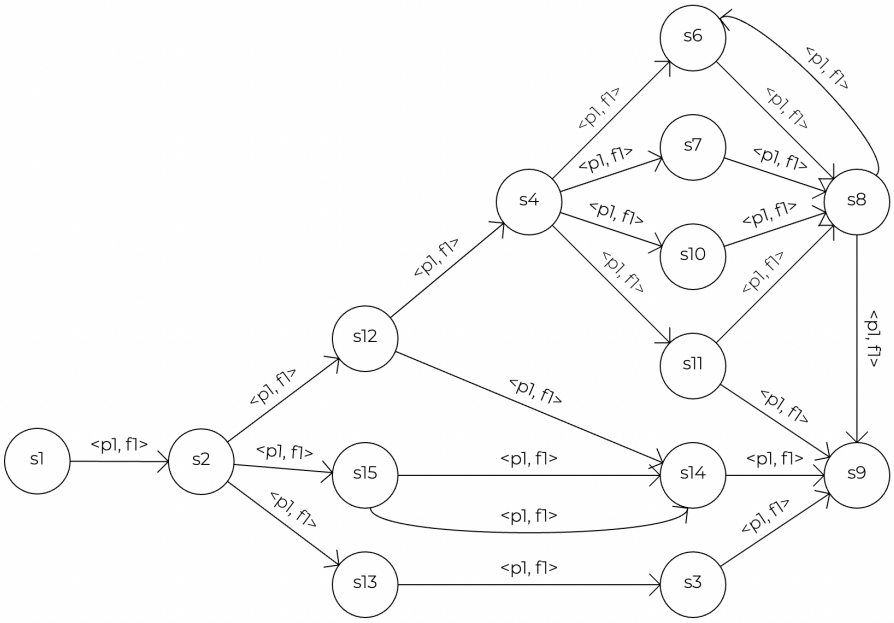


Рис. 9. Визуализация графовой модели TEST

циях, 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"]}],
"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 (10) представлен граф с выделенным уровнем №5.

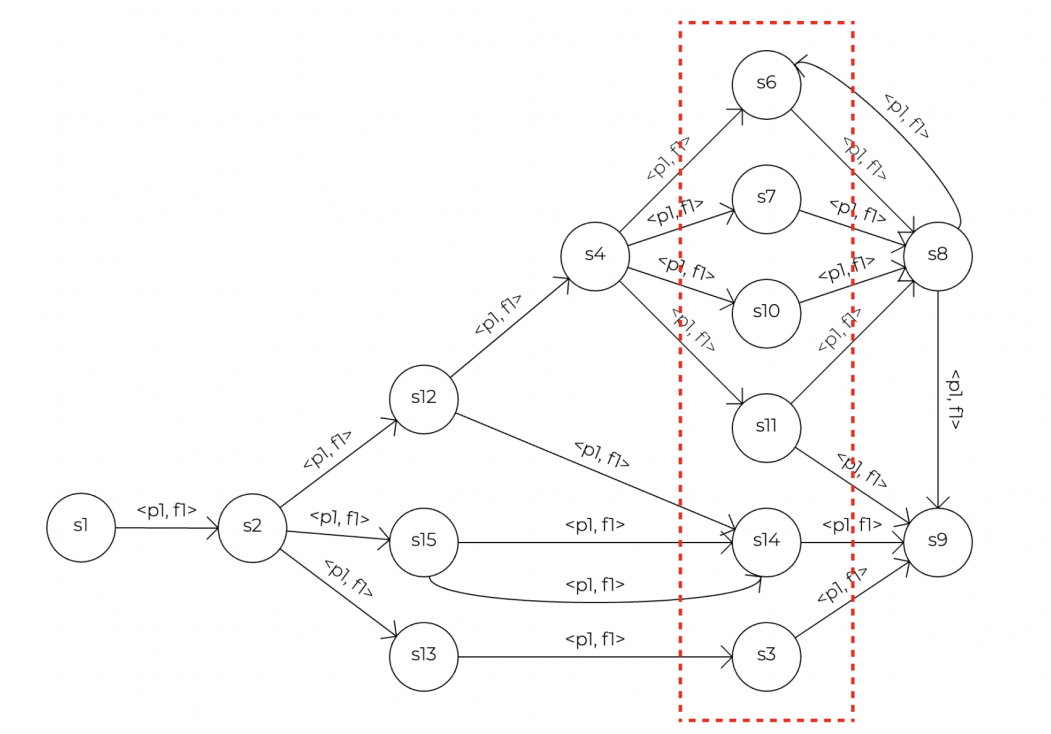


Рис. 10. Уровень №5 на графе

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 темой 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 11.

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.

Рис. 11. Создание вершины

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.

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 12.

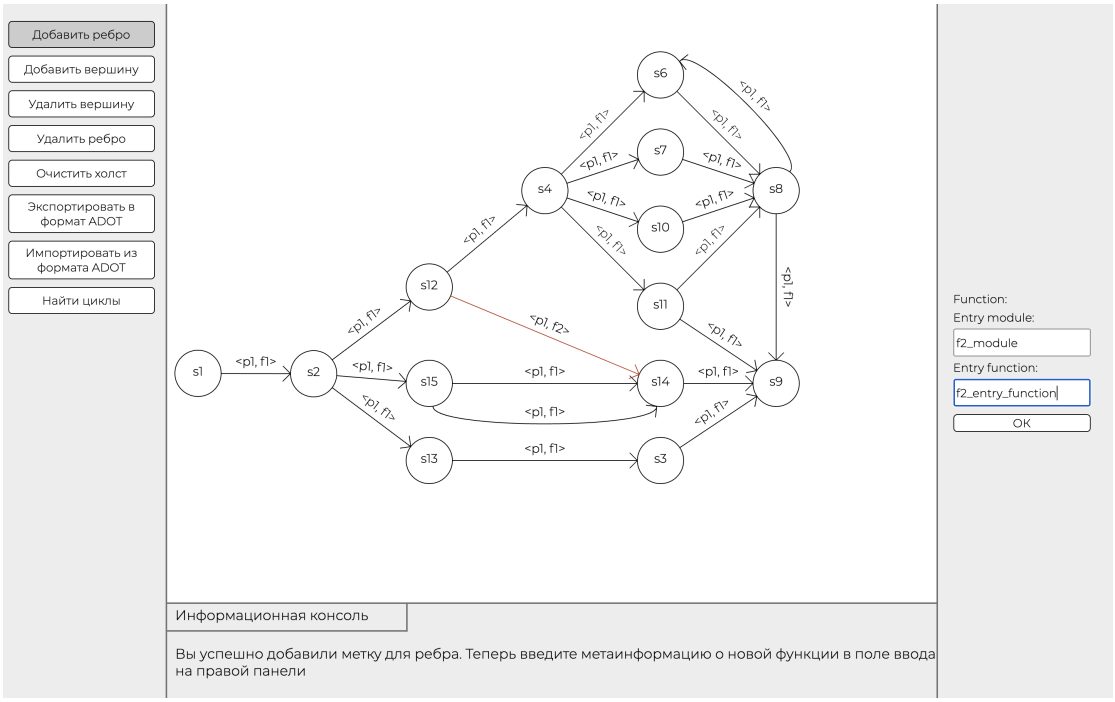


Рис. 12. Создание ребра

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 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.

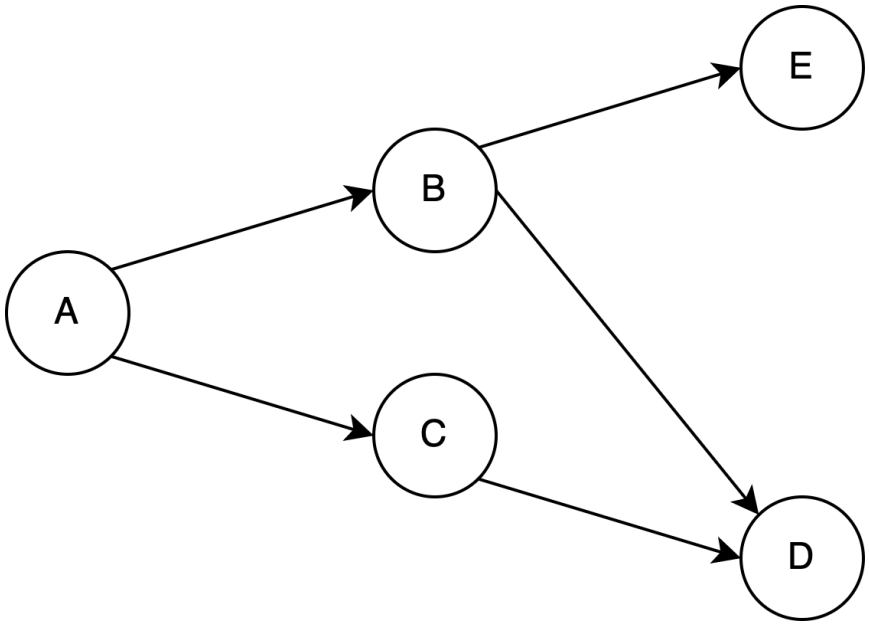
Подготовлено: *Ершов В. (РК6-72Б), 2021.12.05*

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

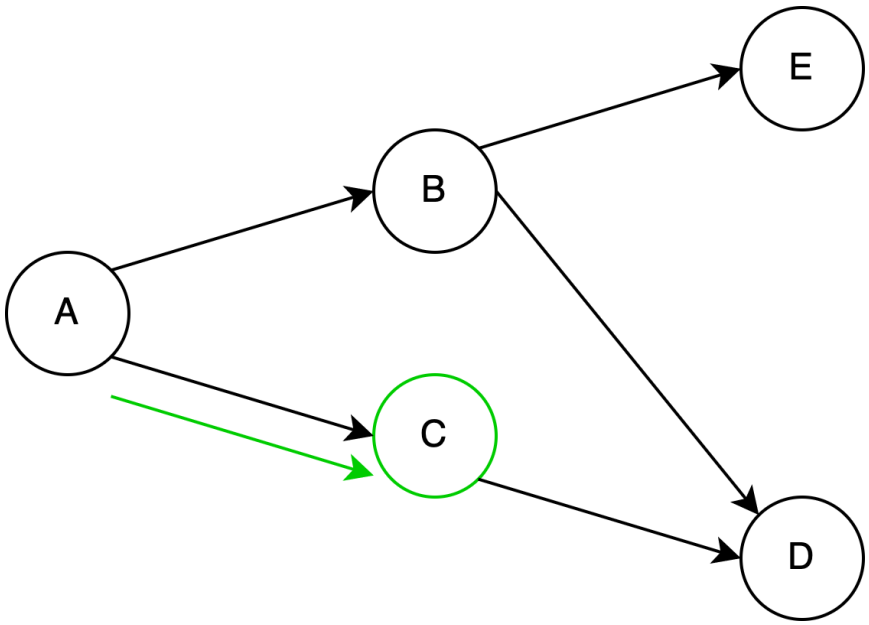
Для поиска циклов в ориентированном графе необходим алгоритм обхода графа. Обход графа - это переход от одной вершины графа к другой с целью поиска ребер или вершин, которые удовлетворяют некоторому условию.

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.

