

ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

Отчет о программном проекте

на тему: Программа для исправления сетей Петри с использованием гамаков

Выполнил:

Студент группы БПМИ201

14.05.2022

Дата

Подпись

Е. А. Егоров

И.О.Фамилия

Принял:

Руководитель проекта

Алексей Александрович Мицюк

Имя, Отчество, Фамилия

заведующий кафедрой, доцент, канд. комп. наук

Должность, ученое звание

Базовая кафедра компании JetBrains ФКН НИУ ВШЭ

Место работы (Компания или подразделение НИУ ВШЭ)

Дата проверки 18 мая 2022

отлично (10)

Оценка (по 10-ти бальной шкале)

А.А.Мицюк

А.А.Мицюк

Подпись

Москва 2022

Содержание

1	Реферат	2
2	Основные термины, определения и сокращения	2
3	Введение	3
4	Сравнительный анализ источников и аналогов	5
4.1	Источники	5
4.2	Аналоги	6
4.3	Требования к программе	7
5	Теоретическая часть	7
5.1	Описание метода	7
5.1.a	Выделение «проблемных» вершин в сети	7
5.1.b	Выделение гамаков и их замена	10
5.1.c	Предварительное исправление модели	12
5.2	Нахождение гамаков в сети	16
5.2.a	Задача поиска наименьшего гамака, содержащего выбранное мн-во вершин . . .	16
5.2.b	Задача покрытия мн-ва вершин гамаками с условиями нахождения вершин в одном гамаке	20
6	Реализация	21
7	Оценка результатов	24
7.1.a	Используемые метрики	24
7.1.b	Результаты тестирования	25
7.1.c	Выводы	26
8	Заключение	27
	Список источников	28
	Приложение	28
A	Алгоритмы	29
A.1	Выделение «плохих» пар	29
A.2	Поиск наименьшего гамака, содержащего заданное мн-во вершин	30
B	Примеры применения алгоритмов	32
B.1	Пример выделения «плохих» пар	32
B.2	Простой пример всего алгоритма исправления	33

1 Реферат

Объектом исследования в рамках проекта является возможность применения *гамаков* для исправления моделей процессов, представленных в виде сетей Петри. Конечной целью является реализация алгоритма на языке Python в качестве части библиотеки PM4PY [5]. В ходе работы был предложен новый метод исправления модели, использующий гамаки, а также алгоритм их нахождения в сетях. Относительно формально и подробно сам метод и все промежуточные алгоритмы описаны в теоретической части отчёта (5). Предложенный алгоритм был реализован и протестирован (6). Результаты его работы были оценены с помощью различных релевантных метрик (7): алгоритм, как и ожидалось, оказывается лучше наивного подхода по качеству в случае незначительных точечных изменений сети.

Ключевые слова: process mining, model repair, petri net, pm4py, hammocks

2 Основные термины, определения и сокращения

Модель процесса (process model) — формальное математическое описание некоторого, зачастую естественного, процесса. Будем считать, что для рассматриваемого процесса выделено мн-во *действий* (activities) и *названий действий* (activity labels), входящих в него. Мн-во названий действий в контексте рассматриваемого процесса далее обозначается как \mathcal{A} .

Журнал или **лог событий** (event log) [в простом смысле]: Пусть \mathcal{A} — мн-во названий действий некоторого процесса. Будем называть *следом* (trace) произвольную последовательность действий из \mathcal{A} , а *журналом* (*логом*) *событий* — мультимножество следов.

Сеть Петри (Petri net) — тройка $N = (P, T, F)$, где P (places) — конечное мн-во *позиций*, T (transitions) — конечное мн-во *переходов* такое, что $P \cap T = \emptyset$, и

F (flow relation) $\subseteq (P \times T) \cup (T \times P)$ — мн-во ориентированных рёбер между позициями и переходами.

Маркированная сеть Петри (marked Petri net) — пара (N, M) , где $N = (P, T, F)$ — сеть Петри, а M (marking) $\in \mathbb{B}(P)$ — мультимножество на P , называемое *маркировкой* и зачастую интерпретируемое как мн-во *токенов* в сети.

Размеченная сеть Петри (labeled Petri net) — набор $N = (P, T, F, A, l)$, где (P, T, F) — сеть Петри, $A \subseteq \mathcal{A}$ — подмножество названий действий процесса, и $l : T \mapsto A$ — функция разметки, сопоставляющая переходам названия действия.

Пусть $N = (P, T, F)$ — сеть Петри. Будем рассматривать мн-во $P \cup T$ как мн-во вершин сети, как в ориентированном графе с рёбрами F (это подразумевается далее при рассмотрении сети Петри как орграфа). Для вершины v обозначим мн-во соседних с ней по входящим рёбрам вершин как $\bullet v$, а мн-во соседних по исходящим как $v \bullet$.

Сеть рабочего процесса (workflow net): Пусть $N = (P, T, F, A, l)$ — размеченная сеть Петри, и $\bar{t} \notin P \cup T$ — добавленный переход. Тогда N называется *сетью рабочего процесса*, если

- a) $\exists i \in P$ такая, что $\bullet i = \emptyset$; позиция i называется *начальной* или *истоком*
- b) $\exists o \in P$ такая, что $o \bullet = \emptyset$; позиция o называется *конечной* или *стоком*
- c) Сеть $\bar{N} = (P, T \cup \{\bar{t}\}, F \cup \{(o, \bar{t}), (\bar{t}, i)\}, A \cup \{\tau\}, l \cup \{(\bar{t}, \tau)\})$ сильно связна как оргграф

Граф потока управления (control flow graph) — набор $G = (V, E, v_s, v_t)$, где (V, E) — ориентированный граф с мн-вом вершин V и мн-вом рёбер E , где в каждую вершину есть путь из $v_s \in V$, и из каждой вершины есть путь в $v_t \in V$.

Гамак (hammock): пусть (V, E, v_s, v_t) — граф потока управления. Тогда его подграф $H = (V', E', s, t)$ называется *гамаком* с истоком $s \in V'$ и стоком $t \in V'$, если

$$\forall e = (u, v) \in E, u \notin V', v \in V' \Rightarrow v = s$$

$$\forall e = (u, v) \in E, u \in V', v \notin V' \Rightarrow u = t$$

Неформально говоря, все входящие извне в гамак рёбра входят в исток, а все исходящие из него рёбра выходят из стока.

Заметим, что сеть рабочего процесса $N = (P, T, F, A, l)$ с начальной позицией i и конечной позицией o , если рассматривать её как оргграф, является графом потока управления $G = (P \cup T, F, i, o)$. Таким образом, приведённое определение гамака применимо и в отношении сетей рабочего процесса (какими и будут являться в большинстве случаев рассматриваемые сети Петри).

3 Введение

В последние годы наблюдается стремительное развитие науки о данных (data science). Одним из наиболее распространённых способов её приложения является изучение и анализ бизнес-процессов. Совмещая в себе использование данных от data science и построение классических моделей от business process management (BPM), относительно недавно, в качестве самостоятельной области зародилась теория process mining (в некоторых источниках название адаптируется как *процессная аналитика*). Process mining предлагает ряд методов и подходов для анализа и улучшения бизнес-процессов путём построения их модели и её сравнения с реальным журналом событий.

Само понятие process mining было введено голландским учёным в области компьютерных наук — проф. Вилом ван дер Аалстом. Основой данной теории выступает его книга [1], используемая в качестве главного источника в ходе работы над данным проектом (в том числе, большинство приведённых определений были взяты из неё).

Основными направлениями process mining являются:

- *Обнаружение процесса* (process discovery) — построение модели процесса на основании его журнала событий
- *Проверка соответствия* (conformance checking) — сравнение существующей модели и журнала событий с целью выявления несоответствий между ними
- *Улучшение* (enhancement) — изменение существующей модели на основании журнала событий, разделяемое на *исправление* (repair) — когда изменения нацелены на её большее соответствие реальному процессу, и *расширение* (extension) — когда изменения нацелены на отражение новой информации моделью,

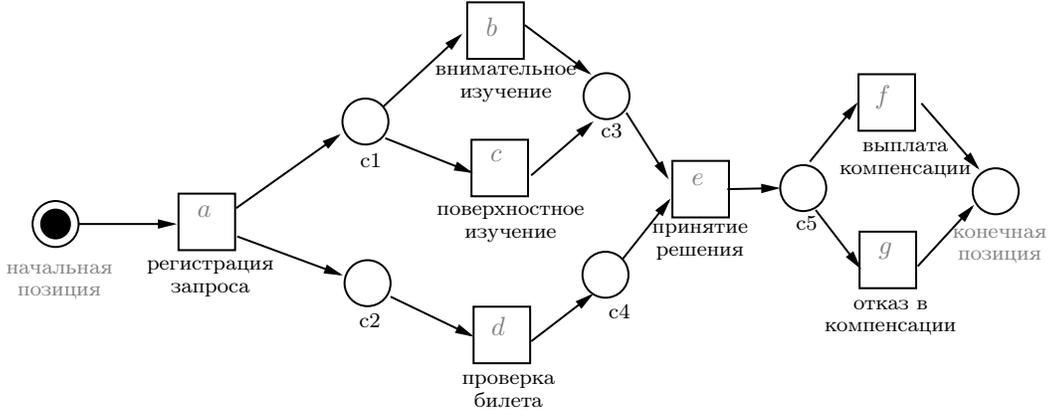


Рис. 1: Пример модели процесса оформления компенсации за авиабилет в виде сети Петри.

например, о времени, затрачиваемом на действия

Модель процесса может быть представлена множеством различных способов. Одним из самых распространённых и, наверное, естественных из них является представление в виде сети Петри.

На рис. 1 представлен пример модели процесса в виде размеченной сети Петри с единственным токеном в начальной позиции, квадраты соответствуют переходам, круги – позициям. Кратко и не совсем формально опишем симуляцию процесса в ней:

Опр. (Правило активации перехода). Пусть (N, M) – маркированная сеть Петри, $N = (P, T, F)$. Будем называть переход $t \in T$ *включённым*, если $\bullet t \subseteq M$.

Результатом *активации* включённого перехода t будем считать маркированную сеть Петри (N, M') , где $M' = (M \setminus (\bullet t)) \cup (t\bullet)$

Токены $\bullet t$ будем называть *поглощёнными* (consumed), а $t\bullet$ – *произведёнными* (produced).

Последовательность переходов $\sigma = (t_1, \dots, t_k), t_i \in T$ будем называть *последовательностью активаций*, если $\forall i : t_i$ включён в (N, M_{i-1}) , где (N, M_{i-1}) – результат последовательной активации переходов t_1, \dots, t_{i-1} в (N, M) , $M_0 = M$.

Например, в приведённой на рис. 1 сети включённым переходом является только (регистрация запроса). После его активации мн-во токенов станет равно $M' = \{c_1, c_2\}$, а включёнными переходами станут (внимательное изучение), (поверхностное изучение) и (проверка билета).

Опр. (Воспроизводимость и соответствие следа). Будем называть след $\sigma = \langle a_1, \dots, a_k \rangle$ из журнала событий *воспроизводимым* для размеченной маркированной сети Петри $(N, M), N = (P, T, F, A, l)$, если $\exists (t_1, \dots, t_k)$ – посл-ть активаций в сети такая, что $\forall i : l(t_i) = a_i \in A$.

Будем считать, что след *соответствует* модели, если дополнительно полученная в результате последовательной активации переходов t_1, \dots, t_k маркировка M' равна M_f для некоторой заданной финальной маркировки M_f .

В случае отсутствия неоднозначности будем использовать одинаковые обозначения для действий и переходов.

Таким образом, *соответствующими* модели на рис. 1 следами при финальной маркировке {конечная позиция} являются, например, $\langle a, b, d, e, f \rangle$ и $\langle a, d, c, e, g \rangle$. В то время как не соответствуют ей следы $\langle a, b, d, e \rangle$ (полученная маркировка не будет совпадать с финальной) и $\langle a, b, c, d, e, f \rangle$ (не является воспроизводимым).

Для заданного следа σ нахождение соответствующей посл-ти активаций в сети (t_1, \dots, t_k) называют его *воспроизведением с помощью токенов* (token-based replay).

