Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

РАЗРАБОТКА АЛГОРИТМОВ РАБОТЫ С ФОРМАЛЬНОЙ МОДЕЛЬЮ ДИАЛОГОВ, ПРЕДСТАВЛЕННЫХ В ВИДЕ ГРАФОВ

Автор: Савон Юлия Константиновна _____

Направление подготовки: 01.03.02 Прикладная математика и информатика

Квалификация: Бакалавр

Руководитель ВКР: Ульянцев В.И., к.т.н. _____

Санкт-Петербург, 2020 г.

Обучающийся Савон Юлия Константиновна
Группа М3437 Факультет ИТиП

Направленность (профиль), специализация
Математические модели и алгоритмы в разработке программного обеспечения

Консультанты:

а) Ступаков И.М., канд. тех. наук, доцент _____

ВКР принята «_____» _____ 20___ г.

Оригинальность ВКР _____%

ВКР выполнена с оценкой _____

Дата защиты «23» июня 2020 г.

Секретарь ГЭК Павлова О.Н. _____

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

УТВЕРЖДАЮ

Руководитель ОП

проф., д.т.н. Парфенов В.Г. _____

«____» _____ 20___ г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Обучающийся Савон Юлия Константиновна

Группа М3437 **Факультет** ИТиП

Квалификация: Бакалавр

Направление подготовки (специальность): 01.03.02 Прикладная математика и информатика

Направленность (профиль): Математические модели и алгоритмы в разработке программного обеспечения

Тема ВКР: Разработка алгоритмов работы с формальной моделью диалогов, представленных в виде графов

Руководитель Ульянцев В.И., к.т.н., доцент факультета информационных технологий и программирования Университета ИТМО

2 Срок сдачи студентом законченной работы до: «1» июня 2020 г.

3 Техническое задание и исходные данные к работе

Требуется провести исследование и разработать набор алгоритмов для выявления отвлечений в графовой модели для голосовой диалоговой системы. Алгоритм принимает набор диалогов, в размере нескольких тысяч. Предварительно выстроенный граф и кластеризацию для фраз оператора.

На выходе ожидается получить набор отвлечений и перестроенный граф. В качестве метрики качества будет использоваться сравнение с уже существующими графами, которые создавались вручную.

4 Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов)

Пояснительная записка должна описывать предметную область диалогов представленных в виде графов. Так же формулировать цель и задачу выделения отвлечений, содержать описание алгоритмов их поиска. Должны быть описаны сложности и методы их разрешения, если они возникали. Кроме того должны быть приведены примеры работы алгоритмов и сравнение с существующими решениями. Кроме того пояснительная записка должна содержать описания задач из смежных областей и их то, как эти задачи связны с задачей решаемой в работе.

5 Перечень графического материала (с указанием обязательного материала)

Графические материалы и чертежи работой не предусмотрены

6 Исходные материалы и пособия

а) Среда разработки Visial Studio Code;

б) ГОСТ 7.32–2001 «Система стандартов по информации, библиотечному и издательскому делу. Отчет о научно-исследовательской работе. Структура и правила оформления».

7 Дата выдачи задания «01» сентября 2019 г.

Руководитель ВКР _____

Задание принял к исполнению _____ «01» сентября 2019 г.

Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

АННОТАЦИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Обучающийся: Савон Юлия Константиновна

Наименование темы ВКР: Разработка алгоритмов работы с формальной моделью диалогов, представленных в виде графов

Наименование организации, где выполнена ВКР: Университет ИТМО

ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

1 Цель исследования: Разработа алгоритма выделяющего отвлечения в диалоге, представленном в виде графа.

2 Задачи, решаемые в ВКР:

а) Разработать алгоритмы выделения отвлечий;
б) Реализовать описанные алгоритмы;
в) Перестроить граф в соответствии с используемой моделью в компании;
г) Проанализировать результаты работы алгоритмов;
д) Интегрировать разработки в инфраструктуру компании.

3 Число источников, использованных при составлении обзора: 0

4 Полное число источников, использованных в работе: 11

5 В том числе источников по годам:

Отечественных			**Иностранных**		
Последние 5 лет	От 5 до 10 лет	Более 10 лет	Последние 5 лет	От 5 до 10 лет	Более 10 лет
0	2	1	2	2	4

6 Использование информационных ресурсов Internet: да, число ресурсов: 1

7 Использование современных пакетов компьютерных программ и технологий:

Пакеты компьютерных программ и технологий	**Раздел работы**
Интегрированная среда разработки `PyCharm`	Глава 2.2
Программное обеспечение для автоматизации развертывания и управления приложениями в средах с поддержкой контейнеризации `Docker`	
Распределённая система контроля версий `Git`	Глава 2.2

8 Краткая характеристика полученных результатов

9 Гранты, полученные при выполнении работы

10 Наличие публикаций и выступлений на конференциях по теме выпускной работы

По теме этой работы был сделан доклад на Конгрессе Молодых Ученых.

Обучающийся Савон Ю.К. _____

Руководитель ВКР Ульянцев В.И. _____

«____» _____ 20___ г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Эта работа посвящена исследованию в области диалоговых систем.

Диалоговая система – это алгоритм, который умеет принимать участие в диалоге на естественном языке и использует правила общения между людьми.

В качестве примера диалоговых систем можно привести:

— Чат-ботов;

— Голосовых помощников;

— Автоответчиков в колл-центрах.

Такие диалоговые системы могут быть как довольно простыми (например, чат-бот отвечающий на заранее известный набор команд), так и сложными (бот, отвечающий на вопрос, сформулированный на естественном языке и возвращающий некоторую информацию из базы знаний).

В последнее время набрали популярность технологии распознавания и генерации речи, которые позволили создавать диалоговые системы, ведущие голосовой разговор. Данная работа рассматривает алгоритмы именно для голосовых диалогов. Пример решения такой диалоговой системы можно рассмотреть в статье [10].

Такие звонки, с одной стороны, должны быть не отличимы от звонков человека, с другой – они должны придерживаться некоторого сценария.

Сценарий диалога с оператором – некоторый алгоритм, предоставленный человеку, который звонит по некоторому набору номеров (либо же принимает входящий звонок или общается посредством программного обеспечения для аудиосвязи). Целью сценария обычно является получение или донесение до клиента информации.

Несмотря на то, что под сценарием диалога понимается некоторый алгоритм, необходимо понимать что для человека и диалоговой системы это принципиально разные сущности. Между скриптом[1] и алгоритмом, с точки зрения набора действий для машины есть большая разница.

Для человека это скорее структурированный список вопросов которые он должен задать и некоторая дополнительная база знаний с ответами на нестандартные вопросы. Кроме того человек может помнить некоторые фак-

[1]Здесь и далее в тексте слова «**скрипт**» и «**сценарий**» будут использоваться как синонимы

ты и выдавать их дополнительно в зависимости от контекста. Он сам умеет обрабатывать ситуации, такие как отвлечение от основных вопросов или переспрашивание.

Для скрипта же, реакцию на любую фразу человека надо прописывать, все возможные данные хранить и обновлять. Кроме того, есть требование поддерживать этот скрипт доступным для восприятия человеком (например лингвистом), поскольку возникает необходимость в ручном анализе и редактировании. В данном случае такой скрипт будет представлен в виде графа с дополнительной информацией.

Голосовая диалоговая система — программа, которая используя сценарий умеет проводить диалог с клиентом, интерпретировать и записывать информацию полученную от клиента, а так же состояния завершенного разговора. Кроме того, она умеет отвечать на заранее прописанный в скрипте набор вопросов и возвращаться к диалогу.

На данный момент существуют графы для диалогов, которые создаются вручную. Но писать их долго, а продумывать все важные случаи реакций сложно и трудозатратно.

Помимо этого, хочется иметь возможность усложнять вариативность диалогов, делая их более похожими на процесс общения двух людей. В связи с этим, ставится глобальная задача по созданию графа из набора проведенных диалогов.

Достаточно часто возникает ситуация, когда некоторый общий вопрос может возникнуть в любом месте диалога (например, «А что это за компания?»). Таким образом, было решено отделить их в отдельные подграфы и сделать возможность переходить в них при некоторых условиях из каждой вершины. В дальнейшем такие случаи будут называться **отвлечениями**.

В работе будут рассмотрены различные алгоритмы автоматического поиска таких отвлечений.

ГЛАВА 1. ОБЗОР ДИАЛОГОВОЙ МОДЕЛИ

1.1. Разговорная диалоговая система

Данная работа выполнялась в компании Dasha.AI Inc. В ней будут рассматриваться интеллектуальные диалоговые системы. Введем несколько определений.

Разговорная диалоговая система – система, позволяющая пользователю-человеку получать доступ к информации и сервисам доступным на компьютере или в сети Интернет, используя разговорный язык как средство взаимодействия, согласно [7].

Разговорные диалоговые системы используют распознавание речи для преобразования фраз человека в формат, удобный для использования диалоговым менеджером.

1.2. Диалоговый менеджер

Диалоговый менеджер – компонент диалоговой, системы ответственный за текущее состояние и ход диалога, согласно[9]. В основном, диалоговые менеджеры оперируют предобработанными текстами. У диалогового менеджера так же есть состояние, которое может хранить в себе например последний неотвеченный вопрос или историю диалога.

Диалоговый менеджер, в том числе можно представить в виде автомата, пример которого можно увидеть на Рисунке 1.

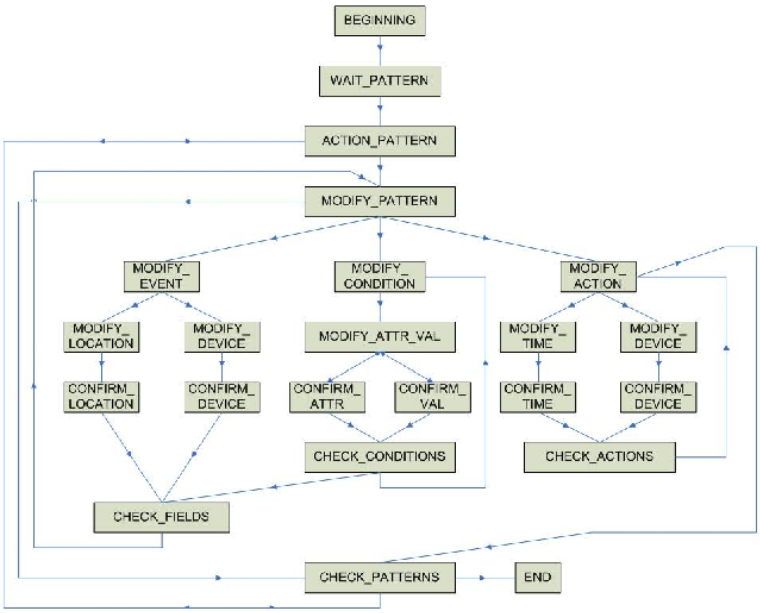


Рисунок 1 – Пример автомата для диалогового менеджера

В этом примере, в зависимости от текущего состояния и информации, предоставленной пользователем – система пытается принять решение о том, какое действие хочет выполнить пользователь. Эта система использует не только текущий ответ, а так же информацию о контексте. Кроме того, если системе не хватает информации, она запрашивает её у пользователя. Рисунок и описание взяты из [4].

Основные способы создания моделей диалоговых менеджеров:

— Составление вручную;
— Генерация модели менеджера с помощью машинного обучения.

Плюсами первого подхода можно назвать прозрачность процесса перевода имеющегося скрипта в модель. Серьёзным минусом создания модели диалогового менеджера вручную является необходимость каждый раз полностью создавать структуру диалога, что очень ресурсозатратно.

