Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Ижевский государственный технический университет имени М. Т. Калашникова»

Кафедра «Программное обеспечение»

ПРОГРАММИРОВАНИЕ ЛОГИЧЕСКИХ ИГР

Учебно-методическое пособие по дисциплинам
«Системы искусственного интеллекта»,
«Математические основы искусственного интеллекта»,
«Методы и алгоритмы принятия решений» и
«Теория принятия решений. Методы и алгоритмы принятия решений»

Ижевск
Издательство ИжГТУ
2013

С о с т а в и т е л и :
А. В. Коробейников, канд. техн. наук, доц. кафедры «Программное обеспечение» ИжГТУ;
П. П. Лугачев, ст. преподаватель кафедры «Программное обеспечение» ИжГТУ

Рекомендовано к использованию на заседании кафедры «Программное обеспечение» ИжГТУ (протокол №37 от 15 октября 2013 г.).

Программирование логических игр : учебно-методическое пособие по дисциплинам «Системы искусственного интеллекта», «Математические основы искусственного интеллекта» и «Методы и алгоритмы принятия решений» и «Теория принятия решений. Методы и алгоритмы принятия решений» / сост. А. В. Коробейников, П. П. Лугачев. – Ижевск : Изд-во ИжГТУ, 2013. – 52 с.

Учебно-методическое пособие предназначено для дисциплин «Системы искусственного интеллекта» и «Математические основы искусственного интеллекта».

Пособие состоит из трех разделов. Первый раздел содержит обзор современного состояния программ для интеллектуальных логических игр (шахматы, шашки, го, спортивный бридж). Во втором разделе излагается теоретические обоснования решения задач путем поиска по пространству игровых состояний. В третьем разделе приведено задание для выполнения работ по дисциплинам.

Пособие предназначено для проведения практических занятий и выполнения лабораторных работ студентов, обучающихся по направлению подготовки 230100.62 «Информатика и вычислительная техника» и специальности 230105.65 «Программное обеспечение вычислительной техники и автоматизированных систем» при изучении дисциплины «Системы искусственного интеллекта» очной и заочной форм обучения.

Пособие предназначено для проведения практических занятий и выполнения курсовой работы студентов, обучающихся по направлению подготовки 231000.62 «Программная инженерия» по дисциплине «Математические основы искусственного интеллекта».

Пособие предназначено для выполнения курсовой работы для студентов, обучающихся специальности 230105.65 «Программное обеспечение вычислительной техники и автоматизированных систем» при изучении дисциплины «Методы и алгоритмы принятия решений».

Пособие предназначено для выполнения курсовой работы для студентов, обучающихся по направлению подготовки 230100.62 «Информатика и вычислительная техника» при изучении дисциплины «Теория принятия решений. Методы и алгоритмы принятия решений» очной и заочной форм обучения.

Оглавление

Введение

Традиционно одним из направлений систем искусственного интеллекта (СИИ) является решение интеллектуальных игровых задач. Многие ранние исследования в области поиска в пространстве состояний совершались на основе распространенных настольных игр: шашки, шахматы и пятнашки. Логические игры обладают интеллектуальным характером, при этом являются идеальным объектом для экспериментов из-за формализации игровых ситуаций и ограниченного набора игровых правил. Это позволяет легко строить пространство поиска и избавляет исследователя от неясности, присущей менее реальным проблемам. Позиции фигур легко представимы в компьютерной программе, они не требуют создания сложных формализмов, необходимых для передачи семантических тонкостей более сложных предметных областей. Тестирование игровых программ не порождает никаких финансовых или этических проблем. Поиск в пространстве состояний – принцип, лежащий в основе большинства исследований в области ведения игр.

В широком смысле слова под игрой понимается некая конфликтная ситуация, участники которой своими действиями не только достигают своих личных целей, но и влияют на достижимость целей другими участниками игры. Ясно, что под такое толкование игры подпадают многие экономические, политические и военные конфликты.

Игры могут порождать необычайно большие пространства состояний. Для поиска в них требуются мощные методики, определяющие, какие альтернативы следует рассматривать. Такие методики называются эвристиками и составляют значительную область исследований СИИ. Примеры эвристик: рекомендация проверять, включен ли прибор в розетку, прежде чем делать предположения о его поломке; выполнять рокировку в шахматной игре, чтобы попытаться уберечь короля от шаха. Большая часть того, что мы называем разумностью, опирается на эвристики, используемые человеком для решения задач.

Поскольку большинство знакомы с простыми играми, можно попробовать самостоятельно разработать свои эвристики и испытать их эффективность. Наличие противника усложняет структуру программы, добавляя в нее элемент непредсказуемости.

Учебное пособие направлено на получение практических навыков решения сложных задач на примере логических игр и закрепление теоретических знаний методом поиска по дереву состояний.

1. Обзор состояния игровых программ

1.1. Всемирные интеллектуальные игры

Интеллиады (*World Mind Sports Games*, *WMSG*) – комплексные соревнования в интеллектуальных видах спорта. I Всемирные интеллектуальные игры прошли в Пекине в 2008 г. Всего было разыграно 35 комплектов наград в 5 интеллектуальных видах спорта (шахматы, бридж, го, шашки, китайские шахматы сянци).

Все последующие подобные мероприятия будут называться Всемирные интеллектуальные игры «Спорт-Аккорд» (*Sport Accord World Mind Games*). Первые игры с новым названием состоялись в Пекине в декабре 2011 г. (по шахматам, бриджу, го и шашкам).

Ниже в этом разделе рассмотрим состояние программ для игры в интеллектуальные игры: шахматы, шашки, го и спортивный бридж.

1.2. Программы для игры в шахматы

Первая шахматная игровая машина создана в 1769 г. венгерским инженером фон Компеленом для развлечения королевы. Машина неплохо играла: внутри сидел сильный шахматист, который и делал ходы.

Число всех возможных позиций, не нарушающих правила на шахматной доске составляет ориентировочно 10^{46}.

Теорема о минимаксе была доказана фон Нейманом в статье «К теории стратегических игр» в 1928 г., что определило становление теории игр в качестве самостоятельного раздела математики.

В 1951 г. Тьюринг написал первый алгоритм шахматной программы. Шеннон пишет статью о программировании шахмат и предлагает 2 варианта шахматных алгоритмов: просмотр дерева на одинаковую высоту по всем ходам и более глубокий просмотр важных вариантов.

В 1952 г. написана программа для компьютера *MANIAC*1. Принятие решение о ходе – 12 минут, а просмотр дерева выполнялся на 4 полухода вперед (по 2 хода «черных» и «белых»). В 1957 г. программа для IBM704: просмотр на 4 полухода, принятие решения о ходе – 8 минут.

В 1962 г. Ньюэл, Саймон и Шоу открывают алгоритм, получивший название, альфа-бета отсечение. Считается, что этот алгоритм ранее открыл советский ученый Брудно.

«Каисса» – советская шахматная программа – в 1974 г. стала первым чемпионом мира по шахматам среди программ (Битман, Арлазоров).

В 1975 г. Хьят начинает разработку программу *CrayBlitz*, которая стала чемпионом среди программ в 1983–1989 гг. В 1983 г. она просматривала 40–50 тысяч позиций в секунду.

В 1977 г. Томпсон и Кондон создают первый специализированный шахматный компьютер, в котором эвристическая оценка игровых состояний выполнялась аппаратно (180 тысяч состояний в секунду, просмотр на 8-9 полуходов).

Советский шахматист и ученый Ботвинник с начала 1970-х гг. руководил созданием шахматной программы, которую в дальнейшем адаптировал для решения задач экономики и планирования.

В 1980-х гг. Берлингер создает *HiTech*, содержащий 64 шахматные микросхемы. Затем его студенты Хэй и Кембелл разрабатывают шахматный компьютер Chip Test, а позже *DeepTrought* (250 шахматных микросхем), оценивающий 750 позиций в секунду.

1992 г. – еще одна известная отечественная программа «Мираж» (Рыбинкин – Шпеер). Современная программа *SmartThink* (Марков).

Отметим также программу «Мефисто», использующую алгоритм выборочного поиска (Лэнг) и написанную на ассемблере.

Наконец в 1997 г. шахматный суперкомпьютер *DeepBlue* (развитие *DeepTrought*) фирмы *IBM* одержал победу в матче из 6 партий с чемпионом мира по шахматам Каспаровым. Оценочная функция использовала около 6000 показателей, просмотр на 8–12 полуходов, оценка 1 млн позиций в секунду. *DeepBlue* состоял из 32 процессоров, в каждом 16 шахматных микросхем.

Сегодня лучшими программами являются: *X3D Fritz* и *DeepFritz*. Отметим алгоритм (null move), позволяющий углубить поиск и использованный в *Fritz*: построение дерева без учета хода противника.

1.3. Программы для игры в шашки

Число возможных позиций, не нарушающих правила, – около 10^{20}.

Наиболее важным исследованием программной игры в шашки считается работа Сэмюэля 1959 г., в которой впервые были реализован ряд важных форм обучения. Накопление – хранение в памяти большого числа просчитанных игровых ситуаций, что сокращает время принятия решения. Обобщение – вычисление эвристической оценки игровой ситуации, как взвешенной суммы набора параметров, описывающих ситуацию. Самообучение – использование нескольких

версий программы: программа обучается сама на себе.

На равных с сильнейшими людьми игроки-программы научились выступать в 1990 г. Программа *Chinook*, она играла в 1990-х гг. с чемпионом мира Тинсли.

В 2007 г. шашки оказались полностью «взломанными», то есть были просчитаны на суперкомпьютере все их возможные ходы и комбинации. Благодаря этому последняя версия шашечной программы *Chinook* стала беспроигрышной. Считается, что человек у нее не сможет выиграть никогда, поскольку машина в абсолютно любой позиции всегда знает самый лучший ход.

1.4. Программы для игры в го

Число возможных конечных позиций в партии го (на поле 19×19) составляет 10^{171}, к которым можно прийти одним из 10^{1100} путей.

Го – древняя игра, даже более сложная, чем шахматы. Сложность го для компьютера вызвана двумя основными факторами. Первый фактор – большое число вариантов ходов, практически равное числу оставшихся свободных пунктов. Это полностью исключает возможность «механического» перебора позиций, используемого в нынешних шахматных программах. Второй фактор – сложность формализованной оценки позиции, при этом особенности игры позволяют человеку сравнительно просто видеть очень длинные цепочки возможных ходов, выстраивая на этой основе тактику и стратегию.

Первого серьезного успеха программы, играющие в го, достигли в 2009 г. Программа *MoGo* обыграла двух профессионалов Чиэня и Чжоу на престижном турнире по го *Taiwan Open* с форой, соответственно, в 6 и 7 камней. Учитывая гандикап, данный машине, она выступила на уровне самого сильного любителя. Другая программа го *Many Faces* в Чикаго обыграла одного из сильнейших игрока США в го Кервина, хотя опять машина имела гандикап в 7 камней.

Создатели *Many Faces*, фирма *Smart Games* из США во главе с Фотландом воспользовались алгоритмом на основе дерева поиска Монте-Карло, который был развит французскими специалистами по ИИ в 2006 г. *MoGo* пользуется сходным же методом, хотя и отличным в деталях. Задача решается не перебором всех возможных комбинаций партии в лоб, а путем просчета конечных результатов ограниченного числа случайных событий.

1.5. Программы для игры в спортивный бридж

1.5.1. Описание игры

Бридж (*bridge*) – карточная интеллектуальная командная или парная игра. Спортивный бридж – единственная из карточных игр, признанная Международным олимпийским комитетом видом спорта. В спортивном бридже играется определенное количество сдач, и каждая сдача разыгрывается на несколько раз разными игроками с последующим сравнением результатов. Задача каждой пары – получить результат лучше других пар в том же раскладе. По одной из версий название произошло от старинной русской игры «бирюч», в свое время распространенной в среде русских посыльных (бирючей). Правила современного бриджа возникли в 1925 г. при участии Вандербилта.

За одним столом играют две пары игроков, игроки одной пары сидят друг напротив друга. Игроки называются по сторонам света: Север (*N*), Восток (*E*), Юг (*S*) и Запад (*W*). Пара *NS* играет против пары *EW*. Для игры используется стандартная колода из 52 карт от двойки до туза четырех мастей. При раздаче каждый игрок получает по 13 карт. Один из игроков является сдающим (*dealer*). Каждая раздача состоит из двух фаз – торговли и розыгрыша. Заявки в торговле и игра картами производятся игроками поочередно по часовой стрелке. Особенности бриджа: наличие кооперации и конкуренции, неполная информация о раскладе.

1.5.2. Программы торговли

Цель торговли – определить контракт: обязательство одной пары взять определенное количество взяток в назначенной ею деноминации (козырной масти или без козыря). Во время торговли игроки, начиная со сдающего, по очереди делают заявки.