С помощью методов проверки соответствия можно находить «проблемные» переходы в построенной модели по некоторым критериям. Возникает интуитивное предположение о том, что, используя эту информацию, можно выделять относительно независимые и изолированные структурные единицы сети (в некотором смысле, подсети), а затем перестраивать их с помощью методов обнаружения процесса на основании соответствующих частей записей в журнале событий. Подобный подход встречается в статьях (например, в [2], откуда было взято определение гамака), описывающих структурирование и рефакторинг программного кода путём выделения гамаков при его интерпретации в виде графа потока управления. Предполагается, что можно аналогично выделять гамаки и в случае исправления моделей, представленных в виде сети Петри.

4 Сравнительный анализ источников и аналогов

4.1 Источники

В качестве основного источника теории по process mining была взята книга [1]. В ней представлены различные методы *проверки соответствия*: воспроизведение следов с помощью токенов в сети Петри (token-based replay); метод сравнения моделей, журналов событий как объектов одного типа после построения определённых таблиц по ним (footprints); нахождение наиболее похожих на заданный воспроизводимых следов в модели (построение alignments). Было решено для реализуемого алгоритма использовать именно метод построения alignments (детально описанный в [1, Раздел 8.3]). Кратко и неформально опишем его:

Суть метода заключается в поиске для заданного следа $\sigma = \langle a_1, \dots, a_k \rangle$ «оптимального» относительно него и модели следа, воспроизводимого рассматриваемой моделью. Пусть след $\sigma = \langle a, d, b, e, h \rangle$ является воспроизводимым в модели. Тогда, что вполне естественно, единственным оптимальным соответствием (alignment) является:

след	a	d	b	e	h
путь в модели	a	d	b	e	h

, где верхняя строка соответствует следу σ , а нижняя – оптимальному воспроизводимому следу (на это можно смотреть, как на посл-ть активаций переходов в сети, задающей модель).

Для случая, когда след не является воспроизводимым, вводится символ "»" , означающий пропуск. Например, в следующем alignment для того же следа (но для некоторой другой модели) в качестве оптимального был найден след $\langle a, b, d, e, h \rangle$:

след	a	»	d	b	e	h
путь в модели	a	b	d	»	e	h

1-ая пара (a, a) означает *синхронный шаг* (sync move) – активацию перехода в сети, соответствующего действию в следе; далее пара (\gg, b) означает шаг *только в модели* (model move), т.е. активации перехода b в модели не соответствует действие в следе; (d, d) – снова синхронный шаг; (b, \gg) означает шаг *только в логге* (следе) (log move), т.е. на соответствующее действие в следе в модели не было активаций переходов; далее вновь следуют синхронные шаги (e, e) и (h, h) .

Для сравнения alignments по «оптимальности» вводится функция стоимости, которая зависит от длины alignment, кол-ва ходов только в логге/только в модели (причём веса у различных типов отклонений могут быть разными). Таким образом, оптимальных alignments может быть несколько.

По построенным alignments для каждого следа в журнале событий можно считать различные характеристики, измеряющие отклонение модели от журнала.

Этот метод был выбран, потому что он позволяет выделять последовательности переходов, которые выполнялись в соответствии со следом – подряд идущие синхронные шаги, а также «проблемные» переходы, например, те, которые встречаются в шагах только в модели/логге. Разбиение на «плохие» и «хорошие» отрезки в alignments интуитивно соответствует локализации не соответствующих журналу участков сети, и его предлагается использовать для выделения гамаков.

Также в РМ4РУ уже реализованы функции для нахождения оптимальных alignments, поэтому их можно легко использовать в реализации алгоритма.

Была изучена статья [3], в которой описан метод исправления сети. Этот метод также использует alignments, некоторые идеи в нём похожи на предлагаемые далее, однако упор в нём делается на анализ и исправление шагов только в логге из alignments в отличие от предлагаемого алгоритма. Метод из статьи был использован в качестве основы предварительного исправления сети (5.1.с).

Также в этой и других статьях описаны способы оценивания результатов алгоритма исправления, которые будут использованы для предлагаемого метода (7).

Для задач process mining и работы с сетями используется фреймворк ProM [4] (open source, плагины к нему написаны на Java). Однако для реализации алгоритма была выбрана Python-библиотека РМ4РУ из-за её интенсивного развития и относительной простоты возможной реализации алгоритма, использующего её.

4.2 Аналоги

В РМ4РУ на момент работы над проектом не реализованы алгоритмы исправления сетей. Для фреймворка ProM [4] существуют пакеты, например, Uma – с реализацией одного из алгоритмов из статьи [3], и плагины с реализацией алгоритмов исправления моделей. Функциональность имеющихся методов представляет собой (что ожидаемо) построение исправленной сети по исходной сети и логгу с некоторой параметризацией.

В ряде статей, в которых описаны алгоритмы исправления сетей, при оценке результатов описываемый алгоритм сравнивается с наивным подходом: *обнаружение* полностью новой сети по логгу, и алгоритм оказывается лучше (что вполне естественно). Этого же и будем ожидать от предлагаемого алгоритма исправления.

4.3 Требования к программе

Функциональные

Результатом проекта должна стать реализация описанного во введении алгоритма в качестве части python-библиотеки PM4PY [5], исходный код которой размещён в GitHub-репозитории¹. Основной должна быть функция, вызываемая от сети Петри, задающей модель процесса, и журнала событий, представленных во внутреннем формате PM4PY, с возможными параметрами. Результатом работы функции должна быть новая сеть Петри, исправленная на основании предоставленного журнала.

Реализация должна следовать общему стилю и уже существующей логике библиотеки. В коде должны использоваться имеющиеся внутренние функции работы с сетями/журналами событий вместо самописных везде, где это возможно. Возможно внесение изменений в функции библиотеки с целью необходимого расширения их функциональности.

Функция должна проверять все необходимые для применения алгоритма инварианты и сообщать об их невыполнении или возникших из-за неправильных аргументов ошибках. Она должна корректно работать при передаче любых аргументах, удовлетворяющим требованиям алгоритма. Для проверки этого должны быть реализованы тесты, проверяющие все части реализации как на корректность логики, например, при работе с крайними случаями входа, так и на корректность работы с неправильными данными.

Реальное время работы алгоритма на входных данных разумного размера не должно превышать времени работы сравнимых операций работы с моделями и журналами событий, реализованных в библиотеке.

Алгоритм должен показывать результат лучше, чем при использовании *обнаружения* полностью новой сети по журналу событий.

Нефункциональные

Реализованный алгоритм должен иметь строгое теоретическое обоснование корректности.

5 Теоретическая часть

5.1 Описание метода

Далее предполагается, что рассматриваемая сеть Петри (исправление которой – цель алгоритма) является маркированной сетью рабочего процесса со стартовой маркировкой, содержащей только начальную вершину, и финальной маркировкой, содержащей только конечную вершину.

5.1.a Выделение «проблемных» вершин в сети

Хочется выделять пары вершин в сети, которые будут соответствовать началу и концу «проблемных» участков, а затем искать гамаки таким образом, чтобы каждая такая пара содержалась в одном гамаке. Для этого можно использовать некоторые алгоритмы проверки соответствия лога сети, а именно, будем использовать alignments.

¹GitHub репозиторий библиотеки PM4PY – <https://github.com/pm4py>

Наивный подход

Выделим в alignment все максимальные по длине последовательности подряд идущих синхронных шагов. Выбросим из рассмотрения те последовательности, которые состоят из одного шага, если только этот шаг не первый и не последний в порядке alignment (предполагаем, что первый и последний шаги – синхронные). Будем считать оставшиеся последовательности «хорошими» участками, их правые концы – началом, а левые – концом «плохих» участков в порядке alignment (пример выделения пар на Рис. 2).

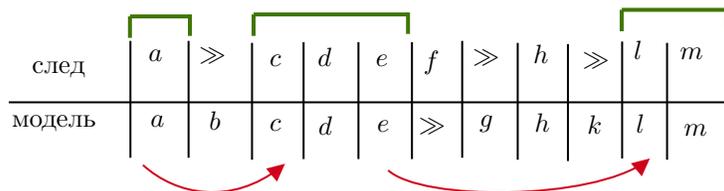
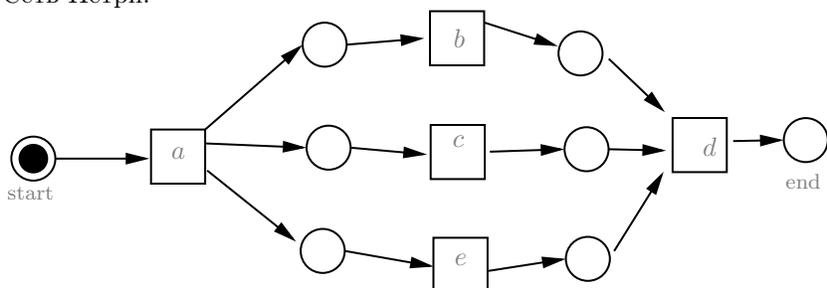


Рис. 2: Пример наивного использования alignment для выделения пар. Зелёным отмечены «хорошие» участки, красные рёбра соответствуют парам выделенных вершин.

Однако данный подход может приводить к выделению большого числа пар, которые на самом деле не соответствуют «проблемным» участкам сети, даже при появлении малейшей параллельности. Что ожидаемо, ведь при выделении пар явно используется линейный порядок alignment без привязки к структуре сети, а при возникающей параллельности действия из разных веток могут быть упорядочены в alignment произвольным образом. Рассмотрим пример:

Сеть Петри:



В следах $\sigma_1 = \langle a, b, e, d \rangle$ и $\sigma_2 = \langle a, e, b, d \rangle$ фактически одно и то же несоответствие с моделью – отсутствует действие c . Возможные оптимальные alignments:

след	a	b	>>	e	d	след	a	>>	b	e	d
модель	a	b	c	e	d	модель	a	c	b	e	d

след	a	e	>>	b	d	след	a	>>	e	b	d
модель	a	e	c	b	d	модель	a	c	e	b	d

И описанный алгоритм может выделить 4 различных пары вершин: (b, e) , (a, b) , (e, b) , (a, e) . При большем размере сети кол-во возможных пар увеличится, и какие-то из них могут и не находиться в действительно «проблемных» участках сети, что и было замечено при применении алгоритма на примерах сетей.

Улучшенный подход

Для улучшения наивного подхода предлагается использовать воспроизведение следов с помощью токенов для alignment с целью использования структуры сети.

Для каждого токена будем поддерживать два мн-ва переходов: *зелёные* и *красные*. Изначально для стартового токена оба мн-ва положим пустыми. Будем последовательно идти по шагам alignment. Шаги только в логе будем просто пропускать (об их исправлении – в 5.1.с). Пусть активируемый в модели переход, соответствующий шагу alignment – T_a .