Плюсом генерации модели менеджера с помощью машинного обучения является невысокая трудозатратность на каждую новую систему. Минусом такого подхода можно назвать необходимость в большом количестве данных для обучения и зависимости от качества исходных данных.

Пример модели диалогового менеджера, улучшенного с помощью методов машинного обучения представлен в статье [6]. В указанной работе байесовский подход к изучению пользовательской модели применялся одновременно с политикой менеджера диалогов. Данное улучшение позволило организовать управление инвалидным креслом с помощью голоса, посредством инструкций более похожих на естественный язык.

В основном для промышленного использования диалоговых систем в компаниях используются диалоговые менеджеры на основе правил, составленных вручную. Создание одного такого диалогового менеджера может требовать много ресурсов, которые при таком подходе нельзя значительно соптимизировать.

В этой работе рассматривается построение на основе графов.

Задача, решаемая командой состоит в следующем: необходимо по набору диалогов восстановить диалоговую систему. Данная работа решает одну из подзадач.

Диалоговая система в данном случае представлена в виде графа, но так же существуют и другие представления. На Рисунке 2 представлен пример упрощенной модели, с несколькими вариантами ответов человека.

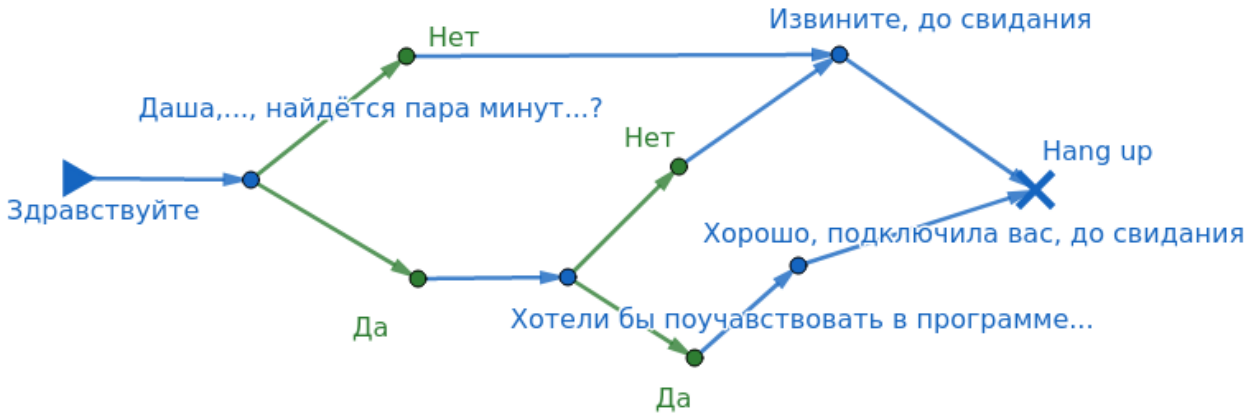


Рисунок 2 – Простой пример модели

На данный момент, кроме восстанавливаемой модели – разработаны графовые модели, которые составляются вручную. В упрощенном виде, в этих моделях вершинами являются фразы диалоговой системы, а рёбра ассоциированы с группами возможных ответов человека. То есть, при различных ответах человека происходит переход в разные вершины.

Особенностью данной структуры, которую важно отметить, является то, что помимо обычных переходов существуют также скрытые переходы, которые по умолчанию могут встретиться в любом месте. В качестве примера можно привести вопрос: «А откуда вы взяли мой номер телефона?». При звонке люди могут задать этот вопрос не сразу.

Для того, чтобы не рассматривать каждый такой случай при переходе из каждой вершины, выделяются **отвлечения**.

Отвлечение – это вопрос, который может быть задан в любом месте диалога. Кроме того, к одному из подтипов отвлечений относятся случаи, когда оператор не услышал фразу или вопрос и вынужден переспросить. В зависимости от модели, это может выделяться в специальную сущность или являться отвлечением.

В задаче восстановления графа по набору диалогов используется видоизменённая модель. В ней вершинами являются, как кластеры фраз оператора,

так и кластеры фраз человека. А ребро соответствует наличию перехода между соответствующими кластерами в диалоге.

Кластеризация текстов (фраз)[8] – автоматическая группировка текстовых документов, например веб-страниц, электронных писем, человеческих фраз, основанная на схожести содержимого. В качестве входных данных алгоритмы принимают наборы фраз и количество желаемых кластеров. В качестве выходных данных, алгоритм возвращает фразы, сгруппированные в указанное количество кластеров – кластеры $K_1, K_2, ...$

1.3. Анализ задачи выделения отвлечений

На вход подаётся набор диалогов, по которым нужно получить граф для диалоговой системы, которая бы могла проводить аналогичные диалоги.

Граф, если получать его путём обычной кластеризации операторских фраз, получается избыточно большим. В нём плохо видно структуру, его сложно анализировать.

Конечной целью является научиться строить графы, аналогичные тем, которые создаются другой командой в компании вручную. Исходя из этого, появляется требование привести его в состояние, когда человек мог бы изучить и проанализировать такой восстановленный граф.

То есть, так же как и в графе создаваемом вручную, появляется необходимость выделять отвлечения. В этом случае его структура становится более удобной для анализа. Алгоритм выделения, в частности, позволяет анализировать особенности графа непосредственно во время процесса выделения отвлечений.

Как следствие, особенности структуры отвлечений позволяют грамотно обрабатывать сценарии, которых не было в изначальном наборе диалогов. Если такие отвлечения не выделить, то в случае вопроса, не предусмотренного в этом месте диалоговая система либо даст некорректный ответ, либо несвоевременно закончит работу.

Хорошая кластеризация текстов является важным предварительным шагом для алгоритма, поскольку любой алгоритм выделения отвлечений непосредственно опирается на информацию, полученную в результате кластеризации. Полезно, при этом учитывать имеющуюся информацию о контексте, то есть фразы до текущей и после.

1.4. Постановка задачи

Целью данной работы является реализация возможности поиска отвлечений в алгоритме восстановления графа по набору диалогов.

Отвлечения могут встретиться в любом месте, таким образом после выделения и перестроения графа, граф будет более структурирован, что позволит строить более сложные конструкции для диалогов, которые всё ещё будут поддаваться анализу и контролю вручную.

Если провести аналогию для человека-оператора, то в его скриптах нет необходимости расписывать вопросы-отвлечения. Человек сам прекрасно понимает, когда ответ отклоняет его от скрипта. Но в отличие от оператора, диалоговой системе нужны однозначные инструкции, которые бы покрывали большинство возможных типов вопросов.

Необходимо выделить отвлечения и перестроить граф таким образом, чтобы отвлечения выделились в отдельный подграф, куда можно было бы перейти из любого места диалога. Такой граф будет покрывать большее количество возможных сценариев.

1.5. Выводы по первой главе

В данной главе была рассмотрена предметная область диалоговых систем, введены определения диалоговой системы, менеджера диалогов, отвлечения. Описаны подходы к построению менеджеров диалогов с соответствующими достоинствами и недостатками. Так же описано представление диалоговой системы в виде графа. Разобрана структура графа, используемого в продуктовом окружении, и структура графа для восстанавливаемой модели. Проведён анализ задачи восстановления графа. Поставлена задача выделения отвлечений.

ГЛАВА 2. ОПИСАНИЕ АЛГОРИТМОВ ПОИСКА ОТВЛЕЧЕНИЙ И РЕКЛАСТЕРИЗАЦИИ

2.1. Процесс разработки

Восстановление графа делается поэтапно, в этот процесс включены следующие этапы в соответствующем порядке:

1. Распознавание аудио.
2. Преобразование аудио в тексты;
3. Препроцессинг в виде преобразования и исправления текстов;
4. Кластеризация и восстановление графа;
5. Выделение отвлечений;
6. Конвертация графа для возможности запуска в существующей системе.

Данная работа не будет затрагивать первые 2 этапа.

В процессе разработки были сделаны улучшения для третьего этапа и затронут четвёртый. Основная работа касается этапа выделения отвлечений, кроме того написаны алгоритмы для преобразования, что даёт возможность сравнить графы с существующими.

Важным будет отметить, что поиск отвлечений встроен в алгоритм построения графа и таким образом появляется возможность внести некоторые изменения в его структуру. Одним из таких изменений может быть выделение отвлечений в отдельный подграф, что позволит рассматривать части независимо.

2.2. Задача выделения отвлечений

Поскольку на этапе выделения отвлечений в графе его перестроение всё ещё возможно, а информация о произнесенных человеком фразах является ценной, так как содержит в себе контекст, то фразы человека оставлены в качестве вершин.

Нужно понимать, что изначально кластеризация проводилась только по фразам оператора. Фразы вершин же были разделены на группы, где для каждой группы совпадала предыдущая и последующая вершины оператора. И уже внутри этих групп вершины разделялись на некоторые подгруппы.

Рассмотрим два подхода в решении задачи выделения отвлечений:

— Первый подход заключается в том, чтобы выделить вершины, у которых много входящих ребер. Порог считается функцией от количества кластеров на которые бьются фразы оператора.

  Мы предполагаем, что поскольку отвлечение встречается в разных местах, то и рёбра будут идти в него из множества различных вершин. Такая гипотеза обосновывается следующим наблюдением: в части графа без отвлечений большинство вершин имеет рёбра из одной или двух вершин, поскольку в противном случае скрипт становится слишком сложным, в случае же отвлечения их должно быть много (исключением может быть лишь отвлечение, которое встречается крайне редко).

— В качестве другого подхода можно выделить циклы и сказать, что вершина следующая в диалоге за вершиной повторения с некоторой вероятностью будет являться началом отвлечения.

  Здесь используется наблюдения, о том что после отвлечения на сторонний вопрос, оператор зачастую повторяет ту же или схожую фразу для возвращения в сценарий. Более того, эта идея используется в графе, который реализован для реального окружения. Там диалоговая система так же повторяет фразу, её сокращенную версию или её иную формулировку, которую произносил, перед тем, как перейти в отвлечение.

Поскольку подходы используют разные идеи их так же имеет смысл комбинировать и использовать данные полученные в обоих подходах.

2.3. Подход поиска отвлечений с большим количеством рёбер входящих в одну вершину

Первым этапом данного подхода является выбор вершин: выбираются вершины, у которых много входящих ребер. Для значений размеров кластеров в интервале от двенадцати до пятнадцати был выбран параметр три. При увеличении количества кластеров соответственно должен увеличиваться и порог.

В некоторых случаях отвлечения не ограничиваются одной вершиной. В связи с этим появляется необходимость определить, где заканчивается то или иное отвлечение, и где, следовательно, оператору необходимо вернуться в вершину с которой он в это отвлечение ушёл. Для этого предположим следующую гипотезу – если в вершину можно попасть только пройдя через отвлечение, то это значит, что фраза является частью соответствующего отвлечения.

Для того, чтобы найти соответствующую часть отвлечения после первой вершины, будет использован следующий алгоритм: запретим проход через вершину и рассмотрим все вершины, достижимые из вершины старта. Важно так же помнить о возможности того, что в кластерах могут быть допущены небольшие ошибки, поэтому если почти весь кластер недостижим после блокировки, то он так же входит в отвлечение.

