Содержание

Аннотация

Дипломная работа посвящена разработке системы решения задач, основанной на графе планирования. Актуальность поставленной в работе задачи подтверждается тем, что исследования по проблемам планирования играют центральную роль в искусственном интеллекте практически со времени появления этого научного направления, а статьи по планированию занимают основной объём ведущих журналов и материалов конференций по искусственному интеллекту.

Представленная в данной дипломной работе система не является предметно-ориентированной и может находить решение задач из различных проблемных областей. Постановки задач и описания проблемных областей система получает в виде предложений на подмножестве языка PDDL (Planning Domain Definition Language). Результатом работы системы является план решения задачи, представляющий собой последовательность действий по достижению искомой цели. Поиск решения задачи осуществляется в пространстве состояний с использование эвристических знаний, извлекаемых из, так называемого, графа планирования. Работа системы решения продемонстрирована на задачах различной степени сложности из разных предметных областей.

Введение

Одним из основополагающих направлений исследований в области искусственного интеллекта (ИИ) является планирование. Планированием называется процесс выработки последовательности действий, позволяющих достичь цели[4].Конечно, многие из задач ИИ, даже сложные, можно решать путём полного перебора всевозможных вариантов и выбрать подходящий по определённым критериям план. Однако в достаточно масштабных задачах такой подход часто не годится; в связи с этим интересны алгоритмы, которые используют эффективные методы сокращения перебора вариантов решений.

При этом создаются системы решения задач (решатели), которые, работая над проблемной областью и задачей над ней без участия человека, представляют пользователю план решения задачи в определённом для данной системы виде.

Важно, планирование решений каких задач, могут осуществлять данные решатели. Поэтому интересны классификации проблемных областей. Можно поделить проблемные области на монотонные и немонотонные. Так, в случае действия принципа монотонности (если на некотором шаге вывода получено утверждение, то его истинность на последующих шагах вывода не может изменяться), говорят о монотонной предметной области. Иначе - о немонотонной. Например, планиметрия - монотонная предметная область. А если рассмотреть задачу «Обезьяна и бананы», то можно показать, что она относится к задаче немонотонной проблемной области. Задача состоит в том, чтобы обезьяна собрала все бананы в комнате, повешенные к потолку. Для этого она может перемещаться по комнате, передвигать ящик, забираться на него, хватать банан с ящика. Поскольку факт нахождения обезьяны в какой-либо точке - факт непостоянный (обезьяна может перемещаться по комнате), то получается, что не выполняется принцип монотонности.

Также имеет смысл говорить о том, нужны ли какие-либо вычисления или нет в мире, который мы рассматриваем. То есть, должен ли решатель производить какие-то арифметические операции, вычислять значения функций, или же – нет. Так, если ввести дополнительный параметр в предыдущей задаче – количество собранных бананов, то можно в цели задачи поставить цель, например, собрать 5 бананов.

Соответственно, при таком подходе решатели должны уметь производить нужные вычисления.

Далее, есть отдельное направление развития систем решения задач – темпоральные системы. В данных системах учитывается такой параметр, как время. Если его ввести для задачи с обезьяной и бананами, то можно ставить цель задачи так: собрать все бананы за определённое время, учитывая то, что все действия имеют затраты по времени.

Данные деления на группы, позволяет выделить основные части систем решения задач. Первая часть отвечает за формальное представление задачи, домена, назовём её «База знаний». Вторая – за реализацию плана решения задачи, назовём её «Планировщик». К каждой из этих двух частей можно предъявить какие-то требования.

Для реализации первой части обычно используется какой-либо язык описания. Требования к данным языкам, как и к языкам программирования, достаточно объективны и просты: выразительность, удобство, простота, понятность.

Основным подходом решения задачи планирования является подход, основанный на использовании графовых структур. В данном случае за структурную основу алгоритма берётся какой-либо граф. В нём вершины символизируют состояния или части состояний системы, а дуги регламентируют переходы и связи между ними. Постольку поскольку, работа с графами сложна, да и даже в простых задачах данные структуры быстро разрастаются, то задача построения или же выбора алгоритма серьёзно сказывается на скорости работы систем решения задач.

Алгоритм может быть основан на использование эвристической информации – то есть кроме задачи и домена алгоритму даётся информация о том, какое действие лучше применять в определённой ситуации. Зачастую, это какая-то количественная характеристика действия. Данный подход помогает сократить объём перебора. Есть алгоритмы, которые перебирают практически все варианты, т.е. не используют никакой эвристической информации, они различаются по стратегии перебора.

Алгоритмы планировщиков выдают план решения задачи, который может быть оптимальным по какому-либо признаку. Алгоритмы планирования можно разделить на те, которые реализуют оптимальный план, и те, которые не реализуют.

Предполагается разработать систему решения задач, на вход которой даётся формальное описание задачи, на выходе же реализуется план решения задачи. При этом система должна быть независимой от проблемной области.

1 Постановка задачи

Основной целью данной работы является создание системы решения задач. Создаваемая система не должна быть ориентированной на конкретную проблемную область, это означает, что помимо условий задачи на вход системы подаётся описание проблемной области (домена задачи). Задачи и домены описываются на входном языке, в качестве которого следует использовать какой-либо из современных языков планирования или его подмножество.

Домен задачи должен представлять собой детерминированную событийно-дискретную систему без внутренней динамики, то есть тройку D = (S, A, γ), где:

- $S = \{s_1, s_2 \ldots . s_n\}$ конечное множество состояний;
- $A = \{a_1, a_2, \ldots a_m\}$ конечное множество действий;
- γ: S x A -> S функция переходов между состояниями, описывающая в какое состояние переходит система D из текущего состояния при выполнении некоторого действия. Домен задачи также должен быть прозрачен, то есть в любом состоянии домена система планирования располагает полными сведениями о нём. Течение времени в домене должно быть неявным, то есть действия не имеют длительности.

Поставим задачу планирования в домене D, обладающем указанными свойствами: по начальному состоянию $s_0 \in S$, множеству целевых состояний $Sg \subseteq S$, найти план – последовательность действий $(a_{i1}, a_{i2} \ldots a_{ik})$, где $a_{ij} \in A$ такую, что соответствующая последовательность состояний $(s_o, s_1, \ldots, s_k)$ удовлетворяет условиям $s_1 = \gamma(s_o, a_{i1})$, $s_2 = \gamma(s_1, a_{i2})$, ... $s_k = \gamma(s_{k-1}, a_{ik})$, и $s_k \in Sg$.

В рамках дипломной работы найденный план считается оптимальным, если в нём количество действий минимально. Поиск оптимального плана в системе, не зависящей от предметной области, в худшем случае требует полного перебора, который занимает неприемлемо большое время для сложных задач. Исходя из этих особенностей, разрабатываемая система планирования должна находить план, близкий к оптимальному, в котором количество действий имеет тот же порядок, что и в оптимальном плане.

2 Обзор существующих решений

2.1 Языки формального описания проблемных областей и задач

Существует несколько языков формального описания задач и проблемных областей. Среди них: STRIPS[10], ADL[11], PDDL[8].

Именно решателю STRIPS[10] мы обязаны современной терминологией в языках описания действий и доменов. Было введено структурное понятие для описания мира, а именно, понятие состояния. Таким образом, предложена идея моделировать изменения в мире, как переходы мира из одного состояния в другое.

Итак, в любой момент времени мир находится в некотором состоянии. Действия могут переводить мир в другие состояния. Состояние в STRIPS представляет собой множество фактов о конкретном моменте эволюции мира, записанных при помощи формул исчисления предикатов первого порядка. Все, что постулируют эти формулы, а также все, что выводимо из этих формул, считается истинным в данном состоянии мира (в данный момент эволюции мира). Все прочие утверждения считаются ложными. Все атомарные формулы во множестве формул, описывающих состояние, не должны иметь свободных переменных.

Действия описываются при помощи специальных конструкций, состоящих из трех элементов: имени, предусловия и эффекта. Предусловие позволяет определить, возможно ли применение действия в некотором состоянии и задается означенной формулой логики предикатов первого порядка. Если эта формула выводима из формул, входящих в описание состояния, то действие считается применимым в данном состоянии. Эффект описывается двумя списками, содержащими означенные формулы логики предикатов первого порядка. Один из списков называется списком удаления (delete list). Он состоит из формул, которые станут гарантированно ложными в новом состоянии (полученном вследствие применения действия). Другой список, называемый списком добавления (add list), содержит формулы, которые станут гарантированно истинными в новом состоянии. Все формулы, которые содержатся в состоянии, в котором действие применяется, но не упоминающиеся в эффекте действия, будут содержаться и в результирующем состоянии действия. Это последнее утверждение в литературе называют STRIPS-

допущением. STRIPS-допущение разрешает проблему фрейма, детерминируя, что изменится после выполнения действия, а что останется неизменным.

Действия группируются в классы действий, называемые *схемами действий*. Именно схемы действий использовались в STRIPS для описания домена. Схема действия — это абстракция действия, в которой константы в описании предусловия и эффекта заменены (частично или полностью) параметрами. Подстановка в схему действия конкретных значений вместо параметров являет собой обратный переход (от схемы к действию). Параметризация дает нам сжатое описание целой группы действий.

Однако, после создания данного языка, у него выявилось множество недостатков, как формальных, так и выразительных. Первый момент обусловлен необходимостью удалять неатомарные формулы и использованием метасимвола "$" в симметричных предикатах. Второй момент связан с тем, что не было краткости в описании сложных проблем.

На помощь пришёл новый язык ADL, в котором были исправлены недостатки языка STRIPS.

Отличия ADL от STRIPS:

- В ADL предлагается представлять состояния при помощи множества позитивных и негативных литералов. Т.е. здесь явно указывается, какие свойства мира истинны, а какие ложны. Если в STRIPS мы считали, что все что не упомянуто в состоянии считается ложным, то здесь мы полагаем, что неупомянутые литералы имеют неопределенное значение истинности. Т.е. в ADL мы отходим от допущения о замкнутости мира.
- Эффект не разделяется на список добавления и удаления. Литералы, на которые производится воздействие, представляются в эффекте конъюнкцией литералов.. Если в эффекте записана формула (P & ¬Q), то это значит, что в новое состояние будут добавлены литералы P и ¬Q, а также будут удалены ¬P & Q.
- К числу достоинств ADL причисляют также возможность использовать в описании цели квантифицированные и дизъюнктивные формулы. В STRIPS явно не указывается, что их использование запрещено, тем не менее, мы отметим и этот момент.

- В ADL появилась возможность задавать условные эффекты действий. Это можно сделать при помощи конструкции when P: E, которую можно трактовать так. Если P истинно в текущем состоянии, то эффект E будет произведен действием. Наряду с условным эффектом, могут быть другие эффекты действия, которые будут иметь место в любом случае, если действие будет применено.
- Предложено считать предикат (x = y) элементом языка.
- Переменные и параметры в формулах могут быть типизированы. Это может существенно сократить множество возможных подстановок и здорово сказаться на эффективности.

Некоторое время ADL пользовался популярностью, хотя наряду с ним разрабатывались и другие языки. Но вскоре появился новый, еще более мощный язык, который является, по сей день, стандартом для описания задач планирования. Речь идет о языке PDDL.

Язык PDDL появился в 1998 году в ответ на потребность разработки единого языка для всех видов планировщиков. Потребность такая возникла в связи с предложением проводить соревнования планировщиков, чтобы эмпирически оценить производительность и эффективность тех или иных методов планирования.

При помощи PDDL можно описывать домены планирования и задачи планирования (привязанные к определенным доменам). Каждое такое описание оформляется отдельным файлом. Т.е. описание домена физически отделено от описания задачи. Следовательно, легко можно описать несколько задач для одного и того же домена.

Описание домена планирования включает в себя:

- типы объектов, которые могут фигурировать в описании задачи планирования;
- перечень отношений между объектами – набор предикатов;
- константы предметной области;
- доменные функции, которые ставят в соответствие некоторому множеству объектов домена числовую величину;
- определения схем действий (то же самое, что и в STRIPS);
- определения схем аксиом, под средством которых можно выводить справедливость одних фактов из справедливости других (например, можно

написать такую аксиому для какой-нибудь задачи: если один объект лежит на другом, то он находится выше другого);

- определять длительность у действий (для темпоральной логики);

Условия задачи содержит:

- указание на домен, к которому относится эта задача;
- перечень объектов;
- начальное состояние;
- целевое состояние.

Разработчики языка хотели, чтобы PDDL мог использоваться любым планировщиком. Но каждый планировщик умеет работать только с конструкциями определенной сложности (для которых он спроектирован). Чтобы планировщик мог быстро определить, может ли он работать с предложенным ему описанием домена, каждое такое описание объявляет требования к способностям планировщика.

Таким образом, язык PDDL – это достаточно гибкое средство формального описания задач и доменов предметных областей, которое поддерживается большинством современных планировщиков.

2.2 Основные подходы планирования, основанные на графах

Далее рассмотрим различные алгоритмические подходы планирования, проблемы, связанные с ними, и структуры, которые задействованы в их работе.

Существует несколько основных идей в планировании, основанном на графах. Первый – метод поиска в пространстве состояний. Рассмотрим его. Есть множество действий и множество состояний, из которых выделены состояния: начальное (ему соответствует начальные данные задачи) и конечное (ему соответствует цель задачи). Применяя действие к какому-нибудь состоянию, мы получаем новое состояние, к которому опять можно применить действие, получив очередное состояние и т.д. Таким образом, у нас появляется пространство состояний – совокупность состояний, в которых может находиться рассматриваемый нами мир.

Это пространство состояний можно отобразить на граф. Вершины будут соответствовать состояниям из пространства состояний, направленные дуги – действиям. Тогда алгоритмы, основанные на данном методе, работают с данным

графом – строят его, ищут в нём путь от вершины, ассоциируемой с начальным состоянием, до вершины, соответствующей конечному состоянию. Действия, ассоциирующиеся с дугами в данном пути, будут составлять план решения задачи.

В зависимости от того, как идёт построение данного графа, строится в реальности полноценный граф или же – дерево, используется ли эвристическая информация или – нет, можно говорить о различных методах поиска в пространстве состояний.

Для метода поиска в пространстве состояний известно множество простых алгоритмов поиска решения на деревьях: методы полного перебора, перебора в глубину[4]. Однако в данных алгоритмах, из-за простоты выбора отсечения ненужных ветвей поиска решения, поиск производится практически во всём пространстве состояний. Данный подход годится лишь для небольшого круга задач, в которых пространство состояний достаточно не велико.

Во многих задачах пространство состояний огромно. Поэтому нет никакой возможности использовать простые переборные методы для построения плана решения. Чтобы выйти из сложившейся ситуации, используют эвристическую информацию для уменьшения объёма перебора. Построение графа такими алгоритмами идёт в заведомо приоритетную сторону. Конечно, возникают проблемы. Первая – найти эвристику. Вторая - если эвристика получена эмпирическими путями, не программно, то данный подход становится предметно ориентированным. То есть, при смене проблемной области придётся найти заново и эвристику.

Алгоритм A*[4] использует эвристические оценки для отсечения ненужных ветвей перебора и также основан на работе с деревом.

Пусть n – вершина графа пространства состояний. Предположим, что через эту вершину проходит путь от начальной вершины к целевой. Тогда f(n) – оценочная функция, f(n)=g(n)+h(n), где g(n) – цена минимального пути от начальной вершины до вершины n, равная сумме цен дуг данного пути (с каждым действием ассоциирована цена, соответствующая дуге); h(n) – эвристическая оценка длины пути от вершины n до целевой вершины. Далее изложен алгоритм A*:

1. Поместить начальную вершину s в список, называемый ОТКРЫТ, и вычислить f(s).
2. Если список «ОТКРЫТ» пуст, то на выход даётся сигнал о неудаче.

