Аннотация

Ермаков А.К. Построение сценариев диалога для интеллектуального агента на основе условных планов. В работе рассматривается применение теории автоматического планирования в условиях частичной наблюдаемости к задаче диалога на естественном языке путем построения условных планов для заданной предметной области.

ОГЛАВЛЕНИЕ

Стр.

Введение

В работе рассматривается метод построения сценариев диалога на естественном языке с использованием условных планов в качестве модели. Актуальность темы обуславливается вектором развития программных продуктов в экосистеме наиболее известных операционных систем для мобильных устройств.

В апреле 2010 года Apple приобрела стартап “Siri” за $250 млн. Основным приобретаемым продуктом было одноименное приложение – персональный помощник и вопросно-ответная система. 4 октября 2011 года оно было представлено как часть программного обеспечения iPhone 4S. Принцип работы - голосовой диалог с пользователем, в ходе которого тот может получить информацию о погоде, котировках акций, новых сообщениях электронной почты, контролировать поведение устройства, включая воспроизведение музыки, создание событий в календаре и т.д.

Немногим позднее, в июне 2012 года, на очередной конференции Google I/O был представлен персональный помощник Google Now, имеющий сходную функциональность и поставляемый по умолчанию как часть операционной системы Android версии 4.1. Jelly Bean и выше.

Рисунок 1 — Приложение Siri

Работа этих продуктов основывается на экспертных системах с базой знаний, а диалоговые возможности подкрепляются технологиями голосового поиска.

Другим примером является российская разработка “Ассистент на русском” компании i-Free. Это приложение для Android, по функционалу аналогичное описанным выше продуктам. Доступно открытое API для создания собственных модулей. Для распознавания контекста диалога используется сопоставление по базе шаблонов.

Анализ решений, рассмотренных выше, привел к пониманию, что для более естественного взаимодействия с пользователем необходимо:

- Максимально полно использовать контекст предметной области диалога
- По возможности работать без постоянного подключения к сети Интернет.

В этой работе обсуждаются способы построения актуального контекста диалога, не зависящие от конкретной предметной области. В дальнейшем для реализации собственно диалогового модуля предполагается использовать доступные технологии распознавания

голоса (Google Voice API) и подходы с использованием библиотек морфологии и синтаксического анализа русского языка (Yandex Tamitha), а также базу шаблонов, аналогичную той, что используется в приложении i-Free.

В первой главе излагаются основы теории интеллектуальных агентов и известные классические подходы к проблеме автоматического планирования в условиях полной наблюдаемости. Далее вводятся понятия для рассмотрения планирования в условиях частичной наблюдаемости на примере “игрушечной” проблемы планирования.

Вторая глава дает концептуальное описание построенного алгоритма планирования, затем детально разбираются использованные алгоритмы. Приводятся оценки сложности.

Третья глава посвящена программной реализации: выбору языковых средств, библиотек; общей архитектуре системы и взаимодействию ее компонентов. Дается описание внешних программных интерфейсов (API).

В четвертой главе приводятся примеры работы программы, пользовательского интерфейса и описание работы с помощью командной строки. Описываются подходы к тестированию созданного программного обеспечения.

Глава 1

Автоматическое планирование

1.1 Интеллектуальный агент. Проблемная среда

Под интеллектуальным агентом понимают сущность, действующую в некоторой проблемной среде, получая информацию о ней с помощью сенсоров и воздействуя через исполнительные механизмы. Введем классификацию таких сущностей в форме PEAS:

- Performance – критерии производительности, которые стремиться максимизировать агент
- Environment – описание окружающей среды, предметной области[1]
- Actuators – исполнительные механизмы, или действия
- Sensors – доступные агенту датчики

Термином восприятие обозначаются полученные агентом сенсорные данные в любой конкретный момент времени.

Последовательностью актов восприятия агента называется полная история всего, что было когда-либо им воспринято.

Рациональный агент для каждой возможной последовательности актов восприятия должен выбрать действие, которое, как ожидается, максимизирует его показатели производительности, с учетом фактов, предоставленных данной последовательностью актов восприятия и всех встроенных знаний, которыми обладает агент.

Проблемная среда сама по себе также может быть охарактеризована с помощью ряда признаков:

Полностью или частично наблюдаемая

: Среда называется полностью наблюдаемой, если датчики агента предоставляют ему доступ к полной информации о состоянии среды в каждый момент времени. Частичная наблюдаемость может возникнуть, например, из-за создающих шум или неточных датчиков или из-за того, что отдельные характеристики ее состояния отсутствуют в информации, получаемой от датчиков.

Детерминированная или стохастическая

[1] Здесь и далее понятия проблемная, окружающая среда, предметная область и система будем считать взаимозаменяемыми.

Если следующее состояние среды полностью определяется текущим состоянием и действием, выполненным агентом, то такая среда называется детерминированной. В противном случае она является стохастической. Может создаться впечатление, что среда стохастическая в случае, когда она является частично наблюдаемой.

Эпизодическая или последовательная

В эпизодической среде опыт агента состоит из неразрывных эпизодов. Каждый эпизод включает в себя восприятие среды агентом, а затем выполнение одного действия. При этом каждый следующий эпизод не зависит от действий,
предпринятых в предыдущих. В противоположность этому, в последовательных вариантах среды каждое текущее решение может повлиять на все будущие.

Статическая или динамическая

Если среда может измениться в ходе того, как агент выбирает очередное действие, то такая среда называется динамической для данного агента, в противном случае – статической.

Дискретная или непрерывная

Различия между дискретными и непрерывными средами относятся к описанию состояния, способу учета времени, восприятиям и действиям агента.

Одноагентная или мультиагентная

Иногда целесообразно рассматривать пользователя как второго агента.

Поведение агента определяется его программой. Далее в этой главе будет рассматриваться класс планирующих целенаправленных агентов. Их можно описать следущим образом: при заданных описаниях начального и целевого состояний среды, построить последовательность доступных действий, которая переводит среду из начального состояния в целевое.

Описываемые ниже подходы решают так называемую задачу полного планирования, когда план по достижению цели строится один раз и целиком. Это накладывает определенные ограничения на типы среды. Предполагается, что она является полностью наблюдаемой, детерменированной, последовательной, статической, дискретной, одноагентной.

1.2 Классические алгоритмы планирования

1.2.1 Прямой и обратный вывод

В случае, если состояние системы описано полностью и не совпадает ни с каким другим, назовем его физическим состоянием. В таком случае исполнительные механизмы, или действия агента, будут представлять собой дуги направленного графа, где множество вершин будет совпадать с пространством всех возможных физических состояний среды.

Рассмотрим два классических подхода, являющихся разновидностями поиска в пространстве состояний. Особенностью этих алгоритмов является большой размер графа состояний. Пусть физическое состояние среды описывается набором из N бинарных призна-

ков. Тогда общее число физических состояний среды равно:

$$S = \{S_j : \langle A_1, \ldots A_N \rangle, A_i \in \{0,1\}\} \implies |S| = 2^N. \tag{1.1}$$

На практике при использовании только бинарных признаков для описания их число может достигать нескольких сотен. Очевидно, что построить полный граф физических состояний не представляется возможным. Вводится понятие неявного графа, вершины и ребра которого не имеют физического представления, а генерируются алгоритмически.

Как правило, описания доступных действий разделяются на пред- и постусловия, заданные на языке формальной логики[2]. Таким образом, действие может быть выполнено тогда, когда выполняется его предусловие, а результатом выполнения будет физическое состояние, полученное путем применения постусловия к текущему состоянию.

Прямой логический вывод осуществляет попытку рекурсивно построить цепочку логических высказываний, соответствующую определенным пред- и постусловиям действий, которая приведет из старта в финиш.

В противоположность ему, обратный логический вывод решает аналогичную задачу, но в качестве начальной вершины выбирается целевое состояние.

1.2.2 Сведение к задаче CSP (SAT)

Другим подходом к решению задачи планирования является сведение к задаче CSP[3]. При этом необходимо построить логическую формулу, которая выполнима тогда и только тогда, когда выполним план.

Неформально алгоритм построения формулы можно описать следующим образом. Пусть план содержит k действий. Для каждого возможного действия a введем булеву переменную:

$$Action(a,i) = true \iff \text{действие a используется на шаге i} \tag{1.2}$$

Для каждого бинарного признака в описании физического состояния также введем булеву переменную:

$$Proposition(p,i) = true \iff \text{Признак p истинен на шаге i} \tag{1.3}$$

Кроме того, вводятся дополнительные ограничения:

- На каждом шаге может быть выполнено только одно действие (XOR-ограничения)
- Ограничения, описывающие постусловия действий
- Ограничения, описывающие неизменность признака, если он не был изменен действием

[2]Далее будет использоваться язык пропозициональной логики для записи логических формул. Приложение 1 содержит его описание

[3]CSP – (англ. Constraint satisfaction problem) задача удовлетворения ограничений.

