МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Юго-Западный государственный университет»
(ЮЗГУ)

УТВЕРЖДАЮ
Проректор по учебной работе
_____ О.Г. Локтионова
«___» _____ 2018 г.

Технологии мягких вычислений

Методические указания по лабораторным работам
по дисциплине «Технологии мягких вычислений»

Курск 2017

УДК 004.032.26

Составители: Кабус Д.А. Кассим, С.А. Филист

Рецензент
Доктор технических наук, профессор А.Ф. Рыбочкин

Технологии мягких вычислений: методические указания по лабораторным работам / Юго-Зап. гос. ун-т; сост.: Кабус Д.А. Кассим, С.А. Филист Курск, 2017. 59 с.

Рассмотрены основные технологии мягких вычислений.

Предназначено для студентов направлению подготовки 12.04.04 «Биотехнические системы и технологии».

Текст печатается в авторской редакции

Подписано в печать 19.01.18 . Формат 60×84 1/16. Бумага офсетная.
Усл. печ. л. 3,43 . Уч.-изд. л. 3,1 . Тираж 100 экз. Заказ 4 .
Юго-Западный государственный университет.
305040, г. Курск, ул. 50 лет Октября, 94.
Отпечатано в ЮЗГУ.

Лабораторная работа №1

Изучение методов нечеткого логического вывода с помощью инструментальных средств

Цель: Закрепить знания по разделу нечеткие системы управления, систематизировать эти знания, научиться реализовывать модели на ЭВМ.

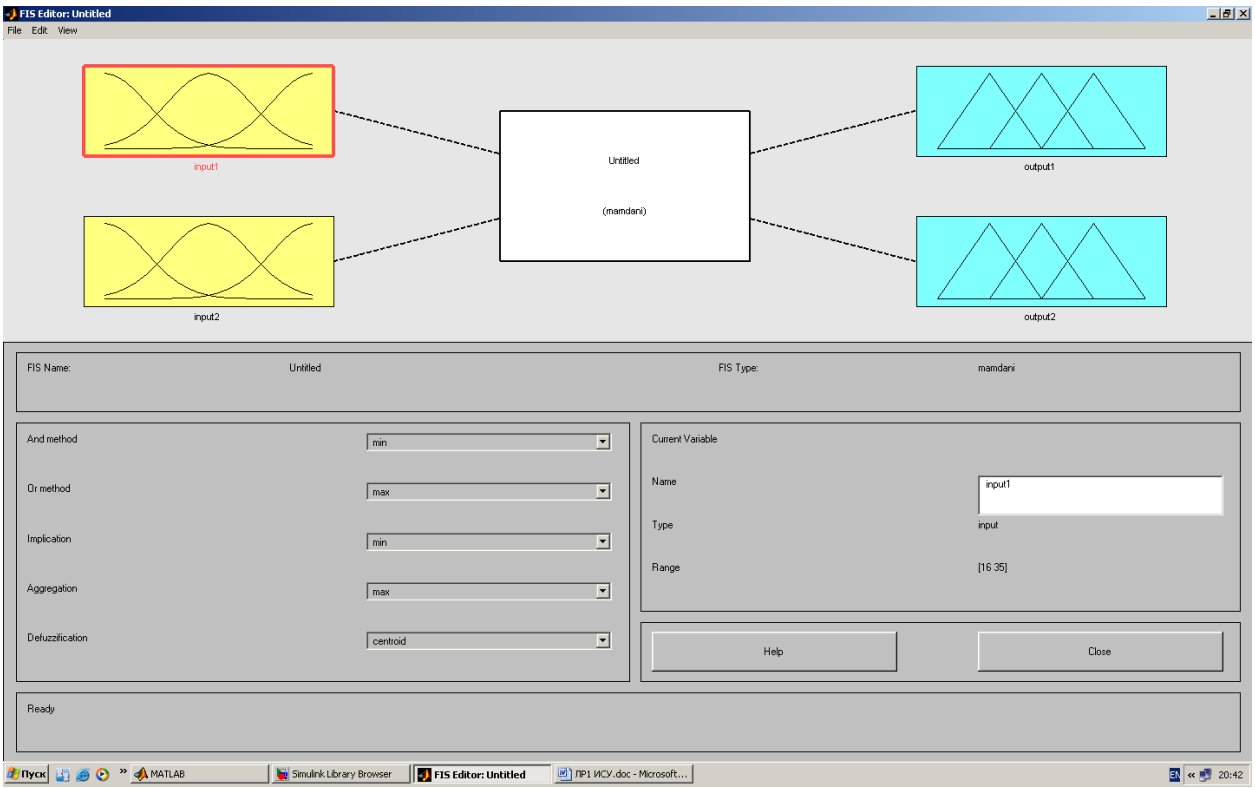
Ход работы

1. Постановка задачи

Определиться с количеством входных, выходных параметров, получить структурную схему системы управления. Реализовать модель управления.

2. Построение модели

Создадим двумерную модель кондиционера:



Входные параметры: температура внутри помещения и скорость изменения температуры в помещение .

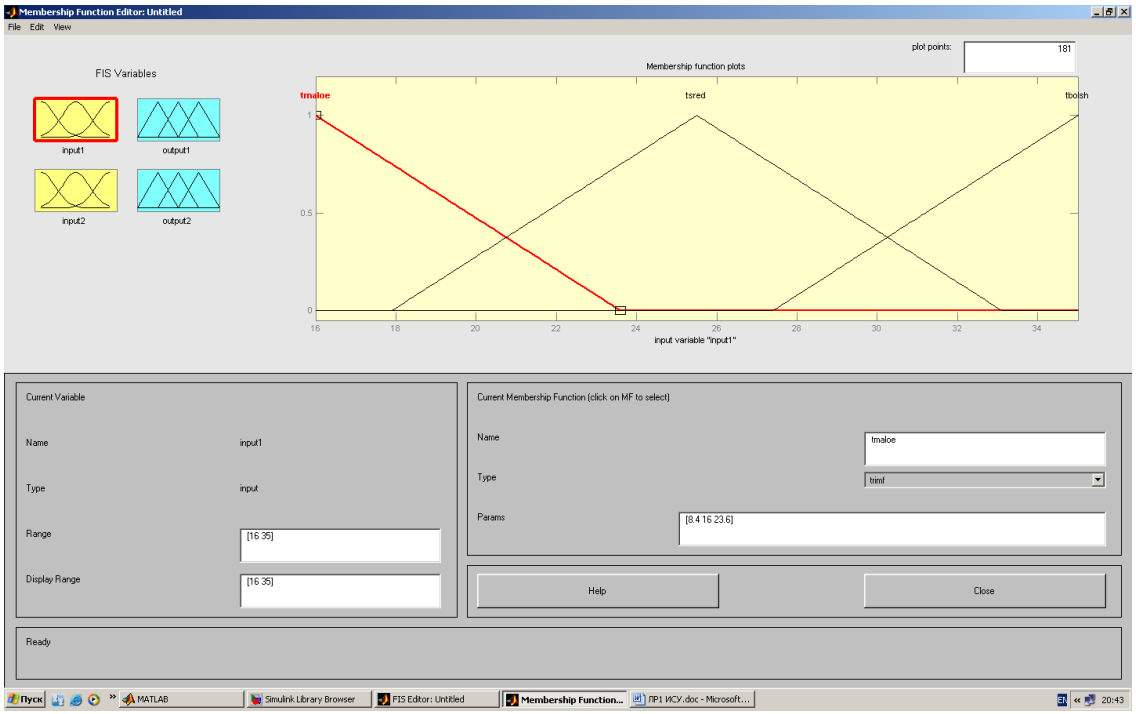
Выходные параметры: скорость вращения вентилятора и

расход охлаждающей жидкости.

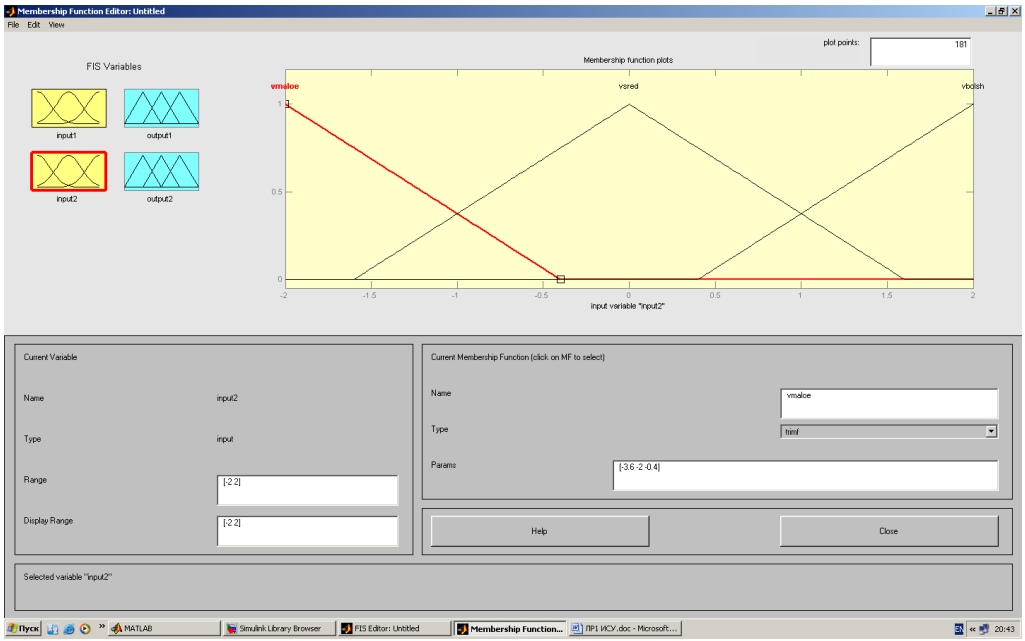
3. Создание термов

Для каждого параметра создадим термы:

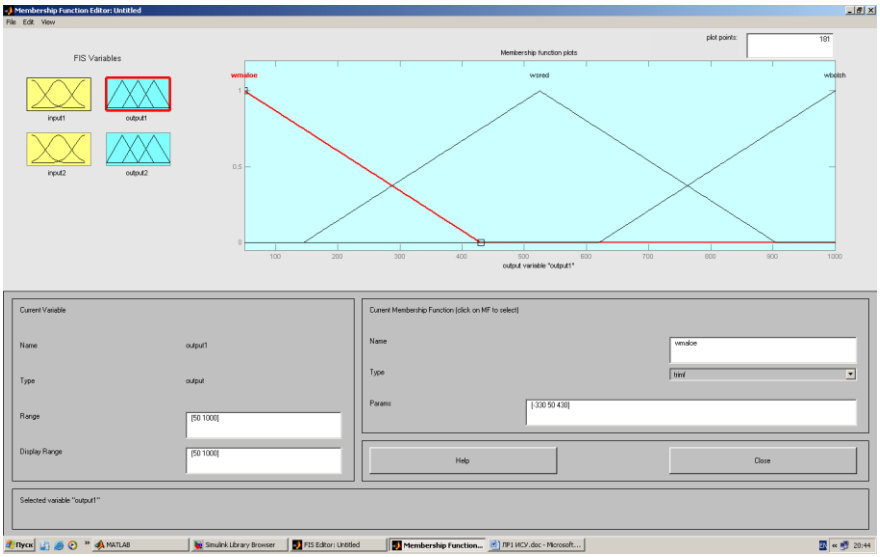
На данном графике указан входной сигнал «температура внутри помещения»



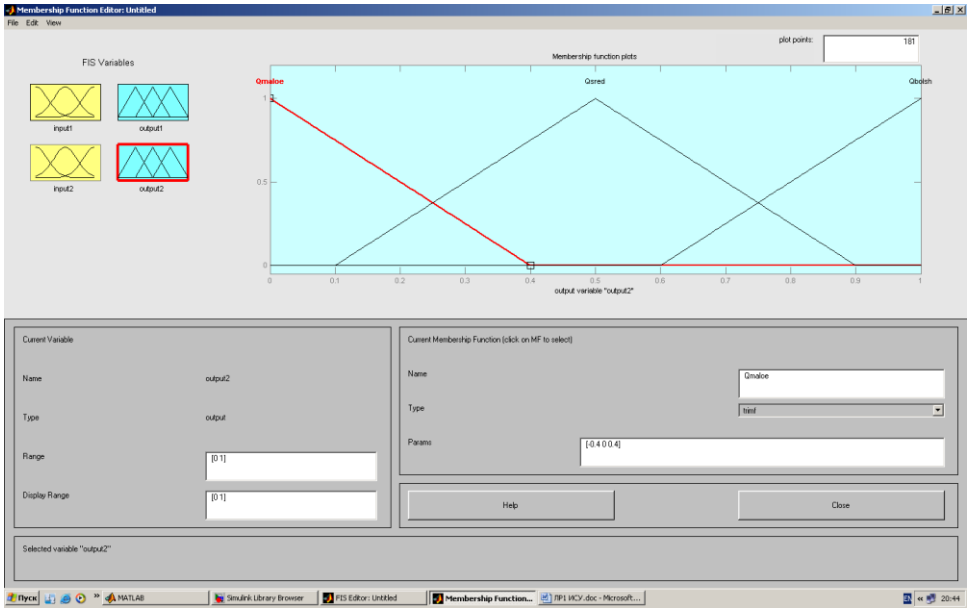
На данном графике указан входной сигнал «скорость изменения температуры в помещение»



На данном графике указан выходной сигнал «скорость вращения вентилятора»