3. Взять из списка «ОТКРЫТ» вершину с минимальным значением f и поместить её в список «ЗАКРЫТ». Дать имя n. (В случае совпадения значений f для нескольких вершин выбирать вершину произвольно, но отдавать предпочтение целевой вершине.)
4. Если n – целевая вершина, то на выход подать решающий путь, получаемый прослеживанием соответствующих указателей.
5. Раскрыть вершину n, построив все непосредственно следующие за ней вершины (если таковых нет, то перейти на шаг 2). Для каждой дочерней вершины n_i вычислить значение $f(n_i)$.
6. Связать с вершинами n_i, которых нет в списках «ОТКРЫТ» и «ЗАКРЫТ» только что вычисленные значения $f(n_i)$. Поместить эти вершины в список «ОТКРЫТ» и провести от n к ним указатели.
7. Связать с теми из непосредственно следующих за n вершинами, которые уже были в списках «ОТКРЫТ» и «ЗАКРЫТ», меньшее из прежних и только что вычисленных значений f. Поместить в список «ОТКРЫТ» те из непосредственно следующих за n вершин, для которых новое значение f оказалось ниже, и изменить направление указателей для всех вершин, для которых значение f уменьшилось, направив их к n.
8. Перейти на шаг 2.

Для данного алгоритма доказана теорема:

Теорема 1: Если для всех вершин n h(n) – нижняя граница минимальной цены пути из n в целевую вершину, то алгоритм A* строит оптимальный план.

Рассмотрим пример работы. Задача – «Пятнашки». Для простоты изображения возьмём поле игры 3x3. В элементах поля, кроме одного, находятся числа от 1 до 8 без повторений. Один элемент пуст. Есть действие перемещения пустого элемента влево, вправо, вверх, вниз. При этом предусловием к нему будет факт того, что движение это возможно, т.е. пустой элемент не выйдет за рамки поля. Цель игры – с помощью данного набора действий привести поле в такое состояние, когда все его элементы упорядочены по возрастанию слева направо и сверху вниз, за исключением пустого элемента, который должен находиться в нижнем правом углу поля. Состояние будет представлять собой текущее положение элементов (фишек) в поле игры, и соответствовать вершине дерева. Алгоритм поиска: A*. Цена любого действия равна 1. Эвристическая функция вершины равна количеству фишек,

расположенных не на своих местах, в состоянии, ассоциированном с данной вершиной. На рисунке 1 показано дерево, которое получено в результате 2 итераций алгоритма.

В прямоугольниках изображены вершины дерева, которые ассоциируются с состояниями (внутри – положение фишек на поле). Стрелки регламентируют действия, применённые к состояниям, ассоциированным с вершинами из которых стрелки выходят. Соответственно, вершина, в которую стрелка входит, ассоциируется с состоянием, что получается при применении соответствующего действия. В кружочках находятся цены вершин, которые расположены рядом. Жирными линиями выделены вершины, которые раскрывались.

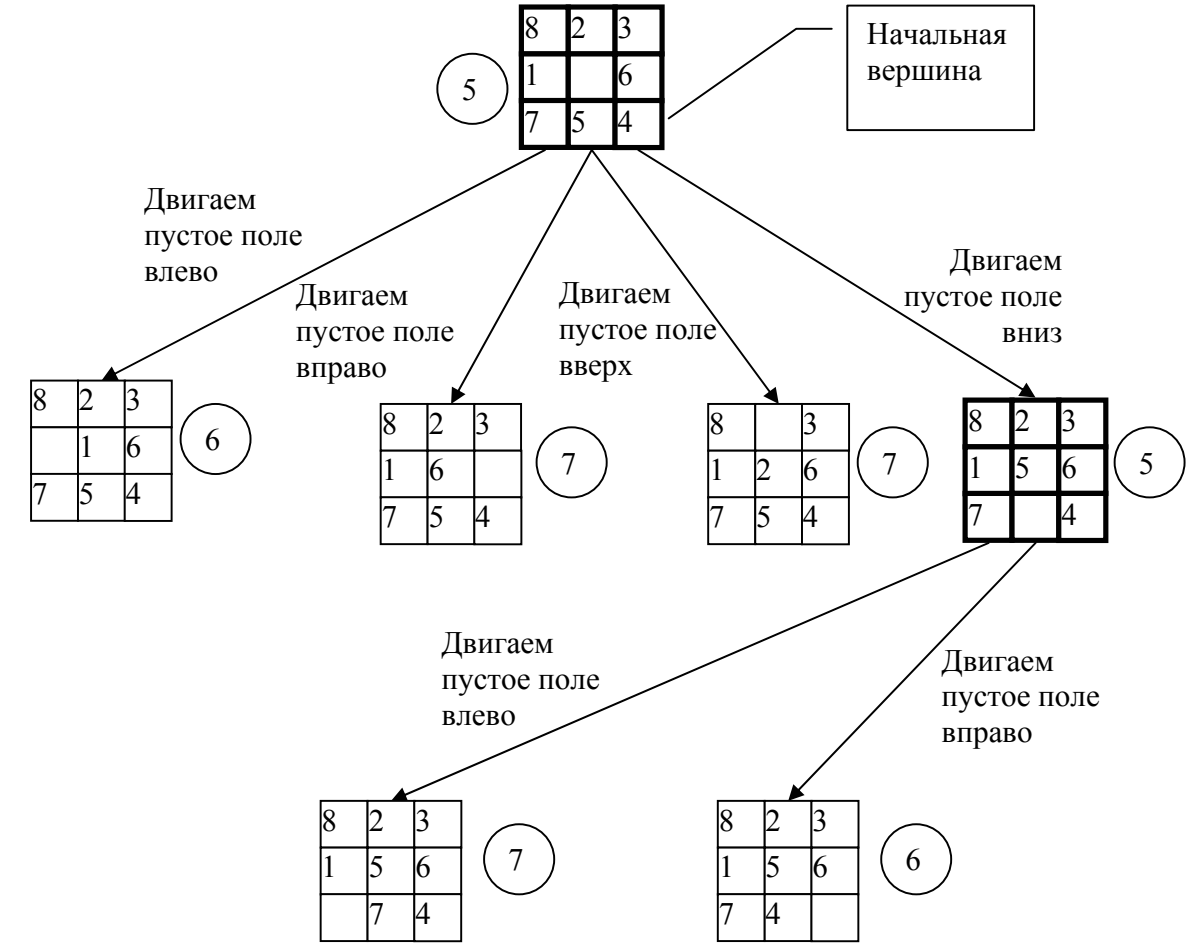
Начальную вершину помещаем в список «ОТКРЫТ», считаем цену. Она равна 5. Далее берём начальную вершину и раскрываем, получая четыре новых вершины. Считаем для них цены, помещаем в список «ОТКРЫТ» (таких вершин нет ни в списке «ОТКРЫТ», ни в списке «ЗАКРЫТ»), проводим дуги. На этом первая итерации закончена.

Далее выбирается вершина с минимальной ценой (крайняя правая во втором ряду). Происходит раскрытие. По идее нужно добавить в дерево 3 вершины (так как одно действие нельзя применить), но добавляется 2: одна из получаемых вершин уже существует. При этом новая цена больше старой, поэтому цена не меняется, не меняются также и направление указателей.

Планом решения же будет последовательность действий соответствующая действиям в пути, принадлежащему дереву, ведущему из начальной вершины в терминальную вершину.

Зачастую определённым алгоритмом и нельзя решить определённую задачу: так, в примере, если бы использовался алгоритм перебора в глубину, то, скорее всего, никогда бы не было найдено решения, потому что по циклу бы добавлялся определённый набор вершин. То есть круг задач, решаемый некоторыми алгоритмами, достаточно ограничен.

Однако несомненным плюсом является то, что данные алгоритмы достаточно просты и понятны.



Условные обозначения:

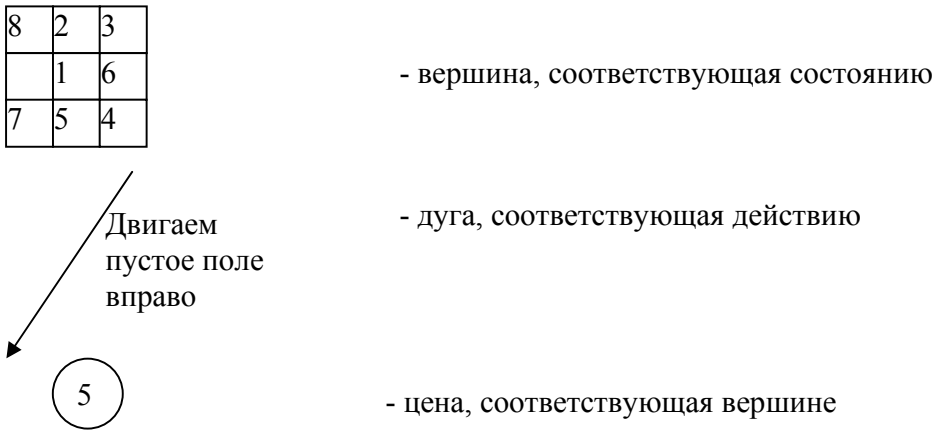


Рисунок 1 Пространство состояний для задачи «Пятнашки».

Рассматривая алгоритмы, использующие какие-то эвристические оценки для отсечения ненужных ветвей перебора, то встаёт разумный вопрос о том, где данные

оценки брать. Можно ограничить круг решаемых задач и просто-напросто вложить определённые оценки, основанные на априорных оценках человека, в алгоритм поиска решения (так в стандартных алгоритмах для игры «Шахматы» зачастую и делают). Однако данный подход слишком урезает возможности системы поиска решения задач, и он не годится.

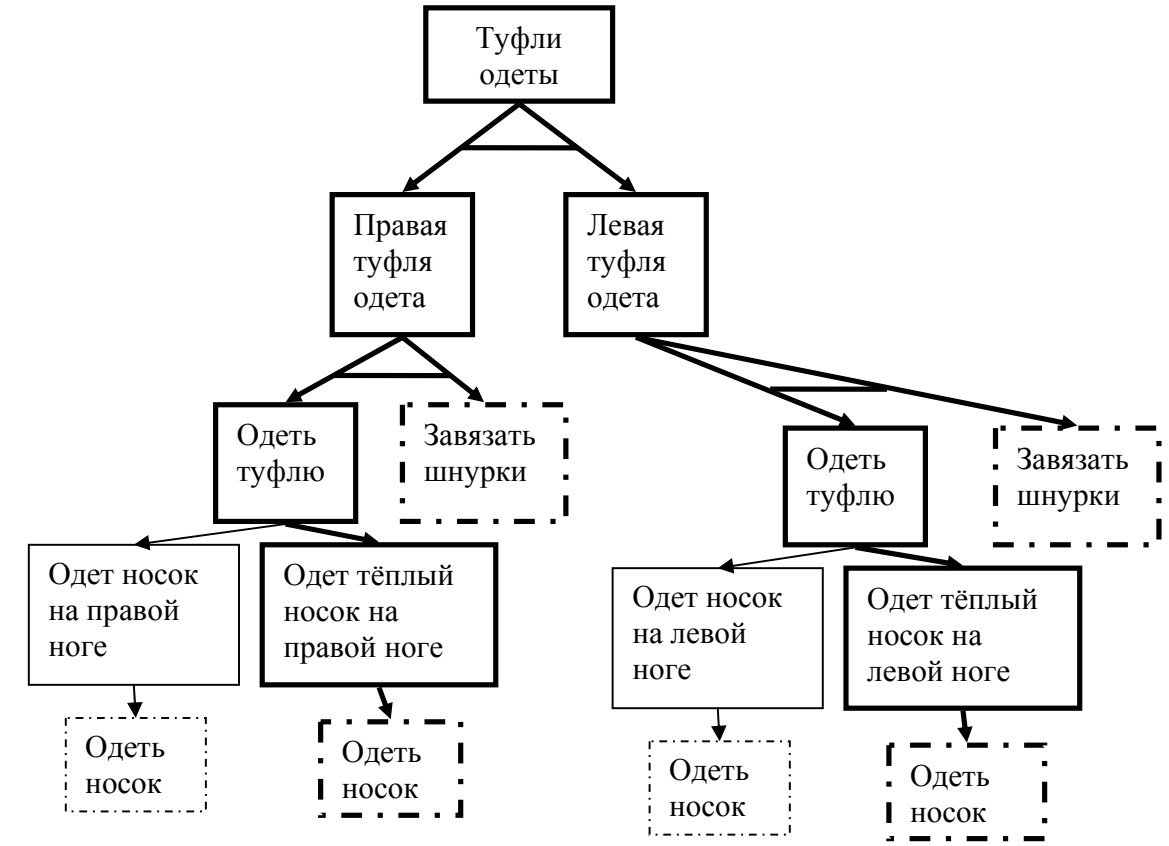
Возможен такой вариант: использование какого-нибудь другого алгоритма для вычисления данных оценок, а решение уже строить алгоритмом, основанным как раз на этих оценках.

Можно подойти к решению задачи боле интересным подходом: сводить цели задачи к более простым подцелям – производить редукцию целей. Постепенно сводя полученные подцели к более простым подцелям, получаем элементарные цели, которые соответствуют задачам, решение которых известно.

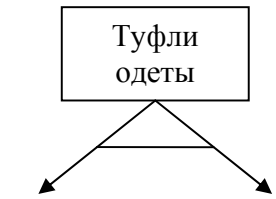
В данном случае идёт разговор об алгоритмах поиска, использующие И/ИЛИ деревья: метод полного перебора, поиск в глубину [4]. Данные алгоритмы большей частью используются для обратного хода поиска решения задач, разбиения задачи на подзадачи и, зачастую, как и все алгоритмы работы с И/ИЛИ графами, требуют от предметной области монотонности или же дополнительной информации о целях – упорядоченности.

Для примера возьмём задачу надевания пары туфель. Есть действия по надеванию туфли и носка, завязыванию шнурков. Цель задачи – одеть туфли. Пример И/ИЛИ дерева планирования приведён на рисунке 2.

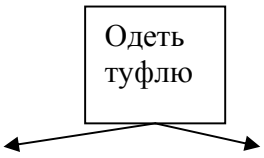
Для достижения начальной вершины «Туфли одеты» требуется достижение подцелей «Правая туфля одета» и «Левая туфля одета». Соответствующая вершина в И/ИЛИ дереве планирования имеет тип И. Для достижения цели «Правая туфля одета» требуется достижение подцелей: «Одеть туфлю», «Завязать шнурки». Соответствующая вершина в И/ИЛИ дереве планирования имеет тип И. И так далее. Штрих-пунктиром изображены терминальные вершины – вершины, которые ассоциируются с элементарными целями.



Условные обозначения:



- И вершина, соответствующая цели, с дугами



- ИЛИ вершина, соответствующая цели, с дугами

- терминальная вершина, соответствующая элементарной цели

Рисунок 2 И/ИЛИ дерево планирования для задачи с туфлями.