Длительность последовательности торговли составляет от 4 до 319 заявок; число деноминаций – 5, число уровней контракта – 7.

Система торговли используется игроками одной пары, и ее содержание является общедоступным. Задачи торговли: обмен информацией об имеющихся у партнеров картах, нахождение оптимального контракта для данной сдачи карт и помеха подобным действиям соперников. Набор значений различных заявок пары называется системой торговли.

Как правило, системы торговли формируются путем умозрительных предположений, а также основываются на опыте игроков. Некоторые элементы торговли основываются на расчетах вероятностей

определенных раскладов карт (комбинаторные задачи).

Система торговли представляет из себя конечный автомат (КА). Переход в новое состояние из текущего выполняется под действием информации о раскладе карт руки игрока (входные данные) и при этом генерируется очередная заявка торговли (выходные данные).

Самый простой вариант при разработке программы торговли – это реализация известных правил общепринятых систем торговли. Однако при этом возникает проблема формализации неопределенностей, присущих описанию раскладов на естественном языке, а также проблема неполноты в описании игровых раскладов рук игроков.

Для преодоления этих проблем возможно использование алгоритма принятия решений в условиях неполной информации (*PIDM – Partial Information Decision Making*). Он основан на методе Монте-Карло и позволяет преодолевать неполноту информации. *PIDM* неявным образом моделирует процесс торговли и охватывает разные случаи, в которых игрок запрашивает у партнера информацию.

Для оценки типа расклада руки игрока используют искусственные нейронные сети (ИНС). Нет одной глобальной сети, но есть набор сетей для решения локальных задач. Такая программа предлагает верное решение приблизительно для 90–92 % раскладов тестовой выборки.

BRIBIP (*Bridge Bidding Program*) была разработана на языке *POPCORN*. Программа выполняет тщательный анализ торговли и розыгрыша для получения дополнительной информации. Затем эта информация используется при планировании торговли. Программа выполняет торговлю приблизительно на скорости игрока-человека.

Реальные игроки в бридж осваивают систему торговли шаг за шагом с помощью объяснений различных ситуаций со стороны более опытного спортсмена. Подобный подход возможен и для игрока-программы и известен в системах ИИ как обучение на основе объяснения (*Explanation-Based Learning – EBL*). Проблему противоречий в объяснениях предлагают решать с помощью абдуктивного *EBL* (*A-EBL*).

Отдельной задачей является поиск системы торговли, оптимальной на наборе сдач. Это задача эволюционного обучения – поиск оптимального КА с помощью генетического алгоритма (ГА). При этом рассматривается популяция систем торговли, ряд из которых тренеры (неизменны) для остальных систем (изменяемых). Система торговли хранится в виде И-ИЛИ-дерева, в узлах записываются условия для со-

стояний КА. Для преодоления неполноты и неопределенности информации в работе КА используется нечеткая логика. Возможна реализация КА посредством ИНС с обратными связями. Однако, несмотря на положительные результаты, при данном подходе невозможна трактовка каждой заявки, что обусловлено природой ИНС.

1.5.3. Программы розыгрыша

После торговли выполняется розыгрыш, который начинает оппонент разыгрывающего, сидящий слева. После этого «болван» (партнер разыгрывающего) кладет свои карты на стол и его картами управляет разыгрывающий. Игра ведется так же, как и в других играх на взятки, бить козырем необязательно. Результат розыгрыша – количество взяток, полученных каждой парой.

Количество вариантов раскладов в бридже по основной формуле комбинаторики составляет $1{,}6 \cdot 10^{31}$. Количество вершин в дереве игровых состояний при розыгрыше: $1{,}5 \cdot 10^{39}$, но с учетом требования ходить в масть карты захода примерно получим: $3{,}86 \cdot 10^{20}$.

Bridge Baron, 1983 г. – первая полностью автоматизированная программа по бриджу. Розыгрыш реализован на основе эвристик и методов планирования иерархической сети задач (*Hierarchical Task-Network – HTN*) для решения задачи игры с одним «болваном».

В 2000 г. Гинзберг предложил программу *GiB*, в которой розыгрыш сводился к двум этапам: с двумя «болванами» (полная информация) и одним «болваном» (неполная информация). При игре с двумя «болванами» предполагается, что известен расклад карт по рукам, для которого применяется метод минимакс с альфа-бета-отсечением и оценивается результат. Затем алгоритмом игры с двумя «болванами» определяется результат ограниченного подмножества сдач, входящего в множество сдач при игре с одним «болваном» – применяется метод Монте-Карло.

Отметим сходство применяемых алгоритмов для го и розыгрыша бриджа. Использование метода Монте-Карло в го обусловлено невозможностью поиска по всему дереву игровых состояний из-за комбинаторного взрыва, хотя полное дерево теоретически можно построить: игра с полной информацией. При розыгрыше в бридже информация неполная, и проверка всех возможных раскладов методом минимакс также приведет к комбинаторному взрыву.

Программа *SPAN* объединяет несколько различных задач при по-

иске тактики розыгрыша в согласованный пакет задач. На верхнем уровне системы есть планировщик со списком задач.

Планирование на основе доказательства теорем использует программа *FINESSE* для задачи розыгрыша. Программа написана на языке Пролог. Система показывает достаточно компетентную игру и с большой вероятностью получает максимальное число взяток.

Для игры в защите лучшая модель основана на эвристиках и предположении о самой сильной из возможных вариантов игровой руке разыгрывающего, также действует и человек. Новая эвристика бета-редукции и итерационного смещения представляет первый обобщенный алгоритм поиска по дереву, способный к последовательной игре.

Методы подрезки дерева минимаксного поиска могут привести к более точному результату в розыгрыше. Минимакс с предварительным отсечением (*forward pruning*) показывает лучшие результаты, чем обычный минимакс в случаях, где была высокая корреляция среди минимаксных ценностей узлов родного брата в дереве игры.

Розыгрыш может быть рассмотрен как планирование, с начальным состоянием, одним или более целевыми состояниями и группой операторов для преобразования состояния. План – упорядоченный набор операторов для преобразования состояний. При последовательном обучении на основе текущего момента (*Sequential Instant-Based Learning – SIBL*) действия выбираются на основе последовательности состояний. Алгоритм полагается на последовательные наблюдения, а не полный анализ задачи. Строится инкрементальная база данных из частей задачи, а затем алгоритм *SIBL* может принимать решения. Авторы сформировали базу из 936 случаев для выбора карты хода.

При решении задачи розыгрыша с одним «болваном» (неполная информация) строят минимаксное дерево на основе вероятностей положения неизвестных игровых карт, что можно считать эвристикой.

1.5.4. Чемпионаты среди компьютеров

В 1997 г. *ACBL* (*American Contract Bridge League*) провела первый чемпионат мира среди компьютерных программ, играющих в бридж (*World Computer-Bridge Championship*). Начиная с 1999 г. к организации присоединилась Мировая федерация бриджа (*WBF*). Чемпионат проводится ежегодно. Первым чемпионом стала программа *Bridge Baron*. В 2011 г. победителем среди 7 участников стала программа

Shark Bridge (Норрис, Дания). Последнее десятилетие господствовали программы *Jack* (авторы Куиджи и Хеемскерк, Нидерланды; чемпионы 2001–2004, 2006, 2009, 2010 гг.) и *Wbridge* (автор Костел, Франция; чемпионы 2005, 2007, 2008 гг.). Программы *Jack* и *Wbridge* по слухам написаны с использованием метода Монте-Карло и игры с двумя «болванами», впервые предложенные Гинзбергом.

Система для соревнования среди программ бриджу (своеобразный «стол») состоит из центрального сервера и узла управления столом, который распределяет сдачи игрового тура по подключенным компьютерам, на каждом из которых установлена программа-игрок в бридж. Перед началом матча операторы противников обмениваются конвенционными картами по системе торговле и вводят эту информацию в базы данных программ. Игра производится при помощи узла управления, через который программы-игроки обмениваются информацией. Программы-игроки установлены на персональных компьютерах офисной производительности. Скорость игры составляет 2 минуты на одну сдачу, что примерно в 2 раза меньше, чем для игроков-людей.

2. Теоретические основы

2.1. Поиск в пространстве состояний

Поиск в пространстве состояний – основной принцип большинства исследований в области ведения игр. Игры могут порождать большие пространства состояний, для поиска в которых используются эвристические методы, определяющие, какие альтернативы рассматривать.

Для разработки алгоритмов поиска необходимо анализировать и прогнозировать их работу. При этом возникают следующие вопросы.

Гарантировано ли нахождение решения в процессе поиска? Является поиск конечным или возможны зацикливания? Оптимально ли найденное решение? Как процесс поиска зависит от времени выполнения и используемой памяти? Как наиболее эффективно упростить поиск?

На эти вопросы отвечает теория поиска в пространстве состояний (*state space search*). Представив задачу в виде графа пространства состояний, можно использовать теорию графов для анализа структуры и сложности как самой задачи, так и процедуры поиска ее решения.

Граф состоит из множества вершин и дуг, соединяющих пары вершин. В модели пространства состояний решаемой задачи вершины графа представляют дискретные состояния процесса решения, например различные конфигурации игровой доски. Дуги графа описывают переходы между состояниями. Эти переходы соответствуют логическим заключениям или допустимым ходам в игре.

Одно или несколько начальных состояний, соответствующих исходной информации поставленной задачи, образуют корень дерева. Граф также включает одно или несколько целевых условий, которые соответствуют решениям исходной задачи. Поиск в пространстве состояний характеризует решение задачи как процесс нахождения пути решения от исходного состояния к целевому.

Пространство состояний представим следующими элементами:

– множество вершин графа (состояний) в процессе решения;

– множество дуг между вершинами (шаги процесса решения);

– непустое множество начальных состояний задачи;

– целевые состояния – непустое подмножество множества вершин.

Эти состояния описываются одним из следующих способов:

1. Измеряемыми свойствами состояний.

2. Свойствами путей, например стоимостью пути.

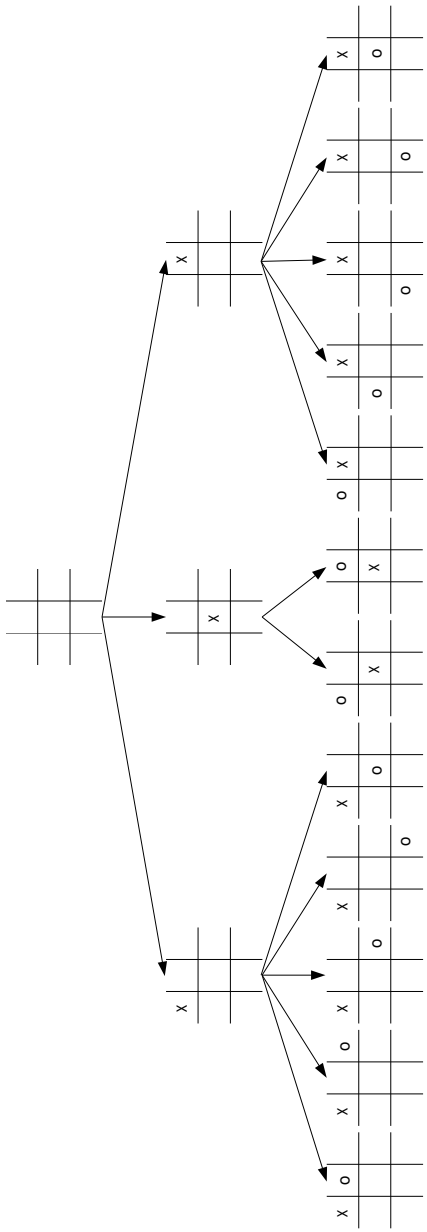

Рис. 2.1.1. Фрагмент пространства состояний игры «крестики-нолики»

Допустимым называется путь из начального состояния в целевое.

Цель может описывать состояние (например, выигрышную конфигурацию в игре), а также некоторые свойства допустимых путей.

Дуги в пространстве состояний соответствуют шагам процесса решения, а пути представляют решения на различной стадии завершения. Путь – цель поиска, начинается из исходного состояния и продолжается, пока не достигнуто условие цели. Порождение новых состояний вдоль пути выполняется на основе допустимых ходов игры.

Задача алгоритма поиска состоит в нахождении допустимого пути в пространстве состояний. Алгоритмы поиска должны направлять пути от начальной вершины к целевой. Одна из общих проблем, возникающих при поиске по графу: состояния иногда могут быть достигнуты разными путями. Например, вершины *L* и *P* на рис. 2.5.1. Поэтому важно выбрать оптимальный путь решения данной задачи. Кроме того, множественные пути к состоянию могут привести к петлям или циклам.

