Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский физико-технический институт (государственный университет)»

Факультет радиотехники и кибернетики
Кафедра теоретической и прикладной информатики

Направление подготовки/специальность: 03.04.01 Прикладные математика и физика
Направленность (профиль) подготовки: Математические и информационные технологии
Форма обучения: очная

Допущен к ГИА
Руководитель учебного подразделения
_____/_____
(подпись) *расшифровка подписи*
«____» _____2018.

Оптимальный по времени поиск узких мест производительности и проблем конфигурации в вычислительной системе.

(магистерская диссертация)

Обучающийся: Хацкевич Алексей Владимирович

_____
(подпись обучающегося)

Научный руководитель: Емельянов Павел Владимирович, к.ф.-м.н.

_____
(подпись научного руководителя)

Москва 2018

Аннотация

Данная работа посвящена проблеме поиска и устранения проблем конфигурации серверов и запущенных на них приложений.

Идея, вокруг которой строится данная работа, заключается в том, что многие проверки конфигурации могут быть разбиты на составные элементы – более мелкие проверки. Эти элементы могут переиспользоваться или не быть исполненными вовсе.

Например, в случае, если серверное приложение работает медленно, проблема может заключаться в том, что одному из процессов не хватает процессорного времени. Дальнейшее расследование может показать, что одна из функций занимает 40% всего времени и при этом 90% времени в ней расходуется на кеш-миссы. Проблема может оказаться и в том, что процесс проводит слишком много процессорного времени в переключениях контекста из-за неэффективной организации работы с сетью. Данный пример демонстрирует две возможных ветви поиска проблемы. Однако ветвей может быть значительно больше, а связи между ними сложнее.

Таким образом, поиск проблемы и уточнение деталей может быть представлен в виде обхода направленного ациклического графа. В случае, когда в графе есть условные конструкции и логические операции, можно, изменяя порядок обхода графа и выкидывая ненужные зависимости, влиять на время исполнения проверок.

Данная работа исследует:

1. Способы представления проверок в виде графа
2. Подходы к оптимизации графа зависимостей

Подход к описанию проверок в виде графа, для последующих оптимизаций, является уникальным в своем роде.

Преимущества представления проверок в виде графа:

1. Скорость работы
2. Простота добавления уточняющих проблему проверок

Всё алгоритмы разрабатывались с расчетом на большое число вершин графа. Многие из них требуют существенной вычислительной

мощности (1сек для 10.000 вершин) лишь на этапе предрасчёта. Созданный инструмент может стать крайне полезным для системных администраторов и инженеров при расширении базы проверок.

Содержание

Словарь терминов **7**

Аббревиатуры **8**

1 Введение **11**

1.1 Мотивация 11

1.2 Цель исследования 12

2 Объект исследования **14**

2.1 Профилировщик 14

2.2 Структура профилировщика 14

2.2.1 Проверки и зависимости 14

2.2.2 Планировщик 15

2.2.3 Процесс, исполняющий зависимости 16

2.2.4 Репортер 16

2.2.5 Логгер 16

2.3 Модель профилировщика 17

2.3.1 Базовая модель 17

2.3.2 Расширенная модель 17

3 Реализация профилировщика **18**

4 Реализация: описание графа зависимостей **18**

4.1 Семантика языка описания зависимостей 18

4.2 Выбор инструмента 19

4.3 Реализация фреймворка описания ГЗ на языка Python 19

4.4 Конструкции описания зависимости 20

4.4.1 Описание нового класса 20

4.4.2 Декорирование функции 21

4.4.3 Использование логических/арифметических операций . . . 21

4.5 Передача результата НЗА в главная процедура (ГП) 22

4.6 Передача результата ЗА в ГП 22

4.7 Поиск зависимости, в случае если такая зависимость уже была создана . . . 23
4.7.1 Необходимость автоматического поиска существующих аналогичных экземпляров зависимостей . . . 24
4.7.2 Проверка наличия аналогичной зависимости при создании нового экземпляра зависимости . . . 24
4.8 Дифференциация зависимостей генерирующих данные и проверок 25
4.9 Некоторые детали реализации . . . 26
4.9.1 Использование библиотеки numpy . . . 26
4.9.2 Визуализация графов зависимостей, graphviz . . . 26

5 Реализация: планировщик 27

5.1 Дополнительная информация для алгоритмов минимизации времени . . . 28
5.1.1 Подход к сбору информации . . . 28
5.1.2 Собираемые данные . . . 28
5.1.3 Сопоставление зависимостей и их описания по собранным данным . . . 29
5.2 Предрасчитанные значения и их применение . . . 30
5.2.1 Время исполнения зависимости . . . 30
5.2.2 Вероятность срабатывания проверки . . . 31
5.2.3 Вероятности положительного значения результата зависимости . . . 31
5.2.4 Количество проверок которым может понадобиться зависимость . . . 32
5.3 Оптимизации . . . 33
5.4 Метрика времени исполнения . . . 35
5.5 Рассматриваемые варианты метрик . . . 36
5.6 Минимизация времени обнаружения следующей проблемы . . . 36
5.7 Минимизация ассоциативного пенальти до следующей проблемы . 40
5.8 Минимизация суммы времен обнаружения проблем . . . 42
5.9 Эмпирические алгоритмы . . . 42

5.10 Вспомогательные алгоритмы 43
5.10.1 Алгоритм обхода графа в глубину с синхронизацией 43
5.10.2 Расчёт веса зависимости 45
5.10.3 Расчёт усредненного времени исполнения зависимости . . 47
5.11 Факторы, учитываемые эмпирическим алгоритмом 48
5.12 Жадные алгоритмы выбора следующей зависимости 50
5.12.1 Веса зависимостей 50
5.13 Расчет времени исполнения ветви зависимостей 51
5.14 Итеративный расчёт параметров зависимостей 56
5.14.1 Расчёт весов 56
5.14.2 Расчёт усредненных времен 57
5.15 Взвешенная метрика offline 57
5.16 Взвешенная метрика offline с предрасчётом порядка 59
5.17 Взвешенная метрика online 60
5.18 Дополнительное пространство для оптимизаций 61
5.18.1 Поиск кластеров проверок на матрице корреляции зависимостей 61
5.19 Предрасчёт долгих легковесных зависимостей 62

6 Выводы **62**

7 Дальнейшая работа **62**

Словарь терминов

Вес зависимости Число, выражающее одновременно количество и важность проверок, в которых задействована (прямым и косвенным образом) данная зависимость. Это определение не является строгим, и позволяет в разных случаях иметь разные формулы для точного её доопределения при сохранении общей сути. Сложность может заключаться в том, что вес зависимости может отражать вес проверок, которые пройдут через эту зависимость лишь с определённой вероятностью, но не гарантированно.. 41

Вес проверки Число, обозначающее значимость реультата проверки для пользователя. Пример: провека на возможность взломать систему через известную уязвисомть более весомая, чем проверка количества разбитых из-за неподходящего размера MTU пакетов.. 41

Усредненное время исполнения зависимости Число, равное статистическтму времени исполнения зависимости, деленному на количество использований её результата. $< t_i >= t(d_i)/n$.. 43, 46

главная процедура функция, принадлежащая зависимости, которая принимает на вход результаты исполнения зависимостей-аргументов, и выдает результат исполнения зависимости.. 4,

граф зависимостей направленный ациклический граф зависимостей, отражающий взаимные связи зависимостей друг от друга. 10,

зависимость сущность хранящая код который генерирует новую информацию о системе или проверяет какую-либо проблему и результат выполнения; результат выполнения может быть в дальнейшем использован другими зависимостями. 10,

зависимость-аргумент зависимость, передающаяся в новую зависимость при её создании. В последствии результат исполнения зависимости-аргумента передается на вход главной процедуры новой зависимости.

Прим: зависимость, вычисляющая количество тредов процесса по его PID,

требует PID как аргумент. В этом случае PID вычисляется в другой зависимости, например, которая ищет PID процесса а самым высоким потреблением CPU. 14,

неявная зависимость-аргумент зависимость, результат исполнения которой передается на вход главной процедуры зависимости. Отличается от ЗА тем, что её ненужно указывать явно при создании новой зависимости. Прим: проверка что свободной оперативной памяти более 50mb, имеет неявную зависимость на задачу по вычислению общего количества свободной оперативной памяти. 18,

профилировщик Програмный продукт, различные аспекты создания которого рассматриваются в данной работе. Латинское название – "gprofile", образовывается словами "graph" и "profiling". 8, 10, 15, 19

Аббревиатуры

ГЗ граф зависимостей. 10, 11, 13, 29, 31, 33–35, 38–40, 42–44, 54–57, *Glossary:* граф зависимостей

ГП главная процедура. 4, 10, 12, 15, 16, 18, 54, *Glossary:* главная процедура

З зависимость. 10, 18, 29, *Glossary:* зависимость

ЗА зависимость-аргумент. 14–16, 19, *Glossary:* зависимость-аргумент

НЗА неявная зависимость-аргумент. 15, 18, *Glossary:* неявная зависимость-аргумент

П проверка. 16, 21, *Glossary:* проверка

Список иллюстраций

1 Пример визуализации графа проверок. 26
2 Использование информации о времени исполнения зависимости. . 31

3 Использование информации о вероятности принятия возвращаемым значением положительного значения 32
4 Использование информации о вероятности принятия возвращаемым значением положительного значения 32
5 Пример зависимостей, которые можно не исполнять. 34
6 Выбор с учётом переиспользования. 34
7 Минимизация времени обнаружения следующей проблемы, выбор между одной близкой и двумя далёкими проверками. 37
8 Демонстрация метрики минимизирующей время обнаружения следующей проблемы. 37
9 Демонстрация метрики минимизирующей время обнаружения следующей проблемы. Увеличение вариантов обхода с добавлением проверки. 38
10 Демонстрация метрики минимизирующей ассоциативное пенальти обнаружения следующей проблемы. 41
11 Распространение информации в графе при обходе в глубину. . . . 43
12 Переиспользование зависимости 45
13 Учёт посещённых к текущему моменту зависимостей. 49
14 Неоптимальный план при учете только весов зависимостей 50
15 Неверный расчет минимального времени исполнения 52
16 Обоснование необходимости учета вероятности положительного исхода при расчете времени исполнения ветви. 54
17 Необходимость полного перестроения маршрута при необходимости исполнить маловероятную сложную ветвь. 61

Список литературы

[1] *Principles of programming languages.* ACM, Baltimore, 1981.

[2] Steve Best. *Linux debugging and performance tuning.* Prentice Hall Professional Technical Reference, Upper Saddle River, NJ, 2006.

[3] J. Brumbaugh-Smith and D. Shier. An empirical investigation of some bicriterion

shortest path algorithms. *European Journal of Operational Research*, pages 216 – 224, 1989.

[4] Thomas H Cormen. *Introduction to algorithms.* MIT Press, Cambridge, Mass., 1990.

[5] Vilim Feller. *An introduction to probability theory and its applications.* John Wiley and sons New York, 1968.

[6] Michael R Garey. *Computers and intractability.* W.H. Freeman, San Francisco, 1979.

[7] Brendan Gregg. *DTrace.* Prentice Hall, Upper Saddle River, NJ, 2011.

[8] Saif Ur Rehman Khan, Sai Peck Lee, Raja Wasim Ahmad, Adnan Akhunzada, and Victor Chang. A survey on test suite reduction frameworks and tools. *International Journal of Information Management*, pages 963 – 975, 2016.

1 Введение

1.1 Мотивация

Сегодня мы можем наблюдать неуклонный рост сложности и количества вычислительных систем. Каждый день появляются новые технологии для внедрения и эксплуатации которых необходимы специалисты с опытом использования этой технологии. Некоторые технологии наоборот перестают развиваться, но продолжают требовать поддержку. Инженерам, на плечи которых ложится ответственность разбираться с этими продуктами, в случае, когда происходят аварии, становится все сложнее разбираться в зоопарке технологий, с которыми приходится иметь дело. Для молодых специалистов проблема усложняется тем, что помимо новых технологий им приходится изучать еще и старые.

Простое серверное приложение может быть запущено внутри контейнера, которым управляет контейнерный оркестратор, который запущен внутри виртуальной машины, которой управляет оркестратор виртуальных машин, которые плотно запущены на физических машинах, которые соединены между собой через виртуальную сеть. Кроме того, само приложение может быть запущено в собственном виртуальном окружении и состоять из множества различных компонент. Каждой из перечисленных сущностей отводятся различные ресурсы, за эффективным использованием которых приходится следить. При этом необходимо держать в голове особенности виртуализации, контейнеризации, физические особенностей работы оборудования (жесткие диски, сеть, охлаждение), особенности конкретной прослойки, операционной системы и даже библиотек используемых в приложении. Описанная архитектура часто применяется в распространенных сегодня контейнеризованных приложениях и не является надуманной.