Ниже представлен псевдокод функции поиска таких хвостов для отвлечений:

function DIGRESSION_FINDER($graph, node, dialogs$)
  for $dialog \in dialogs$ **do**
    $flag \leftarrow True$
    for $msg \in dialog$ **do**
      $msgs[msg.cluster] \leftarrow msgs[msg.cluster] + 1$
      if $(msg.cluster == node.cluster) \wedge flag$ **then**
        $achieved.add(msg.cluster)$
        $achieved\_msgs[msg.cluster].add(msg)$
      else
        $flag \leftarrow False$
      end if
    end for
  end for
  for $cluster \in graph.clusters$ **do**
    if $clusternotinachieved$ **then**
      $digression\_clusters.add(cluster)$
    else
      if $\frac{achieved\_msgs[cluster]}{msgs[cluster]} > 0.9$ **then**
        $digression\_clusters.add(cluster)$
      end if
    end if
  end for
  return $digression\_clusters$
end function

Во время тестирования на реальных диалогах человека с человеком была выявлена важная особенность: на работу алгоритма очень сильно влияет каче-

ство кластеризации. Поскольку фразы в голосовых диалогах не всегда верно переводятся в текст, сами фразы сравнительно короткие (а в случае людей-операторов ещё и очень вариативные), поэтому кластеризация оказалась очень некачественной.

2.4. Подход поиска отвлечений с поиском циклов

Предполагается, что оператор всегда действует согласно скрипту, а человек может отвлекаться, поэтому в подавляющем большинстве случаев после ответа на сторонний вопрос, оператор задаёт свой вопрос заново. В связи с данной спецификой, можно выделить несколько полезных особенностей:

— Подавляющее большинство поддиалогов отвлечений достаточно короткие;
— Вопрос повторяет оператор, поэтому циклы можно искать с повторением только операторской вершины;
— Возможно появление отвлечения внутри отвлечения, поэтому необходимо найти оба.

Для решения особенности первого пункта было добавлено ограничение на расстояние между повторяющимися кластерами в диалоге. Оно должно быть не слишком велико, поскольку отвлечения длины больше 2 операторских фраз встречаются в диалогах очень редко, при этом попадаются длинные циклы, которые не являются отвлечениями и при их удалении, удаляется большая содержательная часть диалога.

Проблема последнего пункта решается тем, что циклы будут удаляться по мере нахождения. Таким образом внутренний цикл будет удалён раньше внешнего.

В этом случае в качестве потенциальных отвлечений выберем вершины человеческих фраз, которые являются первыми после начала цикла. После перевода графа в продуктовый режим, соответствующие вершины станут рёбрами и таким образом в графе появятся рёбра-триггеры, которые будут перенаправлять ход диалога в вершину-отвлечение.

Ниже находится псевдокод для поиска одного цикла:

function REMOVE_CYCLES($dialog$)
    for $msg \in dialog$ **do**
        $msg\_num \leftarrow msg\_num + 1$

if $(last\_msg[msg.cluster] - msg\_num) <= 5$ **then**
$cycle\_end \leftarrow last\_msg[msg.cluster]$
$dialog.remove\_cluster(cycle\_start, msg\_num)$
end if
end for
end function

2.5. Выборка хороших кластеров

В качестве решения проблемы некачественной кластеризации было решено использовать данные из фраз человека. На предыдущем этапе фразы человека разделялись на группы по принципу совпадения вершины оператора до и после. Данные о соседних вершинах больше никак не использовались.

Было решено кластеризировать тексты пользователей. Но поскольку, как описывалось выше, кластеризация недостаточно хорошая, то необходимо было отсеять плохие кластеры во избежание каскадных ошибок.

Для этого использовалось попарное сравнение фраз внутри каждого кластера. Для этого сравнивались наборы слов внутри фразы с весами. Если точность превышала порог равный 0.5, то такая пара считалась хорошей.

Константа 0.5 была выведена эмпирически. Для большего значения в кластере начинало содержаться большое количество фраз разных по смыслу.

Проверка всех пар фраз имеет асимптотику по времени $O(n^2)$, где n – количество фраз. Так как на предыдущем этапе восстановления все операции имели ассимтотику не более чем $O(n \log n)$, то получилось так, что эта часть занимала значительно больше половины времени от всего восстановления графа.

Было решено для каждого кластера брать случайную выборку, равная утроенному размеру кластера, асимптотически это занимало уже $O(n)$. В силу достаточно больших размеров кластеров (размер их для основной массы данных составляет несколько сотен фраз), статистически, это показывало те же результаты, что и перебор всех пар.

Утверждение 1. При размере диалога от 40 фраз необходимое количество экспериментов для попадания в доверительный интервал $Q = 0.95$ и доверительной вероятности[1]. $\varepsilon = 0.1$ достаточно $3n$ экспериментов.

[1]Определение доверительного интервала можно изучить здесь[3]

Доказательство. Значение $p = 0{,}5$ – наихудшее, в том смысле, что для него вероятность порогового отклонения превысит выбранное значение ε, при наибольшем количестве экспериментов. Таким образом достаточно доказать утверждение для него. Для такого случая результат равен 96, причём вне зависимости от размера возможных исходов, т.е. вне зависимости от размера кластера. Эксперименты рассчитанные для разных значений приведены в учебном пособии[1].

Таким образом, для более маленьких кластеров можно рассчитать эти значения перебрав все пары полностью, а для больших кластеров, 3n экспериментов экспериментов будет достаточно.

Ниже приведён псевдокод для функции, считающей метрику для вершин с большим количеством фраз.

function IS_GOOD_CLUSTER($msgs$)
    $num\_of\_comparings \leftarrow 3 \cdot msgs.size()$
    for $(msg1, msg2) \in msgs.getPairs(num\_of\_comparings)$ **do**
        $good\_pairs+ = msg\_compare(msg1, msg2)$
    end for
    $cluster\_threshold \leftarrow 0.5$
    return $\frac{good\_pairs}{num\_of\_comparings} >= cluster\_threshold$
end function

Функция $msg\_compare(msg1, msg2)$ отвечает за сравнение двух сообщений.

2.6. Функция сравнения сообщений

Для алгоритма выше необходима функция сравнения двух сообщений. Для неё должны быть выполнены следующие условия:

— Функция должна быть бинарной. $True$ – в случае схожести сообщений, $False$ – иначе;
— Функция должна работать за $O(1)$, при условии, что длина сообщений так же принята за $O(1)$.

Второе условие необходимо, поскольку в противном случае при среднем размере групп сообщений от нескольких сотен разница во времени будет в разы увеличиваться.

В качестве алгоритма для сравнения сообщений был выбран следующий: берутся наборы слов в виде множеств и пересекаются с учётом весов слов. Это значение записываем в числитель. В качестве знаменателя берём объединение наборов слов с теми же коэффициентами слов и соответствующими количественными коэффициентами и записываем в знаменатель.

Полученная дробь должна превышать некоторый порог, который ищется вручную.

В качестве альтернативных вариантов функций могут быть использованы следующие:

— Сравнение фраз на основе семантической близости. Об этом алгоритме можно прочитать в следующей статье[2];
— Сравнение фраз с помощью эмбеддингов и на основе косинусного расстояния, которое было использовано в этой работе[5].

Все перечисленные и описанные функции подходят под перечень изначальных условий, поскольку функцию возвращающую нецелые значения в диапазоне от 0 до 1 можно превратить в ступенчатую, сделав её бинарной. Все они работают за асимптотически необходимое время.

2.7. Слияние кластеров операторских сообщений

Изначальный алгоритм, который создавал вершины операторов, предполагал точный выбор количества кластеров. Новое решение предполагает избыточное разбиение на кластеры. Поскольку количество кластеров на которые разделяются операторские сообщения можно контролировать вручную, это регулируется достаточно просто.

После такого разбиения выделяются кластеры, фразы в которых максимально похожи между собой, в таком разбиении будет меньше ошибок связанных с кардинальными смысловыми различиями между фразами из одного кластера.

Далее, после выбора хороших человеческих вершин, для каждой операторской вершины рассматриваются все соседние вершины фраз человека. Было опробовано несколько метрик для сравнения соседних вершин.

В первой версии рассматривались все соседские вершины. Они не делились на группы по положению до или после, и, соответственно, это показывало худшие результаты по сравнению со второй версией.

Во второй версии выбирались все вершины до, исходя из предположения, что они будут больше коррелировать с содержанием операторских вершин и не будет проблемы с тем, что не учтён порядок.

2.8. Оценка для сравнения графов

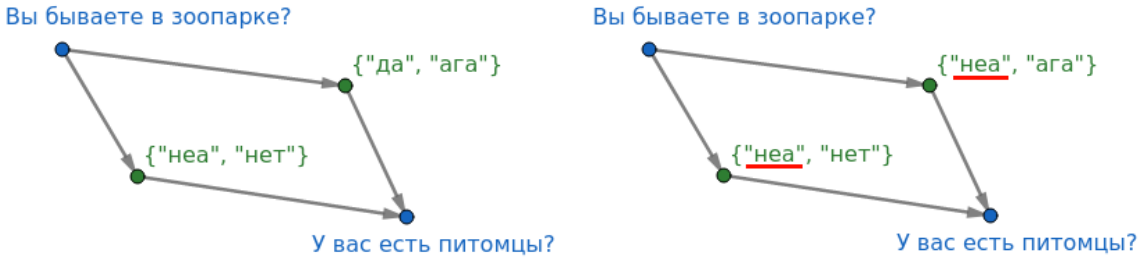
Следующий шаг заключался в том, что, для каждого графа сопоставляется контрольная модель, написанная вручную (если таковая существует). В ней каждому ребру из операторской вершины к операторской вершине сопоставляется группа фраз человека. Модели хранятся в формате `JSON`.

Для того, чтобы можно было сравнивать текущую модель и изначальную, необходимо было привести вторую модель к формату первой, поскольку для формата первой модели существует готовая платформа, на которой можно запустить модель.

Здесь стоит напомнить, что в восстанавливаемой модели кластеры человеческих фраз так же являлись вершинами. Таким образом для того, чтобы была возможность перевести их в рёбра и соответственно сопоставить изначальному графу, для каждой вершины с человеческими кластерами должны были быть верны следующие условия:

— У вершины должна быть ровно одна предыдущая, причём эта вершина должна быть вершиной операторской фразы;
— Последующая вершина должна быть так же операторской вершиной, причём она должна являться единственной последующей вершиной для данной;
— Ни одна из набора фраз данного кластера не должна совпадать с фразами из кластеров, в которые ведут рёбра из предыдущей операторской вершины.

Ниже на рисунке 3 представлено хорошее и плохое разбиения.



(a) Хорошее разбиение (b) Плохое разбиение

Рисунок 3 – Корректное и ошибочное разбиения

Реализация была сделана при помощи классов графа, новых вершин, рёбер. После проверки на то, что нет коллизий описанных выше можно провести инициализацию с помощью псевдокода приведенного ниже.

function CREATE_NODES($dialogs, digression$)
  for $dialog \in dialogs$ **do**
    for $msg \in dialog$ **do**
      if $msg.is\_incoming$ **then**
        if $msg.next$ **then**)
          $transactions[msg.prev.cluster].add(vertex[msg.next.cluster])$
        else
          $end\_vertexes.add(msg.prev.cluster)$
        end if
      else
        $vertex\_msgs[msg.cluster].add(msg)$
        $vertex\_clusters.add(msg.cluster)$
      end if
    end for
  end for
  for $cluster \in vertex\_clusters$ **do**
    $vertexes.add(Vertex(transactions[cluster], vertex\_msgs[cluster],$
    $cluster \in end\_vertexes, cluster \in digressions))$
  end for
  return $vertexes$
end function

Проверив, что все необходимые условия соблюдены, можно преобразовывать граф, выделяя отвлечения следующим образом:

— Каждая вершина новой модели сопоставлялась одной изначальной. Сопоставление проводилось путём нахождения косинусного расстояния между группой фраз, относящихся к восстановленной вершине, и фразами, заложенными в вершину изначального графа.;
— Вершины фраз человека были выделены в группы, каждая группа относилась к одному ребру и являлась будущем набором исходящих рёбер;
— Технические вершины[2] в изначальном графе игнорировались, но учитывалось их место в последовательности модели;
— Сравнивались наборы вершин отвлечения в изначальном графе и в полученном;
— При создании метрики для графа была использована статья [11].