Для токена t обозначим за $G(t), R(t)$ – мн-ва его *зелёных* и *красных* переходов соответственно. Пусть $C = \{t_1, \dots, t_m\}$ – мн-во токенов, *поглощённых* при активации T_a , а $P = \{t_1, \dots, t_k\}$ – мн-во *произведённых* токенов. Тогда:

1. Если шаг – только в модели, и T_a – *скрытый* переход, т.е. не имеющий метки (hidden transition):

$$\forall t_p \in P: \text{положим: } \begin{cases} G(t_p) = \bigcup_{t_c \in C} G(t_c) \\ R(t_p) = \bigcup_{t_c \in C} R(t_c) \end{cases}, \text{ как бы сохранив состояние}$$

2. Если шаг – только в модели, и T_a – не *скрытый* переход:

$$\forall t_p \in P: \text{положим: } \begin{cases} G(t_p) = \emptyset \\ R(t_p) = \bigcup_{t_c \in C} G(t_c) \cup R(t_c) \end{cases}$$

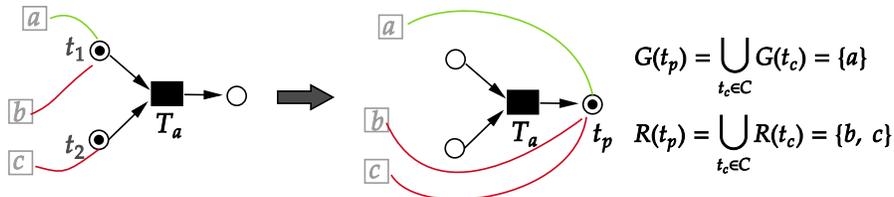
3. Если шаг – синхронный:

Отметим пары $\left\{ (v, T_a) \mid \forall v \in \bigcup_{t_c \in C} R(t_c) \right\}$ как «плохие».

$$\forall t_p \in P: \text{положим: } \begin{cases} G(t_p) = \{T_a\} \\ R(t_p) = \emptyset \end{cases}$$

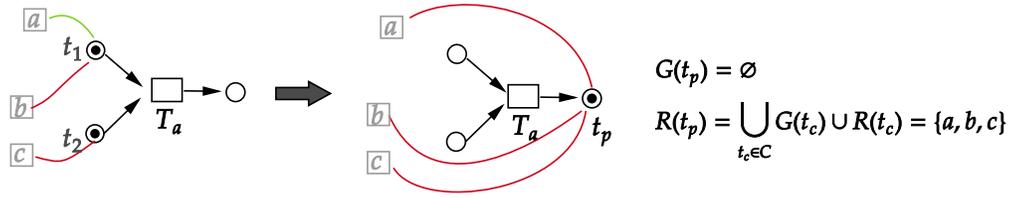
Рассмотрим все эти случаи на примере одного шага, где $C = \{t_1, t_2\}$, $\begin{cases} G(t_1) = \{a\} \\ R(t_1) = \{b\} \end{cases}$, $\begin{cases} G(t_2) = \emptyset \\ R(t_2) = \{c\} \end{cases}$ для некоторого $t_p \in P$

1. Если шаг – только в модели, и T_a – *скрытый* переход, т.е. не имеющий метки (hidden transition):

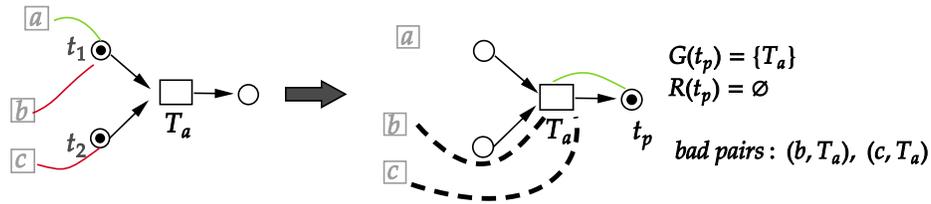


Введённые мн-ва переходов для токенов изображены как переходы, которые соединены с токенами линиями соответствующего цвета

2. Если шаг – только в модели, и T_a – не *скрытый* переход:



3. Если шаг – синхронный:



Чёрные пунктирные линии соответствуют парам, выделенным в качестве «плохих»

Осталось рассмотреть возможные особые случаи, связанные с первым и последним шагом в alignment.

Для случая, если первый шаг alignment не является синхронным:

отметим для стартового токена стартовую позицию сети как *зелёную*. Тогда *зелёное* и *красное* мн-ва перестанут быть мн-вами только из переходов, и в ходе алгоритма станет возможным выделение плохой пары со стартовой позицией, а не только между переходами, что, однако, останется корректным для дальнейших шагов общего алгоритма.

Для случая, если маркировка, полученная после воспроизведения всего следа (*итоговая*), не совпадает с финальной маркировкой сети (токеном в стоке сети)¹:

для всех *красных* вершин токенов *итоговой* маркировки – v отметим пару (v, end) как «плохую» (где end – сток сети).

Псевдокод описанного алгоритма приведён в Приложении А.1.

Пример выделения плохих пар по одному alignment приведён в Приложении В.1.

Сложность описанного алгоритма в худшем случае составит $O(|\sigma| \cdot (|N| + |E|))$, где $|\sigma|$ – кол-во ходов в alignment, а N, E – мн-ва вершин и рёбер сети соответственно. Однако сложность получения alignment по следу из лога точно не меньше.

5.1.b Выделение гамаков и их замена

¹ Вообще, такой случай не должен быть возможен при корректном нахождении alignments

Выделим «плохие» пары для каждого alignment, как описано в предыдущем пункте, и объединим их. Можно учитывать, сколько раз каждая такая пара была выделена, и рассматривать только те из них, которые были выделены наибольшее число раз.

Найдём минимальное по кол-ву включённых вершин мн-во непересекающихся гамаков в сети, такое, что гамаки покрывают все вершины из «плохих» пар, и для каждой такой пары обе вершины лежат в одном гамаке. Пример изображён на Рис. 3.

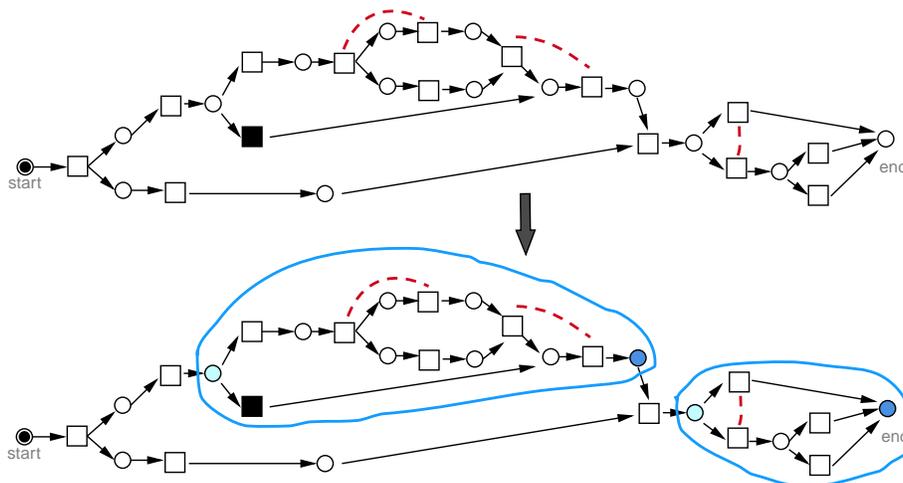


Рис. 3: Пример выделения гамаков по «плохим» парам в предположении, что истоками/стоками гамаков могут быть только позиции. «Плохие» пары отмечены красными пунктирами, выделенные гамаки обведены, их истоки и стоки выделены голубым и синим цветом соответственно.

Здесь возникает возможность выбора: допускать только такие гамаки, чьими истоком/стоком является позиция, или же разрешать им являться и переходами. В случае допущения переходов найденные гамаки могут содержать меньшее число вершин. Эту возможность выбора можно легко учесть в параметрах реализации.

Используемый для этого алгоритм подробно описан в разделе 5.2, сложность поиска гамаков с его использованием составит $O(p \cdot (|N| + |E|))$, где p – кол-во «плохих» пар, а N, E – мн-ва вершин и рёбер сети соответственно.

Далее для каждого найденного гамака рассмотрим мн-во меток переходов внутри него (действий внутри гамака) и выделим из каждого следа журнала событий подпоследовательность действий из этого мн-ва (подслед). Получим как бы часть лога, соответствующую гамаку. Затем по этой части лога с помощью некоторого алгоритма обнаружения процесса¹ построим сеть рабочего процесса (*подпроцесса* исходной сети). Заменяем гамак на построенную сеть.

Небольшой тонкостью является замена гамаков при допущении переходов в качестве их истоков/стоков. Замена в случае истока–перехода изображена на Рис. 4, случай стока–перехода симметричен.

Здесь важно, чтобы в исправляемой сети у всех переходов были различные метки. Иначе в ходе описанной замены гамаков может получиться некорректный результат (можем взять часть лога, которая не соответству-

¹В текущей реализации используется inductive miner, реализованный в PM4PY

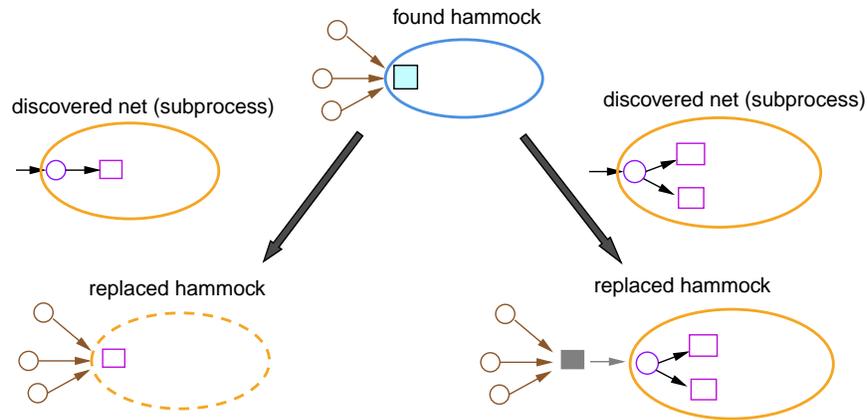


Рис. 4: Замена гамака на сеть подпроцесса, если источник гамака – переход. Рассмотрены два возможных случая: когда у стартовой позиции подпроцесса единственное исходящее ребро, и когда их несколько.

ет рассматриваемому гамаку) в том смысле, что ранее невозпроизводимый след из лога после исправления может остаться невозпроизводимым, и, более того, ранее воспроизводимые могут перестать быть таковыми (даже в случае отсутствия игнорируемых шагов только в логе). В случае наличия таких переходов можно наложить дополнительное условие для выделенных гамаков – чтобы для каждой метки внутри гамака все переходы с такой меткой содержались в нём. Конечно, можно наложить ограничение на исправляемую сеть – чтобы в ней метки переходов не повторялись, однако в предлагаемом далее (5.1.с) методе предварительного исправления это ограничение не будет соблюдаться.

5.1.с Предварительное исправление модели

Описанный метод уже можно использовать для исправления сетей. Однако в нём совсем игнорируются шаги только в логе из alignments, что, конечно, не позволит добиться 100%-процентного соответствия следов из лога исправленной сети (100%-го значения fitness) в случае их наличия.

Один из способов решения этого – изменить функцию стоимости при вычислении alignments так, чтобы минимизировать кол-во шагов только в логе. Другим способом, который и будет использоваться, является дополнительное исправление шагов только в логе перед применением общего алгоритма.

Далее будет описан используемый метод исправления шагов только в логе, который является достаточно простым и подходит для дальнейшего выделения гамаков.

Метод исправления

В статье [3] предложен алгоритм исправления сети. В нём для этого также используются alignments. Для шагов только в модели применяется достаточно простой метод исправления, а для шагов только в логе из следов выделяются соответствующие участки, по которым строятся так называемые *подпроцессы*, встраиваемые в соответствующие места сети.

Опишем неформально упрощённую версию этого алгоритма (подробно и строго полный алгоритм описан в статье).

Опр. (Локация шага только в логе (location)).

Пусть рассматривается некоторый alignment. Тогда для шага только в логе из него *локация* – мн-во позиций, в которых находятся токены после воспроизведения действий в модели из предшествующих шагов alignment в рассматриваемой сети.

Заметим, что это определение можно использовать для любого шага из alignment.

Проще всего пояснить используемые определения и сам алгоритм на примерах, которые далее будут взяты из той же статьи.

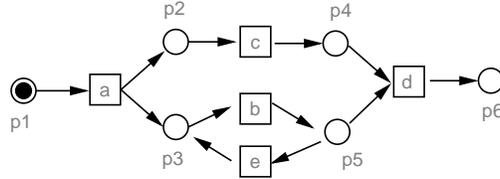


Рис. 5: Пример сети

Для сети на Рис. 5 рассмотрим два alignment (для каждого шага указана его *локация*):

след (лог)	a	c	f	c	»	e	»	d		след (лог)	a	b	c	c	f	e	»	d	
модель	a	c	»	»	b	e	b	d		модель	a	b	c	»	»	e	b	d	
локация		[p2, p3]	[p4, p3]	[p4, p3]	[p4, p3]	[p4, p5]	[p4, p3]	[p4, p5]		локация		[p2, p3]	[p2, p5]	[p4, p5]	[p4, p5]	[p4, p5]	[p4, p3]	[p4, p5]	

Тогда в 1-ом alignment у каждого из шагов (f, »), (c, ») *локация* равна [p4, p3], а во 2-ом у шагов (c, »), (f, ») – [p4, p5].

В статье рассматриваются посл-ти шагов только в логе (например, в 1-ом alignment из примера: [(f, »), (c, »)]), определяемые как *subtrace*, но мы не будем этого делать и остановимся на рассмотрении отдельных шагов.