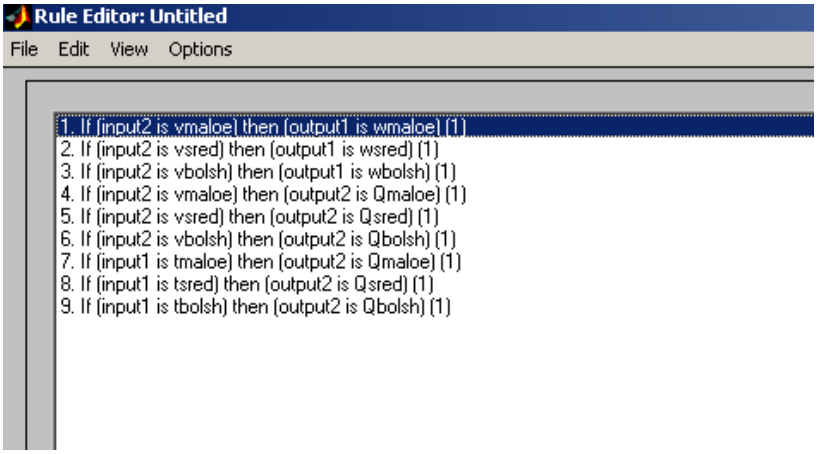


На данном графике указан выходной сигнал «расход охлаждающей жидкости»

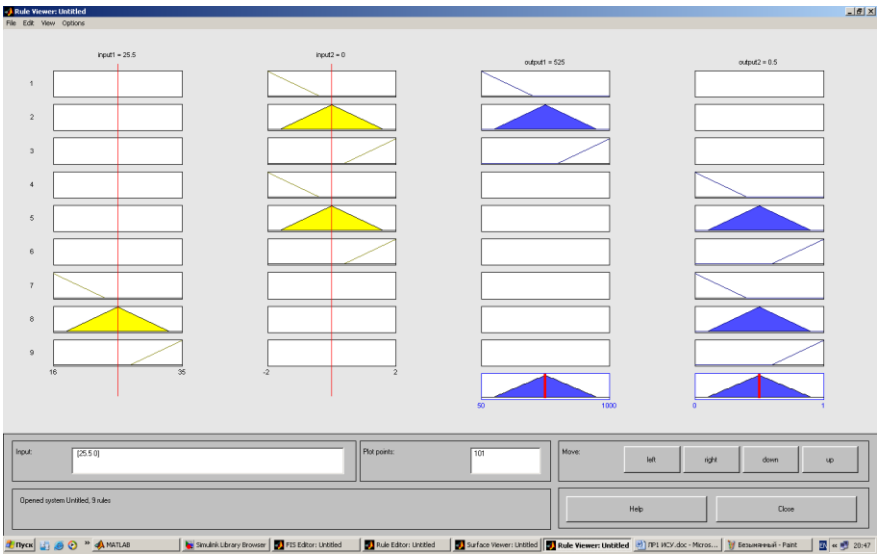


4. Создание правил

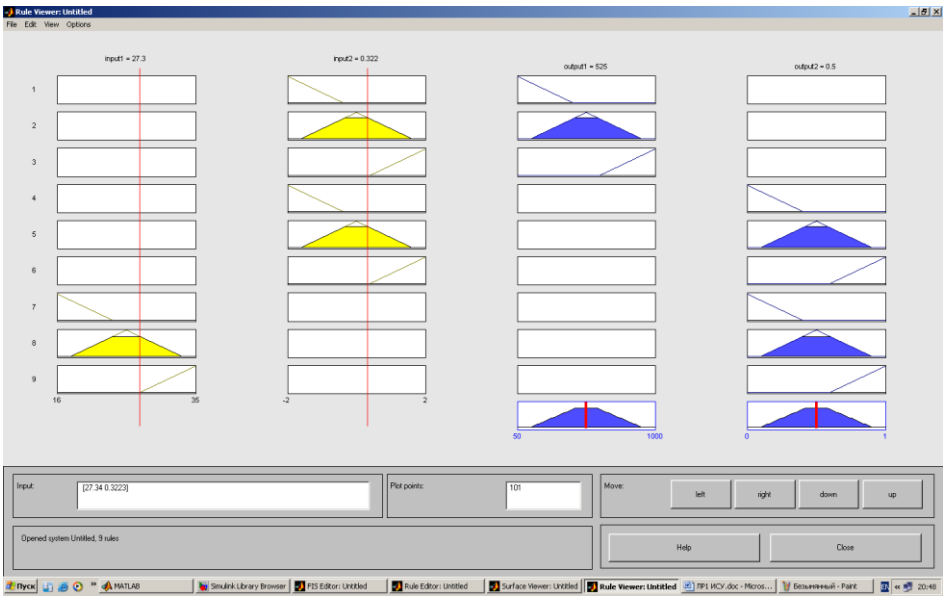
Для различных ситуаций создадим правила задания режима работы



Эти правила можно представить графически:

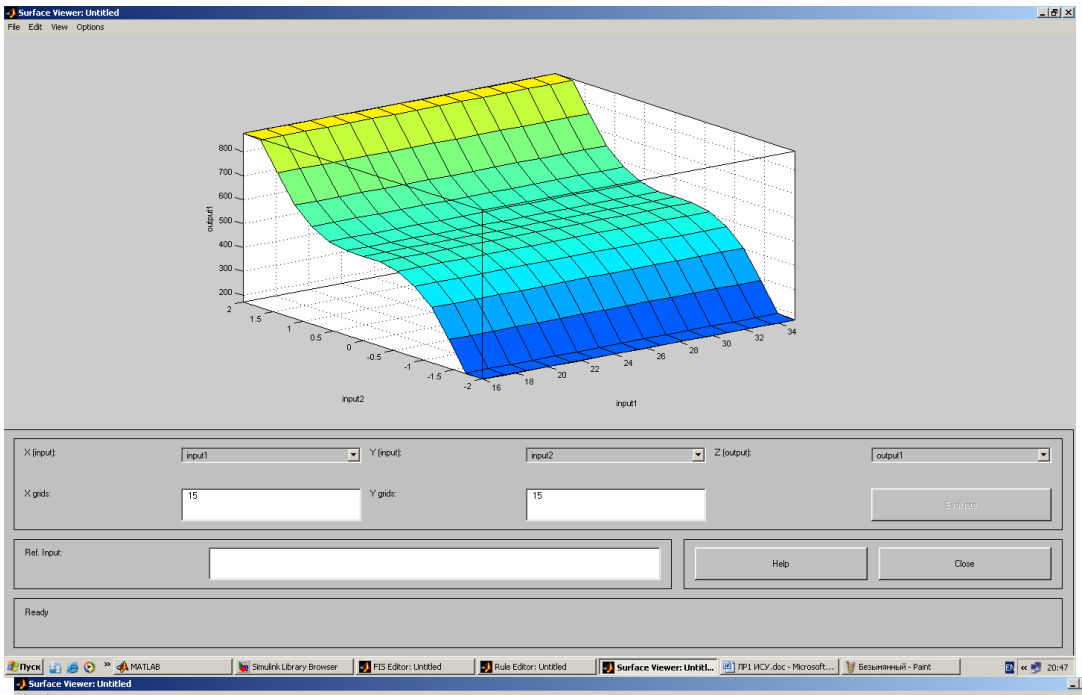


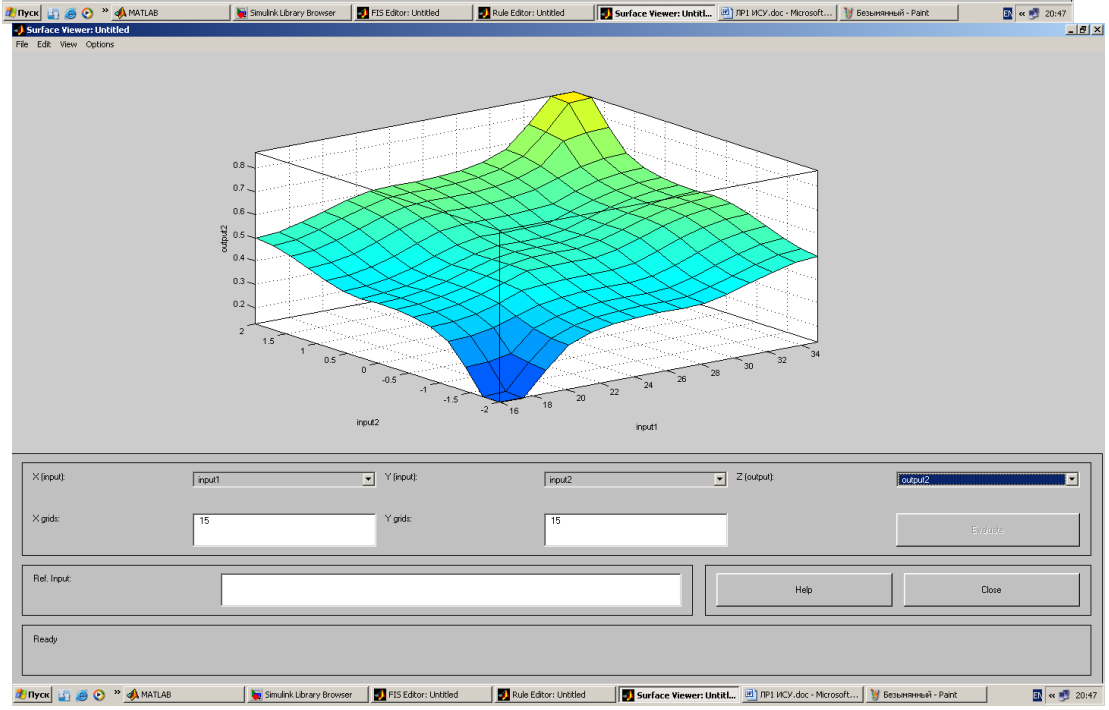
На этом графике мы изменили входные сигналы



5. Анализ полученной модели

Построим графики зависимости скорости вращения вентилятора, температуры в помещение и скорости изменения температуры.





Лабораторная работа №2

Реализация алгоритма Мамдами-Заде в среде МАТЛАБ

Цели и задачи работы:

Построение нечеткой экспертной системы с алгоритмом вывода Mamdani.

Задание

1. Создать нечеткую экспертную систему с алгоритмом вывода Mamdani, которая должна оценить уровень работы предприятия общественного питания.

Использовать 2 входа, 1 выход, 3 правила типа «если... то», «если... или… то».

2. Построить усложненный вариант нечеткой экспертной системы для оценки работы предприятия или другого объекта (число входов и правил долж-но соответствовать заданному варианту). Использовать правила типа «если...

то», «если... или… то», «если... и… то». Самостоятельно предложить входные переменные и правила вывода.

Варианты заданий:

Оценка квартиры для бюро обмена жилья: 5 входов, 7 правил.

Уровень обслуживания в пассажирском поезде: 4 входа, 6 правил.

Оценка успеваемости студентов: 3 входа, 5 правил.

Методические указания

При решении п.1 могут быть заданы следующие инструкции.

1. Если сервис плохой или еда несвежая, то оценка низкая.

2. Если обслуживание хорошее, то оценка средняя.
3. Если сервис отличный или еда вкусная, то оценка высокая.

При создании новой системы нечеткого вывода по умолчанию создается система с алгоритмом вывода Mamdani. Качество обслуживания и еды будем оценивать по 5-балльной системе.

В пункте меню EditVariable/Input добавить второй вход (появится второй блок с именем input2). Однократным щелчком левой кнопкой мыши по блоку input1 заменить его имя на «service», input2 – на «food», output1 – на «grade».

Задать функции принадлежности. Для переменной «service» в полях Range и DisplayRange установить диапазон изменения и отображения этой пе-ременной – от 0 до 5 (подтверждая ввод нажатием клавиши Enter). Удалить за-данные по умолчанию функции принадлежности с помощью мыши. Через пункт меню Edit/AddMFs задать функции принадлежности гауссова типа (gaussmf) в количестве 3. Заменить их имена на «bad», «good» и «excellent».

Щелчком мыши по «food» задать диапазон изменения от 0 до 5. После удаления заданных по умолчанию функций принадлежности задать две функ-ции принадлежности трапецеидальной формы trapfm с параметрами [0 0 1 3] и [2 4 5 5] и именами «notfresh» и «nice».

Для выходной переменной «grade» указать диапазон [0 5], задать три функции принадлежности треугольной формы с именами «bad», «good», «excel-lent».

Конструирование правил реализовано в пункте меню Edit/Rules. В первом и третьем правилах использовать «or» (единица в скобках после каждого правила указывает его «вес», который нужно оставить равным 1). При вводе второго правила, где отсутствует переменная «food», выбрать опцию none.

Из пункта меню View/Rules вызывать окно графического отображения функционирования системы, где можно задавать значения входных перемен-ных в соответствующих полях, ответ отображается в правой части окна. Можно также перемещать отметку шкалы с помощью мыши.

Из пункта меню View/Surface вызвать отображение двумерной функции оценки работы предприятия общественного

питания.

Сохранить созданную систему на диске: **File/Export/Todisk.**

При выполнении задания п.2 нужно усложнить созданную экспертную систему: ввести заданное число правил и входов. Экспертная система должна быть достаточно продуманной и иметь практическое значение.

Используемые операторы и команды: fuzzy – редактор нечеткой системы вывода. Программное обеспечение: MATLAB

Лабораторная работа №3

Применение генетических алгоритмов для решения задач оптимизации

1.1. Теоретические сведения

Стандартная математическая задача оптимизации формулируется таким образом.