[2]Технические вершины – вершины, находясь в которых автомат не произносит фраз, но в которых он, например, обрабатывает технические детали звонка или записывает ответы.

ГЛАВА 3. РЕЗУЛЬТАТЫ ЭКСПЕРИМЕНТОВ

3.1. Используемые данные

В качестве данных использовалось два принципиально разных типа наборов диалогов. Первые проводились уже с существующим скриптом. Второй тип, это данные диалогов человека с человеком. По каждому набору диалогов восстанавливался свой собственный граф.

1.Особенность первого набора заключается в том, что там легко кластеризовать фразы оператора, так как они произносятся всегда одинаково, за исключением вариаций для коротких вариантов.

В том числе, в таких графах прописаны по-другому сформулированные фразы для случаев повтора. Эти фразы повторов имеют других по смыслу соседей. Поскольку при избыточном разделении они и оригинальные фразы разбивались на разные кластеры, появлялась необходимость в их объединении для корректной работы алгоритма циклов. Но объединение по соседям требовало корректировки, поскольку пересечение по соседним вершинам было недостаточным для объединения. Таким образом, необходимо было искать баланс между слишком маленьким количеством кластеров и возможностью объединить избыточные разделения.

Важным вариантом использования таких данных являлась возможность перевести полученный граф в тот же формат и сравнить полученный результат с оригиналом.

2.Особенности второго соответственно следующие: во-первых там говорят разные операторы и у них разный стиль подачи одних и тех же данных. Во вторых диалоги более сложные и зачастую отходят от скрипта. В третьих люди решают более сложные вопросы и умеют давать ответы на не предусмотренные скриптом вопросы. На этот вариант данных стоит ориентироваться, но в связи с отсутствием оригинального скрипта в удобном формате напрямую сравнить полученный результат представляется возможным только вручную.

В обоих случаях есть сложности с переводом речи людей в текст, поэтому иногда даже рассматривая текст диалога вручную нельзя понять что человек имел в виду.

Для всех датасетов первого типа, существуют файлы, содержащие в себе оригинальный граф, который принимается за верный. Все данные сериализованы в формате `JSON`.

На Рисунке 4 представлен восстановленный граф, получившийся после одного из запусков. С этой моделью происходила непосредственная работа.

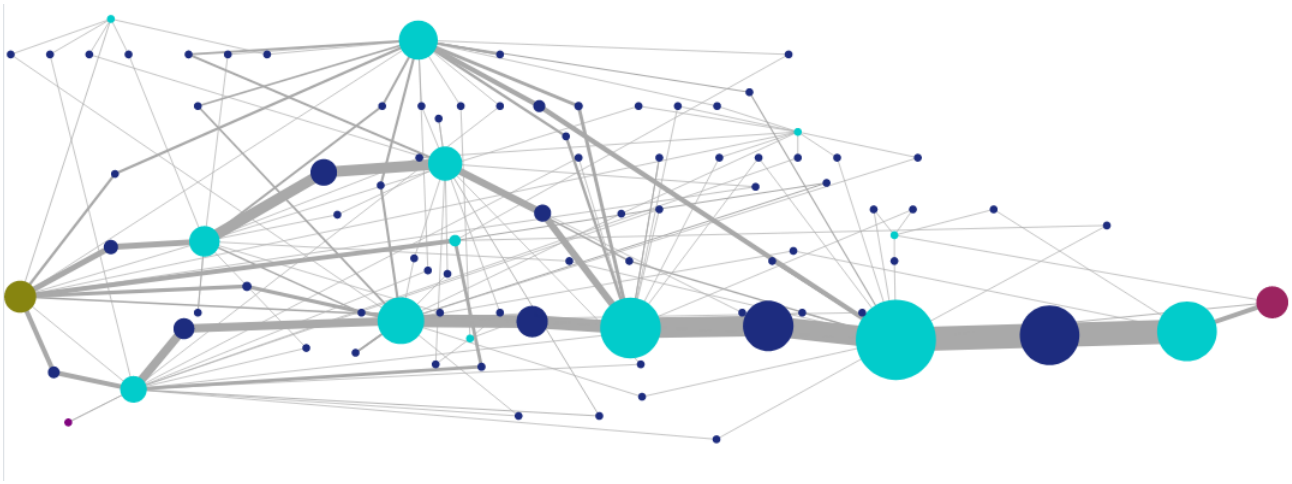


Рисунок 4 – Восстановленный граф

В этой модели голубым цветом обозначены операторские вершины, синим соответственно человеческие. Кроме того, для удобства анализа выделены две фиктивные вершины: стартовая и вершина окончания диалогов. В дальнейшем, для приведённых данных они в расчёт не брались.

На Рисунке 5 представлена модель адаптированная для текста работы. На ней будут показаны результаты работы алгоритмов.

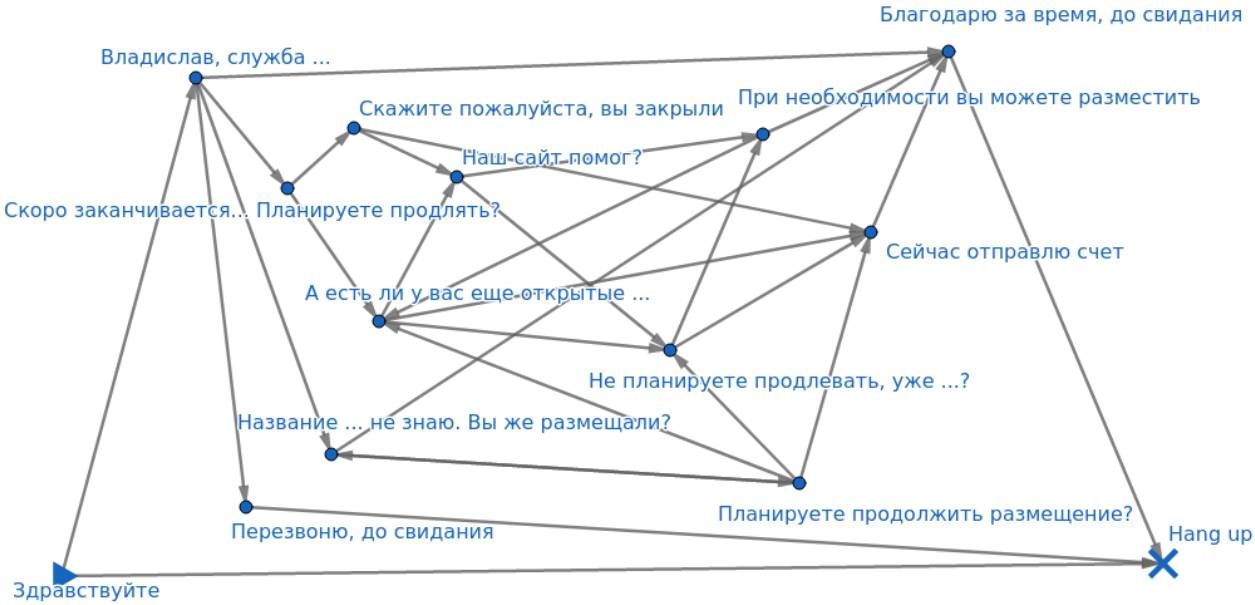


Рисунок 5 – Адаптированный граф

NB: В данном примере вершины человеческих кластеров и вершины операторских кластеров с фразами «Алло», «Я говорю» и «Повторите пожа-

луйста» были намеренно убраны, а так же склеены вершины разных приветствий, поскольку их наличие делало восприятие графа более затруднительным. Про эти вершины и особенности будут написаны отдельные заметки везде, где граф будет использоваться в качестве примера.

По тем же причинам были удалены большинство рёбер ведущих в конец диалога – человек может бросить трубку в любой момент, на алгоритмы эти рёбра не влияют.

Изначально при создании графа вершины человеческих фраз просто группируются по тому, какие 2 кластера находятся до и после, поэтому становится возможным заменить их на рёбра, не потеряв ценной информации.

3.2. Результаты улучшения кластеризации

Алгоритм вносит небольшие изменения и способен объединять некоторые одинаковые кластеры.

Основная сложность состояла в анализе, так как для данных из разговоров с искусственным интеллектом кластеризация фраз оператора изначально была очень хорошей в силу того, что фразы повторялись. Для случаев разговоров человека с человеком анализ результатов приходилось проводить вручную в силу отсутствия разметки данных.

Были проведены исследования по влиянию параметров алгоритма на результат. В результате чего были выявлены следующие закономерности:

— Пороговое значение схожести сообщений для каждого кластера должно варьироваться от 0.2 до 0.5. В этом промежутке в качестве пар схожих сообщений выбираются схожие пары сообщений. Для меньшего значения соответственно практически полностью одинаковые. Для большего у них появляется вариативность;
— Пороговое значение количества хороших пар в кластере должно варьироваться так от 0.25 до 0.5. Для больших значений в одном кластере начинают появляться наборы фраз несущие разный смысл.

Ниже, на Рисунке 6 представлены примеры различных значений промежутков порога для количества хороших пар в кластере:

нет
нет нет
нет
нет александр александрович

(a) 0.3

до свидания
спасибо до свидания
спасибо всего доброго до свидания
всё спасибо вам за напоминание всего доброго

(b) 0.5

але
на какую тему подскажите
даша
сколько минут
интернет
я

(c) 0.6

Рисунок 6 – Различные значения пороговой функции количества хороших пар

Для графа с Рисунка 5, в качестве примера работы данного алгоритма, рассмотрим пример объединения 2 вершин приветствия[1]. Набор входящих и исходящих рёбер для вершин приветствия можно увидеть на Рисунке 7:

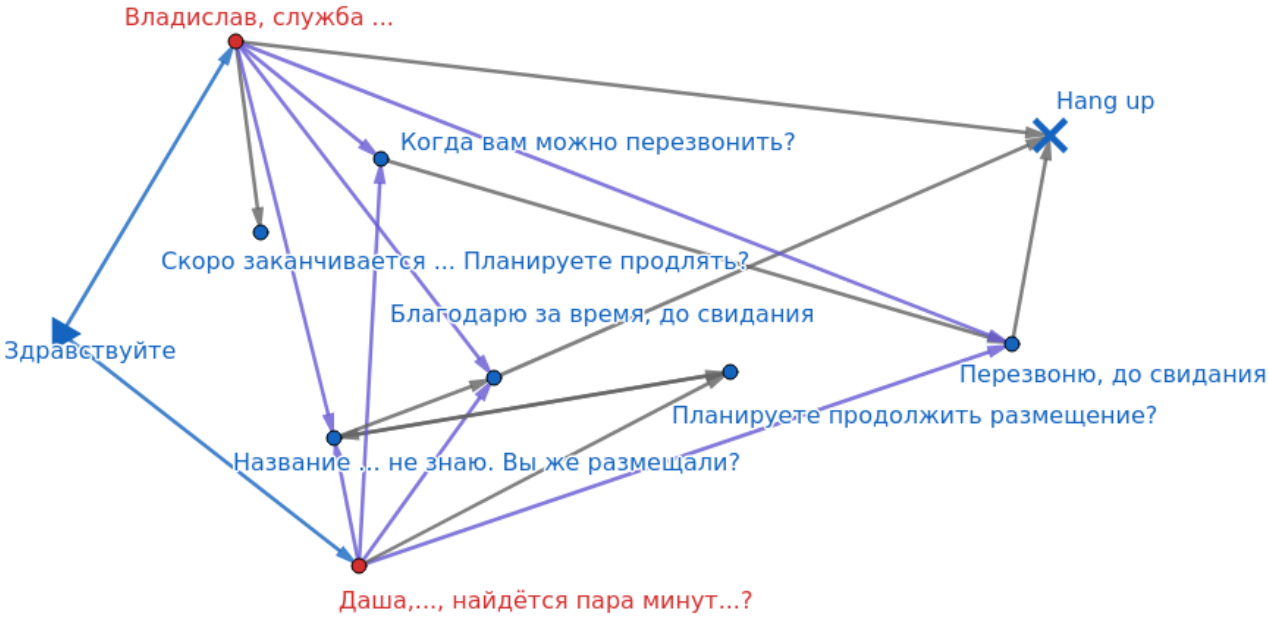


Рисунок 7 – Восстановленный граф

Среди данных вершин были найдены две вершины с приветствием из разных запусков. Важно отметить, что при уменьшении количества кластеров, на которые будут поделены вершины будут сливаться такие пары как «Скоро заканчивается ..., планируете продлевать?» и «Не планируете продлевать, уже ...», в связи с тем, что семантика этих фраз похожа, а приветствия сформулированы по-разному. Таким образом уменьшение количества кластеров проблему не решает.

