Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

Кафедра компьютерных технологий

Д. И. Елкин, П. А. Скорынин

Применение алгоритма имитации отжига для построения автомата управления виртуальным роботом-футболистом

Санкт-Петербург
2009

Оглавление

Введение

В данной работе рассматривается алгоритм имитации отжига и его применение для генерации конечных автоматов при построении модели управления виртуальным роботом-футболистом. Для исследования была использована программа *OPF Virtual Football*, разработанная студентами кафедры компьютерных технологий СПбГУ ИТМО А. А. Кошевым и М. Н. Царевым. Эта программа является удобной платформой, позволяющей визуализировать и сравнивать сгенерированные различными методами автоматы, управляющие роботами-футболистами.

Целью работы является построение конечного автомата с помощью алгоритма имитации отжига и сравнение его с автоматом, построенным генетическим алгоритмом.

1. Описание программы *OPF Virtual Football*

Приложение *OPF Virtual Football* разработано на языке программирования *Java*. После запуска программы отображается окно формирования команд для игры в виртуальный футбол (рис. 1). В правой части представлен список всех доступных роботов. С помощью соответствующих кнопок можно изменять составы команд, которые отображаются в левой части окна. *OPF Virtual Football* предоставляет возможность добавлять в одну команду роботов разных типов.

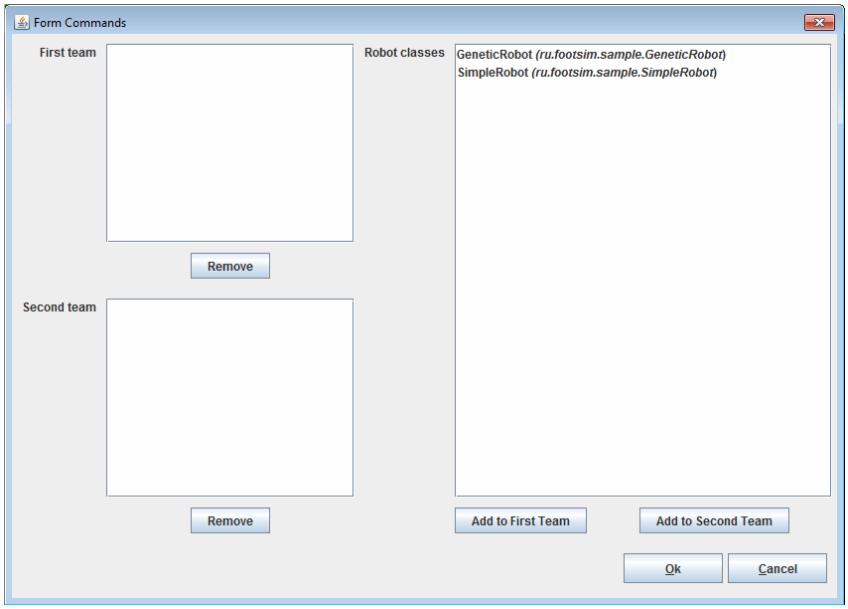

Рис. 1. Окно выбора команд

После того, как составы команд сформированы, открывается окно визуализации (рис. 2). В этом окне отображается ход игры.

Рис. 2. Окно визуализации

Авторами приложения *OPF Virtual Foorball* были разработаны два робота-футболиста:

- *SimpleRobot* – робот с тривиальной логикой. В каждый момент времени робот движется в сторону мяча, пытаясь направить его в чужие ворота;
- *GeneticRobot* – робот, управляемый автоматом, построенным с помощью генетических алгоритмов.

Для краткости робота, управляемого автоматом, построенным с помощью алгоритма имитации отжига, в дальнейшем будем называть *AnnealRobot*. Исходный код программы, реализующей управление этим роботом на основе построенного автомата, приведен в приложении 1.

Помимо режима визуализации, описанного выше, *OPF Virtual Football* может работать в режиме построения автомата (этот режим актуален только для роботов, управляемых автоматом). В данном режиме не производится графическое отображение хода игры, вместо этого вспомогательная информация выводится на консоль. За счет отсутствия визуализации каждый матч проходит значительно быстрее. В приложении 1 приведен исходный код программы на языке *Java*, реализующей генерацию управляющего автомата методом имитации отжига.

2. Алгоритм имитации отжига

Алгоритм имитации отжига [1–4] основан на моделировании физического процесса, который происходит при кристаллизации вещества из жидкого состояния в твердое (например, при отжиге металла). В этой модели предполагается, что, во-первых, отжиг происходит при понижении температуры, а во-вторых, атомы в веществе уже выстроились в кристаллическую решетку, но отдельные атомы еще могут перейти из одной ячейки в другую. Вероятность таких переходов, в свою очередь, зависит от температуры. Устойчивая кристаллическая структура вещества соответствует минимуму энергии.