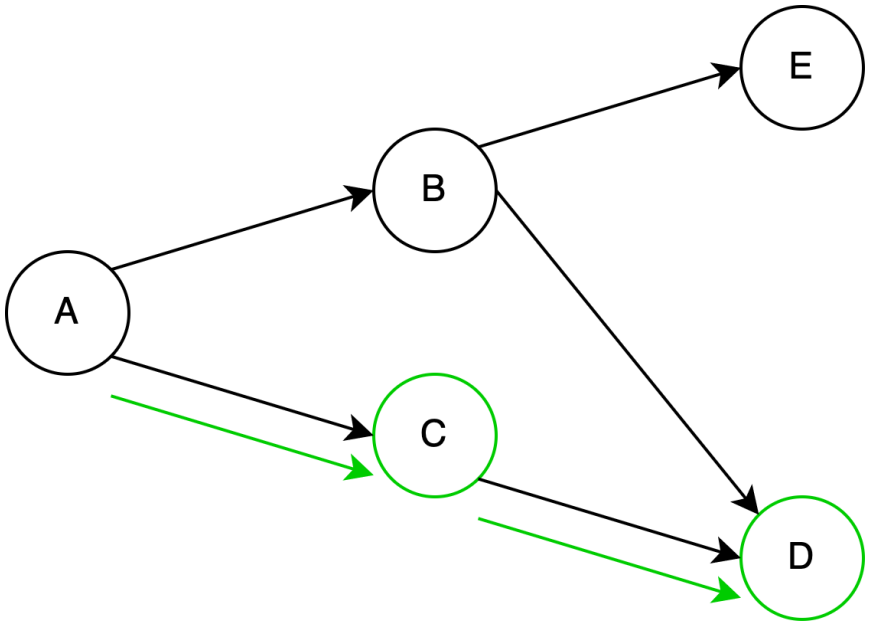
Как ранее было сказано, алгоритм DFS двигается от начальной вершины до тех пор пока не будет достигнута конечная вершина. Если был достигнут конец пути, но искомая вершина так и не была найдена, то необходимо вернуться назад (к точке разветвления) и пойти по другому маршруту. Рассмотрим работу алгоритма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



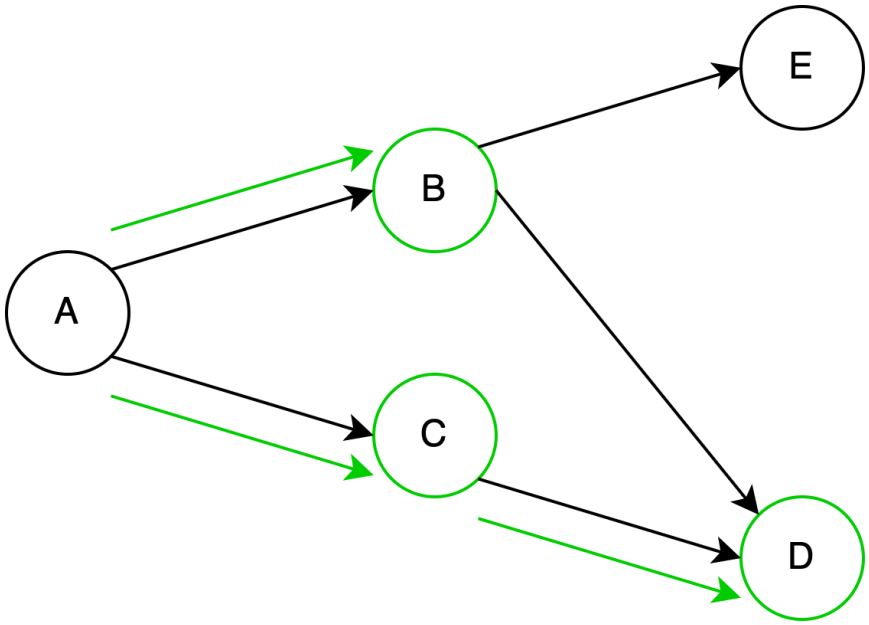
Мы находимся в точке “A” и хотим найти вершину “E”. Согласно принципу DFS, необходимо исследовать один из возможных маршрутов до конца, если не будет обнаружена вершина “E”, то возвращаемся и исследуем другой маршрут.



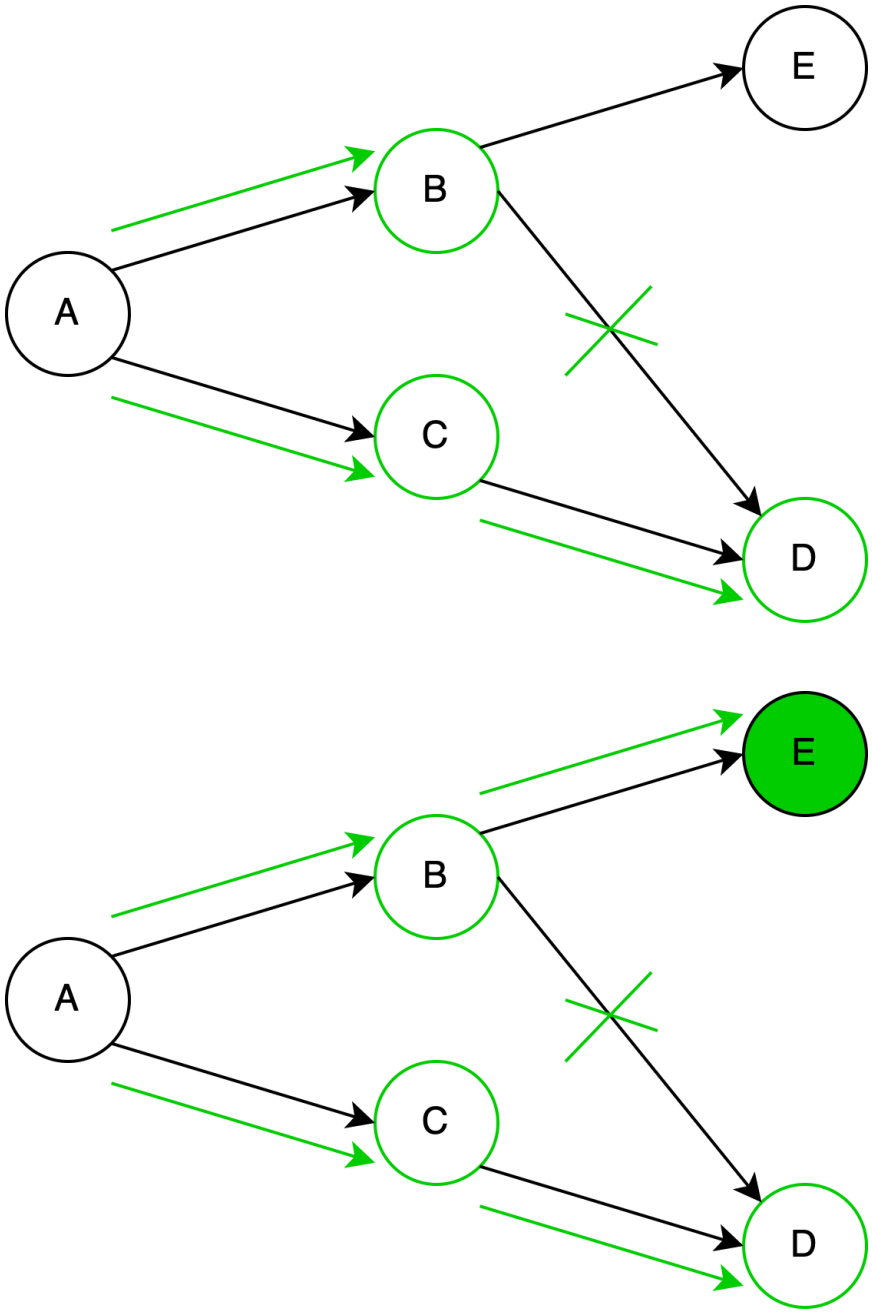
В данном случае мы двигаемся к ближайшей вершине “C”, поскольку это не конец пути, то переходим к следующей вершине.



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

```
void FindCycles(const std::vector<int> &adjacency_list)
{
    std::vector<std::string> visited(adjacency_list.size(), "NOT_VISITED");

    for (int vertex = 0; vertex < adjacency_list.size(); ++vertex)
    {
        if (visited[vertex] == "NOT_VISITED")
        {
            std::stack<int> stack;
            stack.push(vertex);
            visited[vertex] = "IN_STACK";
            processDFS(adjacency_list, visited, stack);
        }
    }
}

void processDFS(const std::vector<int> &adjacency_list,
                std::vector<std::string> &visited,
                std::stack<int> &stack)
{
    for (const int &vertex : adjacency_list[stack.top()])
    {
        if (visited[vertex] == "IN_STACK")
        {
            printCycle(stack, vertex);
        }
        else if (visited[vertex] == "NOT_VISITED")
        {
            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*

4 Реализация поддержки различных стратегий распараллеливания в рамках графоориентированного программного каркаса

2021.11.14: Известные алгоритмы поиска циклов в ориентированных графах

Существует несколько групп алгоритмов поиска[17]:

1) алгоритмы, основанные на обходе ориентированного графа (ОГ);

2) алгоритмы, основанные на использовании матриц смежности, описывающих конкретный граф.

Одним из представителей алгоритмов первой группы (обхода ОГ) является алгоритм поиска в глубину (англ., depth-first-search (DFS)), который предполагает, что обход осуществляется из фиксированного и заранее заданного узла. Если число узлов ОГ равно n, то сложность алгоритма DFS $O(n^2)$ и $O(n^3)$ – для случая "многократного обхода" из каждого узла[2]. Поэтому для ускорения алгоритма необходимо применение модификаций, например распараллеливание алгоритма поиска в глубину [18].

Зададим граф $G(V, E)$, где $V = \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \ldots\}$ – множество вершин, $|V| = n$, а $E = \{\alpha\beta : \alpha\beta \in V \times V\}$ – множество рёбер (пример, $\{\boldsymbol{ab}, \boldsymbol{ad}, \ldots, \boldsymbol{ge}\}$).

Для определения ОГ, часто, применяют понятие матриц смежности[19]:

$$\|A\| = \|a_{ij}\|_{n\times n};$$
$$a_{ij} = \begin{cases} 1, & \text{– определено ребро с началом в узле } i \text{ и концом в узле } j; \\ 0, & \text{– ребро не определено.} \end{cases} \tag{1}$$

$$A = \begin{bmatrix} & \boldsymbol{a} & \boldsymbol{b} & \boldsymbol{c} & \boldsymbol{d} & \boldsymbol{e} & \boldsymbol{f} & \boldsymbol{g} \\ \boldsymbol{a} & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ \boldsymbol{b} & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ \boldsymbol{c} & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \boldsymbol{d} & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ \boldsymbol{e} & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \boldsymbol{f} & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \boldsymbol{g} & 0 & 0 & 1 & 1 & 1 & 01 & 0 \end{bmatrix} \tag{2}$$

Пример графа, построенного на основе его матрицы смежности A, приведен на рис. 13.

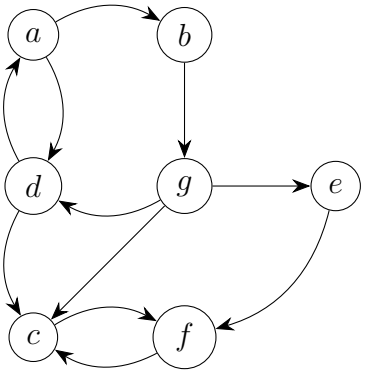


Рис. 13. Граф, построенный по матрице смежности A

Элемент матрицы смежности a_{ij} указывает на факт наличия определённого рёбра с началом в узле i и концом в узле j, тогда как транспонированная матрица A^T задаёт

[2]Для случая ОГ возможность идентификации всех циклов требует осуществления многократных обходов, начиная с каждого из узлов ОГ.

новый ОГ с теми же узлами и рёбрами, противоположно направленными относительно исходного ОГ.

$$A^T = \begin{bmatrix} & \boldsymbol{a} & \boldsymbol{b} & \boldsymbol{c} & \boldsymbol{d} & \boldsymbol{e} & \boldsymbol{f} & \boldsymbol{g} \\ \boldsymbol{a} & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \boldsymbol{b} & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \boldsymbol{c} & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ \boldsymbol{d} & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ \boldsymbol{e} & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ \boldsymbol{f} & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ \boldsymbol{g} & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3}$$

Введём обозначение матрицы, возведённой в степень m:

$$A^m = \underbrace{A \times A \times \ldots \times A}_{m}. \tag{4}$$

Матрица A^m обладает следующим свойством: элемент матрицы a^i_j равен числу путей из вершины i в вершину j, состоящих ровно из m рёбер[19]. Длиной цикла называют количстево рёбер в этом цикле. Для того, чтобы определить циклы длины $2m$, нужно найти матрицу циклов C_m, определяемую следующим образом:

$$C_m = A^m \wedge (A^T)^m; \tag{5}$$

где $\wedge$ – оператор конъюнкции, применяемый поэлементно.

Таким образом, матрица C_m будет содержать все циклы длины $2m$. Пример для поиска циклов длины 2. 6.

$$C_2 = A \wedge A^T = \begin{bmatrix} & \boldsymbol{a} & \boldsymbol{b} & \boldsymbol{c} & \boldsymbol{d} & \boldsymbol{e} & \boldsymbol{f} & \boldsymbol{g} \\ \boldsymbol{a} & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ \boldsymbol{b} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \boldsymbol{c} & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \boldsymbol{d} & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \boldsymbol{e} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \boldsymbol{f} & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \boldsymbol{g} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{6}$$

Подготовлено: *Муха В. (РК6-73Б), 2021.11.14*

2023.01.07: Задача параллельного обхода ориентированного графа с циклами

Введение в предметную область

На сегодняшний день существует класс прикладного программного обеспечения (ПО), направленного на автоматизацию подготовки и проведения вычислительных экспериментов. К такому ПО, среди прочего, относятся так называемые научные системы

управления потоком задач (англ. scientific workflow systems). Данные системы предполагают описание вычислительного эксперимента в виде набора вычислительных задач, которые должны быть решены, и информации о последовательности их выполнения. Наиболее распространены системы, где такой информацией служат зависимости по данным между задачами. Под зависимостью по данным понимается ситуация, когда выходные данные решения одной задачи являются входными для решения другой.

По списку задач и зависимостей между ними автоматически определяется очерёдность их решения, и система автоматически осуществляет решение задач в установленной последовательности. Такой подход даёт следующие преимущества.

1. Становится возможным разделение обязанности по разработке программных реализаций решений отдельных задач между разработчиками.
2. Становится возможным переиспользовать разработанные программные решений вычислительных задач в других вычислительных экспериментах.
3. Становится возможным создание базы программных реализаций решений типовых задач.

Кроме того, в зависимости от реализации системы, для проведения того или иного вычислительного эксперимента могут быть задействованы различные вычислительные мощности: от нескольких потоков процессора на одном персональном компьютере до тысяч узлов на вычислительном кластере. Для задач, легко поддающихся распараллеливанию и масштабированию, научные системы управления потоком задач дают универсальный интерфейс взаимодействия с различными вычислительными ресурсами, например при помощи процедуры «отображения»[20], что даёт возможность проводить вычислительные эксперименты на машинах с самыми различными архитектурами.

В основе работы научных систем управления потоком задач лежит представление списка задач и зависимостей между ними в виде ориентированного графа. Некоторые системы, такие как DAGMan[21] и VisTrails[22], поддерживают только ациклические графы. Другие, как, например, pSeven[23], имеют специализированные встроенные средства для обработки циклов в ориентированном графе. Для выполнения задач в порядке, определяемым ориентированным графом, необходимо произвести его обход. Как правило, для повышения производительности в научных системах управления потоком задач предусмотрено одновременное выполнение задач, не имеющих между собой зависимостей по данным. Таким образом, возникает дополнительное требование выполнять обход графа в параллельном режиме там, где это возможно.

Применение стандартных алгоритмов на графах

В задаче параллельного обхода ориентированного графа могут быть применены некоторые стандартные алгоритмы на графах.

Так для алгоритма обхода может быть важна проверка на наличие в входном графе циклов, поскольку в зависимости от её результата для обхода могут быть выбраны, например, алгоритмы предназначенные только для ациклических графов. Для поиска циклов применяется так называемый поиск в глубину. Его псевдокод приведён в

листинге 1

Algorithm 1 Поиск в глубину