Для шага только в логе $m = (a_i, \gg)$ обозначим за $loc(m)$ мн-во его *локаций* по всем alignment из рассматриваемых alignments. В примере выше (для alignments состоящего из двух alignment):

$loc((f, \gg)) = loc((c, \gg)) = \{[p4, p3], [p4, p5]\}$. Тогда интуитивным способом исправления является следующий:

пусть $m = (a, \gg)$ – некоторый шаг только в логе, тогда $\forall l \in loc(m)$ добавим в сеть переход T_a с меткой a , и проведём для всех позиций из l рёбра в T_a и обратно

Пример такого наивного исправления изображён на Рис. 6: слева – для 1-го alignment выше, справа – для 2-го.

Естественной идеей в данном примере является добавление двусторонних рёбер в новые переходы с метками f и c только из $p4$, потому что эта позиция лежит в каждой *локации* обоих шагов. Будем использовать эту идею в алгоритме:

Для каждого шага только в логе из рассматриваемых alignments: $m = (a, \gg)$ разобьём мн-во

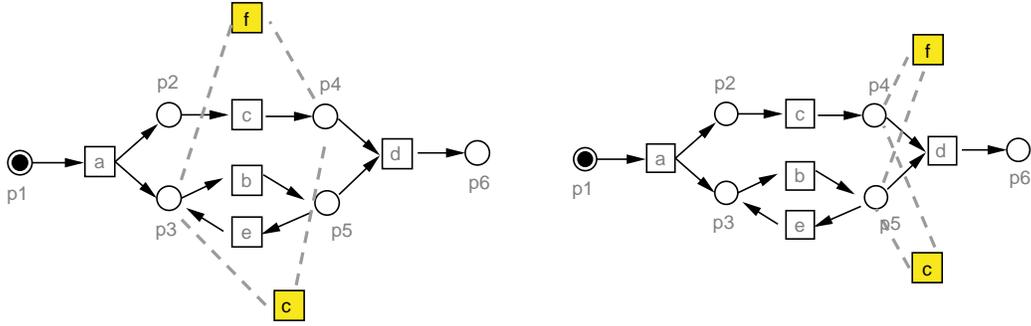


Рис. 6: Пример наивного исправления сети. Пунктирные линии между вершинами соответствуют проведённым рёбрам в обе стороны.

локаций шага $loc(m) = \{l_1, \dots, l_q\}$ на наборы $\{\rho_1, \dots, \rho_t\} : \rho_i = \{l_{j_1^i}, \dots, l_{j_{c_i}^i}\}$, $\bigsqcup_{i=1}^t \rho_i = loc(m)$ таким

образом, что в любом наборе пересечение локаций не пусто: $\forall i : \bigcap_{l \in \rho_i} l \neq \emptyset$.

Заметим, что подходящим разбиением всегда является $\{\{l_1\}, \{l_2\}, \dots, \{l_q\}\}$

Далее для каждого набора ρ_i добавим переход T_a с меткой a и проведём между ним и каждой позицией из пересечения локаций набора $\bigcap_{l \in \rho_i} l$ рёбра в обе стороны.

Остаётся только выбрать какое-то подходящее разбиение $loc(m)$. Разумным кажется минимизация его мощности. Будем использовать следующий простой интуитивный алгоритм:

- Находим позицию, которая входит в наибольшее кол-во локаций из $loc(m)$
- Выделяем все локации, в которые входит эта позиция, в отдельный набор и убираем их из рассмотрения
- Повторяем, пока остаются невыделенные локации

В примере выше результатом применения этого алгоритма будет, как и было предложено, сеть на Рис. 7.

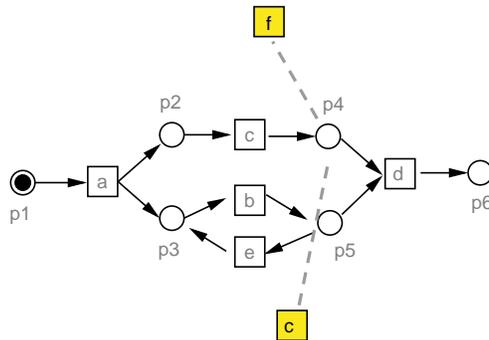


Рис. 7: Пример улучшенного исправления сети.

Дополнительно отметим, что в случае, если новые переходы в ходе исправления были добавлены к стартовой или конечной вершине исходной сети, то, чтобы сохранить её свойство сети рабочего процесса, потребуется

добавить скрытый переход и новую позицию вместо стартовой/конечной (пример – на Рис. 8).

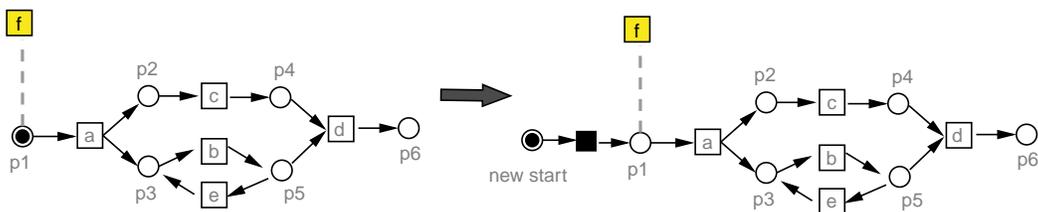


Рис. 8: Пример изменения модели во время предварительного исправления в случае добавления нового перехода к стартовой позиции.

Использование в рамках общего алгоритма

Описанный метод исправления шагов только в логе гарантирует воспроизводимость соответствующих частей следов в исправленной сети, однако её структура может стать «плохой» из-за достаточно наивного добавления одиночных переходов (они могут попасть не на своё место). Хотелось бы, чтобы выделяемые далее гамаки содержали переходы, добавленные в ходе предварительного исправления, чтобы учесть структуру модели и целые участки следов лога. Тогда ожидается, что в результате замены гамаков добавленные переходы «встанут на свои места». Остаётся только некоторым образом «сообщить» алгоритму выделения гамаков о том, что нужно включить эти переходы в гамаки.

Помимо этого, описанный метод исправления шагов только в логе, также как и общий алгоритм, использует alignments. После предварительного исправления они перестают быть корректными, и их следует вычислить заново, чего, т.к. вычисление alignments является самой вычислительно сложной частью общего алгоритма, хотелось бы как-то избежать.

Решением обеих этих задач является изменение alignments в ходе предварительного исправления. Описанный метод предварительного исправления как бы заменяет шаги только в логе из alignments на синхронные в том смысле, что, например, следующие alignments:

след (лог)	a	c	f	c	»	e	»	d
модель	a	c	f	c	b	e	b	d

след (лог)	a	b	c	c	f	e	»	d
модель	a	b	c	c	f	e	b	d

становятся корректными для исправленной сети на Рис. 7. Заметим, что эти alignments уже могут не быть оптимальными для соответствующего следа из лога в исправленной сети, однако это не повлияет на корректность дальнейшего алгоритма. Тогда можем соответствующим образом заменить шаги только в логе на синхронные шаги и уже по изменённым таким образом alignments выделять гамаки в предварительно исправленной сети. При добавлении переходов к стартовой/конечной вершине (как на Рис. 8) добавим вставленный скрытый переход как шаг в модели в начало/конец alignment.

Однако это не «заставит» алгоритм выделять гамаки, содержащие добавленные переходы. Для этого можем изменять шаги только в логе не на синхронные, а на шаги только в модели. Тогда, ввиду алгоритма

выделения плохих пар, соответствующие переходы будут покрыты гамаками. Для рассматриваемого примера:

след (лог)	a	c	»	»	»	e	»	d	след (лог)	a	b	c	»	»	e	»	d
модель	a	c	f	c	b	e	b	d	модель	a	b	c	c	f	e	b	d

5.2 Нахождение гамаков в сети

5.2.a Задача поиска наименьшего гамака, содержащего выбранное мн-во вершин

В статье [2], где приводится используемое определение гамака, описан алгоритм его поиска лишь в контексте рассматриваемой задачи (связанной с улучшением структуры программного кода), и он не может быть использован в общем случае для произвольного графа потока управления.

Был предложен алгоритм нахождения наименьшего гамака, покрывающего заданное мн-во вершин в графе, который описан далее.

Пусть $G = (N, E, n_s, n_t)$ – некоторый граф потока управления.

Тогда, т.к. всегда найдётся хотя бы один покрывающий выбранное мн-во гамак $H = (N, E, n_s, n_t)$, всегда найдётся такой наименьший по включению.

Введём ряд обозначений:

$N_c \subseteq N$ – выбранное мн-во вершин, которые должны накрываться гамаком

$P_t(V), V \subseteq N$ – мн-во всех вершинно-простых путей из вершин V в n_t

$P_s(V), V \subseteq N$ – мн-во всех вершинно-простых путей из вершин V в n_s в транспонированном графе G^T

Для некоторого мн-ва множеств X обозначим $I(X) = \bigcap_{x \in X} x$

УТВ 1. Пусть $V \subseteq N$ – некоторое подмн-во вершин, $t \in N$ – выбранная вершина, P – мн-во вершинно-простых путей из вершин V в вершину t . Тогда вершины $I(P)$ можно строго упорядочить так, чтобы они входили в каждый путь из P в определённом порядке.

□

Заметим, что мн-во $I(P)$ не пусто, т.к. в него, конечно, входит t . Предположим, что утверждение неверно. Тогда найдутся две вершины $v_1 \neq v_2 \in I(P)$ такие, что $\exists p_1 \neq p_2 \in P : p_1 = (v_i, \dots, v_1, \dots, v_2, \dots, t), v_i \in V, p_2 = (v_j, \dots, v_2, \dots, v_1, \dots, t), v_j \in V$.

Тогда можно выбрать некоторый префикс p_1 не более (v_i, \dots, v_1) и суффикс p_2 не более (v_1, \dots, t) такие, что, совместив их, получим вершинно-простой путь $p' = (v_i, \dots, t) \in P, v_2 \notin p'$, что противоречит тому, что $v_2 \in I(P)$.

■

Тогда и вершины из $I(P_s(N_c))$ и $I(P_t(N_c))$ можно аналогично упорядочить. Естественным «кандидатом» на исток гамака s является 1-ая вершина в таком порядке из $I(P_s(N_c))$, а «кандидатом» на сток t – из $I(P_t(N_c))$ (отметим, что они они могут ими и не являться).

Заметим, что в условиях утв. 1 $|I(P) \cap V| \leq 1$, потому что иначе в пересечении есть две вершины $v_1 <_o v_2$ (где $<_o$ – описанное отношение порядка входа в пути на вершинах из $I(P)$), то $v_1 \notin I(P)$, т.к. есть вершинно-простой путь из $v_2 \in V$ в t , не содержащий v_1 .

Алгоритм поиска наименьшего гамака, содержащего N_c

Положим сначала $V = N_c$ и будем далее расширять это мн-во, пока оно не станет гамаком.

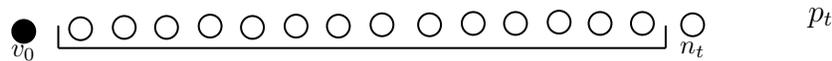
Шаг алгоритма:

Пусть s', t' – найденные упомянутые кандидаты на исток и сток гамака, содержащего V , т.е. $s' = I(P_s(V))$, $t' = I(P_t(V))$. При добавлении новых вершин в V s' и t' могут измениться только на эти новые вершины (а могут и остаться прежними), т.к. все вершины $V \setminus \{s', t'\}$ уже включены в искомый наименьший гамак и не могут являться в нём ни истоком, ни стоком (потому что эти вершины уже не попадут в $I(P_*(V))$, а исток и сток точно будут там лежать). Будем далее называть такие вершины *промежуточными* в гамаке. Включим всех соседей этих вершин (по входящим и исходящим рёбрам) в V , они точно попадут в гамак. Также включим всех соседей s' по исходящим рёбрам ($s' \bullet$) и соседей t' по входящим ($\bullet t'$). Если мн-во добавленных вершин не пусто, то повторим шаг. Иначе мн-во V является искомым гамаком с истоком $s = s'$, и стоком $t = t'$.

Остаётся лишь научиться искать s' и t' .