- Признак истинен на определенном шаге только как результат постусловия какого-либо действия
- Ограничения, описывающие начальное и конечное состояния

1.2.3 Graphplan

Также возможен подход с использованием специально сконструированного графа, обладающего следующими свойствами:

- Направленный многоуровневый граф, где уровни чередуются
- Нечетные уровни (уровни состояний) представляют собой признаки, которые возможно являются положительными на i-м шаге плана.
- Четные уровни (уровни действий) представляют возможные действия, которые могли бы быть произведены на каждом шаге, включая бездействие.
- Дуги графа представляют собой пред- и постусловия

Построение графа происходит по правилам:

1. Добавить действие на уровень A_i, если все его предусловия выполнены на уровне S_i
2. Добавить литерал на уровень S_i, если он истинен в результате эффекта хотя бы одного действия на уровне A_{i-1}
3. Уровень S_0 содержит все литералы из начального состояния

Рис. 1.1 содержит пример построенного графа для одной из задач планирования.

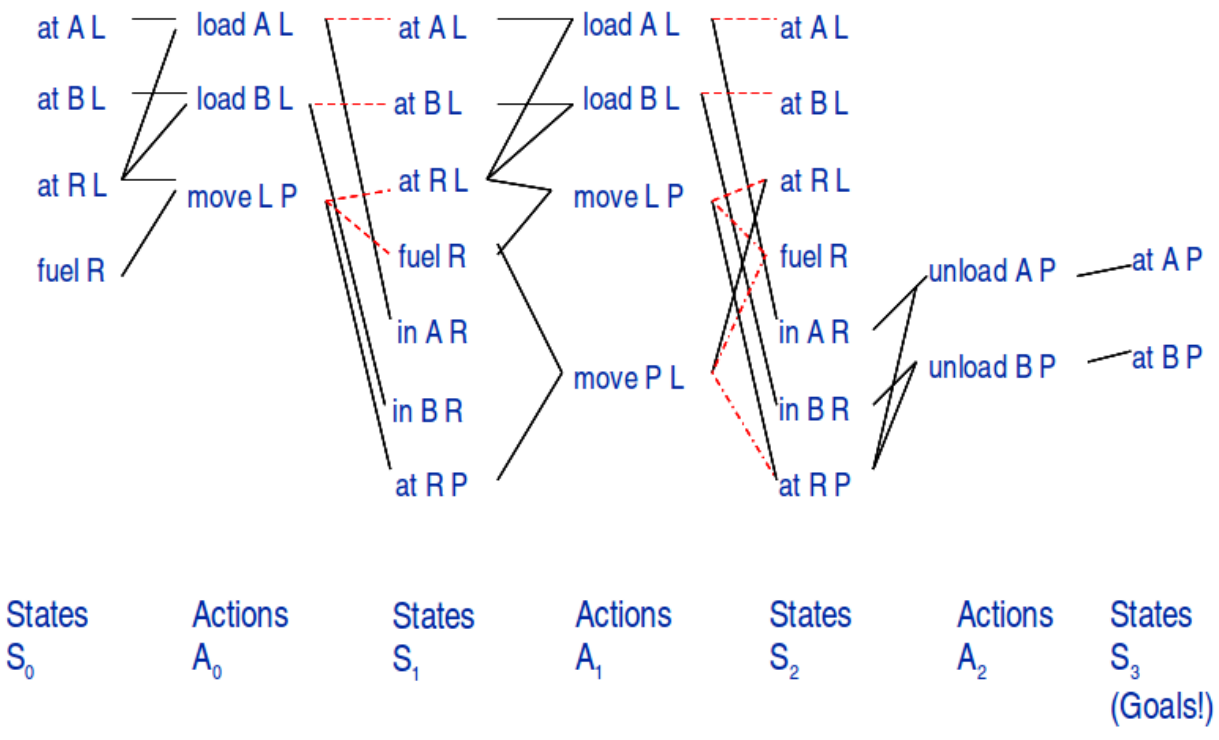

Рисунок 1.1 — Пример работы алгоритма Graphplan

1.3 Планирование в условиях частичной наблюдаемости

Ранее предполагалось, что в любой момент времени агенту полностью известно текущее физическое состояние среды. Как правило, это не так. В этом разделе даются определения, необходимые для описания программ агентов, действующих в среде с частичной наблюдаемостью, и приводится одна из возможных стратегий – условное планирование. Попутно вводится описания языка PDDL, являющегося академическим стандартом для описания проблем планирования.

1.3.1 Пропозициональная логика

При обсуждении условного планирования используется язык пропозициональной логики (лат. propositio — «высказывание»). Это раздел символической логики, изучающий сложные высказывания, образованные из простых, и их взаимоотношения. С точки зрения выразительности, логику высказываний можно охарактеризовать как классическую логику нулевого порядка.

Язык

Алфавит языка разделен на три группы.

- Пропозициональные переменные: $A, B, C, ...; A \in \{0, 1\}$;
- Логические союзы: : $\neg$ — знак отрицания, $\wedge$ — знак конъюнкции, $\vee$ — знак дизъюнкции, $\Longrightarrow$ — знак импликации, $\Longleftrightarrow$ — знак эквивалентности;
- технические знаки: (— левая скобка,) — правая скобка.

Сам язык можно индуктивно определить следующим образом:

1. пропозициональная переменная есть формула;
2. если A – произвольная формула, то $\neg A$ – тоже формула;
3. если A и B – произвольные формулы, то $(A \Longrightarrow B)$, $(A \wedge B)$, $(A \Longleftrightarrow B)$, $(A \vee B)$ – также формулы

Выполнимость булевых формул

При работе с высказываниями пропозициональной логики возникает задача выполнимости булевых формул, SAT[4], которая формулируется так: можно ли назначить всем переменным, встречающимся в формуле, значения ложь и истина так, чтобы формула стала истинной. Известно, что эта задача в общем случае обладает свойством NP-полноты.

1.3.2 Язык PDDL

Стандартом де-факто для описания задач планирования является язык PDDL[5]. Основными его понятиями являются проблемная среда и задача (проблема), поставленная

[4]SAT – англ. boolean SATisfability problem
[5]Planning Domain Definition Language

на ней в виде начального и конечного состояний. В дальнейшем понятия проблемная среда и предметная область будут рассматриваться как взаимозаменяемые.

В описании проблемной среды положим, что ее состояние задано набором признаков. Для простоты будем все признаки полагать бинарными. Каждый возможный набор их значений называется физическим состоянием среды. Далее, представим датчики агента как акты восприятия значений одного или более признаков. Исполнительные механизмы будут представлять собой действия, переводящие среду из одного физического состояния в другое.

Проблемной среде исоответствуют следующая конструкция языка:

```
(define (domain <name>)
    <description>
)
```

1.3.3 Пример: мир Лампы

Для более наглядного обсуждения планирующих агентов введем "игрушечную" проблему, которую неформально можно описать следующим образом: в некой комнате находится лампа накаливания, соединенная выключателем с источником питания. Там же находятся кондиционер и окно (считаем, что на улице ночь). Необходимо включить свет.

1.3.4 Доверительное состояние

Более формально, введем набор бинарных признаков, описывающих среду:

- L. Свет горит (не горит)
- B. Лампа исправна (неисправна)
- S. Выключатель включен (выключен)
- W. Окно открыто (закрыто)
- C. Кондиционер включен (выключен)

Тогда например, состояние среды в котором лампа исправна, выключатель выключен и свет не горит, представляет собой четыре различных физических состояния:

$$\langle \neg L, B, \neg S, \neg W, \neg C \rangle, \langle \neg L, B, \neg S, \neg W, C \rangle, \langle \neg L, B, \neg S, W, \neg C \rangle, \langle \neg L, B, \neg S, W, C \rangle, \tag{1.4}$$

Они получены перебором различных значений последних двух признаков. Введем понятие доверительного состояния среды как подмножества множества ее физических состояний. Для записи признаков, чье значение неизвестно, введем сокращенную запись

$$A? = A \vee \neg A \tag{1.5}$$

Тогда запись выше можно сократить до:

$$\langle \neg L, B, \neg S, W?, C? \rangle \tag{1.6}$$

В PDDL для описания признаков используется следующая конструкция:

```
(: predicates (L) (S) (B) (W) (C) )
```

Формализуем также описание задачи, задав начальное и конечное состояние в виде логических формул:

```
(define (problem <name>)
    (: domain <domain_name> )
    (: init (~L) )
    (: goal (L) )
)
```

1.3.5 Действия

Далее, введем действия, позволяющие изменять значения признаков. Понятно, что в реальной среде не каждое действие доступно для выполнения в любой момент времени. Поэтому вводятся понятия предусловия(precondition) и постусловия(effect) действия как логических выражений с переменными в множестве признаков. Если текущее физическое состояние не совпадает с предусловием действия, то и его постусловие не будет выполнено.