Описание работы любого из последующих алгоритмов достаточно громоздко, чтобы объяснить её полностью. Суть данных алгоритмов такова, что на каждом шаге итерации по критериям, заложенным в алгоритм (большей частью основаны на эвристических оценках), выбирается вершина для раскрытия, после чего она раскрывается, т.е. задача, которой соответствует выбранная вершина, разбивается на подзадачи. Это происходит до тех пор, пока не будет найдено И/ИЛИ дерево решения. И/ИЛИ дерево решения – поддерево дерева планирования, которое строится алгоритмом на протяжении работы:

- Первоначальная цель дерева планирования является корневым узлом дерева решения
- Если P — узел ИЛИ, то в дереве решения находится один и только один из его преемников (по дереву планирования) вместе с его собственным деревом решения.
- Если P — узел И, то в дереве находятся все его преемники (по дереву планирования) наряду с их деревьями решения. [3]

В данном примере дерево решения выделено жирными линиями на рисунке 2.

Далее следуют алгоритмы поиска, которые используют И/ИЛИ ориентированные графы с циклами: алгоритм CF, алгоритм REV*, интеграция алгоритмов CF и REV* - алгоритм INT, алгоритм CFCREV*[6]. Эти алгоритмы являются логическим продолжением предыдущих алгоритмов. Так же используются для обратного хода решения. В ходе работы алгоритмов строится ориентированный И/ИЛИ граф планирования редукций целей, а результатом работы уже является не И/ИЛИ дерево решения, а И/ИЛИ ориентированный граф решения. Данный подход даёт преимущество в том, что нет одинаковых вершин, что значительно сокращает их количество в графе. Однако работа с полученными графами достаточно сложна и настроена на то, что есть какие-то эвристические оценки в задачах. Данные алгоритмы также предназначены для монотонных предметных областей.

Отдельного внимания заслуживают алгоритмы, основанные на графе планирования[9]. Данный граф раздёлен на слои. Всего 2 типа слоёв – слой фактов и слой действий. При этом слои чередуются и нумеруются: слой фактов №0, слой действий №0, слой фактов №1, слой действий №1 и т.д. В каждом слое находится множество вершин, которое соответствует множеству фактов, либо множеству

действий, т.е. каждая вершина соответствует либо факту, либо действию. Дуги регламентируют связи между действиями и фактами.

Возьмём такую задачу: вездеход (робот) должен отправить данные о ресурсах – нефти, грунте, воде, которые расположены в точках А, Б, В соответственно. Для этого есть действия – движение из одной точки в другую, сбор ресурса, передача информации о ресурсе. Для того, что бы собрать ресурс, вездеход должен находиться в той же точке, где находится ресурс. Для передачи данных о ресурсе, вездеход должен собрать данный ресурс. Начальное состояние системы – вездеход находится в точке А, также в точке А находится нефть, в точке Б – грунт, в точке В – вода. Описание домена и задачи над доменом приведены в Приложении Б.

Пример графа планирования представлен на рисунке 3:

Рисунок 3 Граф планирования для задачи с роботом.

В прямоугольниках с закругленными углами изображены факты. В обыкновенных – действия. Построение графа начинается с инициализации нулевого слоя фактов начальными фактами задачи. Так, появляются вершины с надписями «Нефть в т. А», «Грунт в т. Б» и остальные данного слоя.

Затем строится слой действий: те действия, предусловия которых имеются в построенном слое фактов, добавляются в соответствующий слой. Факты эффекта добавляются в следующий слой фактов, который таким образом и строится.

Необходимо отметить так называемые «пустые» действия – они переводят каждый факт из слоя i в слой (i+1). Они изображены пунктирными стрелками.

Построенный до определённого уровня граф планирования используется различными алгоритмами для построения плана решения задачи, выборки эвристической информации.

Алгоритм GRAPHPLAN[9] строит план на основе графа планирования. В рамках данного алгоритма важно понятие взаимных исключений – это связи между действиями или фактами в слоях графа, которые говорят о том, что данные действия или факты не могут быть выполнены или получены одновременно. Данные связи появляются в слоях фактов и распространяются в слои действий согласно правилам. Разметка взаимных исключений происходит на стадии построения графа.

Для данного алгоритма граф планирования строится до тех пор, пока:

- два соседних слоя фактов не будут одинаковы (в данном случае плана решения задачи нет) или;
- в последнем слое фактов не будет находиться множество фактов из целевого состояния, которые не будут связаны взаимными исключениями (в этом случае запускается процедура поиска плана решения).

Разметка взаимных исключений – серьёзная работа с точки трудозатрат процессора. Однако возможен такой вариант, что не все исключения нужно помечать и при этом этих пометок хватит для последующей работы алгоритма построения плана[9].

Взаимные исключения появляются в слоях действий и распространяются в слои утверждений.

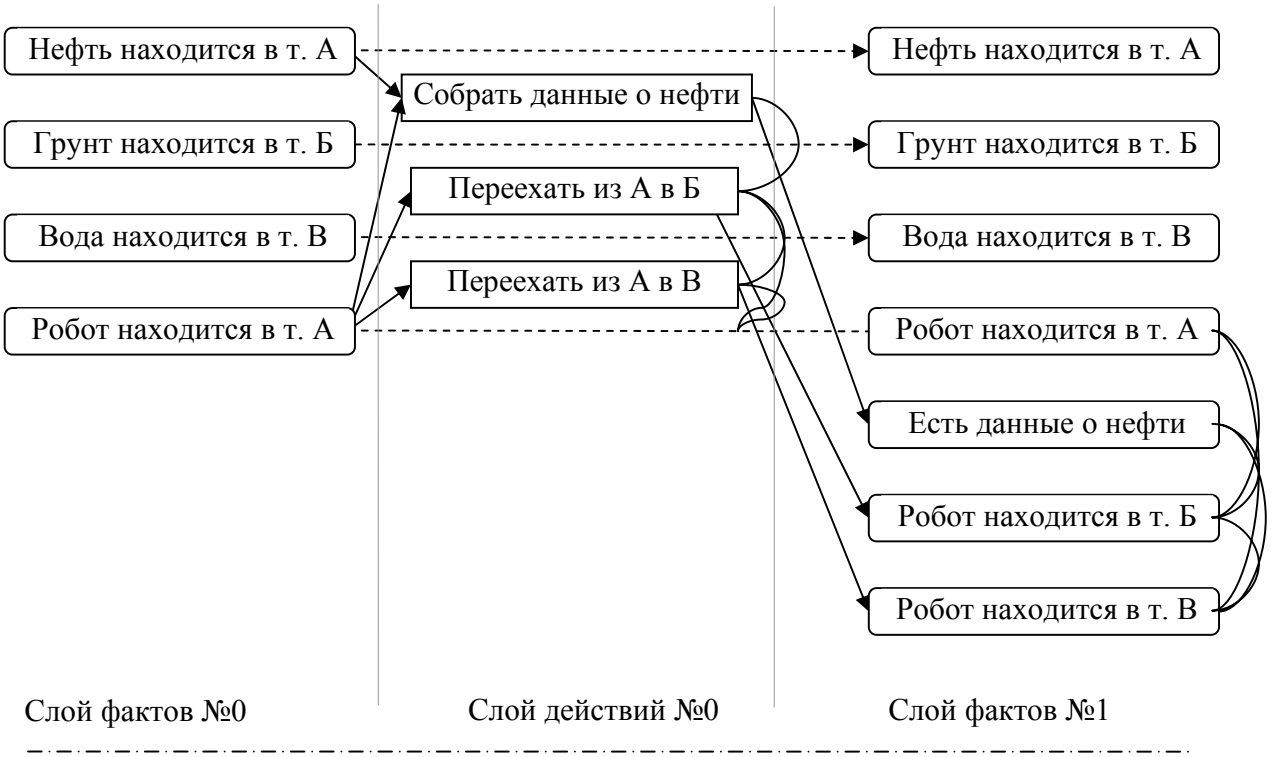
Появление в слоях действий:

1. Если какое-либо действие «удаляет» утверждение из предусловия или эффекта другого действия.

2. Если хотя бы одно утверждение из предусловия какого-либо действия помечено взаимным исключением с утверждением из предусловия другого действия.

Распространение в слой утверждений: если все действия, которые добавляют утверждение p, помечены взаимным исключением со всеми действиями, которые добавляют утверждение q, то утверждения p и q помечаются взаимным исключением.

Рассмотрим пример с роботом – рисунок 4.



Условные обозначения:

- дуга, соответствующая взаимному исключению

Рисунок 4 Граф планирования для задачи с роботом. Разметка взаимных исключений.

Взаимные исключения появляются между всеми непустыми вершинами нулевого слоя действий и в соответствии с правилами распространяются на первый слой утверждений. Так, помечаются, например, действия «Переехать из А в Б»,

«Переехать из А в Б», поскольку факт с логической связкой отрицания эффекта первого действия not «Робот в т. А» «удаляет» предусловие второго - «Робот в т. А».

План же строится обратным рекурсионным проходом. Алгоритм поиска работает со слоями - ищет действия, не помеченные метками взаимных исключений и содержащие в эффекте факты, которые находятся во множестве целей (на первом шаге в нём находятся утверждения из целевого состояния). Если такие действия находятся, то запускается следующая итерация алгоритма, с множеством целей из набора предусловий выбранных действий. Рекурсия продолжается до тех пор, пока множество целей не будет содержаться в списке начальных фактов или же не удаётся сделать выбор действий. В первом случае последовательность выбранных действий будет являться планом решения задачи. В последнем случае идёт возврат на шаг назад и идёт иной выбор действий, поддерживающих заданное множество целей. Если и это не удаётся сделать, то опять идёт возврат. И т.д. Если же возврат доходит до первого выбора действий и там также не удаётся его сделать, то план построить невозможно.

Если указанным образом не удаётся построить план решения задачи, то граф планирования достраивается на один слой и идёт попытка построения плана решения.

Данный подход использует повторяемые вершины, но они находятся на заранее известных местах. Большим минусом алгоритма является то, что достаточно быстро распространяются взаимные исключения по графу и их вычисление занимает большое количество процессорного времени. Также построение решения задачи может потребовать достроить граф планирования. Что бывает достаточно часто за счёт тех же взаимных исключений.

Спектр задач, решаемый данным алгоритмом достаточно широк.

2.3 Выводы обзора

Подводя итоги обзора, можно сделать определённые выводы.

Для описания предметных областей и задач, в принципе, подходит любой из представленных языков: STRIPS, ADL, PDDL. Однако, поскольку язык PDDL был создан с целью обобщения предыдущих языков, для того, что бы различным

планировщикам можно было подавать на вход задачи, описанные одним языком, то данный язык будет взят за основу описания входных данных для системы решения задач.

Задачи и домены следует представлять на подмножестве языка PDDL, т.к. использование полной функциональности данного языка не нужно в рамках поставленной задачи реализации доменов и задач.

В работе представляется необходимым использование эвристик из графа планирования для пространства состояний[7].

3 Средства обеспечения автоматического планирования решений задач

3.1 База знаний системы планирования

Можно выделить основные части систем решения задач. Первая часть отвечает за формальное представление задачи, домена, назовём её «База знаний». Вторая – за реализацию плана решения задачи, назовём её «Планировщик».

Для описания доменов, задач был выбран язык PDDL из-за его удобства, выразительных языковых средств и массового использования в качестве входного языка систем планирования.

Язык PDDL позволяет описывать более широкий класс доменов и задач, чем те, которые определены в главе 1. Поэтому, в разрабатываемой системе поддерживаются только необходимые для неё языковые средства PDDL:

- описание типов;
- описание предикатов;
- описание действий;
- описание множества рассматриваемых объектов;
- описание начального состояния;
- описание целевого состояния.

Рассмотрим их более подробно (РБНФ, описывающая синтаксис реализованного подмножества PDDL приведена в приложении А).

Все примеры приведены из домена «Вездеход». Полное описание домена и задач дано в приложении Б.

Домен во входном языке системы будет представлять собой набор типов, предикатов над переменными этих типов, действий. Любой тип домена описывается конструкцией:

<types-def>::= (:types **<type>**$^{+}$)

<type>::= <name>

Например: (:types location data). Описаны 2 типа домена: location и data.

Каждый предикат домена описывается конструкцией:

<predicates-def>::= (:predicates **<atomic-formula-with-or-without-types>**$^{+}$)

<atomic-formula-with-or-without-types>::= (**<predicate> <list-varibles-with-or-without -type>***)

<predicate>::=<name>

<list-varibles-with-or-without -type>::= <variable>+ - <type> | <variable>+

<type>::=<name>

<variable>::= ?**<name>**

Например:

(:predicates

(at ?x - location)

(avail ?d - data ?x - location))

Описаны предикаты at, avail. При этом, в определении данных предикатов фигурируют параметры определённых ранее типов location и data. В данной задаче предикат (at ?x) предполагает нахождение робота в точке ?x. (avail ?d ?x) – ресурс ?d находится в точке ?x. При определении предикатов используются параметры. Если их инициализировать объектами из задачи, то получаем факты (утверждения). Возможен вариант объявления предиката, у которого будут не обозначены типы переменных. В этом случае, подразумевается то, что переменная принадлежит любому типу.

Для действий определяются набор переменных, предусловие и эффект. Последние два состоят из формул над определёнными ранее предикатами над переменными данного действия. Причём, в формуле эффекта логическими связками могут быть только конъюнкции и отрицания, а в предусловии – только конъюнкции.

Описание действий:

<actions-def>::= <action>+

<action>::= (:action **<name>** :parameters (**<list-varibles-with-type>**) :precondition (and **<atomic-formula>+**) :effect (and **<atomic-formula>+**))

<atomic-formula>::= (not (**<predicate> <variable>***)) | (**<predicate> <variable>***)

Например:

(:action drive

:parameters (?x ?y - location)

:precondition (and(at ?x))

:effect (and (at ?y) (not(at ?x))))

Определена схема действия drive. После идентификатора ‘:parameters’ идёт объявление параметров, нужных для определения действия. Далее идут предусловие и эффект. Предусловие – конъюнкция предикатов. Эффект – конъюнкция предикатов или их отрицаний. Таким образом, действие drive применимо тогда когда в текущем состоянии есть факт (at ?x). Данное состояние действие переводит в состояние с тем же набором фактов, за исключением того, что факт (at ?x) меняется на факт (at ?y).

Условие применения действия к состоянию определяется так: если формула предусловия выводима из формулы состояния, то действие применимо. Действие переводит данное состояние в состояние, определяемое формулой эффекта: все те предикаты, которые связаны в формуле с остальными логической связкой отрицания, удаляются из состояния, другие же, при отсутствии в формуле состояния, - добавляются. Так строится новое состояние из предыдущего.

Задачи представляют собой набор объектов определённых в домене типов, начальное состояние и конечное. Состояние – формула над определёнными в домене предикатами над множеством заданных объектов; при этом, логической связкой в данной формуле может быть лишь конъюнкция. Ниже представлена часть РБНФ для описания задач. Объявление объектов:

<list-objects>::= (:objects **<objects-types>**$^{+}$)

<objects-types>::=<object>$^{+}$ - **<type>|** (either **<type>**$^{+}$)

<object>::=<name>

Например:

(:objects
soil water rock - data
alpha beta gamma - location)

Определены объекты soil, water, rock типа data и alpha, beta, gamma типа location. Использование идентификатора ‘either’ нужно для определения объектов, которые относятся к разным типам.

Начальное состояние и целевое:

<list-init>::=’(:init’ <list-predicates-initialization >$^{+}$ ‘)’

<list-predicates-initialization>::=’(‘ <predicate> <object>* ‘)’

<list-goal>::=’(:goal (and’ <list-predicates-initialization >$^{+}$ ‘))’

Например:

(:init (at alpha)
(avail soil alpha)
(avail rock beta)
(avail water gamma))
(:goal (and
(comm soil)
(comm water)
(comm rock)
(at alpha)))

Здесь задаётся начальное состояние, состоящее из фактов (at alpha), (avail soil alpha), (avail rock beta), (avail water gamma). Соответственно задаётся целевое состояние.