```
1:  procedure DFS_Visit(u)
2:      color[u] := GRAY
3:      time := time + 1
4:      d[u] := time                          ▷ Устанавливаем момент входа в вершину
5:      for each v ∈ Adj[u] do
6:          if color[v] = WHITE then
7:              DFS_Visit(v)
8:          end if
9:      end for
10:     color[u] = BLACK
11:     time := time + 1
12:     f[u] := time                          ▷ Устанавливаем момент выхода из вершины
13: end procedure
```

Доказано[24], что в ориентированном графе нет циклов, если при поиске в глубину в нём не обнаружено ни одного обратного ребра, т.е. по завершении поиска в глубину нет такой пары вершин u и v, что $d[u] < d[v]$ и $f[u] > f[v]$, где $d[u]$ – временная метка входа в вершину, а $f[u]$ – временная метка выхода из неё.

Для определения очерёдности посещения вершин в графе может быть применена топологическая сортировка. Цель топологической сортировки заключается в том, чтобы упорядочить вершины ациклического ориентированного графа таким образом, чтобы каждая вершина была посещена до того, как будут обработаны все вершины, на которые она указывает [25].

Подготовлено: *Тришин И.В. (РК6-11М), 2023.01.07*

2023.01.22: Анализ предыдущих реализаций алгоритма обхода

Алгоритм обхода графовой модели в pycomsdk

Фреймворк pycomsdk предоставляет пользователю программный интерфейс для создания и интерпретации графовых моделей сложных вычислительных методов, созданных по методологии ГПИ, на языке python. В нём в процессе обхода графовой модели задействуются следующие сущности:

1) узел графовой модели, соответствующий состоянию данных СВМ (State);

2) ребро графовой модели, соответствующее функции перехода между состояниями (Transfer);

3) объект, связывающий ключи в ассоциативном массиве данных (keys_mapping);

4) объект, отвечающий за стратегию выполнения функций перехода (parallelization_policy).

Функции перехода в ГПИ описываются парой $t =< f, p >$, где f – функция-обработчик, p – функция-предикат. В pycomsdk для каждого узла графовой модели S_i хранится список исходящих из него рёбер $T(S_i)$, опционально ссылка на функцию-селектор[26] h_i и ссылка на подграф G_i. Для каждого ребра e_{ij} хранится вершина, в которую оно входит S_j. Кроме того, для каждого узла сохраняется предыдущая история обхода H_i, количество необходимых вхождений в неё N_i^r, количество совершённых вхождений N_i^c и количество циклов, содержащих данную вершину $L(S_i)$.

В общем виде, без учёта процесса распределения функций перехода между вычислительными ресурсами, процесс обхода графовой модели можно разделить на два этапа:

1) инициализация графовой модели (листинг 2);
2) выполнение функций перехода (листинг 3).

Algorithm 2 Алгоритм инициализации графовой модели