Среди элементов *x*, образующих множество *X*, найти такой элемент *x**, который доставляет минимальное значение *f(x*)* заданной функции *f(x)*. Чтобы корректно поставить задачу оптимизации, необходимо задать: допустимое множество *X*; Целевую функцию — отображение $f : X \rightarrow R$; критерий поиска (max или min).

Если минимизируемая функция не является выпуклой, то часто ограничиваются поиском локальных минимумов и максимумов – точек *x0* таких, что всюду в некоторой их окрестности $f(x) \geq f(x_0)$ для минимума и $f(x) \leq f(x_0)$ – для максимума.

Если допустимое множество $X = R^n$, то такая задача называется задачей безусловной оптимизации, в противном случае – задачей условной оптимизации.

Существует множество методов оптимизации, которые можно разделить на три группы:

детерминированные; случайные (стохастические); комбинированные.

В частности, большой интерес представляют собой эволюционные методы, являющиеся стохастическими. Упоминания о применении генетических алгоритмов для решения задачи оптимизации относятся к концу 1960-х гг.

Эволюционные методы основываются на примере работы эволюции и обучения, к таким методам относят нейронные сети, генетические алгоритмы.

Генетический алгоритм – это эвристический алгоритм поиска, используется для решения задач оптимизации и моделирования путем случайного подбора, комбинирования и вариации искомых параметров с применением механизмов, напоминающих

биологическую эволюцию, является разновидностью эволюционных вычислений. Отличительная особенность генетического алгоритма – акцент на использовании оператора «скрещивания», производящего операцию рекомбинации решений-кандидатов, роль которой аналогична роли скрещивания в живой природе. Схематически алгоритм представлен на рисунке 1.

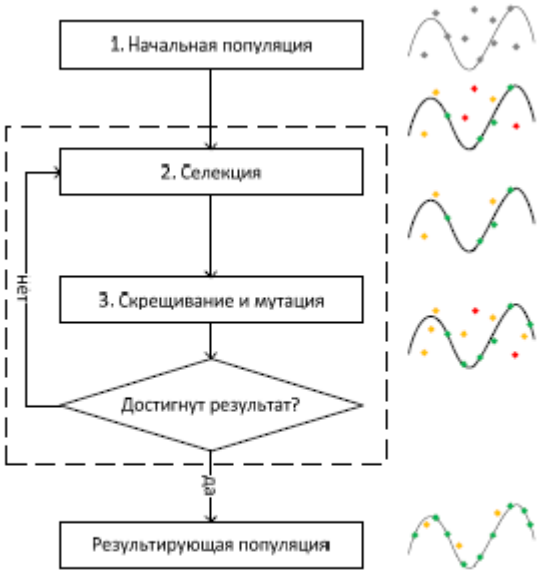


Рисунок 1 – Схема работы генетического алгоритма

Для применения алгоритма задачи приводятся к виду, при котором решение может быть представлено как набор более мелких составных частей (аналог генотипа и его составных частей –генов). Длина генотипа может быть как фиксированной, так и переменной.

1.2. Алгоритм метода

0. Подготовительный шаг – формирование начальной популяции (начального набора решений). Алгоритм для формирования может быть различным, но чаще всего используют случайный процесс с целью охватить большее разнообразие для поиска решений. Возможно применение других способов формирования, – например, с заранее известными свойствами, но следует иметь в виду, что это может повлиять на ход развития системы в дальнейшем.

1. Отбор – важный этап в алгоритме, отвечает за выбор направления развития популяций, чаще всего отбрасываются решения с низким значением функции приспособленности (fitness function), что способствует улучшение средней приспособленности всей популяции.

2. Скрещивание – этап, на котором происходит образование новых решений в популяции, прошедшей через отбор, для восстановления численности. Особенность его в том, что при использовании скрещивания берутся два или более существующих решений в популяции, а из них – составные части (гены) и соединяются в новом решении, которое остается в популяции.

3. Скрещивание не позволяет в полной мере охватить все возможные варианты сочетаний и значений генов, поэтому не менее важен процесс мутации. Он состоит в том, что в некоторых решениях из популяции происходят случайные изменения в генах.

Этот процесс способствует увеличению разнообразия в популяции.

4. Оценка решений и остановка алгоритма – в большинстве случаев; если для решения задачи необходимо применять генетический алгоритм, то нет критерия останова, основанного на самих решениях, вместо него применяется подход с числом вычислений (числом создаваемых популяций). Иногда останов можно производить заранее, если возможен случай вырождения популяций.

Основными шагами алгоритма являются шаги с 1 по 4, один проход по которым «создает новую популяцию».

Эвристические алгоритмы позволяют решать практически любые задачи оптимизации. Но их эффективность ниже, чем у локальных методов. Таким образом, фактически данные алгоритмы, хотя и могут использоваться для решения любых задач, чаще всего применяются к задачам, для которых не разработаны специальные локальные методы или решение такими методами является при заданных параметрах неэффективным.

К подобным задачам можно отнести очень популярную задачу оптимизации – задачу коммивояжера. При определенных условиях решение ее с помощью известных точных методов становится невозможным из-за большого числа вариантов.

Рассмотрим эту задачу подробнее.

Задача коммивояжера – одна из самых известных задач комбинаторной оптимизации, заключающаяся в отыскании самого выгодного маршрута, проходящего через указанные города хотя бы по одному разу, с последующим возвратом в исходный город. В условиях задачи указываются критерий выгодности маршрута (кратчайший, самый дешевый, совокупный критерий и т.п.) и соответствующие матрицы расстояний, стоимости и т.д. Как правило, указывается, что маршрут должен проходить через каждый город только один раз – в таком случае выбор осуществляется среди гамильтоновых циклов.

Существует несколько частных случаев общей постановки задачи, в частности:

- геометрическая задача коммивояжера (также называемая планарной или евклидовой, когда матрица расстояний отражает расстояния между точками на плоскости);
- треугольная задача коммивояжера (когда на матрице стоимостей выполняется неравенство треугольника);
- симметричная и асимметричная задачи коммивояжера.

Существует и так называемая обобщенная задача коммивояжера.

Решим данную задачу с помощью генетического алгоритма.

2. Постановка задачи коммивояжера

Рассмотрим метрическую задачу коммивояжера, когда расстояния между городами можно вычислить (аналог точек на плоскости). При данной постановке задачи мы имеем: число городов, координаты каждого города на плоскости:

$$\{g = (x, y\}, \quad i = \overline{1, n}$$

Таким образом, расстояние между городами можно находить как расстояние между двумя точками:

$$S(g_i, g_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Необходимо найти такой путь через города g_i, чтобы суммарное расстояние были

минимальным.

3. Построение генетического алгоритма для задачи коммивояжера

Для применения генетического алгоритма необходимо определить основные структурные элементы: вид элемента популяции, процесс скрещивания, мутации и вид фитнесс-функции.

За элемент популяции (одно возможное решение) принимаем маршрут через все города.

Каждый такой маршрут является возможным решением и не противоречит условиям задачи, хотя может быть совсем не оптимальным. Предполагаем, что все элементы популяции корректны, т.е. все они – потенциальные решения поставленной задачи и не являются противоречивыми.

В качестве фитнесс-функции мы принимаем функцию вида:

$$S = \sum_{i,j \in P} (g_i, g_j)$$

где P – множество всех связей в маршруте.

Данная функция определяет минимизируемый параметр и позволяет оценивать получаемые решения.

Метод мутации реализуется следующим образом: выбираем случайный город; находим

связи соответствующие этому городу; соединяем соседние города прямой связью исключаем выбранный город из этой связи); вставляем город в случайное место.

Пример.

Предположим, что решение до мутации имеет вид, представленный на рисунке 2.

Рисунок 2 – Решение до мутации

Применяя к нему алгоритм мутации, можем получить решение в виде представленном на рисунке 3.

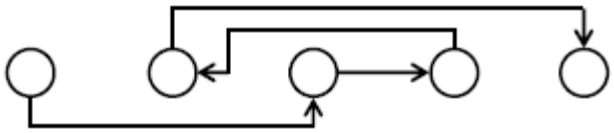


Рисунок 3 – Решение, но после мутации

Метод скрещивания реализуется сложнее, так как требует дополнительных проверок на корректность получаемого маршрута.

Алгоритм метода скрещивания:

- выбираем два маршрута из популяции, которые будут выступать в роли родителей;
- в образуемые маршруты переносим связи, которые существуют в обоих маршрутах-родителях;
- при несовпадающих связях предпочтение отдается связи в первом родителе для городов с четными номерами; если ее применить невозможно, то берем связи из второго родителя (предпочитаются связи из второго маршрута для городов с нечетными номерами), если это также невозможно, то берем из первого маршрута.

При таком задании скрещивания для получения двух решений можно применять данный метод дважды, меняя маршруты-родители местами. Тогда второй потомок будет получен при условии, что при несовпадении связей предпочтение будет отдаваться второму маршруту.

Пример. На рисунке 4 представлен пример применения данного алгоритма скрещивания к двум решениям:

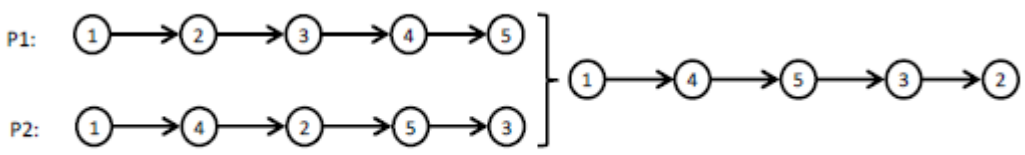


Рисунок 4- Пример «скрещивания».

Анализ полученных результатов

Применяя данный алгоритм при различных начальных условиях (размер популяций, процент мутаций, вес ближайших городов), получаем различные результаты.

1) 15 городов.

Размер популяции 1000, 15% мутации, вес ближайших городов 0%.

Результаты:

число итераций до вырождения: 1200, время выполнения: 0.321 с, погрешность от оптимального (если возможно посчитать): <5%.

2) 40 городов. Размер популяции 10000, 15% мутации, вес ближайших городов 0%.

Результаты:

число итераций до вырождения: 143000 / 375000, время выполнения: 4.3 с / 9 с.

3) 40 городов.

Размер популяции 10000, 5% мутации, вес ближайших городов 50%.

Результаты:

число итераций до вырождения: 101000, время выполнения: 3.7 с.

Последние два примера: первое минимально найденное значение с погрешностью между ними в 3% было найдено менее чем за 5 сек. Последующие решения отличаются погрешностью меньше 1%. При правильном задании размеров начальной популяции и процента мутации можно ускорить работу алгоритма.

4. Задание к лабораторной работе №3

1. Изучить методы работы генетических алгоритмов.

2. Разработать программу, реализующей генетический алгоритм для решения задачи оптимизации.

3. Программа должна иметь удобный пользовательский интерфейс. Предусмотреть вывод промежуточных результатов. В качестве входных данных использовать оптимизируемую функцию. В качестве выходных – найденные значения

переменных.

4. С помощью разработанной программы решить задачу, согласно варианту задания. (вариант выбирается по последней цифре номера зачетной книжки (если эта цифра 0, то выбирается 10 вариант)).

5. По результатам выполнения лабораторной работы оформить и защитить отчет.

5. Варианты заданий к лабораторной работе №3

Вариант 1. Найти точку пересечения функции с осью Ox.

f(x) = ln (x+1) – 2,25, x > –1. Использовать целочисленное кодирование.

Вариант 2. Найти точку пересечения функции с осью Ox.

f(x) = ln (x+1) – 2,25, x > –1. Использовать вещественное кодирование.

Вариант 3. Аппроксимировать набор точек экспоненциальной функцией:

y(x) = a*exp(b * x). Использовать целочисленное кодирование.

Вариант 4. Аппроксимировать набор точек экспоненциальной функцией:

y(x) = a*exp(b * x). Использовать вещественное кодирование.

Вариант 5. Найти минимум функции:

y(x) = x2 + 4. Использовать целочисленное кодирование.

Вариант 6. Найти минимум функции:

y(x) = x2 + 4. Использовать вещественное кодирование.

Вариант 7. Найти максимум функции:

y(x) = *k*x; x ∈[– 4; 0). Использовать целочисленное кодирование.

Вариант 8. Найти максимум функции:

y(x) = *k*x; x ∈[– 4; 0). Использовать вещественное кодирование.

Вариант 9. Найти точку перегиба функции:

f(x) = (x–1,5)3 + 3. Использовать целочисленное кодирование.

Вариант 10. Найти точку перегиба функции:

f(x) = (x–1,5)3 + 3. Использовать вещественное кодирование.

Лабораторная работа № 4

Исследование методов построения гибридных нейронных сетей

Цель работы: создание и исследование моделей сети PNN в системе MATLAB.

Общие сведения

Нейронные сети PNN (ProbabilisticNeuralNetwork) предназначены для решения вероятностных задач и, в частности, задач классификации.

Архитектура сети

Архитектура сети PNN базируется на архитектуре радиальной базисной сети, но в качестве второго слоя использует так называемый конкурирующий слой, который подсчитывает вероятность принадлежности входного вектора к тому или иному классу и в конечном счете сопоставляет вектор с тем классом, вероятность принадлежности к которому выше. Структура сети PNN представлена на рисунке 1.