3.2 Использование графа планирования в системе решения задач

Граф планирования даёт информацию о задаче. На основе графа планирование поиск плана решения задачи можно свести в сторону метода поиска в пространстве состояний. По сути, граф планирования даёт возможность получить эвристическую информацию о состояниях системы весьма дешёвыми способами. Так же на основе анализа некоторых взаимных исключений можно получить информацию о фактах, которые не могут находиться одновременно в одном состоянии. Данная информация важна для обратного хода поиска в пространстве состояний.

Далее приведёно формальное описание алгоритма построения графа планирования для системы решения задач.

1. Инициализировать граф планирования начальными фактами. Каждой вершине присвоить уровень i = 0.
2. Добавить все возможные действия, которых нет в слое действий, и у которых все предусловия имеются в слое утверждений. Добавить соответствующие дуги от фактов к действиям. Назначить данным действиям уровень i.

3. Добавить факты эффекта от вновь полученных действий в слой фактов, если таких фактов ещё нет, и назначить им уровень (i + 1). Добавить соответствующие дуги от действий к фактам.
4. Пометить дугой взаимного исключения те действия, у которых: одно из двух действий удаляет факт предусловия или факт эффекта второго действия (этот шаг нужен только для стратегии, использующей обратный ход).
5. Проверить, все ли факты из цели задачи есть в слое фактов. Если все – то прекращаем работу построения графа планирования.
6. Проверить, добавились ли какие-нибудь вершины в слой фактов за последнюю итерацию. Если нет – выходим – решения нет. Иначе переходим на шаг 2.

В данном графе планирования нет структур, соответствующих пустым действиям, повторяющихся вершин, практически нет взаимных исключений. При этом он даёт туже эвристическую информацию и подбор нужных действий с инициализированными переменными объектами задачи. Данный подход позволяет говорить о полученном графе планирования как о хранилище информации о задаче, которая потребуется для поиска решения.

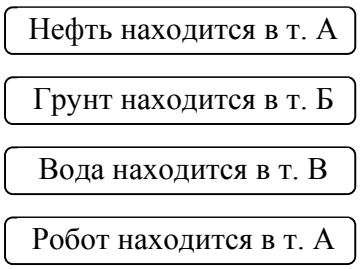


Рисунок 5 Граф планирования, инициализированный фактами из начальной вершины задачи с роботом.

Пример построения графа планирования для задачи с роботом при инициализации утверждениями начального состояния приведён ниже. Поскольку, модификация графа планирования достаточно запутанно выглядит, то на рисунках будет изображён «стандартный» граф планирования.

Как и раньше факты представлены вершинами графа (прямоугольники с округленными краями и фактами внутри). На первом шаге инициализируется слой

фактов фактами начального состояния. Им присуждается нулевой уровень. Соответственно, на рисунке это слой фактов №0.

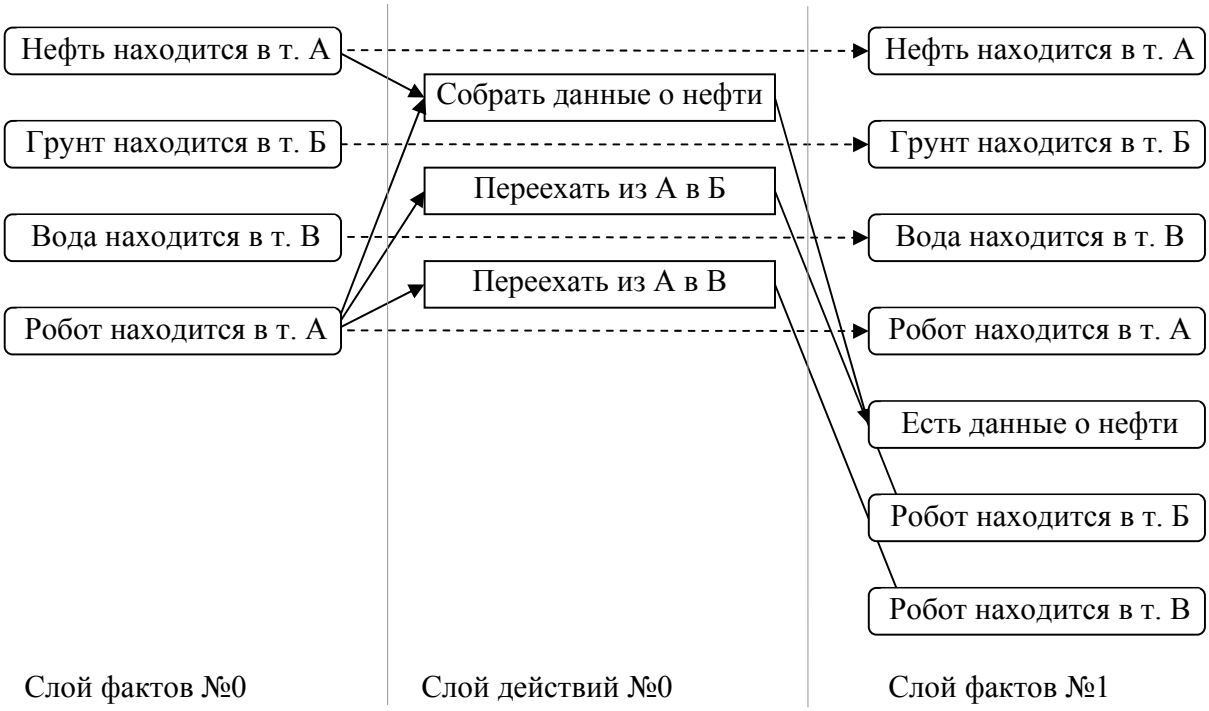


Рисунок 6 Граф планирования после первой итерации работы алгоритма построения.

После первой итерации работы алгоритма, получаем такой граф планирования: в слое действий (он находится в правой части) находятся действия «Собрать данные о нефть», «Переехать из А в Б», «Переехать из А в В»; им присваивается 0 уровень (на рисунке - слой действий №0). Эффекты данных действий так же добавляются в граф в слой фактов – это «Есть данные о нефти», «Робот находится в точке Б», «Робот находится в точке В» соответственно. Данным фактам присуждается уровень равный 1 в соответствии с алгоритмом построения графа.

Полностью граф планирования изобразить нельзя в связи с его размерами.

Как для прямого хода решения задачи, так и для обратного, из данного графа мы можем получить эвристическую оценку вершины, а также набор возможных действий для последующего раскрытия вершины.

Многие методы получения эвристических оценок, описанные в [7], можно применить к построенному графу планирования. Среди них: Set-level, Sum, Relaxed-plan.

Первый и третий методы дают эвристические оценки количества действий до достижения цели меньшие реальных. Для алгоритмов, подобных A*, - это хорошая информация, т.к. на основе данной информации поиск плана решения задачи происходит достаточно быстро.

Далее приведён алгоритм relaxed-plan получения эвристической оценки для графа планирования из системы решения задач для прямой стратегии поиска плана решения задачи в пространстве состояний:

1. Произвести пересчёт уровней в графе планирования с учётом того, что нулевой уровень наполняют факты состояния, для которого ищется эвристическая оценка. Если пересчёт уровней невозможен, то достроить граф планирования.
2. Внести в пустое множество фактов Ф все факты из целевого состояния. Пусть максимальный уровень среди уровней данных фактов равен N. i=N -1. Множество действий Д пусто.
3. Если i<0, то вернуть количество элементов в множестве действий Д.
4. Добавить действия i уровня в множество действий Д, которые имеют в эффекте какой-нибудь факт из множества Ф. Факты предусловий добавленных действий добавить в множество Ф.
5. i=i-1. Перейти на шаг 3.

Так для полученного ранее графа планирования данный алгоритм даст эвристическую оценку 8.

Для получения эвристической информации о каждом состоянии нужно пересчитывать уровни графа планирования с инициализацией нулевого уровня фактами состояния, если это выходит, или достраивать граф планирования.

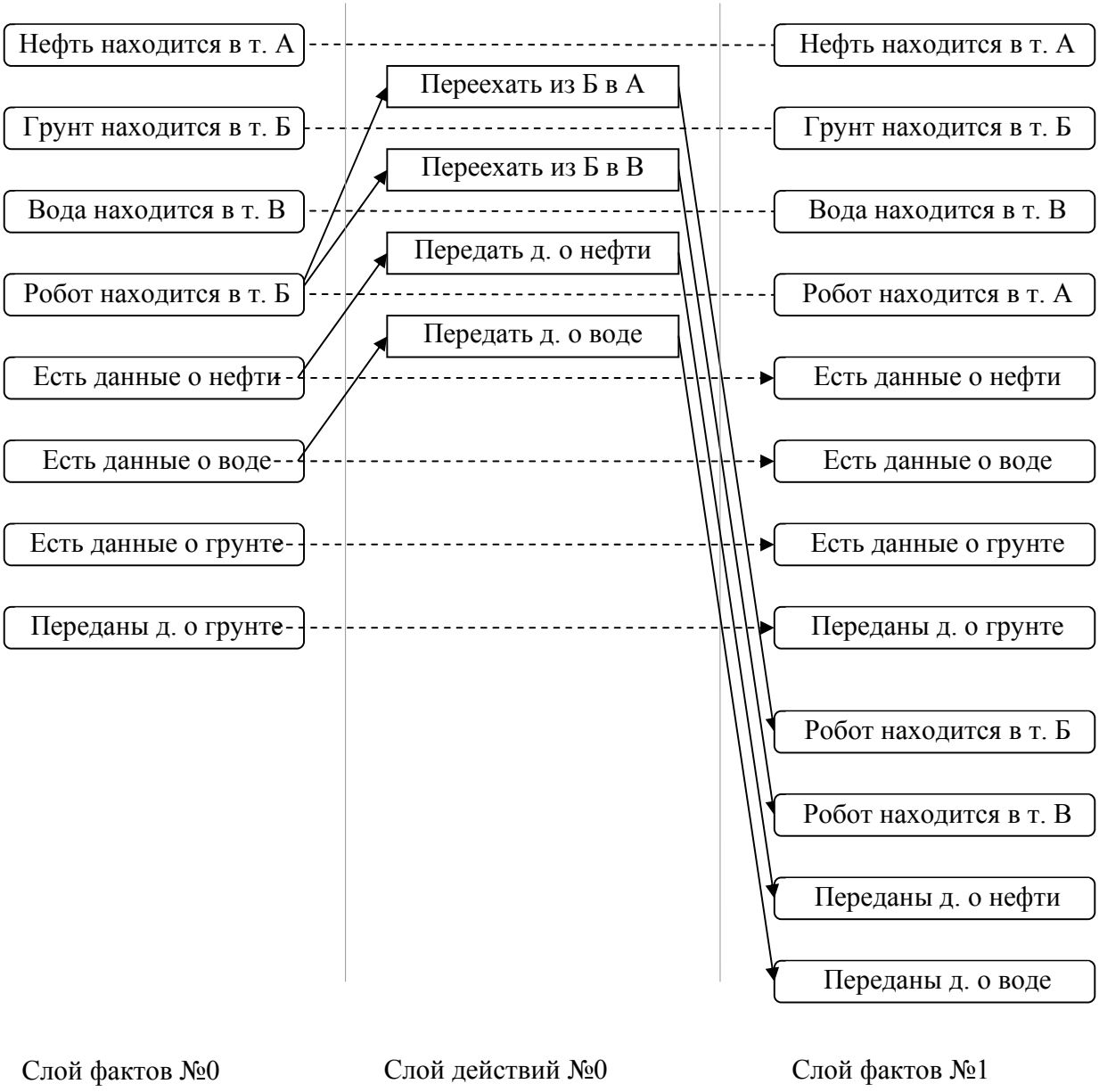


Рисунок 7 Граф планирования после пересчёта уровней.

Рассмотрим пример пересчёта уровней для состояния («Робот находится в точке Б», «Нефть находится в точке А», «Грунт находится в точке Б», «Вода находится в точке В», «Переданы дынные о нефти», «Есть данные о воде», «Есть данные о грунте», «Есть данные о нефти») для планирования. На рисунке 7 изображена его часть, как и раньше в стандартной интерпретации.

Всем фактам из данного состояния в графе проставлен уровень 0. Всем остальным вершинам присуждается уровень, равный бесконечности. Далее, проходя

по связям к дочерним вершинам-действиям, алгоритм проставляет максимальный уровень из уровней отцовских вершин для данных вершин, если начальный уровень равен бесконечности. И т.д. Алгоритм заканчивает работу, когда все утверждения из целевого состояния имеют конечный уровень, либо когда на последнем шаге цикла не произошло никаких изменений. В данном случае алгоритм дошёл до той точки, когда у всех фактов из целевого состояния все уровни конечны. Эвристическая оценка состояния равна 2 действиям.

Для получения эвристической оценки для обратного хода в алгоритм вводятся поправки:

- нужно пересчитывать уровни графа планирования с учётом того, что нулевой уровень наполняют факты начального состояния (1 шаг соответствующего алгоритма);
- Внести в пустое множество фактов Ф все факты из состояния, эвристическую оценку которого надо получить (2 шаг соответствующего алгоритма).

Эвристика Set-level. Для прямого хода: если в нулевом уровне находится множество фактов, которые задают данное состояние, то эвристическая оценка данной вершины равна номеру последнего уровня или максимальному номеру уровня из уровней фактов целевой вершины. Для обратного хода: максимальный уровень среди утверждений из данного состояния для графа построенного с инициализацией в начальном состоянии. Данная эвристическая оценка <= реальной цены. Так для графа, полученного в предыдущем примере, эвристическая стоимость достижения из начальной вершины конечной будет равна 3 действиям.

Эвристика Sum. Для прямого хода: если в нулевом уровне находится множество утверждений, которые задают данное состояние, то эвристическая оценка данной вершины равна сумме уровней утверждений из целевого состояния задачи. Для обратного хода: сумма уровней среди утверждений из данного состояния для графа построенного с инициализацией в начальном состоянии. Данная эвристическая оценка >= реальной цены. Для данного метода эвристическая стоимость достижения из начальной вершины конечной будет равна 8.

3.3 Алгоритм работы планировщика

Граф планирования нужен для получения вспомогательной информации. Эта информация далее используется для шагов прямого и обратного хода в методе поиска плана решения задачи в пространстве состояний с использованием эвристических оценок для отсечения ненужных ветвей перебора. Таким образом, основой для построения пространства состояния или графа, соответствующего ему, основано на графе планирования. Данный граф пространства состояний состоит из вершин, которые ассоциированы с состояниями задачи. Направленные дуги ассоциируются с применением действия к вершине-состоянию, из которой дуга выходит, и получившейся вершине-состоянию, в которую дуга входит. Поскольку используется алгоритм, похожий на A*, для прямого/обратного хода, эвристические оценки, то каждой вершине приписывается цена.

Далее приведена общая схема работы получившегося алгоритма поиска решения, использующая прямой ход, для решателя. Граф, вершины которого будут ассоциироваться с состояниями из пространства состояний, будем называть граф пространства состояний.

1. Построить граф планирования, который инициализируется фактами из начального состояния задачи.
2. Инициализировать граф пространства состояний начальной вершиной, соответствующей начальному состоянию с ценой равной эвристической оценке, полученной из графа планирования.
3. Занести начальную вершину в список открытых вершин O с соответствующей ценой.
4. Выбрать вершину для раскрытия из списка O с минимальной ценой и минимальной эвристической оценкой.
5. Раскрыть выбранную вершину. Для каждой вновь полученной вершины:

5.A. Вычислить эвристическую стоимость состояния, соответствующего данной вершине, с помощью графа планирования. Присвоить цену вершине равную сумме 1, эвристической оценки и количества дуг, принадлежащих минимальному пути от раскрываемой вершины до начальной вершины.