Список вещей, о которых необходимо помнить инженеру данной системы огромен. Не стоит надеяться, что любой сотрудник разберется с легкостью с любой проблемой, возникшей внезапно в такой системе.

Хочется отметить, что при локализации проблем, инженеры совершают одни и те же действия: проверяют диски, память, загруженность системы. Когда находится параметр, который отличается от ожидаемого значения, начинается расследование причин отличия. Проблема может возникнуть в абсолютно лю-

бой части системы, начиная от особенности работы базы данных, и заканчивая неправильным распределением прерываний сетевой карты по ядрам процессора. Это наводит на мысль что процесс локализации проблемы может быть автоматизирован.

Данная работа исследует подходы к:

1. Автоматизации поиска проблем производительности и безопасности
2. Описанию проблем в виде графа зависимостей
3. Оптимизации графа зависимостей для ускорения его исполнения

Преследуемые данной работой цели диктуются потребностями инженеров быстро решать возникающие аварии и позволяет организациям экономить деньги и сохранять репутацию.

Касательно самого метода, хочется подчеркнуть, что представление проблем в виде графа зависимостей позволяет очень легко добавлять уточняющие проверки, которые, оперируя уже собранными данными для общей проблемы, делают несколько дополнительных проверок и раскрывают дополнительные подробности, давая больше информации для её устранения.

1.2 Цель исследования

Разработать инструмент для поиска узких мест производительности и проблем конфигурации в вычислительной системе.

В ходе создания инструмента выполнить следующие задачи:

1. Разработать подход к представлению проверок в виде графа зависимостей.
2. Разработать алгоритм планирования обхода дерева зависимостей.
3. Реализовать инструмент (библиотеку) для описания и поиска проблем.

Далее будем называть программу, которая ищет зависимости "Профилировщик"или "gprofile"от слов graph и profiling.

Предполагается, что профилировщик будет применяться на серверах, находящихся под нагрузкой. Нагрузка не должна сильно меняться во времени, так как сбор информации о системе происходит не мгновенно. [1]

Количество ребер графа на который предлагается ориентироваться при расчёте производительности планировщика – 10.000.

Целевая платформа – Linux.

Примеры проблем:

1. Все фреймы бьются пополам из-за неправильной настройки MTU на прокси-сервере
2. Веб-приложение написано в синхронной модели и большая часть ресурсов сервера уходит на переключение контекста
3. Диск не успевает обрабатывать очереди запросов, ожидая позиционирования головки из-за абсолютно случайного характера ввода-вывода
4. Количество bad-блоков диска подходит к критическому
5. Открытие и закрытие соединения в базе
6. Время на сервере выставлено неправильно
7. Веб-приложение запущено с root-привилегиями
8. Приложение nginx использует сетевой сокет вместо unix для проксирования внутренних запросов
9. Приложение с определенным PID потребляет 80% CPU при этом проводя 90% времени в одном фрейме

[1] Веб-сервер, с постоянным RPS является подходящим примером. Сервер на котором запускаются разные кратковременные задачи, считается неподходящим.

2 Объект исследования

2.1 Профилировщик

Профилировщиком будем называть программу, которая непосредственно ищет проблемы. Исследование подходов к реализации различных аспектов профилировщика было главной задачей данного исследования.

С точки зрения пользователя, профилировщик выглядит как набор зависимостей, или проверок, которые выполняются в каком-то порядке и сообщают о проблемах в системе.

2.2 Структура профилировщика

Основные компоненты профилировщика:

1. Проверки и их зависимости
2. Планировщик
3. Процесс, исполняющий зависимости
4. Репортер[2]
5. Логгер

2.2.1 Проверки и зависимости

Рассмотрим зависимость (З) подробнее свойства зависимости:

- несет в себе определенный код, который исполняясь генерирует данные или сообщение о проблеме, который будем называть ГП
- может быть одного из типов:

  Простая зависимость возвращает данные; может зависеть от других зависимостей (параметров)[3]

[2]Reporting - сообщение о найденных проблемах

[3]Для расчета значения этой зависимости требуются данные полученные при выполнении других зависимостей. Таким образом и строится граф зависимостей (ГЗ)

Проверка также является зависимостью (может быть параметром других зависимостей), однако в некоторых случаях, может так же сообщать о найденных проблемах. Поэтому такая зависимость не может быть удалена при оптимизации плана исполнения.

Логическая принимает на вход другие зависимости и выдает результат соответствующий логической операции. Примеры: or, and, not.[4]

- может использовать результат исполнения (зависеть от) других зависимости
  - использовать данные генерируемые другими зависимостями
  - иметь необходимое условие запуска[5]

Все зависимости образуют ГЗ, который отражает взаимные отношения между зависимостями и накладывает ограничения на порядок их исполнения.

2.2.2 Планировщик

Этот модуль профилировщика отвечает за оптимизации связанные с порядком обхода ГЗ. Оказывается, что изменяя порядок обхода ГЗ, можно сократить общее время работы профилировщика. Методы оптимизациям, которые рассматриваются в исследовании, разрабатывались безотносительно того как именно устроено ГЗ и какие конкретно проверки там присутствуют. Инструмент разрабатывался с целью быть универсальным и выдавать приемлемый результат на любой топологии ГЗ.

Планировщик состоит из двух модулей:

Времени исполнения запускается после запуска очередной зависимости, для выбора следующей зависимости

Предрасчета выполняет максимум сложных вычислительных операций по оптимизации плана исполнения в offline режиме. Результат offline опти-

[4]В процессе разработки моделей планировщика и профилировщика рассматривались различные варианты механики логических зависимостей. Наиболее удачным оказался вариант, в котором любое значение отличное от $None$ и $False$ воспринимается как $True$.

[5]В случае если это условие принимает значение $False$, исполнение зависимости прекращается, а результатом её исполнения становится $False$

мизаций сохраняется в файле, и в последствии может быть использован много раз на любой машине.

Более подробно этот модуль рассматривается в разделе посвященном реализации.

2.2.3 Процесс, исполняющий зависимости

После того как планировщик спланировал, какую зависимость необходимо запускать следующей, исполнитель подготавливает аргументы главной процедуры зависимости и запускает её. В усложненной модели исполнителя, которая будет рассмотрена далее, так же добавляется оптимизация, которая дает возможность исполнять долгие задачи в фоновом процессе. Это так же входит в зону ответственности исполнителя.

2.2.4 Репортер

Модуль, к которому обращается ГП в случае необходимости сообщить о проблеме. Проблемы могут выставлять различные уровни критичности.

2.2.5 Логгер

Модуль логгинга собирает всю информацию о ходе работы профилировщика, для последующего её анализа планировщиком.

Собираемые данные:

1. Информация о системе
2. Последовательность в которой были запущены зависимости
3. Время работы каждой зависимости
4. Зависимости, вернувшие положительный результат
5. Проверки, сообщившие о проблеме
6. Другие данные

2.3 Модель профилировщика

Для того чтобы оптимизировать работу программы нам необходимо обозначить ограничения системы в рамках которой работает профилировщик.

Факторы ограничивающие модель должны выбираться на основе ограничений реального мира, а так же максимально упрощать последующий анализ при этом давая достаточно свободы для качественных оптимизаций.

Для упрощения последующего анализа введем две модели.

2.3.1 Базовая модель

Базовая модель профилировщика, к которая применялась в рамках исследования для первичного анализа эффективности того или иного алгоритма планирования.

Базовая модель обладает следующими свойствами и ограничениями:

1. известно сколько исполняется каждая зависимость [6]
2. известна вероятность принятия результатом зависимости положительного значения
3. известна вероятность срабатывания каждой проверки [7]
4. зависимости исполняются в один поток

На основе этой модели будут сравниваться различные подходы к исполнению ГЗ между собой.

2.3.2 Расширенная модель

Эта модель наследует всё из предыдущей, дополняя её следующим свойством:

Паралельно с главным потоком исполнения, существует несколько потоков, в которых могут запускаться нетребовательные к ресурсам, но долгие по времени исполнения задачи.

Эта модель будет использоваться для более глубокого анализа планировщика и анализа дополнительных оптимизаций.

[6] Собирается статистика из предыдущих запусков и усредненные значения считаются достоверными.

[7] По статистике предыдущих запусков вычисляется вероятность срабатывания проверки в случае её запуска.

3 Реализация профилировщика

Рассмотрение деталей реализации профилировщика пришлось разбить на несколько секций из-за сложной структуры описания.

Следующие несколько секций описывают то, какие факторы влияли на создание профилировщика и какие подходы к решению технических проблем выбирались.

4 Реализация: описание графа зависимостей

4.1 Семантика языка описания зависимостей

Инструмент, который описывает проверки в виде графа должен иметь возможность передавать следующие семантические конструкции:

1. Создание зависимости
2. Описание аргументов зависимости
3. В случае если аргументом является зависимость-аргумент (ЗА), необходимо иметь возможность указать на неё
4. Дифференциация одинаковых зависимостей, принимающих различные аргументы[8]
5. Базовые логические операции над зависимостями:
   a) NOT
   b) AND
   c) OR
   d) EQ
   e) NEQ
   f) LT

[8]Зависимости считаются одинаковыми, только в случае, если у них совпадают все типы и аргументы всего дерева аргументов.

g) LE

h) GT

i) GE

6. ГП, результатом исполнения которой являются произвольные данные, а на вход которой подаются результаты исполнения ЗА

7. Сообщение о найденной проблеме

4.2 Выбор инструмента

К описанию графа зависимостей рассматривалось два подхода.

1. Реализация своей грамматики для описания связей зависимостей и встраивание уже существующего интерпретируемого языка программирования для описания ГП

2. Адаптация интерпретируемого языка программирования под семантику профилировщика

В качестве языка описания ГП был выбран Python третьей версии, из-за его популярности и распространенности в новых дистрибутивах как установленного по умолчанию.

4.3 Реализация фреймворка описания ГЗ на языка Python

Интерпретируемый язык Python третьей версии был выбран для описания семантических конструкций отношения зависимостей в графе и для описания ГП.

Каждый из типов зависимостей было решено представлять как отдельный класс, экземпляры которого содержат в себе зависимости-аргументы, необходимые для исполнения ГП.

Чтобы реализовать фреймворк, потребовалось решить следующие задачи:

1. Конструкции описания зависимости

2. Передача результатов неявных ЗА в ГП

3. Передача результатов ЗА в ГП
4. Поиск зависимости, в случае если такая зависимость уже была создана
5. Дифференциация зависимостей генерирующих данные и проверок

Рассмотрим реализацию каждой из задач.

4.4 Конструкции описания зависимости

Было реализовано несколько конструкций описания зависимости:

1. Описание нового класса
2. Декорирование функции
3. Использование логических/арифметических операций

Рассмотрим каждый из них немного подробнее:

4.4.1 Описание нового класса

Для описание нового типа зависимости в виде класса, необходимо наследование от класса Dependency.

Описание ГП должно содержаться в перегруженном методе run.

```
class NewCheck(Dependency):
    def run(arg1, arg2):
        return func(arg1, arg2)


nc1 = NewCheck(5, 7)
nc2 = NewCheck(nc1, 9)
```

Листинг 1: Описание зависимоти при помощи нового класса

4.4.2 Декорирование функции

Предыдущий подход является довольно громоздким. В случае когда без него можно обойтись, предлагается другой метод: декорирование функции.

Был разработан специальный декоратор, который принимая на вход функцию возвращает новый экземпляр класса Dependency с соответствующей функцией run.

Аргументы передаются в такую функцию посредством параметров функции по умолчанию. [9]

Выглядит это следующим образом:

```
@data_provider
def my_func(arg1=arg1, arg2=arg2):
    return func(arg1, arg2)


@data_provider
def my_func2(arg1=my_func):
    return func(arg1)
```

Листинг 2: Описание зависимоти при помощи декорации функции

Такой подход практически не требует дублирования кода и требует описания лишь необходимых элементов.

4.4.3 Использование логических/арифметических операций

Результат применения логических/арифметических операций над результатами выполнения зависимостей так же должен быть переиспользован, поэтому был добавлен и такой способ описания зависимостей.

```
dep3 = dep1 | dep2
dep4 = dep3 & dep2
dep5 = dep1 > dep2
```

Листинг 3: Описание зависимоти при помощи логических/арифметичесих операций

[9] Python позволяет получить доступ к параметрам по умолчанию любой функции

Реализация такого метода создания зависимостей на языке Python представляет из себя перегрузку операторов класса Dependency.

4.5 Передача результата НЗА в ГП

Неявная зависимость-аргумент (НЗА) – зависимость от которой зависит новая зависимость, даже в случае отсутствия явно переданных аргументов.

Например, проверка наличия свободной оперативной памяти, всегда зависит от задачи по подсчету общего размера свободной памяти. В таком случае, задача подсчета общего количества памяти может быть неявной зависимостью.