```
1:  function IdleRun(S_i, H)                ▷ H –текущая накопленная история обхода
2:      if G_i ≠ ∅ then
3:          IdleRun(S_0^{G_i}, H)           ▷ Запуск обхода из начальной вершины подграфа G_i
4:      end if
5:      N_i^r := N_i^r + 1
6:      if N_i^r ≠ 1 then
7:          if H_i ⊂ H then                 ▷ Проверка на нахождение в цикле
8:              L(S_i) := L(S_i) + 1
9:          end if
10:         End
11:     end if
12:     if H_i = ∅ then
13:         H_i := H
14:     end if
15:     if |T(S_i)| = 1 then
16:         S_j : e_ij ∈ T(S_i)
17:         IdleRun(S_j, H)
18:     else
19:         for each e_ij ∈ T(S_i) do
20:             S_j : e_ij ∈ T(S_i)
21:             H_j := H ∪ {S_j}
22:             IdleRun(S_j, H_j)
23:         end for
24:     end if
25: end function
```

Можно заметить, что алгоритм инициализации представляет собой модифицированный поиск в глубину.

Algorithm 3 Основной алгоритм обхода

1: **function** RUN(D) ▷ S – начальное состояние, D – данные
2: $S = S_0$
3: **while** $S \neq \varnothing$ **do** ▷ Пока не достигнуто конечное состояние
4: F = RunState(S, D) ▷ Получить функцию(-ции) перехода для текущего состояния
5: $S = F(D)$ ▷ Выполнить функции перехода и получить новое состояние
6: **end while**
7: **end function**
8: **function** RUNSTATE(S_i, D)
9: **if** $G_i \neq \varnothing$ **then**
10: RunState($S_0^{G_i}, D$)
11: **end if**
12: $N_i^c := N_i^c + 1$
13: **if** $N_i^c \neq N_i^r$ **then** ▷ Если не все необходимые переходы ещё совершены
14: **return**
15: **end if**
16: **if** $|T(S_i)| = 0$ **then** ▷ Если из текущего узла нет исходящих рёбер
17: **return**
18: **end if**
19: $U = h_i(D)$ ▷ Получить набор флагов, разрешающих или запрещающих переходы по исходящим рёбрам
20: $T = \varnothing$ ▷ Инициализировать набор ф-ций перехода, запланированных к выполнению
21: **for each** $u_j \in U$ **do**
22: **if** $u_j = true$ **and** $p_j(D) = true$ **then** ▷ Если переход разрешён селектором и проверка предикатом прошла успешно
23: $T := T \cup \{t_j\}, t_j \in T(S_i)$
24: **end if**
25: **end for**
26: F = makeTransfer(parallelization_policy, T) ▷ Получить функции перехода согласно стратегии их выполнения
27: **return** F
28: **end function**

Необходимо отметить, что алгоритм упомянутой функции makeTransfer зависит от стратегии выполнения рёбер, поэтому не может быть приведён в общем виде.

В приведённом алгоритме можно выделить следующие особенности.

1. Функция перехода планируется к выполнению только при истинности соответствующего предиката. Это позволяет избежать лишних проверок и повышает производительность.

2. Для проверки принадлежности вершины к циклу в ориентированном графе используется история предыдущих посещений, что требует хранения этой истории для

каждой из вершин.

3. Используются рекурсивные вызовы функции обхода, что может затруднить отладку алгоритма.

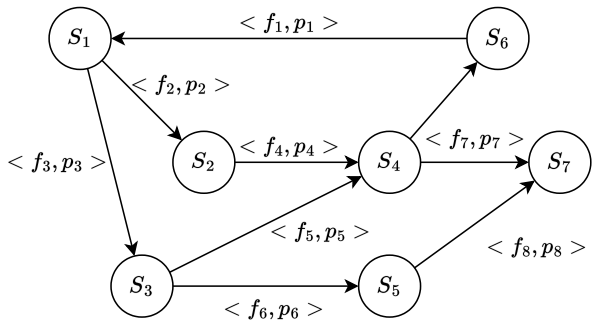
4. Решение о том, какие переходы совершать далее, принимается непосредственно при входе в вершину.

5. Не предусмотрено параллельное исполнение функций перехода.

Алгоритм обхода в новой версии comsdk

Фреймворк comsdk является ещё одной, исторически более ранней, реализацией программных средств, позволяющих пользователю создавать и автоматически интерпретировать графовые модели сложных вычислительных методов. Данный фреймворк реализован на языке C++. В рамках ВКР бакалавра над данным фрейморком велась работа по обновлению некоторых его компонентов, среди которых так же находится интерпретатор графовых моделей. В рамках ВКР для интерпретатора был разработан обновлённый алгоритм, поддерживающий возможность задействовать различные вычислительные ресурсы для выполнения функций перехода.

В текущей реализации comsdk графовая модель представляется набором узлов – состояний данных S_i, рёбер – функций перехода t_j – и матрицей смежности A. Пример графовой модели с её матрицей смежности представлен на рисунке 14.



	S_1	S_2	S_3	S_4	S_5	S_6	S_7
S_1	0	1	1	0	0	0	0
S_2	0	0	0	1	0	0	0
S_3	0	0	0	1	1	0	0
S_4	0	0	0	0	0	1	1
S_5	0	0	0	0	0	0	1
S_6	0	0	0	0	0	0	0
S_7	0	0	0	0	0	0	0

Рис. 14. Графовая модель с матрицей смежности

С каждым узлом S_i связывается только стратегия выполнения исходящих из него ветвей графовой модели и опционально селектор h_i. Для данного алгоритма были разработаны две абстрактные управляющие структуры – контейнер выполнения и ветвь выполнения. Назначение первой – отслеживание выполнения ветвей и управление задействованными вычислительными ресурсами. Задача второй – последовательный обход одной ветви графовой модели, в которой гаратировано отсутствие ветвления. Алгоритмы обхода одной ветви и работы контейнера выполнения представлены на рисунках 15 и 16.

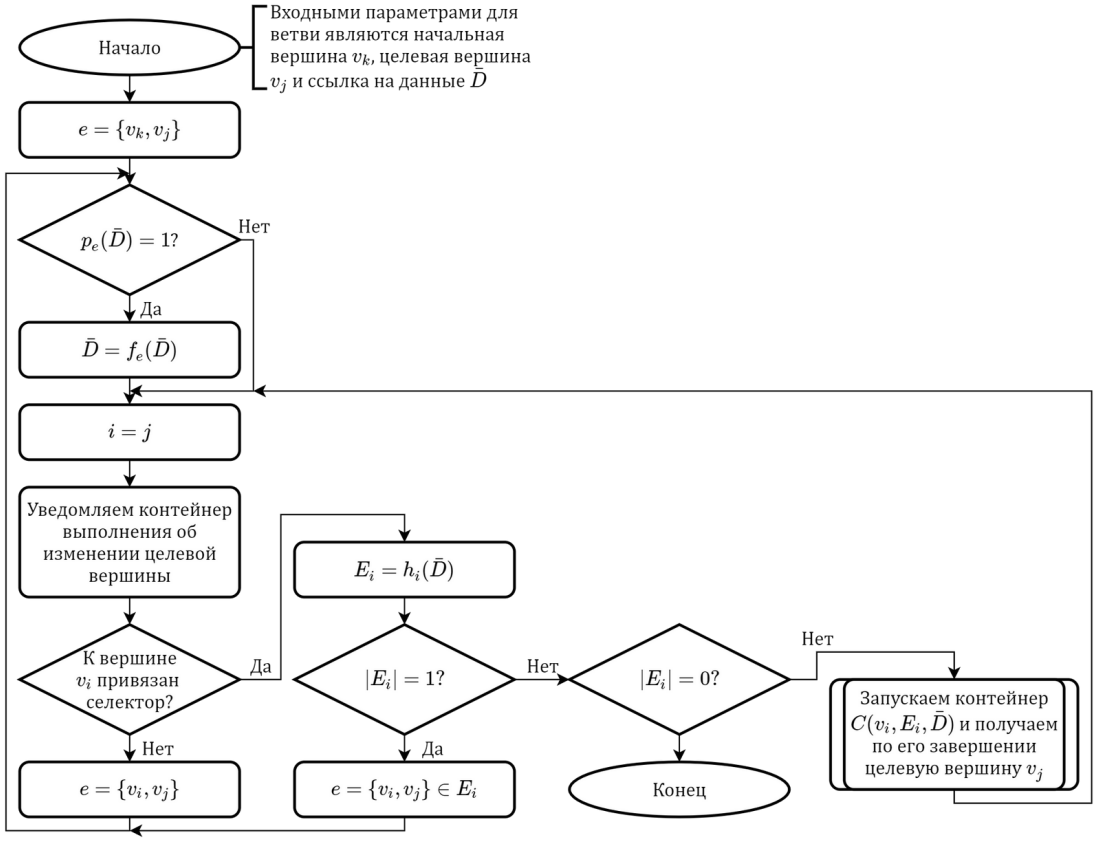


Рис. 15. Блок-схема алгоритма обхода одной ветви графовой модели

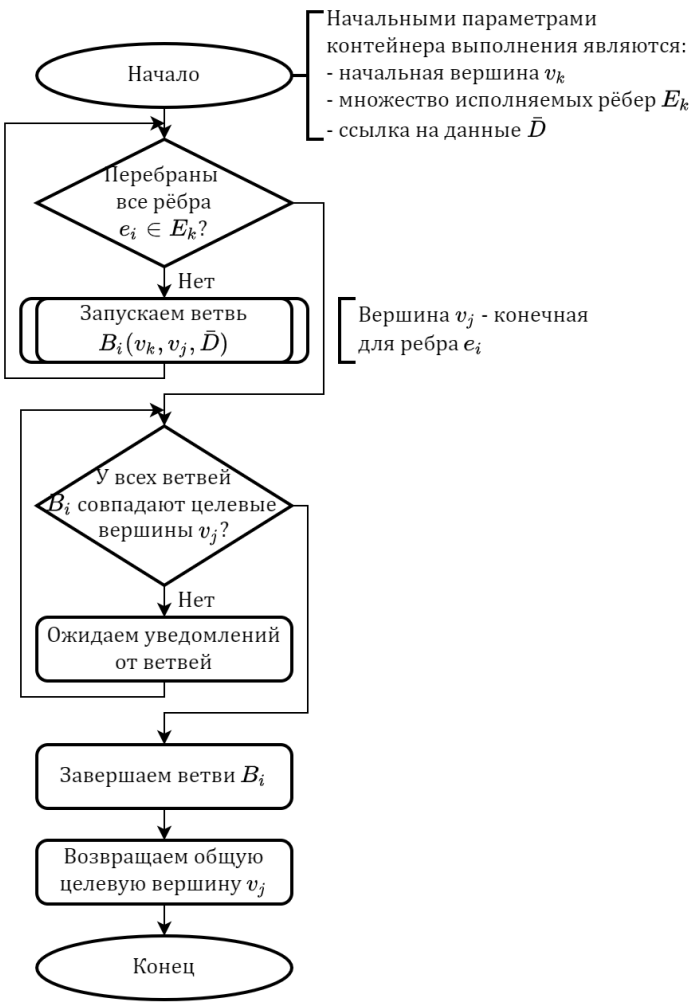


Рис. 16. Блок-схема алгоритма работы контейнера выполнения для обхода нескольких ветвей графовой модели

У разработанного алгоритма можно выделить следующие достоинства и недостатки.

1. Структура контейнера даёт разработчику достаточно высокий уровень абстракции при разработке алгоритма обхода. Все особенности взаимодействия с вычислительными ресурсами закладываются в функции запуска ветвей и уведомления контейнера, интерфейс сохраняется общий. Для задействования различных ресурсов достаточно сменить реализацию контейнера.

2. Контейнеры и ветви создаются рекурсивно, что может создать сложности при отладке.

3. Вычислительные ресурсы для параллельного выделяются и освобождаются в режиме реального времени, что снижает производительность;

4. Имеет место большое количество операций обмена данных между потоками обработки.

5. Алгоритм не устойчив к наличию в графовой модели циклов и в худшем случае

может уйти в бесконечную рекурсию.

Подготовлено: *Тришин И.В. (РК6-11М), 2023.01.22*

2023.01.23: Возможные модификации алгоритма обхода

Формулировка задачи

Решение задачи обхода графовой модели и выполнения функций перехода в описанной ориентированным графом последовательности может быть переформулирована, как задача о распределении заданий между несколькими потоками обработки (англ. worker threads) (при параллельном обходе) или определения очерёдности выполнения задач в одном потоке (при последовательном обходе). В данном контексте под потоками обработки подразумеваются любые аппаратные или программные средства выполнения инструкций: потоки процессора, процессы в операционной системе, узлы распределённого вычислительного кластера и т.п.

Дано:

1) набор функций перехода, которые необходимо выполнить, $T = \{t_i = < f_i, p_i >\}_{i=1}^{n}$, где f_i и p_i – соответствующие функции-обработчики и функции предикаты;

2) набор потоков обработки $W = \{w_j\}_{j=1}^{m}$;

3) ориентированный граф, показывающий порядок переходов между состояниями данных S_k сложного вычислительного метода $G =< V, E >$;

4) данные D.

При этом

1) узлам из V ставятся в соответствие тройка $< S_k, h_k, G_k >$, где h_k – ссылка на функцию-селектор, необходимую для проверки условий при достижении данного узла, а G_k – ссылка на подграф, который требуется обойти, прежде, чем совершать переходы из состояния S_k;

2) рёбрам из E ставятся в соответствие функции перехода из T.

Требуется:

1) совершить переход из начального состояния данных в конечное, выполняя функции перехода;

2) при достижении каждой вершины v_i, если в ней определена функция-селектор $h_i \neq \varnothing$, то перед дальнейшими переходами совершать её вызов и определять рёбра, по которым разрешён переход;

3) если из состояния S_i требуется совершить только один переход, выполнять его в том же потоке обработки, что и функцию перехода, приведшую в S_i;

4) в параллельном режиме если из вершины v_i выходит несколько рёбер графа и после вызова функции-селектора h_i оказывается разрешённым переход по нескольким рёбрам сразу, то выполнять соответствующие им функции перехода в различных потоках обработки;

5) если в вершину v_i входит несколько рёбер графа, совершать переход из неё только после выполнения функций перехода, связанных со всеми входящими рёбрами;

6) если в вершину v_i входит несколько рёбер графа e_k и потоки обработки, использованные для выполнения функций перехода t_k, не используют общую память, агрегировать полученные в результате их выполнения данные перед дальнейшими переходами;

7) учесть возможность наличия в графе G циклов;

Предложенные решения

Для определения числа потоков обработки, необходимого для обхода графа G с максимальной степенью распараллеливания предложено использовать поиск в ширину с учётом максимального размера очереди. За основу был взят алгоритм поиска в ширину, описанный в [24].

Algorithm 4 Поиск в ширину на ориентированном графе с заданной начальной вершиной