5.B. Если такой вершины ещё нет в графе пространства состояний, то добавить ее туда с соответствующими дугами. Если есть, то заменить вершину вновь

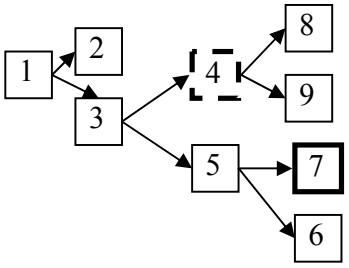
построенной в том случае, когда цена получившейся вершины меньше цены той, которая находится в графе; добавить и удалить соответствующие дуги.

5.C. Если эвристическая оценка вершины равна 0, значит решение найдено (достаточно подняться из данной вершины до начальной вершины, записывая действия, ассоциированные с дугами; полученный список действий будет представлять собой решение). Перейти на 7 шаг.

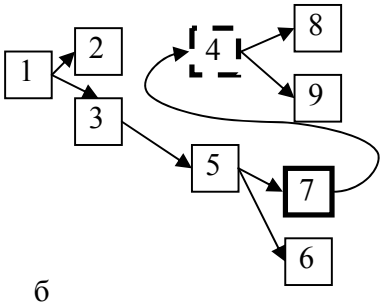
5.D. Добавить вершину в список открытых вершин O с соответствующей ценой.

6. Перейти на шаг 4.
7. Завершить работу алгоритма.

В сущности, граф пространства состояний, получаемый при работе данного алгоритма – это дерево. Однако построенный алгоритм не создаёт повторяющихся вершин, что является несомненным плюсом: если на очередной итерации работы алгоритма появляется вершина, которая уже была создана, то, при условии меньшей цены, она заменяется. Таким образом, данный граф остаётся всегда деревом, но без повторяющихся вершин. Рассмотрим пример замены вершины:



а
Граф перед добавлением вершины. Раскрывается вершина №7.



б
Граф после добавления вершины эквивалентной вершине № 4.

Условные обозначения:

- вершина графа пространства состояния

- дуга графа пространства состояния

- раскрываемая вершина

- вершина эквивалентная той, которую получаем после раскрытия

Рисунок 8 Замена вершины в графе пространства состояний.

Пусть раскрывается вершина №7 (а), и одна из вновь добавляемых вершин ассоциируется с состоянием, эквивалентным состоянию вершины №4. При этом цена вершины №4 больше цены вершины, которую получили. Поэтому идёт замена дуг и цены вершины №4.

Поскольку алгоритм построения графа пространства состояний аналогичен алгоритму A* то, при том что эвристическая оценка вершины <= реальной оценки, мы на выходе имеем минимальный план решения задачи.

Выбор вершин для раскрытия из списка открытых вершин О происходит в таком ключе: выбираются вершины с минимальной стоимостью и минимальной эвристической оценкой. Такой подход даст определённые плюсы: развитие графа пространства состояний будет происходить в заведомо приоритетную сторону. А это означает и создание меньшего количества объектов, и наиболее быструю работу алгоритма по сравнению с иными подходами. Конечно, данные плюсы даются лишь с использованием графа планирования, его вспомогательной информации.

Примеры работы алгоритма для прямого хода решения с применением эвристики relaxed-plan представлены ниже.

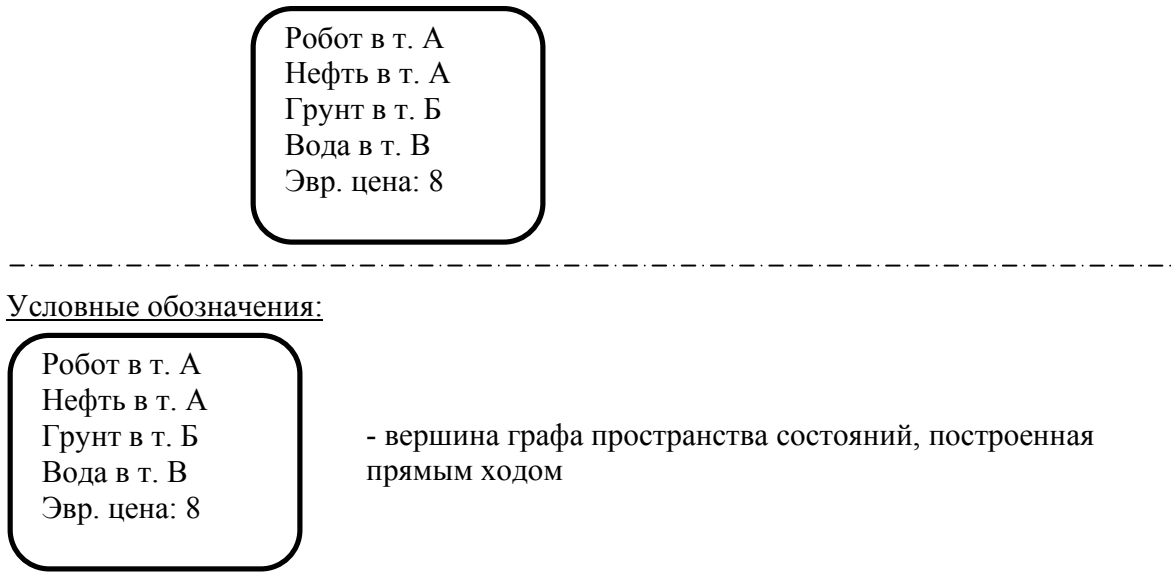


Рисунок 9 Граф пространства состояний. Создание начальной вершины.

На первом шаге создаётся граф планирования и начальная вершина, которая соответствует начальному состоянию, заносится в граф пространства состояний. Она заносится в список открытых вершин О с ценой 8. Так как работа с графом планирования разобрана достаточно ясно, на рисунках будет отображен только граф пространства состояний. Вершина изображена в виде прямоугольника с округленными краями. Последние две надписи – говорят о эвристической цене и цене дуг. Остальные записи – факты состояния, ассоциированного с данной вершиной.

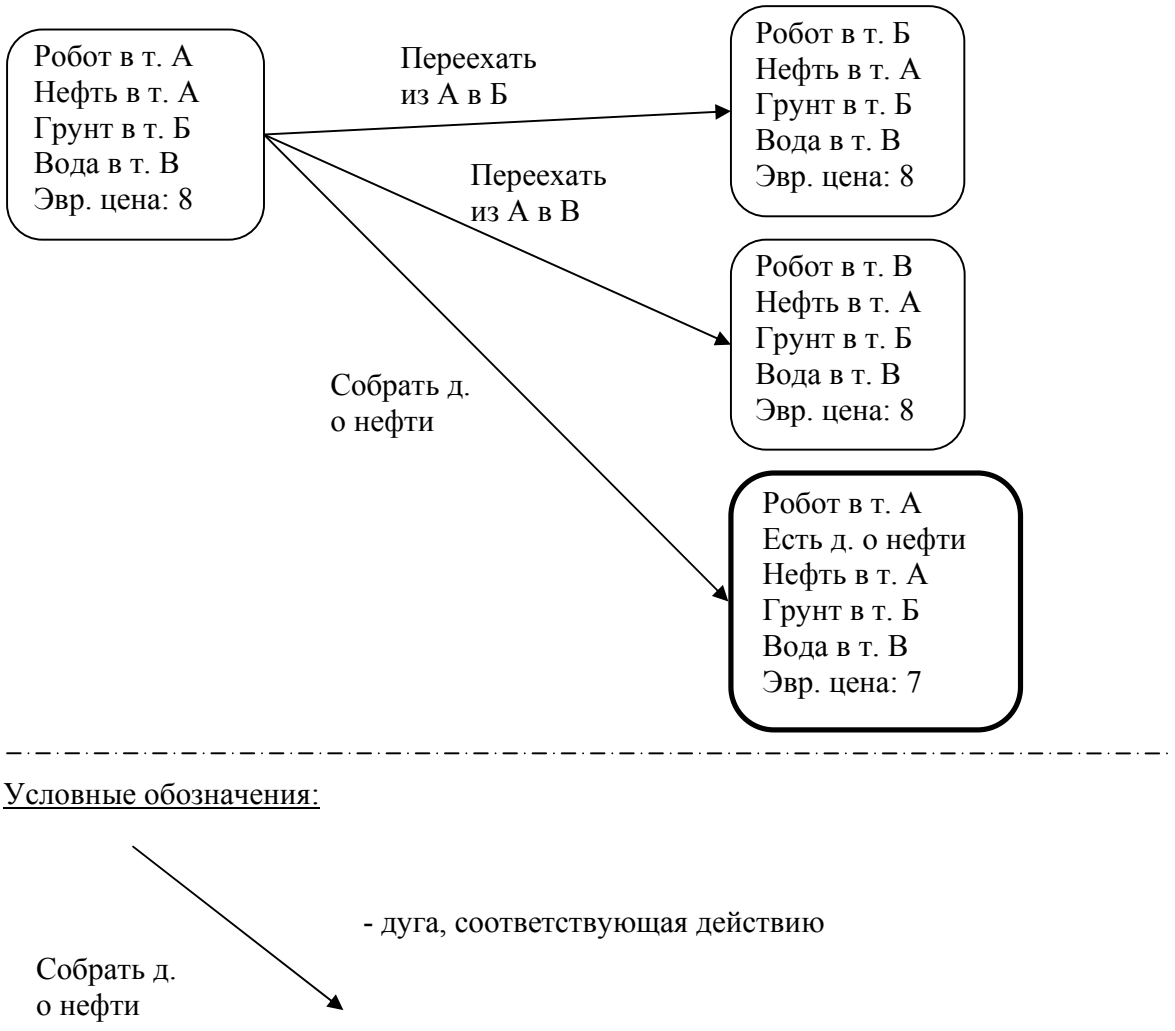


Рисунок 10 Раскрытие начальной вершины.

Далее идёт раскрытие начальной вершины, так как это вершина с минимальной ценой. Строятся дочерние вершины. Для каждой из них считается

эвристика relaxed-plan, для чего идёт пересчёт уровней в графе планирования. После этого вершины добавляются в список открытых вершин с соответствующими ценами.

Затем идёт поиск вершины с минимальной ценой среди вершин из списка открытых вершин. Выбор падает на вершину, выделенную жирной линией. Цена равна 8 (1 – цена дуг, 7 – эвристическая цена достижения конечной цели задачи). После чего данная вершина раскрывается, у появившихся вершин считаются эвристические оценки на основе графа планирования. И т.д.

На заключительной стадии работы алгоритма появляется вершина, множество фактов которой включает в себя множество фактов конечной цели. К сожалению, из-за величины графа нет возможности показать его на рисунке.

После этого, от данной вершины идёт подъём от полученного листа до начальной вершины графа с записью действий в список, который будет составлять план решение задачи.

Аналогично строится алгоритм с использованием попеременно прямого/обратного хода, за исключением данных поправок:

- Добавления вершинам метки прямого/обратного хода.
- Работа алгоритма завершается, не когда эвристическая оценка равна 0, а когда множество фактов у какой-либо вершины, построенной прямым ходом, включает в себя множество фактов какой-либо вершины, построенной обратным ходом.
- Решение строится из двух половинок: одна по вершинам, построенным прямым ходом, другая – обратным.
- В шаге 2 нужно также добавить конечную вершину с меткой обратного хода.
- Кроме прямого хода (шаги 4-6) также есть обратный ход. Шаги аналогичны.
- Для определения утверждений, которые не могут находиться одновременно в одном состоянии, строятся «урезанные» взаимные исключения, которые не распространяются по графу планирования. При построении новых вершин в графе пространства состояний нужно учитывать этот факт при обратном ходе решения. Для прямого хода это не важно, так как сами действия уже определены на языке PDDL так, что таковые случаи невозможны.

Говорить о правильности построенного плана в данном случае нам позволяет тот факт, что с одной стороны вершина прямого хода А, в которой нашлись все утверждения из вершины Б, достижима из начальной вершины, с другой стороны из вершины Б достижима целевая вершина. Поскольку имеет место вложение множеств, то следует факт правильности построенного плана.

Если отходить от формальностей, то в данном алгоритме граф пространства состояний можно представить как два дерева, которые растут друг к другу до тех пор, пока не будет достаточно «одинаковых» вершин по одной у каждой половинки графа.

Пример работы алгоритма с графом для попеременно прямого/обратного хода решения с применением эвристики relaxed-plan приведён ниже.

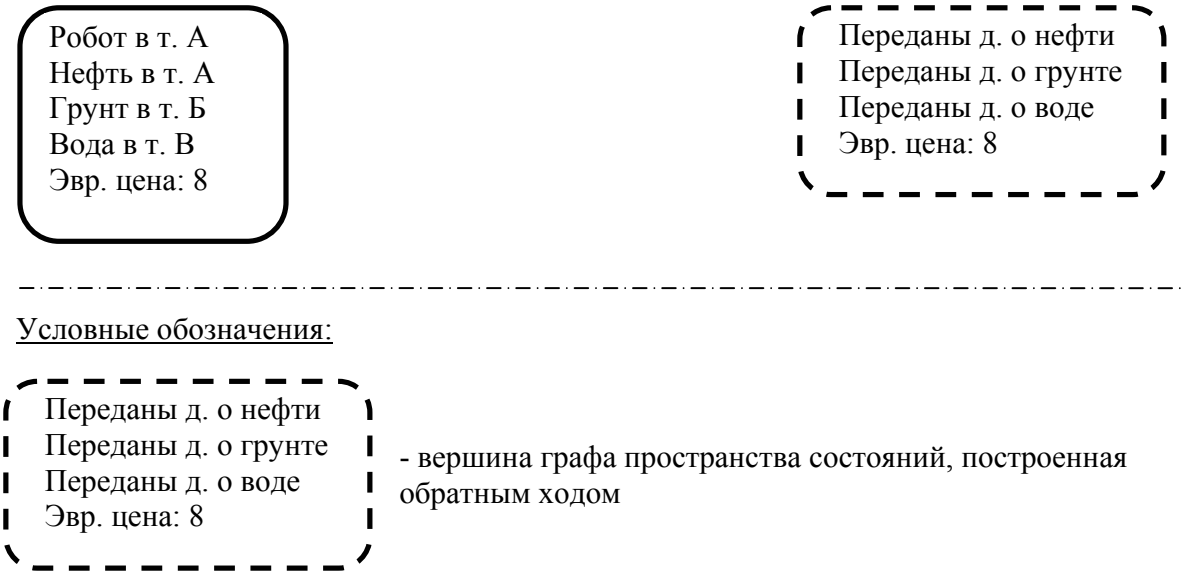


Рисунок 11 Граф пространства состояний. Создание начальных вершин.

Сначала идёт построение вершин в графе для начального состояния и целевого. Вершины обратного хода помечены пунктирной линией.

Затем идёт шаг прямого хода – рисунок 12. Раскрывается начальная вершина, считаются эвристические оценки, они заносятся в список открытых вершин. Вершины прямого хода показаны слева вверху. То же самое происходит и на шаге обратного хода с конечной вершиной. Вершины обратного хода показаны справа внизу.

Далее, на шаге прямого хода выбирается вершина, построенная прямым ходом, с минимальной ценой и раскрывается. Аналогично идёт шаг обратного хода. И т.д.

Алгоритм завершает работу, когда множество утверждений какой-либо вершины, построенной прямым ходом, включает в себя множество утверждений какой-либо вершины, построенной обратным ходом.