Ниже приведен листинг ключевых процедур класса З:

```
class Dependency:
    def __init__(self, *args, **kwargs):
        self.args = args
        self.kwargs = kwargs
        # implicit `run` arguments may be passed by overriding t
        # method
        self.args.append(dep1)
        self.kwargs["dep2"]) = Dep2(dep1)
        sef.super()

    def run(*args, **kwargs):
        raise NotImplemented("This method should be overriden")
```

Листинг 4: Ключевые процедуры класса З

Как видно из листинга, функция __init__ может произвольным образом менять аргументы передаваемые в конструктор объекта класса. Эти же аргументы в дальнейшем попадут в функцию run, которая является ГП. Все необходимые НЗА предлагается добавлять здесь.

4.6 Передача результата ЗА в ГП

Реализовано два основных варианта передачи аргументов:

1. Передача атрибутов ГП конструктор класса З

2. Передача экземпляров ЗА через параметры по умолчанию функции

Оба этих метода представлены в следующем листинге:

```
method1 = SomeDependency(arg1, arg2, arg3=arg3)

@data_provider
def method2(arg1=arg1, arg2=arg2):
    # some code
    return ret
```

Листинг 5: Передача ЗА в функцию

В обоих случаях, в конструкторы классов мы передаем не сами значения, а объекты классов зависимостей, которые представляют запрашиваемое значение. В ходе разрешения зависимостей, ЗА отрабатывают, и в следующую зависимость подставляется результат их выполнения. В случае использования одной зависимости несколько раз, результат будет переиспользован.

4.7 Поиск зависимости, в случае если такая зависимость уже была создана

Зависимости считаются одинаковыми в случае, если их типы, типы и значения их аргументов совпадают. Это определение распространяется и на совпадение значений аргументов, и, следовательно, все дерево аргументов совпадает у одинаковых зависимостей.

Для эффективной и прозрачной работы профилировщика, необходимо:

1. переиспользовать результаты в случае передачи одного и того же экземпляра объекта зависимости

2. при конструировании нового экземпляра, аналогичного существующему, возвращать существующий объект

Существенных вопросов в реализации первого требования не возникло.

Рассмотрим подробнее второе требование

4.7.1 Необходимость автоматического поиска существующих аналогичных экземпляров зависимостей

Гипотетически можно считать зависимости одинаковыми только в случае, если их экземпляры совпадают. Однако это вызывает неудобства в описании зависимостей.

В следующих случаях было бы удобно автоматически определять, что зависимости одинаковые, даже в случае, когда это два различных экземпляра класса:

1. На каждую логическую операцию пораждается новый экземпляр зависимости. В случае, когда аргументом является сложная логическая конструкция, неудобно отдельно сохранять и в дальнейшем переиспользовать экземпляры зависимостей отражающие различные стадии конструирования всего выражения

2. Аналогично зависимостям логических выражений, выражения с использование конструкторов классов крайне коротки и могут быть вложены друг в друга, что делает неудобным их ручное переиспользование.

3. Одна и та же проверка может быть создана в разных модулях. Лучше не заставлять разработчика проверять, не была ли уже где-то использована именно такая зависимость, если этого можно избежать.

4.7.2 Проверка наличия аналогичной зависимости при создании нового экземпляра зависимости

Для решения задачи поиска аналогичной зависимости был придуман метод последовательного хеширования атрибутов зависимости.

Для каждого объекта зависимости создается строка, на основе класса зависимости и переданных в конструктор аргументов. Строка представляет из себя конкатенацию с имени класса, хешей его аргументов, упорядоченных особым образом, и специальных символов.

Безымянные аргументы идут по порядку. Именованные аргументы идут за безымянными в лексикографическом порядке.

Рассмотрим, каким образом получается хеш от аргументов различного типа.

зависимость-класс описано выше

зависимость-функция вместо имени класса, берется название функции

json-тип возвращается хеш от json-представления данных, с упорядоченными в лексикографическом порядке ключами ассоциативных массивов

lambda-фунцкия есть желание производить объединение одинаковых зависимостей даже в случае, если они принимают на вход разные экземпляры lambda-функций, имеющих одинаковый код. Однако это сопряжено с различного рода рисками и может работать не детерминировано. В данный момент lambda-функции обладают одинаковым хешем только в случае если это один и тот же экземпляр. Хеш вычисляется на основе названия модуля, первой строки и аргументов функции. Более подробно проблемы связанные с ними будут рассмотрены в разделе сохранения результатов выполнения зависимостей.

Таким образом каждая зависимость получает уникальный идентификатор, который сохраняется в глобальной структуре, и перед созданием новой зависимости, каждый раз, проверяет наличие аналогичной зависимости. Если зависимость с таким идентификатором уже существует, её экземпляр возвращается вместо создания нового.

Такого поведения Python удалось достичь благодаря использованию механизмов метаклассов и перегрузке их методов.

4.8 Дифференциация зависимостей генерирующих данные и проверок

Зависимость которая просто предоставляет данные отличается от зависимости, которая предоставляет данные и может сообщить о проблеме (проверка) в предложенной имплементации посредством подмешивания специального класса в случае множественного наследования. В случае, если MustBeRunned класс обнаружен в цепочке наследования, эту зависимость нельзя выкидывать при оптимизации.

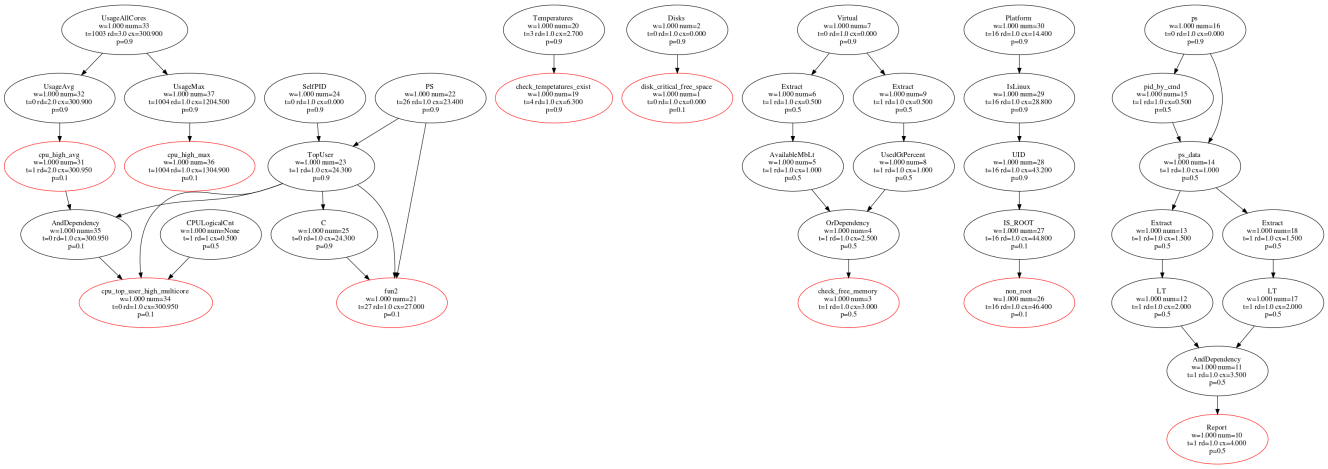

Рис. 1: Пример визуализации графа проверок.

4.9 Некоторые детали реализации

4.9.1 Использование библиотеки numpy

В секции описывающей реализацию алгоритмов планирования используется много векторных операций: побитовое OR, скалярные произведения. Как известно, CPython реализация языка Python3 справляется с этой задачей не эффективно. Для эффективной реализации этих операций предполагалось написать отдельный модуль на языке $C++$ для необходимых векторных операций. Однако было обнаружено, что библиотека для Python3 numpy имеет весь необходимый функционал. Этот модуль был использован при написании библиотеки.

4.9.2 Визуализация графов зависимостей, graphviz

В определённый момент стало невозможно анализировать ход исполнения графов зависимостей по логам, и было принято решение их визуализовать. Визуализация графов, а вернее автоматическое расположение узлов и связей – очень сложная задача, однако решённая. Для визуализации был написан небольшой модуль, генерирующий описание графов в формате graphviz, и использующий утилиту dot для генерации графов в формате svg.

На Рис 1 приведен пример пример графа зависимостей, используемого для проверки алгоритмов планировщика.

5 Реализация: планировщик

Прежде чем рассматривать конкретные модели планировщика, стоит обозначить, какие задачи будут стоять перед ним.

Планировщик занимается разработкой порядка запуска (маршрута) зависимостей.

От маршрута, составленного планировщиком, требуется лишь одно свойство: быть оптимальным с точки зрения выбранной метрики времени исполнения. Более подробно метрики будут рассмотрены дальше.

Порядок обхода зависимостей учитывает, что зависимости:

1. выполняются разное время[10]
2. принимают положительное значение с разной вероятностью, информация о которой предоставляется
3. сообщают об ошибке с разной вероятностью, информация о которой предоставляется
4. зависят от зависимостей, которые может не обязательно запускать, в некоторых случаях
5. могут быть востребованы большим/малым числом других зависимостей
6. могут быть важными/неважными[11]

Таким образом хороший планировщик позволяет сократить время проведения проверок.[12]

[10] Информация о времени исполнения, вероятностях срабатывания и прочее собирается отдельно и считается достоверной. Более подробно это будет описано далее.

[11] Важность проверки определяется на этапе её описания, является числом и говорит о том, насколько критичная проблема проверяется.

[12] Ускоряется процесс запуска проверок в целом. Так же, в зависимости от метрики, планировщик может создавать план, нацеленный на раннее исполнение большинства проверок, или раннее исполнение самых важных проверок...

5.1 Дополнительная информация для алгоритмов минимизации времени

В предыдущих секциях уже упоминалась дополнительная информация для модели планировщика. В этой секции мы рассмотрим подробнее какую дополнительную информацию предлагается использовать и подходы к сбору этой информации.

5.1.1 Подход к сбору информации

В данной работе предполагается что продукт, который может быть создан на базе данного исследования, распространяется свободно, и его пользователи заинтересованы делиться информацией о запусках приложения, собирается вся доступная информация, далее, производятся предрасчёт данных (работает offline планировщик) для упрощения работы планировщика.

Для реализации этого механизма необходимо иметь сервер, принимающий информацию о новых измерениях, и выдающий последний предрасчитанный план выполнения.

Такой подход помогает уменьшить время работы планировщика на основе статистики, и перенести тяжелые с точки зрения вычислительной мощности оптимизации в единый центр, избавив пользователей приложения от необходимости тратить на это свои ресурсы.

5.1.2 Собираемые данные

Для возможности в последующем замены алгоритма планирования и анализа его эффективности, как уже было сказано, предлагается использовать две фазы сбора данных:

1. Сбор всей доступной информации с минимальными потерями

2. Предрасчёт значений для конкретной модели

На первом этапе сбора информации необходимо абстрагироваться от конкретной модели планировщика, и просто сохранить всю информацию, которая

может оказаться полезной. Так же, необходимо тщательно подойти к сохранению анонимности, т.к. предполагается сделать собранные данные общедоступными.

Желательно собирать как можно больше информации, чтобы, при необходимости, быстро расширить модель и ввести новые виды оптимизаций.

Вот список параметров которые пердлагается собирать:

1. Информация о системе
    a) модель процессора
    b) операционная система
    c) версия ядра
    d) версии системных библиотек
2. информация о запуске
    a) порядок запуска проверок
    b) время исполнения зависимости
    c) положительный ли результат исполнения зависимости
    d) сработала ли проверка
    e) путь обхода дерева зависимостей
3. информация о профилировщике
    a) версия профилировщика
    b) алгоритм планировния
    c) используемая версия предрасчитанных данных
4. другая информация

5.1.3 Сопоставление зависимостей и их описания по собранным данным

Восстановление осуществляется с помощью хешей, однозначно идентифицирующих каждую зависимость. В предыдущей секции рассказывалось то, каким образом вычисляется хеш от зависимости.

Информация о зависимостях сохраняется и восстанавливается по тем же самым ключам, которые использовались для поиска аналогичных зависимостей при инициализации.

Вот интересующие нас свойства хеша:

1. зависит от всего дерева аргументов
2. не меняется от запуска к запуску и при перемещении на другую машину
3. вычисляется на основе уникальной строки, описывающей данную зависимость, при помощи криптографический хеш-функции

Зависимость от всего дерева аргументов позволяет отдельно собирать статистику по одинаковым типам аргументов, принимающим на вход различные данные.

Свойство неизменности при изменении внешних параметров является необходимым для сбора и анализа статистики.

Описанный подход к вычислению хеш-значения, позволяет избежать коллизий для количества уникальных вершин порядка $\sqrt{256^{n_{bytes}}} = 16^{n_{bytes}}$ [5]. Для целей работы, было выбрано $n_{bytes} = 8$, чего должно хватить с запасом.

5.2 Предрасчитанные значения и их применение

Вторая фаза сбора данных заключается в предрасчёте данных на которые удобно опираться планировщику при построении оптимального пути обхода дерева зависимостей.