Действие	Описание	Предусловие	Постусловие
chbulb	Заменить лампу	Выключатель выключен	Лампа исправна
brbulb	Разбить лампу	Лампа исправна	Лампа неисправна. Свет не горит
onlight	Включить выключатель	Нет	Выключатель включен. Если лампа исправна, свет горит
offlight	Выключить выключатель	Выключатель включен	Выключатель выключен. Свет не горит
onwind	Открыть окно	Окно закрыто	Окно открыто
offwind	Закрыть окно	Окно открыто	Окно закрыто
oncond	Включить кондиционер	Кондиционер выключен	Кондиционер включен
offcond	Выключить кондиционер	Кондиционер включен	Кондиционер выключен

Таблица 1.1 — Описание действий доступных в мире Лампы

В PDDL действия содержатся в описании предметной области и задаются следующей конструкцией:

```
    (: action ch_bulb ;Заменить лампу
        :precondition (B? & ~S)
        :effect (B)
    )
```

Для заданного доверительного состояния выполнимость предусловия какого-либо действия может быть неизвестной. Например, в состоянии $\langle \neg L, B?, S? \rangle$ результатом действия *chbulb* может оказаться как то же самое состояние в случае, когда S положителен, так и состояние $\langle \neg L, B, \neg S \rangle$, когда предикат отрицателен.

1.3.6 Аксиомы

Можно заметить, что описания действий содержат дублирующуюся информацию. Язык PDDL содержит понятие аксиомы - логической формулы, справедливой в любом доверительном состоянии. Для мира Лампы справедлива следующая аксиома: свет горит тогда и только тогда, когда лампа исправна и выключатель включен.

$$L \iff B \wedge S \tag{1.7}$$

Ей соответствует конструкция языка:

```
(: axiom (L <-> B & S) )
```

Приложение Б. Результаты выполнения программы содержит полный вариант PDDL-описания мира Лампы.

1.3.7 План – граф доверительных состояний

Так как мы рассматриваем частично наблюдаемые среды, то будем считать, что агенту недоступно восприятие значений каких-либо признаков напрямую. Результатом работы планировщика будет граф, отображающий возможные пути из начального доверительного состояния в целевое. Классическим способом представления неопределенности результата действия, рассмотренного ранее, является *AND-OR* граф.

Рис. 1.2 содержит пример построенного $AND-OR$ графа для мира Лампы.

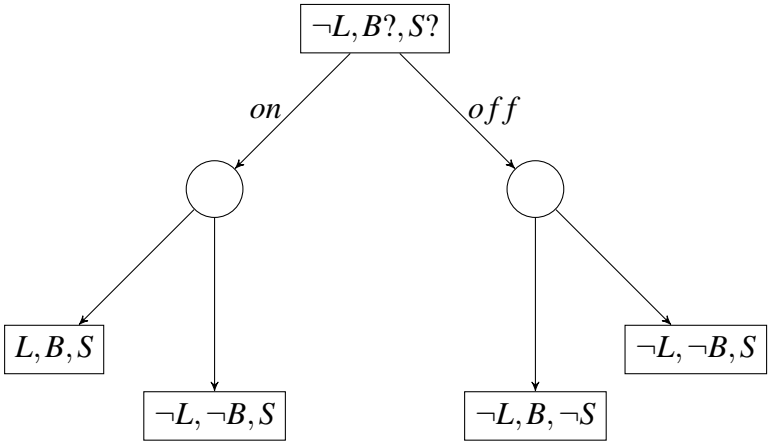

Рисунок 1.2 — Пример частичного $AND-OR$ графа для проблемы включения света в мире Лампы.

Алгоритм 1 Алгоритм построения AND-OR графа

```
function AndOrGraphSearch(problem)
    return OrSearch(Init, problem)
function OrSearch(state, problem, path)
    if IsGoal(problem, state) then
        return ∅
    if state ∈ path then
        return Failure
    for (action, states) ← Successors(problem, state) do
        plan← AndSearch(states, problem, statepath)
        if plan ≠ Failure then
            return [action|plan]
    return Failure
function AndSearch(states, problem, path)
    for s_i ← states do
        plan_i ← OrSearch(s_i, problem, path)
        if plan_i = Failure then
            return Failure
    return [
    if s_1 then plan_1 ...
    else if s_{n-1} then plan_{n-1}
    elseplan_n
    ]
```

Глава 2

Интерактивный агент

Как упоминалось ранее, академические планирующие агенты нацелены на полностью автономную работу. Основной критерий производительности для них - скорость построения оптимального плана.

В случае, когда агент, кроме выполнения последовательности действий, должен взаимодействовать с другими агентами, в качестве которых могут выступать пользователи, возникают две новых задачи:

- Диалог. Мультиагентная среда предполагает взаимодействие с другими участниками для получения наилучших результатов.
- Онлайн-планирование. Свойство динамичности проблемной среды требует регулярного перестроения плана в условиях изменения начальных и конечных условий поставленной задачи.

Вопросы

Введем новый вид действий под названием вопрос. Изменениям всегда будет подвергаться только один признак. В качестве предусловия будем полагать, что значения выбранного признака неизвестно. Вопрос порождает две дуги из текущего доверительного состояния, соответствующие возможным полученным значениям признака. Фактически, мы инкапсулируем два различных действия в одном. Например, для вопроса о значении признака L:

```
(: action L_yes :precondition (L?) :effect (L) )

(: action L_no :precondition (L?) :effect (~L) )
```

Для избежания избыточности в PDDL-описании проблемной среды вопросы явно указываться не будут.

Интерактивное планирование

В мультиагентной проблемной среде агент фактически разбивается на несколько. Некоторые из них взаимодействуют с другими агентами в среде. Мы можем выделить:

- Планирующий агент. Действует в статической одноагентной среде, заданной в виде PDDL описания предметной области и поставленной задачи планирования. Цель агента - построение условного плана
- Диалоговый агент. Обладает исполнительными механизмами ведения диалога с другими агентами. Действует в динамической мультиагентной среде в соответствии с программой заданной условным планом. При изменении задачи обращается к планирующему агенту.

2.1 Планирующий агент. Построение условного плана

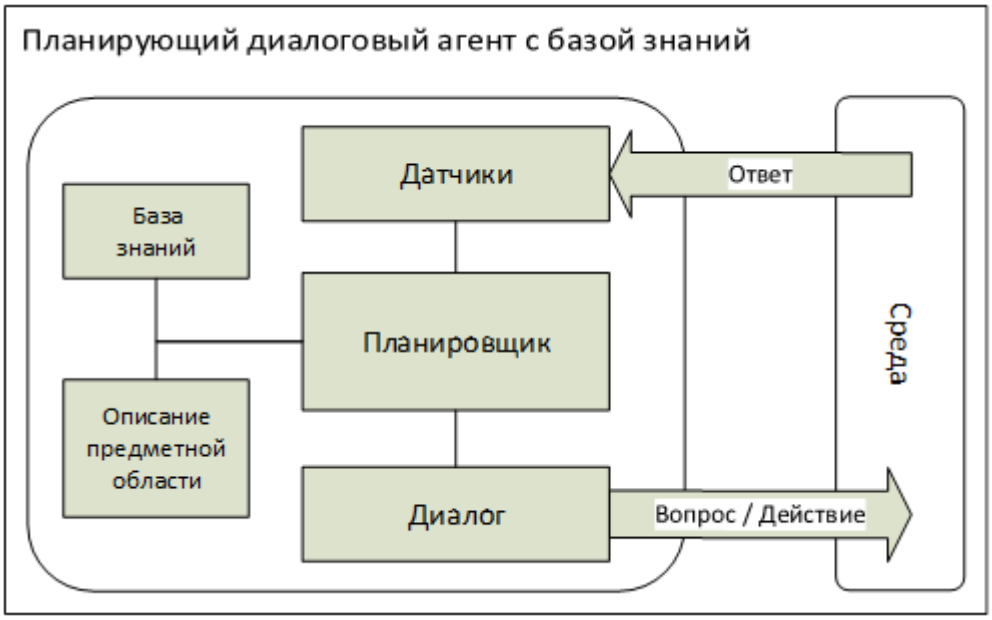

Рисунок 2.1 — Схема планирующего диалогового агента с базой знаний

Условный план строится рекурсивно из начального состояния. Приведем неформальное описание алгоритма:

1. Определить начальное доверительное состояние
2. Построить преемников начального состояния
   (a) Ветвления по неизвестным признакам
   (b) Гарантированный путь до цели
   (c) Смешанный путь до цели
3. Добавить все полученные вершины в очередь
4. Если очередь не пуста, выбрать ее вершину в качестве начального состояния и перейти на шаг 2
5. Если очередь пуста - план построен.