Игры «крестики-нолики», «пятнашки» (упрощенный вариант «8-головоломка» на рис. 2.6.1) используют для демонстрации поиска. Пространство состояний игры «крестики-нолики» изображено на рис. 2.1.1. Исходным состоянием является пустая игровая доска, а конечным или целевым состоянием – доска, на которой в одной строке, столбце или по диагонали располагается 3 крестика или 3 нолика в зависимости от игрока. Путь из исходного состояния в конечное содержит последовательность ходов, ведущих к победе игрока.

Состояниями в пространстве являются все возможные конфигурации из крестиков и ноликов, которые могут возникнуть в процессе игры. Конечно, большинство из возможных игровых состояний никогда не возникает в реальной игре. Граф пространства состояний не является деревом, поскольку некоторые состояния ниже второго уровня могут быть достигнуты различными путями. Однако в пространстве состояний нет циклов, так как убирать значки с игровой доски нельзя, то есть движение по графу выполняется по направленным дугам сверху вниз. Следовательно, нет необходимости проверки на цикл – ациклический ориентированный граф.

Вид пространства состояний дает возможность определить сложность задачи. В игре «крестики-нолики» возможны 9 первых ходов, 8 возможных ответов, а затем еще 7 возможных вариантов и так далее. Всего имеется $9 \cdot 8 \cdot 7 \cdot \ldots \cdot 1 = 9! = 362880$ возможных путей. Такое количе-

ство путей компьютер способен обработать полным перебором. Однако существует немало задач, имеющих очень высокую факториальную или экспоненциальную сложность, которая на много порядков выше (смотри раздел 1, обзор состояния). Такие пространства чрезвычайно сложны, и их невозможно исследовать полным перебором вариантов.

После первого хода в игре «крестики-нолики» возможно 9 состояний. Однако за счет поворотов и отражений игровой доски их можно свести к 3. На следующем уровне получим 9·8 состояний, число которых также можно сократить до 12. Подход выявления инвариантных признаков состояний позволяет сократить граф и для других игр.

На рис. 2.6.1 в игре «8-головоломка» 8 пронумерованных фишек на поле из 9 клеток. Одна клетка остается пустой, так что фишки можно двигать и получать их различные конфигурации. Цель игры – найти такую последовательность перемещений фишек в пустую клетку, которая привела бы к заранее заданной целевой конфигурации.

Некоторые аспекты этой игры оказались весьма интересными для исследований в СИИ. Пространство состояний этой игры является достаточно большим, чтобы представлять интерес, однако вполне доступным для анализа (число возможных состояний составляет 9!, если различать симметричные состояния). Состояния игры легко представимы. Игра интересна для апробации интересных эвристик.

Для описания хода в игре удобнее говорить о перемещении пустой клетки. Допустимыми являются 2...4 различных хода пустой клеткой (вверх, вниз, вправо, влево) в зависимости от ее положения на доске. Если определены начальное и конечное состояние в головоломке, то процесс решения задачи можно описать явно. Состояния описываются массивом размерности 3·3. Дуги в пространстве состояний определяют 4 процедуры, описывающие возможные перемещения пустой клетки. В графе данной головоломки возможны циклы.

2.2. Поиск на основе цели и на основе данных

Поиск в пространстве состояний можно вести в направлениях от исходных данных задачи к цели и от цели к исходным данным.

При поиске на основе данных (управляемый данными), который иногда называют прямой цепочкой, исследователь начинает процесс решения задачи, анализируя ее условие, а затем применяет допустимые ходы или правила изменения состояния. В процессе поиска пра-

вила применяются к известным фактам для получения новых фактов, которые, в свою очередь, используются для генерации новых фактов. Этот процесс продолжается, пока мы не достигнем цели.

Возможен и обратный подход. Рассмотрим цель, которую мы хотим достичь. Проанализируем правила или допустимые ходы, ведущие к цели, и определим условия их применения. Эти условия становятся новыми целями, или подцелями, поиска. Поиск продолжается в обратном направлении от достигнутых подцелей до тех пор, пока мы не достигнем исходных данных задачи. Определяется путь от данных к цели, который строится в обратном направлении. Этот подход называется поиском от цели. Он похож на трюк: поиск выхода из лабиринта из конечного искомого состояния к заданному начальному.

Поиск на основе данных начинают с условий задачи и выполняют путем применения правил (допустимых ходов) для получения новых фактов, ведущих к цели. Поиск от цели начинают с обращения к цели и продолжают путем определения правил, которые могут привести к цели, и построения цепочки подцелей, ведущей к исходным данным.

В обоих подходах работают с одним графом пространства состояний, но порядок и число проверяемых состояний различаются. Выбор стратегии зависит от задачи. Нужно учитывать сложность правил и пространства состояний, природу и доступность данных.

Как пример зависимости сложности поиска от выбора стратегии рассмотрим задачу, в которой нужно подтвердить или опровергнуть утверждение «Я – потомок Петра I». Положительным решением является путь по генеалогическому дереву от «Я» до «Петр I». Поиск на этом графе можно вести в двух направлениях. Простая оценка позволяет сравнить сложность поиска в обоих направлениях. Петр I родился примерно 300 лет назад. Считая, что поколение составляет 25 лет, то длина искомого пути составляет 12. Каждый потомок имеет двух родителей, и путь от «Я» требует анализа 2^{12} предков. С другой стороны, поиск от вершины «Петр I» требует анализа большего числа состояний, поскольку родители обычно имеют более двух детей. Допустим, каждая семья имеет в среднем троих детей. В процессе поиска нужно проверить 3^{12} вершин генеалогического дерева. Второй путь сложнее. Но оба способа имеют экспоненциальную сложность.

Процесс поиска от цели рекомендован в следующих случаях:

1. Цель поиска или гипотеза явно присутствует в постановке зада-

чи или может быть легко сформулирована. Если задача состоит в доказательстве математической теоремы, то целью является сама теорема. Диагностические системы рассматривают возможные диагнозы, систематически подтверждая или отвергая некоторые из них.

2. Имеется большое число правил, которые на основе полученных фактов продуцируют возрастающее число заключений или целей. Отбор целей позволяет отсеять множество возможных ветвей, что повышает эффективность поиска. При доказательстве теорем число используемых правил вывода обычно значительно меньше количества, формируемого на основе полной системы аксиом.

3. Исходные данные не приводятся в задаче, но считается, что они должны быть известны решателю. В медицинской диагностике имеется большое число диагностических тестов. Доктор выбирает из них позволяющие подтвердить или опровергнуть конкретную гипотезу.

Поиск на основе данных рекомендован в следующих случаях:

1. Все или большинство исходных данных заданы в постановке задачи. Задача интерпретации состоит в выборе этих данных и их представлении в виде, подходящем для использования в интерпретирующих системах более высокого уровня.

2. Существует большое число потенциальных целей, но всего лишь несколько способов применения фактов.

3. Сформировать цель или гипотезы очень трудно. Изначально может быть мало информации о возможном решении.

Важен анализ конкретной задачи. Необходимо учитывать такие особенности, как фактор ветвления при использовании правил, доступность данных и простота определения потенциальных целей.

В логических играх, как правило, решение ищется от данных. В большинстве игр известно начальное состояние, есть набор возможных ходов и условия выигрыша, которые формируют большое число выигрышных игровых состояний.

2.3. Реализация поиска на графах

2.3.1. Поиск с возвратами

При решении задач путем поиска на основе данных либо от цели требуется найти путь от начального состояния к целевому на графе пространства состояний. Последовательность дуг этого пути соответствует упорядоченной последовательности этапов решения задачи.

Если решатель задач имеет в своем распоряжении оракула (непогрешимое средство предсказания для построения пути решения), то и поиска не требуется. Алгоритм решения задачи должен безошибочно двигаться прямо к цели, запоминая путь движения. Для сложных задач оракулов не существует, поэтому решатель должен рассматривать различные пути, пока не достигнет цели. Поиск с возвратами (*backtracking*) – метод систематической проверки различных путей.

Алгоритм поиска с возвратами – один из первых алгоритмов поиска в информатике. Алгоритм запускается из начального состояния и следует по некоторому пути, пока не достигнет цели или тупика. Если цель достигнута, поиск завершается. Решение задачи – путь к цели. Если поиск привел в тупиковую состояние, то алгоритм возвращается в ближайшую из пройденных вершин и исследует все поддеревья вершин-братьев. Процесс описывается рекурсивным правилом.

Если текущее состояние не удовлетворяет требованиям цели, то из списка его потомков выбираем первый и к этой вершине рекурсивно применяем процедуру поиска с возвратами. Если в результате поиска с возвратами в подграфе цель не обнаружена, то повторяем процедуру для вершины-брата. Процедура поиска продолжается, пока не найдена цель или не рассмотрены все вершины-братья. Если ни одна из вершин-братьев вершины не привела к цели, возвращаемся к предку вершины и повторяем процедуру с вершиной-братом.

На рис. 2.3.1 изображен процесс поиска с возвратами в абстрактном пространстве состояний. Пунктирные стрелки в дереве указывают направление поиска. Числа возле вершин указывают порядок их посещения. В алгоритме поиска используются три списка, позволяющих запоминать путь в пространстве состояний:

– *SL* (*state list*) – список исследованных состояний пути. Если цель найдена, то *SL* содержит состояния пути решения;

– *NSL* (*new state list*) – список новых состояний содержит вершины, подлежащие рассмотрению;

– *DE* (*dead ends*) – список тупиков, то есть вершин, которые не привели к цели. Список позволяет исключить повторный поиск.

При описании алгоритма поиска с возвратами не нужно учитывать повторные появления состояний во избежание их повторного анализа, а также петель – зацикливания алгоритма. Если состояние находят в списках *SL* или *DE*, то оно рассматривалось, и его игнорируют.

Алгоритм поиска с возвратами:

1. Инициализация: SL := [Start]; NSL := [Start]; DE := []; CS := Start.
2. Если CS = Goal, то результат поиска – SL. Переход к пункту 10.
3. Если CS не имеет потомков, кроме узлов входящих в списки DE, SL, NSL, то переход к пункту 4, иначе переход к пункту 7.
4. Добавить CS в DE. Удалить первый элемент в SL и в DE. CS := первый элемент NSL.
5. Повторять пункт 4 пока (SL <> []) и (CS = первый элемент списка SL).
6. Добавить CS в SL. Переход к пункту 9.
7. Если потомок CS не содержится в DE, SL, NSL, то поместить его в NSL.
8. CS := первый элемент NSL. Добавить CS в SL.
9. Повтор пунктов 2-8, пока NSL <> [].
10. Останов.

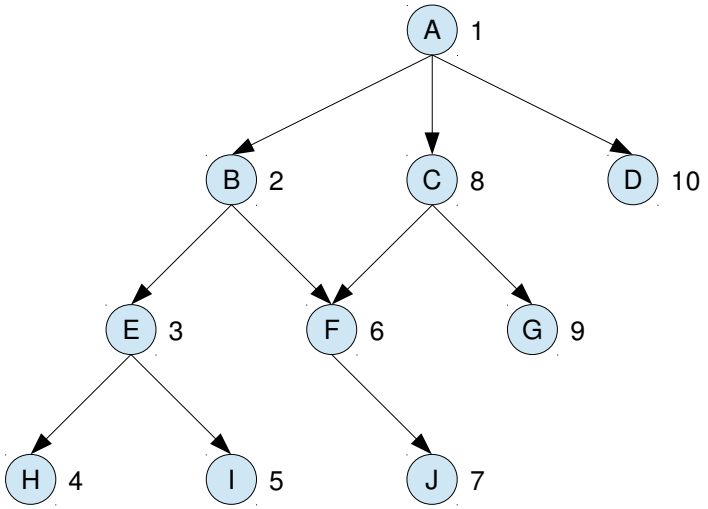

Рис. 2.3.1. Поиск с возвратами в абстрактном пространстве состояний

Обозначим текущее состояние при поиске с возвратами *CS* (*current state*). Состояние *CS* всегда равно последнему состоянию, занесенному в список *SL*, и представляет текущую вершину пути. Ходы игры упо-

рядочиваются и применяются к *CS*. Возникает упорядоченное множество новых состояний, потомков *CS*. Первый из этих потомков объявляется новым *CS*, а остальные заносятся в *NSL* для дальнейшего анализа. Новое *CS* заносится в *SL*, и поиск продолжается. Если *CS* не имеет потомков, то оно удаляется из списка *SL* (алгоритм возвращается назад), и исследуется один из оставшихся потомков его предка в *SL*.