Формализуем процесс отжига. Фактически, при моделировании ищется точка (или набор точек), на которой достигается минимум некоторой числовой функции $E(\vec{x})$. Строится последовательность точек $\vec{x}_0, \vec{x}_1, ..., \vec{x}_n$, где $\vec{x}_0$ соответствует начальному значению. При достижении точки $\vec{x}_n$ алгоритм завершает свою работу. Пусть рассматривается текущая точка $\vec{x}_i$. К ней применяется некоторый оператор A_i, который зависит от температуры. После модификации получаем точку $\vec{x}^*$. Вероятность выбора $\vec{x}^*$ в качестве следующей точки $\vec{x}_{i+1}$ равна $P(\vec{x}^*, \vec{x}_i)$, где P – распределение Гиббса:

$$P(\vec{x}^*, \vec{x}_i) = \begin{Bmatrix} 1, & E(\vec{x}^*) - E(\vec{x}_i) < 0 \\ \exp\left(-\dfrac{E(\vec{x}^*) - E(\vec{x}_i)}{Q_i}\right), & E(\vec{x}^*) - E(\vec{x}_i) \geq 0 \end{Bmatrix}, \qquad (1)$$

где $Q_i > 0$ – элементы произвольной убывающей сходящейся к нулю последовательности. Эта последовательность является аналогом понижающейся температуры в реальном физическом процессе.

Весь алгоритм можно разбить на шесть этапов (рис. 3):

1. Создание начального решения.
2. Оценка решения.
3. Изменение решения случайным образом.
4. Оценка нового решения.
5. Выбор нового решения в качестве рабочего, если оно прошло критерий допуска.
6. Уменьшение температуры.

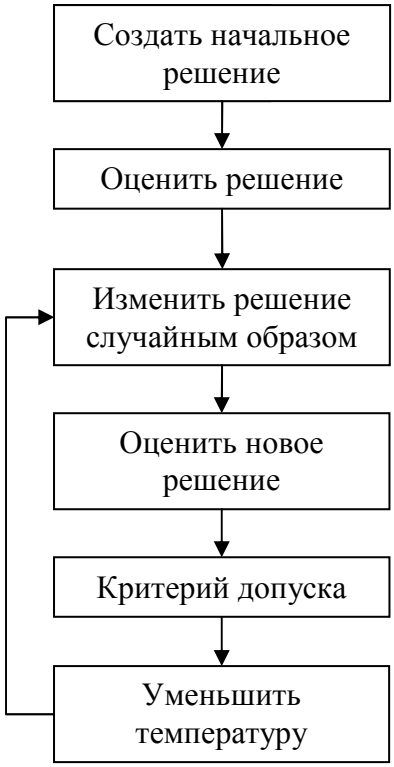

Рис. 3. Схема алгоритма имитации отжига

Рассмотрим подробнее каждый из этапов. Будем считать, что решение представляет собой битовую строку фиксированной длины. При этом любая битовая строка этой длины является решением.

Создание начального решения. На этом этапе выбирается некоторая начальная точка $\vec{x}_0$, которую принимают в качестве рабочего решения. Этой точкой может быть уже существующее решение (например, полученное другими алгоритмами) или сгенерированное случайным образом решение. В данной работе начальное решение генерируется случайным образом.

Оценка начального решения. Данный этап необходим для того, чтобы в дальнейшем сравнивать новые решения с рабочим. Для оценки решения используется функция энергии $E(\vec{x})$. Более подробно об этой функции речь пойдет в разд. 3.5.

Построение нового решения. На этом этапе рабочее решение изменяется случайным образом. Для этого задается порождающая функция $\gamma(T,\vec{x})$. В данной работе каждый бит рабочего решения может изменяться с вероятностью $\frac{T}{T_{\max}}$.

Оценка нового решения. После построения решения его следует оценить для того, чтобы далее сравнить с рабочим. Для этих целей используется та же функция, что и во втором этапе.

Выбор нового рабочего решения. На предыдущем этапе была получена оценка нового решения $E(\vec{x}^*)$. По формуле (1) находим вероятность принятия нового решения в качестве рабочего. Таким образом, в качестве $\vec{x}_{i+1}$ выбирается $\vec{x}^*$ или $\vec{x}_i$.