2.1.1 Поиск начального состояния

Как описано далее, в каждом новом состоянии необходимо множество раз решать задачу SAT с участием конъюнкции участвующих в описании состояния среды. Как правило, не все они изменяются в процессе выполнения условного плана. Поэтому для ускорения дальнейшей работы алгоритма целесообразно попытаться исключить те из предикатов, значения которых не принимаются во внимание.

Для этого предлагается подход с поиском пути из целевого состояния в какое-либо подмножество начального, состоящего целиком из действий. На каждом шаге алгоритма предикаты, отсутствующие в описании целевого или начального состояний, добавляются в специальное множество. Далее начальное состояние, заданное в описании задачи планирования, дополняется этим множеством с пометкой, что каждый предикат имеет неизвестное значение.

В дальнейшем, если при прямом поиске из начального состояния алгоритм сталкивается с ситуацией, когда изменяется значение предиката, не участвующего в выборке, начального состояния дополняется и производится поиск неучтенных веток условного плана, возможно ведущих к цели.

Открытым остается вопрос выбора равнозначных наборов предикатов, соответствующих различным независимым путям из начального в целевое состояние.

Алгоритм 2 Обратный поиск начального состояния из целевого

```
Require: predicates, actions, axiom
  function BackwardSearch(init, goal)
      clause ← axiom ∧ init ∧ goal
      models ← RetrieveModels(clause)
      if models = ∅ then
          return BackwardAction(init, goal)
      else if ∃model ∈ models then
          return RetrieveSymbols(model, axiom ∧ init)
  function BackwardAction(init, goal)
      for all action ← actions do
          models ← RetrieveModels(axiom ∧ goal ∧ action.effect)
          if models ≠ ∅ then
              goalModels ← models
          models ← RetrieveModels(axiom ∧ goal ∧ action.precondition)
          if models ≠ ∅ then
              initModels ← models
      if initModels = ∅ ∧ goalModels = ∅ then
          return ∅
      else if goalModels ≠ ∅ then
          action ← goalModels.head
          return BackwardAction(init, action.precondition)
```

2.1.2 Рекурсивный прямой поиск из начального состояния

Пусть выбрано текущее начальное состояние *init*, соответствующее вершине графа. Необходимо построить множество исходящих из него дуг. Для этого вводится функция *Successors*, включающая в себя следующие шаги конструирования:

Ветвление по неизвестным признакам

Для каждого из признаков с неизвестным значением P_i строится две дуги с действием $Question(P_i)$, которые ведут в состояния вида:

$$\langle P_1, \ldots, P_i, \ldots, P_n \rangle, \langle P_1, \ldots, \neg P_i, \ldots, P_n \rangle, \ldots \tag{2.1}$$

Поиск гарантированного пути до начального состояния

С помощью алгоритма поиска оптимального пути в дереве – например, A^* – строится путь из текущего начального состояния в целевое. На каждом шаге алгоритма поиска переходы определяются дугами, соответствующими эффектам всех возможных действий, примененных к этому состоянию. Так как в итоге получается путь на графе без ветвлений, он гарантированно выполним вне зависимости от значений неизвестных признаков в исходном состоянии.

В случае, если из текущего начального состояния нельзя придти к целевому, оно помечается тупиковым и отбрасывается.

Поиск смешанного пути до начального состояния

Аналогично строится смешанный путь. Отличие состоит в том, что для каждого состояния к дугам, соответствующим действиям, добавляются дуги ветвления по неизвестным признакам.

Алгоритм 3 Прямой ход алгоритма построения плана

```
function BuildPlan(domain, problem)
    init ← problem, goal ← problem
    initState ← BackwardSearch(init, goal)
    frontier ← Queue(init)
    plan ← DoBuild(frontier, ∅)
    return PlanDescription(init, plan, problem)
function DoBuild(frontier, plan)
    if frontier = ∅ then
        return plan
    else
        (next, Q_next) ← Dequeue(frontier)
        if ∃p ∈ Predicates(next) , p ∉ Predicates(plan) then
            return Rebuild
        (E_next, S_next) ← Successors(next)
        return DoBuild(Q_next ∪ S_next, plan ∪ E_next)
```

2.1.3 Алгоритм построения пути

Для построения гарантированных и смешанных путей из заданной вершины графа используется алгоритм поиска A^*. Он относится к классу информированных алгоритмов поиска, т.е. тех, которые для поиска используют дополнительную информацию о задаче. Здесь в качестве таковой используется эвристическая функция $h(x)$.

Алгоритм 4 Алгоритм поиска пути (A^*)

```
function A*(start, goal)
    h ← H(init, goal)
    node ← Node(init, h, None)
    return DoSearch(node, ∅, goal)
function DoSearch(frontier, explored, goal)
    if frontier = ∅ then
        return Failure
    else
        best ← Pop(frontier)
        if IsGoal(best, goal) then
            return Solution(best)
        else
            newExplored = explored ∪ {best}
            newFrontier = frontier \ {best}
            for child ←Successors(best) do
                h_child ←H(child, goal)
                node ←Node(child, h_child, best)
                if ¬(child ∈ explored) ∧ ¬(child ∈ frontier) then
                    newFrontier ← node
                else if ∃n ∈ frontier : TotalCost(n) > TotalCost(best) + Cost(child) then
                    newFrontier = newFrontier \ {n} ∪ {node}
            DoSearch(newFrontier, newExplored, goal)
```

Как видно, для работы алгоритма необходимо определить вспомогательные функции:

$Successors(state)$

Исходящие дуги и порождаемые ими вершины для заданного состояния

$Cost(node)$

Стоимость перехода для заданной дуги

$H(state_1, state_2)$

Эвристическая функция расстояния между двумя вершинами

$IsGoal(state)$

Проверка, входит ли состояние в множество целевых

Как уже говорилось ранее, основные различия гарантированных и смешанных путей заключаются в определении функции $Successors(state)$. А именно, во втором случае результат ее работы дополняется результатами работы алгоритма ветвления для заданного состояния.

$Cost(node)$ может быть определена исходя из стоимости различных действий, заданной

в описании предметной области. В тестовой задаче мира Лампы для простоты полагается стоимость любого действия равной единице, а вопроса - нулю. Таким образом, развитие плана через диалог всегда более предпочтительно.

2.1.4 Перестроение плана

Одним из ключевых вопросов реализации алгоритма, который до сих пор не решен, является случай введения в рассмотрение новых признаков в ходе прямого хода алгоритма. Очевидно, что полное перестроение плана при введении каждого нового признака в описание начального состояния - крайне неэффективно. Оптимизировать этот подход можно в двух направлениях:

1. Определение ситуаций, в которых фактическое введение нового признака в рассмотрение не повлияет на конфигурацию плана, и таким образом, может не производится
2. Способы частичного перестроения плана, дополнения уже существующего графа

2.1.5 База знаний. Выбор наилучшего набора аксиом

В этом разделе обсуждается эвристика, применяемая для минимизации количества различных моделей в возникающих комбинациях конъюнкций правил в базе знаний. Если формула имеет только одну модель, и заранее известна истинность самой формулы, можно получить значения всех входящих в нее переменных. Так, для формулы

$$(L \leftrightarrow B \wedge S) \wedge L \tag{2.2}$$

Имеем:

$$(L \rightarrow True, B \rightarrow True, S \rightarrow True) \tag{2.3}$$

С другой стороны, для формул, содержащих непересекающиеся наборы переменных, наверняка можно сказать, что общее число моделей будет не менее, чем NM, где N – число моделей первой формулы, а M – второй. Пример:

$$(L \iff B \wedge S) \wedge (W \iff C) \tag{2.4}$$

Исходя из выше сказанного, для определения истинности какого-либо высказывания из правил базы знаний следует выбирать только формулы, имеющие наименьшую мощность симметрической разницы между множеством символов формулы и множеством символов высказывания, чью выполнимость необходимо выяснить. Пусть

$$S(A) = \{a_1, \ldots, a_n\} \tag{2.5}$$

– множество символов формулы A. Для выяснения выполнимости высказывания B

строится конъюнкция правил базы знаний R по принципу:

$$E_i = E_{i-1} \wedge R : R = min\{R_j \rightarrow |S(R_j) \otimes E_{i-1}|\} \tag{2.6}$$

2.2 Диалоговый агент

Условный план в общем случае представляет собой ориентированный граф доверительных состояний и переходов между ними. В приложении этой структуры к построению диалога с пользователем необходимо различать и обрабатывать различные шаблонные подграфы, на основе которых строится сценарий диалога. Например:

- Листья графа, не являющиеся целевыми состояниями. Соответствуют тупиковым переходам в сценарии диалога
- Пути, построенные с помощью алгоритма смешанного поиска
- Тривиальный случай, когда начальное состояние является подмножеством конечного

Для того, чтобы наглядно рассмотреть варианты действий в этих ситуациях, введем новую задачу.