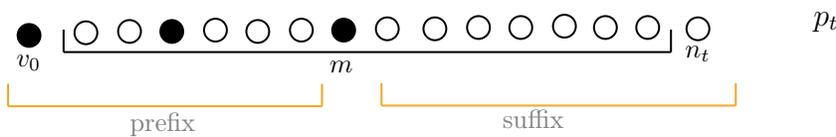
Опишем поиск t' , поиск s' симметричен:

Рассмотрим *произвольный* путь из $P_t(V) - p_t$. Введём мн-во *отмеченных* вершин, в которое изначально будет входить лишь первая вершина пути $p_t - v_0 \in V$.



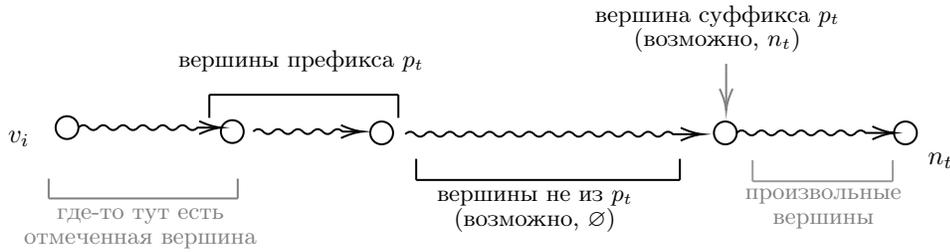
Запустим обходы из остальных вершин V . При входе в некоторую вершину p_t (а этот вход произойдёт для любого пути) будем *отмечать* эту вершину и не будем продолжать из неё обход.

Пусть m – последняя отмеченная вершина p_t в порядке обхода пути. Назовём *префиксом* пути p_t все вершины строго до неё, а *суффиксом* – строго после (оба могут быть пусты). Заметим, что никакая вершина префикса не может лежать в $I(P_t(V))$. Будем в ходе алгоритма поддерживать данный инвариант.



Если все пути из $P_t(V)$ проходят через вершину m , то она лежит в $I(P_t(V))$ и, согласно инварианту префикса, является искомой t' .

Если нет, то рассмотрим пути, не проходящие через m . Они имеют следующий вид:



Никакая вершина из p_t до достижимых таким образом вершин суффикса не может лежать в $I(P_t(V))$.

Запустим обход из вершин префикса по вершинам не из p_t , отмечая первые встреченные вершины суффикса и не продолжая из них обход (в таком случае позиция вершины m сдвигается вправо, и соответствующий префикс p_t увеличивается с сохранением инварианта). Будем продолжать обходы вершин префикса. Оставшаяся в конце вершина m и будет искомой, т.к. через неё будут проходить все пути из $P_t(V)$.

Для обхода из всех вершин префикса можно делать как бы один обход из них, сохраняя общее состояние посещённых вершин. Таким образом, поиск t' , как и s' , будет осуществляться за $O(|N| + |E|)$, а весь алгоритм поиска гамака – за $O(|N|(|N| + |E|)) = O(|N| \cdot |E|)$

Заметим, что полученный в результате работы алгоритма гамак – минимально возможный среди гамаков, содержащих N_c , потому что каждый раз добавляются только те вершины, которые уже точно будут лежать в любом гамаке, содержащем N_c . Таким образом, для любого мн-ва N_c существует минимальный по включению гамак, его содержащий.

Эффективный алгоритм

Основываясь на описанных выше идеях и используя введённые обозначения, опишем более эффективный алгоритм поиска гамака.

Найдём произвольные пути p_s, p_t из n_s в некоторую вершину V и из некоторой вершины V в n_t соответственно.

Будем поддерживать несколько множеств вершин (исток и сток искомого гамака будем обозначать как s и t соответственно):

- *серые*: *промежуточные* (т.е. не являющиеся истоком/стоком) вершины в искомом гамаке, из которых ещё не просмотрены соседи
- *чёрные*: *промежуточные* вершины в искомом гамаке, из которых уже просмотрены соседи
- *новые*: вершины, которые точно лежат внутри гамака, но не обязательно являются *промежуточными* (т.е. *серые* вершины с потенциально новыми s, t)
- вершины s и t , которые будут являться кандидатами на исток и сток (гарантированно вершины до них на путях p_s и p_t не могут быть истоком/стоком, и никакая вершина после них не является *серой* или *чёрной*). Другими словами, исток s будет соответствовать вершине m из предыдущего пункта для пути p_s , если считать *серые* и *чёрные* вершины *отмеченными*. Аналогично сток t будет соответствовать вершине m для пути p_t .

Каждая из рассмотренных вершин будет принадлежать ровно одному из этих множеств.

Пусть мн-ва корректны, согласно их определениям, тогда шаг алгоритма:

1. Просматриваем *новые* вершины. Если среди них есть вершины суффиксов p_s и p_t , то обновим соответственно выбранные s и t . Те *новые* вершины, которые не стали новыми s и t , поместим в *серые*. Теперь мн-во *новых* вершин пусто.
2. Если были обновлены s, t , то добавим промежуточные вершины суффиксов p_s, p_t как *серые* (они не могут быть *чёрными* в силу инварианта для s, t , однако какие-то из них уже могут быть *серыми* после 1-го действия).

Если была обновлена вершина s , и $s \neq t$, то добавим $s \bullet$ в качестве *новых* вершин, если они ещё не являются *серыми* или *чёрными*¹ (кажется, что они могут уже лежать в любом из этих мн-в). Если была обновлена вершина t , и $t \neq s$, то аналогично добавим $\bullet t$.

3. Просматриваем *серые* вершины, добавляя их соседей как *новых*, если те ещё не являются *серыми* или *чёрными*. Теперь мн-во *серых* вершин пусто.²

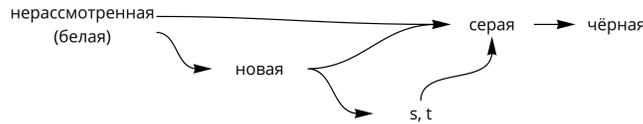


Рис. 9: Возможные изменения класса вершины в ходе алгоритма поиска гамака.

Будем повторять этот шаг, пока остаются *новые* вершины. Тогда, если на каком-то шаге *новых* вершин не было добавлено, то это означает, что мн-во *чёрных* вершин вместе с s, t образует гамак.

Описанный шаг соответствует достаточно простому расширению мн-ва V , равного объединению *чёрных* вершин с s, t , а введённые обозначения и поддерживаемые инварианты нужны лишь для выделения шагов этого расширения и аккуратного отслеживания уже рассмотренных вершин.³

Оценим сложность алгоритма. Пусть H – мн-во вершин искомого гамака, а E_H – мн-во ребёр в индуцированном относительно H подграфе G . Каждая вершина гамака будет *новой* или *серой* не более одного раза. Считаем, что можем проверять принадлежность вершины определённому мн-ву за $O(1)$ (например, с помощью хеш-таблицы).

- На каждую *новую* вершину будет потрачено $O(1)$ времени. Суммарно – $O(|H|)$
- На каждую *серую* вершину v будет потрачено $O(d_v)$ времени, где d_v – степень вершины. Т.к. *серая* вершина является *промежуточной* в гамаке, то суммарно – $O(|E_H|)$
- При каждом обновлении вершин s, t будут просмотрены их соседи, лежащие в гамаке. Суммарно – $O(|E_H|)$. Также будут просмотрены вершины суффиксов путей, которые являются *промежуточными*, суммарно – $O(|H|)$

¹Условие на несовпадение s и t необходимо для того, чтобы допускать гамаки, в которых исток и сток совпадают

²3-е действие шага является шагом BFS, поэтому, вероятно, можно было бы вместо него делать полноценный обход, каким-то образом изменив выделяемые множества вершин

³Возможно, существует более простой способ описать этот же алгоритм

Таким образом, сложность цикла шагов: $O(|H| + |E_H|)$.

С учётом нахождения путей p_s, p_t за $O(|N| + |E|)$ сложность всего алгоритма составит $O(|N| + |E|)$.

Нерассмотренным осталось стартовое состояние до первого шага алгоритма:

Можем положить мн-ва *серых* и *чёрных* вершин пустыми, мн-во *новых* – равным N_c , а в s, t записать специальные значения так, чтобы на первом шаге алгоритма в ходе 1-го действия они были корректно обновлены.

Псевдокод описанного алгоритма приведён в Приложении А.2.

5.2.b Задача покрытия мн-ва вершин гамаками с условиями нахождения вершин в одном гамаке

Задача: Заданы граф потока управления $G = (N_G, E_G, n_s, n_t)$, а также мн-во пар вершин из $N_G - E_C$.

Необходимо найти мн-во непересекающихся гамаков в G такое, что

- каждая из вершин, встречающихся в парах E_C , содержится в каком-то гамаке
- для любой пары из E_C обе её вершины содержатся в одном гамаке

Рассмотрим E_C как мн-во рёбер неориентированного графа $G_C = (N_C, E_C)$. Тогда условие означает попадание каждой компоненты связности G_C в один гамак. Опишем простой алгоритм:

Будем поддерживать некоторое мн-во гамаков и следующий инвариант: каждая компонента либо полностью покрыта одним гамаком, либо не покрыта вовсе ни одним из них. Псевдокод:

while остались непокрытые компоненты **do**

 Положим мн-во V равным какой-то непокрытой компоненте

while True do

 Найдём мин. гамак, покрывающий V

 ▷ Этот гамак может пересекаться с другими вершинами G_C

if гамак пересекается с непокрытой компонентой **then**

 Добавим всю пересекаемую компоненту в V

end if

if гамак пересекает другой гамак **then**

 Добавим все вершины пересекаемого гамака в V и удалим этот гамак из мн-ва

end if

if гамак не пересекает никакую другую компоненту **then**

 Добавим гамак в мн-во

break

end if

end while

end while

Оценим сложность алгоритма. Пусть поиск мин. гамака, покрывающего мн-во вершин V занимает $T(G)$ операций. Пусть k – число компонент связности в G_C . Тогда суммарно пересечений V с другими компонентами будет не более $k - 1$ (т.к. каждое пересечение объединяет хотя бы две компоненты в один гамак). В худшем

случае пересечений будет ровно $k - 1$ (таким образом, каждое пересечение происходит ровно с одной другой компонентой), и мин. гамак будет искаяться суммарно $2k - 1$ раз. Тогда сложность всего алгоритма можно оценить как $O(k \cdot T(G))$.

При использовании описанного в пункте 5.2.а алгоритма поиска мин. гамака, покрывающего V : $T(G) = O(|N_G| + |E_G|)$, и тогда общая сложность описанного алгоритма составит $O(k \cdot (|N_G| + |E_G|))$.

Вероятно, можно улучшить этот алгоритм, возможно, как-то умнее учитывая уже найденные гаммаки. Однако имеющаяся сложность кажется удовлетворительной, и в общем алгоритме исправления сетей она не оказывается «узким местом».

6 Реализация

Все описанные алгоритмы реализованы на языке Python с использованием библиотеки RM4PY, исходный код размещён в Github-репозитории¹. Структура репозитория изображена на Рис. 10.

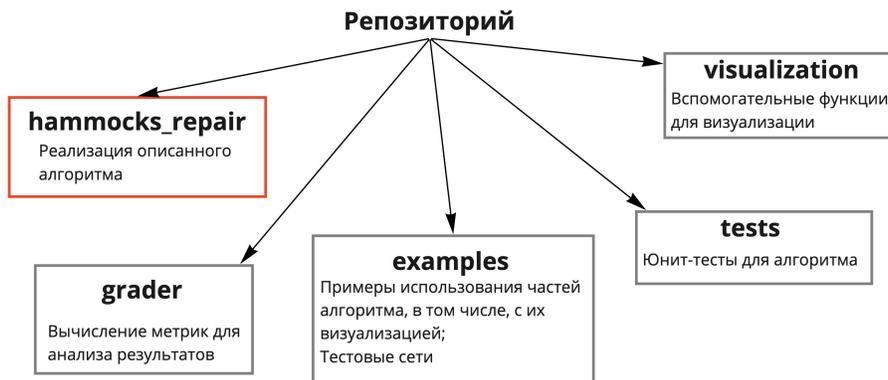


Рис. 10: Структура репозитория.