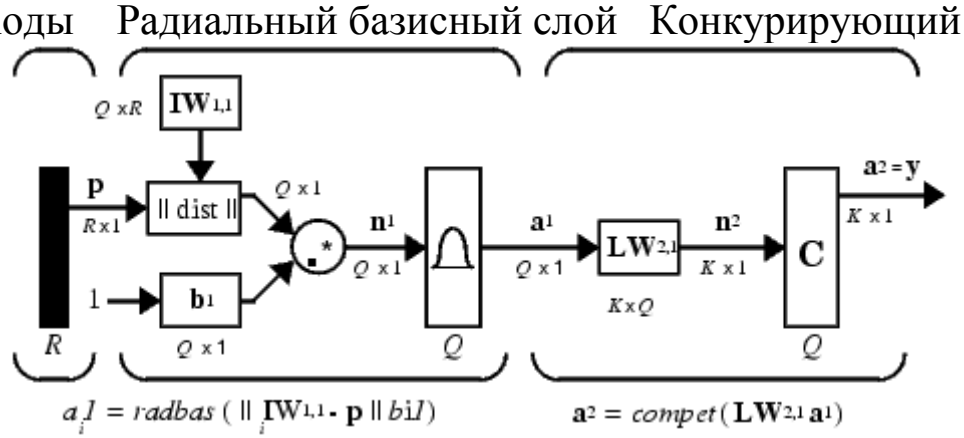


Рисунок 1 – Структура сети PNN

Предполагается, что задано обучающее множество, состоящее из Q пар векторов вход/цель. Каждый вектор цели имеет K элементов, указывающих класс принадлежности, и, таким образом, каждый вектор входа ставится в соответствие одному из K классов.

В результате может быть образована матрица связности **T** размера K×Q, состоящая из нулей и единиц, строки которой соответствуют классам принадлежности, а столбцы – векторам входа. Таким образом, если элемент **T**(i,j) матрицы связности равен 1, то это означает, что j-й входной вектор принадлежит к классу /.

Весовая матрица первого слоя $\mathbf{IW}^{11}$ (net.IW{1,1}) формируется с использованием векторов входа из обучающего множества в виде матрицы $\mathbf{P}^T$. Когда подается новый вход, блок ||dist|| вычисляет близость нового вектора к векторам обучающего множества; затем вычисленные расстояния умножаются на смещения и подаются на вход функции активации radbas. Вектор обучающего множества, наиболее близкий к вектору входа, будет представлен в векторе выхода $\mathbf{a}^1$ числом близким к 1.

Весовая матрица второго слоя $\mathbf{LW}^{21}$ (net.LW{2,1}) соответствует матрице связности T, построенной для данной обучающей последовательности. Эта операция может быть выполнена с помощью функции ind2vec, которая преобразует вектор целей в матрицу связности **T**. Произведение $\mathbf{T}^*\mathbf{a}^1$ определяет элементы вектора $\mathbf{a}^1$, соответствующие каждому из K классов. В результате конкурирующая функция активации второго слоя compet формирует на выходе значение, равное 1 для самого большего по величине элемента вектора $\mathbf{n}^2$ и 0 в остальных случаях. Таким образом, сеть PNN выполняет классификацию векторов входа по K классам.

Синтез сети

Для создания нейронной сети PNN предназначена функция newpnn. Определим 7 следующих векторов входа и соотнесем каждый из них к одному из 3 классов:

P = [0 0;1 1;0 3;1 4;3 1;4 1;4 3];

Tc = [1 1 2 2 3 3 3].

Ставится задача определения: к какому классу принадлежит данный вектор.

Запишем матрицу связности:

1 1 0 0 0 0 0

T = 0 0 1 1 0 0 0

0 0 0 0 1 1 1

Первая строка – 1 класс; вторая – 2 класс; третья – 3 класс.

В данном случае имеем двумерный вектор (функция от двух переменных).

Графически эта функция представлена на рисунке 2.

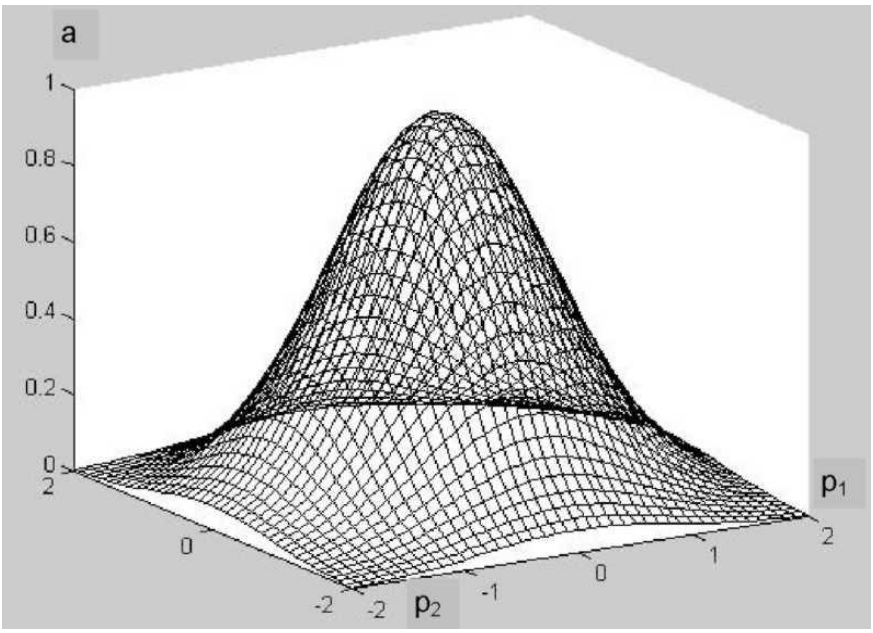


Рисунок 2 – График функции

Найдем выходы первого слоя по формуле

$$a = e^{-\left(\sqrt{(p_1-w_1)^2+(p_2-w_2)^2 b}\right)^2}.$$

Подадим на все нейроны первый обучающий вектор $p_1 = 0$; $p_2 = 0$.

Тогда на выходах имеем:

$$a_1^1 = 1;\ \ a_2^1 = 0,25;\ \ a_3^1 = 0,02;\ \ a_4^1 = 0;\ \ a_5^1 = 0,001;\ \ a_6^1 = 0;\ \ a_7^1 = 0.$$

Веса первого слоя:

$$w_{11}^1 = 0;\ \ w_{12}^1 = 1;\ \ w_{13}^1 = 0;\ \ w_{14}^1 = 1;\ \ w_{15}^1 = 3;\ \ w_{16}^1 = 4;\ \ w_{17}^1 = 4;$$

$$w_{21}^1 = 0;\ \ w_{22}^1 = 1;\ \ w_{23}^1 = 3;\ \ w_{24}^1 = 4;\ \ w_{25}^1 = 1;\ \ w_{26}^1 = 1;\ \ w_{27}^1 = 3.$$

Подадим на все нейронывторой обучающий вектор $p_1 = 0$; $p_2 = 0$.

Тогда на выходах имеем:

$$a_1^1 = e^{-\left(\sqrt{(1-0)^2+(1-0)^2 b}\right)^2};$$

$$a_2^1 = e^{-\left(\sqrt{(1-1)^2+(1-1)^2 b}\right)^2};$$

$$a_3^1 = e^{-\left(\sqrt{(1-0)^2+(1-3)^2 b}\right)^2};$$

$$a_4^1 = e^{-\left(\sqrt{(1-1)^2+(1-4)^2 b}\right)^2};$$

$$a_5^1 = e^{-\left(\sqrt{(1-3)^2+(1-1)^2 b}\right)^2};$$

$$a_6^1 = e^{-\left(\sqrt{(1-4)^2+(1-1)^2 b}\right)^2};$$

$$a_7^1 = e^{-\left(\sqrt{(1-4)^2+(1-3)^2 b}\right)^2};$$

Весам второго слоя присваивают значения разреженной матрицы:

$$w_{11}^2 = 1;\ \ w_{12}^2 = 1;\ \ w_{13}^2 = 0;\ \ w_{14}^2 = 0;\ \ w_{15}^2 = 0;\ \ w_{16}^2 = 0;\ \ w_{17}^2 = 0;$$

$$w_{21}^2 = 0;\ \ w_{22}^2 = 0;\ \ w_{23}^2 = 1;\ \ w_{24}^2 = 1;\ \ w_{25}^2 = 0;\ \ w_{26}^2 = 0;\ \ w_{27}^2 = 0;$$

$$w_{31}^2 = 0;\ \ w_{32}^2 = 0;\ \ w_{33}^2 = 0;\ \ w_{34}^2 = 0;\ \ w_{35}^2 = 1;\ \ w_{36}^2 = 1;\ \ w_{37}^2 = 1.$$

Тогда на выходе 2-го слоя имеем:

$a^2 = \text{compet}(LW^{2,1}a^1);$

$$\begin{bmatrix} W_{11} & W_{12} & W_{13} & W_{14} & W_{15} & W_{16} & W_{17} \\ W_{21} & W_{22} & W_{23} & W_{24} & W_{25} & W_{26} & W_{27} \\ W_{31} & W_{32} & W_{33} & W_{34} & W_{35} & W_{36} & W_{37} \end{bmatrix} \begin{bmatrix} a_1^1 \\ a_2^1 \\ a_3^1 \\ a_4^1 \\ a_5^1 \\ a_6^1 \\ a_7^1 \end{bmatrix}$$

Подставляем значения

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0,25 \\ 0,02 \\ 0 \\ 0,001 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1,25 \\ 0,02 \\ 0,001 \end{bmatrix}$$

получаем:

$$a^2 = \text{compet} \begin{bmatrix} 1,25 \\ 0,02 \\ 0,001 \end{bmatrix}$$

Итак, данный вектор принадлежит к 1-му классу.

На этом ручной расчет закончен.

Перейдем к созданию и исследованию модели в системе MATLAB:

```
clear, P = [0 0;1 1;0 3;1 4;3 1;4 1;4 3]';
Tc = [1 1 2 2 3 3 3];
```

Вектору**Tc** поставим в соответствие матрицу связности **T** в виде разреженной матрицы вида

```
T = ind2vec(Tc)
T =
(1,1)        1
(1,2)        1
(2,3)        1
(2,4)        1
(3,5)        1
(3,6)        1
(3,7)        1
```

которая определяет принадлежность первых двух векторов классу 1, двух последующих – классу 2 и трех последних – классу 3. Полная матрица**T** имеет вид

```
T = full(T)
T =
1     1     0     0     0     0     0
0     0     1     1     0     0     0
0     0     0     0     1     1     1
```

Массивы P и T задают обучающее множество, что позволяет выполнить формирование сети, промоделировать ее, используя массив входов P, и удостовериться, что сеть правильно решает задачу классификации на элементах обучающего множества. В результате моделирования сети формируется матрица связности, соответствующая массиву векторов входа. Функция vec2ind предназначена для преобразования матрицы связности в индексный вектор:

```
net = newpnn(P,T);
net.layers{1}.size % Число нейронов в сети PNN
ans =
7
Y = sim(net,P); Yc = vec2ind(Y)
Yc =
1     1     2     2     3     3     3
```

Результат подтверждает правильность решения задачи классификации.

Теперь выполним классификацию некоторого набора произвольных векторов **p**, не принадлежащих обучающему множеству, используя ранее созданную сеть PNN:

```
p = [1 3; 0 1; 5 2]';
```

Осуществляя моделирование сети для этого набора векторов, получаем

```
a = sim(net,p);  ac = vec2ind(a)
ac =
2     1     3
```

Порядок выполнения работы

1. Для заданных преподавателем параметров радиальной базисной нейронной сети (таблица) подготовить массивы входных векторов **P** и целей**T** для нейронной сети PNN.

2. Разработать структурную схему нейронной сети PNN.

3. Выполнить ручной расчет определения принадлежности к классу всех векторов из обучающего множества.

4. Создать полученную нейронную сеть в системе MATLAB и сравнить полученные результаты с ручным расчетом.

5. Определить параметры созданной нейронной сети (количество нейронов в каждом слое, веса и смещения нейронов).

6. Выполнить классификацию набора из трех произвольных входных векторов, не принадлежащих обучающему множеству, используя созданную сеть.

7. Составить отчет, который должен содержать:

– цель лабораторной работы;

– массивы входных векторов P и целей T;

– структурную схему нейронной сети;

– ручной расчет;

– текст и результаты работы программы;

– выводы.

Номер варианта	Значения векторов входа	Значения вектора индексов классов
1	[1 3; 2 4; 1 1; 0 0; 4 2; 3 1; 5 3]	[1 1 2 2 3 3 3]
2	[3 2; 4 1; 1 5; 2 4; 0 3; 1 1; 2 2]	[3 3 2 2 2 1 1]
3	[2 5; 1 3; 1 2; 0 1; 1 0; 3 2; 4 1]	[2 2 1 1 1 3 3]
4	[0 1; 1 0; 2 2; 4 2; 5 3; 2 4; 1 5]	[1 1 1 3 3 2 2]
5	[3 2; 4 0; 5 1; 2 4; 0 3; 1 1; 0 0]	[3 3 3 2 2 1 1]
6	[2 3; 1 4; 0 1; 1 1; 2 0; 4 2; 5 3]	[2 2 1 1 1 3 3]
7	[2 1; 1 0; 1 3; 2 4; 0 5; 4 2; 5 0]	[1 1 1 2 2 3 3]
8	[4 0; 5 1; 3 1; 2 0; 1 1; 2 4; 0 3]	[3 3 3 1 1 2 2]
9	[1 3; 2 4; 0 5; 4 1; 3 3; 0 0; 1 1]	[2 2 2 3 3 1 1]
10	[0 1; 1 0; 3 2; 4 0; 5 2; 1 3; 2 4]	[1 1 3 3 3 2 2]

Лабораторная работа №5

Изучение принципа поиска решения в пространстве состояний

1 Цель работы

Ознакомиться с методами поиска решений в пространстве состояний и научиться реализовывать их на практике.