Уменьшение температуры. Перед тем, как перейти к новому шагу алгоритма, нужно уменьшить температуру по некоторому закону $T(k)$, где k – номер шага. В данной работе функция $T(k)$ задана рекурсивно:

$$T(k) = \alpha T(k-1) \ , \qquad (2)$$

где $\alpha = \sqrt[N]{\frac{T_{\min}}{T_{\max}}}$, $T(0) = T_{\max}$, N – число итераций.

3. Математическая модель виртуального робота-футболиста

3.1. Автоматное представление

Для управления роботом-футболистом был использован конечный автомат Мили из восьми состояний. Выходные действия такого автомата генерируются в зависимости от текущего состояния и входного воздействия. На рис. 4 в качестве примера приведен автомат Мили с тремя состояниями. Каждое входное воздействие в этом автомате – числитель, а каждое выходное – знаменатель дроби, помечающей соответствующую дугу графа.

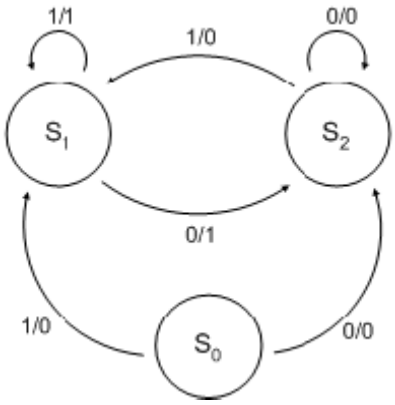

Рис. 4. Пример диаграммы переходов автомата Мили из трех состояний

3.2. Входные и выходные воздействия конечного автомата

Во многом представление входных и выходных воздействий влияет на эффективность применения метода отжига для нахождения оптимальной модели управления. Если выбрать сложное представление входных данных, то эффективная модель может быть получена за очень большое время или не получена вовсе. Если же выбрать простое представление, то полученная оптимальная модель может оказаться слишком примитивной. В ходе экспериментов было найдено представление, которое позволило добиваться хороших результатов за приемлемое время. Во входных данных кодировалась следующая информация:

- знак векторного произведения скорости игрока на направление соперника относительно положения игрока;
- знак векторного произведения скорости игрока на направление мяча относительно положения игрока;
- знак векторного произведения скорости игрока на направление ворот игрока относительно положения игрока;
- знак векторного произведения скорости игрока на направление ворот соперника относительно положения игрока.

Как видно, в этих *четырех входных переменных* используется только относительное положение объектов. Эта модель соответствует действительности: игрок не может видеть все поле, но может оценить положение относительно себя других игроков, мяча и ворот.

Следующим важным фактором являются выходные воздействия. Для эффективного сравнения автоматов, построенных методом, описанным в данной работе, и методом генетического программирования, рассмотренным в бакалаврской работе А. А. Кошевого

(кафедра компьютерные технологии СПбГУ ИТМО), выходные воздействия должны совпадать. Они приняты следующими:

- скорость футболиста всегда равна трем;
- в зависимости от принятого решения, футболист либо поворачивает направо, либо налево на 1°.

3.3. Деревья решений

Каждое состояние автомата представляется деревом решений [5], в листьях которого хранятся переходы (пара {воздействие, номер нового состояния}), а во внутренних вершинах – номер входной переменной. Пример дерева решений представлен на рис. 5. Из рисунка видно, что на каждом из путей от корня к листьям используются не все входные переменные.

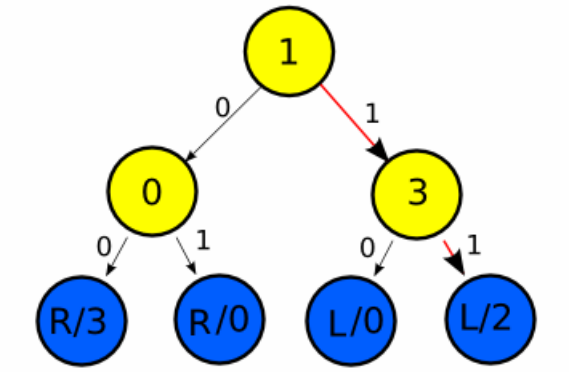

Рис. 5. Дерево решений

Для определения перехода автомата, моделирующего поведение футболиста, по значениям четырех входных переменных необходимо, начиная с корня дерева, переходить в левое или правое поддерево, пока не будет достигнут лист. При этом переход осуществляется по тому ребру, которое помечено значением переменной, соответствующей рассматриваемой вершине.

В данной работе значения входных переменных представляются в виде строки из четырех бит.