Рассмотрим пример работы алгоритма поиска по графу на рис. 2.3.1, где рядом с вершинами подписан порядок их обхода:

Номер итерации	Текущее состояние CS	Список состояний SL	Список новых состояний NSL	Список тупиков DE
1	A	[A]	[A]	[]
2	B	[BA]	[BCDA]	[]
3	E	[EBA]	[EFBCDA]	[]
4	H	[HEBA]	[HIEFBCDA]	[]
5	I	[IEBA]	[IEFBCDA]	[H]
6	F	[FBA]	[FBCDA]	[EIH]
7	J	[JFBA]	[JFBCDA]	[EIH]
8	C	[CA]	[CDA]	[BFJEIH]
9	G	[GCA]	[GCDA]	[BFJEIH]
10	D	[DA]	[DA]	[CGBFJEIH]

Идеи поиска с возвратами:

1. Формируется список неисследованных состояний (*NSL*), чтобы иметь возможность возвратиться к ним.

2. Поддерживается список тупиков состояний (*DE*), чтобы оградить алгоритм от проверки бесполезных путей.

3. Поддерживается список узлов текущего пути (*SL*) – результат.

4. Каждое новое состояние проверяется на вхождение в эти списки, чтобы предотвратить зацикливание.

2.3.2. Поиск в глубину и в ширину

Определив направление поиска (от данных или от цели), алгоритм поиска должен определить порядок исследования состояний дерева или графа. Рассмотрим два варианта порядка обхода узлов графа: поиск в глубину (*depth-fist*) и поиск в ширину (*breadth-first*).

При поиске в глубину после исследования состояния сначала необходимо оценить всех его детей и рекурсивно их детей, а затем исследовать очередную из вершин-братьев. Поиск в глубину углубляется в область поиска. Если углубление невозможно, то рассматриваются

вершины-братья. Поиск в глубину исследует состояния графа на рис. 2.3.1 в порядке: *A*, *B*, *E*, *H*, *I*, *F*, *J*, *C*, *G*, *D*. Алгоритм поиска с возвратами, рассмотренный в разделе 2.3.1, осуществляет поиск в глубину.

Поиск в ширину исследует пространство состояний по уровням. Если состояний на данном уровне больше нет, алгоритм переходит к следующему уровню. При поиске в ширину на графе из рис. 2.3.1 состояния рассматриваются в порядке: *A*, *B*, *C*, *D*, *E*, *F*, *G*, *H*, *I*, *J*.

Поиск в ширину. Поиск осуществляется с использованием списков *Open* и *Closed*, позволяющих отслеживать продвижение в пространстве состояний. Список *Open* содержит сгенерированные состояния, потомки которых еще не были исследованы. Порядок удаления состояний из списка *Open* определяет порядок обхода графа. В список *Closed* заносятся уже исследованные состояния.

Алгоритм поиска в глубину:

```
   1. Инициализация: Open := [Start]; Closed := [].
   2. Удалить X – крайнее слева состояние в Open.
   3. Если X = Goal, то решение найдено, переход к
пункту 8, иначе переход к пункту 4.
   5. Сгенерировать детей узла X.
   6. Каждого ребенка X, если его нет в списках Open и
Closed, занести в Open справа.
   6. Поместить X в Closed.
   7. Повтор пунктов 2–6 пока Open <> [].
   8. Останов.
```

Дочерние состояния соответствуют возможным игровым ходам. На каждой итерации генерируются все дочерние вершины состояния *X* и записываются в *Open*. Список *Open* действует как очередь и обрабатывает данные в порядке поступления. Это структура данных *FIFO* (*first in – first out*, первый пришел – первый ушел). Состояния добавляются в список справа, а удаляются слева. Дочерние состояния, которые были уже найдены в списках, игнорируются. Если алгоритм завершается из-за невыполнения условия цикла (*Open* = []), то граф исследован, а цель не достигнута – поиск потерпел неудачу.

Поскольку при поиске в ширину узлы графа рассматриваются по уровням, сначала исследуются те состояния, пути к которым короче, чем гарантируется нахождение кратчайшего пути от начального состояния к цели. Вначале исследуются состояния, найденные по кратчайшему пути, и повторная проверка вершин исключается.

Рассмотрим работу поиска в ширину по графу на рис. 2.3.1:

Номер итерации	Список Open	Список Closed
1	[A]	[]
2	[BCD]	[A]
3	[CDEF]	[BA]
4	[DEFG]	[CBA]
5	[EFG]	[DCBA]
6	[FGHI]	[EDCBA]
7	[GHI]	[FEDCBA]
8	[HI]	[GFEDCBA]
9	[I]	[HGFEDCBA]
10	[J]	[IHGFEDCBA]
11	[]	[JIHGFEDCBA]

Поиск в глубину. Поиск – упрощенный алгоритм поиска с возвратами, представленного в разделе 2.3.1. Алгоритм подобен алгоритму поиска в ширину, но меняется порядок работы со списком *Open*. Дочерние состояния добавляются и удаляются с левого конца списка, то есть список реализован как стек магазинного типа или структура данных *LIFO* (*last in – first out*, последним пришел – первым ушел). При организации списка в виде стека предпочтение отдается новым узлам (недавно сгенерированным состояниям).

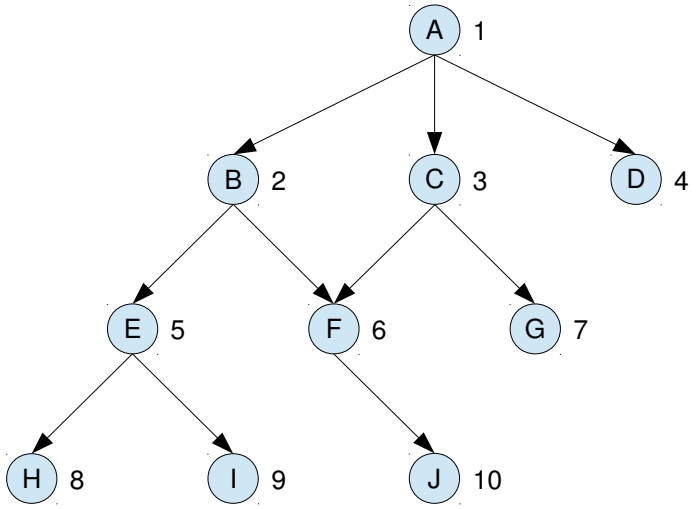

Рис. 2.3.2. Поиск в ширину в абстрактном пространстве состояний

Алгоритм поиска в ширину:

```
1. Инициализация: Open := [Start]; Closed := [].
2. Удалить X – крайнее слева состояние в Open.
3. Если X = Goal, то решение найдено, переход к
пункту 8, иначе переход к пункту 4.
5. Сгенерировать детей узла X.
6. Каждого ребенка X, если его нет в списках Open и
Closed, занести в Open слева.
6. Поместить X в Closed.
7. Повтор пунктов 2-6 пока Open <> [].
8. Останов.
```

Как и при поиске в ширину, в списке *Open* перечислены все обнаруженные, но еще не оцененные состояния (текущий фронт поиска), а в *Closed* записаны уже рассмотренные состояния.

Рассмотрим работу поиска в глубину по графу на рис. 2.3.1:

Номер итерации	Список Open	Список Closed
1	[A]	[]
2	[BCD]	[A]
3	[EFCD]	[BA]
4	[HIFCD]	[FBA]
5	[IFCD]	[HFBA]
6	[FCD]	[IHFBA]
7	[JCD]	[FIHFBA]
8	[CD]	[JFIHFBA]
9	[GD]	[CJFIHFBA]
10	[D]	[GCJFIHFBA]
11	[]	[DGCJFIHFBA]

В отличие от поиска в ширину, поиск в глубину не гарантирует нахождение оптимального пути к состоянию, если оно встретилось впервые. Позже в процессе поиска могут быть найдены различные пути к состоянию. Если длина пути имеет значение в решении задачи, то в случае нахождения алгоритмом состояния повторно необходимо сохранить самый короткий путь.

После выбора стратегии поиска (на основе данных или от цели), оценки графа и выбора метода поиска (в глубину или в ширину) дальнейший ход решения будет зависеть от конкретной задачи. Влиять на выбор стратегии поиска может необходимость обнаружения именно кротчайшего пути к цели, коэффициент ветвления пространства, доступное время вычислений и доступная память под данные,

средняя длина пути к целевой вершине графа, а также необходимость получения всех решений или только первого. У любого подхода всегда имеются определенные преимущества и недостатки.

Что лучше: поиск в глубину или поиск в ширину? Необходимо проанализировать пространство состояний или проконсультироваться с экспертами в данной области. В шахматах, например, поиск в ширину просто невозможен. В более простых играх поиск в ширину не только возможен, но даже может оказаться единственным способом избежать проигрышей или потерь в игре.

2.3.3. Поиск с итерационным заглублением

Одним из решений проблем поиска в глубину является использование ограничений значения глубины поиска заданным числом уровней. Это обеспечивает развертку области поиска в ширину при поиске в глубину. Ограниченный поиск в глубину наиболее полезен, если известно, что решение находится в пределах некоторой глубины, имеются ограничения во времени или пространство состояний чрезвычайно велико (как в шахматах). Такая модификация позволяет исправить немало недостатков как поиска в глубину, так и поиска в ширину.

Итерационным заглублением называется поиск в глубину, который стартует из узлов, которые получаются при обходе графа в ширину. Если уровень стартового узла поиска в глубину равен n, то из нее выполняется поиск в глубину на n следующих уровней. При переходе на следующий уровень старта заглубления происходит исследование вершин на 2 уровня глубже относительно предыдущего уровня.

При переходе на следующий уровень старта заглубления происходит повторная проверка некоторых вершин, так как информация о просмотренных состояниях после каждого заглубления удаляется. Этим достигается простота реализации метода. Кажется, что итерационное продвижение в глубину менее эффективно по времени, чем поиск в глубину или в ширину. Однако временная сложность алгоритма (время выполнения алгоритма в зависимости от размерности задачи) в действительности имеет тот же самый порядок. Поскольку число вершин на данном уровне дерева растет экспоненциально с увеличением глубины, почти все время вычислений тратится на самом глубоком уровне, просмотр которых не дублируется.

На рис. 2.3.3 показан процесс заглубления из стартовых вершин,

соответствующих поиску в ширину по графу на рис. 2.3.2.

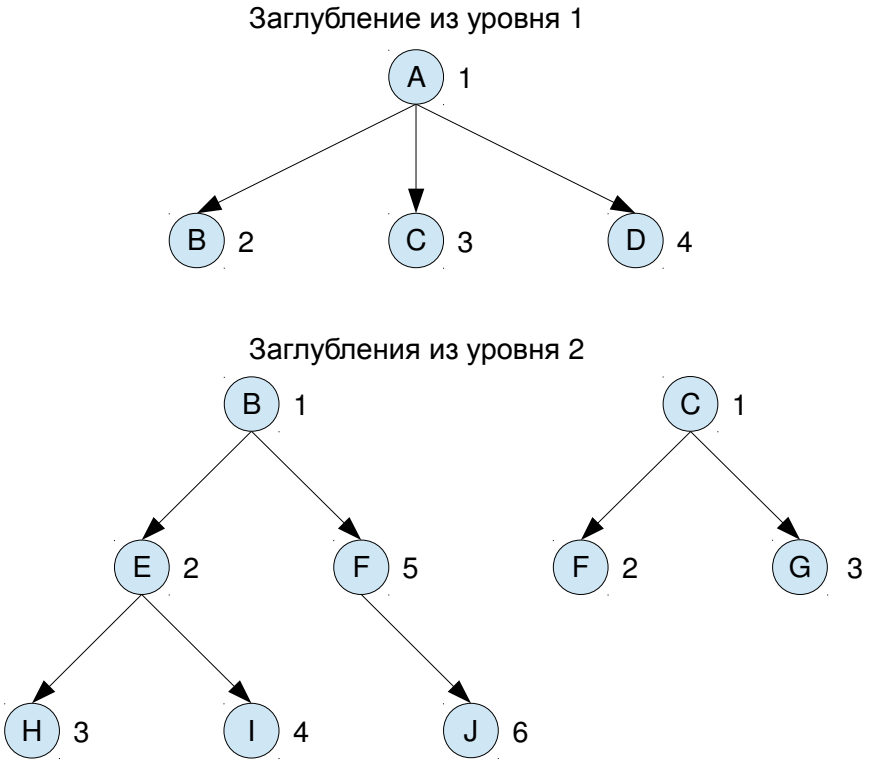

Рис. 2.3.3. Заглубления из стартовых уровней при поиске с заглублением

Поскольку поиск в глубину ограничен, то алгоритм может гарантировать нахождение кратчайшего пути к цели. Поскольку на каждой итерации осуществляется только поиск в глубину, степень использования пространства на каждом уровне n составляет $B \cdot n$, где B – среднее число дочерних состояний узла.

К сожалению, рассмотренные в разделе 2.3 стратегии поиска обладают экспоненциальной сложностью. В худшем случае решение может оказаться в самом правом состоянии самого нижнего уровня графа, а значит, во всех предложенных алгоритмах потребуется проверить все вершины графа.