2.2.1 Пример: мир Сигнализации

В этом разделе описывается более сложная, но все еще “игрушечная” проблемная среда и задача планирования.

Неформальное описание

Иммобилайзер (англ. immobiliser — «обездвиживатель») — вид электронного противоугонного устройства. Предназначено для исключения возможности запуска двигателя автомобиля в отсутствие разрешающего сигнала. Для этого используется либо контроллер впрыска двигателя, либо устройства разрыва цепей стартера, зажигания. Существуют также противоугонные устройства основанные на физическом блокировании систем автомобиля. Их также включим в это понятие.

Пусть в автомобиле установлены следующие устройства: электронный иммобилайзер, датчик открытой двери, двигатель, зажигание. Водителю также доступны ключ зажигания и брелок сигнализации.

Опишем свойства объектов в проблемной среде следующими предикатами:

Возможные действия, доступные агенту:

Полное PDDL-описание для мира Сигнализации приводятся в Приложении Б.

2.2.2 Выполнение плана

Для достижения цели применяется рекурсивный подход, аналогичный тому, что используется при построении плана. Из дуг, исходящих из начального состояния, по некоторому правилу выбирается одна и осуществляется попытка перехода по ней. В случае,

Предикат	Описание
D_{in}	Водитель находится внтури автомобиля
DS_b	Датчик открытой двери неисправен
DS_{on}	Датчик открытой двери сигнализирует
KC_b	Брелок сигнализации неисправен
KC_{on}	Брелок сигнализации подает сигнал
$Door$	Дверь открыта
$Alarm$	Сигнализация включена
$Drive$	Автомобиль в движении
K	Ключ зажигания в замке
E_{on}	Двигатель включен
$Immo$	Иммобилайзер блокирует работу двигателя

Таблица 2.1 — Предикаты, описывающие физическое состояние мира Сигнализации.

Имя	Предусловие	Эффект	Описание
$EngineOn$	$D_{in} \wedge K$	$E_{on} \wedge Alarm?$	Завести двигатель
$InsertKey$	D_{in}	K	Вставить ключ в замок зажигания
$GetIn$	$\neg D_{in} \wedge Door$	D_{in}	Сесть в автомобиль
$GetOut$	$D_{in} \wedge Door$	$\neg D_{in}$	Выйти из автомобиля

Таблица 2.2 — Действия, доступные агенту в мире Сигнализации

если попытка была успешной, в качестве начальной выбирается новая вершина и процесс повторяется. В противном случае дуга помечается и снова осуществляется выбор. Если у вершины нет исходящих дуг, или ни по одной из них не удалось осуществить переход, выполнение плана останавливается.

Можно увидеть, что в случае тривиальной задачи, где целевое состояние является подмножеством начального, алгоритм закончит работу на первом же шаге.

2.2.3 Смешанные пути

В случае смешанного пути, состоящего из вопросов и действий, возможна ситуация, когда в зависимости от ответа пользователя будет совершен либо переход в следующее состояние на пути к целевому, либо в тупиковое. Рассмотрим пример такой ситуации.

Зададим следующую задачу в мире Сигнализации: пусть брелок удаленного управления и датчик открытой двери неисправны. Водитель открывает дверь, садится за руль и хочет начать движение. Текущее состояние иммобилайзера неизвестно, и может либо произойти блокировка дверей и двигателя для защиты от угона, либо водителю удастся успешно сдвинутся с места.

$$\langle D_{in} \wedge DS_b \wedge KC_b \wedge Alarm?, Drive \rangle \tag{2.7}$$

Алгоритм 5 Рекурсивное выполнение плана

```
function ProcessState(plan, state, questions, actions)
    if IsGoal(state) then
        return Success
    if questions ≠ ∅ then
        question ← ChooseQuestion(state, questions)
        result ← Ask(question)
        if result ≠ Failure then
            questions_result ←Questions(plan, result)
            actions_result ←Actions(plan, result)
            ProcessState(plan, result, questions_result, actions_result)
        else
            ProcessState(plan, state, questions\{question}, actions)
    else if actions ≠ ∅ then
        action ← ChooseAction(state, actions)
        result ← Execute(action)
        if result ≠ Failure then
            questions_result ←Questions(plan, result)
            actions_result ←Actions(plan, result)
            ProcessState(plan, result, questions_result, actions_result)
        else
            ProcessState(plan, state, questions, actions\{action})
    else
        return Failure
```

В этом случае будет построен план, последними двумя действиями которого будут:

$$(EngineOn, Alarm?) \tag{2.8}$$

Для последней дуги, представляющей вопрос, возможны два перехода в зависимости от значения предиката *Alarm*:

$$\langle Drive, \neg Immo \rangle, \langle \neg Drive, Immo \rangle \tag{2.9}$$

Первое из этих состояний – целевое. Второе соответствует вершине без исходящих дуг. При попадании в него выполнение плана автоматически завершается неудачей. Для того, чтобы отслеживать такие ситуации, предлагается ввести дополнительные критерии расчета стоимости пути, в частности, эвристической функции:

$$Succesors(n) = \emptyset \implies H(n, goal) = \infty \tag{2.10}$$

Тогда для смешанных путей можно построить верхние и нижние оценки стоимости, и применять их уже при выполнении плана. Таким образом, путь с возможными тупиковыми ветвями буедт выбран в последнюю очередь.

Глава 3

Проектирование и реализация

3.1 Программная модель планировщика

Ранее было выделено две основные сущности, из которых состоит проектируемый интеллектуальный агент: планирующий и диалоговый агенты. Целесообразно выделить в отдельный компонент обход плана в ходе его выполнения и алгоритмы выбора действий и вопросов. Тогда эта алгоритмическая база может быть в дальнейшем использована в множестве различных реализаций функциональности диалога.

Еще одна часть системы возникает из необходимости преобразования описаний предметной области из текстовой формы языка PDDL во внутреннее программное представление. Кроме того, предполагается, что в будущем будет возможно изменение этого представления с помощью диалогового агента.

Рис. 3.1 отображает взаимодействие этих компонентов.

Рисунок 3.1 — Общая схема работы программной реализации агента

Рис. 3.2 описывает процесс достижения цели.

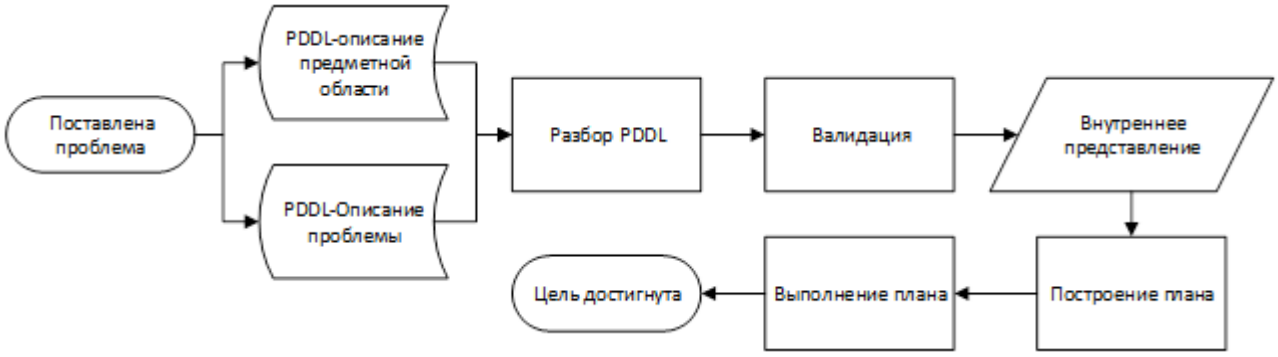

Рисунок 3.2 — Блок-схема процесса достижения цели

3.2 Инструментальные средства

Разработка проекта ведется на языках высокого уровня Scala, Java. Общий объем кодовой базы составляет порядка 2300 LOC[1]. Основная функциональность покрыта приблизительно 50 юнит-тестами, объем кода которых занимает около 450 LOC от общего.

3.2.1 Scala

В качестве основного средства разработки был выбран язык Scala - мультипарадигмальный язык программирования, спроектированный кратким и типобезопасным для простого и быстрого создания компонентного программного обеспечения, сочетающий возможности функционального и объектно-ориентированного программирования (Wikipedia, 2014).

- Мощная система статической типизации с поддержкой абстрактных типов данных: traits, типажи
- Полная совместимость с Java на уровне байт-кода JVM[2]
- DSL[3] для работы с логическими формулами, повышающий читаемость и выразительность кода: `val exp = 'L iff 'B & 'S`
- Функциональная парадигма программирования позволяет писать потокобезопасный код с использованием неизменяемых объектов, выражать алгоритмы в краткой рекурсивной форме без потери производительности с помощью хвостовой рекурсии[4]
- Простая модель многопоточности без непосредственного управления жизненным циклом потока на основе передачи сообщений, с помощью классов Actor.
- Встроенный генератор грамматических парсеров, аналогичный Java-библиотеке ANTLR.