3.4. Кодирование состояний конечного автомата

Как отмечено в разд. 3.3, каждое состояние представлено деревом решений. Для того чтобы каждое состояние можно было кодировать битовой строкой фиксированной длины (это необходимо для алгоритма имитации отжига), строилось дерево решений. При этом возможна ситуация, при которой на пути от корня к какому-либо листу *одна и та же переменная встречается более одного раза*. Однако это не влияет на работоспособность алгоритма в целом.

Пусть каждое состояние представлено деревом решений глубины четыре (выбрано как степень двух). Каждая внутренняя вершина, помеченная номером входной переменной, кодируется двумя битами. Листья кодируются четырьмя битами (три бита на номер нового состояния и еще один – на выходное воздействие). При этом длина строки, представляющей состояние автомата, составляет $(2^4 - 1) \cdot 2 + 2^4 \cdot 4 = 94$ бита. Это составляет 12 байт.

3.5. Оценка полученных автоматов

Для работы метода имитации отжига необходимо задать функцию $E(\vec{x})$, которая бы позволяла оценивать эффективность полученного решения. При этом, чем меньше значение этой функции, тем эффективнее решение. В данной работе в качестве эталонного робота использован *SimpleRobot*, описанный в главе 1. Проводилось $n = 20$ игр между разработанным авторами роботом *AnnealRobot* и известным роботом *SimpleRobot*. По результатам этих игр находилось значение функции энергии $E(\vec{x})$ для робота *AnnealRobot*.

В каждой игре подсчитывалось число мячей, забитых *SimpleRobot'ом.* Далее отбрасывались максимальное и минимальное значения и находилось среднее арифметическое. Полученное значение выбиралось в качестве $E(\vec{x})$:

$$E(\vec{x}) = \frac{\sum_{j=0}^{20} B_j(\vec{x}) - B_{\max}(\vec{x}) - B_{\min}(\vec{x})}{18}.$$

Как можно заметить, при таком задании $E(\vec{x})$ должен получиться робот-вратарь или робот-защитник, так как при подсчете учитывались только число пропущенных мячей. И, действительно, получившийся робот в некоторых ситуациях действовал как защитник: отбегал к воротам и перекрывал к ним доступ для соперника. Однако при этом, когда мячом «владел» разработанный авторами робот *AnnealRobot*, он стремился отправить мяч в чужие ворота, активно атакуя.

4. Построение конечного автомата

При генерации автомата были выбраны следующие параметры алгоритма имитации отжига:

- начальная температура – 100;
- конечная температура – 0.05;
- число итераций – 100.

На каждой итерации алгоритма имитации отжига полученное решение проверялось в течение 20 игр (разд. 3.5), каждая игра состояла из 100 000 итераций.

На хранение состояния автомата выделяется 12 байт (разд. 3.4). Число состояний автомата было выбрано равным восьми. Таким образом, для хранения одного автомата требуется 96 байт.

В ходе генерации автомата было видно, что в первых поколениях часто принималось решение о выборе нового робота в качестве текущего решения, а с уменьшением температуры текущее решение изменялось все реже.

В 79-ом поколении было построено решение, функция энергии $E(\vec{x})$ которого оказалась минимальной. Деревья решений для каждого из состояний автомата, соответствующего этому решению, приведены в приложении 2. На рис. 6 приведен график зависимости энергии рабочего решения от номера поколения.