В данном алгоритме построение идёт в двух частях: первая из дерева обратного хода, вторая – из дерева прямого хода.

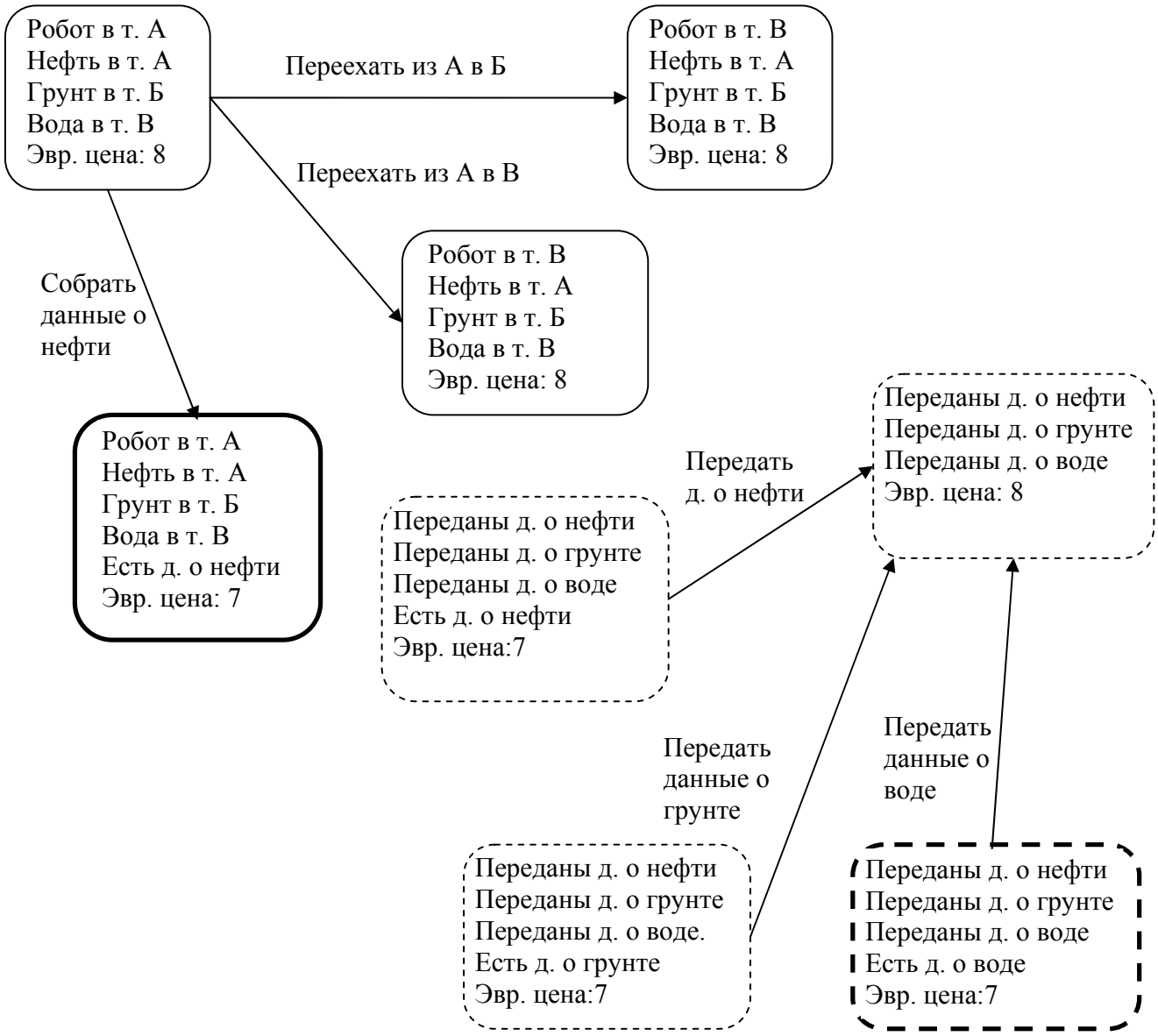


Рисунок 12 Итерация работы алгоритма – раскрытие вершин при прямом и обратном ходах.

4 Реализация системы решения задач

Архитектура разработанной системы решения задач приведена на рисунке 13:

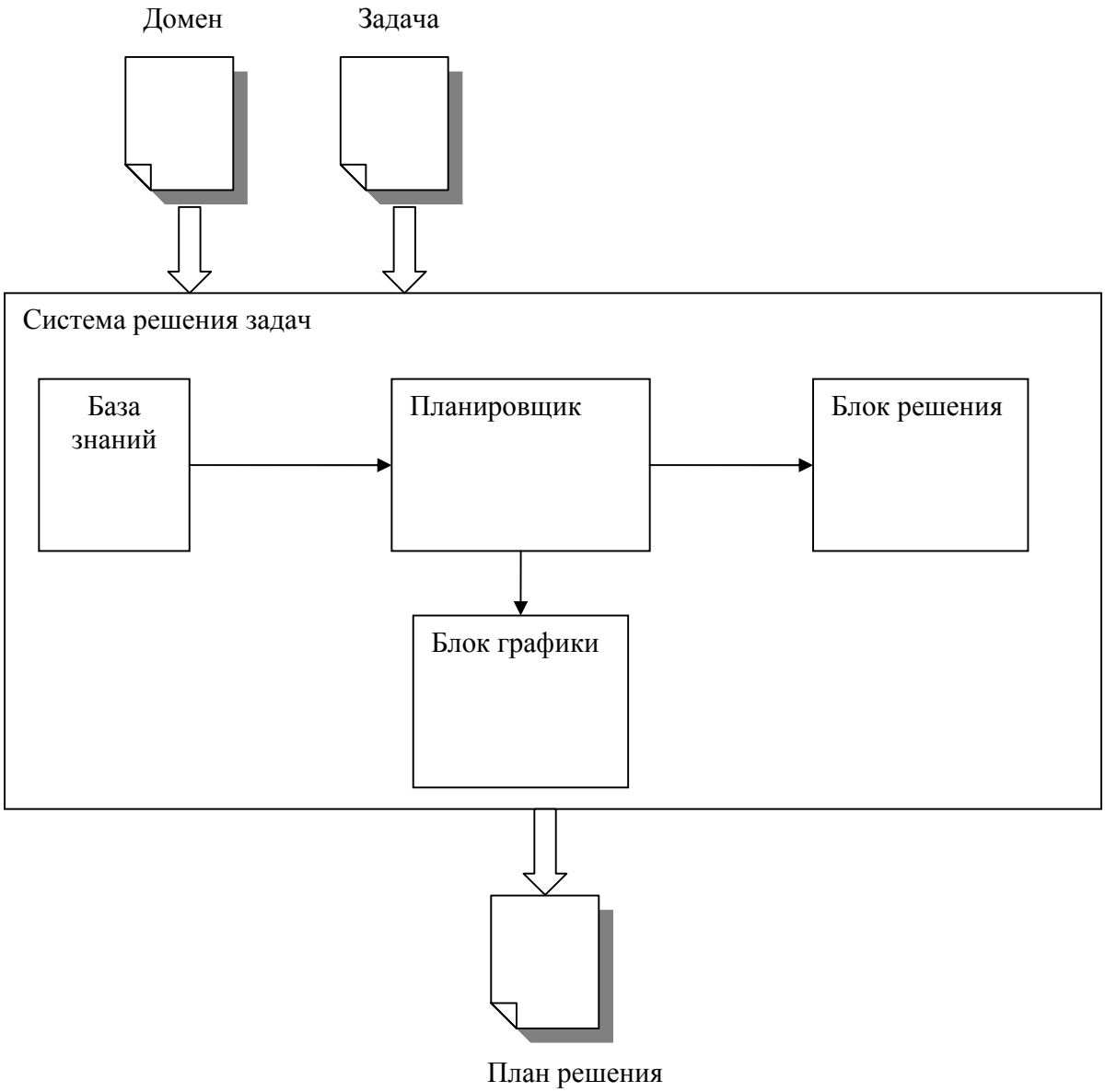


Рисунок 13 Архитектура системы решения задач.

Система была реализована на языке C++.

База знаний реализует парсер подмножества языка PDLL в соответствии с предложенной РБНФ. Проводятся лексический, синтаксический и семантический анализы.

В парсере использован стандартный метод рекурсивного спуска, поскольку использование более серьёзных инструментов для создания данного анализатора было посчитано не нужным.

Парсер был выделен в отдельный модуль «pddl.cpp».

Поскольку на вход парсеру поступает информация о задаче, то здесь же инициализируются начальные данные о задаче: начальное, конечное состояния, типы, объекты, предикаты – всё это заносится экземплярами соответствующих созданных классов в определённые списки, множества, основанные на шаблоне list и map стандартной библиотеки STL[5]. Так, типы записываются в список, определяемый такой конструкцией:

```
typedef  std::list <_bstr_t> list_types;
```

Основой для его создания является класс _bstr_t, который является классом-обёрткой для типа BSTR. Данный класс содержит член - строку, методы, операторы, позволяющие обращаться к члену класса.

Объекты – экземпляры класса parameters:

```
class parametrs
{
public:
	_bstr_t name_v;
	std::list<_bstr_t> name_t;

	void clear();

	parametrs();
	~parametrs();
};
```

Данный класс содержит члены: имя объекта и список типов, к которым принадлежит данный объект. Функция-член производит очистку памяти, выделяемую для объекта данного класса. Экземпляры хранятся в структуре list:

```
typedef  std::list <parametrs> list_parametrs;//лист параметров
```

У базы знаний есть связь по данным с планировщиком. Ему передаются данные, полученные ранее, в виде определённых структур (списки, представления). Данный блок реализует предложенные ранее алгоритмы планирования. В

зависимости от входных данных он реализует план решения задачи прямым ходом или попеременно прямым/обратным ходом решения в пространстве состояний на основе эвристической информации, полученной из графа планирования.

Граф планирования хранится в структурах map. Так как он разбит на две части идейно, то и хранятся эти части отдельно: в одном представлении вершины-факты, в другом – вершины-действия. При этом ключом является имя вершины. Поскольку в графах нет одинаковых вершин, то следует единственность ключа для вершины. Представление выбрано для хранения вершин графа из-за того, что часто требуется производить поиск вершин по имени. Представление же производит данный поиск с логарифмической сложностью. Класс, объекты которого являются вершинами графа планирования:

```
class node
{
public:
	_bstr_t name;

	int level;

	std::list<parametrs> param;

	std::list<std::map<_bstr_t,node> ::iterator>fathers;
	std::list<std::map<_bstr_t,node> ::iterator>daughter;

	std::map<_bstr_t,node_mutex>mutex;

	void clear();

	node();
	~node();
};
```

Если смотреть члены сверху вниз, то здесь описано: имя вершины, уровень, 2 множества ссылок на родительские и дочерни вершины, множество ссылок на вершины, с которыми есть связь взаимного исключения. Далее идёт функция-член очистки памяти, конструктор, деструктор.

Граф пространства состояний также хранится в структуре map. При этом все факты берутся из графа планирования, а вершины графа пространства состояний содержат ссылки на них.

```
class node_main
{
```

```
public:
	int coast;
	int coast_arc;

	int type;

	_bstr_t name;

	list_iterators state;
	std::list<std::map<_bstr_t,node_main> ::iterator>daughter;
	std::list<std::map<_bstr_t,node_main> ::iterator>father;


	void clear();
	node_main();
	~node_main();
};
```

Сначала идут: эвристическая цена, цена пути до начальной вершины, тип вершины (для определения того на прямом ходе поиска была построена вершина или на обратном), имя вершины. Далее идут списки ссылок на факты из графа планирования, которые составляют состояние, ссылки на отцовские и дочерние вершины графа пространства состояний. В конце есть стандартные функции-члены.

Данные классы, функции, реализующие алгоритмы планирования, выделены в модуль «graphplan.cpp».

Блоку решения по связям по данным предоставляется список действий по достижению цели поставленной задачи (в случае, если задача имеет решение) или пустой список (в случае, если решения нет). Блок записывает данные действия на языке PDDL в файл, а также выводит на экран. Также выводятся дополнительные сведения о графах, построенных системой – количества вершин в графе планирования, в графе пространства состояний.

Возможна работа блока графики. У данного блока есть связи по данным и связи по управлению с планировщиком: работа его начинается по инициативе планировщика. Даная часть реализует отображение на экране графа планирования и графа пространства состояний. Данный блок выделен в модуль «graphics.cpp».

При реализации алгоритма построения графа планирования была реализована система бэктрекинга, подобная существующей в языке Prolog[3]. Дело в том, что при выборе действия, добавляемого в текущий момент, так или иначе, нужна определённая система инициализации переменных для предикатов из предусловия и эффекта. Такие системы зачастую реализованы в самих языках

(например Prolog), но в C++ такого нет, поэтому она была реализована, как простой вариант перебора. В данном примере просматривается плюс графа планирования для системы решения задач: он является хранилищем действий с инициализированными параметрами, эвристической информации. При чём для решения задачи данных действий хватает и в других местах переборный механизм, подобный бэктрекингу, использовать уже не нужно.

Для реализации бэктрекинга использовалась структура stack, для запоминания точек возврата.

Модульная структура представлена на рисунке 14:

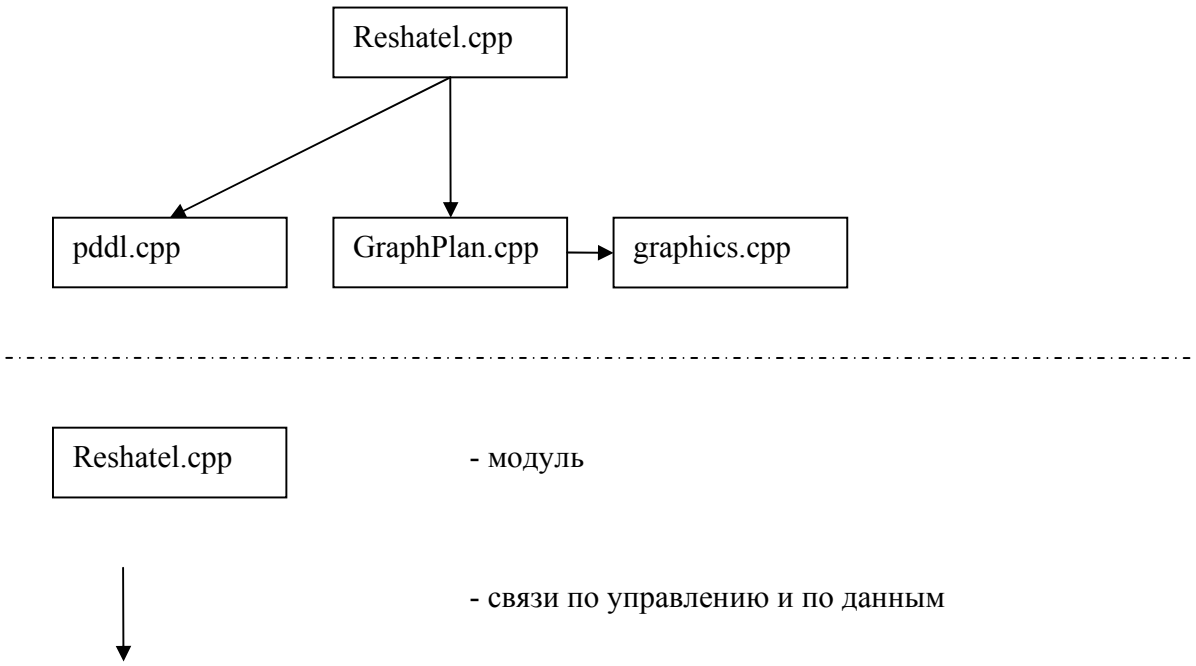


Рисунок 14 Модульная структура системы решения задач.