2 Теоретические сведения

Решение таких задач, как планирование расписаний, игровых задач, задач обучения и т. д. предполагает решение задачи поиска в пространстве состояний. Большинство задач поиска формулируются следующим образом: найти путь из начального состояния (узла или позиции) в целевое состояние (узел или позицию). Задача поиска обычно определяется указанием:

- пространства состояний;
- начального состояния;
- функции генерации соседних для данного состояний;
- целевого условия, т.е. условием, к достижению которого следует стремиться. «Целевые вершины» – это вершины, удовлетворяющие этим условиям. Целевое условие может выполняться для нескольких состояний.

Пространство состояний – это граф, вершины которого соответствуют ситуациям, встречающимся в задаче («проблемные ситуации»), а дуги – разрешенным переходам из одних состояний в другие. Решение задачи сводится к поиску пути в этом графе, т.е. в нахождении любой серии перемещений из начального состояния в состояние, удовлетворяющее целевому условию. Например, для задачи поиска кратчайшего пути между двумя городами пространство состояний представляет собой города на карте и его можно представить в виде графа следующим образом:

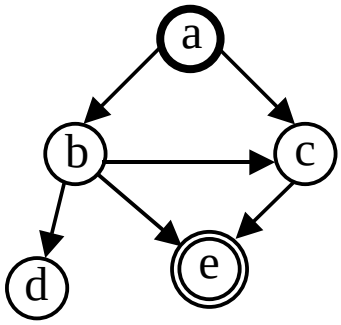


Рисунок 1 - Представление пространства состояний в виде графа

На этом рисунке состояние a является стартовым состоянием, а состояние e – целевым. Решение задачи сводится к нахождению кратчайшего пути из города a в город e.

Процесс решения задачи включает в себя поиск в графе, при этом, как правило, возникает проблема, как обрабатывать альтернативные пути поиска. Существуют две основные стратегии перебора альтернатив, а именно поиск в глубину и поиск в ширину. Пространство состояний некоторой задачи определяет «правила игры»: вершины пространства состояний соответствуют ситуациям, а дуги – разрешенным ходам или действиям, или шагам решения задачи. Каждому разрешенному ходу или действию можно приписать его стоимость. В тех случаях, когда каждый ход имеет стоимость, мы заинтересованы в отыскании решения минимальной стоимости. Стоимость решения – это сумма стоимостей дуг, из которых состоит «решающий путь» – путь из стартовой вершины в целевую. Даже если стоимости не заданы, все равно может возникнуть оптимизационная задача: нас может интересовать кратчайшее решение.

Представление пространства состояний

Когда поиск проводится в графе состояний, граф фактически разворачивается в дерево, причем некоторые пути, возможно, дублируются в разных частях этого дерева. Поэтому для простоты все приведенные ниже примеры касаются поиска в дереве. На рисунке 2 приведен граф состояний и его представление в виде дерева.

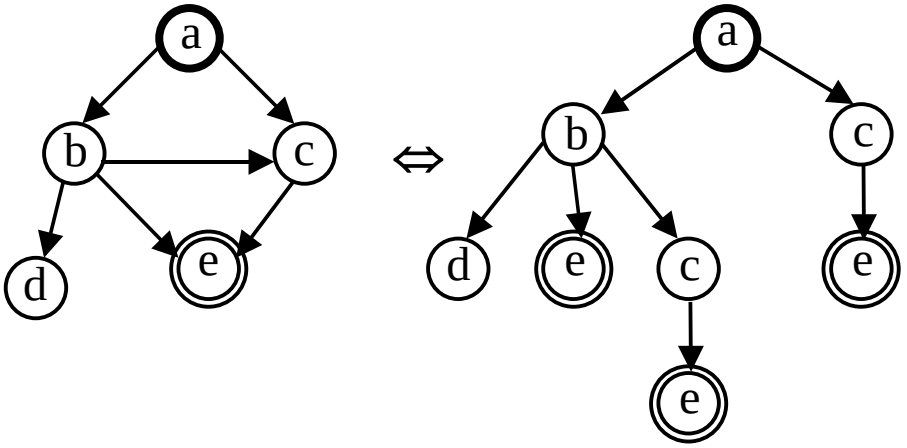


Рисунок 2 - Представление пространства состояний в виде дерева

Для дальнейшего изложения будем использовать дерево состояний, представленное на рисунок 3. Для простоты реализации алгоритма поиска такое дерево может быть представлено как список всех возможных путей от его листьев до корня.

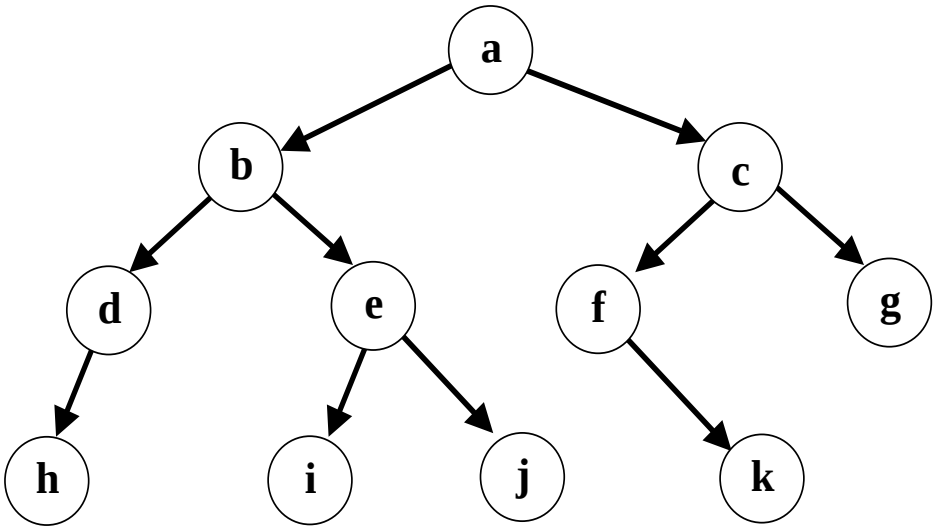


Рисунок 3 - Дерево состояний

Ниже дано представление дерева состояний (рисунок 3) в виде списка всех возможных путей:

((h d b a)
(i e b a)
(j e b a)
(k f c a)
(g c a))

Алгоритм поиска

В общем, алгоритм поиска в дереве состояний следующий:

1. Создать очередь, содержащую один частичный путь (начальное состояние).

2. Расширить первый в очереди частичный путь. При этом первый путь извлекается из очереди, а для первой вершины этого пути находится множество связанных с ней вершин, т.е. строятся следующие частичные пути. Затем очередь частичных путей обновляется.

3. Если какой-то из частичных путей содержит целевой узел, вывести этот частичный путь.

Существует много различных подходов к проблеме поиска решающего пути (т.е. пути, ведущего из стартовой вершины в целевую) для задач, сформулированных в терминах пространства состояний. К основным относятся две стратегии поиска: поиск в глубину и поиск в ширину. Поиск в глубину и поиск в ширину различаются только способом пополнения очереди частичных путей. Поиск в ширину использует принцип FIFO – старый путь уничтожается и новый добавляется в конец очереди. Поиск в глубину использует принцип LIFO – старый путь уничтожается и новый добавляется в начало очереди.

Поиск в глубину

Под названием «поиск в глубину» понимается порядок рассмотрения альтернатив в пространстве состояний. Всегда, когда алгоритму поиска в глубину надлежит выбрать из нескольких вершин ту, в которую следует перейти для продолжения поиска, он предпочитает самую «глубокую» из них. Самая глубокая вершина – это вершина, расположенная дальше других от стартовой вершины. На рисунке 4 показано в каком порядке алгоритм поиска в глубину обходит вершины дерева состояний.

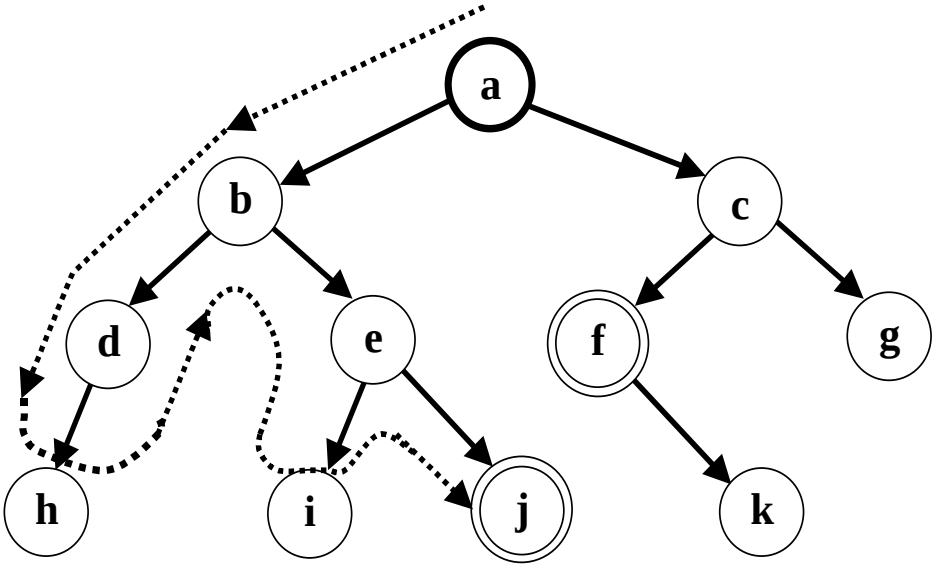


Рисунок 4 - Обход вершин дерева состояний при поиске в глубину

На этом рисунке: a – стартовая вершина, f и j – целевые вершины. Порядок, в котором происходит обход вершин в пространстве состояний при поиске в глубину следующий: a, b, d, h, e, i, j. Вначале алгоритм находит решение: [a, b, e, j]. После возврата обнаружено другое решение: [a, c, f].

Ниже приведена реализация алгоритма поиска в глубину в дереве состояний на языке Паскаль.

Реализацию поиска в глубину выполняет процедура DephtFirstSearch. Процедура в качестве параметров получает указатель на вершину дерева и указатели на начало и конец двунаправленной очереди, в которой размещаются указатели на соответствующие узлы дерева. Двунаправленная очередь выбрана для удобства занесения и удаления элементов в конце очереди.

На первом этапе в очередь заносится указатель на вершину дерева. Для реализации первого поиска процедура DephtFirstSearch вызывает процедуру PBDPPm, которая в качестве параметров получает указатели на вершину дерева и на начало и конец очереди из одного элемента – указатель на вершину дерева. Процедура PBDPPm отыскивает путь от корня дерева к его самому левому листу и помещает этот путь в очередь, которая является результатом первого поиска и возвращается процедурой DephtFirstSearch в вызывающую программу для последующей обработки. Все последующие поиски выполняются по следующей

схеме. После очередной обработки очереди вызывается процедура DephtFirstSearch. В качестве параметров она получает указатели на вершину дерева и на начало и конец очереди, являющейся результатом предыдущего поиска. В процедуре очередь обрабатывается следующим образом. В соответствии с описанием пути, содержащимся в очереди, выполняется спуск по дереву. Когда достигается лист дерева, из конца очереди удаляется элемент, соответствующий листу дерева, и начинается откат по пути спуска. В процессе отката возможны следующие случаи:

– если возврат к узлу выполняется из правого поддерева, то из конца очереди удаляется соответствующий этому узлу элемент и откат продолжается;

– если возврат к узлу выполняется из левого поддерева, то выполняется проверка – есть ли у него правый сын;

– если правого сына у узла нет, то из конца очереди удаляется соответствующий этому узлу элемент и откат продолжается;

– если правый сын у узла есть, то откат прекращается. Указатель на правого сына узла передается в процедуру PBDPPm как указатель на вершину поддерева. Процедура отыскивает путь от корня поддерева к его самому левому листу и дописывает соответствующие элементы в конец двунаправленной очереди, которая, является результатом очередного поиска. Результат очередного поиска возвращается процедурой DephtFirstSearch в вызывающую программу.