В зависимости от конкретной модели и планировщика потенциально требуются различные предрасчитанные значения, однако в этой секции предлагается абстрагироваться от конкретного планировщика и сфокусироваться на ток какая информация и для чего могла бы быть потенциально использована при построении пути обхода дерева.

5.2.1 Время исполнения зависимости

Как уже отмечалось ранее, такие параметры как время исполнения должны сохраняться отдельно для зависимостей с различными аргументами.

Время исполнения зависимости прямым образом влияет на маршрут, который строит планировщик. Самый простой пример, который можно привести, это проверка, зависящая от двух зависимостей. Если вероятности срабатывания этих зависимостей одинаковый, то очевидно, что математическое ожидание времени исполнения проверки будет меньше, если начать исполнение с ветки которая исполняется быстрее.

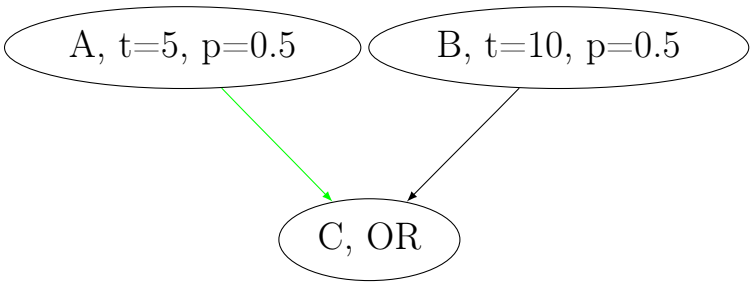

Рис. 2: Использование информации о времени исполнения зависимости.

5.2.2 Вероятность срабатывания проверки

Эту метрика не зависит он вероятности добраться до проверки. Вероятность срабатывания говорит о том, сколько раз проверка сработала по отношению к тому сколько раз она была запущена. Стоит обратить внимание на то, что в некоторых случаях проверка могла вообще не запускаться, такой исход не оказывает влияния на данную метрику.

Проверки с более низкой вероятностью срабатывания должны исполняться позже, чем проверки с высокой вероятностью срабатывания. Этот принцип обеспечивает ускорение работы профилировщика в целом.

5.2.3 Вероятности положительного значения результата зависимости

Некоторые проверки могут иметь бинарные зависимости, которые гарантируют невыполнение проверки.

Данная статистика может быть использована в бинарных операциях и при проверке необходимых условий проверки при построении плана исполнения.

Например, для бинарной операции AND, предпочтение лучше отдавать зависимостям с низкой вероятностью, чтобы сократить математическое ожидание времени исполнения.

Пример. Проверка требует одновременно отсутствия свободной оперативной памяти и количества запущенных процессов больше 10000. В таком случае, поскольку количество запущенных процессов редко превосходит 10000, эта зависимость является более селективной, что играет в её пользу в смысле приоритета исполнения. С другой стороны, проверка свободной оперативной памяти может быть востребована для отсечения других проверок, что повышает вероятность её исполнения.

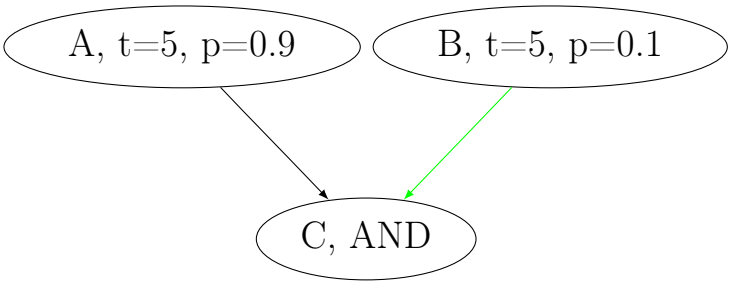

Рис. 3: Использование информации о вероятности принятия возвращаемым значением положительного значения

5.2.4 Количество проверок которым может понадобиться зависимость

В случае, если все прочие условия (время исполнения и вероятность положительного исхода) одинаковы, но при этом одна из зависимостей стоящих на пути к проверке может так же быть использована в другой проверке, имеет смысл отдать предпочтение ей.

На рисунке проверки будут обозначены красными кружками.

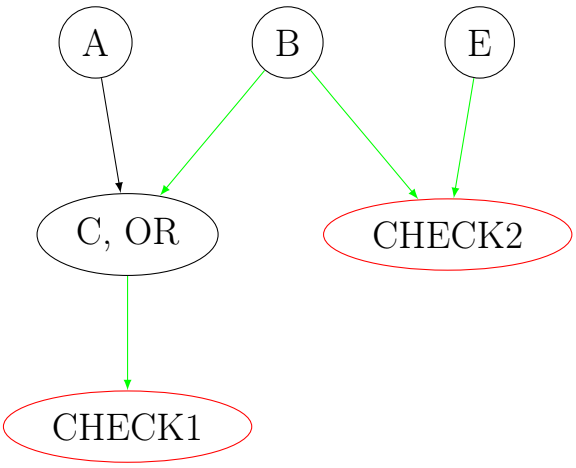

Рис. 4: Использование информации о вероятности принятия возвращаемым значением положительного значения

5.3 Оптимизации

Процесс запуска проверок очень прост и состоит из следующих шагов:

1. выбор следующей З
2. запуск основной процедуры зависимости и сохранение результата её исполнения
3. при отсутствии неисполненных проверок - окончание работы

Главная цель данного исследования это реализовать гибкий подход, который работает эффективно на любом ГЗ и предоставляет удобный способ к добавлению новых.

Изменение главной процедуры проверки может повлиять на результат ее исполнения, поэтому этот вариант оптимизации не рассматривается.

Оставшихся переменных на которые можно повлиять совсем немного, рассмотрим их все. Вот список всего на что можно повлиять в данной модели:

1. некоторые зависимости могут быть опушены в случае если их запуск заведомо не влияет на результат (результат их исполнения не используется)
   a) обнаружена фатальная проблема
   b) проблема имеет необходимое условие, принявшее отрицательное значение
   c) зависимость не самостоятельная и требуется только в проблемах на результат которых не влияет
2. порядок обхода дерева зависимостей может быть выбран любым из допустимых

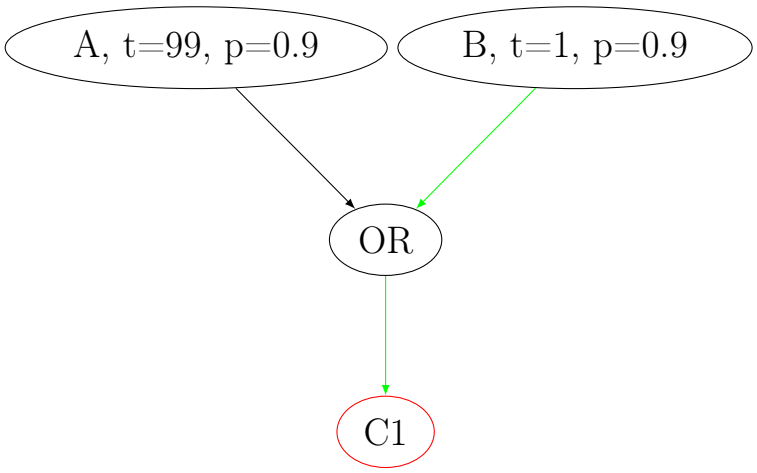

Рис. 5: Пример зависимостей, которые можно не исполнять.

На рисунке представлена ситуация, в которой в большинстве случаев значение OR можно определить вычислив лишь значение B, и пропустив вычисление долгой зависимости A.

Можно придумать много проверок, соответствующих схеме изображенной на рисунке. Примером может быть проверка на влияние cache-misses в приложении которое потребляет много CPU. Сложный и долгий анализ на cache-miss не будет запущен вовсе, если рассматриваемый процесс не потребляет много CPU.

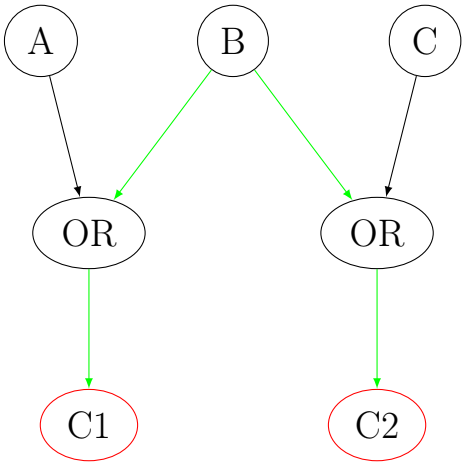

Рис. 6: Выбор с учётом переиспользования.

На рисунке представлена ситуация, в которой при прочих равных условиях, исполнение необходимо начинать с зависимости B, чтобы максимально переиспользовать результаты зависимостей.

5.4 Метрика времени исполнения

Одна из целей работы заключается в том, чтобы ускорить обход ГЗ, поэтому необходимо уточнить что считается ускорением. Необходимо ввести метрику времени исполнения.

Хочется отметить, что определение метрики существенно влияет на план исполнения.

Вот наброски характеристик, на которые можно влиять метрикой:

1. срабатывание более важных проверок в начале исполнения
2. запуск более быстрых проверок в начале
3. уменьшать математическое ожидание среднего времени срабатывания
4. минимизировать математическое ожидание срабатывания одной следующей проверки
5. другие варианты...

Сценарий использования профилировщика предполагает запуск его на машине, на которой работает софт в который по неизвестным причинам работает не так, как ожидается, либо для профилактики.

Желаемый результат это получение необходимой для устранения проблемы диагностической информации через минимальное после запуска профилировщика время.

Для наиболее эффективного решения задачи нам необходимо выбрать разумную метрику которая:

1. напрямую связана с интересующей нас характеристикой
2. не слишком сложна для анализа и реализации
3. существует реализация профилировщика, рассчитывающего последовательность обхода дерева зависимостей за разумное время [13]

[13] Под разумным временем в первую очередь подразумевается сопоставимость времени работы планировщика со временем нахождения первой проблемы. Однако поскольку такое определение тяжело формализовать – будет рассмотрено более общее определение. На количестве узлов порядка 10000 алгоритм планирования должен работать меньше секунды для режима online, и меньше суток для предрасчёта в один поток.

5.5 Рассматриваемые варианты метрик

Введем необходимые обозначения:

$$D \text{ – множество всех зависимостей}$$

$$d \text{ – зависимость}$$

$c \in D$ – зависимость, которая является проверкой и может сообщить о проблеме

$$I \subseteq D \text{ – множество зависимостей сообщивших о проблеме}$$

$$t_i \text{ – время исполнения проверки } i$$

$$p_i \text{ – вероятность положительного результата проверки } i$$

$$m \text{ – значение метрики}$$

$$s = d_1, d_2..., d_n, \text{ путь обхода дерева зависимостей}$$

Время исполнения проверки t_i и вероятность положительного результата p_i считается известным и берется из информации о предыдущих запусках исполнителя.

5.6 Минимизация времени обнаружения следующей проблемы

Один из подходов к обходу дерева зависимостей заключается в том, чтобы минимизировать время нахождения каждой следующей проблемы, т.е. математическое ожидание этого времени (далее $< t_{next} >$ или m).

Следует оговориться, что существует исход, при котором следующая проблема не будет обнаружена вовсе. Обнаружение этого события предлагается учитывать так же как и любое другое, по его вероятности и времени, которое необходимо затратить на его достижение.

На понятийном уровне, минимизация такой метрики приводит к тому, что каждая следующая проблема обнаруживается быстрее. Хочется так же отметить, что вектор исполнения графа смещается не только к одиночным проверкам, которые не исполняются быстро, но, так же и к группам проверок, стоящим рядом. Это достигается по той причине, что данная метрика учитывает

и непредвиденные варианты развития событий, на случай, если не сработает проверка к которой вел основной план.

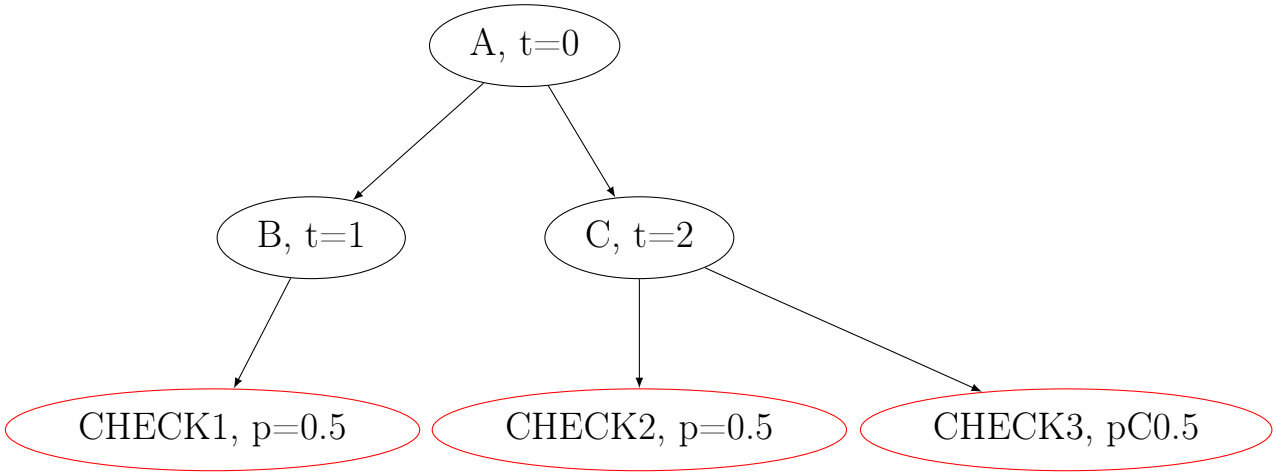

Рис. 7: Минимизация времени обнаружения следующей проблемы, выбор между одной близкой и двумя далёкими проверками.