3.2.2 Обзор библиотек

Проект не использует проприетарных решений и полностью построен на программных библиотеках с открытым исходным кодом. Хотя текущая реализация по качеству далека от необходимого для использования где-либо кроме учебных задач, она опирается на проверенные решения, имеющие большой запас производительности:

- SAT4J
- JGraphT
- Swing

Отдельно можно выделить структуры данных и алгоритмы пакета scala.* – стандартной библиотеки языка. Повсеместно используются реализации списков, множеств, хэш-таблиц. По умолчанию подключаются неизменяемые (immutable) версии. В будущем это

[1] LOC – lines of code
[2] JVM – Java Virtual Machine, виртуальная машина Java
[3] DSL – Domain Specific Language
[4] Хвостовая рекурсия (англ. tail recursion) – частный случай рекурсии, при котором любой рекурсивный вызов является последней операцией перед возвратом из функции.

позволит избежать ошибок многопоточности с одновременным изменением одного объекта из нескольких потоков и отменяет нужду в синхронизации. Тем не менее, такой подход требует больших затрат оперативной памяти.

SAT4J

Пакет org.sat4j.scala.∗ предоставляет набор классов и функций для записи и вычисления пропозициональных логических формул в форме DSL. Основным же компонентом является решатель задачи выполнимости булевых формул (SAT), который находится в пакете org.sat4j.core.∗. Библиотека предоставляет два варианта его реализации: основной и облегченный, оба доступны через шаблон проектирования "Фабрика":

```
1 ISolver default = org.sat4j.minisat.SolverFactory.defaultSolver();
2 ISolver light = org.sat4j.minisat.SolverFactory.lightSolver();
```

Последний предназначен для задач с числом формул порядка 100-1000. Интерфейс ISolver принимает логические формулы в формате Dimacs, который представляет собой набор массивов целых чисел, где номер - индекс соответствующей переменной, знак минус соответствует отрицанию. Далее, отдельный массив представляет собой дизъюнкцию входящих в него литералов, а все вместе они образуют конъюктивную нормальную форму.

Доступны классы для конвертирования в этот формат из других, например привычной записи логических формул с помощью литералов и операций над ними – эта функциональность используется при импорте из PDDL.

JGraphT

Java-библиотека работы с графами. Используемый API - DirectedMultigraph и его стандартная реализация, DefaultDirectedMultigraph.

Swing

Библиотека для создания графического интерфейса для программ на языке Java. Swing был разработан компанией Sun Microsystems. Он содержит ряд графических компонентов (англ. widgets), таких как кнопки, поля ввода, таблицы и т. д.

Примитивный графический интерфейс был создан с его использованием.

3.2.3 Средства сборки, организация исходного кода

Исходный код организован в типичную для Maven-проектов структуру директорий:
src/main/scala

Исходный код программы
src/main/resources

Описание предметной области на языке PDDL

src/test/scala

Юнит-тесты

src/test/resources

Частичные описания предметных областей, конфигурации нагрузочного тестирования

Классы разбиты в иерархию пакетов, которая явно отделяет внешние API от внутренних. Права доступа заданы соответствущим образом. Базовым является пакет com.glowingavenger.∗.

com.glowingavenger.agent.*

Классы исполнения плана

com.glowingavenger.dialog.*

Примеры реализации диалога с помощью графического интерфейса пользователя

com.glowingavenger.plan.*

Планировщик

impl.* Компоненты построения условного плана

importexport.* Импорт и экспорт

model.* Классы предметной области планировщика: состояния, действия

util.* Утилиты для работы с описанными выше библиотеками

Для сборки используется Maven и/или SBT[5]. В качестве среды разработки – Intellij IDEA, Eclipse.

[5]Simple Build Tool – популярный инструмент в среде Scala.

Глава 4

Использование и тестирование

4.1 Практическое применение программы

Минимальная конфигурация для запуска приложения требует наличия виртуальной машины Java версии не ниже 1.7. Оптимальным выбором является Oracle HotSpot JVM.

Для учебных и исследовательских целей полезно работать напрямую с планировщиком, без стадии выполнения плана. Это требуется например, для проверки корректности его работы. Специальный класс PlanExport позволяет построить план в виде описания графа на языке DOT и его изображения по заданному описанию на языке PDDL.

Доступ к классу осуществляется с помощью командной строки. Необходимо установить утилиту командной строки и сопутствующие библиотеки языка Scala. Следующая команда позволит сгенерировать условный план по описанию, содержащемуся в файле `bulb.pddl`. Результат работы будет помещен в файлы `plan.dot, plan.png`:

```
scala com.glowingavenger.plan.importexport.PlanExport bulb.pddl plan.dot
```

Доступен также примитивный диалоговый агент с пользовательским интерфейсом, построенным на основе диалоговых окон библиотеки Swing. При выполнении плана для каждого вопроса отображается окно вида:

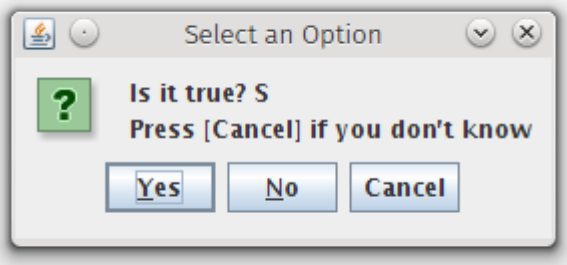

Рисунок 4.1 — Окно для выбора ответа на вопрос

При необходимости выполнить действие выводится соответствующее сообщение:

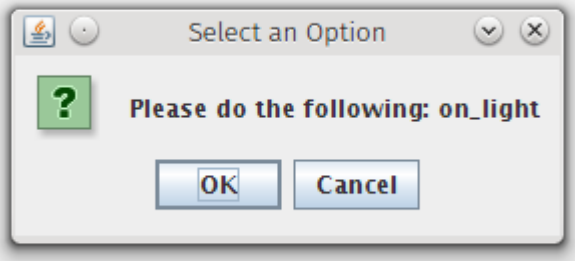

Рисунок 4.2 — Сообщение о необходимости выполнить действие

4.2 Тестирование

4.2.1 Юнит-тесты

FlatSpec + Matchers notation BDD[1]

4.2.2 Нагрузочное тестирование

Для целей нагрузочного тестирование необходимо выполнить две задачи:

1. Сгенерировать случайным образом описание предметной области и проблемы на языке PDDL
2. Сделать замеры работы планировщика и определить узкие места алгоритмов

Генерирование PDDL-описания В качестве параметров используются количество предикатов N, число действий M, число аксиом L.

Введем правила наименования:

- Предикаты: P1, P2, ...
- Действия: A1, A2, ...
- Домены: D1, D2, ...
- Проблемы: PR1, PR2, ...

Алгоритм создания случайного PDDL-описания рассмотрим на примере. Пусть генерируется K-я предметная область, и $N = 4, M = 2, L = 1$. Тогда выполним следующие шаги:

1. Запишем в результирующий файл описание предметной области и перечисление предикатов:

```
( define (domain D1) (: predicates (P1) (P2) (P3) (P4) )
```

2. M описаний действий. Под многоточием будем понимать логические формулы, алгоритм создания которых будет описан далее.

[1]BDD - Behavior Driven Development

```
(: action A1 :precondition (...) :effect (...) )
(: action A2 :precondition (...) :effect (...) )
```

3. L описаний аксиом: `(: axiom (...) )`
4. Описание проблемы

```
( define (problem PR1)
 (: domain D1 )
 (: init (...) )
 (: goal(...) )
)
```

Алгоритм генерирования логических выражений следующий:

1. Выбрать (случайным образом) $n, 1 < n < N$.
2. Найти слуйчаное двоичное число из N знаков с n единицами R_1.
3. Найти случайное троичное число $(0,1,2)$ из n знаков R_2.
4. Из множества всех предикатов $\{P_i\}$ выбрать те, для которых i-я позиция в R_1 равна единице. Обозначим их через $\{P_j\}$.
5. Заменить каждый из выбранных предикатов в соответствии с цифрой на j-й позиции в числе R_2. Так, 0 соответствует P_j, $1 - \neg P_j$, $2 - P_j \vee \neg P_j$ или сокращенно, $P_j?$.
6. Взять конъюнкцию по всем полученным выражениям.