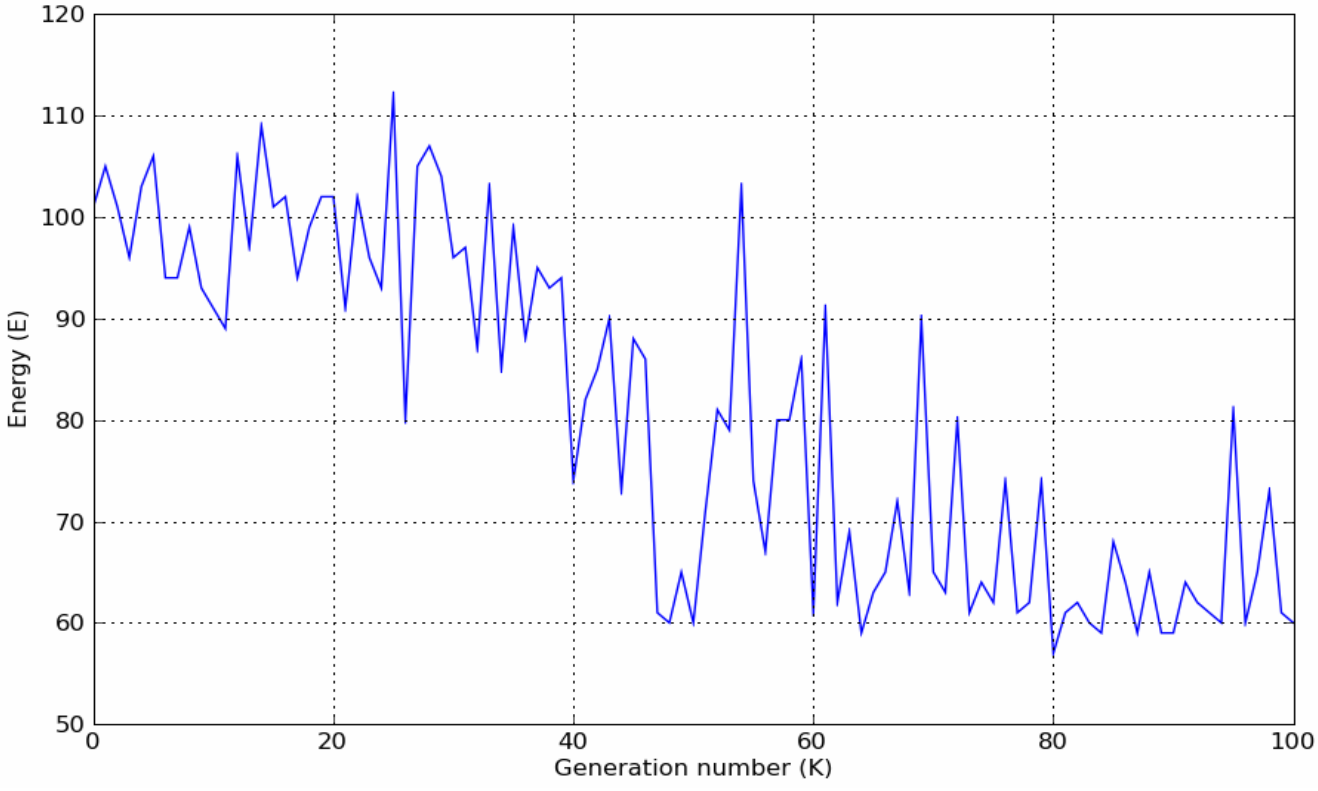

Рис. 6. График зависимости энергии решения от номера поколения

5. Сравнение автоматов, построенных различными методами

Робот, управляемый автоматом, построенным с помощью алгоритма имитации отжига, сравнивался с роботом *GeneticRobot*. Основным критерием сравнения является итоговый счет матча. Для наглядности рассмотрим график зависимости разницы в счете (число мячей, которые забил *AnnealRobot* минус число пропущенных мячей) от номера игры (рис. 7). Из графика видно, что результаты *AnnealRobot* превосходят результаты *GeneticRobot*.