Рассмотрим пример расчета математического ожидания для простого ГЗ:

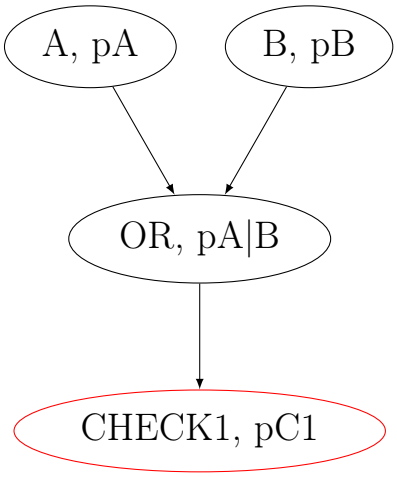

Рис. 8: Демонстрация метрики минимизирующей время обнаружения следующей проблемы.

На рисунке изображен пример простой топологии дерева зависимостей. Попытаемся выразить математическое ожидание времени обнаружения следующей зависимости для всех возможных маршрутов:

$$m_{AB} = p_A(t_A + t_C) + (1 - p_A)p_B(t_A + t_B + t_C) + (1 - p_A)(1 - p_B)(t_A + t_B)$$

$$m_{BA} = p_B(t_B + t_C) + (1 - p_B)(t_B + t_A + t_C) + (1 - p_B)(1 - p_A)(t_B + t_A)$$

Здесь m_{AB} и m_{BA} в обозначают математические ожидания первых сработавших проверок в случае желаемых (от которых возможно придется отклониться) маршрутов $A \to B \to C$ и $B \to A \to C$ соответственно.

Добавим ещё одну проверку в граф:

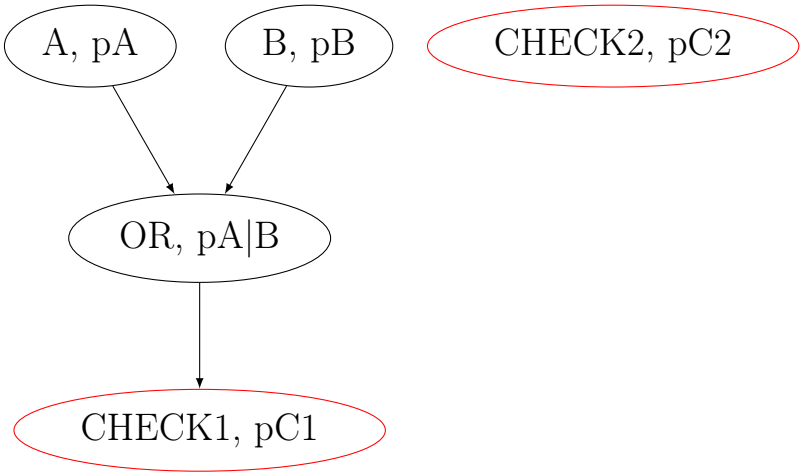

Рис. 9: Демонстрация метрики минимизирующей время обнаружения следующей проблемы. Увеличение вариантов обхода с добавлением проверки.

Рассмотрим варианты обхода такого ГЗ:

1. A->C2 (Если B - дорогое, продолжаем с C2)
2. A->C2->B
3. A->C2->B->OR->C1
4. A->B->C2
5. A->B->OR->C2
6. A->B->OR->C1->C2
7. B->C2
8. B->C2->A
9. B->C2->A->OR->C1
10. B->A->C2
11. B->A->OR->C2
12. B->A->OR->C1->C2
13. C2
14. C2->A
15. C2->A->C1
16. C2->A->B
17. C2->A->B->C1
18. C2->B
19. C2->B->C1
20. C2->B->A
21. C2->B->A->C1

Такой простой граф зависимостей имеет 21 вариант обхода.

Как строится вариант обхода:

1. Выбрать следующую зависимость из доступных
2. Исполнить зависимости
3. По результатам исполнения и топологии, выбросить зависимости, исполнение которых не потребуется
4. Вернуться к пункту 1

Количество вариантов при выборе следующей зависимости к исполнению на каждом шаге ограничено сверху количеством неразрешенных зависимостей. Это дает факториальную оценку количества вариантов.

При обычном обходе ГЗ мы могли бы выбирать предпочтительный путь при получении новых данных от исполнения зависимостей. При оценке математического ожидания времени мы должны учитывать все варианты развития событий с соответствующими вероятностями.

Это значит что мы не можем просто рассчитать по очереди $< t_{next} >$ для каждого из факториала возможных вариантов обхода ГЗ, а затем выбрать путь с наименьшим значением. В следствие неопределенности исхода зависимостей, на каждой развилке будет существовать вероятность что продолжать обход по этому пути не получится, и необходимо учитывать другие пути.

Таким образом, говоря m для пути мы подразумеваем не просто путь, обход графа по которому имеет найденное значение математического ожидания, но семейство путей, по одному пути для каждого из возможных вариантов развития событий.

Запишем формулу математического ожидания $< t_{next} >$, или m для общего случая графа зависимости.

Пусть есть множество вариантов исходов всех зависимостей: Q. Элементом этого множества является фиксированный набор исходов зависимостей. $q_i \in Q : q_i = (d_1 = c1_i, d_2 = c2_i ...)$.

Составным путем j назовем функцию, которая сопоставляет простой путь s_{ij}[14] каждому варианту исхода q_i. Т.е. $j \in J, j : Q \longrightarrow S_j; s_{ij} = j(q_i)$. Поиск j_{min}, соответствующего минимальному m_j и есть задача минимизации метрики.

[14]Подразумевается, что s_{ij} заканчивается в первой сработавшей на своем пути проверке.

Каждому значению q_i соответствует $p_i = p(q_i) \in P$ – вероятность наступления события q_i.

Тогда m_j определяется просто:

$$m_j = \sum_{q_i \in Q} t(j(q_i))p(q_i) = \sum_{q_i \in Q} t_{ij}p(q_i) \tag{1}$$

Здесь использовалось обозначение $t(s_{ij})$ – время пути s_{ij}.

Тогда искомая j_{min} находится следующим образом:

$$j_{min} : m_{ij_{min}} = \min_{j \in J} m_j \tag{2}$$

Минимизация данной метрики является NP полной задачей.[15] В этой работе не рассматривается вопрос изменения маршрута при непосредственном обходе графа (он должен пересматриваться после исполнения каждой зависимости), однако кратко пометим, что эта задача так же является NP сложной, или требует предрасчёта и факториального количества памяти.

5.7 Минимизация ассоциативного пенальти до следующей проблемы

Предыдущую задачу минимизации метрики можно упростить путем замены метрики в виде времени на ассоциативное пенальти.

На понятийном уровне, пенальти так же зависит от времени и стимулирует скорейшее исполнение ближайшей проверки. Для некоторых случаев, в которых приходится выбирать между количеством и удаленностью решение планировщика основывающегося на другой метрике может поменяться, и возможно в не лучшую сторону, однако это может положительно сказаться на времени работ самого планировщика.

Рассмотрим пример:

[15] В данной работе факт принадлежности этой задачи к классу NP не доказывается. Однако отметим, что задача обладает явными признаками NP полной задачи. Наличие не полиномиального количества вариантов, а так же необходимость учитывать параметры уже построенного маршрута при его последовательном "наращивании".

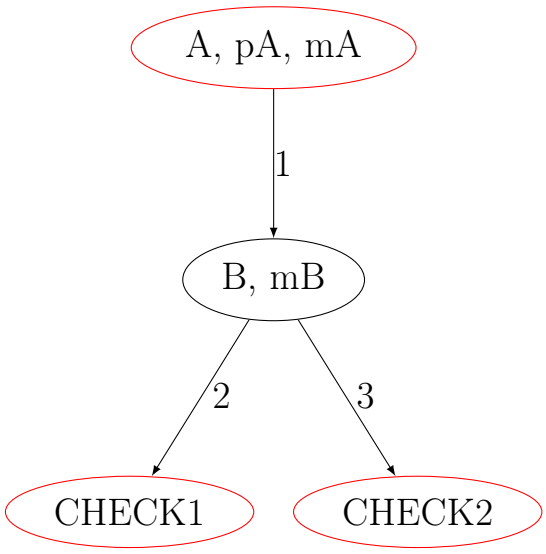

Рис. 10: Демонстрация метрики минимизирующей ассоциативное пенальти обнаружения следующей проблемы.

Пусть мы зафиксировали $j \in J$ составной путь, проходящий через проверку A. Процесс расчета метрики бы ускорился, если бы значение $m_{start=B} = mB$ можно было использовать для нахождения mA. В предыдущей секции время $t_{ij} = t(s_{ij})$ линейно входило в математическое ожидание (1)[16], что не позволяло быстро пересчитать всю сумму целиком. Для быстрого пересчета всей суммы с добавлением новой зависимости, через которую проходят все последующие требуется функция времени, удовлетворяющая следующему условию[17]:

$$f(t_A) \cdot \sum_{q_i \in Q} f(t_i) p(q_i) = \sum_{q_i \in Q} f(t_A + t_i) p(q_i) \tag{3}$$

Следовательно

$$f(t) = C_t^t \text{– степенная функция времени, } C_t \text{ – константа.} \tag{4}$$

Хоть пенальти такого вида помогает быстрее рассчитывать математическое ожидание метрики, оно навязывает очень сильное влияние времени пути к зависимости на её вклад в метрику.

С таким пенальти существует только одно значение $t = \log_{C_t}(2)$, такое, что $f(t) * 2 = f(2 * t)$, т.е. обнаружение двух зависимостей через время $2t$ имеет такой же вес как и обнаружение одной через время t.

[16]Имеется в виду, что $t_{ij} = \sum_{d_k \in s_{ij}} t(d_k)$ определялась как сумма времен исполнения всех зависимостей на пути.

[17] Рассматривались только варианты, в которых функция времени входит в формулу метрики, как произведение, т.к. они не должны нелинейным образом влиять на p_i.

Введение такого пенальти, как и другие способы ускорить расчет метрики не позволяют вывести задачу из NP класса, т.к. это не избавляет от рассмотрения факториала потенциальных вариантов обхода ГЗ.

5.8 Минимизация суммы времен обнаружения проблем

Данная метрика как и две предыдущие не может быть оптимизирована за разумное время.

Рассмотрим метрику времени исполнения профилировщика которая равна сумме времен в которые проблемы были обнаружены.

В отличии от предыдущих метрик, простой путь s_{ij} на составном пути j не прерывается при достижении сработавшей проверки.

На понятийном уровне, эта метрика стимулирует исполнение в начале групп быстро запускающихся проверок, и оставляет на конец долгие, стоящие удаленно от остальных, проверки.

Обозначим значение метрики на пути s_{ij} как m_{ij}, тогда:

$$m_{ij} = \sum_{d_k \in I} t_{ij}(d_k) \tag{5}$$

Здесь I – множество сработавших проверок, а $t_{ij}(d_k)$ – момент времени, в который была обнаружена проблема d_k при обходе по пути s_{ij}.

Так же как и в прошлых метриках, на самом деле мы имеем дело с составными путями j, и для расчета математического ожидания, необходимо учесть вероятности перехода на каждый из простых маршрутов.

$$m_j = \sum_{d_k \in I} m_{ij} * p(s_{ij}) \tag{6}$$

Аналогично предыдущим метрикам, оптимальному пути соответствует составной путь j, при котором достигается минимальное значение метрики m_j.

5.9 Эмпирические алгоритмы

Как видно из предыдущих секций, точный расчет значения метрики и её минимизация – крайне сложная задача, которая не может быть эффективно решена

и относится к классу NP из-за факториального количества вариантов и условий, накладываемых на маршрут (учет пройденных маршрутов, их связей).

Далее будут рассматриваться варианты более простые с точки зрения реализации и потенциально менее эффективные с точки зрения оптимизации ГЗ.

5.10 Вспомогательные алгоритмы

Перед тем как рассматривать конкретные алгоритмы, необходимо познакомиться с их составными частями – вспомогательными алгоритмами.