```
1:  procedure BFS(G)
2:      for each v_i ∈ V do
3:          color[v_i] := WHITE
4:      end for
5:      Пусть m = 1                          ▷ Максимальный размер очереди
6:      color[v_0] := GRAY
7:      Пусть Q := ∅ – очередь
8:      push(Q, v_0)
9:      while Q ≠ ∅ do
10:         u = pop(Q)
11:         for each v : ∃edge(u, v) do
12:             if color[v] = WHITE then
13:                 color[v] := GRAY
14:                 push(Q, v)
15:             end if
16:         end for
17:         color[u] := BLACK
18:         if size(Q) > m then
19:             m := size(Q)
20:         end if
21:     end while
22: end procedure
```

Данная процедура может быть проведена перед началом обхода, поэтому необходимые вычислительные ресурсы могут быть выделены заранее.

Для выполнения требования **??** предложено ввести для каждого потока обработки w_i очередь функций перехода, которые данный поток должен выполнить Q_i. Тогда при

необходимости перехода по нескольким рёбрам одновременно предложено помещать соответствующие функции перехода в очереди доступных потоков. Тогда, пока очередь пуста, поток обработки будет ждать поступления функций для выполнения.

Для выполнения требований 5) и 6) необходимо находить вершины, в которые входит несколько рёбер. Самый простой способ это сделать – использовать матрицу смежности графа G. Тогда для вершины v_i информацию о количестве входящих рёбер можно получить из i-того столбца матрицы смежности. Данная информация может быть получена до начала обхода графовой модели.

Для выполнения требования 5) при использовании нескольких потоков обработки необходимо предусмотреть некоторый аналог барьерной синхронизации в вершине, в которую входит несколько ветвей. Благодаря сделанному ранее выводу, можно определить необходимый «размер» барьера из матрицы смежности графа G. Данные барьеры могут быть подготовлены до начала обхода графовой модели.

Кроме того, пусть в вершину v входят рёбра $v_1, \ldots, v_k$, функции перехода которых выполнялись в потоках $w_1, \ldots, w_k$, и из вершины v выходит только одно ребро. Тогда для удовлетворения требования 3) предложено явно продолжить выполнение в потоке w_1.

Подготовлено: *Тришин И.В. (РК6-11М), 2023.01.23*

5 Графоориентированная методология разработки средств взаимодействия пользователя в системах автоматизированного проектирования и инженерного анализа

2021.11.06: Графовое описание процессов обработки данных в pSeven (DATADVANCE)

pSeven – это платформа для анализа данных, оптимизации и создания аппроксимационных моделей, дополняющая средства проектирования и инженерного анализа. pSeven позволяет интегрировать в единой программной среде различные инженерные приложения, алгоритмы многодисциплинарной оптимизации и инструменты анализа данных для упрощения принятия конструкторских решений [27].

Принципы функционирования

На концептуальном уровне в pSeven вводятся следующие понятия.

– Проект – набор файлов, используемых в pSeven для описания решений одной или нескольких задач и хранения результатов их решения.

– Расчётная схема (workflow) – формальное описание процесса решения некоторой задачи в виде ориентированного графа, узлами которого являются блоки, а рёбрами - связи. Такое описание хранится в бинарном файле с расширением .p7wf, использующем некоторый специализированный формат хранения подобного рода описаний.

– Блок – функциональный элемент расчётной схемы, отвечающий за обработку входных данных и формирование выходных данных [28].

– Порт – переменная определённого типа, описанная в блоке и имеющая в нём уникальное имя, значение которой может быть передано в другие блоки или получено от них через связи.

– Связь (link) – одностороннее соединение между двумя портами, обеспечивающее передачу данных от одного к другому.

Проекты в pSeven имеют единую базу данных, куда записываются все результаты запусков расчётных схем и откуда берутся данные для их последующей презентации пользователю и их анализа. Для определения переменных, значения которых должны быть записаны в неё записаны, предусмотрены специализированные порты для самих расчётных схем, с которыми связываются те блоки, результаты выполнения которых интересуют пользователя.

Связи служат для маршрутизации данных. С их помощью осуществляется и взаимодействие между блоками и, кроме того, определяется очерёдность их запуска. В момент добавления связи в пакете pSeven выполняется проверка портов на совместимость. Они считаются совместимыми, если тип данных источника можно преобразовать к типу данных адресата [28].

Поддержка циклов и ветвлений

Расчётная схема может включать расчётные циклы. Для их создания применяются специализированные блоки, имеющие функциональную возможность управления запуском других блоков в теле цикла и принятия решения о его прекращении. Такие блоки называются управляющими блоками циклов. Одним из характерных примеров является *оптимизационный цикл*, управление которым осуществляется из блока Optimizer [28], позволяющего, например, настроить максимальное число итераций или требуемую точность, как условия окончания.

Если речь идёт о задачах анализа данных, существует отдельный блок, обозначающий входную точку цикла (Loop). Такой цикл принимает на вход список наборов аргументов, каждый из которых будет обработан на соответствующем шаге цикла, и выдаёт список наборов выходных значений, которые сохраняются в базе данных проекта.

Кроме того, существует возможность включения в расчётную схему условного и безусловного ветвления. Первое достигается за счёт создания связей для подключения одного и того же порта вывода к различным портам ввода. В этом случае по каждой

связи передаётся копия выходных данных, полученных у источника [28]. Условные ветвления создаются с помощью специального блока Condition, который по определённому условию передаёт входные данные одному из подключенных блоков. Их целесообразно использовать для устранения ошибок в работе блока, отбраковки некорректных входных данных и других аналогичных целей [28].

Особенности выполнения расчётных схем

При выполнении расчётных схем каждый блок запускается в отдельном процессе на уровне операционной системы. Как сказано ранее, начало выполнения блока определяется его связями с другими. Любой блок будет ожидать завершения работы другого блока только в том случае, если ему необходимо получить от него входные данные. Это означает, что два блока, не имеющих связей друг с другом, входящие в состав различных ветвлений расчётной схемы, могут запускаться параллельно, поскольку они не зависят друг от друга по используемым данным [28].

Кроме того, при обработке больших выборок данных может потребоваться обрабатывать по несколько наборов данных одновременно в независимых потоках исполнения. Помимо прочего этой цели служит блок Composite, который является контейнером для нескольких блоков. В его настройках можно включить опцию параллельного исполнения, указать, с какого порта данные будут обрабатываться параллельно, и указать максимальное число потоков. При запуске расчётной схемы пакет pSeven создаёт несколько виртуальных экземпляров блока Composite и автоматически распределяет входные наборы данных между ними. Как только в одном из таких виртуальных блоков завершается расчёт, он получает из выборки следующий набор для обработки [29].

Подготовлено: *Тришин И.В. (РК6), 2021.11.06*

2022.02.09: Сравнительная характеристика GBSE, pSeven, Pradis

В рамках сравнения были рассмотрены известные программные комплексы, в основе которых в том или ином виде лежит идея организации вычислений, описываемая с помощью ориентированных графов (графоориентированный подход).

Для проведения сравнения с программным каркасом GBSE выбирались программные продукты, в которых так или иначе реализован описанный выше подход. В первую очередь был рассмотрен программный комплекс pSeven, разработанный отечественной компанией DATADVANCE. Он направлен в первую очередь на решение конструкторских, оптимизационных задач и, помимо этого, задач анализа данных, что в первом приближении делает его аналогом GBSE по предметному назначению. У автора работы присутствует опыт работы с этим комплексом в рамках прохождения курса лабораторных работ по изученной на кафедре дисциплине "Методы оптимизации". В данном курсе были освещены основы работы с pSeven и основные принципы организации вычислений в нём. Таким образом, на момент выбора программных комплексов для

сравнения уже имелась некоторая информация о pSeven, которая позволила включить его в рассмотрение.

Кроме того, научным руководителем данной работы была рекомендована разработка отечественной компании «Ладуга» Pradis – комплекс, так же направленный на решение конструкторских задач. В данной разработке основной упор сделан на задачи анализа проектных решений на микро- и макроуровне.

Выделение признаков для сравнения

Среди прочих должны были быть выделены признаки, относящиеся как к общей структуре программного комплекса, так и к особенностям реализации в нём графоориентированного подхода и, кроме того, к особенностям взаимодействия с пользователем при решении задач, требующих действий с его стороны.

Сравнение осуществлялось с учётом следующих характерных признаков:

1) предметное назначение;
2) принципы формирования графовых моделей;
3) формат описания графовых моделей;
4) файловая структура проекта проведения анализа ТО;
5) особенности работы с входными и выходными данными графовых моделей;
6) особенности передачи данных между узлами графовых моделей;
7) поддержка ветвлений и циклов в топологии графовых моделей;
8) поддержка параллельной обработки данных;
9) возможность выбрать из набора однотипных промежуточных результатов расчётов некоторые экземпляры и продолжить расчёт только для них;
10) возможность доопределять входные данные непосредственно во время обхода графовой модели.

Программный комплекс Pradis

Программный комплекс Pradis, разработанный отечественной компанией «Ладуга» предназначен для анализа динамических процессов в системах разной физической природы (механических, гидравлических и т.д.). Как правило, при помощи данного комплекса решаются нестационарные нелинейные задачи, в которых характеристики системы зависят от времени и пространственных координат. Круг задач, которые могут быть решены с помощью Pradis, достаточно широк: возможен анализ любых технических объектов, модели поведения которых представимы системами обыкновенных дифференциальных уравнений (СОДУ). Анализ статических задач обеспечивается как частный случай динамического расчета.

Практические возможности по решению конкретных задач определяются текущим составом библиотек комплекса, прежде всего библиотек моделей элементов [30]. Данный комплекс был рекомедован к обзору и сравнению, однако, после проведённого обзора официальной документации [31] не было получено достаточного представления об использовании элементов графоориентированных подходов в данном комплексе, поэтому было принято решение исключить его из дальнейшего рассмотрения.

Программный комплекс pSeven