Реализация алгоритма находится в директории `hammocks_repair`, она независима от остального кода, который использовался для генерации тестов, проверки корректности, визуализации и оценки результатов применения алгоритма и составляет большую часть репозитория. Схема реализации алгоритма (содержимое `hammocks_repair`) вместе с функциями, доступными извне (их сигнатуры схематичны), изображена на Рис. 11. Каждая из упомянутых на схеме функций снабжена документацией в виде `docstring`.

В `hammocks_repair` я постарался организовать код приближённо к RM4PY (параметры, сигнатуры функций) для последующей возможности встраивания в библиотеку.

Рассмотрим базовый пример применения алгоритма исправления сети:

```
> git clone https://github.com/TrickmanOff/hammocks_repair_project
```

¹https://github.com/TrickmanOff/hammocks_repair_project

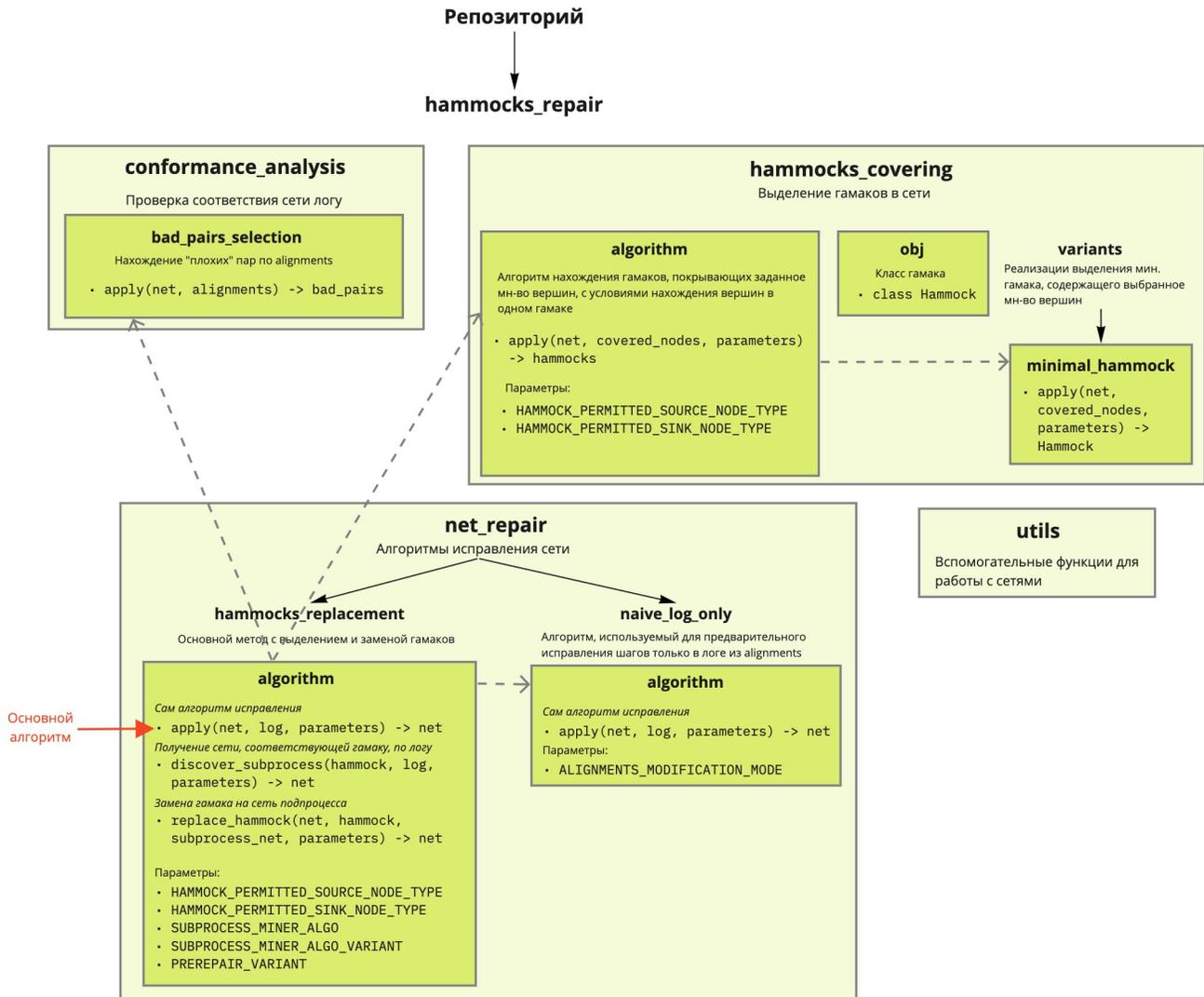


Рис. 11: Схема реализации алгоритма.

```
> pip install ./hammocks_repair_project
```

Напишем Python-скрипт:

Пусть в отдельной директории data находятся сеть Петри data/given_net.pnml и лог data/log.xes для исправления (в данном примере сеть была составлена вручную, а лог сгенерирован по ней, затем сеть была вручную «испорчена»). Импортируем их:

```
# importing net and log
from pm4py.objects.petri_net.importer import importer as pnml_importer
from pm4py.objects.log.importer.xes import importer as xes_importer

net, im, fm = pnml_importer.apply('data/given_net.pnml')
log = xes_importer.apply('data/log.xes')
```

Применим алгоритм исправления с некоторыми параметрами:

```
from hammocks_repair.net_repair.hammocks_replacement import algorithm as ham_repl_algo
from hammocks_repair.net_repair.naive_log_only import algorithm as naive_log_algo
```

```

NodeTypes = ham_repl_algo.NodeTypes
# parameters
parameters = {
    ham_repl_algo.Parameters.HAMMOCK_PERMITTED_SOURCE_NODE_TYPE:
        NodeTypes.PLACE_TYPE | NodeTypes.NOT_HIDDEN_TRANS_TYPE,
    ham_repl_algo.Parameters.HAMMOCK_PERMITTED_SINK_NODE_TYPE:
        NodeTypes.PLACE_TYPE | NodeTypes.NOT_HIDDEN_TRANS_TYPE,
    naive_log_algo.Parameters.ALIGNMENTS_MODIFICATION_MODE:
        naive_log_algo.AlignmentsModificationMode.LOG2SYNC,
    # to prevent alignments recalculation by modifying the alignments during the "prerepair"
}

# applying the algorithm
rep_net, rep_im, rep_fm = ham_repl_algo.apply(net, im, fm, log, parameters=parameters)

```

Визуализируем результат (Рис. 12 и 13)

```

# visualization
from pm4py.visualization.petri_net import visualizer as pn_visualizer
pn_visualizer.save(pn_visualizer.apply(net, im, fm), 'initial_net.png')
pn_visualizer.save(pn_visualizer.apply(rep_net, rep_im, rep_fm), 'repaired_net.png')

```

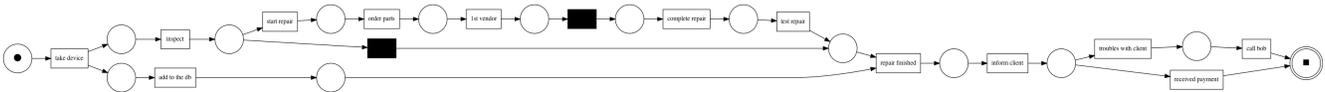


Рис. 12: Исходная сеть из примера (*initial_net.png*).

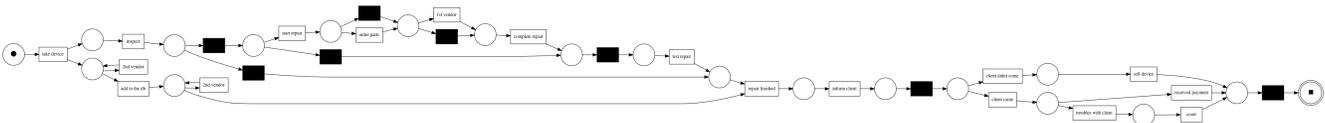


Рис. 13: Исправленная сеть из примера (*repaired_net.png*).

С помощью функций из `grader` можно визуализировать шаги применения алгоритма, а также получить значения метрик из Раздела 7. Рассмотрим пример (находясь в директории вместе с клонированным репозиторием):

```

from hammocks_repair_project.grader import grader

# указываем рассматриваемые тестовые директории,
# в них должны быть log.xes и given_net.pnml
test_dirs = ['data']

parameters = ... # из предыдущего примера

# далее указываем используемые метрики, убирая вычисление graph edit distance similarity с помощью ProM
# для его использования нужно изменить константы в grader/metrics/graph_edit_similarity_prom.py
metrics = set(grader.DEFAULT_METRICS_USED)
metrics.difference_update([grader.Metrics.EDIT_SIM])

# применяем алгоритм замены гамаков
grader.apply_hammocks_repair(test_dirs, parameters=parameters)
# применяем complete rediscovery для сравнения
grader.apply_complete_rediscovery(test_dirs)
# результаты каждого метода - в data/repaired_nets

# вычисляем метрики (будут записаны в grade_info.json)
grader.grade(test_dirs, metrics_used=metrics)

# далее можем вывести информацию в относительно удобном формате

```

```
from hammocks_repair_project.grader import pretty_printer
pretty_printer.pretty_print(test_dirs, 'results.txt') # таблица будет записана в results.txt
```

В результате запуска этого скрипта в директории `data` появятся:

- поддиректория `repaired_nets`, в которой будут храниться исправленные сети в формате `pnml`
- поддиректория `visualization`, в которой будут визуализированы промежуточные шаги применения алгоритма замены гамаков
- файл `grade_info.json`, в котором будет записана информация о сетях, применённых методах и вычисленных метриках

И информация из `data/grade_info.json` в удобном формате будет записана в `results.txt`.

Визуализации из `data/visualization` и содержимое `results.txt` представлены в Приложении В.2.

В `tests` написаны небольшие, но покрывающие некоторые крайние случаи, юнит-тесты для

- Алгоритма выделения плохих пар
- Алгоритма выделения мин. гамака, покрывающего заданное мн-во
- Алгоритма выделения мин. мн-ва гамаков с условиями на покрытие вершин

и проверки корректности работы общего алгоритма исправления.

Запуск тестов:

```
> python3 hammocks_repair_project/tests/execute_tests.py
```

7 Оценка результатов

7.1.a Используемые метрики

Для оценки работы алгоритма были использованы распространённые метрики, которые приводятся во многих статьях с алгоритмами исправления модели:

- *fitness*

Характеризует то, насколько лог воспроизводим в модели (соответствует `recall` в Data Science, если считать сеть классификатором следов по их воспроизводимости в ней). Здесь для оценки будет использоваться воспроизводимость следа по `alignment`. Соответствующий алгоритм реализован в РМ4РУ.

- fit_{avg} (от 0 до 1) – для каждого следа определяется «степень» его воспроизводимости в модели, далее берётся среднее значение по всем следам из лога
- fit_{tr} (от 0 до 1) – доля полностью воспроизводимых следов из лога

Если алгоритм исправления реализован корректно, то при использовании предварительного исправления шагов только в логе лог должен полностью воспроизводиться в исправленной сети, следовательно, должны быть значения $fit_{avg} = fit_{tr} = 1$ (вполне естественно ожидать 100%-го значения `fitness` при исправлении сети).

- *precision*

Характеризует то, насколько модель допускает поведение (воспроизводимость следа), которое не встречается в логге (соответствует *precision* в Data Science). Здесь для оценки также будет использоваться воспроизводимость следа по *alignment*. Соответствующий алгоритм реализован в РМ4РУ.

- (a) *prec* (от 0 до 1) – значение *precision* исправленной модели по отношению к заданному логге. Большее значение соответствует меньшим возможным отклонениям от логга

- *graph edit distance similarity*

Соответствует степени «похожести» сетей по некоторому редакторскому расстоянию (похоже на расстояние Левенштейна) от 0 до 1 (1 – идентичные сети). Вычисление этой метрики для сетей Петри, являющихся моделями процесса, описано в статье [6]. Для вычисления был использован плагин *Calculate Graph Edit Distance Similarity* для ProM [7]. Однако он отработывал за разумное время только для сетей с размером не более ~100 вершин, поэтому для больших сетей был использован алгоритм из Python-библиотеки *NetworkX* [8], который вычисляет некоторое приближение редакторского расстояния между графами. При его использовании применялись формулы стоимости операций с графом, приближённые к представленным в статье [6], чтобы получить значение, похожее на *Calculate Graph Edit Distance Similarity*.