Ниже приводится полное описание всех необходимых типов данных и вспомогательных процедур.

```
type TPNode= ^TNode;
    TNode = record
            Inf: array [1..10] Of Char;
            Left, Right: TPNode;
           end;
    TPointDQueue = ^TDQueue; {Двунаправленный список}
    TDQueue = record
              PNTree: TPNode;
              Next, Prev: TPointQueue;
             end;
```

```
var Rut: TPNode;
    BeginQueue, EndQueue: TPointDQueue;
{InsertDQ - процедура занесения элемента в двунаправленную очередь}
procedure InsertDQ(var BegQue, EndQue:TPointDQueue;
                   NewElm: TPNode);
var WP: TpointDQueue;
begin
  New(WP);
  WP^.PNTree := NewElm;
  WP^.Next := Nil;
  WP^.Prev := Nil;
  if  EndQue = Nil then
     begin
       BegQue := WP; {случай первого элемента}
       EndQue := WP
     end
  else
     begin
       EndQue^.Next := WP;
       WP^.Prev := EndQue;
       EndQue := WP
     end
end; {InsertDQ}
{EmptyDQ - функция проверки очереди на пустоту}
function EmptyDQ(BegQue: TPointDQueue): boolean;
begin
   if BegQue = Nil then Empty: = True
   else Empty := False
end; {EmptyDQ}
{RemoveDQEnd – процедура удаления последнего элемента
  двунаправленной очереди}
procedure RemoveDQEnd(var BegQue,EndQue:
                      TpointQueue;
                  PrevEndQue:TpointQueue);
var WP: TpointQueue;
begin
```

```
        if Empty(BegQue) = True then Exit;
            {элемент удаляется из конца очереди}
        WP := EndQue;
        EndQue := EndQue^.Prev;
        Dispose(WP);
            {если в очереди оставался один (последний) элемент, то его
              удаление приводит к пустой очереди, следовательно, BegQue = Nil и
              EndQue = Nil}
        if EndQue = Nil then BegQue := Nil;
    end; {RemoveQEnd}
```

{PBDPPm – выполняет поиск самого левого из возможных путей от вершины, переданной как корень дерева. Или иначе, процедура выполняет поиск пути от корня к самому левому листу дерева}

```
    procedure PBDPPm(PTree: TPNode;
                  var BegQ, EndQ: TPointQueue);
    begin
        {вставка очередного элемента в очередь}
      InsertQ (BegQ, EndQ, PTree);
      if Ptree^.Left <> Nil then
        PBDPPm(PTree^.Left, BegQ, EndQ)
      else
        if Ptree^.Right <> Nil then
          PBDPPm(PTree^.Right, BegQ, EndQ)
    end; {PBDPPm}
```

{DephtFirstSearch – процедура возвращает указатель на начало двунаправленной очереди, содержащей указатели на все узлы дерева, полученные в результате очередного поиска в глубину.}

```
    procedure DephtFirstSearch(var WRoot: TNode;
                    var BegQ, EndQ: TPointQueue);
    var Prev, Curr: TPointQueue;
    begin
        {В очередь занесен только один элемент – корень дерева.}
```

```
        if BegQ = EndQ then
                {Поиск первого пути в дереве - пути от корня к самому левому
                 листу дерева }
          PBDPPm(WRoot, BegQ, EndQ)
        else
                {В очередь занесено несколько элементов. Поиск пути от корня к
                  очередному листу дерева}
          begin
            Prev := BegQ;
            Curr := BegQ^.Next;
            if Curr <> Nil then
                        {Спуск по предыдущему пути до листа дерева. Выполнять
                         контроль на достижение листа дерева нет
                         необходимости т.к. такой контроль выполняется
                         автоматически по концу очереди}
              if (WRoot^.Left^.Inf=Curr^.Inf) then
                begin
                                {Спуск по левой ветви узла}
                  DephtFirstSearch
                       (WRoot^.Left, Curr, EndQ);
                                {Возврат из рекурсии из левого поддерева}
                  if (WRoot^.Right = Nil) then
                                {При возврате из левого поддерева, если у
                                 узла нет правого сына, то узел удаляется из
                                 пути поиска}
                    RemoveQEnd
                        (BegQ, EndQ, PrevEndQue)
                  else
                                {При возврате из левого поддерева, если у
```

```
                                    узла есть правый сын, он
становится                                              корнем
поддерева,                          для                 которого
определяется путь к его самому левому
листу. Этот путь дописывается в конец
очереди}
                    PBDPPm
                      (WRoot^.Right, BegQ, EndQ);
                  end
                else
                  begin
                                {Спуск по правой ветви узла}
                   DephtFirstSearch
                      (WRoot^.Right, Curr, EndQ);
                                {Возврат из рекурсии из правого
поддерева}
                   RemoveQEnd(BegQ, EndQ,PrevEndQue);
                                {При возврате из правого поддерева
узел
                                  удаляется из пути поиска}
                  end
              else
                            {Лист дерева из предыдущего пути поиска
автоматически
                              удаляется из дерева и начинается откат по
пути поиска}
                 RemoveQEnd(BegQ, EndQ, PrevEndQue);
            end;
        end; {DephtFirstSearch}
        Обращение к этим процедурам имеет вид:
        begin
          • • •
          BeginQueue := Nil;
          EndQueue := Nil;
              {занесение корня дерева в очередь}
          InsertQ(BeginQueue, EndQueue, Rut);
              {первый поиск}
```

```
DephtFirstSearch(Rut, BeginQueue, EndQueue);
• • •
    {очередной поиск}
DephtFirstSearch(Rut, BeginQueue, EndQueue);
end.
```

В случае приведенного выше примера дерева состояний этот процесс будет развиваться следующим образом:

1. Начинаем с начального множества кандидатов:

[[a]]

2. Порождаем продолжения пути [a]:

[[b,a], [c,a]]

(Обратите внимание, что пути записаны в обратном порядке.)

3. Удаляем первый путь из множества кандидатов и порождаем его продолжения: [[d,b,a], [e,b,a]], а затем добавляем список продолжений в начало очереди кандидатов. Получим:

[[d,b,a], [e,b,a], [c, a]]

4. Удаляем [d,b,a] из очереди кандидатов, а затем добавляем все его продолжения в начало очереди. Получаем:

[[h,d,b,a], [e,b,a], [c,a]]

5. Путь [h,d,b,a] не имеет продолжений. Поэтому просто удаляем его из очереди кандидатов. Затем удаляем путь [e,b,a] из очереди кандидатов и добавляем все его продолжения в начало очереди. Получаем:

[[i,e,b,a], [j,e,b,a], [c,a]]

6. После удаления пути [i,e,b,a] из очереди обнаруживается, что путь [j,e,b,a] содержит целевую вершину. Этот путь выдается в качестве решения.

Поиск в ширину

В противоположность поиску в глубину стратегия поиска в ширину предусматривает переход в первую очередь к вершинам, ближайшим к стартовой вершине. В результате процесс поиска имеет тенденцию развиваться более в ширину, чем в глубину, что иллюстрирует рисунок 5.

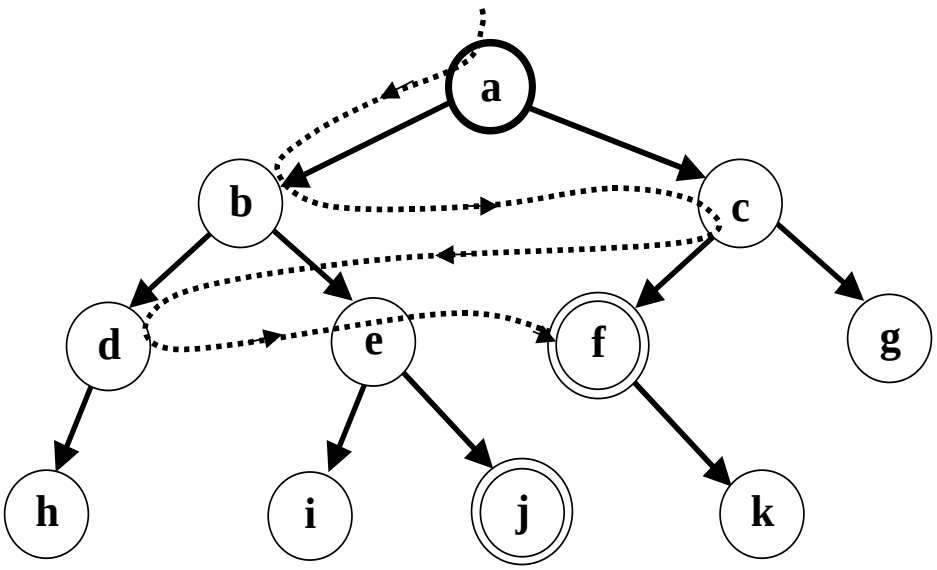


Рисунок5 - Обход вершин дерева состояний при поиске в ширину

На этом рисунке: a – стартовая вершина, f и j – целевые вершины. Применение стратегии поиска в ширину дает следующий порядок обхода вершин: [a, b, c, d, e, i]. Более короткое решение: [a, c, f] найдено раньше, чем более длинное [a, b, e, j].

Поиск в ширину программируется не так легко, как поиск в глубину. Причина состоит в том, что нам приходится сохранять все множество альтернативных вершин-кандидатов, а не только одну вершину, как при поиске в глубину. Более того, если мы желаем получить при помощи процесса поиска решающий путь, то одного множества вершин не достаточно. Поэтому надо хранить не множество вершин-кандидатов, а множество путей-кандидатов. В случае спискового представления множества кандидатов само множество будет списком путей, а каждый путь – списком вершин, перечисленных в обратном порядке, т. е. головой списка будет самая последняя из порожденных вершин, а последним элементом списка будет стартовая вершина. Для того, чтобы выполнить поиск в ширину при заданном множестве путей-кандидатов, нужно:

– если голова первого пути является целевой вершиной, взять этот путь в качестве решения;

– иначе удалить первый путь из множества кандидатов и породить множество всех возможных продолжений этого пути на

один шаг; множество продолжений добавить в конец множества кандидатов, а затем выполнить поиск в ширину с полученным новым множеством.

Ниже приведена реализация алгоритма поиска в ширину в дереве состояний на языке Паскаль.

Для реализации поиска в ширину корень дерева помещается в очередь и она передается в процедуру. В процедуре очередь обрабатывается следующим образом. Отмечается начало и конец очереди. Для первого элемента очереди – узла дерева, раскрываются его левый и правый сыновья и соответствующие элементы последовательно помещаются в очередь вслед за признаком конца очереди. Первый элемент удаляется из очереди и все повторяется заново до тех пор, пока исходная очередь не будет исчерпана. Таким образом, если на вход процедуры передается очередь из узлов дерева уровня К, то после своего выполнения процедура возвращает очередь из узлов дерева уровня К + 1. Полученная очередь может подвергнуться обработке, после чего выполняется очередное обращение к процедуре и т.д.

Ниже приводится полное описание всех необходимых типов данных и вспомогательных процедур.

```
type TPNode= ^TNode;
    TNode = record
              Inf: array [1..10] Of Char;
              Left, Right: TPNode;
            end;
    TPointQueue = ^ TQueue;
    TQueue = record
              PNTree: TPNode;
              Next: TPointQueue;
            end;
var Rut: TPNode;
   BeginQueue, EndQueue, BeginList: TpointQueue;
{процедура занесения элемента в очередь}
procedure InsertQ(var BegQue, EndQue: TPointQueue;
                              NewElm: TPNode);
var WP: TpointQueue;
begin
```

```
        New (WP);
        WP^.PNTree := NewElm;
        WP^.Next := Nil;
        if  EndQue = Nil then
           begin
             BegQue := WP; {случай первого элемента}
             EndQue := WP
           end
        else
           begin
             EndQue^.Next := WP;
             EndQue := WP;
           end
      end; {InsertQ}
      {функция проверки очереди на пустоту}
      function EmptyQ(BegQue: TPointQueue): boolean;
      begin
        if BegQue = Nil then Empty := True
        else Empty := False
      end; {EmptyQ}
      {RemoveQL - процедура извлекает элемент из очереди, но не
удаляет его}
      procedure RemoveQL(var BegQue, EndQue, BegQueL:
                                  TPointQueue);
      begin
        if Empty (BegQue) = False then
          begin
                    {элемент удаляется из очереди}
           BegQueL := BegQue;
           BegQue := BegQue^.Next;
           if BegQue = Nil then EndQue:= Nil
                      {если в очереди оставался один (последний)
элемент,                                                    то
                       его удаление приводит к пустой очереди,
следовательно,
                       BegQue = Nil и EndQue = Nil}
          end
```

```
end; {RemoveQL}
```

Следующая процедура возвращает указатель на начало очереди, содержащей указатели на все узлы дерева, полученные в результате горизонтального обхода узлов дерева уровня К = 1- N.

```
procedure BreadthFirstSearch(var BegQ, EndQ:
                                TPointQueue);
var WP: TPointQueue;
begin
   WP:= EndQ;
   repeat
         {элементы дописываются в очередь до тех пор, пока не будут
          перебраны все узлы дерева уровня К.}
    if BegQ^.PNTree^.Left <> Nil then
            {занесение левого потомка узла дерева в очередь.}
      InsertQ (BegQ, EndQ, WPB^.PNTree^.Left);
    if BegQ^.PNTree^.Right <> Nil then
            {занесение правого потомка узла дерева в очередь.}
      InsertQ (BegQ, EndQ, WPB^.PNTree^.Right);
            {изъятие элемента из очереди и его занесение в список}
     RemoveQL(BegQ, EndQ, BegL: TPointQueue);
   until (BegQ = WP);
end; {BreadthFirstSearch}
```

Обращение к этим процедурам имеет вид:

```
begin
   • • •
   BeginQueue := Nil;
   EndQueue := Nil;
       {занесение корня дерева в очередь}
   InsertQ (BeginQueue, EndQueue, Rut);
       {первый поиск}
   BreadthFirstSearch(BeginQueue, EndQueue);
```

```
• • •
    {очередной поиск}
  BreadthFirstSearch(BeginQueue, EndQueue);
end.
```

В случае приведенного выше примера дерева процесс поиска в ширину будет развиваться следующим образом:

1. Начинаем с начального множества кандидатов:

[[a]]

2. Порождаем продолжения пути [a]:

[[b,a], [c,a]]

(Обратите внимание, что пути записаны в обратном порядке.)