2.4. Эвристический поиск

Эвристика определяется как изучение методов и правил открытий и изобретений. Греческий корень слова означает «исследовать». Когда Архимед выскочил из ванны, он закричал «Эврика!», что значит «Нашел!». В пространстве состояний поиска эвристика определяется как набор правил для выбора тех ветвей из пространства состояний, которые с наибольшей вероятностью приведут к приемлемому решению проблемы. Специалисты по СИИ используют эвристику в двух ситуациях.

1. Проблема может не иметь точного решения из-за неопределенности в постановке задачи и / или в исходных данных. Например, медицинская диагностика. Определенный набор признаков может иметь несколько причин; поэтому врачи используют эвристику, чтобы поставить наиболее точный диагноз и подобрать соответствующие методы лечения. Другой пример задачи с неопределенностью – система технического зрения. Визуальные сцены часто неоднозначны, вызывают различные предположения относительно размеров и ориентации объектов. Системы технического зрения часто используют эвристику для выбора наиболее вероятной интерпретации из нескольких возможных.

2. Проблема может иметь точное решение, но стоимость его поиска может быть слишком высокой. Во многих задачах (например, игра в шахматы) рост пространства возможных состояний носит экспоненциальный характер, другими словами, число возможных состояний растет экспоненциально с увеличением глубины поиска. В этих случаях прямые методы поиска решения типа поиска в глубину или в ширину могут занять слишком много времени. Эвристика позволяет избежать этой сложности и вести поиск по наиболее перспективному пути, исключая из рассмотрения неперспективные состояния и их потомки. Эвристические алгоритмы могут (на это рассчитывают разработчики) сократить комбинаторный рост и найти приемлемое решение.

К сожалению, подобно всем правилам для открытий и изобретений, эвристика может ошибаться. Эвристика – это только предложение следующего шага, который будет сделан на пути решения проблемы. Такие предположения часто основываются на опыте или интуиции. Поскольку эвристика использует такую ограниченную информацию, как описание текущих состояний в списке, то она редко способна предсказать дальнейшее поведение пространства состояний. Эвристика может привести алгоритм поиска к локальному экс-

тремума в решении или не найти решение. Это ограничение присуще эвристическому поиску и не может быть устранено улучшением эвристики или более эффективным алгоритмом поиска.

Эвристика и разработка алгоритмов для эвристического поиска в течение достаточно долгого времени остаются основным объектом исследования в проблемах СИИ. Игра и доказательство теорем – самые старые приложения СИИ – нуждаются в эвристике для сокращения числа состояний в пространстве поиска. Невозможно рассмотреть каждый вывод, который может быть сделан в той или иной области математики, или каждый возможный ход на шахматной доске. Эвристический поиск – часто единственный практический метод решения таких проблем. Также эвристики используются в экспертных системах.

Эвристический поиск состоит из эвристической меры и алгоритма, который использует ее для поиска в пространстве состояний.

В следующих разделах рассмотрим алгоритмы эвристического поиска, продемонстрируем примеры эвристических оценок и обсудим теоретические понятия, связанные с эвристическими оценками.

2.5. Эвристические алгоритмы

2.5.1. «Жадный» алгоритм

Наиболее простой путь эвристического поиска – это применение процедуры поиска экстремума (*hill climbing*). При поиске экстремума оценивают текущее состояние поиска и его потомков. Для продолжения поиска выбирается наилучший потомок; а о его братьях и родителях просто забывают. Поиск прекращается при достижении состояния, которое лучше, чем любой из его детей. Иногда такой подход сравнивают с действиями энергичным, но слепым альпинистом, поднимающимся вдоль наиболее крутого склона до тех пор, пока он не достигнет первого локального холма и не сможет идти дальше. Если при этом данные о предыдущих состояниях не сохраняются, то алгоритм не может быть продолжен из точки, которая привела к неудаче. Если найденное состояние является не решением задачи, а только локальным максимумом, то алгоритм неприемлем для данной задачи.

Пример такого локального экстремума появляется в игре в «пятнашки». Часто для того, чтобы передвинуть фишку в нужную позицию, необходимо сдвинуть фишку, находящуюся в наилучшей позиции. Без этого невозможно решить головоломку, но в то же время на

данном этапе это ухудшает состояние системы. Методы поиска без механизмов возврата или других приемов восстановления не могут отличить локальный максимум от глобального. Гарантированно решать задачи с использованием техники поиска экстремума нельзя.

Несмотря на этот недостаток, алгоритм поиска экстремума может быть эффективным, если оценивающая функция позволяет избежать локального максимума. В общем случае, эвристический поиск требует более гибкого метода. Например, в модификации «жадного» алгоритма поиска с восстановлением из точки локального максимума и продолжением поиска в еще непроверенных вершинах дерева.

Подобно алгоритмам поиска в глубину и алгоритмам поиска в ширину «жадный» алгоритм поиска использует списки сохраненных состояний: список *Open* отслеживает текущее состояние поиска, а в список *Closed* записываются уже проверенные состояния. На каждом шаге алгоритм записывает в список состояние с учетом некоторой эвристической оценки его «близости к цели». На каждой итерации рассматриваются наиболее перспективные состояния из списка open.

Алгоритм функции, реализующей «жадный» алгоритм поиска:

1. Инициализация: Open := [Start], Closed := [].
2. Удалить первое состояние X из списка Open.
3. Если X = Goal, то результат поиска – путь от Start до X. Переход к пункту 12.
4. Сгенерировать потомков X.
5. Если потомок не содержится в списке Open или Closed, то вычислить эвристическое значение и добавить потомка в список Open.
6. Если потомок уже содержится в списке Open и он был достигнут по кратчайшему пути, то записать это состояние в список Open.
7. Если потомок уже содержится в списке Closed и он был достигнут по кратчайшему пути, то перенести потомка из списка Closed в список Open.
8. Выполнить пункты 5–7 для всех потомков X.
9. Поместить X в список Closed.
10. Отсортировать список Open по ухудшению эвристики (приоритетная очередь).
11. Повтор пунктов 2–10 пока список Open не пуст.
12. Останов.

На каждой итерации функция удаляет первый элемент из списка.

Достигнув цели, алгоритм возвращает путь, который ведет к решению. Каждое состояние сохраняет информацию о предшествующем, чтобы затем восстановить его и найти кратчайший путь к решению.

Если первый элемент в списке *Open* не является решением, то алгоритм генерирует всех возможных потомков. Если потомок уже в списках, то алгоритм выбирает кратчайший путь до этого состояния.

Затем функция вычисляет эвристическую оценку состояний в списке и сортирует список в соответствии с эвристиками.

На рис. 2.5.1 показан пример пространства состояний с эвристическими оценками некоторых состояний (через тире). Состояния, по которым велся эвристический поиск, отмечены жирнее. Поиск не ведется по всему пространству, так как цель «жадного» алгоритма поиска – прийти к целевому состоянию кратчайшим путем. Чем более обоснованной является эвристика, тем меньше состояний нужно проверить.

Путь поиска целевого состояния показан сплошными стрелками. В отличие от поиска экстремума, который не сохраняет приоритетную очередь для отбора следующих состояний, данный алгоритм восстанавливается после ошибки и находит целевое состояние.

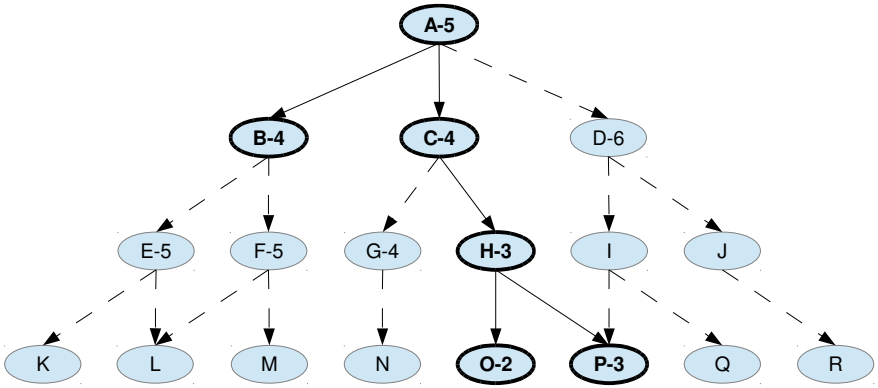

Рис. 2.5.1. «Жадный» алгоритм поиска в пространстве состояний

Рассмотрим работу алгоритма по дереву на рис. 2.5.1. Предположим, что решение содержится в узле *P*.

```
1. Open = [A-5], Closed = [].
2. Проверяем A-5: Open = [B-4, C-4, D-6], Closed =
[A-5].
```

3. Проверяем B-4: Open = [C-4, E-5, F-5, D-6], Closed = [B-4, A-5].
4. Проверяем C-4: Open = [H-3, G-4, E-5, F-5, D-6], Closed = [C-4, B-4, A-5].
5. Проверяем H-3: Open = [O-2, P-3, G-4, E-5, F-5, D-6], Closed = [H-3, C-4, B-4, A-5].
6. Проверяем O-2: Open = [P-3, G-4, E-5, F-5, D-6], Closed = [O-2, H-3, C-4, B-4, A-5].
7. Проверяем P-3: решение найдено.

На каждой итерации цикла работы алгоритма состояния списка *Open* формируют текущую искривленную граница поиска.

«Жадный» алгоритм поиска всегда выбирает наиболее перспективное состояние для продолжения. Поскольку при этом используется эвристика, которая может оказаться ошибочной, алгоритм не отказывается от остальных состояний и сохраняет их в списке *Open*.

Таким образом, в худшем случае при работе «жадного» алгоритма придется просмотреть все вершины. Однако алгоритм предлагает просматривать вершины в соответствии с приоритетом на основе эвристики, что в большинстве случаев сокращает число проверок состояний.

2.5.2. Минимаксный алгоритм на полную глубину

Игры для двух человек более сложны, чем головоломки, из-за наличия непредсказуемого противника. Такие игры не только обеспечивают некоторые интересные возможности для развития эвристик, но и создают большие трудности в разработке алгоритмов поиска.

Рассмотрим игры с небольшим пространством состояний, обеспечивающим возможность исчерпывающего поиска – систематического перебора всех возможных ходов и контрмер противника.

Игра «ним» (*nim*) – одна из игр, пространство состояний которой допускает исчерпывающий поиск. В этой игре на стол между двумя противниками выкладывается некоторое число фишек. При каждом ходе игрок должен разделить группу фишек на два непустых набора разных размеров. Например, 6 фишек можно разделить на группы 5 и 1 или 4 и 2, но не 3 и 3. Первый игрок, который не сможет больше сделать ход, проигрывает. Для разумного числа фишек в пространстве состояний этой игры можно реализовать полный перебор. На рис. 2.5.2 показано пространство состояний игры для 7 фишек.

Данная игра относится к играм с нулевой суммой выигрышей, то

есть сколько выиграл игрок, столько же проиграл его противник, и наоборот. Назовем противников *Min* и *Max*. *Max* представляет игрока, пытающегося выиграть, или максимизировать свое преимущество, *Min* – противника, который пытается минимизировать счет своего оппонента.

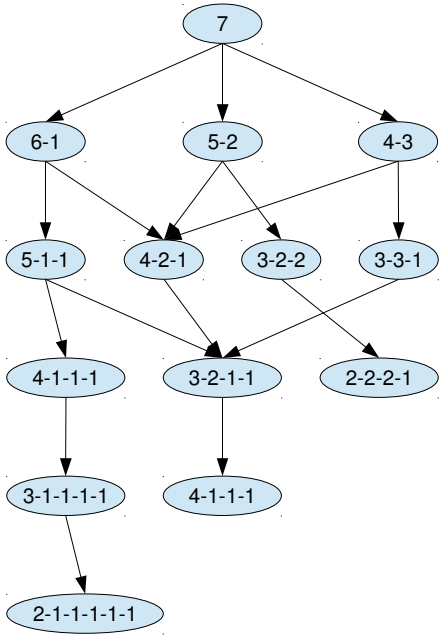

Рис. 2.5.2. Пространство состояний игры «ним» для 7 фишек

При реализации стратегии минимакса пометим каждый уровень в области поиска именем игрока, выполняющего очередной ход: *Min* или *Max*. В примере на рис. 2.5.3 первый ход делает *Min*. Каждому листовому узлу присвоим значение +1 или −1, в зависимости от победителя: *Max* или *Min*. В процессе реализации процедуры минимакса эти значения передаются вверх по графу согласно следующему правилу:

1. Если родительское состояние является узлом *Max*, то ему присваивают максимальное значение среди его детей.
2. Если родительское состояние является узлом *Min*, то ему присваивают минимальное значение среди его детей.