Пример для $N = 4$:

1. $n = 3$
2. $R_1 = 1011$
3. $R_2 = 102$
4. $\{P_j\} = \{P_1, P_3, P_4\}$
5. $P_1 \implies P_1, P_2 \implies \neg P_2, P_3 \implies P_3?$
6. $P_1 \wedge \neg P_2 \wedge P_3?$

Генератор имеет программный интерфейс:

```
1  public interface PDDLGenerator {
2      String[] generate(int predicates, int actions, int axioms);
3  }
```

И может быть вызван из командной строки:

```
~# java com.glowingavenger.highload.PDDLGenerator out.pddl 10 10 3
```

Здесь значения 10, 10, 3 находятся на тех же позициях, что и в методе `generate`, а результат будет записан в файл `out.pddl`.

Измерение скорости работы Бенчмарк. Профайлер

Заключение

Достигнутые результаты

В ходе практической части работы удалось проанализировать и реализовать алгоритм построения условных планов, пользовательский интерфейс к нему, утилиты импорта описания предметной области и экспорта описания графа условного плана.

Еженедельные обсуждения теоретической части позволили обнаружить слабые места алгоритма планирования и найти пути их устранения. Итоговый продукт будет доработан в соответствии с планом работ, представленным ниже, для возможного дальнейшего коммерческого применения.

План дальнейшей работы

Можно выделить следующие компоненты, в которых будет производится дальнейшая разработка:

- Оптимизация алгоритма планирования для обработки случаев, рассмотренных в § 2.2.2
- Повышение производительности и максимальных размеров описания задачи за счет кеширования частей плана. Исследование возможностей генерирования плана по мере развития диалога
- Реализация инструментария для нагрузочного тестирования, описанного в § 4.2.2
- Разработка системы обучения с учителем для создания описаний предметных областей и задач на них без использования языка PDDL.
- Создание интерфейса пользователя в виде веб-приложения

Список иллюстраций

Список таблиц

Список литературы

[1] A. Blum and M.L. Furst. Fast Planning Through Planning Graph Analysis. Number т. 95-221 in Fast planning through planning graph analysis. School of Computer Science, Carnegie Mellon University, 1995.

[2] M. Ghallab, D. Nau, and P. Traverso. Automated Planning: Theory & Practice. The Morgan Kaufmann Series in Artificial Intelligence. Elsevier Science, 2004.

[3] S.J. Russell and P. Norvig. Artificial Intelligence: A Modern Approach. Prentice Hall series in artificial intelligence. Prentice Hall, 2010.

Приложение А. Исходный код алгоритмов

Обратный поиск

```
1  class BackwardSearch(val attrs: List[Symbol], val actions: List[LogicAction], kb: Option[BoolExp]) {
2
3   val axiom = if (kb.isDefined) kb.get else True
4
5   /**
6    * Depth−first search for actions with any models which satisfy a provided goal.
7    * Then constructs a sequence of actions where each next is required to satisfy the condition of the previous one.
8    * The process continues until the space of symbols used in actions' effects and preconditions is closed, that is,
9    * no more new symbols were added for the last required action in the sequence.
10   *
11   * Such space of symbols is then checked individually to be satisfiable against current (initial) state and provided axioms.
12   *
13   * @param goal logic expression which describes a goal in terms of this Agent's symbols.
14   * @return a sub−sequence of this Agent's symbols which are required to be inferred in order to achieve the goal
15   * along with their inferred values (if it's possible)
16   */
17  def search(init: BoolExp, goal: BoolExp): Option[Map[Symbol, Option[Boolean]]] = {
18    val clause = axiom & init & goal
19    retrieveModels(clause) match {
20      case None => backwardAction(init, goal)
21      case Some(model) => retrieveSymbols(model, axiom & init)
22    }
23  }
24
25  private def backwardAction(init: BoolExp, goal: BoolExp): Option[Map[Symbol, Option[Boolean]]] = {
26    val actionModels = for {
27      action <− actions
28      goalClause = axiom & goal & action.effect
29      initClause = axiom & init & action.precond
30      goalModels = retrieveModels(goalClause)
31      initModels = retrieveModels(initClause)
32    } yield (initModels, goalModels, action)
```

```
33      val satGoalModels = actionModels filter (ig => ig._2.isDefined) map (ig => (ig._1, ig._2.get, ig._3))
34      val allSatModels = satGoalModels filter (ig => ig._1.isDefined) map (ig => (ig._1.get, ig._2, ig._3))
35      if (!allSatModels.isEmpty) {
36        retrieveSymbols(allSatModels.head._1, init & axiom)
37      } else if (satGoalModels.isEmpty) {
38        None
39      } else {
40        backwardAction(init, satGoalModels.head._3.precond) match {
41          case None => None
42          case something =>
43            something
44        }
45      }
46    }
47  }
```

Построение плана

Listing 1 — Axioms

```
1  trait Axioms {
2    /**
3     * Produces a function which chooses the axiom from the list for given belief states
4     * @param axioms list of axioms to choose from
5     * @return axiom boolean expression
6     */
7    protected def axiom(axioms: Iterable[BoolExp]): (Iterable[BeliefState] => BoolExp)
8
9    protected def axiom(axioms: BoolExp*): (Iterable[BeliefState] => BoolExp) = axiom(axioms)
10
11   protected def chooseAxiom(axioms: Iterable[BoolExp])(states: BeliefState*): BoolExp = axiom(axioms)(states
12
13   protected def chooseAxiom(axioms: BoolExp*)(states: BeliefState*): BoolExp = axiom(axioms)(states)
14
15   /**
16    * Applies the axiom to given states using conjunction
17    */
18   protected def applyAxiom(clauses: BeliefState*): BeliefState
19 }
20
21 /**
22  * Always returns a conjunction of all axioms.
```

```
23   * A naive approach that would actually work only with a single axiom
24   */
25  trait AxiomConjunction extends Axioms {
26    protected def axiom(axioms: Iterable[BoolExp]): (Iterable[BeliefState] => BoolExp) = {
27      val axiom = if (axioms.isEmpty) True else (axioms.head /: axioms.tail)(_ & _)
28      stateList => axiom
29    }
30  }
31
32  /**
33   * We have this in a separated trait to simplify unit testing
34   */
35  trait DefaultAxioms extends Axioms with AxiomConjunction with ProblemAware {
36    protected val problemAxiom = super.chooseAxiom(problem.domain.axioms)_
37
38    protected def applyAxiom(states: BeliefState*): BeliefState = {
39      if (states.isEmpty)
40        BeliefState(problemAxiom(Nil))
41      else
42        BeliefState((problemAxiom(states) /: states)(_ & _.toExpr), states.map(_.predicates.keys).flatten)
43    }
44  }
```

Listing 2 — Successors

```
1   trait Successors {
2     def successors(state: BeliefState): (List[ActionEdge], List[BeliefState]) = (Nil, Nil)
3   }
4
5   trait QuestionSuccessors extends Successors with Axioms {
6     override def successors(state: BeliefState): (List[ActionEdge], List[BeliefState]) = {
7       val (edges, states) = super.successors(state)
8       val questions = state.unknown.map(Question).map {
9         question =>
10          question.result(state) match {
11            case a: Answer =>
12              ActionEdge(state, applyAxiom(a.yes), question) ::
13                ActionEdge(state, applyAxiom(a.no), question) :: Nil
14            case _ => Nil
15          }
16      }.flatten
17      (edges ::: questions, states ::: questions.map(_.to))
18    }
19  }
```