В модуле «reshatel.cpp» расположена функция main, которая вызывает функции других модулей, передавая им в параметрах ссылки на различные структуры, которые предназначены для хранения информации о графах, типах, объектах. Соответственно, данные объекты также определяются в этом модуле. В этом же модуле расположен блок решения.

5 Описание примеров работы системы решения задач

Для проверки работоспособности, сравнительного анализа были выбраны три различных проблемных области: домен «Ханойские башни», домен «Вездеход», домен «Планирование перевозки грузов».

Данные домены различаются по количеству типов, объектов, действий, их структуре.

Далее приведены словесные описания доменов.

Опишем домен «Ханойские башни», описывающий хорошо известную логическую игру с таким же названием. Условие задачи в домене «Ханойские башни» содержит n – количество дисков различного диаметра, m – количество мест для дисков, где n и m – натуральные числа, меньше 13, начальное состояние игры, описывающее расположение дисков на местах, и целевое состояние, которое необходимо достигнуть, перекладывая диски. При переносе любого диска следует соблюдать правило: на диске, который переносим, не должно находиться иных дисков, а место, куда переносим диск, должно быть пусто или диаметр верхнего диска в этом месте должен быть больше переносимого.

Описание данного домена на PDDL содержится в приложение Б. Описывается тип DISCC, значение которого играют роль дисков и мест для дисков. Предикат clear(x) истинен тогда, когда на месте x нет дисков или на диске x нет других дисков. Предикат on(x,y) истинен, когда диск x находится на диске y или диск x находится на месте y. Предикат smaller(x,y) истинен тогда, когда диаметр диска y меньше диаметра диска x. Перенос диска x с места (или диска) y на место (или диск) z описывается действием move(x, y, z). Предусловие действия move(x, y, z): диск x свободен, место (или диск) z свободно, диск x находится на месте (или диске) y, размеры x и z таковы, что x можно разместить на z. Эффект действия move(x,y,z): диск (или место) y свободен, диск x свободен, диск (или место) z занят, диск x находится на диске (или месте) z.

Домен «Вездеход» мы рассматривали выше в параграфе 2.2, обсуждая алгоритмы планирования. Описание домена на PDDL содержится в приложении Б. В нём описаны 2 типа – location (точка) data (данные и ресурсы). Определены 4 предиката:

1) at(x) – говорит о том, что вездеход находится в точке x;
2) avail(x,y) - говорит о том, ресурс x находится в точке y;
3) comm(x) – описывает, что данные о ресурсе x переданы;
4) have(x) – описывает, что собраны данные о ресурсе x, и они готовы к передаче.

Определены 3 действия:

1) Drive(x,y) - действие передвижение вездехода из точки x в точку y. Предусловием действия drive(x,y): вездеход находится в точке x. Эффект действия drive(x,y): вездеход находится в точке y и робот не находится в точке x.
2) Sample(d,x) – действие сбора информации о ресурсе d в точке x. Предусловие действия sample(d,x): вездеход находится в точке x, в точке x находится ресурс d. Эффект действия sample(d,x): собраны данные о ресурсе d.
3) Commun(d) - передача информации о ресурсе d. Предусловие действия commun(d): собраны данные о ресурсе d. Эффект действия commun(d): данные о ресурсе d переданы.

В домене «Перевозки грузов» требуется составить план доставки нескольких пакетов в различные места. Планы составляются из следующих действий: загрузки и разгрузки грузовика; передвижения грузовика; загрузки и разгрузки самолёта; передвижения самолёта. Места находятся в городах. Грузовики перемещают грузы между местами, находящимися в одном городе. Самолёты доставляют грузы из аэропорта одного города в аэропорт другого города. В каждом городе по одному аэропорту и грузовику.

Описание домена «Перевозки грузов» находятся в приложении Б. В нём определены типы – PACKAGE (пакет), TRUCK (грузовик), LOCATION (место), AIRPLANE (самолёт), CITY (город), AIRPORT (аэропорт). 3 предиката определены без типизации параметров:

1) at(x,y) – описывает нахождение объекта x в объекте y;
2) in(x,y) – описывает нахождение объекта x внутри объекта y;
3) in-city(x,y) – описывает расположение объекта x в объекте y.

Над данными предикатами определено 6 действий:

1) Load-truck(p,t,x) - действие загрузки пакета p в грузовик t в месте x. Предусловие действия load-truck(p,t,l): грузовик t находится в месте x и пакет p

находится в месте x. Эффект действия load-truck(p,t,l):пакет p не находится в месте x и пакет p находится внутри грузовика t.

2) Load-airplane(p,a,x) - действие загрузки пакета p в самолёт a в аэропорту x. Предусловие действия load-airplane(p,a,x): самолёт a находится в аэропорту x и пакет p находится в аэропорту x. Эффект действия load-airplane(p,a,x):пакет p не находится в аэропорту x и пакет p находится внутри самолёта a.
3) Unload-truck(p,t,x) - действие разгрузки пакета p из грузовика t в месте x. Предусловие действия unload-truck(p,t,l): грузовик t находится в месте x и пакет p находится внутри грузовика t. Эффект действия unload-truck(p,t,l):пакет p находится в месте x и пакет p не находится внутри грузовика t.
4) Unload-airplane(p,a,x) - действие загрузки пакета p в самолёт a в аэропорту x. Предусловие действия unload-airplane(p,a,x): самолёт a находится в аэропорту x и пакет p находится внутри самолёта a. Эффект действия unload-airplane(p,a,x):пакет p находится в аэропорту x и пакет p не находится внутри самолёта a.
5) Drive-truck(t,x,y,c) – действие передвижения грузовика t из места x в место y по городу c. Предусловие действия drive-truck(t,x,y,c): грузовик t находится в месте x, места x, y располагаются в городе c. Эффект действия drive-truck(t,x,y,c): грузовик t находится в месте y и грузовик t не находится в месте x.
6) Fly-airplane(a,x,y) – действие передвижения самолёта a из аэропорта x в аэропорт y. Предусловие действия fly-airplane(a,x,y): самолёт a находится в аэропорту x. Эффект действия fly-airplane(a,x,y): самолёт a находится в аэропорту y и самолёт a не находится в аэропорту x.

Для экспериментов системы планирования решений сформулируем задачи различной сложности в описанных доменах. Сложность задачи зависит от:

- сложности действий в домене (количество предикатов в эффекте и предусловии);
- количества различных действий в домене;
- количества объектов в задаче.

Условно разделим задачи на 3 типа сложности – простые, средние, сложные. В домене «Вездеход» реализуем 3 задачи. В первой задаче будет 3 точки и 3 ресурса, которые находятся в этих трёх точках соответственно; в начальном состоянии вездеход находится в точке №1, никакие данные о ресурсах не переданы, не

собраны; целевое состояние – вездеход должен отправить данные о всех ресурсах. Так как в данной задаче небольшое количество объектов, действий не так много, и они имеют от в эффекте и предусловии 2-3 предиката, то отнесём её к простому типу. Её имя «Т1». Аналогично сформулируем задачу с 10 точками и 10 ресурсами. Её отнесём к задачам средней сложности за счёт количества объектов. Имя – «Т3».Последняя задача – задача с 30 точками и 30 ресурсами будет отнесена к сложным задачам так же за счёт количества объектов. Имя – «Т5».

В домене «Перевозки грузов» опишем 3 задачи. Такие объекты в первой задаче: 2 города, 2 грузовика, 4 места, 2 аэропорта (они также являются местом), пакет, самолет. В начальном состоянии 2 места располагаются в городе №1, 2 – в городе №2, в каждом городе располагается аэропорт, самолёты находятся в аэропортах, пакет и грузовик №1 находятся в месте №1, грузовик №2 находится в месте №2 города №2. Целевое состояние: пакет находится в месте №1 города №2. Данная задача отнесена к простому типу, так как объектов в не так много. Её имя «Т2». Во второй задаче будет 8 пакетов, 2 самолёта, 3 города, 3 грузовика, 3 аэропорта (также являются местом), 3 места. В каждом городе расположены: место, аэропорт. В одном месте каждого города находится грузовик. Пакеты находятся в месте и аэропорте первого города. Целевое состояние: доставить 4 пакета в другие места города№1, 2 пакета в место и аэропорт города №2, 2 пакета в место и аэропорт города №3. Данная задача имеет большее количество объектов. В домене определены 6 действий, количество предикатов в эффекте и предусловии варьируется от 3 до 5. Данную задачу отнесём к среднему типу. Её имя – «Т4». В последняя задача из этого домена определим объекты: 9 пакетов, 5 городов, 5 мест, 5 грузовиков, 5 аэропортов, 2 самолёта. Аналогично предыдущим задачам задаются начальное и конечное состояния. В данной задаче много объектов, её отнесём к сложным задача, имя – «Т6».

Для домена «Ханойские башни» опишем такую задачу: возьмём 8 дисков и 3 места для дисков; все диски в начальном состоянии расположены на первом месте для дисков; целевое состояние – все диски находятся на 3 месте для дисков. Не смотря на то, что в данном домене всего одно действие, количество дисков и мест для дисков выбрано с той целью, чтобы задача решалась «тяжело». Объекты задачи данного домена имеют один тип - данным определением увеличивается объём перебора. Задача отнесена к сложному типу. Её имя «Т7».

В набор вошло 7 задач с действиями различной сложности, с различным количеством действий и различным количеством объектов. Описание задач на входном языке системы планирования представлено в приложении Б.

Для каждой задачи из набора были подсчитаны временные затраты на ее решение с использованием различных программ-планировщиков по следующему сценарию:

1. Выбрать программу-планировщик. Если замерено время для всех планировщиков, то окончить испытание.
2. Запустить программу, подав на вход условие задачи, и замерить время, затраченное на решение.
3. Заполнить ячейку таблицы по следующим правилам: если программа не смогла построить план, то вписать пометку «Ошибка»; если план построен программой за время, превышающее 2 минуты, то вписать пометку «Таймаут»; если план построен не более чем за 2 минуты, то вписать в ячейку время, потраченное на поиск плана.
4. Перейти на шаг 1.

Работа программ замерялась на компьютере с 2-х ядерным процессором с тактовой частотой 2,4 GHz и 2 ГБ оперативной памяти. При этом программам для работы отдавался 1 процессор.

В эксперименте были задействованы следующие решатели, которые, так или иначе, используют граф планирования:

- Blackbox[12];
- Ff[13];
- Gp-csp[14];
- реализованная система решения задач, строящая план прямым ходом;
- реализованная система решения задач, строящая план прямым и обратным ходом.

Ниже приведена таблица, заполненная результатами экспериментов.

Задача / Решатель	T1	T2	T3	T4	T5	T6	T7
Blackbox	0.20	0.14	Таймаут	7.24	Таймаут	11.18	Таймаут
Ff	0.10	Ошибка	0.30	Ошибка	0.36	Ошибка	1.12
Gpcsp	0.12	0.20	Таймаут	Таймаут	Таймаут	Таймаут	Таймаут
Решатель*	0.05	0.11	0.29	0.85	2.45	14.15	6.92
Решатель**	0.05	0.12	0.26	1.45	4.45	Таймаут	7.59

Таблица 1 Время работы систем планирования на модельных задачах (в секундах).

Условные обозначения: * - реализованная система запущена с использованием прямого хода поиска решения;

** - реализованная система запущена с использованием попеременно прямого/обратного хода поиска решения.

Появление меток «Таймаут» в таблице объясняется большим размером пространств состояний в задачах средней и высокой степени сложности, а также использованием эвристик, недостаточно эффективных в данных задачах. Так в задаче «T7» по результатам работы решателя в графе планирования было 443 вершины, в графе пространства состояний - 2972 вершины. В задаче «T5»: граф планирования содержит 1050 вершин, граф пространства состояний - 3048 вершины. В задаче «T6»: граф планирования включает 758 вершин, граф пространства состояний - 3546 вершины.

Из таблицы видно, что реализованная система планирования для большинства рассмотренных задач строит планы быстрее, чем планировщики Blackbox и Gpcsp, но на задачах «T5», «T7» проигрывает планировщику Ff, а на задаче «T6» – планировщику Blackbox. В тоже время планировщик Ff не справился с поиском планов в задачах «T2», «T4», «T6». Можно сделать вывод, что на данном наборе задач созданная в рамках дипломной работы система планирования работает достаточно эффективно по сравнению с другими планировщиками.

Ниже приведена таблица, полученная аналогично предыдущей, в ячейках которой находятся длины планов (количество действий в плане), построенных

системами, участвующими в экспериментах. В нижней строке указаны длины оптимальных планов. Таблица

Задача / Решатель	T1	T2	T3	T4	T5	T6	T7
Blackbox	8	10	Таймаут	65	Таймаут	92	Таймаут
Ff	8	Ошибка	29	Ошибка	89	Ошибка	255
Gpcsp	8	10	Таймаут	Таймаут	Таймаут	Таймаут	Таймаут
Решатель*	8	10	29	72	89	118	255
Решатель**	8	10	29	72	89	Таймаут	255
Опт. длина	8	10	29	63	89	Неиз-вестно	255

Таблица 2 Длины планов, построенных системами планирования на модельных задачах.

Из таблицы видно, что планировщики для большинства рассмотренных задач строят оптимальные планы. Различия появляются в задачах «T4» и «T6», в которых достаточно много объектов, а в домене данных задач – действий.

Из данных, представленных в таблице №2, можно сделать вывод, что реализованная система на данном наборе задач строит планы близкие к оптимальным планам.

Заключение

Основные результаты дипломной работы состоят в следующем:

Разработана система решения задач, которая является независимым от предметной области планировщиком, осуществляющим поиск близкого к оптимальному плана в пространстве состояний с использованием эвристики, извлекаемой из графа планирования. При реализации системы был успешно использован и апробирован способ описания исходной задачи и особенностей предметной области на основе языка PDDL.

Измерение показателей работы созданной системы на примерах и сравнение с показателями, зафиксированными при работе других планировщиков на тех же примерах, позволяет сделать вывод о достаточно эффективной работе системы.

Система решения задач была реализована на языке программирования C++ в среде Microsoft Visual Studio 2005.

Литература

1. Корухова Л.С., Любимский Э.З., Соловская Л.Б. Альтернативное немонотонное планирование на основе графа. Препринт № 108. М.: ИПМ им. М. В. Келдыша РАН, 1995.
2. Нильсон Н. Принципы искусственного интеллекта. М.: Радио и связь, 1985.
3. Братко И. Алгоритмы искусственного интеллекта на языке Prolog. М.: Вильямс, 2004.
4. С. Рассел, П. Норвиг Искусственный интеллект: Современный подход. М.: Вильямс, 2006.
5. Лишнер Р. STL. Карманный справочник. – М.: Питер, 2005.
6. Jiménez P., Torras C. An efficient algorithm for searching implicit AND/OR graphs with cycles. // Institut de Robòtica i Informàtica Industrial (CSIC—UPC), Gran Capità 2-4 (Edifici NEXUS) E-08034 Barcelona, Spain, 17 May 2000.
7. Bryce D., Kambhampati S. A Tutorial on Planning Graph–Based Reachability Heuristics. // International Conference on Automated Planning and Scheduling'06 Tutorial 6, 2006.
8. Ghallab M., Howe A., Ram A., Veloso M., Weld D., Wilkins D. PDDL - The Planning Domain Definition Language Yale Center for Computational Vision and Control Tech Report CVC TR-98-003/DCS TR-1165 October, 1998.
9. Blum A. Fast Planning Through Planning Graph Analysis // Artificial Intelligence, Intelligence, 90:281–300, 1997.
10. Fikes R., Nilsson N. STRIPS: a new approach to the application of theorem proving to problem solving. // Artificial Intelligence, 2:189-208, 1971.
11. Pednault E. ADL: Exploring the middle ground between STRIPS and the situation calculus // 1-st Int. Conf. on Principles of Knowledge Representation and Reasoning, 1989, p. 324-332.
12. Henry A. Kautz, Bart Selman Unifying SAT-based and Graph-based Planning // In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, 1999, p. 318 – 325.
13. Kautz H. The FF Planning System: Fast Plan Generation Through Heuristic Search. // Artificial Intelligence Research, Volume 14, 2001, Pages 253 – 302.