[1] на Рисунке 5 для удобства, они были уже объединены

Пары соответствующих рёбер, ведущих к одинаковым вершинам (рёбра помечены фиолетовым и синим) имеют схожие фразы, которые объединяются при рекластеризации (кластеры : («слушаю, да»), («нет вакансий»), («здравствуйте») и т.д.). Таким образом у вершин появляются множества кластеров до и после с большим пересечением и они объединяются.

Подход с пересечением групп операторских кластеров до и после, не может быть использован, в силу того, что в этом случае фразы «Алло» и «Повторите пожалуйста» будут объединяться в силу того, что в них идут рёбра и из них идут рёбра практически во все вершины.

3.3. Результаты работы алгоритма поиска отвлечений по рёбрам

Для данного алгоритма критично качество кластеризации и наличие небольшого количества выбросов.

В случае с данными из реальных диалогов зачастую не было видно правильно выделенных отвлечений из-за смешения кластеров и операторских и человеческих вершин.

Для данных из диалогов с существующей диалоговой системой алгоритм находил все отвлечения и так же отмечал некоторое количество дополнительных вершины, которые отвлечениями не являются. Основная причина столь большого различия в том, что операторские фразы во втором случае кластеризуются практически идеально.

На данном этапе этот алгоритм корректно использовать в качестве отсечения вершин, которые не подходят в качестве отвлечений, но были ошибочно выбраны другими алгоритмами. Также было сделано следующее наблюдение: алгоритм хорошо подсвечивает последовательности фраз, которые часто произносятся друг за другом. Например, после фразы «Когда вам перезвонить» – фраза «Перезвоню, до свидания» встречается **74.8%** случаев[2].

3.4. Результаты алгоритма поиск циклов

После того, как циклы найдены, их необходимо удалить. Таким образом основной граф уменьшается в размерах и его становится легче анализировать. Из особенностей реализации важно отметить, что удаляются не все циклы, а только размера не более чем 5. Рассмотрим пример следующего диалога:

Оператор: Здравствуйте

[2]Среди диалогов, которые не обрываются после первой фразы.

Человек: Ало

*Оператор: Меня зовут Даша, вы ****[3]*?*

*Человек: *шуршание**[4]

Оператор: Извините, я вас не расслышал, повторите пожалуйста

Человек: Я говорю давайте задавайте

*Оператор: Вы сейчас ****?*

Человек: сейчас нет

*Оператор: Вы нашли ****?*

Человек: да

*Оператор: Убрать **** в архив?*

*Человек: *неразборчивый текст**

Оператор: Извините, я вас не расслышал, повторите пожалуйста

*Человек: не надо пока убирать ****

Оператор: Извините за беспокойство, всего доброго, до свидания

Если удалить все фразы между двумя просьбами, то состояния которые не являются отвлечениями, будут утеряны. В данном случае они составляют основную часть диалога.

Ниже, на Рисунке 8 приведен график, в котором сравниваются количества вершин, до и после применения алгоритма удаления циклов. Для каждого графа так же указано, датасет какого размера был использован при создании:

---

[3]Здесь и далее *** будут использоваться для обезличивания частей диалога, которые не содержат важных для алгоритма данных, но несут в себе персональную информацию или информацию о компаниях.

[4]Здесь и далее *описание* для пропуска не содержательных частей с коротким описанием.

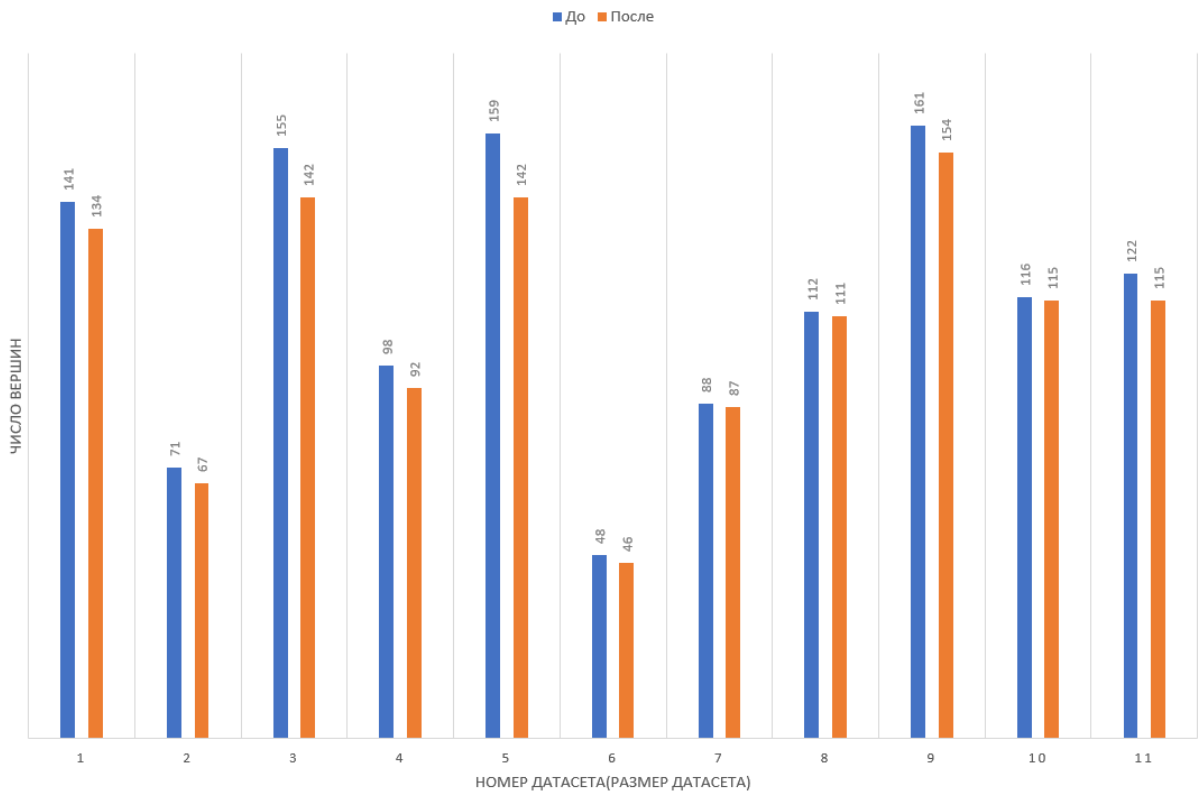


Рисунок 8 – Изменение количества вершин

Ниже, на Рисунке 9 приведен график, в котором сравниваются количества рёбер, до и после применения алгоритма удаления циклов.

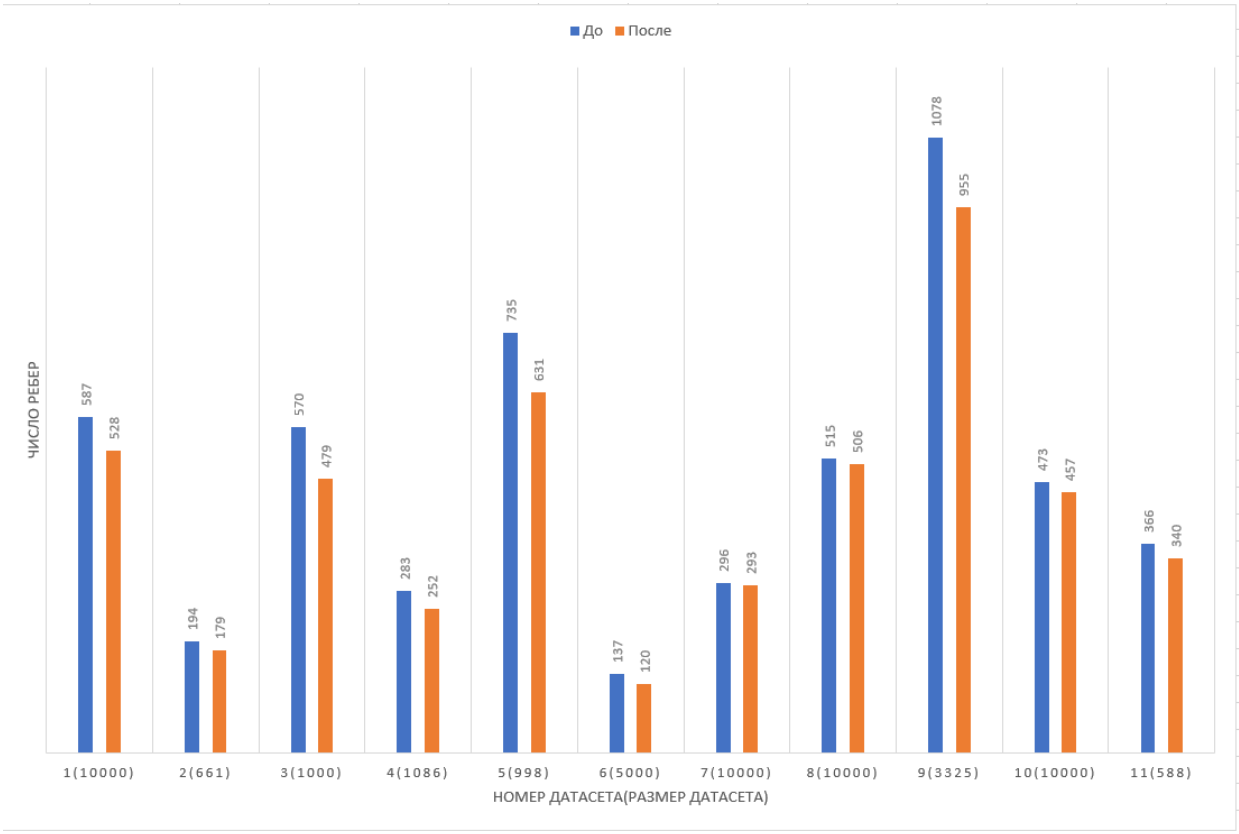


Рисунок 9 – Изменение количества вершин

Важным будет отметить, что и на обоих графиках отображены только те вершины и рёбра, которые удалились полностью, т.е. гарантированно попали в группы отвлечений и триггеров. Также отвлечениями считались вершины и рёбра, в которых большинство фраз и переходов попали в циклы.

Среди этих датасетов есть 2 принципиально разных типа:

Первый тип – **оператор + человек**, эти датасеты были предоставили компании заказчиками, и в них человек придерживается некоторого скрипта, но заметно от него отклоняется, поэтому при меньших размерах порядок вершин в них такой же. К этому типу относятся соответственно датасеты 2, 4, 5, 9, 11.