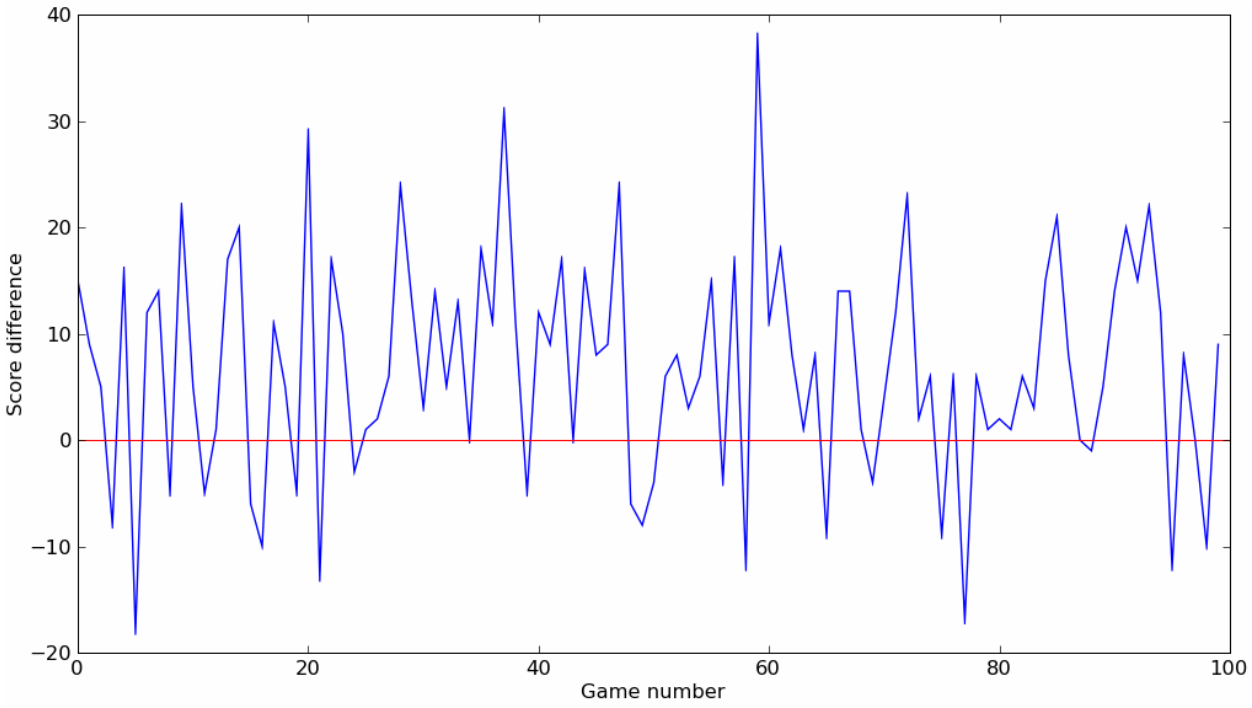

Рис. 7. Результаты игр *AnnealRobot* против *GeneticRobot*

Помимо сравнения результатов матчей было выполнено сравнение входных и выходных воздействий роботов. Информация, кодирующаяся в состоянии *AnnealRobot*, уже была описана в разд. 3.2. На вход *GeneticRobot* подается информация об абсолютной позиции игрока и мяча, скорости игрока и мяча, а также о векторе направления мяча относительно игрока. Выходные воздействия *GeneticRobot* совпадают с выходными воздействиями *AnnealRobot*.

Стоит отметить, что для построения управляющего автомата методом имитации отжига потребовалось намного меньше времени, чем для построения генетическим алгоритмом. Это объясняется тем, что в алгоритме имитации отжига на каждой итерации рассматривается не более двух решений (рабочее и новое), в то время как в генетическом алгоритме в каждом поколении рассматриваются сотни особей. Очевидно, что это приводит к ощутимой разнице во времени построения решения.

Заключение

В данной работе было предложено применение алгоритма имитации отжига для генерации конечных автоматов, управляющих виртуальным роботом-футболистом. В ходе работы был получен *AnnealRobot*. При сравнении его с *GeneticRobot*, разработанным авторами программы *OPF Virtual Football,* было установлено, что *AnnealRobot* играет лучше, чем *GeneticRobot*, а алгоритм имитации отжига находит оптимальное решение быстрее, чем генетический алгоритм, и для своей работы требует меньше памяти.

Таким образом, был сделан вывод, что метод имитации отжига больше подходит для генерации управляющего роботом-футболистом автомата, чем генетический алгоритм.

Источники

1. *Джонс М. Т.* Программирование искусственного интеллекта в приложениях. М.: ДМК-Пресс, 2004.
2. *Елкин Д. И., Тяхти А. С.* Искусственный интеллект. Алгоритм имитации отжига. 2008. http://rain.ifmo.ru/cat/view.php/theory/unsorted/ai-annealing-2008/article.pdf
3. *Лопатин А. С.* Метод отжига в задачах оптимизации. 2004. http://www.math.spbu.ru/user/gran/students/cothesis.pdf
4. *Ясницкий Л. Н.* Введение в искусственный интеллект. М.: Академия, 2005.
5. *Данилов В. Р., Шалыто А. А.* Метод генетического программирования для генерации автоматов, представленных деревьями решений. http://is.ifmo.ru/download/2008-03-07-danilov.pdf