3. Удаляем первый путь из множества кандидатов и порождаем его продолжения:

[[d,b,a], [e,b,a]]

Добавляем список продолжений в конец списка кандидатов:

[c,a], [d,b,a], [e,b,a]]

4. Удаляем [c, a], а затем добавляем все его продолжения в конец множества кандидатов. Получаем:

[[d,b,a], [e,b,a], [f,c,a], [g,c,a]]

Далее, после того, как пути [d,b,a] и [e,b,a] будут продолжены, измененный список кандидатов примет вид:

[[f,c,a],[g,c,a],[h,d,b,a],[i,e,b,a],[j,e,b,a]]

В этот момент обнаруживается путь [f, c, a], содержащий целевую вершину f . Этот путь выдается в качестве решения.

Поиск с предпочтением: эвристический поиск

Поиск в графах при решении задач, как правило, невозможен без решения проблемы комбинаторной сложности, возникающей из-за быстрого роста числа альтернатив. Эффективным средством борьбы с этим служит эвристический поиск. Один из путей использования эвристической информации о задаче (т.е. информации, относящейся к решаемой задаче и используемой для управления поиском) – это получение численных эвристических оценок для вершин пространства состояний. Оценка вершины указывает нам, насколько данная вершина перспективна с точки зрения достижения цели. Идея состоит в том, чтобы всегда продолжать поиск, начиная с наиболее перспективной вершины,

выбранной из всего множества кандидатов. Подобно поиску в ширину, поиск с предпочтением начинается со стартовой вершины и использует множество путей-кандидатов. В то время, как поиск в ширину всегда выбирает для продолжения самый короткий путь (т.е. переходит в вершины наименьшей глубины), поиск с предпочтением вносит в этот принцип следующее усовершенствование: для каждого кандидата вычисляется оценка и для продолжения выбирается кандидат с наилучшей оценкой. Допустим s – стартовая вершина дерева состояний, а t – целевая. Для каждого узла n, лежащего на пути из s в t, вычисляется оценочная функция f(n). f – это эвристическая оценочная функция, при помощи которой мы получаем для каждой вершины n оценку f(n) «трудности» этой вершины». Тогда наиболее перспективной вершиной-кандидатом следует считать вершину, для которой эта функция принимает минимальное значение. Функция f(n) должна быть построена таким образом, чтобы давать оценку стоимости оптимального решающего пути из стартовой вершины s к одной из целевых вершин при условии, что этот путь проходит через вершину n. Оценку f(n) можно представить в виде суммы из двух слагаемых (рисунок 6.): f(n) = g(n) + h(n). Здесь g(n) – оценка оптимального пути из s в n; h(n) –- оценка оптимального пути из n в t.

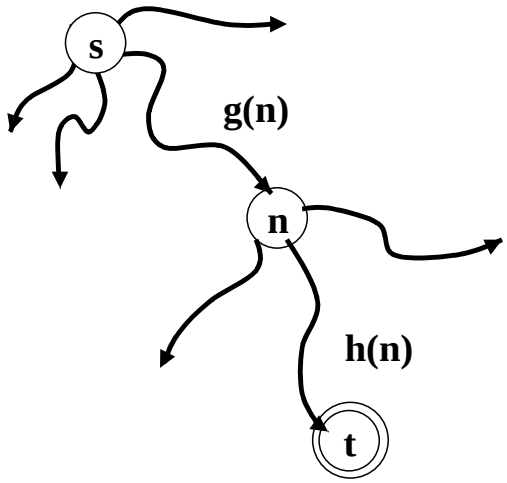


Рисунок 6 - Построение эвристической оценки f(n) стоимости самого дешевого пути из s в t .

Когда в процессе поиска мы попадаем в вершину n, мы

оказываемся в следующей ситуации: путь из s в n уже найден, и его стоимость может быть вычислена как сумма стоимостей составляющих его дуг (будем предполагать, что для дуг пространства состояний определена функция стоимости c(n, n') – стоимость перехода из вершины n к вершине-преемнику n'). Этот путь не обязательно оптимален (возможно, существует более дешевый, еще не найденный путь из s в n), однако стоимость этого пути можно использовать в качестве оценки g(n) минимальной стоимости пути из s в n. Что же касается второго слагаемого h(n), то о нем трудно что-либо сказать, поскольку к этому моменту область пространства состояний, лежащая между n и t, еще не «изучена» программой поиска. Поэтому, как правило, о значении h(n) можно только строить догадки на основании эвристических соображений, т.е. на основании тех знаний о конкретной задаче, которыми обладает алгоритм. Поскольку значение h зависит от предметной области, универсального метода для его вычисления не существует. Будем считать, что тем или иным способом функция h задана, и сосредоточим внимание на деталях алгоритма поиска с предпочтением.

Можно представлять себе поиск с предпочтением следующим образом. Процесс поиска состоит из некоторого числа конкурирующих между собой подпроцессов, каждый из которых занимается своей альтернативой, т.е. просматривает свое поддерево. У поддеревьев есть свои поддеревья, их просматривают подпроцессы подпроцессов и т.д. В каждый данный момент среди всех конкурирующих процессов активен только один – тот, который занимается наиболее перспективной к настоящему моменту альтернативой, т.е. альтернативой с наименьшим значением f. Остальные процессы спокойно ждут того момента, когда f-оценки изменятся и в результате какая-нибудь другая альтернатива станет наиболее перспективной. Тогда производится переключение активности на эту альтернативу. Механизм активации-дезактивации процессов функционирует следующим образом: процесс, работающий над текущей альтернативой высшего приоритета, получает некоторый «бюджет» и остается активным до тех пор, пока его бюджет не исчерпался. Находясь в активном состоянии, процесс продолжает углублять свое

поддерево. Встретив целевую вершину, он выдает соответствующее решение. Величина бюджета, предоставляемого процессу на данный конкретный запуск, определяется эвристической оценкой конкурирующей альтернативы, ближайшей к данной.

На рисунке 7 показан пример поведения конкурирующих процессов. Дана карта, а задача состоит в том, чтобы найти кратчайший маршрут из стартового города s в целевой город t . В качестве оценки стоимости остатка маршрута из города n до цели мы будем использовать расстояние по прямой расст(n, t) от n до t.. Таким образом, f(n)= g(n) + h(n) = g(n) + расст(n, t).

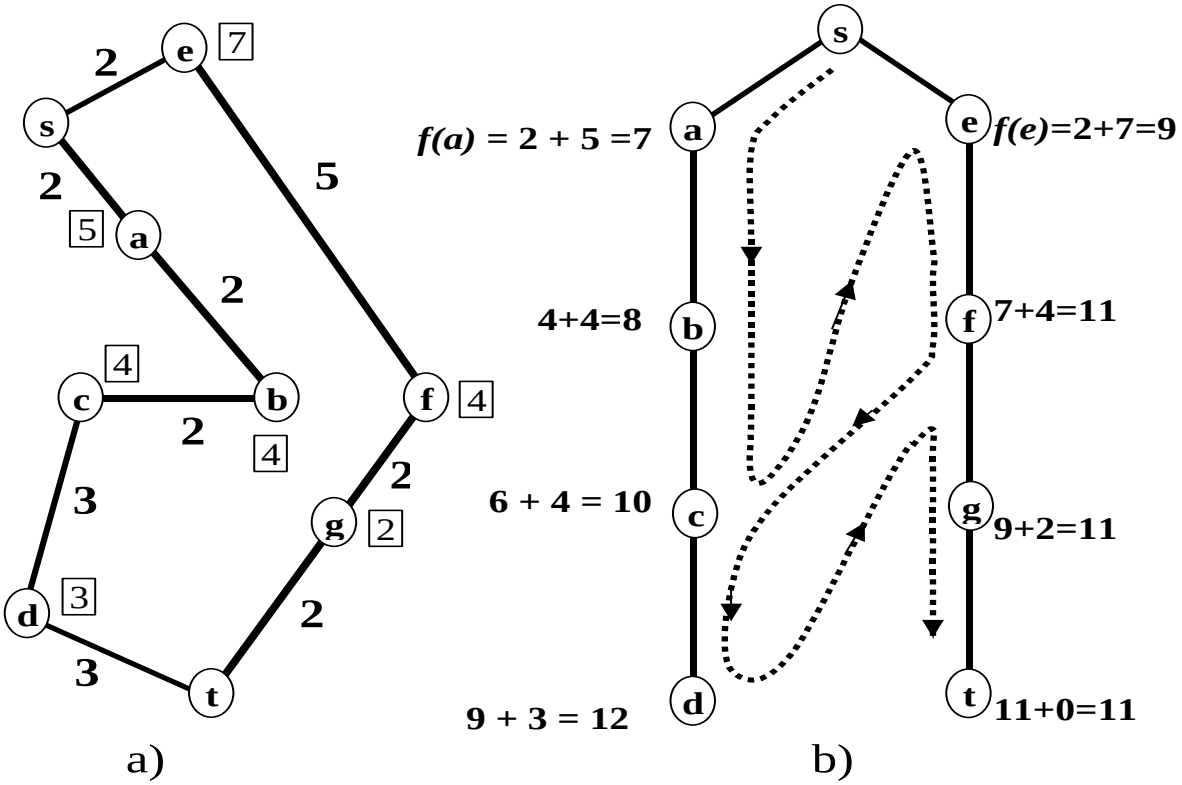


Рисунок 7 - Поиск кратчайшего маршрута из s в t

На рисунке 7(a) приведена карта со связями между городами. Связи помечены своими длинами; в квадратиках указаны расстояния по прямой до цели t. На рисунок 7(b) показан порядок, в котором при поиске с предпочтением происходит обход городов. Эвристические оценки основаны на расстояниях по прямой. Пунктирной линией показано переключение активности между альтернативными путями. Эта линия задает тот порядок, в

котором вершины принимаются для продолжения пути, а не тот порядок, в котором они порождаются.

Мы можем считать, что в данном примере процесс поиска с предпочтением состоит из двух процессов. Каждый процесс прокладывает свой путь – один из двух альтернативных путей: процесс_1 проходит через a, процесс_2 – через e. Вначале процесс_1 более активен, поскольку значения f вдоль выбранного им пути меньше, чем вдоль второго пути. Когда процесс_1 достигает города c , а процесс_2 все еще находится в e, ситуация меняется:

f(c) = g(c) + h(c) = 6 + 4 = 10

f(e) = g(e) + h(e) = 2 + 7 = 9

Поскольку f(e) < f(c), процесс_2 переходит к f, а процесс_1 ждет. Однако

f(f) = 7 + 4 = 11

f(c) = 10

f(c) < f(f)

Поэтому процесс_2 останавливается, а процессу_1 дается разрешение продолжать движение, но только до d, так как f(d)=12>11. Происходит активация процесса_2, после чего он, уже не прерываясь, доходит до цели t.

Замечания относительно поиска в дереве состояний

– Наиболее распространены две основные стратегии поиска в пространстве состояний: поиск в глубину и поиск в ширину. Поиск в глубину программируется наиболее легко, однако подвержен зацикливаниям. Существуют два простых метода предотвращения зацикливания: ограничить глубину поиска и не допускать дублирования вершин. Поиск в ширину в отличии от поиска в глубину всегда обнаруживает первым самое короткое решение.

– В случае обширных пространств состояний существует опасность комбинаторного взрыва. Обе стратегии плохо приспособлены для борьбы с этой трудностью. В таких случаях необходимо руководствоваться эвристиками.

– Эвристический принцип поиска с предпочтением направляет процесс поиска таким образом, что для продолжения поиска всегда выбирается вершина, наиболее перспективная с точки зрения эвристической оценки.

4 Выполнение работы

В этой лабораторной работе мы подробно рассмотрим технику, известную как поиск в пространстве состояний. Каждая конфигурация, в которой может пребывать система, называется состоянием. Из каждого состояния можно переместиться в некоторое множество других состояний. Одно состояние называется стартовым, другое – целевым. Задача состоит в том, чтобы найти последовательность перемещений, ведущих из стартового состояния в целевое. В лабораторной работе применение алгоритмов поиска в пространстве состояний мы рассмотрим на примере задачи поиска пути в лабиринте. Пусть, есть такой лабиринт:

#####
#**O**#**X**#
# #
#####

Позиция, помеченная как 'O' – стартовая позиция; позиция, помеченная 'X' – целевая. Знаком '#' изображаются непрозрачные стены. Мы можем на каждом шаге перемещаться ВВЕРХ, ВЛЕВО, ВПРАВО или ВНИЗ. Одно из правильных решений этого лабиринта можно задать следующей серией перемещений: ВНИЗ ВПРАВО ВПРАВО ВВЕРХ

Конечно, другим правильным решением может быть следующая серия перемещений:

ВНИЗ ВВЕРХ ВНИЗ ВВЕРХ ВНИЗ ВПРАВО ВЛЕВО ВПРАВО ВПРАВО ВВЕРХ ВНИЗ ВВЕРХ

Итак, каждое состояние описывается:

- расположением стен;
- стартовой позицией;
- целевой позицией.

После того как сделано перемещение, новое состояние подобно старому, но содержит другую стартовую позицию.