В программном комплексе pSeven, разработанном компанией DATADVANCE, используется методология диаграмм потоков данных (англ. Dataflow diagram, DFD). В комплексе применяются ориентированные графы (орграфы) для описания процесса решения некоторой задачи проектирования технического объекта. В терминах pSeven: графовое описание процесса решения задачи называется *расчётной схемой* (англ. workflow), что, помимо прочего, соответствует классическому термину из области математического моделирования; узлам орграфа поставлены в соответствие процессы обработки данных (используется термин *блоки*), а рёбра определяют *связи* между блоками и направления передачи данных между процессами [32].

Используются следующие базовые понятия pSeven:

- расчётная схема – формальное описание процесса решения некоторой задачи в виде орграфа;
- блок – программный контейнер для некоторого процесса обработки данных, входные и выходные данные для которого задаются через порты (см. ниже);
- порт – переменная конкретного[3] типа, определённая в блоке и имеющая уникальное имя в его пределах;
- связь – направленное соединение типа «один к одному» между выходным и входным портами разных блоков.

С учётом данных понятий можно описать используемую методологию диаграмм потов данных следующим образом. Расчётная схема содержит в себе набор процессов обработки данных (блоков), каждый из которых имеет (возможно, пустой) набор именованных входов и выходов (портов). Данные передаются через связи. Для избежания т.н. гонок данных (англ. data races) множественные связи с одним и тем же входным портом не поддерживаются. Для начала выполнения каждому блоку требуются данные на всех входных портах. Все данные на выходных портах формируются по завершении исполнения блока [32].

На рисунке 17 A, B, C, D - блоки обработки информации, x_i - входные порты, y_i - выходные. Стрелками показаны связи. Согласно изложенному выше принципу, сначала будет запущен блок A, по его завершении - блок B. Затем - блоки C и E (параллельно). По завершении блока C будет запущен блок D. На этом обход расчётной схемы завершится, и значения y_7, y_8 и y_9 будут сохранены в локальной базе данных.

Замечание 2 (принцип обхода в pSeven)
Все порты, которые не привязаны к другим блокам, автоматически становятся внешними входами и выходами для всей расчётной схемы. Для начала обхода расчётной схемы должен быть предоставлен набор входных данных и указаны внешние

[3] Динамическая типизация не поддерживается.

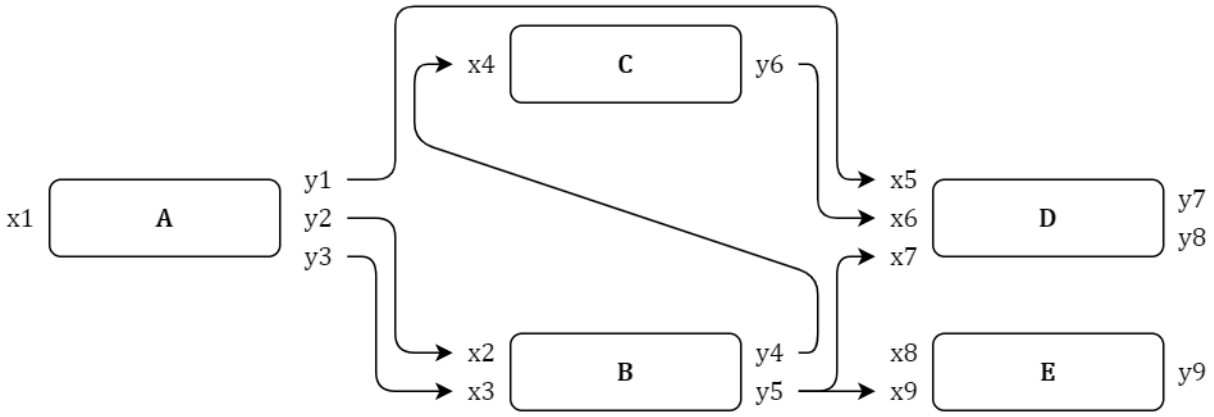


Рис. 17. Пример диаграммы потоков данных

выходные порты, значения которых обязательно должны быть вычислены в результате обхода. Обход производится в несколько этапов: сперва отслеживаются пути от необязательных выходных портов к входным, все встреченные на пути блоки помечаются, как неактуальные и не будут выполнены в дальнейшем; затем отслеживаются пути от обязательных выходных портов к входным и все встреченные на пути блоки помечаются, как обязательные к исполнению. Наконец, обязательные к исполнению блоки запускаются, начиная с тех, которые подключены к внешним входам расчётной схемы, а неактуальные игнорируются. Обход прекращается, когда не остаётся необходимых для выполнения блоков [32].

По завершении обхода расчётной схемы результаты расчётов сохраняются в локальной базе данных проекта. В pSeven встроен инструментарий для визуализации и анализа результатов. Схемы, графики, гистограммы и результаты анализа сохраняются в т.н. отчётах – специальных файлах, сохраняемых в директории проекта.

Результаты проведённого сравнения представлены в таблице 1.

Таблица 1. Сравнительная таблица

№	Признак	pSeven	GBSE
1	Предметное назначение	Задачи оптимизации, анализ данных	Задачи автоматизированного проектирования, алгоритмизация сложных вычислительных методов, анализ данных
2	Принцип формирования графовых моделей	Узлы – блоки (процессы), рёбра – связи (направление передачи данных) [32].	Узлы – состояния данных, рёбра – переходы между состояниями, с указанием функций перехода [16].
3	Формат описания орграфа	Расчётная схема (в форме орграфа) сохраняется в двоичный файле закрытого формата с расширением .p7wf.	Графовая модель (определяет алгоритм проведения комплексных вычислений в форме орграфа) сохраняется в текстовом файле открытого формата, подготовленного на языке aDOT[1], являющегося «сужением» (частным случаем) известного формата DOT (Graphviz).
4	Файловая структура проекта проведения анализа ТО	Проект состоит из непосредственно файла проекта, в котором хранятся ссылки на созданные расчётные схемы и локальную базу данных, сами расчётные схемы, файлы с их входными данными, файлы отчётов, где сохраняются выходные данные последних расчётов и результаты их анализа.	Проект состоит из .aDOT файла с описанием графа, .aINI-файлов с описанием форматов входных данных, библиотек функций-обработчиков, функций-предикатов и функций-селекторов , файлов, куда записываются выходные данные.

5	особенности работы с входными и выходными данными графовых моделей	Входные данные должны быть указаны при настройках внешних входных портов расчётной схемы. Данные с выходных портов схемы сохраняются в локальной базе данных. Для их записи в файлы для обработки/анализа вне pSeven необходимо воспользоваться специально предназначенными для этого блоками.	Входные данные хранятся в файле в формате aINI[2], откуда считываются при запуске обхода графа [33]. Для записи выходных/промежуточных данных в файлы или базы данных необходимо добавить соответствующие функции-обработчики. Формат выходных данных не регламентирован.
6	Особенности передачи параметров между узлами графовых моделей	Данные между узлами передаются согласно определённым связям, которые на уровне выполнения создают пространство в памяти для ввода и вывода данных для выполняемых в раздельных процессах блоков. Транзитная передача данных, которые не изменяются в данном блоке, на выход невозможна.	Поскольку узлами графа являются состояния данных, существует возможность задействовать в расчётах только часть данных, оставляя их другую часть неизменной.
7	Поддержка ветвлений и циклов	Присутствует. Достигается за счёт специальных управляющих блоков, которые отслеживают выполнение условий: для ветвления используется блок «Условие» (англ. condition), который перенаправляет данные на один из выходных портов в зависимости от выполнения описанного условия (подробнее см. [23]); Для реализации циклов в общем случае используются блоки «Цикл» (англ. loop)[28], но для некоторых задач существуют специализированные блоки, организующие логику работы цикла (например, блок «Оптимизатор» (англ. optimizer))	Присутствует по умолчанию

8	Поддержка параллельной обработки данных	Присутствует. Блоки, входящие в состав различных ветвлений схемы могут быть выполнены параллельно, поскольку они не зависят друг от друга по используемым данным.	Присутствует. Существует возможность обойти различные ветвления графа одновременно.
9	Возможность выбрать из набора однотипных промежуточных результатов расчётов некоторые экземпляры и продолжить расчёт только для них;	Производится на этапе анализа результатов с помощью отчётов, где можно задать фильтрацию выходных данных согласно указанным критерия. В случае, если результаты являются промежуточными, расчётную схему приходится разбивать на части.	Планируется реализовать средство визуализации данных, которое в совокупности с автоматической генерацией форм ввода[33] позволят отбирать корректные результаты промежуточных вычислений во время обхода графовой модели.
10	Возможность доопределения значений входных данных в процессе обхода графа	Отсутствует	Частично реализована при помощи функций-обработчиков специального типа, создающих формы ввода

Подготовлено: *Тришин И.В. (РК6), 2022.02.09*

2022.02.09: Требования к представлению графовых моделей в comsdk

Введение

На сегодняшний день существуют две реализации графоориентированного программного каркаса GBSE [26]. Обе они представляют собой подключаемые объектно-ориентированные библиотеки, включающие в себя API для создания состояний СВМ (узлов графовой модели) и их объединения с помощью функций-предикатов и функций-обработчиков [26]. Хронологически первой из них является реализованная на языке C++ библиотека comsdk. Более поздней и потому более актуальной в некоторых аспектах является библиотека на языке Python pycomsdk. Поскольку их разработка проводится не параллельно, между ними существует большое количество алгоритмических и архитектурных отличий. Среди прочих особо выделяется подход к работе с языком описания графовых моделей aDOT и, как следствие, программное представление этих моделей в описанных выше API.

Краткое описание синтаксиса языка aDOT

При описании графовых моделей в GBSE вводятся следующие понятия:

– *состояние данных* – некоторый строго определённый набор именованных переменных фиксированного типа, характерных для решаемой задачи;

– *морфизм* – некоторое отображение одного состояния данных в другое;

– *функция-предикат* – функция, определяющая соответствие подаваемого ей на вход набора данных тому виду, который требуется для выполнения отображения;

– *функция-обработчик* – функция, отвечающая за преобразование данных из одного состояния в другое;

– *функция-селектор* – функция, отвечающая в процессе обхода графовой модели за выбор тех рёбер, которые необходимо выполнить на следующем шаге в соответствии с некоторым условием.

Самая актуальная версия представленного формата содержится в [34].

В текущей версии comsdk поддерживается только уже устаревшая версия формата aDOT, в связи с чем актуальна потребность в обновлении модуля этой библиотеки, содержащего в себе работу с графовыми моделями, для поддержки новейшей версии этого формата.

Список требований

В результате анализа текущего синтаксиса языка aDOT были сформулированы следующие требования к представлению графовых моделей в новой версии библиотеки comsdk:

1) Каждое ребро графа должно иметь возможность привязать к нему до трёх морфизмов – препроцессор, обработчик и постпроцессор

2) Каждый морфизм должен содержать в себе функцию-предикат и функцию-обработчик;

3) Каждый узел графа должен хранить состояние данных (т.е. сведения о типах и именах переменных);

4) Каждый узел графа должен хранить данные о стратегии выполнения рёбер, исходящих из него (поочерёдное выполнение, выполнение в отдельных потоках, выполнение в отдельных процессах, выполнение на удалённом узле через SSH-подключение);

Кроме того, были сформулированы требования к формированию данных обновлённых моделей:

1. При формировании графовой модели все узлы N_i, которые обозначены в aDOT файле с её описанием, как подграфы, должны заменяться загружаемыми из соответствующих aDOT-файлов графовыми моделями G_i, при этом рёбра, входящие в N_i должны быть подключены к начальному узлу G_i, а исходящие из N_i - к конечному узлу G_i;

Подготовлено: *Тришин И.В. (РК6), 2022.02.09*

2022.03.09: Текущее представление графовых моделей в библиотеке comsdk

На рисунке 18 представлена UML-диаграмма классов, связанных с представлением в comsdk ориентированного графа, описывающего организацию вычислительных процессов.