Второй тип – **искусственный интеллект + человек**, эти датасеты были собраны соответственно компанией при звонках со скриптами написанными вручную. К этому типу соответственно относятся 1, 3, 6, 7, 8, 10.

В обоих типах при увеличении размера датасета увеличивается количество вершин. Важно отметить, что большинство из вершин, это ответы человека, количество же операторских вершин варьируется от 12 до 25. Для случая диалогов человека с человеком размеры графа могут быть изначально больше, это объясняется тем, что в нём большее разнообразие фраз.

Вырезанные ребра и вершины соответственно отделяются и становится проще анализировать граф.

Если рассмотреть какие вершины из операторских попадают в циклы, то хорошо видно тенденцию того, что есть несколько вершин у которых большое количество ответов попадает в циклы. Некоторые такие вершины, как можно заметить исчезают вообще. Для остальных можно подобрать пороговое значение при котором можно будет считать, что вершина является триггером отвлечения.

Для примера графа указанного выше отметятся следующие вершины, содержащие большое количество фраз в циклах (см. Рисунок 10):

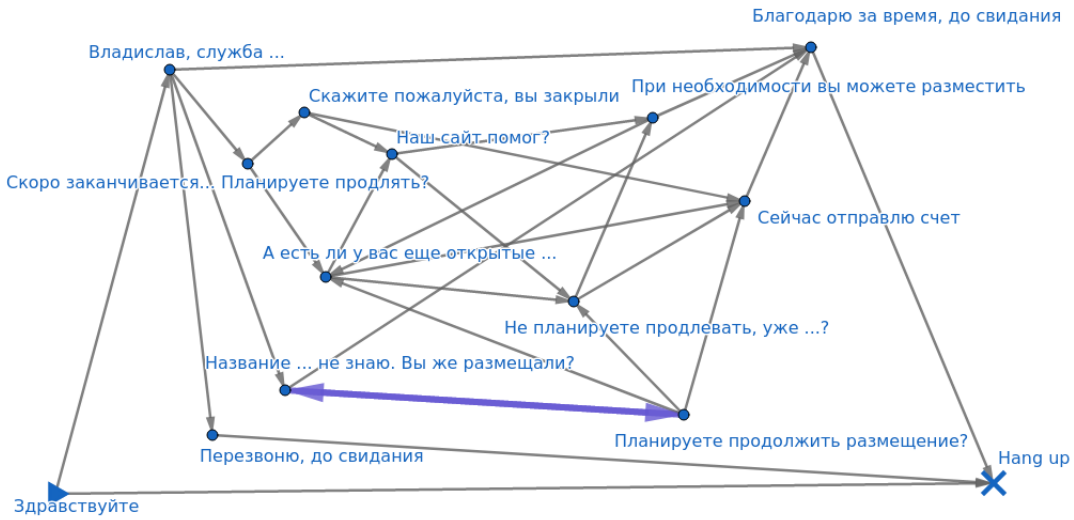


Рисунок 10 – Отмеченный цикл

Интересным будет заметить, что из двух вершин именно «Название ... не знаю, вы же размещали.» попадёт в отвлечения, т.к. внутри цикла находятся именно её сообщения, а сообщения «Планируете продолжить размещение?» находятся на границах цикла.

Более того, самый большой процент (84.7%) попавших в циклы вершин будет иметь фраза «Я говорю», которая была исключена, т.к. сильно усложняла граф. Она является примером **общего** для всех графов отвлечения (ответом на вопрос «повторите» и т.п.). Таким образом, поведение алгоритма выделившего её является корректным.

3.5. Программная реализация

Предложенные алгоритмы были реализованы на языке `Python` с использованием интегрированной среды разработки `Pycharm`. Кроме того ис-

пользовался текстовый редактор Visual Studio Code с внутренним плагином для визуализации графов описанных в формате JSON. Разработка велась в компании Dasha.AI.

Компания использует приватный Git-репозиторий, находящийся на хостинге Gitlab. Для удобства локальной разработки приложение запускалось в Docker-контейнере.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы была рассмотрена задача выделения отвлечений в графовой модели для голосовой диалоговой системы. Для её решения были разработаны и написаны алгоритмы основанные на особенностях разговоров оператора с человеком. Были учтены особенности существующей модели графовых диалогов и данные, полученные при её использовании.

Были написаны алгоритмы выделения циклов и поиска отвлечений по рёбрам. Эти алгоритмы позволяют выделить значительное количество отвлечений и уменьшают размер основного графа. Кроме того в ходе разработки стало ясно, что алгоритмы очень чувствительны к кластеризации, и возникла необходимость улучшить её качество. Для чего был разработан алгоритм на основе данных о соседних фразах.

Поскольку работа является частью большего проекта. То интеграция в реальное окружение запланирована по завершению всех частей проекта. Разработка должна будет автоматизировать часть рабочих процессов.

Одно из возможных направлений развития работы, использовать информацию о пользователе в диалоге, сделав его таким образом более естественным. Так же существует возможность выделить некоторые общие отвлечения для всех диалогов, получая таким образом возможность предсказывать отвлечения в виде подсказок.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Мухин О.* Моделирование систем: учебник [Электронный ресурс]. — 2014. — URL: http://stratum.ac.ru/education/textbooks/modelir/lection34.html (дата обр. 30.04.2020).

2 *Нгок Н. Б.*, *Тузовский А. Ф.* Классификация текстов на основе оценки семантической близости терминов // Izvestiya Tomskogo Politekhnicheskogo Universiteta Inziniring Georesursov. — 2012. — Т. 320, № 5. — С. 43–48.

3 *Чернова Н.* Теория вероятностей: Учеб. пособие/Новосиб. гос. ун-т // Новосибирск. 2007, 160 с. — 2007.

4 Dialogue-based Management of user Feedback in an Autonomous Preference Learning System. / J. Lucas-Cuesta [et al.] //. Vol. 1. — 09/2010. — P. 330–336.

5 *Dönmez İ.*, *Pashaei E.*, *Pashaei E.* Word Vector Space for Text Classification and Prediction According to Author. —.

6 *Doshi F.*, *Roy N.* Efficient model learning for dialog management // 2007 2nd ACM/IEEE International Conference on Human-Robot Interaction (HRI). — 2007. — P. 65–72.

7 *Jokinen K.*, *McTear M.* Spoken Dialogue Systems. — Morgan & Claypool Publishers, 2010. — (Synthesis lectures on human language technologies). — ISBN 9781598295993. — URL: https://books.google.ru/books?id=ualwulnD020C.

8 *Li H.* Text Clustering // Encyclopedia of Database Systems / ed. by L. LIU, M. T. ÖZSU. — Boston, MA : Springer US, 2009. — P. 3044–3046. — ISBN 978-0-387-39940-9. — DOI: 10.1007/978-0-387-39940-9_415. — URL: https://doi.org/10.1007/978-0-387-39940-9_415.

9 *Wikipedia contributors*. Dialog manager — Wikipedia, The Free Encyclopedia. — 2020. — URL: https://en.wikipedia.org/w/index.php?title=Dialog_manager&oldid=941891414 (visited on 04/18/2020).

10 *Williams J. D.*, *Young S.* Partially observable Markov decision processes for spoken dialog systems // Computer Speech & Language. — 2007. — Vol. 21, no. 2. — P. 393–422.