5.10.1 Алгоритм обхода графа в глубину с синхронизацией

Этот алгоритм был использован для уменьшения алгоритмической сложности распространения и аккумуляции информации о графе по его узлам.

Рассмотрим простой пример ГЗ:

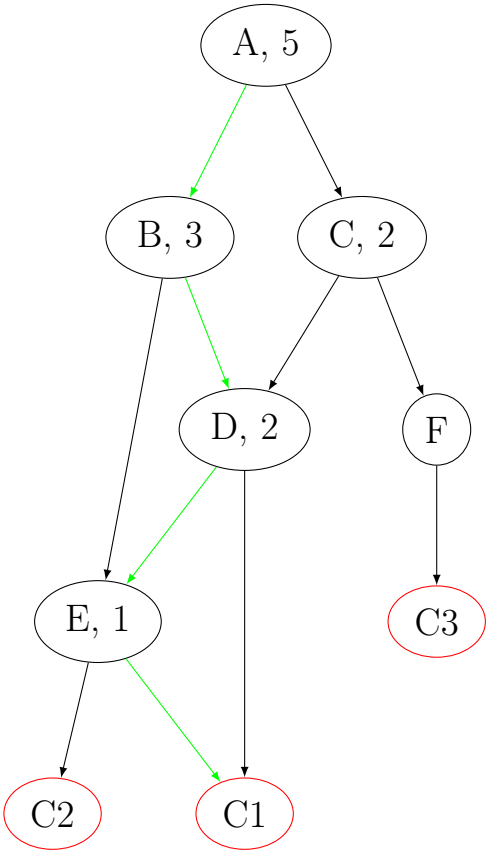

Рис. 11: Распространение информации в графе при обходе в глубину.

Предположим перед нами стоит задача "размазать"веса всех проверок по зависимостям, от которых они зависят, и тем самым рассчитать некий вес зависимостей. Когда в цепочке зависимостей встречается разветвление – вес про-

верки распределяется между ними, а при соединении веток – суммируется в родительской.

На Рис. 11 изображено распространение веса проверки $C1$ по ГЗ при классическом обходе в глубину. Как видно, при этом зависимость A была посещена 5 раз. Для произвольной топологии графа, количество посещений зависимостей при обходе в глубину не ограничивается полиномом.[18]

Для более эффективного обхода ГЗ предлагается использовать алгоритм, который в последствии будет называться, как "обход с синхронизацией который представляет из себя что-то среднее, между обходом в глубину, в ширину и поиском "лучший вперед".

Суть его заключается в следующем:

1. Внести в список активных узлов все зависимости первого (последнего) уровня
2. Достать из списка первый узел
3. Распространить информацию по всем дочерним, увеличивая число посещений в каждой из них на 1
4. При достижении числом посещений числа связей в дочернем узле – внести её в список активных вершин
5. Перейти к шагу 2, если список не пуст
6. Завершить исполнение

Такой алгоритм имеет сложность $O(m)$, где m – количество ребер в ГЗ.

Кроме того, хочется отметить, что такой алгоритм позволяет разнести информацию обо всех зависимостях за один проход.

Алгоритм может быть использован для распространения информации в обоих направлениях: к родителям и к детям.

[18]Точное ограничение с верху посчитать тяжело, однако можно привести примеры в которых эта зависимость степенная. Например две параллельные цепочки зависимостей, зависимости на каждом уровне которых зависят от зависимостей из своей и соседней цепочек предыдущего уровня.

5.10.2 Расчёт веса зависимости

При построении эмпирического алгоритма планирования, существует необходимость стимулировать переиспользование результатов зависимостей.

Рассмотрим следующую ситуацию:

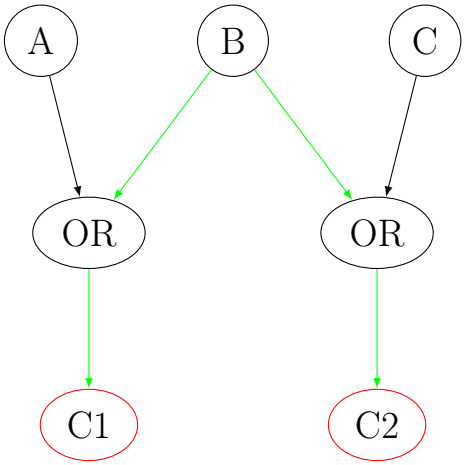

Рис. 12: Переиспользование зависимости

На рисунке изображена ситуация, в которой при прочих равных условиях имеет смысл начинать исполнение с зависимости B, потому, что её результат может пригодиться сразу для двух проверок, и существует вероятность, что исполнение зависимостей A и C может не потребоваться вовсе.

Рассмотрим алгоритм, который мог бы помочь решить эту проблему. Для начала введем несколько понятий:

Вес проверки Число, обозначающее значимость реультата проверки для пользователя. Пример: провека на возможность взломать систему через известную уязвисомть более весомая, чем проверка количества разбитых из-за неподходящего размера MTU пакетов.

Вес зависимости Число, выражающее одновременно количество и важность проверок, в которых задействована (прямым и косвенным образом) данная зависимость. Это определение не является строгим, и позволяет в разных случаях иметь разные формулы для точного её доопределения при сохранении общей сути. Сложность может заключаться в том, что вес зависимости может отражать вес проверок, которые пройдут через эту зависимость лишь с определённой вероятностью, но не гарантированно.

В основе алгоритма лежит "обход графа с синхронизацией"из предыдущей секции.

Для аккумуляции весов проверок, будет использоваться массив чисел с плавающей точкой. Числа в этом массиве будут соответствовать весам проверок дошедшим до данной зависимости.

Описание алгоритма:

1. Начать исполнение алгоритма "обход с синхронизацией"
2. Исполнение 3 пункта алгоритма с синхронизацией:
   a) В случае, если данный узел является проверкой – прибавить в свой вектор весов вес своей проверки.
   b) Расчет векторов весов, которые будут переданы зависимостям (родительским)[19]
   c) Добавление к векторам весов зависимостей, весов текущей вершины
3. Суммирование элементов векторов весов в каждой зависимости для расчета веса зависимости.

Данный алгоритм описывает весь процесс за исключением расчета векторов весов для зависимостей (родительских вершин). Остановимся немного подробнее на том, какими способами можно это делать:

При равномерном распределении веса текущей зависимости делятся в равных долях между всеми родительскими зависимостями. Сам по себе такой подход не позволяет выделить те зависимости, которые наиболее важны для исполнения данной. Однако применяя такое распределение в совокупности с другими подходами, можно добиться желаемого результата.

Распределение в соответствии с вероятностями срабатывания подразумевает то, что, например, в случае с OR зависимостью, будет отдаваться больше веса в ветвь с большей вероятностью положительного результата, т.к. это будет увеличивать вероятность того, что не придется выполнять две ветви.

Сложность данного алгоритма составляет $O(n * m)$, где n –количество вершин, а m – количество ребер ГЗ.

[19]Расчет векторов весов родительских зависимостей может быть выполнен по разному в зависимости от конкретной задачи

5.10.3 Расчёт усредненного времени исполнения зависимости

Усредненное время исполнения зависимости – Число, равное статистическтму времени исполнения зависимости, деленному на количество использований её результата. $< t_i >= t(d_i)/n$.

Для начала, для простоты, будем полагать, что наличие зависимости в любой из ветвей исполнения проверки является её использованием.[20]

Этот алгоритм очень похож на предыдущий. В данном случае, необходимо избежать учитывания одной проверки для зависимости несколько раз, в случае, если к ней ведут несколько маршрутов. Для этого будем использовать битовый массив, положительный бит номер i которого говорит о том, что данная зависимость используется для вычисления проверки номер i.

Описание алгоритма:

1. Начать исполнение алгоритма "обход с синхронизацией"
2. Исполнение 3 пункта алгоритма с синхронизацией:
   a) В случае, если данный узел является проверкой – выставить в своем векторе проверок $true$ в соответствующую позицию.
   b) Применить операцию "побитовое или"с векторами проверок родительских зависимостей
3. Суммирование положительных элементов вектора проверок даст количество проверок в которых участвует данная зависимость.[21]
4. Деление времени исполнения зависимостей взятое из статистических данных на расчетное количество проверок, дает среднее время исполнения

Сложность данного алгоритма составляет $O(n * m)$, где n –количество вершин, а m – количество ребер ГЗ.

В последующем будет предложено решение, избавляющее этот алгоритм от того, что в делитель времени входят проверки, которые, вероятно, не будут проходить через зависимость.

[20] Т.е. опускается тот факт, что проверка может быть проверена в итоге по другому пути и зависимость на самом деле не будет использована проверкой.

[21] В зависимости от задачи, можно так же учесть важности проверок, которые проходят через данную зависимость, посредством скалярного произведения вектора проверок на вектор весов проверок.

Этот алгоритм, как и предыдущий, помог бы в примере, приведенном на Рис. 13. Однако эти расчетные значения предполагается использовать для расчета времени, необходимого на исполнение ветвей целиком. То, каким образом это будет достигнуто, рассматривается в самих эмпирических алгоритмах планирования т.к. это прямым образом влияет на качества планировщика, и в меньшей степени относится к алгоритмам.

5.11 Факторы, учитываемые эмпирическим алгоритмом

Эмпирический алгоритм не оптимизирует конкретную метрику, вместо этого он предугадывает, какой путь окажется более удачным, с учетом возможного развития событий в будущем, и смещает вектор исполнения ГЗ в соответствующую сторону.

Вот список факторов, которые должен учитывать хороший эмпирический планировщик:

1. Переиспользование результата зависимостей
2. Выбор ветки с учетом времени её исполнения
3. Выбор ветки с учетом вероятности её исполнения
4. Учет весов проверок
5. Исполненные к текущему моменту зависимости исполняются мгновенно[22]
6. Учет результата уже исполненных проверок[23]
7. Исполнять зависимость только тогда, когда стало известно, что её результат будет использован[24]

[22]Под этим подразумевается то, что если существует группа зависимостей расположенных близко друг к другу с точки зрения времени исполнения, и одна из них исполнилась, то стоимость соседних стала намного ниже, т.к. большинство их зависимостей уже исполнено.

[23]Пока исполнялись зависимости для одной проверки, могло оказаться что уже достаточно данных, или почти достаточно для разрешения другой проверки (например не выполнилось обязательное условие).

[24]Это в первую очередь подразумевает, что исполнение идет от проверок к зависимостям, а не наоборот. Т.е. выбирается проверка, а затем выбирается путь (зависимости) по которому она проверяется.

Большинство из перечисленных факторов интуитивно понятны и не требуют пояснения. Однако всё не так просто с фактором №5.

Для того чтобы правильным образом учесть пройденные к этому моменту зависимости, необходимо распространить о них информацию по всему дереву. Рассмотрим следующий пример:

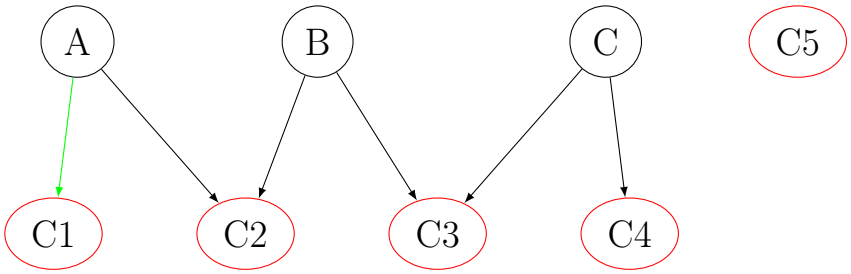

Рис. 13: Учёт посещённых к текущему моменту зависимостей.

В данном случае, например, исполнение A повышает приоритет исполнения всех зависимостей и проверок изображенных на рисунке, кроме $C5$.

Убедимся в этом. Исполнение A уменьшает время исполнения $C2$, что в свою очередь увеличивает приоритет исполнения B. Увеличенный приоритет B говорит о том, что одна из зависимостей $C3$ потенциально исполнится раньше чем планировалось, что отражается на C и соответственно $C4$. Этот эффект заметен на простой конфигурации, однако топология может предполагать и более сложные зависимости.

Далее рассмотрим несколько эмпирических планировщиков и то, какие факторы они учитывают и в какой степени.

Согласно фактору №7, будут рассматриваться только алгоритмы которые начинают обход графа с выбора проверок. Т.е. процесс выбора имеет две составляющие:

1. Выбор следующей проверки

2. Выбор одной из возможных зависимостей при ветвлении, при разрешении зависимостей выбранной проверки

5.12 Жадные алгоритмы выбора следующей зависимости

Рассмотрим несколько алгоритмов планирования, которые принимают решение на основе статистических данных и предрасчитанных в предыдущей секции значений: веса, усреднённого времени исполнения зависимостей.