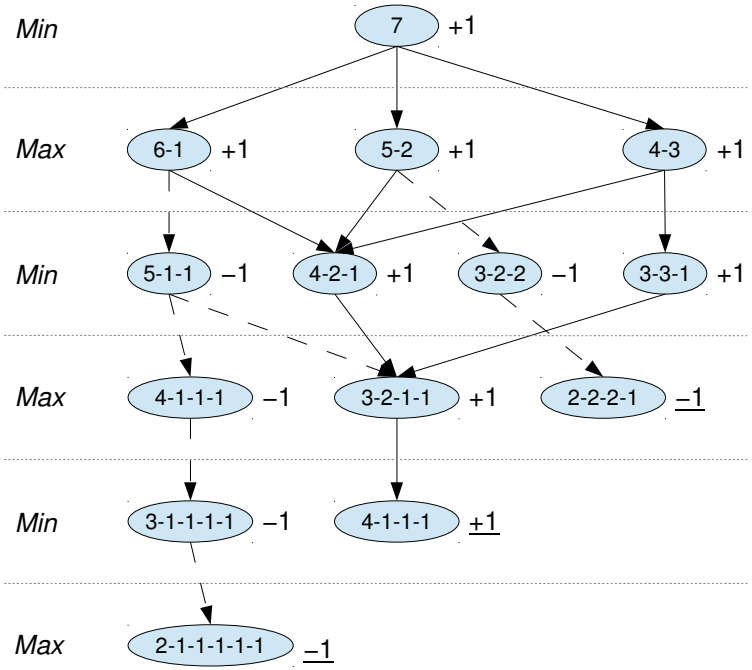

Рис. 2.5.3. Минимаксная стратегия в игре «ним» для 7 фишек

Данное правило обусловлено целью игроков: одного максимизировать, а другого минимизировать результат игры своим ходом. Значение, связываемое таким образом с каждым состоянием, отражает наилучшее состояние, которого этот игрок может достичь (принимая во внимание действия противника в соответствии с алгоритмом минимакса). Таким образом, полученные значения используются для того, чтобы выбрать наилучший среди возможных шагов. Результат применения алгоритма минимакса к графу пространства состояний для игры «ним» показан на рис. 2.5.3.

Значения вершинам присваиваются снизу вверх на основе принципа минимакса. Поскольку любой из возможных первых шагов *Min* ведет к вершинам со значением +1, второй игрок *Max* всегда может победить, независимо от первого хода *Min*. *Min* может победить только в том случае, если *Max* допустит ошибку. На рис. 2.5.3 видно, что может выбрать любой из возможных вариантов первого хода –

все они открывают возможные пути победы *Max*, отмеченные на рисунке сплошными линиями со стрелками.

Алгоритм минимаксной стратегии на полную глубину:

1. Построение дерева игровых состояний на полную глубину.

2. Определение результата игры в конечных состояниях (висячие вершины, листья графа). Результат: положительное или отрицательное значение (выигрыш одного из игроков) или нулевое значение (ничья).

3. Минимаксный перенос результатов игры от листьев графа вверх по графу к корню в соответствии с минимаксным правилом.

4. Принятие решения об очередном ходе в соответствии со значениями минимаксного переноса: *Max* ходит в сторону максимального из детей, *Min* – в сторону минимального.

2.5.3. Минимаксный алгоритм на ограниченную глубину

При использовании минимакса в сложных играх редко удается построить полный граф пространства состояний. Обычно поиск выполняют вглубь на определенное число уровней, которое определяется располагаемыми ресурсами времени и памяти компьютера. Эта стратегия называется прогнозированием *n*-го слоя, где *n* – число исследуемых уровней. Такой подграф не отражает всех состояний игры, и поэтому его узлам невозможно присвоить значения, отражающие победу или поражение. Поэтому с каждой вершиной связывают значение, определяемое некоторой эвристической функцией оценки. Значение, которое присваивается каждой из вершин на пути от корневой вершины, – не показатель возможности достичь победы (как в предыдущем примере). Это просто эвристическое значение оптимального состояния, которое может быть достигнуто за n шагов от текущей корневой вершины (вершины текущего хода). Прогнозирование увеличивает силу эвристики, позволяя применить ее на большей области пространства состояний. Стратегия минимакса интегрирует эти отдельные оценки в значение, которое присваивается корню.

Графы игр рассматриваются по уровням. Как показано на рис. 2.5.3, *Max* и *Min* делают шаги поочередно. Каждый ход игроков определяет новый уровень графа. Типичная игровая программа просматривает все варианты ходов игроков от текущей корневой вершины до определенной фиксированной глубины, которая определяется пространственны-

ми или временными ограничениями компьютера. Состояния до этой глубины оцениваются эвристикой и на основе минимакса распространяются обратно к начальному состоянию. Алгоритм поиска затем использует эти производные значения, чтобы выбрать наилучший среди возможных ходов из очередной корневой вершины.

Присвоив оценку каждому состоянию выбранного слоя, программа передает это значение каждому родительскому состоянию. Минимаксный перенос оценок выполняется по правилу, представленному в предыдущем разделе (2.5.2, минимакс на полную глубину).

Максимизируя значения для родительских узлов *Max* и одновременно минимизируя их для *Min*, алгоритм осуществляет проход по графу, присваивая значения потомков текущего состояния родительскому. Затем эти значения используются алгоритмом для выбора наилучшего потомка текущего состояния. На рис. 2.5.4 алгоритм минимакса продемонстрирован на абстрактном пространстве состояний.

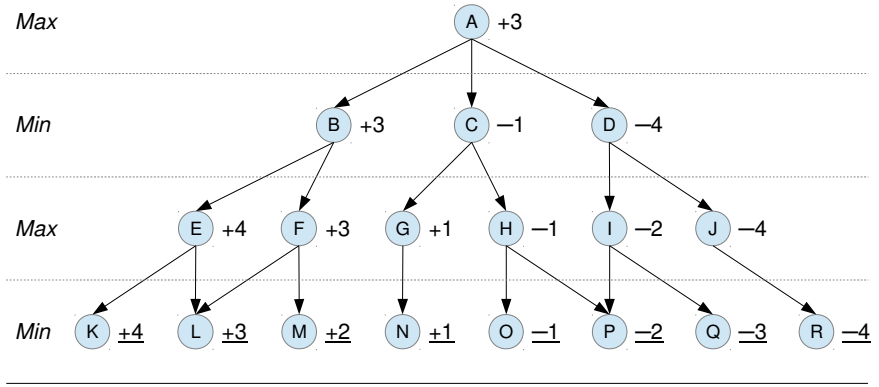

Рис. 2.5.4. Минимакс на ограниченную глубину

Оценки любого предварительно выбранного уровня могут быть неверными. При использовании эвристики с ограниченным прогнозированием сохраняется вероятность, что путь, приводящий в дальнейшем к плохой ситуации, не будет обнаружен вовремя. Если ваш противник по шахматам предлагает ладью в качестве приманки, чтобы забрать вашего ферзя, а оценка не идет дальше того уровня, где предлагается ладья, то у такого состояния будет наилучшая оценка. Подобный выбор состояния приведет к проигрышу в игре. Это называется эффектом

горизонта. Для противостояния такой ситуации обычно просматривают граф состояний на несколько уровней глубже состояния с хорошей оценкой. Но и это углубление поиска в важных областях не исключает полностью эффект горизонта, так как поиск все равно должен остановиться на определенном уровне.

Алгоритм минимаксной стратегии на ограниченную глубину:

1. Построение дерева игровых состояний из текущего состояния (корневой вершины) на заданное число уровней.

2. Определение эвристических оценок для состояний на последнем (горизонтном) уровне дерева состояний (листья).

3. Минимаксный перенос результатов игры от листьев графа вверх по графу к корню в соответствии с минимаксным правилом.

4. Принятие решения об очередном ходе в соответствии со значениями минимаксного переноса: *Max* ходит в сторону максимального из детей, *Min* – в сторону минимального.

5. Для принятия решения по следующему ходу игрока необходимо повторное построение дерева состояний из новой текущей (корневой) вершины.

2.5.4. Альфа-бета-отсечение

Прямой алгоритм минимакса требует двух проходов по области поиска. Первый позволяет пройти вглубь области поиска и там применить эвристику, а второй – присвоить эвристические значения состояниям, следуя вверх по дереву. Минимакс рассматривает все ветви в пространстве поиска, включая те из них, которые могли быть исключены более интеллектуальным алгоритмом. В конце 1950-х годов Ньюэл и Саймон создали алгоритм альфа-бета-отсечения (α-β) и позволил повысить эффективность поиска в играх с двумя участниками.

Идея α-β-поиска проста: вместо того чтобы рассматривать все состояния некоторого уровня графа, алгоритм выполняет поиск в глубину. В процессе поиска участвуют два значения α и β. Значение α, связанное с узлами *Max*, никогда не уменьшается, а значение β, связанное с узлами, никогда не увеличивается. Предположим, что для некоторого узла *Max* значение α равно *z*. Тогда *Max* может не рассматривать состояния, которые связаны с нижележащими узлами *Min* и имеют значения меньше или равные *z*. α – самое плохое значение, которое может принимать *Max*, если *Min* будет придерживаться наилучшей стратегии. Точно так же, если значение β для узла *Min* равно

z, то не следует рассматривать нижележащие вершины *Max*, с которыми связаны значения *z* или более.

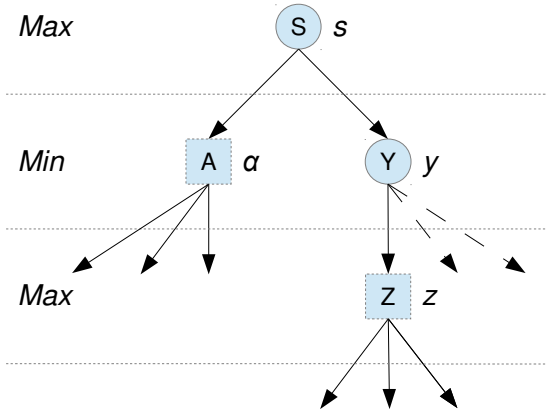

Рис. 2.5.5. α-отсечение на 2 уровня

Начиная α-β-поиск, опускаемся на полную глубину графа, используя поиск в глубину, и вычисляем эвристические оценки для всех состояний этого уровня. Предположим, что все они – узлы *Min*. Максимальное из этих значений *Min* возвращается родительскому состоянию (узлу *Max*, как и в алгоритме минимакса). Затем это значение передается прародителям узлов *Min* как потенциальное граничное значение β.

Затем алгоритм опускается к другим потомкам и не рассматривает их родительские состояния, если их значения больше или равны данному значению β. Аналогичные процедуры можно описать для α-отсечения по потомкам второго поколения вершины *Max*.

Условия останова алгоритма поиска на основе значений α и β:

1. Поиск может быть остановлен ниже любого узла *Min*, значение *β* которого меньше или равно значению *α* любого из его предков *Max*.

2. Поиск может быть остановлен ниже любого узла Max, значение *α* которого больше или равно значению *β* любого из его предков *Min*.

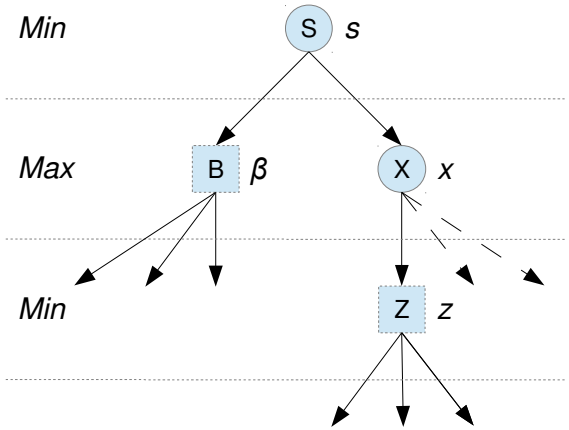

Рис. 2.5.6. β-отсечение на 2 уровня

На рис. 2.5.5 и 2.5.6 приведены примеры α- и β-отсечений при анализе на 2 уровня. Круглые вершины – состояния, которые еще не оценены, квадратные – оценка которых уже известна. Пунктиром показаны отсекаемые поддеревья. На рис. 2.5.7 приведен пример α-отсечения при анализе на 4 уровня.

Условия отсечения:

1. Если $z \leq \alpha$, то отсечь остальных детей вершины Y.
2. Если $x \geq \beta$, то отсечь остальных детей вершины X.

Таким образом, α-β-отсечение выражает отношение между узлами уровней n и n+2. При этом из рассмотрения могут быть исключены целые поддеревья с корнями на уровне n+1. Например, на рис. 2.7.1 изображено пространство поиска, к которому применен алгоритм α-β-отсечения. Заметим, что результирующее возвращаемое значение идентично полученному в алгоритме минимакса, но при этом процедура α-β-поиска значительно экономичнее простого минимакса.