Listing 3 — Path Successors

```
trait PathSuccessors extends Successors with Axioms with ProblemAware {
  protected val search = new ASearch[(BeliefState, Action)] {
    override def isGoal(node: (BeliefState, Action), goal: (BeliefState, Action)): Boolean = isSearchGoal(node, g

    override def hCost(init: (BeliefState, Action), goal: (BeliefState, Action)): Int = 0

    override def cost(node: (BeliefState, Action)): Int = 1

    override def successors(node: (BeliefState, Action)): List[(BeliefState, Action)] = searchSuccessors(node)
  }

  protected def searchSuccessors(node: (BeliefState, Action)): List[(BeliefState, Action)]

  protected def isSearchGoal(node: (BeliefState, Action), goal: (BeliefState, Action)): Boolean =
    goal._1 includes node._1

  protected def path2Edges(path: List[(BeliefState, Action)]): List[ActionEdge] = {
    if (path.isEmpty || path.tail.isEmpty)
      Nil
    else
      ActionEdge(path.head._1, path.tail.head._1, path.tail.head._2) :: Nil ::: path2Edges(path.tail)
  }

  protected def makePath(state: BeliefState, producer: Action = NoAction()) = {
    search.search((state, producer), (BeliefState(problem.goal), NoAction())) match {
      case Some(path) => path2Edges(path)
      case None => Nil
    }
  }

  protected def pathStates(path: List[ActionEdge]): List[BeliefState] = Nil

  override def successors(state: BeliefState): (List[ActionEdge], List[BeliefState]) = {
    val (edges, states) = super.successors(state)
    val path = makePath(state)
    (edges ::: path, states ::: pathStates(path))
  }
}

trait MixedPathSuccessors extends GuaranteedPathSuccessors {
```

```
41    private val questionSuccessors = new QuestionSuccessors {
42      override protected def applyAxiom(clauses: BeliefState*): BeliefState = MixedPathSuccessors.this.applyAxiom(clauses:
43
44      override protected def axiom(axioms: Iterable[BoolExp]): (Iterable[BeliefState]) => BoolExp =
45        MixedPathSuccessors.this.axiom(axioms)
46
47      def searchSuccessors(node: (BeliefState, Action)): List[(BeliefState, Action)] = {
48        successors(node._1)._1 map {
49          case ActionEdge(f, t, a) => (t, a)
50        }
51      }
52    }
53
54    override protected def searchSuccessors(node: (BeliefState, Action)): List[(BeliefState, Action)] = {
55      super.searchSuccessors(node) ::: questionSuccessors.searchSuccessors(node)
56    }
57
58    override protected def makePath(state: BeliefState, producer: Action = NoAction()) = {
59      val bestPath = super.makePath(state, producer)
60      val answerBranches = bestPath collect {
61        case ActionEdge(f, t, q: Question) =>
62          q.result(f) match {
63            case a: Answer if a.yes includes t => ActionEdge(f, applyAxiom(a.no), q)
64            case a: Answer => ActionEdge(f, applyAxiom(a.yes), q)
65          }
66      }
67
68      bestPath ::: answerBranches
69    }
70
71    override protected def pathStates(path: List[ActionEdge]): List[BeliefState] = path collect {
72      // Need to process vertices with more than one unknown attribute
73      case ActionEdge(_, to, _) if !to.unknown.isEmpty => to
74    }
75  }
76
77  trait GuaranteedPathSuccessors extends PathSuccessors {
78    protected def searchSuccessors(node: (BeliefState, Action)): List[(BeliefState, Action)] = {
79      for {
80        action <- problem.domain.actions
81        result = action.result(node._1)
82        if result != node._1
83      } yield (applyAxiom(result), action)
```

```
84    }
85  }
```

Импорт/Экспорт

Listing 4 — PDDLParser

```
1   private[importexport] object PDDLParser extends StandardTokenParsers with PackratParsers {
2
3     override val lexical = new PDDLLexical
4
5     lazy val domain = ("(" ~> "define" ~> "(" ~> "domain" ~> ident <~ ")") ~ predicatesDef ~ rep(structure
6       case name ~ predicates ~ structures =>
7         val actions = new ArrayBuffer[LogicAction]()
8         val axioms = new ArrayBuffer[BoolExp]()
9         structures foreach {
10          case Left(action: LogicAction) => actions += action
11          case Right(axiom: BoolExp) => axioms += axiom
12        }
13        Domain(predicates, actions.toList, axioms.toList, name)
14    }
15
16    lazy val predicatesDef = ("(:" ~> "predicates") ~> (rep(predicate) <~ ")")
17
18    lazy val predicate = "(" ~> ident <~ ")" ^^ {
19      Symbol(_)
20    }
21
22    lazy val structureDef = actionDef ^^ {
23      Left(_)
24    } | axiomDef ^^ {
25      Right(_)
26    }
27
28    lazy val actionDef = ("(:" ~> "action" ~> ident) ~ (actionBody <~ ")") ^^ {
29      case name ~ body => LogicAction(body._1, body._2, name)
30    }
31
32    lazy val actionBody = (":" ~> "precondition" ~> logicExpr) ~ (":" ~> "effect" ~> logicExpr) ^^ {
33      case init ~ goal => (init, goal)
34    }
35
```

```
36  lazy val axiomDef = ("(:" ~> "axiom") ~> (logicExpr <~ ")")
37
38  lazy val logicExpr = "(" ~> formula <~ ")"
39
40  lazy val problem = ("(" ~> "define" ~> "(" ~> "problem" ~> ident <~ ")") ~ ("(:" ~> "domain" ~> ident <~ ")") ~
41    case name ~ domainName ~ init ~ goal => UnboundProblem(domainName, init, goal, name)
42  }
43
44  lazy val initDef = ("(:" ~> "init") ~> logicExpr <~ ")"
45
46  lazy val goalDef = ("(:" ~> "goal") ~> logicExpr <~ ")"
47
48  lazy val pddl = domain ~ problem ^^ {
49    case d ~ p => Problem(p, d)
50  }
51
52  /**
53   * Copied from BooleanFormulaParserCombinator
54   */
55
56  lazy val formula: PackratParser[BoolExp] = term ~ ("&" ~> formula) ^^ {
57    case f1 ~ f2 => f1 & f2
58  } | term ~ ("|" ~> formula) ^^ {
59    case f1 ~ f2 => f1 | f2
60  } | term ~ ("->" ~> formula) ^^ {
61    case f1 ~ f2 => f1 implies f2
62  } | term ~ ("<->" ~> formula) ^^ {
63    case f1 ~ f2 => f1 iff f2
64  } | term
65
66  lazy val term = "~" ~> "(" ~> formula <~ ")" ^^ {
67    case f => f.unary_~()
68  } | "(" ~> formula <~ ")" | lit
69
70  lazy val lit: PackratParser[BoolExp] = "~" ~> ident ^^ {
71    case s => Symbol(s).unary_~()
72  } | unknown | ident ^^ {
73    case s => Symbol(s): BoolExp
74  }
75
76  lazy val unknown = ident <~ "?" ^^ {
77    case s => Symbol(s) | ~Symbol(s): BoolExp
78  }
```

```
79 }
80
81 private[importexport] class PDDLLexical extends StdLexical {
82   delimiters +=("(", "(:", ")", "=", "&", "|", "~", "->", "<->", "?", ":")
83   reserved +=("define", "domain", "predicates", "actions", "axiom", "problem", "init",
84     "goal", "action", "effect", "precondition")
85
86   override def whitespace: Parser[Any] = rep(whitespaceChar | comment)
87
88   protected override def comment: Parser[Any] = ';' ~ rep(chrExcept(EofCh, '\n'))
89
90   override def identChar = super.identChar | elem('-')
91 }
```

Выполнение плана

Listing 5 — PlanExecutor

```
1  trait PlanTraversalListener {
2    def onStateChange(before: Option[BeliefState], state: BeliefState,
3                      producer: Option[ActionEdge], successors: Iterable[ActionEdge]): ActionEdge
4
5    def onFinish(state: BeliefState, producer: Option[ActionEdge])
6    def onFailure(state: BeliefState, producer: Option[ActionEdge])
7  }
8
9  class PlanExecutor(problem: Problem, listener: PlanTraversalListener) {
10   def exec() {
11     val plan = ContingencyPlan(problem)
12
13     def doAct(before: Option[BeliefState], state: BeliefState, producer: Option[ActionEdge]) {
14       if (problem.goal includes state) {
15         listener.onFinish(state, producer)
16       } else {
17         val successors = plan.plan <<>> state
18         if (successors.isEmpty) {
19           listener.onFailure(state, producer)
20         } else {
21           val newProducer = listener.onStateChange(before, state, producer, successors)
22           val newState = newProducer.to
23           doAct(Some(state), newState, Some(newProducer))
24         }
```

```
25          }
26        }
27
28        doAct(None, plan.init, None)
29      }
30    }
```

Приложение Б. Результаты выполнения программы

Мир Лампы

Описание на языке DOT

```
digraph G {
  1 [ label="(-S -L B)" ];
  2 [ label="(S L B)" ];
  3 [ label="(S L? B?)" ];
  4 [ label="(S -L -B)" ];
  5 [ label="(S? -L B?)" ];
  6 [ label="(-S -L B?)" ];
  7 [ label="(S? -L -B)" ];
  8 [ label="(-S -L -B)" ];
  1 -> 2 [ label="on_light" ];
  3 -> 4 [ label="~B" ];
  5 -> 4 [ label="S" ];
  5 -> 1 [ label="B" ];
  5 -> 3 [ label="on_light" ];
  3 -> 2 [ label="B" ];
  5 -> 6 [ label="off_light" ];
  6 -> 1 [ label="ch_bulb" ];
  5 -> 6 [ label="~S" ];
  5 -> 7 [ label="~B" ];
  7 -> 8 [ label="off_light" ];
  7 -> 4 [ label="S" ];
  8 -> 1 [ label="ch_bulb" ];
  7 -> 8 [ label="~S" ];
  3 -> 2 [ label="L" ];
  6 -> 1 [ label="B" ];
  6 -> 3 [ label="on_light" ];
```

```
  6 -> 8 [ label="~B" ];
  3 -> 4 [ label="~L" ];
  4 -> 8 [ label="off_light" ];
  3 -> 6 [ label="off_light" ];
}
```

Изображение графа