11 *Wills P.*, *Meyer F. G.* Metrics for graph comparison: A practitioner's guide // Plos one. — 2020. — Vol. 15, no. 2. — e0228728.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД НА ЯЗЫКЕ PYTHON

```python
# Copyright (C) 2020 Dasha.AI Inc.
import json
import logging
import os

import numpy as np
import requests
from flask import Flask, Response, jsonify, request
from flask_caching import Cache
from sklearn.cluster import AgglomerativeClustering, KMeans

from reconstruction import algorithm_descriptions, reconstruct_graph
from reconstruction.data_structure import DatasetStorage
from reconstruction.data_structure.graph import save_graph
from reconstruction.data_structure.message import messages_to_json
from reconstruction.embedding import EmbeddingStorage

@app.route('/graph', methods=['POST'], strict_slashes=False)
@cache.cached(key_prefix=make_cache_key)
def build_graph():
    dataset_name, dialogs, alg_name, hparams = parse_request()
    graph = reconstruct_graph(dialogs, alg_name, hparams, embedding_storage)
    save_graph(graph, dataset_name)
    return jsonify(graph)
```

```python
# Copyright (C) 2020 Dasha.AI Inc.
from reconstruction.reconstruct_graph import reconstruct_graph,
↪  algorithm_descriptions
```

```python
# Copyright (C) 2020 Dasha.AI Inc.
from sklearn.cluster import KMeans

import reconstruction.algorithm as alg
from reconstruction.embedding.mean_vectorizer import MeanEmbeddingVectorizer

algorithms = {
    "cluster_all_messages": {
    },
    "cluser_incoming_outgoing": {
    },
    "cluster_incoming_outgoing_on_each_step": {
    },

    "cluster_outgoing_and_incoming_after": {
    },
```

```
    "cluster_prev_and_next": {
    },
    "cluster_prev_and_next_recluster_incoming": {
        "reconstruction": alg.cluster_prev_and_next_recluster_incoming,
        "clustering": {
            "emb_name": "tenth.norm-sz500-w7-cb0-it5-min5.w2v",
            "vectorizer": MeanEmbeddingVectorizer,
            "alg": KMeans
        }
    }
}
```

```
# Copyright (C) 2020 Dasha.AI Inc.
from collections import Counter, defaultdict

import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

from reconstruction.data_structure.graph import Node
from reconstruction.converter.converter import Graph
from reconstruction.digressions.digressions import cycle_finder


def calc_centroid(messages):


def create_nodes(messages, cluster_labels, remove_msgs=[]):
        #secsitive code - create nodes using information about msgs from cycle,
        ↪  remove msgs keep in dialogs for analyse


def add_start_node(graph, dialogs, skip):
    graph["nodes"].append({
        "id": -1,
        "group": -1,
        "centroid": 0,
        "texts": [],
        "seq_num": -1,
        "seq_num_std": 0,
        "incoming": False,
        "size_coef": 0.4,
        "size": len(dialogs),
        "x": 0,
        "y": 0
    })
    start_cluster = -1
    graph["start"] = start_cluster
    weights = defaultdict(int)
    for dialog in dialogs:
```

```
            if (len(dialog) > 0):
                msg = dialog.log[0]
                cluster = msg.cluster
                if cluster and cluster not in skip:
                    weights[cluster] += 1
        for cluster in weights:
            graph["links"].append({
                "source": int(start_cluster),
                "target": int(cluster),
                "weight": int(weights[cluster])
            })


def add_end_node(graph, dialogs, skip):


def nodes_to_graph(nodes, dialogs, node_size_limit=0):
    #secsitive code - create nodes, count size, linkes with weights, work with
    ↪ digressions


def num_of_vertexes(msgs, remove_msgs=[]):
    vertexs = defaultdict(set)
    none_counter = 0
    for msg in msgs:
        if msg in remove_msgs:
            continue
        if msg.next_msg is not None and msg.next_msg not in remove_msgs:
            vertexs[msg.cluster].add(msg.next_msg.cluster)
    edges_num = 0
    for _, set_list in vertexs.items():
        edges_num += len(set_list)
    print("vertexs:", len(vertexs), " edges: ",
          edges_num, " nones: ", none_counter)


def create_graph(messages, cluster_labels, dialogs, node_size_limit=0,
↪ call_remove_cycles=False):
    #sensitive code - create nodes, graph, call cycles finder, count statistics
```

```
# Copyright (C) 2020 Dasha.AI Inc.
from reconstruction.config import algorithms
from reconstruction.clustering import MessageClusterizer
from reconstruction.create_graph import create_graph
from reconstruction.coords import graphviz_coords
import logging


def get_vectorizer(embedding_storage, clustering_config):
```

```

def get_clustertering_alg(clustering_config, **kwargs):


def preprocess_dialogs(vectorizer, dialogs):


def reconstruct_graph(dialogs, alg_name, hparams, embedding_storage):
    alg_config = algorithms[alg_name]
    reconstruction_alg = alg_config["reconstruction"]

    clustertering_alg = get_clustertering_alg(alg_config["clustering"],
                                              **{"n_clusters":
                                               ↪  int(hparams["n_clusters"])})

    vectorizer = get_vectorizer(embedding_storage, alg_config["clustering"])
    preprocess_dialogs(vectorizer, dialogs)

    clustered_messages, cluster_labels = reconstruction_alg(
        dialogs, clustertering_alg, **hparams)

    min_node_size = int(hparams["min_node_size"])

    remove_cycles = hparams["remove_cycles"]
    graph = create_graph(clustered_messages, cluster_labels, dialogs,
                         node_size_limit=min_node_size,
                         ↪  call_remove_cycles=remove_cycles)

    digressions_by_different_income(graph, dialogs)

    graphviz_coords(graph)
    return graph
```

```
# Copyright (C) 2020 Dasha.AI Inc.
from reconstruction.algorithm.cluser_incoming_outgoing import \
    cluser_incoming_outgoing
from reconstruction.algorithm.cluster_all_messages import cluster_all_messages
from reconstruction.algorithm.cluster_incoming_outgoing_on_each_step import \
    cluster_incoming_outgoing_on_each_step
from reconstruction.algorithm.cluster_outgoing_and_incoming_after import \
    cluster_outgoing_and_incoming_after
from reconstruction.algorithm.cluster_prev_and_next import \
    cluster_prev_and_next
from reconstruction.algorithm.cluster_prev_and_next_recluster_incoming import \
    cluster_prev_and_next_recluster_incoming
```

```python
# Copyright (C) 2020 Dasha.AI Inc.
import itertools
import logging
import multiprocessing
import random
import sys
import time
from collections import Counter, defaultdict
from functools import partial
from typing import Dict, List

import numpy as np
from sklearn.cluster import KMeans

from reconstruction.data_structure import Dialog
from reconstruction.data_structure.dialog import get_dialog_messages
from reconstruction.clustering import MessageClusterizer
from reconstruction.algorithm import cluster_prev_and_next
from reconstruction.algorithm.utils import split_incoming_outgoing


def vocab_distribution(words):
    total = len(words)
    vocab_dist = dict(Counter(words))
    for word in vocab_dist.keys():
        vocab_dist[word] /= total
    return vocab_dist


def distribution_text_similarity(msg1, msg2, vocab_dist):
    words1 = msg1.text.split()
    words2 = msg2.text.split()
    counter1 = dict(Counter(words1))
    counter2 = dict(Counter(words2))
    unique_words1 = set(words1)
    unique_words2 = set(words2)

    cross = sum(
        min(counter1[word], counter2[word]) * vocab_dist[word]
        for word in unique_words1 & unique_words2
    )

    union = sum(
        max(counter1.get(word, 0), counter2.get(
            word, 0)) * vocab_dist[word]
        for word in unique_words1 | unique_words2
    )

    similarity = cross / union
```

```
    return similarity


def remove_unvalid_clusters(msgs, clusters, similarity_threshold=0.4,
↪   cluster_threshold=0.4):
    start = time.time()
    words = [w for t in msgs for w in t.text.split()]
    vocab_dist = vocab_distribution(words)
    print("vocab_distribution:", time.time() - start, file=sys.stderr)

    start = time.time()
    cluster_messages = defaultdict(list)
    for cl_n, msg in zip(clusters, msgs):
        cluster_messages[cl_n].append(msg)

    print("cluster_messages:", time.time() - start, file=sys.stderr)

    start = time.time()
    msgs_in_cluster = dict(Counter(clusters))
    good_connects = defaultdict(int)
    msgs_in_stat = defaultdict(int)
    pool = multiprocessing.Pool(5)

    for cl_n, msg_l in cluster_messages.items():
        if cl_n in msgs_in_cluster:
            for i, msg in enumerate(msg_l):
                for j in range(i):
                    # check ~3 * cl_num random pairs
                    if (random.randint(0, msgs_in_cluster[cl_n] // 3) == 0):
                        msgs_in_stat[cl_n] += 1
                        sim = distribution_text_similarity(
                            msg, msg_l[j], vocab_dist)
                        if sim > similarity_threshold:
                            good_connects[cl_n] += 1
    bad_clusters = {
        cl_n for cl_n, correct in good_connects.items()
        if (correct / msgs_in_stat[cl_n] < cluster_threshold)
    }

    print(msgs_in_cluster)
    print("msg list size: ", len(cluster_messages),
          " keys: ", cluster_messages.keys)
    print([
        cl_n for cl_n, correct in good_connects.items()
        if (correct / msgs_in_stat[cl_n] > cluster_threshold)
    ], " good one")
    print([
        cl_n for cl_n, correct in good_connects.items()
        if (correct / msgs_in_stat[cl_n] < cluster_threshold)
    ], "bad one")
```

```
99
100     start = time.time()
101     new_msgs, new_clusters = [], []
102     for msg, cl_n in zip(msgs, clusters):
103         if cl_n not in bad_clusters:
104             new_msgs.append(msg)
105             new_clusters.append(cl_n)
106     print("filter:", time.time() - start, file=sys.stderr)
107
108     return new_msgs, new_clusters
109
110
111 def recluster_incoming(clustered_messages, cluster_labels, dialogs, cluster,
    ↪ **hparams):
112     """Делит входящие сообщения иначе, с учётом того, как они кластеризовались
        ↪ независимо от сообщений оператора
113     1. Кластеризуем входящие сообщения независимо
114     2. Разделяем ноды из входящих, если там много сообщений каких-то типов
115     3. Объединяем ноды с полностью одинаковыми кластерами
116     """
117     print(hparams, file=sys.stderr)
118
119     proportion_threshold = float(hparams.get("proportion_threshold", 0.25))
120     cluster_threshold = float(hparams.get("cluster_threshold", 0.5))
121     similarity_threshold = float(hparams.get("similarity_threshold", 0.5))
122
123     start = time.time()
124
125     messages = get_dialog_messages(dialogs)
126     incoming, _ = split_incoming_outgoing(messages)
127     print(" get_dialog_messages split_incoming_outgoing:",
128           time.time() - start, file=sys.stderr)
129     start = time.time()
130     incoming_cluster_labels = cluster.fit_predict(incoming)
131     print("fit_predict:", time.time() - start, file=sys.stderr)
132
133     start = time.time()
134     incoming, incoming_cluster_labels = remove_unvalid_clusters(
135         incoming, incoming_cluster_labels,
136         similarity_threshold=similarity_threshold,
137         cluster_threshold=cluster_threshold)
138
139     print("remove_unvalid_clusters:", time.time() - start, file=sys.stderr)
140
141     start = time.time()
142     pair_clusters = defaultdict(int)  # new clusters, incoming only
143     for msg, cluster in zip(incoming, incoming_cluster_labels):
144         if msg in clustered_messages:  # cluster_messages doesn't include
            ↪ starts of dialogs
145             pair_clusters[msg] = cluster
```

```
146
147     node_clusters = defaultdict(lambda: defaultdict(list))
148     new_clustering = defaultdict(int)
149
150     for msg, cluster in zip(clustered_messages, cluster_labels):
151         if msg in pair_clusters:
152             node_clusters[cluster][pair_clusters[msg]].append(msg)
153
154     new_cluster_num = cluster_labels.max()
155     alloc_clusters = defaultdict()
156     changing_msgs = 0
157
158     for _, clusters in node_clusters.items():
159         node_size = sum(map(len, clusters.values()))
160         small_part = []
161         for income_cluster_n, msgs in clusters.items():
162             cluster_proportion = (len(msgs) / node_size)
163             if (cluster_proportion > proportion_threshold):
164                 if income_cluster_n not in alloc_clusters:
165                     new_cluster_num += 1
166                     alloc_clusters[income_cluster_n] = new_cluster_num
167
168                 cluster_num = alloc_clusters[income_cluster_n]
169                 for msg in msgs:
170                     new_clustering[msg] = cluster_num  # check
171                     changing_msgs += 1
172             else:
173                 small_part.append(income_cluster_n)
174
175         if len(small_part) == 1:
176             alone_cluster = small_part[0]
177             if alone_cluster not in alloc_clusters:
178                 new_cluster_num += 1
179                 alloc_clusters[alone_cluster] = new_cluster_num
180             cluster_num = alloc_clusters[alone_cluster]
181             for msg in clusters[alone_cluster]:
182                 new_clustering[msg] = cluster_num  # check
183                 changing_msgs += 1
184     print("split clusters:", time.time() - start, file=sys.stderr)
185
186     start = time.time()
187     list_of_changes = []
188     for msg in new_clustering:
189         i = clustered_messages.index(msg)
190         clustered_messages[i].cluster = new_clustering[msg]
191         cluster_labels[i] = new_clustering[msg]
192         list_of_changes.append(i)
193     print("rename clusters:", time.time() - start, file=sys.stderr)
194     return clustered_messages, cluster_labels
195
```

```
196
197 def cluster_prev_and_next_recluster_incoming(dialogs: List[Dialog], cluster:
    ↪ MessageClusterizer, **hparams):
198     """Кластер входящих определяется как единый кластер, если следующий
        ↪ исходящий кластер одинаковый для всех сообщений
199
200     1. Строим кластеры для всех исходящий сообщений