При удачном порядке состояний в области поиска процедура α-β-отсечения может эффективно удваивать глубину поиска в рассматриваемом пространстве при фиксированных пространственно-временных характеристиках компьютера. Если же узлы расположены неудачно, процедура α-β-отсечения осуществляет поиск в таком же пространстве, как и обычный минимакс. Однако поиск проводится только за один проход.

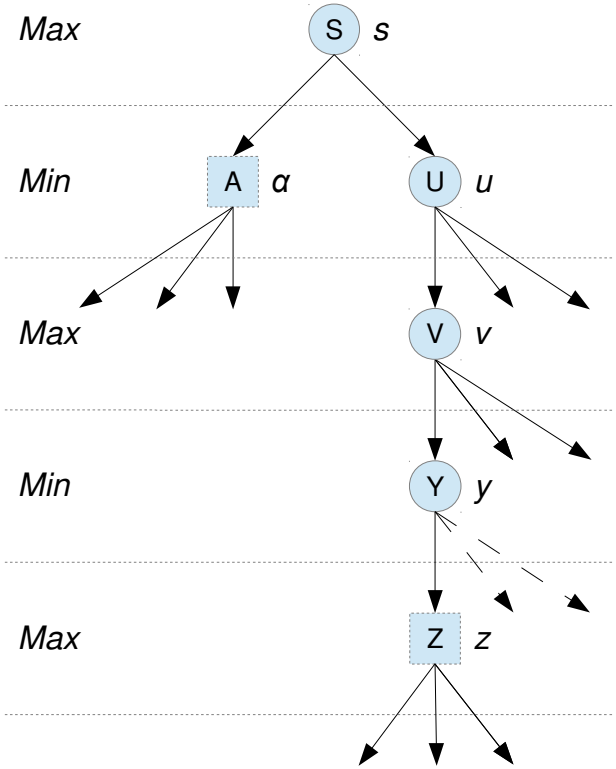

Рис. 2.5.7. α-отсечение на 4 уровня

2.6. Эвристическая мера оценки состояний

2.6.1. Функции эвристической оценки состояний

Рассмотрим эффективность нескольких различных эвристик для решения 8-головоломки. На рис. 2.6.1 показано начальное и целевое состояние вместе с первыми потомками начального состояния.

Самая простая эвристика подсчитывает количество фишек, находящихся не на своих местах, сравнивая с целевым состоянием. Состояние с меньшим числом фишек, находящихся не на своих местах, будет рассмотрено раньше. Результат для потомков исходной вершины: 5, 3, 5.

Однако эта эвристика не использует всю имеющуюся информацию, потому что она не принимает во внимание расстояние, на которое фишки должны быть перемещены. Улучшенная эвристика суммировала бы все расстояния между текущем и целевым положениями

фишек. Результат для потомков исходной вершины: 6, 4, 6.

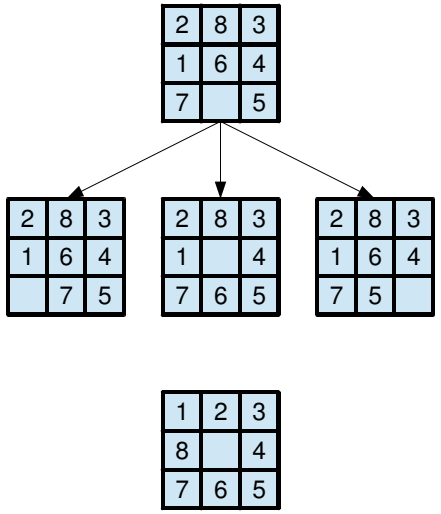

Рис. 2.6.1. Примеры состояний в игре «8-головоломка»

Обе эвристики не учитывают трудности при перестановке фишек. Если необходимо переставить две рядом стоящие фишки, то потребуется более двух ходов, так как фишки должны обойти друг друга.

Эвристика, принимающая во внимание этот факт, для каждой инверсии (необходимость поменять две рядом стоящие фишки) должна умножать эвристическое значение на коэффициент (например, 2). В нашем примере инверсий нет.

Четвертый вариант эвристики предполагает суммирование всех полученных эвристических оценок. Результат для примера: 11, 7, 11.

Это показывает сложность формирования хорошей эвристики. Цель в том, чтобы, используя ограниченную информацию описания состояния, сделать верный выбор. Разработка хорошей эвристики – эмпирическая проблема. В ее решении помогают рассуждения и интуиция, но главный критерий – эффективность при решении практических задач.

Поскольку эвристика может быть ошибочной, есть вероятность, что алгоритм поиска может пойти по ложному пути. Эта проблема возникала при поиске в глубину, где подсчет глубины использовался для определения ложных путей. Эта идея может быть также приме-

нена в эвристическом поиске. Если два состояния имеют одинаковые эвристические значения, то предпочтительнее исследовать состояние, которое расположено ближе к корневому состоянию графа. Расстояние от начального состояния – уровень глубины состояния.

Для начального состояния это значение равно 0 и увеличивается на 1 для каждого уровня поиска, что отражает фактическое число шагов для достижения потомка. Уровень состояния может быть добавлен к эвристической оценке, чтобы находить ближайшие решения.

Оценивающая функция состоит из двух компонентов:

$$f(n) = g(n) + h(n),$$

где $g(n)$ – фактическое расстояние от текущего состояния n к начальному; $h(n)$ – эвристическая оценка расстояния от состояния n к цели.

Компонент функции оценки $g(n)$ придает поиску свойства поиска в ширину. Он предотвращает возможность заблуждения из-за плохой оценки: если эвристика непрерывно возвращает хорошие оценки для состояний на пути, не ведущем к цели, то значение $g(n)$ будет расти и доминировать над $h(n)$, возвращая процедуру поиска к кратчайшему пути. Это избавляет алгоритм от зацикливания.

2.6.2. Допустимость, монотонность и информированность

Поведение эвристики можно оценивать по ряду параметров. Наряду с решением задачи может понадобиться нахождение кратчайшего пути к нему. Такая эвристика называется допустимой. Это важно, если дополнительный шаг к цели имеет высокую стоимость.

Существуют ли лучшие эвристики? В каком смысле одна эвристика лучше другой? Это отражает информированность эвристики.

Можно ли гарантировать, что обнаруженное в процессе эвристического поиска состояние нельзя было найти по короткому пути от начального состояния? Это свойство называется монотонностью.

Допустимость. Алгоритм поиска называется допустимым (приемлемым, *admissible*), если гарантирует нахождение кратчайшего пути к решению. Поиск в ширину является допустимой стратегией поиска. Поскольку сначала рассматриваются все состояния графа на уровне n и лишь затем состояния на уровне n+1, то целевые состояния будут найдены по кратчайшему пути. Однако поиск в ширину часто неэффективен при практическом использовании.

Используя функцию оценки $f(n)$, определим класс допустимых эв-

ристических стратегий поиска. Если n – узел графа пространства состояний, то $g(n)$ представляет глубину, на которой это состояние было найдено, а $h(n)$ – эвристическую оценку расстояния от узла n до цели. В этом смысле $f(n)$ оценивает общую стоимость пути от начального состояния к целевому через узел n. При определении допустимой эвристики определим функцию $f^*(n)$ – фактическую стоимость оптимального пути от начального узла до целевого через узел n:

$$f^*(n) = g^*(n) + h^*(n);$$

где $g^*(n)$ – стоимость кратчайшего пути от начального узла к узлу n; $h^*(n)$ – фактическая стоимость кратчайшего пути от узла n до цели.

Поскольку такие оракулы, как $f^*(n)$, для большинства реальных задач не существуют, то используем функцию оценки f как наиболее близкую к f^*. В алгоритме A $g(n)$ – стоимость текущего пути к состоянию n – является разумной заменой оценки g^*, но они не равны. Они равны только, если на графе найден оптимальный путь к n.

Заменим $h^*(n)$ на $h(n)$ – эвристическую оценку минимальной стоимости пути в целевое состояние из узла n. Как правило, вычислить h^* невозможно, но часто можно определить, действительно ли эвристическая оценка $h(n)$ ограничена сверху, то есть не превосходит ли $h(n)$ фактической стоимости кратчайшего пути $h^*(n)$.

Если алгоритм использует функцию оценки $f^*(n)$, в которой $h(n) \leq h^*(n)$, то он называется алгоритмом A^*. Поисковый алгоритм является допустимым, если для любого графа он всегда выбирает оптимальный путь к решению. Все алгоритмы A^* допустимы.

Поиск в ширину может быть охарактеризован как алгоритм A^*, в котором $h(n) = 0$. Решение о рассмотрении каждого состояния принимается исключительно с учетом его расстояния от начального состояния.

Примерами алгоритмов A^* являются некоторые эвристики для игры «8-головоломка». Для этой задачи невозможно вычислить значение $h^*(n)$, но можно показать, что эвристика ограничена сверху фактической стоимостью кратчайшего пути к цели. Если не всегда можно вычислить фактическую стоимость кратчайшего пути к цели, то часто удается доказать, что эвристика ограничена сверху этим значением.

Монотонность. Определение алгоритмов A^* не требует, чтобы $g(n)$ было равно $g^*(n)$. Это означает, что допустимая эвристика может первоначально достигать промежуточных состояний, следуя по субоптимальному пути, но в конечном счете алгоритм найдет оптимальный

путь ко всем состояниям, лежащим на пути к цели. Существует ли локально допустимая эвристика? То есть позволяющая последовательно находить кратчайший путь к каждому состоянию в процессе поиска? Это свойство называется монотонностью (*monotonicity*):

$$h(n_i) - h(n_j) \leq cost(n_i, n_j);$$

где n_j – потомок n_i; *cost* – фактическая стоимость пути; $h(Goal) = 0$.

Один из способов описания монотонности: считать область поиска всюду локально совместимой с эвристикой. Разность между эвристическими значениями состояния и любого из его потомков связана с фактической ценой пути между ними. То есть эвристика всюду допустима и достигает каждого состояния по кратчайшему пути от его предка.

Если «жадный» алгоритм поиска на графе использует монотонную эвристику, то можно упростить алгоритм. Поскольку такая эвристика находит кратчайший путь к любому состоянию при первом проходе через него, то исчезает необходимость вычисления пути к этому же состоянию при последующих проходах.

При использовании монотонной эвристики в процессе поиска эвристическое значение для каждого состояния n заменяется фактической стоимостью пути к n. Поскольку реальная стоимость в каждой точке пространства не меньше эвристической оценки, f не уменьшается.

Любая монотонная эвристика h является эвристикой A^*, и она допустима. Справедливо и обратное: все допустимые эвристики монотонны.

Информированность. Сравним способности двух эвристик найти кратчайший путь. Если для двух A^*-эвристик $h_1(n)$ и $h_2(n)$ для всех состояний n в пространстве поиска выполняется неравенство $h_1(n) <> h_2(n)$, то эвристика h_2 называется более информированной, чем h_1.

Это определение можно использовать при сравнении эвристик для игры в «8-головоломка».

Если эвристика h_2 более информирована, чем h_1, то множество состояний, проверяемых эвристикой h_2, – подмножество состояний h_1.

Чем более информирован алгоритм A^*, тем меньше состояний требуется проверить, чтобы получить оптимальное решение.

Однако желание использовать более информированную эвристику для уменьшения числа состояний может потребовать неоправданно больших затрат при вычислении самой эвристики.

2.7. Идеи обучения игровых программ

Одной из самых успешных ранних программ для игровых задач – программа игры в шашки, созданная Самюэлем в 1959 г. Эта программа была исключением для того времени, особенно учитывая существующие ограничения вычислительных ресурсов компьютеров тех лет. Мало того, что эта программа игры в шашки применяла эвристический поиск, в ней также присутствовал алгоритм обучения программы. Эта программа была во многом пионерской, и идеи, предложенные в ней, все еще используют при создании игровых и обучающихся программ.

Накопление. Идея заключается в хранении в памяти компьютера большого числа конфигураций на шашечной доске из тех, что реально (а не гипотетически) возникают в ходе шашечных игр. Вместе с каждой конфигурацией в памяти хранится также ее числовая оценка, которая получилась путем построения дерева, применения оценивающей функции к терминальным вершинам и передачи значений вверх по дереву посредством минимаксной процедуры. Имея в памяти некоторое множество конфигураций вместе с их оценками, программа в процессе работы ищет соответствие между конфигурацией, отвечающей каждой из вершин дерева, и конфигурациями из числа запомненных. Если такое соответствие установлено, то хранимая в памяти оценка передается в эту вершину. В результате отпадает необходимость строить какую-либо ветвь, которая могла бы возникнуть под этой вершиной.

Таким образом, накопление позволяет либо экономить время, либо достичь лучшего качества игры за то же время путем исследования дерева на несколько большую глубину.