Приложение 1. Исходные тексты программ

AnnealRobot.java. В этом файле реализован робот, управляемый автоматом, построенным с помощью алгоритма имитации отжига.

```
    package ru.footsim.anneal.robot;

    import ru.footsim.math.Vector;
    import ru.footsim.model.robots.Robot;
    import ru.footsim.model.robots.events.StatusEvent;
    import ru.footsim.model.game.info.FieldInfo;
    import ru.footsim.model.game.info.PlayerInfo;
    import ru.footsim.model.game.info.BallInfo;

    import java.util.Arrays;
    import java.util.Random;

    public class AnnealRobot extends Robot {

        private class AnnealAutomaton {
            private static final int stateCount = 8;
            private static final int DTDepth = 4;
            private static final int VarCount = 4;
            private static final int ActionSize = 4; // bits,
stateIndex included
            private final int VarCountSize =
(int)Math.ceil(Math.log(VarCount) / Math.log(2.0));
            private final int DTSize =
(int)Math.ceil(((double)VarCountSize
                    * ((1 << DTDepth) - 1) + (1 << DTDepth) *
ActionSize) / 8);      // bytes
            int curState;
            byte[][] DT;

            AnnealAutomaton() {
                DT = new byte[stateCount][DTSize];
                Random r = new Random();
                for(int i = 0; i < stateCount; ++i)
                    r.nextBytes(DT[i]);
                curState = 0;
            }

            AnnealAutomaton(String automaton) {
                DT = new byte[stateCount][DTSize];
                for(int i = 0; i < stateCount; ++i) {
                    for(int j = 0; j < DTSize; ++j) {
                        DT[i][j] = 0;
                        for(int k = 0; k < 8; ++k) {
                            DT[i][j] |= (automaton.charAt(k + j * 8 +
i * DTSize * 8) - '0')<<k;
                        }
                    }
                }
                curState = 0;
            }
```

```
            AnnealAutomaton(byte[] automaton) {
                DT = new byte[stateCount][DTSize];
                for(int i = 0; i < stateCount; ++i) {
                    System.arraycopy(automaton, i * DTSize, DT[i], 0,
DTSize);
                }
                curState = 0;
            }

            byte getAction(byte[] input) {
                int cur = 1;
                for(int i = 0; i < DTDepth; ++i) {
                    int a = (cur - 1) * VarCountSize + 1;
                    int b = a + VarCountSize - 1;
                    int index = 0;
                    for(int j = a; j <= b; ++j) {
                        index <<= 1;
                        index |= (DT[curState][j >> 3] & (1 << (j %
8))) >> (j % 8);
                    }
                    cur <<= 1;
                    if((input[index >> 3] & (1 << (index % 8))) == 0)
{
                        ++cur;
                    }
                }
                byte val = 0;
                int a = VarCountSize * ((1 << DTDepth) - 1) + (cur -
(1 << DTDepth)) * ActionSize + 1;
                int b = a + ActionSize - 1;
                for(int j = a; j <= b; ++j) {
                    val <<= 1;
                    val |= (DT[curState][j >> 3] & (1 << (j % 8))) >>
(j % 8);
                }
                curState = val & (stateCount - 1);
                return val;
            }

            byte[] getByteArray() {
                byte[] result = new byte[stateCount * DTSize];
                for(int i = 0; i < stateCount; ++i) {
                    for(int j = 0; j != DTSize; ++j) {
                        result[i * DTSize + j] = DT[i][j];
                    }
                }
                return result;
            }
        }
        private AnnealAutomaton automaton;

        public AnnealRobot() {
            super("AnnealRobot");
            this.automaton = new AnnealAutomaton();
        }

        public AnnealRobot(byte[] automaton) {
            super("AnnealRobot");
```

```
            this.automaton = new AnnealAutomaton(automaton);
        }

        public AnnealRobot(String automaton) {
            super("AnnealRobot");
            this.automaton = new AnnealAutomaton(automaton);
        }

        public byte[] getAutomaton() {
            return automaton.getByteArray();
        }

        private int addVal(byte[] input, int cur_bit, double val,
double max_val, int bits) {
            if(val > max_val) val = max_val;
            int val_ = (int)(((1<<bits) - 1) * val / max_val);
            for(int i = 0; i != bits; ++i, ++cur_bit)
                input[cur_bit >> 3] |= ((val_ & (1 << i)) >> i) <<
(cur_bit % 8);
            return cur_bit;
        }

        @Override
        public void onStatus(StatusEvent event) {
            byte[] input = new byte[(AnnealAutomaton.VarCount + 7) /
8];
            Arrays.fill(input, (byte) 0);
            int cur_bit = 0;
            FieldInfo field = event.getFieldInfo();
            PlayerInfo player = event.getMyInfo();
            PlayerInfo opponent = event.getOthersInfo().get(1);
            BallInfo ball = event.getBallInfo();
            Vector dir =
ball.getPosition().subtract(player.getPosition());
            Vector left = new Vector(0, field.getHeight() /
2.0).subtract(player.getPosition());
            Vector right = new Vector(field.getWidth(),
field.getHeight() / 2.0).subtract(player.getPosition());
            cur_bit = addVal(input, cur_bit,
opponent.getPosition().subtract(player.getPosition()).vectorProductDir
(player.getSpeed()) > 0.0 ? 1.0 : 0.0, 1.0, 1);
            cur_bit = addVal(input, cur_bit,
dir.vectorProductDir(player.getSpeed()) > 0.0 ? 1.0 : 0.0, 1.0, 1);
            cur_bit = addVal(input, cur_bit,
left.vectorProductDir(player.getSpeed()) > 0.0 ? 1.0 : 0.0, 1.0, 1);
            cur_bit = addVal(input, cur_bit,
right.vectorProductDir(player.getSpeed()) > 0.0 ? 1.0 : 0.0, 1.0, 1);

            byte action = automaton.getAction(input);

            setSpeed(3.0);
            if((action & 8) != 0)
                setTurnAngleLeft(1.0);
            else
                setTurnAngleRight(1.0);
        }

        @Override
```

```
        public String toString() {
            StringBuilder sb = new StringBuilder();
            byte[] array = automaton.getByteArray();
            for (byte anArray : array) {
                for (int j = 0; j < 8; ++j) {
                    sb.append((anArray & (1 << j)) >> j);
                }
            }
            return sb.toString();
        }
    }
```