Прежде чем писать «решатель лабиринтов», мы напишем универсальный инструмент поиска (который ничего не знает о лабиринтах), и применим его к задаче поиска пути в лабиринте. Один из алгоритмов поиска пространства состояний называется поиском преимущественно в глубину (depth-firth search). По этому

алгоритму, когда из исходного состояния (s) получено новое (n), это новое состояние включается в поиск. Теперь мы пытаемся перевести систему из состояния n в состояние n1. Если состояние n1 ранее не было достигнуто, то производится его просмотр и поиск продолжается из n1. В противном случае выбирается другое достижимое из n состояние, и т. д. Например, предположим, что мы осуществляем поиск в следующем лабиринте:

```
######
##   #
#O #X#
##   #
######
```

Предположим, что когда мы ищем следующее состояние, мы пытаемся выполнять перемещения в следующем порядке: ВВЕРХ, ВПРАВО, ВНИЗ, ВЛЕВО. Поиск в глубину попытается перевести систему в новое состояние используя вначале перемещение ВВЕРХ. Это перемещение блокируется стеной. Следующим выбирается перемещение ВПРАВО. Получим такое состояние лабиринта:

```
######
##   #
# O#X#
##   #
######
```

Пытаемся переместиться ВВЕРХ. Получим следующее состояние:

```
######
##O  #
#  #X#
##   #
######
```

После нескольких применений перемещения ВПРАВО перемещение ВНИЗ приводит к достижению цели.

В противоположность этому поиску, поиск преимущественно в ширину (breadth-first search) сначала осуществляет поиск по всем узлам уровня k, а затем уже – по узлам уровня k+1. Поиск преимущественно в ширину для этого же лабиринта сначала

применит перемещение ВПРАВО. Получим:

```
######
##   #
# O#X#
##   #
######
```

Затем применим перемещение ВВЕРХ. Получим:

```
######
##O  #
#  #X#
##   #
######
```

Поскольку это не конечное состояние, возвращаемся назад и пытаемся применить перемещение ВНИЗ. Получим:

```
######
##   #
#  #X#
##O  #
######
```

Поскольку это не конечное состояние, возвращаемся назад и пытаемся применить перемещение ВПРАВО. Получим:

```
######
## O #
#  #X#
##   #
######
```

И т. д. Другими словами поиск в ширину просматривает все состояния, которые находятся на расстоянии k перемещений от начального, прежде, чем просматривает какое-либо состояние, которое находится на расстоянии k+1.

Функции поиска

Для перехода в следующее состояние используется функция поиска. Алгоритм поиска в дереве состояний следующий:

1. Если нет доступных состояний, мы не можем найти решение.
2. В противном случае, если исходное состояние целевое, то вернуть это состояние.

3. В противном случае переходим из исходного состояния в новое. Затем формируем новое множество состояний (включаем новое состояние в список состояний).
4. Новое состояние становится исходным.
5. Перейти к пункту 1.

Функция поиска по дереву – **tree-search** получает четыре параметра:

states – список состояний;

is-goal – предикат is-goal получает в качестве аргумента состояние. Если состояние целевое, is-goal возвращает не NIL. В противном случае, возвращает NIL.

expand-state – получает в качестве аргумента какое-то состояние. Возвращает список всех состояний, которые могут быть достигнуты из заданного состояния, за одно перемещение. (Список может быть пустым, если нет состояний, достижимых из заданного).

combine-states – функция combine-states получает два аргумента. Первый аргумент – список уже существующих состояний. Второй аргумент – список сгенерированных функцией expand-state состояний. Функция возвращает список состояний просто объединяя все состояния в обоих списках. Порядок, в котором состояния представлены в списке, это порядок, в котором состояния были получены.

Используя различные версии функций is-goal, expand-state, combine-states мы можем реализовать почти каждый поисковый алгоритм, использующий поиск по дереву.

Рекурсивная версия функции поиска по дереву.

```
(defun tree-search
      (states is-goal expand-state combine-states)
  (cond ((null states) nil)
       ((funcall is-goal (first states))
                          (first states))
       (t (tree-search
          (funcall combine-states (rest states)
           (funcall expand-state (first states)))
           is-goal
```

```
        expand-state
        combine-states ))))
```

Эта функция выполняет поиск по дереву, начиная с состояний заданных аргументом states, пока не будет достигнуто состояние, для которого предикат is-goal вернет значение не NIL. Функция expand-state используется для генерации новых состояний, достижимых из данного. Функция combine-states используется для объединения сгенерированных состояний с предыдущими. Поиск реализован рекурсивно.

Теперь мы можем использовать функцию поиска по дереву – **tree-search** для реализации алгоритмов поиска в глубину и в ширину.

Функция поиска в глубину – **depth-first-search** имеет следующий вид:

```
(defun depth-first-search
      (state is-goal expand-state tree-searcher)
  (funcall tree-searcher
        (list state) is-goal expand-state
        #'(lambda (old-states new-states)
            (append new-states old-states)))
)
```

Функция **depth-first-search** выполняет поиск в глубину, начиная с состояния state, до тех пор, пока не будет найдено целевое состояние. Функция expand-state используется для генерации дочерних состояний. Функция tree-searcher используется для выполнение алгоритма поиска по дереву.

Лабиринт

Лабиринт представим структурой. Структура для описания лабиринта должна иметь следующие поля: строку и столбец начальной позиции, и строку и столбец целевой позиции. В структуру входит также двумерный массив, который показывает расположение стен в лабиринте.

```
(defstruct maze  «Двумерный лабиринт»
  start-row    ; Строка начальной позиции.
  start-column ; Столбец начальной позиции.
  goal-row     ; Строка целевой позиции.
  goal-column  ; Столбец целевой позиции.
```

```
  grid        ; Двумерный массив элементы которого принимают
              ; значение не NIL, там где в лабиринте расположены
              ; стены, и NIL - на свободных местах. Первый индекс
              ; массива - индекс строк, второй - индекс столбцов.
              ; Нумерация начинается с нуля.
              ;Например, индекс элемента, расположенного во
              ; второй строке и третьем столбце - (1, 2).
)
```

Мы можем представить состояние как список из двух элементов. Первый элемент списка – список уже посещенных позиций на пути из стартовой позиции к текущей. Второй элемент списка – лабиринт, который представлен, как описано выше.

Функция печати лабиринта **maze-print** получает два аргумента: лабиринт и список пройденных позиций:

```
(defun maze-print (maze positions)
...
)
```

Для печати переданного лабиринта функция использует несколько символов:

– символ '#' используется для изображения стен;

– символ 'O' используется для изображения стартовой позиции;

– символ 'X' используется для изображения целевой позиции.

Перед лабиринтом не печатается пустая строка. Пустая строка печатается в конце каждой строки лабиринта, включая последнюю. Список positions содержит пройденные позиции (найденный в лабиринте путь). Они изображаются символом '.' .

Функция **maze-is-goal** возвращает не NIL, если указанный лабиринт решен, т. е., если стартовая и целевая позиции одинаковы.

```
(defun maze-is-goal (maze-state)
  (let ((maze (second maze-state)))
   (and
     (= (maze-start-row maze) (maze-goal-row maze))
     (= (maze-start-column maze)
```

```
                (maze-goal-column maze))))
    )
```

Функция **maze-expand** возвращает список состояний лабиринта, достижимых за одно перемещение. Состояние достижимо, если новая стартовая позиция получается одним перемещением влево, вправо, вверх или вниз относительно начальной стартовой позиции и координаты этой позиции не совпадают с координатами стены.

```
    (defun maze-expand (maze-state)
      (let ((maze (second maze-state)))
       (labels ((is-empty (row column)
                          ; Возвращает не-NIL если лабиринт не содержит стену
                          ; в указанной позиции (row column)
          (not (aref (maze-grid maze) row column)))
          (new-maze-if-legal (row column)
                          ; Возвращает состояние лабиринта, соответствующее
                          ; состоянию, достижимому из состояния maze-state
                          ; путем перемещения в новую строку и столбец, или
                          ; NIL, если премещение не допустимо
          (and (array-in-bounds-p (maze-grid maze)
                                  row column)
             (is-empty row column)
             (not (member (list row column)
                          (first maze-state)
                          :test #'equal))
             (list
                          ;Новая позиция добавляется в начало списка
                          ;позиций
               (cons (list row column)
                  (first maze-state))
```

```
                                ; Новый лабиринт такой же как и старый,
                                ; различны только стартовые позиции
            (make-maze :start-row    row
                       :start-column column
                       :goal-row
                       (maze-goal-row maze)
                       :goal-column
                       (maze-goal-column maze)
                       :grid
                       (maze-grid maze))))))
        (let ((row (maze-start-row maze))
              (column (maze-start-column maze)))
          (remove nil
                          ; Формируем список достижимых состояний.
                          ; Некоторые элементы могут быть равны NIL,
                          ; если попытки перемещения закончились неудачей
                  (mapcar #'new-maze-if-legal
                          (list (1- row)
                                (1+ row)
                                row
                                row)
                          (list column
                                column
                                (1- column)
                                (1+ column)))))

    )
```

Примечание: функция labels эквивалентна функции let, но работает с функциями, а не с переменными. Например:

```
    (labels ((factorial (n)
               (if (= n 0) 1
                 (* n (factorial (- n 1)))) ))
      (factorial 3) )
```

Функция **maze-solve** решает заданный лабиринт, печатает решение. Если решения не существует, печатает соответствующее

сообщение. Эту функцию можно определить следующим образом:

```
(defun maze-solve
          (maze search-strategy tree-searcher)
  (format t «Attempting to solve the following
        maze:~%»)
  (maze-print maze nil)
  (terpri)
  (let ((solution (funcall search-strategy
                (list
                 (list
                  (list (maze-start-row maze)
                      (maze-start-column
                            maze))) maze)
                #'maze-is-goal
                #'maze-expand
                tree-searcher)))
    (if solution
       (progn
        (format t «Solution:~%»)
        (maze-print (second solution)
                 (first solution))
    )
    (format t «No solution exists.~%»)) solution))
```

Ниже приведено несколько примеров использования выше перечисленных функций.

Примеры.

```
>(maze-solve complex-maze #'breadth-first-search
                    #'iterative-tree-search)
Attempting to solve the following maze:
##########
#  #     #
## # ### #
#    # # #
# ### #   #
# ## #####
# #      0
```

```
#   ## ###
# # ##   #
X ########

Solution:
##########
#  #     #
## # ### #
#    # # #
# ## #   #
# ## #####
# #.......
#...## ###
#.# ##   #
..########
(((9 0) (9 1) (8 1) (7 1) (7 2) (7 3) (6 3) (6 4)
  (6 5) (6 6) (6 7) (6 8) (6 9))
#S(MAZE START-ROW 9 START-COLUMN 0 GOAL-ROW 9
       GOAL-COLUMN 0 GRID
        #2A((T T T T T T T T T T)
           (T NIL NIL T NIL NIL NIL NIL NIL T)
           (T T NIL T NIL T T T NIL T)
           (T NIL NIL NIL NIL T NIL T NIL T)
           (T NIL T T NIL T NIL NIL NIL T)
           (T NIL T T NIL T T T T T)
           (T NIL T NIL NIL NIL NIL NIL NIL NIL)
           (T NIL NIL NIL T T NIL T T T)
           (T NIL T NIL T T NIL NIL NIL T)
           (NIL NIL T T T T T T T T))))
```

Ниже представлено несколько лабиринтов, которые Вы можете использовать для тестирования программы:

```
(defconstant simple-maze
  (make-maze :start-row    0
             :start-column 0
             :goal-row     2
             :goal-column  2
```

```
        :grid #2A((nil nil   t)
                  (nil   t   t)
                  (nil nil nil)))
    «A very simple maze.»)

(defconstant complex-maze
    (make-maze :start-row    6
               :start-column 9
               :goal-row     9
               :goal-column  0
               :grid #2A(
          (t   t   t   t   t   t   t   t   t   t)
          (t   nil nil t   nil nil nil nil nil t)
          (t   t   nil t   nil t   t   t   nil t)
          (t   nil nil nil nil t   nil t   nil t)
          (t   nil t   t   nil t   nil nil nil t)
          (t   nil t   t   nil t   t   t   t   t)
          (t   nil t   nil nil nil nil nil nil nil)
          (t   nil nil nil t   t   nil t   t   t)
          (t   nil t   nil t   t   nil nil nil t)
          (nil nil t   t   t   t   t   t   t   t)))
    «A more complex maze.»)
```

5 Задание на лабораторную работу

1. Реализовать итерационную версию функции поиска tree-search.
2. Реализуйте функцию поиска в ширину в пространстве состояний (breadth-first-search).
3. Реализуйте функцию, печатающую лабиринт (maze-print).
4. Определите свой лабиринт my-maze (20x20). Оцените сколько времени потребуется для решения каждого из приведенных лабиринтов. (Для этого вы можете использовать функцию time. (time expr) возвращает значение выражения, но также печатает время ушедшее на его вычисление).

6 Контрольные вопросы

1. Когда лучше использовать поиск в ширину, а когда в глубину. (Под лучше понимается находить решения быстрее).
2. Какие параметры (search-strategy и tree-searcher) Вы передадите функции maze-solve? Почему?
3. Поскольку не каждый лабиринт имеет решение, что вы сделаете, чтобы быть уверенным в том, что поиск завершится в любом случае?

Список литературы

1. Медведев В. С. Нейронные сети / В. С. Медведев, В. Г. Потемкин. – М.: Диалог МИФИ, 2002.

2. Дьяконов В. П. MATLAB 5.3.1 с пакетами расширений / В. П. Дьяконов, И. В. Абраменкова, В. В. Круглов. – М: Нолидж,2001.

3. Комашинский В. И. Нейронные сети и их применение в системах управления и связи / В. И. Комашинский, Д. А. Смирнов. – М.: Горячая линия – Телеком, 2002.