Естественно, размер списка конфигураций, который может храниться в памяти и использовался, ограничен сверху. Программа была построена так, что наименее употребляемые конфигурации вычеркиваются, а часто встречающиеся остаются в памяти компьютера.

Обобщение. Самообучение. Этот подход позволяет программе в ходе игры улучшать свои оценивающие функции. Обычно оценивающая функция представляет собой полином, в простейшем случае – это полином первой степени:

$$h(n) = \sum_{i=1}^{m} k_i \cdot a_i(n); \qquad \sum_{i=1}^{m} k_i = 1,$$

где $a_i(n)$ – различные вычисляемые показатели игрового состояния n; k_i – весовые коэффициенты показателей; m – число показателей.

Показатели $a_i(n)$ описывают особенности игрового состояния: тактические преимущества; общий перевес шашек; перевес шашек в определенных частях доски; расположение шашек в определенных клетках; ситуация в центре доски; возможность жертвовать фигурами за тактическое преимущество; тенденцию перемещения фигур одного из игроков у края доски; перевес дамок и так далее.

Коэффициенты специально настраиваются таким образом, чтобы отражать вклад каждого фактора в общую оценку ситуации. Если преимущество в какой-либо части доски важнее, чем, скажем, в центре, то это отражает соответствующий коэффициент.

Качество игры зависит от подходящего выбора коэффициентов k_i, и обобщение является средством их подгонки, обеспечивающей улучшение игры. Метод обобщения представляет собой пример оптимизации с использованием процедуры, называемой «подъем в гору». Имеется начальный набор значений весов k_i, и в каждый момент времени эти коэффициенты определяют рабочую точку. Рабочая точка перемещается в пределах многомерного пространства по мере подгонки значений весов в поисках положения, в котором оптимизируется определенная реакция или целевая функция.

Чтобы воспользоваться методом подъема в гору, следует дать программе возможность сыграть некоторое число игр с определенным партнером, выбрав какое-то начальное множество коэффициентов k_i, а затем, сделав случайные изменения коэффициентов, повторить игру с тем же партнером. Если программа после изменений весов стала играть лучше, то в дальнейшем будут использоваться именно эти значения весов. В противном случае происходит возврат к исходным значениям весов и производится попытка очередного случайного изменения.

Такой подход дает улучшение игры, однако для этого необходима ровная игра противника обучающейся таким образом программы.

Подобный подход в настоящее время часто используется для оптимизации в случаях многоэкстремальной поверхности целевой функции – стохастические методы («тепловая машина»).

Был предложен другой путь нахождения весовых коэффициентов во время игры, основанный на том, что качество игры растет с увеличением глубины просмотра дерева состояний. Если может быть най-

дено средство вычисления оценочной функции, обеспечивающее точное совпадение переданного вверх по дереву (с большой глубиной) значения оценочной функции с результатом его прямого (с небольшой глубиной) определения, то такая оценка должна быть равнозначна изучению дерева построенного на большую глубину.

Если результат прямой оценки h_1 с помощью оценочной функции, а h_2 – результат передачи оценки по дереву (с большой глубиной), то можно считать их разность ошибкой:

$$e = h_1 - h_2.$$

Самюэль сделал так, что в его программе вычислялась корреляция между ошибкой e и k_i. Положительная корреляция указывает, что k_i необходимо уменьшить, а отрицательная – увеличить k_i.

При применении указанного метода требуется уделить внимание обеспечению его устойчивости. Для повышения устойчивости предложено фиксировать один из весовых коэффициентов, тогда как другие коэффициенты изменялись. Обычно это был наиболее важный показатель, оценивающий материальное соотношение, поскольку разумно полагать, что игроку всегда выгодно, чтобы его фигуры на доске сохранялись.

Был создан алгоритм программы, обладающий свойством самообучения (обучение без учителя). Это первая самообучающаяся программа.

Самоорганизация. Самюэль пошел дальше. Он держал в своей программе больший ассортимент критериев a_i, чем тот, что использовался итоговой оценивающей функцией. Используемое множество критериев видоизменялось во время игры: если какое-то из значений весовых множителей k_i оставалось близким к нулю в течение длительного времени, то соответствующий этому весу показатель оценивающей функции изымался из рабочего множества, а его место занимал другой показатель из числа ожидавших своей очереди и мог быть затем добавлен в оценивающую функцию.

Возможность изменения набора критериев a_i придает данному методу обучения новый характер. Теперь его можно воспринимать как некую самоорганизующуюся не только свои весовые коэффициенты, но их структуру.

Замыкание на себя. Следующая идея – замкнуть игровую программу на себя. Он организовал работу программы так, что она могла вести игру и самообучаться непрерывно днем и ночью имитируя игроков x и y.

Игроку *x* (кандидату) разрешалось модифицировать оценивающую функцию согласно описанным методам, тогда как другой игрок *y* (тренер) пользовался фиксированной оценивающей функцией.

Когда кандидат начинал выигрывать у тренера, то он занимал место тренера – параметры кандидата копировались тренеру. Если же кандидат после нескольких модификаций ухудшал свой результат, то его параметры восстанавливались из структур тренера.

Результат. После реализации этих идей через некоторое время обучения программа игры в шашки легко стала обыгрывать у своего автора все партии подряд.

Рассмотрим некоторые особенности. Например, из-за ограниченности глобальной стратегии она чувствительна к функциям оценки, приводящим к ловушкам. Модуль самообучения программы уязвим в случае противоречивой игры оппонента. Например, если противник широко использовал различные стратегии или просто играл «по-дурацки», то веса в полиномиальной оценке начинали принимать неожиданные значения, что приводило к полному снижению эффективности.

Отметим, что на разных этапах игры (дебют, эндшпиль и так далее) для игровой программы может быть более эффективен различный набор показателей и весовых коэффициентов.

2.8. Реализация рекурсивного поиска

Алгоритм минимакс. Рассмотрим реализацию алгоритма минимакс с помощью рекурсивной функции (*Color* – номер игрока):

```
SearchBestMove(Depth, Color)
1. Score := -100000, оценку близка к бесконечности.
2. Если Depth = 0, то в качестве результата вернуть оценку Evalate(Color) текущего состояния, перейти к пункту 10.
3. Сгенерировать все ходы из текущего состояния.
4. Сделать ход MakeMove(Move).
5. Rating := -SearchBestMove(Depth-1, not(Color)), вычислить оценку хода.
6. Сделать ход обратно UnMakeMove(Move).
7. Если Rating > Score, то Score := Rating.
8. Повторить пункты 4-8 для каждого возможного хода.
9. В качестве результата вернуть оценку Score.
10. Останов.
```

Алгоритм минимакс с альфа-бета-отсечением. Рассмотрим реализацию алгоритма с помощью рекурсивной функции:

```
SearchBestMoveAB(Depth, Color, MaxWhite, MaxBlack)
1. Score := -100000, оценку близка к бесконечности.
2. Если Depth = 0, то в качестве результата вернуть оценку Evalate(Color)текущего состояния, перейти к пункту 15.
3. Сгенерировать все ходы из текущего состояния.
4. Сделать ход MakeMove(Move).
5. Rating := -SearchBestMoveAB(Depth-1, not(Color), MaxWhite, MaxBlack), вычислить оценку хода.
6. Сделать ход обратно UnMakeMove(Move).
7. Если Rating <= Score, то переход к пункту 13.
8. Если Color = White, то переход к пункту 9, иначе переход к пункту 11.
9. Если Score > MaxWhite, то MaxWhite := Score.
10. Если -MaxWhite <= MaxBlack, то в качестве результата вернуть MaxWhite, перейти к пункту 15.
11. Если Score > MaxBlack, то MaxBlack := Score.
12. Если -MaxBlack <= MaxWhite, то в качестве результата вернуть MaxBlack, перейти к пункту 15.
13. Повторить пункты 4-12 для каждого возможного хода.
14. В качестве результата вернуть оценку Score.
15. Останов.
```

Первый вызов функции:

```
SearchBestMoveAB(4, White, -100000, -100000).
```

Рассмотрим более компактный вариант рекурсивной реализации алгоритма минимакс с альфа-бета-отсечением:

```
SearchBestMoveAB2(Depth, Color, Alpha, Beta)
1. Если Depth = 0, то в качестве результата вернуть оценку Evalate(Color) текущего состояния, перейти к пункту 9.
2. Сгенерировать все ходы из текущего состояния.
3. Сделать ход MakeMove(Move).
4. Rating := -SearchBestMoveAB2(Depth-1, not(Color), -Beta, -Alpha), вычислить оценку хода.
5. Сделать ход обратно UnMakeMove(Move).
6. Если Rating > Alpha, то Alpha := Rating.
7. Повторить пункты 3-6 для каждого возможного хода.
8. В качестве результата вернуть оценку Alpha.
9. Останов.
```

3. Работа «Программирование логических игр»

3.1. Цель работы

Целью работы является получение практических навыков решения сложных задач на примере логических игр с помощью подходов СИИ и закрепление теоретических знаний методам поиска по дереву состояний.

3.2. Задание на выполнение работы

Решить одну из игровых задач или головоломку методом поиска по дереву игровых состояний. Разработать алгоритм решения и реализовать его в виде программы на языке высокого уровня.

При просмотре дерева состояний использовать один из методов:

– просмотр дерева состояний на полную глубину;

– просмотр дерева состояний на ограниченную глубину.

При просмотре дерева состояний использовать один из алгоритмов:

– «жадный» алгоритм;

– минимаксный алгоритм;

– минимаксный алгоритм с альфа-бета-отсечением.

При просмотре дерева состояний на ограниченную глубину предложить ряд эвристических параметров для оценки состояний. Для каждой эвристики ввести свой весовой коэффициент.

Предусмотреть, если это возможно, обучение алгоритма при замыкании на себя.

3.3. Варианты заданий

Вариант игровой задачи предлагается выбрать студентам самостоятельно или находится из сборника описаний логических игр.

Вариант задания согласовывается с преподавателем. При недостаточной или чрезмерной сложности выбранной игровой задачи задание корректируется: усложняется или упрощается. При достаточной сложности игры возможно выполнение двумя студентами совместно.

Для выполнения работы необходимо выбрать головоломку (1 игрок) или игру (2 игрока) с полной информацией об игровом состоянии. Игры с неполной информацией (карточные, например игра «дурак») значительно сложнее в реализации.

3.4. Содержание и оформление отчета

Отчет должен содержать следующие разделы:

1. Титульный лист с указанием темы работы.
2. Текст задания.
3. Краткое описание игры.
4. Формализация дерева состояний.
5. Описание предложенных эвристик.
6. Краткое описание использованных алгоритмов.
7. Описание адаптации алгоритмов к вашей задаче.
8. Структура программы или алгоритмов.
9. Исходные тексты программ.
10. Пример работы программ (экранные формы).
11. Пример дерева состояний – результат работы вашей программы.
12. Выводы по достигнутым результатам, оценка эффективности использованных методов, алгоритмов, эвристик.
13. Список литературы.

Объем отчета – минимум 20 страниц (без учета исходных текстов программы).

Форма титульного листа дана в приложении.

Список литературы

1. *Люггер, Д. Ф.* Искусственный интеллект. Стратегии и методы решения сложных проблем. – М. : Вильямс, 2003. – 864 с.
2. *Корнилов, Е.* Программирование шахмат и других логических игр. – СПб. : БХВ-Петербург, 2005. – 272 с.
3. *Ясницкий, Л. Н.* Введение в искусственный интеллект. – М. : ACADEMIA, 2005. – 176 с.
4. *Менделеев, В.* Настольные игры для душевной компании. – Харьков : Клуб семейного досуга, 2007. – 416 с.
5. *Коробейников, А. В.* Обзор состояния программ спортивного бриджа / А. В. Коробейников, С. И. Зыкин, Р. Х. Судуров, И. С. Ефремова // Информационные системы в промышленности и образовании : сборник трудов молодых ученых. – Ижевск : Изд-во ИжГТУ, 2012. – С. 69–77.

Приложение

Форма титульного листа отчета

Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Ижевский государственный технический университет
имени М. Т. Калашникова»

Кафедра «Программное обеспечение»

Отчет по курсовой работе
«Программирование логических игр»

Вариант 1
«Шахматы»

Выполнил
студент группы Б06-161-1 Иванов И. И.

Принял Коробейников А. В.

Ижевск
2013

Учебное издание

ПРОГРАММИРОВАНИЕ ЛОГИЧЕСКИХ ИГР

«Системы искусственного интеллекта»,
«Математические основы искусственного интеллекта»,
«Методы и алгоритмы принятия решений» и
«Теория принятия решений. Методы и алгоритмы принятия решений»

Составители:
Коробейников Александр Васильевич
Лугачев Павел Петрович

В редакции составителей