14. Do M., Kambhampati S. Solving planning-graph by compiling it into CSP. // International Conference on AI Planning & Scheduling, 2000, p. 82-91.

Приложение А. РБНФ подмножества языка PDDL для системы решения задач

Ниже приведена расширенная форма Бэкуса-Наура ограниченного языка PDLL, на котором можно описать домены и задачи над этими доменами для системы решения задач.

РБНФ для описания доменов:

<domain> ::= (define (domain **<name>**) (:requirements :strips :typing) **<types-def>** **<predicates-def> <actions-def>**)

<types-def>::= (:types **<type>**$^{+}$)

<type>::=<name>

<variable>::= ?**<name>**

<predicates-def>::= (:predicates **<atomic-formula-with-or-without-types>**$^{+}$)

<atomic-formula-with-or-without-types>::= (**<predicate> <list-varibles-with-or-without -type>***)

<predicate>::=<name>

<list-varibles-with-or-without -type>::= <variable>$^{+}$ - **<type> | <variable>**$^{+}$

<actions-def>::=<action>$^{+}$

<action>::= (:action **<name>** :parameters (**<list-varibles-with-type>**) :precondition (and **<atomic-formula>**$^{+}$) :effect (and **<atomic-formula>**$^{+}$))

<atomic-formula>::= (not (**<predicate> <variable>***)) **|** (**<predicate> <variable>***)

РБНФ для описания задач:

<task>::= (define (problem **<name>**) (:domain **<name>**) **<list-objects> <list-init>** **<list-goal>**)

<list-objects>::= (:objects **<objects-types>**$^{+}$)

<objects-types>::=<object>$^{+}$ - **<type>|** (either **<type>**$^{+}$)

<object>::=<name>

<list-init>::= (:init **<list-predicates-initialization >**$^{+}$)

<list-predicates-initialization>::= (**<predicate> <object>***)

<list-goal>::= (:goal (and **<list-predicates-initialization >**$^{+}$))

Приложение Б. Домены и задачи

Домен «Ханойские башни»:

```
(define (domain hanoi)
(:requirements :strips :typing)
(:types DISCC)
(:predicates (clear ?x – DISCC)
          (on ?x ?y – DISCC)
          (smaller ?x ?y – DISCC))
(:action move
:parameters (?disc ?from ?to – DISCC)
:precondition (and (smaller ?to ?disc)
                (on ?disc ?from)
                (clear ?disc)
                (clear ?to))
:effect  (and (clear ?from)
           (on ?disc ?to)
           (not (on ?disc ?from))
           (not (clear ?to)))
 ))
```

Задача о ханойских башнях:

```
(define (problem hanoi8)
(:domain aria)
(:objects peg1 peg2 peg3 d1 d2 d3 d4 d5 d6 d7 d8 – DISCC)
(:init
(smaller peg1 d1)(smaller peg1 d2)(smaller peg1 d3)(smaller peg1 d4)(smaller peg1
d5)(smaller peg1 d6)(smaller peg1 d7)(smaller peg1 d8)
(smaller peg2 d1)(smaller peg2 d2)(smaller peg2 d3)(smaller peg2 d4)(smaller peg2
d5)(smaller peg2 d6)(smaller peg2 d7)(smaller peg2 d8)
(smaller peg3 d1)(smaller peg3 d2)(smaller peg3 d3)(smaller peg3 d4)(smaller peg3 d5)
(smaller peg3 d6)(smaller peg3 d7)(smaller peg3 d8)
```

(smaller d2 d1)(smaller d3 d1)(smaller d3 d2)(smaller d4 d1)(smaller d4 d2)(smaller d4 d3)(smaller d5 d1)(smaller d5 d2)(smaller d5 d3)(smaller d5 d4)(smaller d6 d1)(smaller d6 d2)(smaller d6 d3)(smaller d6 d4)(smaller d6 d5)(smaller d7 d1)(smaller d7 d2)(smaller d7 d3)(smaller d7 d4)(smaller d7 d5)(smaller d7 d6)(smaller d8 d1)(smaller d8 d2)(smaller d8 d3)(smaller d8 d4)(smaller d8 d5)(smaller d8 d6)(smaller d8 d7)
(clear peg2)(clear peg3)(clear d1)
(on d8 peg1)(on d7 d8)(on d6 d7)(on d5 d6)(on d4 d5)(on d3 d4)(on d2 d3)(on d1 d2))
(:goal (and
(on d8 peg3) (on d7 d8) (on d6 d7)(on d5 d6)(on d4 d5)(on d3 d4)(on d2 d3)(on d1 d2))))

Домен “Вездеходы”:

```
(define (domain rovers_classical)
(:requirements :strips :typing)
(:types location data)
(:predicates
(at ?x - location)
(avail ?d - data ?x - location)
(comm ?d - data)
(have ?d - data))
(:action drive
:parameters (?x ?y - location)
:precondition (and(at ?x))
:effect (and (at ?y) (not(at ?x))))
(:action commun
:parameters (?d - data)
:precondition (and(have ?d))
:effect (and(comm ?d)))
(:action sample
:parameters (?d - data ?x - location)
:precondition (and (at ?x) (avail ?d ?x))
:effect (and(have ?d)))
)
```

Задача с вездеходом (3 вершины):

```
(define (problem rovers_classical1)
```

```
(:domain rovers_classical)
(:objects
soil water rock - data
alpha beta gamma  - location)
(:init (at alpha)
(avail soil alpha)
(avail rock beta)
(avail water gamma)
)
(:goal (and
(comm soil)
(comm water)
(comm rock)
(at alpha)))
)
```

Задача с вездеходом (10 вершин):

```
(define (problem rovers_classical1)
(:domain rovers_classical)
(:objects
soil soil1 soil2 soil3 soil4 soil5 soil6 soil7 soil8 soil9 - data
alpha alpha1 alpha2 alpha3 alpha4 alpha5 alpha6 alpha7 alpha8 alpha9 - location)
(:init (at alpha)
(avail soil alpha)(avail soil1 alpha1)(avail soil2 alpha2)(avail soil3 alpha3)(avail soil4
alpha4)(avail soil5 alpha5)(avail soil6 alpha6)(avail soil7 alpha7)(avail soil8 alpha8)(avail
soil9 alpha9))
(:goal (and
(comm soil)(comm soil1)(comm soil2)(comm soil3)(comm soil4)(comm soil5)(comm
soil6)(comm soil7)(comm soil8)(comm soil9)
(at alpha)))
)
```

Задача с вездеходом (30 вершин):

```
(define (problem rovers_classical1)
(:domain rovers_classical)
```

(:objects
soil soil1 soil2 soil3 soil4 soil5 soil6 soil7 soil8 soil9 water water1 water2 water3 water4 water5 water6 water7 water8 water9 rock rock1 rock2 rock3 rock4 rock5 rock6 rock7 rock8 rock9 - data
alpha alpha1 alpha2 alpha3 alpha4 alpha5 alpha6 alpha7 alpha8 alpha9 beta beta1 beta2 beta3 beta4 beta5 beta6 beta7 beta8 beta9 gamma gamma1 gamma2 gamma3 gamma4 gamma5 gamma6 gamma7 gamma8 gamma9 - location)
(:init (at alpha)
(avail soil alpha)(avail soil1 alpha1)(avail soil2 alpha2)(avail soil3 alpha3)(avail soil4 alpha4)(avail soil5 alpha5)(avail soil6 alpha6)(avail soil7 alpha7)(avail soil8 alpha8)(avail soil9 alpha9)(avail rock beta)(avail rock1 beta1)(avail rock2 beta2)(avail rock3 beta3)(avail rock4 beta4)(avail rock5 beta5)(avail rock6 beta6)(avail rock7 beta7)(avail rock8 beta8)(avail rock9 beta9) (avail water gamma)(avail water1 gamma1)(avail water2 gamma2)(avail water3 gamma3)(avail water4 gamma4)(avail water5 gamma5)(avail water6 gamma6)(avail water7 gamma7)(avail water8 gamma8)(avail water9 gamma9))
(:goal (and
(comm soil)(comm soil1)(comm soil2)(comm soil3)(comm soil4)(comm soil5)(comm soil6)(comm soil7)(comm soil8)(comm soil9)(comm water)(comm water1) (comm water2)(comm water3)(comm water4)(comm water5)(comm water6)(comm water7)(comm water8) (comm water9)(comm rock)(comm rock1)(comm rock2)(comm rock3)(comm rock4)(comm rock5)(comm rock6)(comm rock7)(comm rock8)(comm rock9)
(at alpha)))
)

Домен «Перевозки грузов»:

```
 (define (domain logistics-typed)
  (:requirements :strips :typing)
  (:types PACKAGE TRUCK LOCATION AIRPLANE CITY AIRPORT)
  (:predicates
                (at ?obj ?loc)
                (in ?obj ?vehicle)
                (in-city ?loc-or-truck ?city))
  (:action LOAD-TRUCK
        :parameters
```

```
            (?obj - PACKAGE
             ?truck - TRUCK
             ?loc - LOCATION)
    :precondition
            (and   (at ?truck ?loc)
                   (at ?obj ?loc))
    :effect
            (and   (not (at ?obj ?loc))
                   (in ?obj ?truck)))
(:action LOAD-AIRPLANE
    :parameters
            (?obj - PACKAGE
             ?airplane - AIRPLANE
             ?loc - AIRPORT)
    :precondition
            (and
                   (at ?obj ?loc)
                   (at ?airplane ?loc))
    :effect
            (and   (not (at ?obj ?loc))
                   (in ?obj ?airplane)))
(:action UNLOAD-TRUCK
    :parameters
            (?obj - PACKAGE
             ?truck - TRUCK
             ?loc - LOCATION)
    :precondition
            (and    (at ?truck ?loc)
                   (in ?obj ?truck))
    :effect
            (and   (not (in ?obj ?truck))
                   (at ?obj ?loc)))
(:action UNLOAD-AIRPLANE
```

```
        :parameters
                (?obj - PACKAGE
                 ?airplane - AIRPLANE
                 ?loc - AIRPORT)
        :precondition
                (and    (in ?obj ?airplane)
                        (at ?airplane ?loc))
        :effect
                (and
                        (not (in ?obj ?airplane))
                        (at ?obj ?loc)))
(:action DRIVE-TRUCK
        :parameters
                (?truck - TRUCK
                 ?loc-from - LOCATION
                 ?loc-to - LOCATION
                 ?city - CITY)
        :precondition
                (and    (at ?truck ?loc-from)
                        (in-city ?loc-from ?city)
                        (in-city ?loc-to ?city))
        :effect
                (and    (not (at ?truck ?loc-from))
                        (at ?truck ?loc-to)))
(:action FLY-AIRPLANE
        :parameters
                (?airplane - AIRPLANE
                 ?loc-from - AIRPORT
                 ?loc-to - AIRPORT)
        :precondition
                (and(at ?airplane ?loc-from))
        :effect
                (and    (not (at ?airplane ?loc-from))
```

```
            (at ?airplane ?loc-to)))
)
```

Простая задача планирования перевозки грузов:

```
(define (problem log001)
   (:domain logistics-typed)
   (:objects
      package1 - PACKAGE
      airplane1 - AIRPLANE
      pgh   bos - CITY
      pgh-truck   bos-truck - TRUCK
      pgh-po   bos-po   pgh-central  bos-central - LOCATION
      pgh-airport   bos-airport - (either AIRPORT LOCATION)
   )
   (:init
      (in-city pgh-po pgh) (in-city pgh-airport pgh) (in-city pgh-central pgh) (in-city bos-po
bos) (in-city bos-airport bos) (in-city bos-central bos) (at package1 pgh-po) (at airplane1
pgh-airport) (at bos-truck bos-po) (at pgh-truck pgh-po)    )
   (:goal (and
      (at package1 bos-po)
  ))
)
```

Средняя задача планирования перевозки грузов :

```
(define (problem log004)
   (:domain logistics-typed)
   (:objects
      package1 package2 package3 package4 package5 package6 package7 package8 -
PACKAGE
      airplane1 airplane2 - AIRPLANE
      pgh bos  la - CITY
      pgh-truck  bos-truck  la-truck - TRUCK
      pgh-po  bos-po  la-po - LOCATION
      pgh-airport  bos-airport  la-airport - (either LOCATION AIRPORT)
   )
```

(:init
(in-city pgh-po pgh) (in-city pgh-airport pgh)
(in-city bos-po bos) (in-city bos-airport bos)
(in-city la-po la) (in-city la-airport la)
(at package1 pgh-po) (at package2 pgh-po) (at package3 pgh-po) (at package4 pgh-po) (at package5 bos-po) (at package6 bos-po) (at package7 bos-po) (at package8 la-po)
(at airplane1 pgh-airport) (at airplane2 pgh-airport)
(at bos-truck bos-po) (at pgh-truck pgh-po) (at la-truck la-po)
)
(:goal (and
(at package1 bos-po) (at package2 bos-airport) (at package3 la-po) (at package4 la-airport) (at package5 pgh-po) (at package6 pgh-airport) (at package7 pgh-po) (at package8 pgh-po)
))
)

Сложная задача планирования перевозки грузов:

(define (problem log007)
(:domain logistics-typed)
(:objects
package1 package2 package3 package4 package5 package6 package7 package8 package9 – PACKAGE
airplane1 airplane2 – AIRPLANE
pgh bos la ny sf – CITY
pgh-truck bos-truck la-truck ny-truck sf-truck – TRUCK
pgh-po bos-po la-po ny-po sf-po pgh-central bos-central la-central ny-central sf-central – LOCATION
pgh-airport bos-airport la-airport ny-airport sf-airport – (either LOCATION AIRPORT)
)
(:init
(in-city pgh-po pgh) (in-city pgh-airport pgh) (in-city pgh-central pgh) (in-city bos-po bos) (in-city bos-airport bos) (in-city bos-central bos)
(in-city la-po la) (in-city la-airport la) (in-city la-central la)

(in-city ny-po ny) (in-city ny-airport ny) (in-city ny-central ny) (in-city sf-po sf) (in-city sf-airport sf) (in-city sf-central sf)

(at package1 pgh-po) (at package2 pgh-central) (at package3 pgh-central) (at package4 ny-po) (at package5 bos-po) (at package6 bos-po) (at package7 ny-po) (at package8 sf-airport) (at package9 sf-central) (at airplane1 pgh-airport) (at airplane2 pgh-airport)

(at bos-truck bos-po) (at pgh-truck pgh-airport) (at la-truck la-po) (at ny-truck ny-central) (at sf-truck sf-airport)

)

(:goal (and

(at package1 bos-po) (at package2 ny-po) (at package3 la-central) (at package4 la-airport) (at package5 pgh-po) (at package6 ny-central) (at package7 pgh-po) (at package8 ny-central) (at package9 sf-po)

))

)