- (a) *sim* (от 0 до 1) – значение «похожести» исправленной сети и исходной исправляемой, вычисленное с помощью плагина
- (b) *sim** (от 0 до 1) – приближение значения «похожести» исправленной сети и исходной исправляемой, вычисленное с помощью алгоритма из *NetworkX*

- *footnotes similarity*

Другая метрика «похожести» сетей. Для сети на m -ве переходов задаётся отношение порядка $<_L$: для двух действий a, b , $a <_L b \Leftrightarrow \exists \sigma$ – след, воспроизводимый в сети, такой, что в нём метка b идёт сразу после метки a (заметим, что какие-то пары могут оказаться несравнимыми). Далее для каждой пары действий (a, b) определяется её тип:

последовательная, если $a <_L b$ и $b \not<_L a$

параллельная, если $a <_L b$ и $b <_L a$

несравнимая, иначе

Тогда значение «похожести» сетей – доля пар действий (объединение меток из одной сети и другой) таких, что их типы для сетей не совпадают.

Алгоритм вычисления *footnotes similarity* реализован в РМ4РУ.

- (a) *sim_f* (от 0 до 1) – значение *footnotes similarity* исправленной сети и исходной исправляемой

Про *fitness*, *precision* и *footnotes similarity* подробно и формально написано в [1].

7.1.b Результаты тестирования

Для тестирования были взяты 3 искусственно сгенерированные сети Петри: А, В (из [9]) и С (из [10]). Далее каждая из сетей была слегка точно «испорчена» следующим образом:

В сети выделялись один или несколько непересекающихся гамаков. Далее каждый гамак рассматривался как сеть, по ней генерировался лог (встроенными средствами РМ4РУ). Затем этот лог «портился»: из него удалялось случайное подмножество следов или какие-то случайные действия менялись местами/удалялись. По этому «испорченному» логу строилась сеть, которая вставлялась вместо гамака.

По исходной сети генерировался лог достаточно небольшого размера (150 следов), по которому далее исправлялась «испорченная» сеть.

Результат сравнивался с применением алгоритма обнаружения процесса по всему логу с помощью inductive miner (complete rediscovery). Значения метрик приведены в Табл. 1, индексы у названий сетей в таблице соответствуют различным вариантам их «ухудшения».

		fit_{avg}	fit_{tr}	$prec$	sim	sim^*	sim_f	$size$	$time$	$time_a$	$time_p$	$time_h$
A1	initial	0.986	0.887	0.72	-	0.52	1	110	-	-	-	-
	hammocks	1	1	0.681	0.931	0.509	0.982	116	1.84	1.3	0.48	0.06
	rediscovery	1	1	0.056	0.503	0.394	0.021	77	0.68	-	-	-
A2	initial	0.948	0.653	0.714	-	0.504	1	109	-	-	-	-
	hammocks	1	1	0.67	0.899	0.484	0.968	116	1.79	1.26	0.48	0.05
	rediscovery	1	1	0.055	0.498	0.403	0.021	90	0.9	-	-	-
B1	initial	0.984	0.7	0.617	f	0.5	1	280	-	-	-	-
	hammocks	1	1	0.635	f	0.5	0.999	285	6.37	4.97	1.3	0.1
	rediscovery	1	1	0.492	f	0.462	0.971	236	0.69	-	-	-
B2	initial	0.925	0.46	0.668	f	0.504	1	276	-	-	-	-
	hammocks	1	1	0.615	f	0.504	0.997	284	6.39	4.91	1.34	0.14
	rediscovery	1	1	0.438	f	0.47	0.965	241	0.33	-	-	-
C1	initial	0.985	0.767	0.615	f	0.504	1	704	-	-	-	-
	hammocks	1	1	0.334	f	0.486	f	750	17.03	12.3	2.99	1.74
	rediscovery	1	1	0.093	f	0.476	f	741	6.67	-	-	-
C2	initial	0.985	0.693	0.581	f	0.501	1	706	-	-	-	-
	hammocks	1	1	0.287	f	0.486	f	747	29.22	25.8	2.67	0.75
	rediscovery	1	1	0.097	f	0.463	f	800	6.61	-	-	-

Таблица 1: Результат применения алгоритма. Значение f в ячейке соответствует тому, что соответствующая метрика не была вычислена за разумное время

Другие колонки в таблице:

- $size$ – кол-во вершин в сети
- $time$ – суммарное время работы алгоритма (сек.)
- $time_a$ – время вычисления alignments (сек.)
- $time_p$ – время предварительного исправления сети (сек.)
- $time_h$ – время выполнения алгоритма выделения и замены гамаков (сек.)

7.1.с Выводы

Время работы непосредственно алгоритма исправления (без учёта нахождения alignments) соизмеримо с временем complete rediscovery, что говорит о возможной применимости алгоритма и относительной опти-

мальности его реализации. Предварительное исправление занимает значительно больше времени, чем замена гамаков, но в используемом для этого методе есть простор для улучшения, т.к. в нём был использован достаточно наивный алгоритм. В любом случае, как и ожидалось, время работы основного алгоритма в разы меньше времени нахождения alignments, поэтому на данный момент «узким местом» является именно вычисление alignments.

В ходе тестовых запусков было закономерно выявлено, что алгоритм показывает хороший результат, если в сети есть небольшое число точечных несоответствий с логом, причём в некоторой отдалённости от стартовой и конечной позиций, тогда гамаки могут достаточно точно локализовать «проблемное» место. В Табл. 1 можно видеть заметно лучшие показатели у алгоритма по сравнению с обнаружением процесса по всему логу, что дополнительно связано с ограниченным размером лога, по которому может не получиться качественно восстановить исходную модель.

Однако было замечено, что в некоторых случаях, даже при относительно небольших изменениях сети, в качестве гамака выделяется вся сеть (таким образом алгоритм становится эквивалентным complete rediscovery). Это было связано с тем, что встраиваемые в ходе предварительного исправления переходы соединяли разные параллельные ветки сети, и далее, либо ввиду предлагаемой замены шагов только в логе на шаги только в модели в alignments (5.1.c), либо ввиду необходимости выделять гамаки так, чтобы они содержали все переходы с одной меткой (5.1.b), мин. гамак, покрывающий выделенные плохие пары, становился большим. Обе эти трудности возникают из-за алгоритма предварительного исправления, но без него пришлось бы потерять 100%-й fitness.

8 Заключение

Алгоритм показывает вполне ожидаемые результаты, поэтому предложенное использование гамаков можно считать оправданным и имеющим свои преимущества для рассматриваемых случаев точечного несоответствия сети логу, и можно говорить о том, что алгоритм применим для исправления моделей. Однако, вероятно, стоит ещё подумать над методом исправления шагов только в логе из alignments или его дальнейшем использовании при выделении гамаков, потому что из-за текущего метода на этом этапе зачастую возникают трудности. Возможно, более подходящий метод можно найти в других научных статьях.

В дальнейшем я бы хотел оформить описанный алгоритм в виде научной статьи, чтобы её можно было где-то представить/опубликовать. Без этого, наверное, рано обсуждать включение реализации алгоритма в RM4PY.

Без учёта отсутствия апробации алгоритма, реализация написана похожим на код из RM4PY образом, поэтому ожидается, что её легко будет встроить в библиотеку. Возможно, придётся внести какие-то незначительные правки, написать более подробную документацию и составить покрывающие большее число случаев тесты.

СПИСОК ИСТОЧНИКОВ

- [1] Wil van der Aalst. *Process Mining: Data Science in Action*. Springer, 2016.
- [2] Fubo Zhang и E. H. D'Hollander. «Using hammock graphs to structure programs». в: *IEEE Transactions on Software Engineering* 30 (2004), с. 231—245. ISSN: 0098-5589. DOI: 10.1109/tse.2004.1274043.
- [3] Dirk Fahland и Wil M. P. van der Aalst. «Model repair — aligning process models to reality». в: *Information Systems* 47 (2015), с. 220—243. ISSN: 0306-4379. DOI: 10.1016/j.is.2013.12.007.
- [4] *ProM - Framework for process mining*. <https://www.promtools.org/>. Посещён 02-02-2022.
- [5] Process Mining Group of the Fraunhofer Institute for Applied Information Technology. *PM4PY*. <https://pm4py.fit.fraunhofer.de/>. Посещён 02-02-2022.
- [6] Remco Dijkman и др. «Similarity of business process models: Metrics and evaluation». в: *Information Systems* 36 (2011), с. 498—516. ISSN: 0306-4379. DOI: 10.1016/j.is.2010.09.006.
- [7] B.F.A Hompes. «On Decomposed Process Discovery: How to Solve a Jigsaw Puzzle with Friends». дис. ... мар. Eindhoven University of Technology, 2014.
- [8] *NetworkX - Python library for studying graphs and networks*. <https://networkx.org/>. Посещён 12-05-2022.
- [9] Vincent Bloemen. *Generated and industrial PNML models with generated log traces (in XES)*. <https://doi.org/10.4121/uuid:5f168a76-cc26-42d6-a67d-48be9c978309>. 4TU.ResearchData. Dataset. 2018.
- [10] Jorge Munoz-Gama. «Conformance Checking in the Large» (*BPM 2013*). <https://doi.org/10.4121/uuid:44c32783-15d0-4dbd-af8a-78b97be3de49>. 4TU.ResearchData. Dataset. 2013.

А Алгоритмы

А.1 Выделение «плохих» пар

```
1: struct Token {
2:   node, GreenNodes, RedNodes
3: }
4:
5: start – стартовая позиция сети
6: end – конечная позиция сети
7:
8: init_token := new Token(start, {start},  $\emptyset$ )
9: Tokens := {init_token}
10:
11: BadPairs :=  $\emptyset$  ▷ искомые плохие пары
12:
13: for шаг  $\in$  alignment do
14:   if шаг – только в логе then
15:     | continue ▷ пропускаем шаги только в модели
16:   end if
17:
18:    $T_a$  – активируемый переход в модели, соответствующий шагу
19:
20:   ConsumedTokens :=  $\emptyset$  ▷  $t_1, \dots, t_k$ 
21:   ConsumedFrom :=  $\bullet$  start
22:   for token  $\in$  Tokens do
23:     | if token.node  $\in$  ConsumedFrom then
24:       | | ConsumedFrom  $-=$  token.node
25:       | | ConsumedTokens  $+=$  token
26:     | end if
27:   end for
28:
29:   TpGreenNodes :=  $\emptyset$ 
30:   TpRedNodes :=  $\emptyset$ 
31:
32:   if шаг – только в модели then
33:     | if  $T_a$  – скрытый переход then
34:     | | TpGreenNodes :=  $\bigcup_{\text{token} \in \text{ConsumedTokens}}$  token.GreenNodes
35:     | | TpRedNodes :=  $\bigcup_{\text{token} \in \text{ConsumedTokens}}$  token.RedNodes
36:     | else
37:     | | TpGreenNodes :=  $\emptyset$ 
38:     | | TpRedNodes :=  $\bigcup_{\text{token} \in \text{ConsumedTokens}}$  token.RedNodes  $\cup$  token.GreenNodes
39:     | end if
```

```

40: | else ▷ синхронный шаг
41: | | for  $v \in \bigcup_{\text{token} \in \text{ConsumedTokens}} \text{token.RedNodes}$  do
42: | | |  $\text{BadPairs} += (v, T_a)$ 
43: | | end for
44: | |  $\text{TpGreenNodes} := \{T_a\}$ 
45: | |  $\text{TpRedNodes} := \emptyset$ 
46: | end if
47: |  $\text{Tokens} -= \text{ConsumedTokens}$ 
48: | for  $v \in T_a \bullet$  do
49: | |  $\text{token} := \text{new Token}(v, \text{TpGreenNodes}, \text{TpRedNodes})$  ▷  $t_p$ 
50: | |  $\text{Tokens} += \text{token}$ 
51: | end for
52: end for
53:
54: for  $v \in \bigcup_{\text{token} \in \text{Tokens}} \text{token.RedNodes}$  do
55: |  $\text{BadPairs} += (v, \text{end})$ 
56: end for