5.12.1 Веса зависимостей

Рассмотрим следующий алгоритм:

1. Применить алгоритмы из предыдущей секции для расчета весов зависимостей
2. Выбрать проверку с наибольшим весом
3. Выполнять рекурсивно зависимости проверки; при ветвлении – выбирать зависимость с наибольшим весом
4. Повторять пункт 2, пока не закончатся проверки

Преимуществами такого алгоритма является то, что он выполняется быстро, и стимулирует переиспользование результата зависимостей.

Однако, большинство из обозначенных факторов, которые должен учитывать планировщик, этот алгоритм не учитывает. Вот две ситуации, которые будут оптимизированы совершенно неэффективно данным планировщиком:

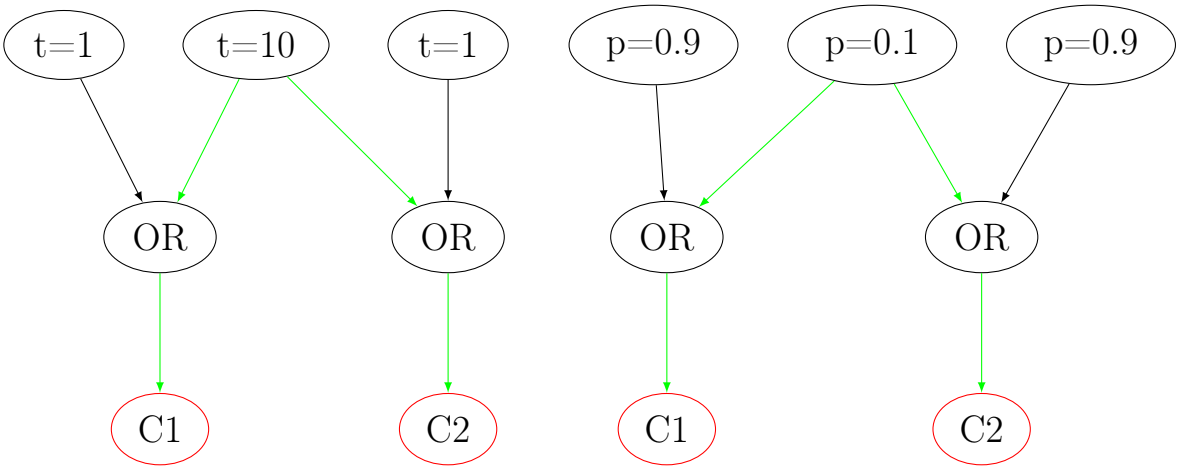

Рис. 14: Неоптимальный план при учете только весов зависимостей

Как видно из рисунка, планировщик отдает предпочтение зависимости, которая используется в двух проверках, не учитывая вероятности и времена исполнения. Для решения проблемы возникшей в этом примере можно было бы добавить учет вероятностей и времен выполнения родительских зависимостей в ветвлении, однако проблема несет более глубокий характер и не может быть так просто решена, в случае, если нужно учитывать более глубоких родителей или боле сложные связи.

5.13 Расчет времени исполнения ветви зависимостей

Предыдущий алгоритм планирования оказался неэффективным из-за того, что не учитывал родительские зависимости глубже чем на один уровень. Для исправления этого недостатка предлагается произвести расчеты дополнительных опорных данных.

Предлагается рассчитать дополнительные метрики, характеризующие время исполнения ветви целиком.

Математическое ожидание времени исполнения ветви – самая очевидная и полезная для принятия решения выборе ветви метрика. Однако для решения этой задачи необходимо найти оптимальный маршрут, и мы не можем использовать эту метрику для нахождения оптимального маршрута.

Минимальное время исполнения ветви – с первого взгляда могло показаться, что эта метрика может быть посчитана простым рекурсивным вызовом, который выбирает путь с минимальным временем при ветвлении, и суммирует ветви при множественной зависимости. Для того чтобы понять, что этого недостаточно, предлагается рассмотреть следующий пример:

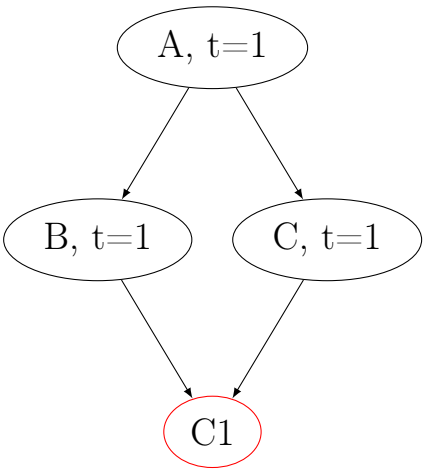

Рис. 15: Неверный расчет минимального времени исполнения

Следуя наивному алгоритму, получаем:

$$t_A = 1$$

$$t_B = t_A + 1 = 2$$

$$t_C = t_A + 1 = 2$$

$$t_{C1} = t_B + t_C = 4$$

По причине двойного учёта зависимости A, получен неверный результат 4 для проверки $C1$.

Чтобы не учитывать зависимости по несколько раз, даже в сложных топологиях, предлагается использовать введенный ранее "обход с синхронизацией". В отличии от предыдущих случаев, где он применялся для прохода от проверок к зависимостям, в этот раз он будет направлен в обратном направлении. Опишем алгоритм:

1. Начать исполнение алгоритма "обход с синхронизацией"с зависимостей, которые не имеют зависимостей; результат аккумулировать в битовый вектор зависимостей, обозначающий необходимые к исполнению на текущем пути проверки

2. Исполнение 3 пункта алгоритма с синхронизацией:

   а) совершить одну из операций с собственным вектором зависимостей

Ничего в случае, если существует развитие событий, при котором значение зависимости заведомо определено (существует необходимое условие, которое может не выполниться)

Запись $true$ в ячейку со своим номером

b) Агрегировать родителей (побитовое OR) в соответствии с типом зависимости

OR/AND выбрать только ветвь с наименьшим временем[25]

Обычная зависимость агрегировать всех родителей

Зависимость с необходимым условием агрегировать только необходимое условие

3. Скалярное произведение вектора зависимостей на вектор усредненных или статистических времен исполнения зависимостей, дает минимальное время пути к зависимости

При подсчете минимального времени предлагается использовать усредненное или статистическое время. Конкретный выбор делается ориентируясь на необходимые свойства конкретной реализации планировщика.

Очевидной проблемой данной метрики является то, что она не учитывает вероятность положительного результата при выборе ветви и обработке зависимостей с необходимым условием. Вот пример, при обработке которого, данная метрика лишь введет в заблуждение алгоритм планирования.

[25]Время рассчитывается путем применения скалярного произведения вектора зависимостей на вектор усредненных или статистических времен исполнения зависимостей

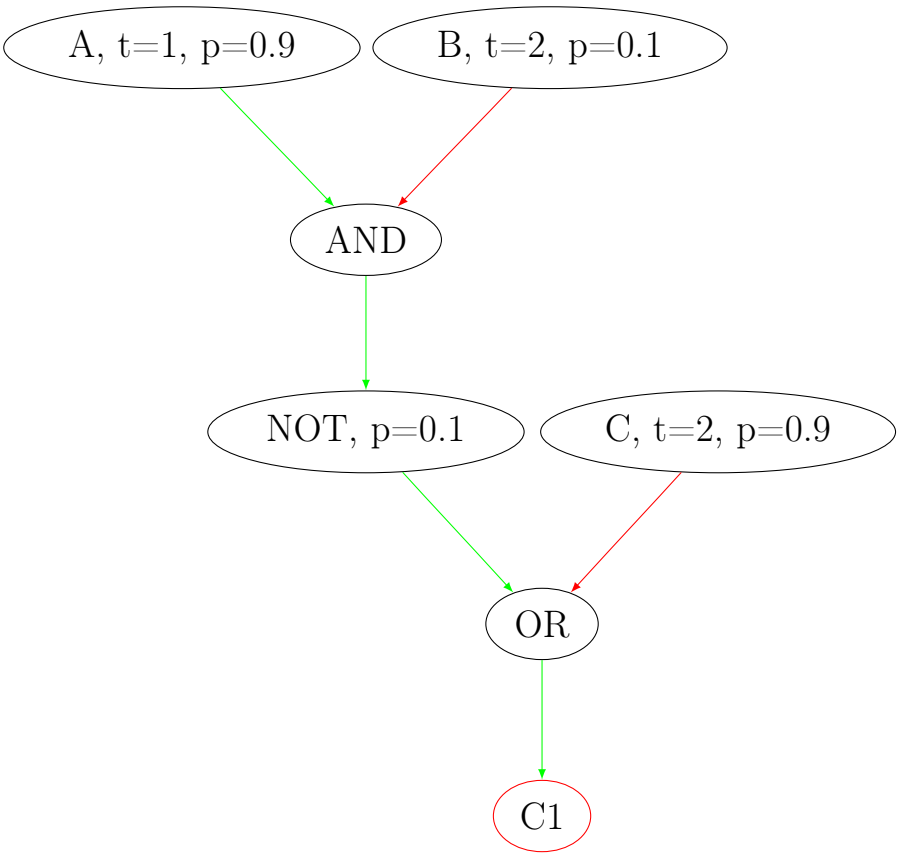

Рис. 16: Обоснование необходимости учета вероятности положительного исхода при расчете времени исполнения ветви.

Как показано на рисунке, планировщик, на основе информации о минимальном времени исполнения ветви принял худшее решение о выборе маршрута из возможных для представленной конфигурации.[26] Этот пример наглядно показывает, что при расчете времени исполнения ветви, необходимо учитывать вероятности положительных исходов.

Вероятное время исполнения ветви не обладает недостатком предыдущей метрики времени исполнения ветви. На понятийном уровне, данная метрика выражает наиболее вероятное время исполнения ветви.

Замечание 1: предлагаемый в этой секции алгоритм рассчитывает приближённое значение.[27]

Замечание 2: При расчете условной вероятности положительного результата

[26] Как и в примере, приведенном ранее, эту проблему нельзя устранить просто начав учитывать вероятности на уровне принятия решения о выборе ветви. Необходимо учитывать вероятности положительных исходов всех уровней аргументов.

[27] Точность оценки не оценивалась, однако есть основания полагать, что точность высокая, т.к. автор не смог придумать конфигурации графа, при которой оценка давала бы качественно неверный результат.

набора ветвей, делается предположение, что исходы различных ветвей независимы.[28] Тогда для условной вероятности положительного исхода одной ветви из набора используется рекурсивная формула:

$$p_{1\ldots n+1} = p_{1\ldots n} + (1 - p_{1\ldots n})p_{n+1} \tag{7}$$

Опишем алгоритм:

1. Начать исполнение алгоритма "обход с синхронизацией"с зависимостей, которые не имеют зависимостей; результат аккумулировать в битовый вектор зависимостей

2. Исполнение 3 пункта алгоритма с синхронизацией:

   a) совершить одну из операций с собственным вектором зависимостей

      Ничего если вероятность рассчитать значение зависимость без её запуска более 50%

      Запись $true$ в остальных случаях

   b) Агрегировать родителей (побитовое OR) в соответствии с типом зависимости

      OR выстроить ветви в порядке убывания вероятности положительного исхода, выбирать ветви до тех пор, пока суммарная условная вероятность положительного исхода не превысит 50%

      AND обратно случаю OR; ветви сортируются в порядке возрастания вероятности, выбираются до тех пор, пока суммарная условная вероятность отрицательного исхода не превысит 50%

      Обычная зависимость агрегировать всех родителей

      Зависимость с необходимым условием агрегировать необходимое условие, и, в случае когда вероятность условия превышает 50%, все остальные аргументы

3. Скалярное произведение вектора зависимостей на вектор усредненных или статистических времен исполнения зависимостей, дает минимальное время пути к зависимости

[28] Учесть зависимость исходов различных ветвей можно, однако это эффект второго порядка значимости и для качественного анализа его можно опустить.

Рассчитанная таким способом метрика времени исполнения ветви в целом является довольно объективной, потому, что она хоть и приближённо, но отражает реальное значение вероятного времени исполнения в случае, когда ни одна зависимость из рассматриваемой ветви не была исполнена.

Хочется отметить, что данная метрика стимулирует исполнение кластеров проверок (находящихся близко друг к другу), т.к. они уменьшают усреднённое время друг другу и уменьшают ожидаемое время исполнения проверки.

Так же интересно то, что в случае, когда усредненное время исполнения в этом алгоритме нормируется на веса соответствующих проверок, то этот алгоритм начинает стимулировать исполнение важных проверок, т.к. оцененное вероятное время становится меньше (больше) чем реальное.

Этот факт может быть применён в ситуации когда рядом расположено множество похожих проверок, их веса могут быть понижены, чтобы отражать в сумме примерное количество уникальных проверок.