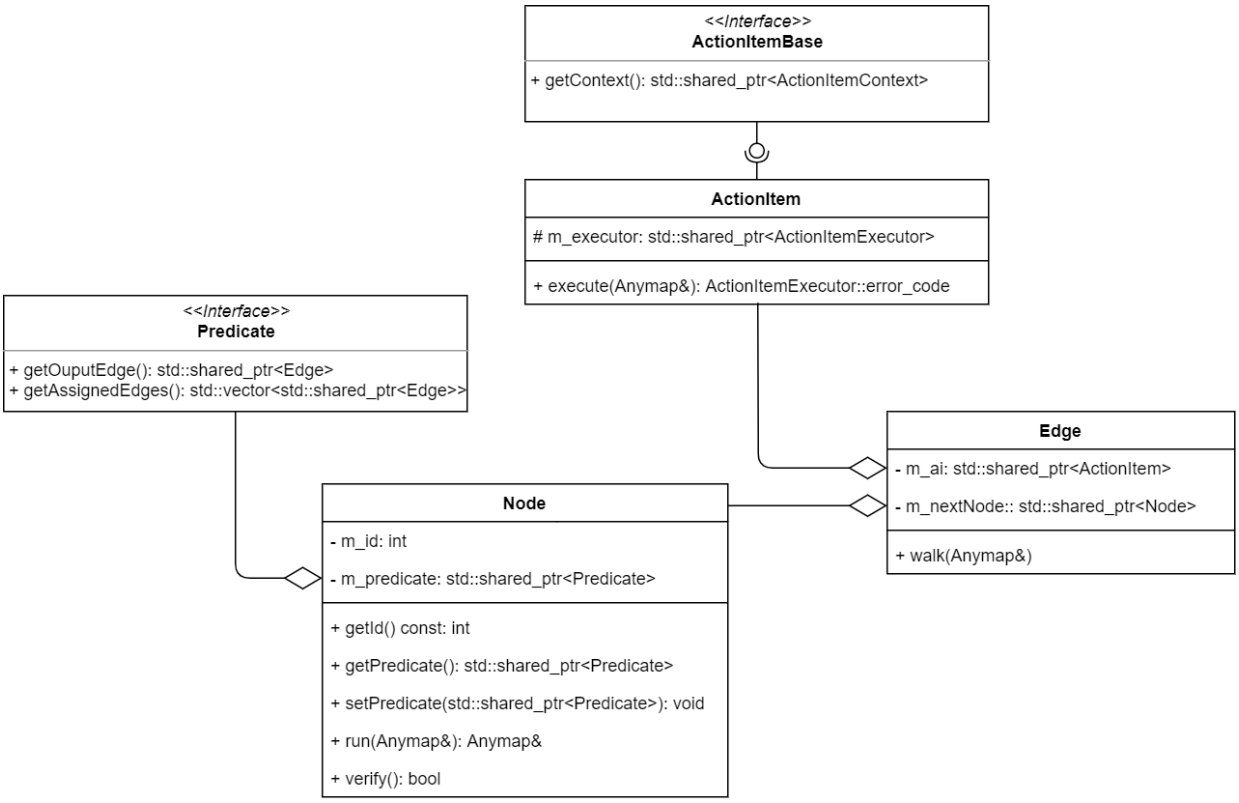


Рис. 18. Текущая структура классов, связанная с графовыми моделями в comsdk

В существующей структуре классов можно выделить следующие недостатки.

1. Отсутствует класс графа, который обеспечивал бы удобный интерфейс графовым моделям.

2. Отсутствует структура данных, обеспечивающая хранение узлов, относящихся к конкретной графовой модели (включая вложенные графовые модели)

3. Индекс узла графа задаётся пользователем при инициализации, что не гарантирует его уникальности.

4. Отсутствует структура данных, обеспечивающая хранение рёбер, относящихся к конкретной графовой модели (включая вложенные графовые модели).

5. Отсутствует объект, который бы описывал связи между узлами и рёбрами; вместо этого эти связи прописаны в самих узлах и рёбрах, что затрудняет операции с графовой моделью (преобразования и проч.).

6. Функции-предикаты привязываются к узлам, а не к рёбрам, что не соответствует требованиям синтаксических конструкций языка aDOT.

7. В текущей версии задачей функций-предикатов фактически является отбор рёбер, которые должны быть выполнены, а не проверка соответствия данных в узле определённому формату.

Подготовлено: *Тришин И.В. (РК6), 2022.03.09*

2022.03.23: Требования к возможностям обхода графовых моделей в GBSE

Графоориентированный подход ГПИ подразумевает параллельное выполнение рёбер графа, выходящих из одной вершины. На рисунке 19 после выполнения функций перехода F_{12} и F_{13}, связанных с рёбрами, будет осуществлён переход в два независимых состояния данных S_2 и S_3 соответственно. Далее возникает задача правильным образом преобразовать данные из этих состояний в общее состояние S_4.

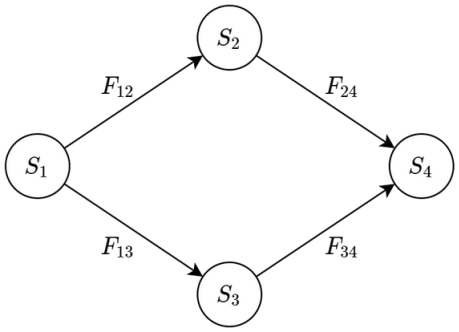


Рис. 19. Пример графовой модели, предполагающей параллельное исполнение

Замечание 3

Далее, допуская определённую нестрогость, говоря о том, что функции перехода, связываемые с рёбрами графовых моделей, выполняются в отдельных потоках выполнения, будем считать, что они выполняются: либо в общем адресном пространстве текущего процесса выполнения, либо в разных процессах на одной или разных вычислительных системах.

Другими словами, в рамках настоящей заметки не будем предполагать, что данные, формируемые в параллельных ветках графовой модели, размещаются в общей памяти.

Рассматриваемый подход может способствовать значительному увеличению эффективности использования ресурсов вычислительной системы и, как следствие, ускорению процесса выполнения[4], однако, предполагает необходимость реализации дополнительных второстепенных задач. Так для примера на рис. 19 функции перехода F_{12} и F_{13} должны выполняться параллельно (в двух разных потоках выполнения), а значит полученные в результате их выполнения данные, в общем случае[5], могут быть размещены в разных адресных пространствах разделённой оперативной памяти. Поэтому встаёт задача сбора этих данных в общую оперативную память при переходе в S_4. В момент разветвления графа должно (в общем случае) происходить «разделение»обрабатываемых данных, чтобы каждая ветвь работала со своим экземпляром

[4] В случае наличия доступных вычислительных ресурсов: нескольких ядер процессора.

[5] Речь идёт о самом общем случае, когда выпонение отдельных функций перехода допускается на разных вычислительных машинах или на кластерных системах с распределённой памятью.

данных. Помимо этого алгоритм обхода графовой модели должен корректно отрабатывать слияние ветвей графа.

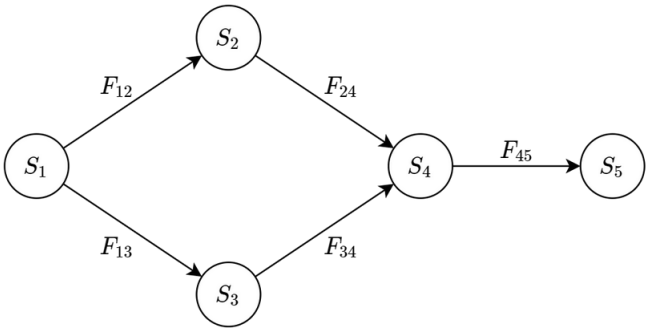


Рис. 20. Пример графовой модели с совмещением ветвей

На рисунке 20 ветви $S_1 \to S_2 \to S_4$ и $S_1 \to S_3 \to S_4$ выполняются в разных потоках выполнения, но ребро F_{45} должно быть выполнено только в одном потоке (если обратного не предполагает функция перехода этого ребра). Таким образом, очевидна необходимость в некоторой управляющей программной структуре, которая бы обеспечивала управление (запуск и завершение) различных потоков выполнения.

Кроме того, указанная управляющая программная структура должна поддерживать несколько вариантов параллельного исполнения. Среди прочих желательна поддержка:

1) поочерёдного выполнения (в первую очередь для отладки);
2) многопроцессного выполнения;
3) многопоточного выполнения;
4) выполнения на удалённых узлах (через SSH-соединение).

Т.о. целесообразна поддержка единого интерфейса для различных режимов (стратегий) выполнения (параллельный, последовательный, распределённый и пр.) в рамках обозначенной управляющей программной структуры.

Подготовлено: *Тришин И.В., Соколов А.П., 2022.03.23*

2022.04.13: Дополнительный обзор литературы и наброски для введения

Методы решения задач, возникающие в процессе современных научно-технических исследований, зачастую предполагают выполнение большого количества операций обработки данных. Каждой такой операции требуются входные данные. По завершении выполнения операции получаются выходные данные. При этом выходные данные одной операции могут являться входными для одной или нескольких других операций. Между ними формируются зависимости по входным и выходным данным. Для учёта

этих зависимостей возникает необходимость правильным образом организовать выполнение операций в пределах отдельно взятого метода и, в частности, при разработке программного обеспечения (ПО), которое реализовало бы данный метод.

В наши дни популярность приобретает применение научных систем организации рабочего процесса (англ. scientific workflow systems). Такие системы позвояют автоматизировать процессы решения научно-технических задач, предоставляя средства организации и управления вычислительными процессами [20]. Процесс работы с подобными системами состоит из 4 основных этапов:

1. составление описания операций обработки данных и зависимостей между ними;
2. распределение процессов обработки данных по вычислительным ресурсам;
3. выполнение обработки данных;
4. сбор и анализ результатов и статистики.

Одной из ключевых особенностей подобного подхода к реализации методов решения научно-технических задач является выделение операций обработки данных в отдельные программные модули (функции, подпрограммы). При известных входных и выходных данных каждого модуля становится возможной их независимая разработка[35]. Это позволяет распределить их разработку между членами команды исследователей. Вследствие этого уменьшается объём работы по написанию исходных кодов, приходящийся на одного исследователя. Это, в свою очередь, облегчает отладку и написание документации, что положительно сказывается на общем качестве реализуемого ПО.

Подготовлено: *Тришин И.В. (РК6-81Б), 2022.04.13*

2022.05.17: Современные форматы описания иерархических структур данных

При выполнении курсового проекта по дисциплине «Технологии Интернет» была поставлена задача разработать программный инструмент описания структуры состояния данных вычислительных методов [26]. Данный инструмент планируется в дальнейшем встроить как компонент в средство визуализации состояний данных. Разработка данного средства направлена на повышение прозрачности и наглядности взаимодействия с программным инструментарием ГПИ.

Кроме того, разработка инструмента описания состояний данных направлена на поддержание идеи документирования алгоритмов и вычислительных методов, реализуемых по методологии GBSE.

Описание структуры состояния данных

Т.н. «состояние данных»[26] представляет собой множество именованных переменных фиксированного типа, характерное для конкретного этапа вычислительного метода или алгоритма. Данные в соответствующем состоянии, как правило, удобно хранить в виде ассоциативного массива. Тип отдельной переменной может быть как скалярным

(целое, логическое, вещественное с плавающей запятой и пр.), так и сложным «векторным» (структурой, классом, массивом и пр.). Примером сложного «векторного» типа является, в свою очередь, ассоциативный массив со строковыми ключами, при этом конкретная переменная этого типа будет хранить, как правило, адрес этого массива. В общем случае элементы данного массива могут иметь разные типы.

В рассматриваемом случае возникает возможность организации хранения состояния данных в виде иерархических структур.

Таким образом, для описания состояний данных требуется формат, который бы поддерживал гетерогенные (т.е. разнотипные) иерархические структуры данных.

Анализ известных форматов

Одними из первых рассмотренных были форматы для хранения научных данных HDF4 и HDF5 [36]. Данные бинарные форматы позволяют хранить большие объёмы гетерогенной информации и поддерживают иерархическое представление данных. В нём используется понятие набора данных (англ. dataset), которые объединяются в группы (англ. group). Кроме того, формат HDF5 считается «самодокументирующимся», поскольку каждый его элемент – набор данных или их группа – имеет возможность хранить метаданные, служащие для описания содержимого элемента. Существует официальный API данного формата для языка C++ с открытым исходным кодом. Одним из гланвых недостатков HDF5 является необходимость дополнительного ПО для просмотра и редактирования данных в этом формате, поскольку он является бинарным.

Альтернативой бинарным форматам описания данных являются текстовые. Среди них были рассмотрены форматы XML (Extensible Markup Language) и JSON (Javascript Object Notation). Главным преимуществом формата XML является его ориентированность на древовидные структуры данных и лёгкость лексико-синтаксического разбора файлов этого формата. Среди недостатков стоит выделить потребность в сравнительно большом количестве вспомогательных синтаксических конструкций, необходимых для структурирования (тегов, атрибутов). Они затрудняют восприятие чистых данных и увеличивают итоговый объём файла.

Формат JSON, так же, как и XML рассчитан на иерархические структуры данных, но является не столь синтаксически нагруженным, что облегчает восприятие информации человеком [37]. Кроме того, крайне важным преимуществом JSON является его поддержка по-умолчанию средствами языкы программирования Javascript, который используется при разработке веб-приложений. При этом JSON также обладает рядом недостатков. Среди них сниженная, по сравнению с XML надёжность, отсутствие встроенных средств валидации и отсутствие поддержки пространств имён, что снижает его расширяемость.

Подготовлено: *Тришин И.В., Соколов А.П., 2022.05.17*

6 Методы удалённого запуска приложений

2022.06.17: Удаленный запуск кода Waleffe_flow.Fortran

Описание структуры программы Waleffe_flow.Fortran

Структура директорий исходных кодов и запусков программы Waleffe_flow.Fortran изображена на рисунке 21.

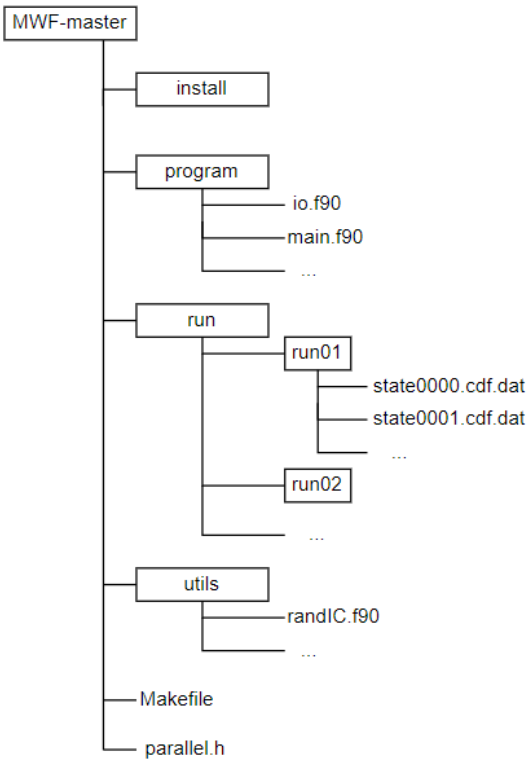


Рис. 21. Структура программы Waleffe_flow.Fortran

Удаленный запуск кода

Для подключения к удаленной машине используется SSH – сетевой протокол прикладного уровня, позволяющий производить удалённое управление операционной системой и туннелирование TCP-соединений [38].

Для запуска программы Waleffe_flow.Fortran на удаленной машине необходимо выполнить следующие команды:

1. Подключение к удаленной машине [39]:

```
ssh user@server
```

В данной работе использовалась виртуальная машина с именем hpc.rk6.bmstu.ru и пользователь amamelkina, поэтому команда выглядела следующим образом:

```
ssh amamelkina@hpc.rk6.bmstu.ru
```

2. Переход в директорию с программой [40]:

```
cd /home/amamelkina/MWF-master
```

3. Чтобы настроить среду для использования библиотек MPI, нужно загрузить соответствующий модуль окружающей среды:

```
module load mpi
```

4. Сборка:

```
make
make install
```

5. Создание начальных условий:

```
make util
./randIC.out
```

6. Далее необходимо создать папку, в которую будут записываться результаты. Такие наборы результатов хранятся в папке run. В папке run создается папка с номером (например run10), в нее копируются файлы /install/main.info, /install/main.out. Файл state0000.cdf.dat, полученный в результате выполнения пункта 5, должен быть переименован в state.cdf.in и тоже скопирован в новую папку результатов.

7. Затем нужно перейти в созданную папку и запустить программу:

```
./main.out
```

8. После остановки программы, когда будет получено необходимое количество данных, можно отключаться от удаленной машины с помощью команды:

```
exit
```

9. Последний шаг – скопировать результат с удаленной машины на локальную [41]:

```
scp -r user@server:<адрес/откуда/копируем>
                   <адрес/куда/копируем>
```

Видно, что запуск данной программы на удаленной машине требует выполнения множества команд. Упростим запуск до одной команды – запуска python-скрипта, в котором будет реализован процесс удаленного запуска.

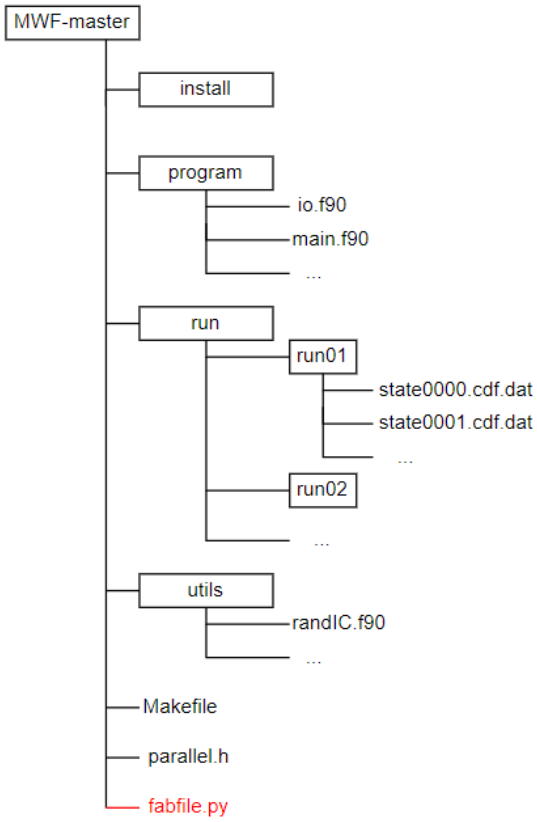


Рис. 22. Размещение fabfile

Программная реализация

Для реализации необходимого python-скрипта была использована библиотека fabric. Fabric – это библиотека Python и инструмент командной строки для оптимизации использования SSH для развертывания приложений или задач системного администрирования [42].

Для работы с fabric первым делом нужно создать fabfile и разместить в структуре файлов так, как показано на рисунке 22.

Fabfile – это то, что контролирует то, что выполняет fabric. Он называется fabfile.py и запускается командой fab. Все функции, определенные в этом файле, будут отображаться как подкоманды fab. Они выполняются на одном или нескольких серверах. Эти серверы могут быть определены либо в fabfile, либо в командной строке [43].

Добавим сервер в fabfile, определив его в переменной окружения env (листинг 4) [44].

Листинг 4. Добавление сервера в переменную окружения env

```
1       env.hosts = ['hpc.rk6.bmstu.ru']
```

Fabric по умолчанию использует локальное имя пользователя при подключении SSH, но при необходимости его можно переопределить, используя env.user. Предоста-

вим пользователю программы возможность ввести имя (листинг 5). Это реализовано с помощью функции prompt(text, default=”, ...) [45]. Данная функция выдаёт пользователю запрос с текстом text и возвращает полученное значение. Для удобства к text будет добавлен одиночный пробел. Если передан параметр default, то он будет выведен в квадратных скобках и будет использоваться в случае, если пользователь ничего не введёт (т.е. нажмёт Enter без ввода текста). По умолчанию значением default является пустая строка.

Листинг 5. Добавление имени пользователя в переменную окружения env

```
1        user = prompt("Enter username", default=''amamelkina)
2        env.user = user
```

Также пользователю нужно ввести время в секундах, в течение которого будет выполняться программа.

Затем с помощью функции run(command, ...), которая запускает команду оболочки на удалённом узле, выполняется запуск программы Waleffe_flow.Fortran.

Также используются функции put(local_path=None, remote_path=None, ...) и get(remote_path, local_path = None) для загрузки файлов на удаленный сервер и скачивания файлов с удалённого сервера соответственно.

Листинг кода python-скрипта представлен в Приложении.

Для запуска fabric предоставляет команду fab, которая считывает свою конфигурацию из файла fabfile.py.

В листинге 6 представлена простая функция, с помощью которой будет продемонстрировано, как использовать fabric [46]. Эта функция сохранена как fabfile.py в текущем рабочем каталоге.

Листинг 6. Пример функции

```
1        def hello():
2            print("Hello!")
```

Функция приветствия может быть выполнена с помощью fab инструмента следующим образом:

```
fab hello
```

В результате выполнения этой команды будет выведено "Hello!".

Таким образом, для запуска программы Waleffe_flow.Fortran на удаленной машине теперь необходимо выполнить только одну команду:

```
fab run_fortran
```

В результате работы данного python-скрипта на персональном компьютере в папке run появится новая папка с результатами запуска.

Подготовлено: *Амелькина А.-М. (РК6), 2022.06.17*

Список литературы

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

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

[3] 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

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

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

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

[7] Frank Eichinger, Klemens Bohm, and Matthias Huber. Mining Edge-Weighted Call Graphs to Localise Software Bugs // Lecture Notes in Computer Science. 2008. P. 333–348.

[8] A. Pop, M. Sjolund, A. Asghar, P. Fritzson, F. Casella. Integrated Debugging of Modelica Models // Modeling, Identification and Control. 2014. Vol. 35, no. 2. P. 93–107.

[9] Alan Humphrey, Qingyu Meng, Martin Berzins, Diego Caminha B. de Oliveira, Zvonimir Rakamaric, Ganesh Gopalakrishnan. Systematic debugging methods for large-scale HPC computational frameworks // Computing in Science and Engineering. 2014. Vol. 16, no. 3. P. 48–56.

[10] Minh Ngoc Dinh, David Abramso1, Chao Jin, Andrew Gontarek, Bob Moench, Luiz DeRose. A data-centric framework for debugging highly parallel applications // Software - Practice and Experience. 2013. Vol. 45, no. 4. P. 501–526.

[11] Dylan Chapp, Nigel Tan, Sanjukta Bhowmick, Michela Taufer. Identifying Degree and Sources of Non-Determinism in MPI Applications via Graph Kernels // IEEE Transactions on Parallel and Distributed Systems. 2021. Vol. 32, no. 12. P. 2936–2952.

[12] Miguel Pignatelli. TnT: a set of libraries for visualizing trees and track-based annotations for the web // Bioinformatics. 2016. Vol. 32, no. 16. P. 2524–2525.

[13] Cantrell K., Fedarko M.W., Rahman G., McDonald D., Y.,Zaw T., Gonzalez A., Janssen S., Estaki M., Haiminen N., Beck K.L., Zhu Q. Empress enables tree-guided, interactive, and exploratory analyses of multi-omic data sets // mSystems. 2021. Vol. 6, no. 2.

[14] Jadeja M., Kanakia H., Muthu R. Interactive Labelled Object Treemap: Visualization Tool for Multiple Hierarchies // Advances in Intelligent Systems and Computing. 2020. P. 499–508.

[15] Esaie Kuitche, Yanchun Qi, Nadia Tahiri, Jack Parmer,Aïda Ouangraoua . DoubleRecViz: A Web-Based Tool for Visualizing Transcript-Gene-Species tree reconciliation // Bioinformatics. 2021. Vol. 37, no. 13. P. 1920–1922.

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

[17] Davidrajuh R. Detecting Existence of Cycles in Petri Nets // Department of Electrical and Computer Engineering. 2016. Vol. 10, no. 12. P. 2936–2951.

[18] Fahad Mahdi, Maytham Safar, Khaled Mahdi. Detecting Cycles in Graphs Using Parallel Capabilities of GPU // Computer Engineering Department. 2011. Vol. 13, no. 12. P. 2936–2952.

[19] Diestel R. Graph Theory: Springer Graduate Text GTM 173. Springer Graduate Texts in Mathematics (GTM). Springer New York, 2012.

[20] Workflows and e-Science: An overview of workflow system features and capabilities / D. E., G. D., S. M. et al. // Future Generation Computer Systems. 2009. Vol. 25, no. 5. P. 528 – 540.

[21] DAGMan Workflows | HTCondor Manual. Электронный ресурс. 2023. (дата обращения: 07.01.2023).

[22] VisTrails 2.2.4 documentation. Электронный ресурс. 2016. (дата обращения: 07.01.2023).

[23] Condition - pSeven 6.31.1 User Manual [Электронный ресурс] [Офиц. сайт]. 2022. (дата обращения 07.03.2022).

[24] Кормен Ч. Алгоритмы: построение и анализ, 2-е издание. Издательский дом "Вильямс 2005.

[25] Седжвик Р. Фундаментальные алгоритмы на C++. Часть 5. Алгоритмы на графах. СПб: ООО "ДиаСофтЮП 2002.

[26] Соколов А.П. Першин А.Ю. Графоориентированный программный каркас для реализации сложных вычислительных методов // Программирование. 2018. № X.

[27] ДАТАДВАНС | Программное обеспечение для анализа данных и оптимизации [Электронный ресурс] [Офиц. сайт]. 2021. (дата обращения 06.11.2021).

[28] Расчётные схемы - Руководство пользователя pSeven 6.27 [Электронный ресурс] [Офиц. сайт]. 2021. Дата обращения: 15.11.2021.

[29] Параллельные вычисления - Руководство пользователя pSeven 6.27 [Электронный ресурс] [Офиц. сайт]. 2021. (дата обращения 15.11.2021).

[30] PRADIS. Общее описание системы [Оф. документация]. 2007.

[31] PRADIS. Методы формирования и численного расчёта математических моделей переходных процессов [Оф. документация]. 2007.

[32] Alexey M. Nazarenko Alexander A. Prokhorov. Hierarchical Dataflow Model with Automated File Management for Engineering and Scientific Applications // Procedia Computer Science. 2015. Т. 66. URL: https://www.sciencedirect.com/science/article/pii/S1877050915034055?pes=vor.

[33] Соколов А.П., Першин А.Ю. Программный инструментарий для создания подсистем ввода данных при разработке систем инженерного анализа // Программная инженерия. 2017. Т. 8, № 12. С. 543–555.

[34] Соколов А.П. Першин А.Ю. Описание формата данных aDOT (advanced DOT). 2020.

[35] Данилов А.М., Лапшин Э.В., Беликов Г.Г., Лебедев В.Б. Методологические принципы организации многопотоковой обработки данных с распараллеливанием вычислительных процессов // Известия вузов. Поволжский регион. Технические науки. 2001. № 4. С. 26–34.

[36] The HDF Group [Электронный ресурс] [Оф. сайт]. 2022. Дата обращения: 17.05.2022.

[37] JSON vs XML: What's the difference? | Guru99 [Электронный ресурс]. 2022. Дата обращения: 18.05.2022.

[38] SSH. // ru.wikipedia.org – URL: https://ru.wikipedia.org/wiki/SSH. (Дата обращения 6.05.2022).

[39] Памятка пользователям ssh. // habr.com – URL: https://habr.com/ru/post/122445/. (Дата обращения 6.05.2022).

[40] Управляем файлами и директориями в Linux. // infobox.ru – URL: https://infobox.ru/community/blog/linuxvps/310.html. (Дата обращения 6.05.2022).

[41] Копирование файлов и папок через SSH. // ru.flamix.software – URL: https://ru.flamix.software/about/news-article/kopirovanie-faylov-i-papokok-cherez-ssh/. (Дата обращения 6.05.2022).

[42] Fabric documentation. // docs.fabfile.org – URL: https://docs.fabfile.org/en/1.11/tutorial.html. (Дата обращения 16.05.2022).

[43] Fabric. // runebook.dev – URL: https://runebook.dev/ru/docs/flask/patterns/fabric/index. (Дата обращения 16.05.2022).

[44] The environment dictionary, env. // docs.fabfile.org – URL: https://docs.fabfile.org/en/1.11/usage/env.html. (Дата обращения 16.05.2022).

[45] Python-блог: Fabric: Операции. // python-lab.blogspot.com – URL: http://python-lab.blogspot.com/2013/02/fabric.html. (Дата обращения 16.05.2022).

[46] Начало работы с библиотекой Fabric Python. // coderlessons.com – URL: https://coderlessons.com/articles/programmirovanie/nachalo-raboty-s-bibliotekoi-fabric-python. (Дата обращения 16.05.2022).