```

A.2 Поиск наименьшего гамака, содержащего заданное мн-во вершин

Пусть s_path, t_path – пути p_s, p_t , а s_path_ind и t_path_ind – отображения из вершин соответствующих путей в порядковые индексы на них.

```

1:  $\text{Black} := \text{Gray} := \emptyset$ 
2:  $s\_ind := t\_ind := -1$  ▷ индексы вершин  $s, t$  на путях
3:  $s, t$ 
4:  $\text{NewNodes} := N_c$ 
5: while  $\text{NewNodes} \neq \emptyset$  do
6: |  $\text{new\_s\_ind} := s\_ind$ 
7: |  $\text{new\_t\_ind} := t\_ind$ 
8: | ▷ 1-е действие
9: | for  $u \in \text{NewNodes}$  do
10: | | if  $u \in s\_path$  then
11: | | |  $\text{new\_s\_ind} := \max(\text{new\_s\_ind}, s\_path\_ind[u])$ 
12: | | |  $\text{new\_t\_ind} := \max(\text{new\_t\_ind}, t\_path\_ind[u])$ 
13: | | end if
14: | end for
15: | for  $u \in \text{NewNodes}$  do
16: | | if  $u \neq s\_path[\text{new\_s\_ind}]$  and  $u \neq t\_path[\text{new\_t\_ind}]$  then
17: | | |  $\text{Gray} += u$  ▷ в смысле множеств
18: | | end if
19: | end for
20: |  $\text{NewNodes} := \emptyset$ 
21: |  $s := s\_path[\text{new\_s\_ind}]$ 
22: |  $t := t\_path[\text{new\_t\_ind}]$ 
23: | ▷ 2-е действие
24: | if  $s\_ind \neq \text{new\_s\_ind}$  then
25: | | for  $i \in [s\_ind, \text{new\_s\_ind})$  do
26: | | |  $\text{Gray} += s\_path[i]$ 
27: | | end for

```

```

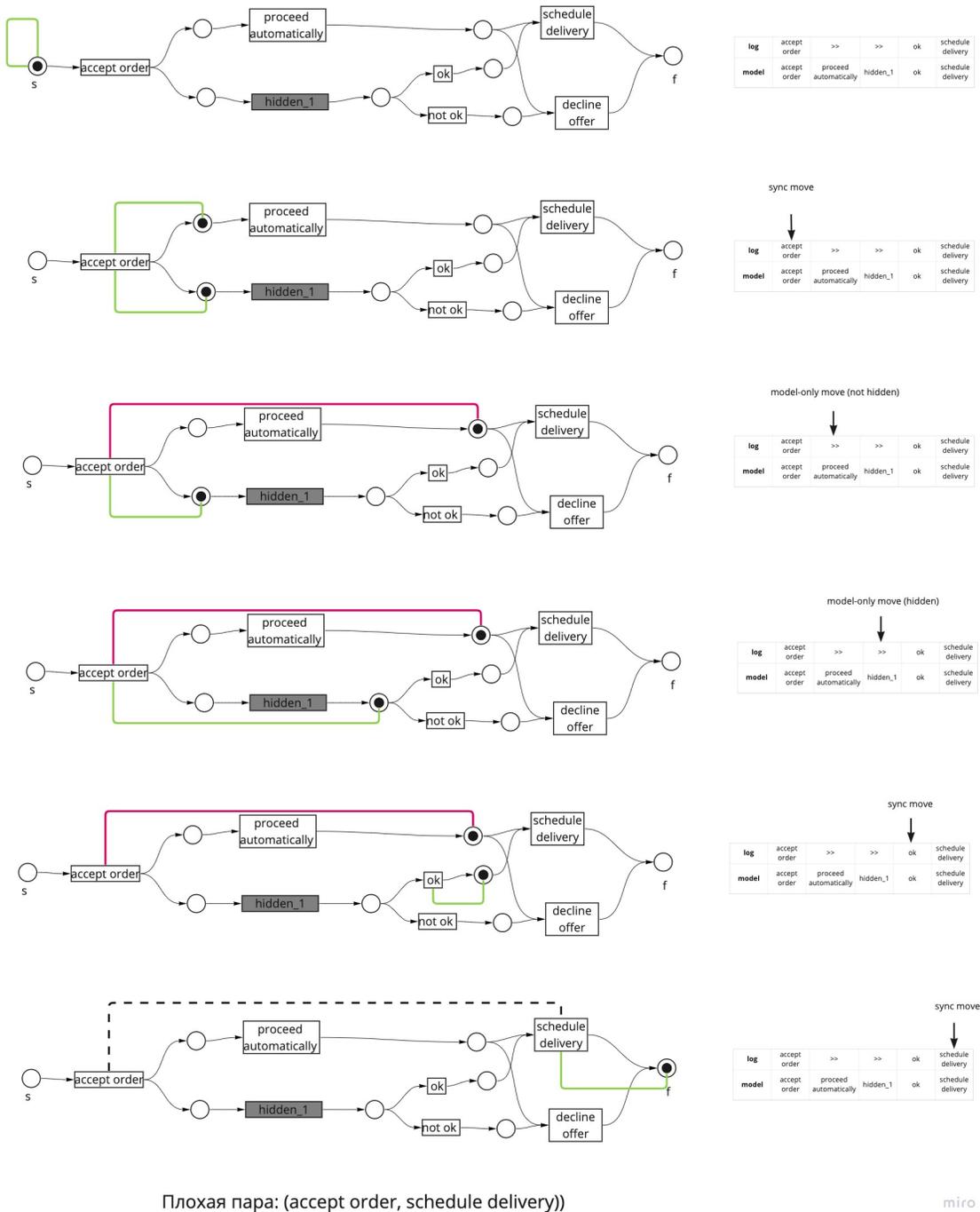
28:   |   if s ≠ t then
29:   |   |   for u ∈ OutNeighbors(s) do
30:   |   |   |   if u ∉ Black and u ∉ Gray then
31:   |   |   |   |   NewNodes += u
32:   |   |   |   end if
33:   |   |   end for
34:   |   end if
35:   |   s_ind := new_s_ind
36: end if
37: if t_ind ≠ new_t_ind then
38:   |   for i ∈ [t_ind, new_t_ind) do
39:   |   |   Gray += t_path[i]
40:   |   end for
41:   |   if s ≠ t then
42:   |   |   for u ∈ InNeighbors(t) do
43:   |   |   |   if u ∉ Black and u ∉ Gray then
44:   |   |   |   |   NewNodes += u
45:   |   |   |   end if
46:   |   |   end for
47:   |   end if
48: end if

49:   |   ▷ 3-е действие
50:   |   for u ∈ Gray do
51:   |   |   for v ∈ Neighbors(u) do
52:   |   |   |   if v ∉ Black and v ∉ Gray then
53:   |   |   |   |   NewNodes += v
54:   |   |   |   end if
55:   |   |   end for
56:   |   end for
57:   |   Gray := ∅
58: end while
59: Hammock := Black + s_path[s_ind] + t_path[t_ind]

```

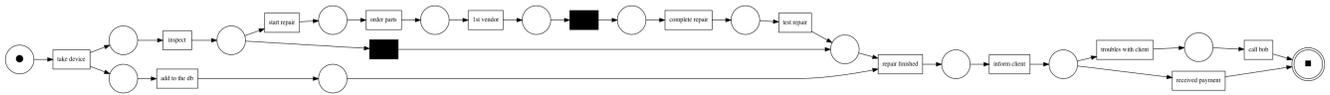
В Примеры применения алгоритмов

В.1 Пример выделения «плохих» пар

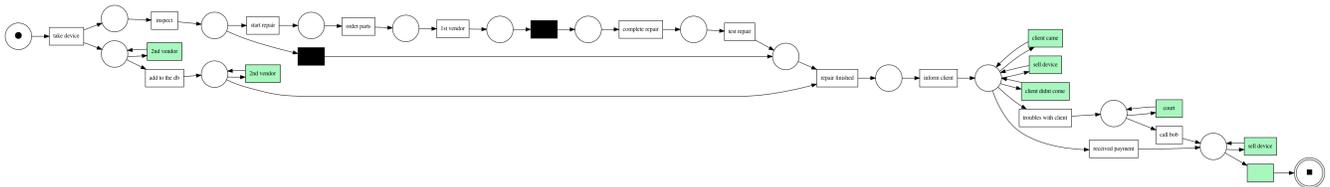


Обозначения такие же, как и при рассмотрении примера одного шага. Заметим, что при применении названного выделения пар была бы выделена пара (accept order, ok), что хуже соответствует несоответствию сети в сравнении с парой (accept order, schedule delivery).

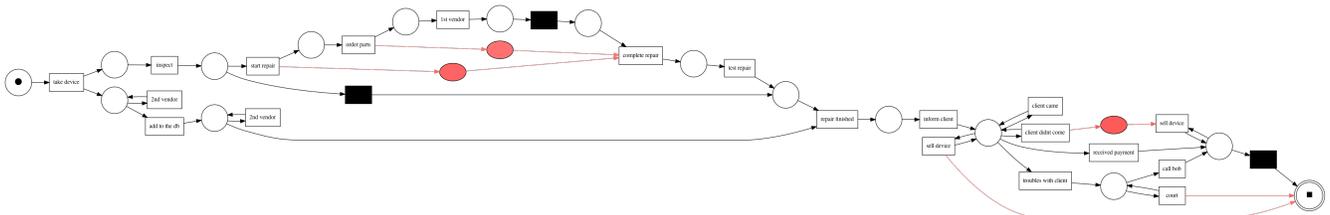
В.2 Простой пример всего алгоритма исправления



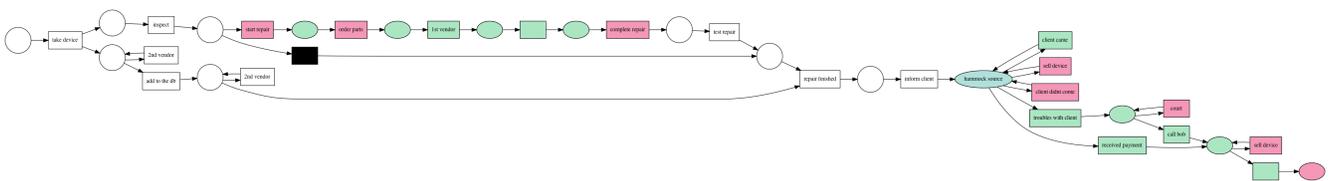
Исходная сеть (файл с именем *given_net.png*)



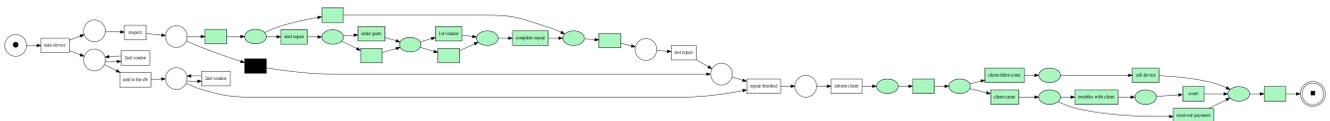
Сеть после предварительного исправления шагов только в логе (файл с именем *10-prerepaired.png*). Зелёным отмечены вставленные переходы.



Выделенные «плохие» пары вершин (файл с именем *20-bad_pairs.png*). Красные позиции вставлены только для того, чтобы сеть оставалась корректной для её визуализации (просто провести ребро между переходами нельзя).



Мин. мн-во гамаков, покрывающее «плохие» пары (файл с именем *30-bad_pairs_hammock.png*). Красным отмечены вершины, входящие в «плохие» пары, зелёным – вершины, входящие в гамаки.



Исправленная сеть (файл с именем *40-repaired_hammocks.png*). Зелёным отмечены подсети, которые были вставлены вместо гамаков.

Содержимое *results.txt*:

	name	method	fit_avg	fit_tr	prec	sim	sim_approx	sim_f	size	time	time_align	time_prerep	time_ham
0	data	given_net	0.793	0	0.876	nan	0.547	1	30	nan	nan	nan	nan
1	data	default_hammocks_replacement	1	1	0.692	nan	0.405	0.907	46	1.16	0.48	0.43	0.25
2	data	complete_rediscovery	1	1	0.589	nan	0.393	0.898	46	0.04	nan	nan	nan
3	-----	-----	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan	nan