5.14 Итеративный расчёт параметров зависимостей

5.14.1 Расчёт весов

Ранее уже было отмечено, что при расчете весов не учитывался тот факт, что существует вероятность того, что зависимость будет иметь вес в том числе тех проверок, которые вероятно через неё не пройдут. Чтобы решить эту проблему необходимо решить задачу курицы и яйца. Дело в том, что вес зависимости зависит от того, по каким путям будут вычисляться аргументы проверок. Пути вычисления проверок, в свою очередь, зависят от весов зависимостей.[29]

Можно придумать конфигурации графа, которые пользуются этим недостатком и заставляют планировщик создавать неэффективный план. Наиболее яркий из примеров – две долгие по времени, зависимости, которые нужны нескольким проверкам. Обычный подход приведет к стимуляции обоих проверок в одинаковой степени, и в последствии, вероятно, они обе будут исполнены. Однако лучшим вариантом было бы решить, какую из зависимостей выбрать и направить на неё все проверки.

[29]Прямо, или косвенно, через усредненное время исполнения.

Для уменьшения эффекта данной проблемы, предлагается применить итеративный алгоритм расчета весов. Приведем его описание:

1. Рассчитать веса по стандартному алгоритму.
2. Применить алгоритм поиска маршрутов.
3. Рассчитать веса зависимостей с учетом информации о том, какие проверки вероятно будут использовать результат этих зависимостей
4. Повторять пункты 2-3 до прекращения изменений или n раз

В зависимости от того, как представлен маршрут, этот алгоритм может иметь разную алгоритмическую сложность.

Для случая когда маршрут к каждой проверке представлен в виде битового массива зависимостей, результат которых используется, [30] алгоритм, после применения дополнительного "обхода с синхронизацией"для распространения информации об используемых проверками зависимостях по зависимостям, имеет сложность $O(nm)$, где n и m – количество вершин и ребер соответственно.

5.14.2 Расчёт усредненных времен

Итеративный расчет усреднённых времен исполнения зависимостей производится точно таким же методом, как и расчёт весов.

5.15 Взвешенная метрика offline

В предыдущих секциях были рассмотрены все необходимые составляющие для расчета данной метрики.

Модификация[31] этого алгоритма была выбрана как наиболее удачный планировщик.

Перечислим блоки из которых собран алгоритм и их особенности:

1. Исполнение проблем в порядке увеличения и вероятного времени исполнения [32]

[30] Например, получаемый при поиске вероятного времени исполнения

[31] Дополнение оптимизациями не имеющими прямого отношения к алгоритму планирования.

[32] Это автоматически стимулирует исполнение кластеров и важных проверок из-за особенностей расчета вероятного времени исполнения.

2. Расчет усредненных времен исполнения итеративным способом с нормировкой на веса проблем

3. Расчёт вероятного времени исполнения ветвей

4. Выбор ветви при ветвлении на основании вероятностей положительного результата и вероятных времен исполнения (единственная часть алгоритма, которая делается не во время предрасчёта)

Хочется отметить, что алгоритм имеет сложность $O(nm)$ для предрасчёта, и $O(1)$ время принятия решения при выборе направления во время непосредственного исполнения.

Данный алгоритм уже принимает во внимание большинство описанных выше факторов, которые должен учитывать эффективный эмпирический алгоритм. Пока не учитывается результат уже исполненных проверок и то что время их исполнения стало равным нулю.

Распространение информации о выполненных зависимостях Конкретно для этого алгоритма предлагается воспользоваться следующим методом для уменьшения влияния проблемы связанной с нераспространением информации о выполненных зависимостях. Предлагается после выполнения каждой зависимости, просматривать рекурсивно все дочерние зависимости с целью найти те, результат которых мог стать известен без вычисления ГП (зависимости с необходимым условием, OR, AND, NOT, LT...). Это приведет к тому, что после исполнения каждой зависимости, будет необходимо произвести операцию с алгоритмической сложностью $O(k)$, где k – максимальное количество исходящих ребер у одной вершины. В рамках решаемой задачи, k можно считать константой так, как оно зависит от фантазии и возможностей человека, который пишет проверку.

Пересмотр пути в в процессе обхода offline алгоритм рассчитан делать минимум вычислений непосредственно при обходе ГЗ. Существует способ не ухудшив сильно алгоритмическую сложность выбора следующей зависимости избавиться в этом алгоритме от крайних случаев неправильно предсказанного маршрута. В случае, когда при выполнении маршрута вопреки предсказаниям

появляется необходимость исполнять маловероятную, долгую ветвь, исполнение текущей проверки может быть перенесено на потом, за счет понижения её приоритета на величину соответствующую времени исполнения этой ветви.

Проблема связанная с учетом того, что выполненные проверки становятся бесплатными, с точки зрения времени их исполнения, в данном алгоритме остается нерешённой. Однако, хочется отметить, что это повлияет на время проведения всех проверок только в том случае, если пути по которым придётся обходить граф будут отличаться от предрасчитанных вероятных путей. Гипотетически, в системах не бывает ситуации суперпозиции большинства известных проблем одновременно, поэтому вероятные маршруты строятся с расчетом на то, что проблемы нет. Однако, по тем же соображениям, отклонение от вероятных маршрутов будет не сильным (проявится у малого количества проверок), т.к. в системе проблем всегда значительно меньше, чем проверок в базе. Это значит, что результирующее время работы всех проверок, вероятно изменится не сильно, однако математическое ожидание суммы времён обнаружения проблем – увеличится (за счёт менее эффективного порядка исполнения проверок).

Таким образом, рассмотренный алгоритм в большой степени удовлетворяет требуемым свойствам. [33]

5.16 Взвешенная метрика offline с предрасчётом порядка

В предыдущей секции был рассмотрен алгоритм, который практически полностью удовлетворяет требуемым свойствам. Попробуем решить проблему с неоптимальным порядком запуска зависимостей путем предрасчёта порядка.

Для этого предлагается следующий алгоритм:

1. Выполнить алгоритм "взвешенная метрика offline"
2. Найти наиболее приоритетную проверку d_{next}
3. Поместить d_{next} в конец результата

[33] Все, кроме одного, необходимые эффекты (перечисленные выше) учтены в той степени, которая не позволяет создать конфигурации, которые могли бы вызвать качественно неоптимальные планы исполнения.

4. Удалить из ГЗ d_{next} и зависимости, через которые она вероятно прошла

5. Повторять 1-4 до тех пор пока не закончатся проверки

В результате работы алгоритма, получится последовательность проверок, при следовании по которой, будет учитываться то, что исполненные зависимости исполняются мгновенно.

Время произведения выбора при непосредственном обходе графа не изменилось.

Проблема 1: Время offline (предрасчёт) фазы исполнения алгоритма увеличится до $O(n^2m)$, и для количества проверок равного 10000 будет примерно равняться 20 минут.

Проблема 2: Этот алгоритм всё ещё не позволяет строить маршрут с учетом реально исполненных зависимостей в режиме реального времени. При непосредственном исполнении может оказаться, что исполнение пошло не по вероятным ветвям, и корректировок сделано не будет.

5.17 Взвешенная метрика online

Идея этого алгоритма очевидна, и заключается в отказе от предрасчёта. Алгоритм планирования работающий таким образом не имеет недостатков offline алгоритмов, однако несёт в себе и очевидный недостаток.

В общем случае, при выполнении каждой зависимости, может произойти полное перестроение плана исполнения. Учёт этого без специфических оптимизаций требует запуск всего offline алгоритма после каждой выполненной зависимости. Процесс выбора следующей зависимости к исполнению требует таким образом $O(nm)$ вычислительных операций и памяти.

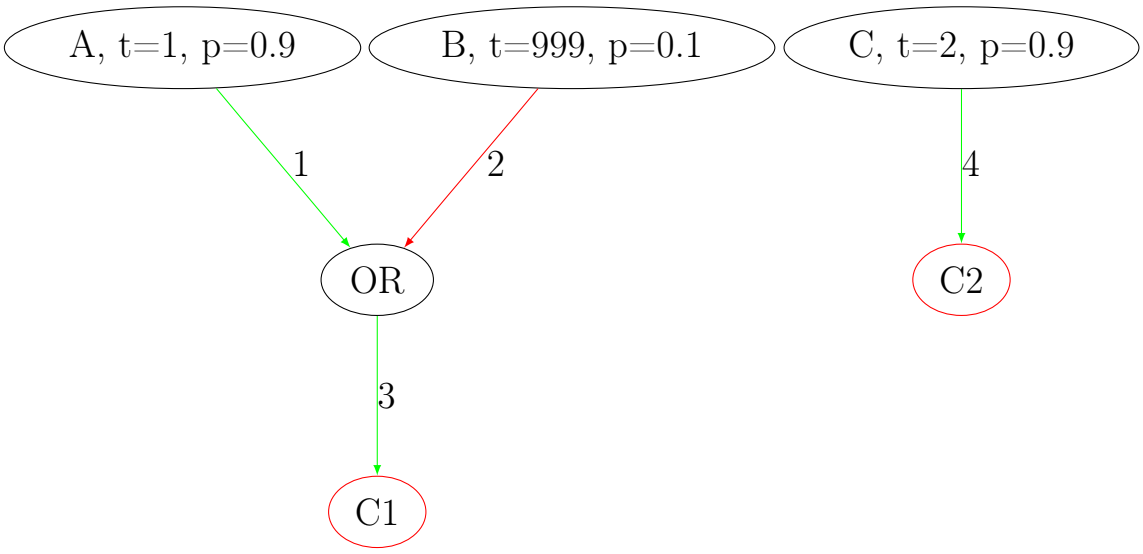

Рис. 17: Необходимость полного перестроения маршрута при необходимости исполнить маловероятную сложную ветвь.

Повышение алгоритмической стоимости, приводит к тому, что время исполнения планировщика по порядку достигает 1 секунды при $n > 1000$.

Этот алгоритм может быть использован до тех пор, пока количество проверок не превышает определенного значения.

5.18 Дополнительное пространство для оптимизаций

В данной секции коротко будет дан обзор дополнительных способов оптимизации пути в ГЗ.

5.18.1 Поиск кластеров проверок на матрице корреляции зависимостей

После расчета вероятного времени исполнения проверок, каждая проверка имеет битовый массив зависимостей, которые вероятно будут исполнены перед запуском проверки. Вектора всех проверок могут быть использованы для построения матрицы корреляции, которая затем будет использоваться для кластеризации (нахождения групп стоящих рядом) проверок и последующего их совместного запуска. [34]

Такой подход так же как и предложенный ранее работает за кубическое время $O(n^3)$ против $O(n^2)$, однако, может быть распараллелен.

[34] Решается проблема, обозначенная во "Взвешенной offline метрике".

5.19 Предрасчёт долгих легковесных зависимостей

Существует ряд случаев, в которых необходимо собрат статистические данные с системы (потребление CPU, сетевая активность, статистика системных вызовов, анализ стеков вызовов, анализ работы дисков...). В таких случаях нерационально прерывать исполнение ГЗ и простаивать в ожидании завершения каждой из этих зависимостей. Поскольку этот класс зависимостей не сильно влияет на систему (не нагружает её, а лишь раз в некоторое время считывает состояние), можно выполнять эти задачи параллельно работе основного алгоритма.

Этого можно добиться очень легко с технической точки зрения, создав отдельных поток для таких зависимостей, и повысив приоритет задач, которые с высокой степенью вероятности заблокируются на такого рода зависимости, передав исполнение другим проверкам.

6 Выводы

Создан инструмент для поиска узких мест производительности и проблем конфигурации в вычислительной системе. В ходе создания инструмента были выполнены следующие основные задачи:

1. Разработан способ описания проверок удобный для пользователя и оптимизации

2. Разработано несколько эмпирический алгоритмов оптимизации графа проверок, предназначенных для оптимизации графов различного размера (в том числе и 10.000 ребер)

3. Написана библиотека на языке Python реализующая предложенный алгоритм

Библиотека созданная в рамках написания работы при должном количестве проверок, тестирующих систему с разных сторон, может быть очень полезным инструментом для администраторов и инженеров-программистов.

Инструмент рассчитан на работу на системах со слабо меняющимися во времени параметрами.

Инструмент полностью кросс-платформенный, отличия проявляются только на уровне проверок.

Таким образом, все поставленные целы достигнуты.

7 Дальнейшая работа

1. Применение подхода для тестового покрытия программ [35]
2. Поднятие централизованной системы сбора статистики о запусках и рассылки предрасчитанных параметров
3. Работа с сообществом, написание большего числа проверок
   a) Анализ стеков исполняющихся бинарных файлов
   b) Анализ стеков интерпретируемых языков/java
   c) Анализ brunch-prediction исполняющихся бинарных файлов
   d) Анализ cache-misses исполняющихся бинарных файлов
   e) Всесторонний анализ статистики системных вызовов, в том числе с помощью eBPF
   f) Работа ввода/вывода
   g) Проверки специфичные для разных популярных приложений (nginx, apache, postgres...)
   h) безопасность ...

[35] Интересно, как эту идею можно применить к написанию автоматических тестов программ. Теоретически, такой подход позволил бы более точно локализовать проблему и избежать повторного тестирования одного функционала в разных тестах, что на данный момент является повсеместной проблемой.