201     2. Для каждого кластера находим все последующие входящие сообщения до
    ↪ следующего исходящего
202     3. Эти входящие сообщения разделяются на кластеры в зависимости от того
    ↪ какой исходящий кластер следующий
203     4. Номер кластера задается на основе следующего исходящего кластера.
204
205     Таким образом переход от кластера входящих сообщений к кластеру исходящих
    ↪ становится однозначным.
206
207     Входящие сообщения разделяются, с учётом того, как они кластеризовались
    ↪ независимо от сообщений оператора
208         1. Кластеризуем входящие сообщения независимо
209         2. Разделяем ноды из входящих, если там много сообщений каких-то типов
210         3. Объединяем ноды с полностью одинаковыми кластерами
211
212     Arguments:
213         dialogs {List[Dialog]} -- Список диалогов
214         cluster {MessageClusterizer} -- Алгоритм кластеризации исходящих
    ↪ сообщений.
215
216     Returns:
217         messages {List[Message]} -- Список сообщений
218         cluster_labels {np.array} -- Список меток кластера для сообщений
219     """
220
221     start = time.time()
222     clustered_messages, cluster_labels = cluster_prev_and_next(
223         dialogs, cluster)
224     print("Init clustering:", time.time() - start, file=sys.stderr)
225     start = time.time()
226     clustered_messages, cluster_labels = recluster_incoming(
227         clustered_messages, cluster_labels, dialogs, cluster, **hparams)
228     print("Overall reclustering:", time.time() - start, file=sys.stderr)
229     return clustered_messages, cluster_labels
```

```
1 # Copyright (C) 2020 Dasha.AI Inc.
2 import json
3 from collections import defaultdict
4
5 class Graph:
6     def __init__(self, messages, digressions=[]):
7         msgs_dict = defaultdict(set)
```

```
8           self.vertexes = []
9           for msg in messages:
10              msgs_dict[msg.cluster].add(msg)
11          for cluster, msgs in msgs_dict.items():
12              tr_dict = defaultdict(set)
13              transactions = []
14              for msg in msgs:
15                  if msg.next_msg:
16                      tr_dict[msg.next_msg.cluster].add(msg)
17              for next_cluster, msgs_tr in tr_dict.items():
18                  transactions.append(Transaction(msgs_tr, int(cluster),
                    ↪  int(next_cluster)))
19              self.vertexes.append(Vertex(int(cluster), transactions, cluster in
                ↪  digressions))
20          self.digressions = digressions
21
22      def __str__(self):
23          str_vert = []
24          for vertex in self.vertexes:
25              str_vert.append(str(vertex))
26          return '{"nodes": {' + ", ".join(str_vert) + '},' +
            ↪  str(self.get_digr()) + '}'
27
28      def get_digr(self):
29          digr_tr = None
30          return '"digressionNodes": '  + json.dumps(
31          {
32              "dig:root": {
33                  "Id": "dig:root",
34                  "OnEnter": {"Type": "None"},
35                  "Transitions": [digr_tr]
36              }
37          })
38
39      def str_transactions_msgs(self):
40          res = ''
41          for vertex in self.vertexes:
42              for transition in vertex.transitions:
43                  res += transition.str_msgs()
44          return res
45
46
47  class Vertex:
48      def __init__(self, cluster_n, transitions, is_digressions=False):
49          self.cluster_n = cluster_n
50          self.transitions = transitions
51          self.is_digressions = is_digressions
52
53      def __str__(self):
54          str_trans = []
```

```
55          for transition in self.transitions:
56              str_trans.append(str(transition))
57          return '"{}":'.format(self.cluster_n) + json.dumps({
58              "Id": str(self.cluster_n),
59              "OnEnter": {
60                  "Type": "Chain",
61                  "InnerReactions": [
62                  {
63                      "Type": "Simple",
64                      "Reactions": [
65                      {
66                          "msgId": "RawTextChannelMessage",
67                          "text": 'text{}'.format(self.cluster_n)
68                      }
69                      ]
70                  }
71                  ],
72              "Transitions": [
73                  [json.loads(json_tr) for json_tr in str_trans]
74              ]
75              }
76          })
77
78  class Transaction:
79      def __init__(self, msgs_set, from_cluster, to_cluster):
80          self.msgs = msgs_set
81          self.from_cluster = from_cluster
82          self.to_cluster = to_cluster
83          self.tr_name = 'fr_' + str(from_cluster) + '_to_' + str(to_cluster)
84
85      def __str__(self):
86          return json.dumps({
87            "Condition": {
88              "Type": "Fact",
89              "FactName": "input_info",
90              "InnerCondition": {
91                  "Type": "Field",
92                  "FieldName": "type",
93                  "InnerCondition": { "Type": "ItemEquals", "Item": self.tr_name
                        ↪  }
94              },
95              "Id": str(self.to_cluster),
96              "Priority": 0
97            }
98          })
99
100     def compare(self, tr):
101         self.msgs.intersection(tr.msgs)
102
103     def str_msgs(self):
```

```
        msgs = []
        for msg in self.msgs:
            msgs.append('{ "label":' + str(msg.labels or '{}') + ', "msg":' +
             ↪  '"{}"'.format(msg.text) + '}')
        return '{ "name":' + '"{}"'.format(self.tr_name) + ', "msgs": [' + ",
         ↪  ".join(msgs) + ']}'
```

```
# Copyright (C) 2020 Dasha.AI Inc.
from collections import defaultdict
from reconstruction.data_structure.dialog import Dialog

import numpy as np


def bfs_digressions_finder_0(graph, node, set_of_outgoing_clusters,
 ↪  set_of_incoming_clusters):
    """Trying to get tail from expected root, and cut vertices
    which can be potential continue after digressions. As a potential and
    we mark vertices befor root and next vertexes.

    Have troubles because of cycles, which does not exists in real dialogs"""
    first_step = set_of_outgoing_clusters[node]
    queue = set()
    marked = set()
    for node in first_step:
        queue = queue | set_of_outgoing_clusters[node]
        potential_end = set_of_incoming_clusters[node]
    for source in set_of_incoming_clusters[node]:
        potential_end = potential_end | set_of_outgoing_clusters[source]
    from_q = queue.copy()
    while queue:
        cur_node = queue.pop()
        for next_node in set_of_outgoing_clusters[cur_node]:
            if (next_node not in potential_end and next_node not in from_q):
                from_q.add(next_node)
                queue.add(next_node)
    marked = from_q.copy()
    # coming to digr part only from dirg + add to marked + add starts
    # queue = {graph["start"]}
    # from_q = queue
    # while queue:
    #     cur_node = queue.pop()
    #     print("Queue: " + len(queue))
    #     for next_node in set_of_outgoing_clusters[cur_node]:
    #         if (next_node != node and (next_node not in from_q)):
    #             from_q.add(next_node)
    #             queue.add(next_node)
    # marked = marked.difference(from_q)
    if len(marked) > 0:
```

```
42          graph['digressions'].append({
43              "root": node,
44              "nodes": list(marked)
45          })
46
47
48  def potential_digr_ver(unachive, dialogs_num, dialogs_out_of_dig):
49      possible_list = list()  # list with possible errors
50      for cluster in unachive:
51          if (len(dialogs_out_of_dig[cluster]) / dialogs_num[cluster] > 0.8):
52              possible_list.append({
53                  "cluster": cluster,
54                  "ex_list": dialogs_out_of_dig[cluster]
55              })
56      return possible_list
57
58
59  def digressions_finder(graph, node, dialogs):
60      cluster_labels = set()
61      achieve = set()
62      dialogs_num = defaultdict(int)
63      dialogs_out_of_dig = defaultdict(list)
64      for dialog in dialogs:
65          stop = True
66          for msg in dialog.log:
67              dialogs_num[msg.cluster] += 1
68              if (msg.cluster != node) and stop:
69                  achieve.add(msg.cluster)
70                  dialogs_out_of_dig[msg.cluster].append(msg.text)
71              else:
72                  stop = False
73              cluster_labels.add(msg.cluster)
74      unachive = cluster_labels - achieve
75      if len(cluster_labels) / 2 > len(unachive):
76          graph['digressions'].append({
77              "root": node,
78              "nodes": list(unachive),
79              "possible": potential_digr_ver(unachive, dialogs_num,
                ↪  dialogs_out_of_dig)
80          })
81
82
83  def digressions_by_different_income(graph, dialogs):
84      """Find digressions with finding potential starts and cut with some
        ↪  heuristic"""
85      set_of_incoming_clusters = defaultdict(set)
86      for link in graph["links"]:
87          set_of_incoming_clusters[link["target"]].add(link["source"])
88      num_of_incoming_clusters = dict()
89      for node in graph["nodes"]:
```

```
90          cluster = node[”id”]
91          num_of_incoming_clusters[cluster] = len(
92              set_of_incoming_clusters[cluster])
93          if (num_of_incoming_clusters[cluster] > 3):  # random const
94              if (not node[”incoming”]):
95                  digressions_finder(graph, cluster, dialogs)
96
97
98  def find_similar_nodes(nodes, income_shared):
99      nodes_sets = defaultdict(set)
100     potential_merge = [[]]
101     for node in nodes:
102         for transaction in node.transitions.keys():
103             if transaction in income_shared:
104                 nodes_sets[node.cluster_n].add(transaction)
105     for cluster1, set1 in nodes_sets.items():
106         if (len(set1) > 0):
107             nodes = []
108             for cluster2, set2 in nodes_sets.items():
109                 if (len(set1 & set2)) / (len(set1 | set2)) > 0.7 and len(set1)
                    ↪  * len(set2) > 1 and cluster1 != cluster2:
110                     nodes.append(cluster2)
111             if (len(nodes) > 0):
112                 nodes.append(cluster1)
113             if (len(nodes) > 1):
114                 potential_merge.append(nodes)
115     return potential_merge
116
117
118 def cut_cycle(dialog, start, end, messages, remove_marks):
119     log = dialog.log
120     if len(log) > end + 1:
121         log[start].next_msg = log[end + 1]
122         log[end + 1].prev_msg = log[start]
123     else:
124         log[start].next_msg = None
125     new_log = log[: start + 1] + log[end + 1:]
126     counter = start
127     for msg in log[start + 1: end]:
128         counter += 1
129         if msg in messages:
130             index = messages.index(msg)
131             remove_marks.append(index)
132     trigger_msg = log[start + 1]
133     end_msg = log[end]
134     while trigger_msg != end_msg and not trigger_msg.incoming:
135         if trigger_msg.next_msg:
136             trigger_msg = trigger_msg.next_msg
137         else:
138             trigger_msg = None
```

```
    return Dialog(dialog._id, dialog.task, new_log), trigger_msg, log[start + 1: end]


def cycle_finder(dialogs, messages):
    """Find digressions with finding cycles"""
    remove_marks = []
    new_dialogs = []
    trigger_list = []
    remove_part = []
    for dialog in dialogs:
        new_dialog = dialog
        have_cycle_flag = True
        cycles_counter = 0
        while have_cycle_flag:
            cycles_counter += 1
            have_cycle_flag = False
            nodes_map = {}
            counter = 0
            for msg in new_dialog.log:
                cluster = msg.cluster
                if cluster in nodes_map and counter - nodes_map[cluster] < 5:
                    #print("Cycle №", cycles_counter)
                    have_cycle_flag = True
                    new_dialog, trigger, remove_part_local = cut_cycle(
                        new_dialog, nodes_map[cluster], counter, messages, remove_marks)
                    remove_part += remove_part_local
                    if trigger:
                        trigger_list.append(trigger)
                    break
                if not msg.incoming:
                    nodes_map[cluster] = counter
                counter += 1
        new_dialogs.append(new_dialog)
    trigger_clusters = defaultdict(int)
    for trigger in trigger_list:
        trigger_clusters[trigger.cluster] += 1
    sorted_dict = {k: v for k, v in sorted(
        trigger_clusters.items(), key=lambda item: item[1])}
    print("TRIGGERS STAT:")
    for cluster, num in sorted_dict.items():
        print("        cluster: ", cluster, " num: ", num, " in ", sum(map(lambda x: x.cluster == cluster, messages)))
    return new_dialogs, remove_marks, remove_part
```

```
# Copyright (C) 2020 Dasha.AI Inc.
from reconstruction.embedding.mean_vectorizer import MeanEmbeddingVectorizer
from reconstruction.embedding.embedding_storage import EmbeddingStorage
```

```python
# Copyright (C) 2020 Dasha.AI Inc.
import re

from reconstruction.embedding.int_to_str import convert_int


def norm_text(text):
    text = re.sub(r'ё', 'е', text).lower()
    text = re.sub(r'”|!|\?|,|\.|%', '', text)
    text = re.sub(r'-', ' ', text)
    text = re.sub(r':', ' ', text)
    text = re.sub(r'::', ' ', text)
    replace_dict = {
        'ненадо': 'не надо',
        ”ололо”: 'ало',
        ”алло”: 'ало',
        ”але”: 'ало',
        ”ален”: 'ало',
        ”дадада”: ”да да да”,
        ”дадад”: ”да да да”,
        ”додо”: ”да да”,
        ”дада”: ”да да”,
        ”\bпасибо\b”: ”спасибо”,
        ”\bпасиб\b”: ”спасибо”,
        ”пасиба”:  ”спасибо”
    }
    for from_w, to_w in replace_dict.items():
        text = re.sub(from_w, to_w, text)

    text_arr = []
    for word in text.split():
        try:
            converted_text = convert_int(word)
        except ValueError:
            converted_text = word
        text_arr.append(converted_text)

    return ” ”.join(text_arr)
```

ПРИЛОЖЕНИЕ Б. ДИПЛОМ КОНГРЕССА МОЛОДЫХ УЧЁНЫХ