AnnealMain.java. В этом файле реализован алгоритм имитации отжига для генерации управляющего автомата.

```
    package ru.footsim.main;

    import java.awt.Color;
    import java.awt.Rectangle;
    import java.io.*;
    import java.util.Random;
    import java.util.Arrays;

    import ru.footsim.anneal.robot.AnnealRobot;
    import ru.footsim.math.Vector;
    import ru.footsim.model.game.TeamNumber;
    import ru.footsim.model.game.impl.PlainGame;
    import ru.footsim.model.game.settings.GameSettings;
    import ru.footsim.model.robots.Robot;
    import ru.footsim.model.robots.events.StatusEvent;

    public class AnnealMain implements Runnable {

        private final static String CONFIG_LOCATION =
"config/conf.xml";

        private final static GameSettings DEFAULT_SETTINGS = new
GameSettings(1, 1,
                new Rectangle(110, 80), 20.0, 30, 4.0, 5.0, 2.5, 1.0,
                60 * 60 * 1000, Color.GREEN, Color.RED);

        private final static int SMALL_ITERATIONS_COUNT = 100000;

        private final static int INTERNAL_ITERATIONS_COUNT = 1;

        private final static int EXTERNAL_ITERATIONS_COUNT = 100;

        private final static int GAMES_COUNT               = 20;

        private final double MAX_T = 100.0;

        private final double MIN_T = 0.05;

        private GameSettings settings;

        private AnnealRobot robot;

        public AnnealMain() {
```

```
            initSettings();
        }

        public void run() {
            final OutputStream os;
            final OutputStream sout;
            try {
                os = new FileOutputStream("out.txt");
                sout = System.out;//new FileOutputStream("sout.txt");
            } catch (FileNotFoundException e) {
                throw new RuntimeException(e);
            }
            final PrintStream ps = new PrintStream(os);
            final PrintStream psout = new PrintStream(sout);

            annealProcess(ps, psout);
        }

        private void initSettings() {
            final InputStream is = AnnealMain.class
                    .getResourceAsStream(CONFIG_LOCATION);
            if (is == null) {
                settings = DEFAULT_SETTINGS;
            } else {
                try {
                    settings = Utils.loadGameSettings(is);
                } catch (IOException e) {
                    settings = DEFAULT_SETTINGS;
                    e.printStackTrace();
                } catch (RuntimeException e) {
                    settings = DEFAULT_SETTINGS;
                    e.printStackTrace();
                } finally {
                    try {
                        is.close();
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }
        }

        private void annealProcess(final PrintStream ps, final
PrintStream psout) {
            psout.println("[Initializing first generation]");
            firstGeneration();
            psout.println("[First generation has been initialized]");
            psout.println();
            Random r = new Random();
            int generation = 0;
            double T = MAX_T;
            double alpha = Math.pow(MIN_T / MAX_T,
1.0/EXTERNAL_ITERATIONS_COUNT);
            double E = getEnergy(robot, psout);
            for(int i = 0; i != EXTERNAL_ITERATIONS_COUNT; ++i) {
                for(int j = 0; j != INTERNAL_ITERATIONS_COUNT; ++j) {
                    psout.println("Generation #" + generation++);
                    AnnealRobot newRobot = nextGeneration(T, r);
```

```
                    ps.println(newRobot.toString());
                    double newE = getEnergy(newRobot, psout);
                    if(newE < E || r.nextDouble() < Math.exp((E -
newE) * 10.0 / T)) {
                        E = newE;
                        robot = newRobot;
                        psout.println("New robot selected!");
                    }
                    psout.println();
                }
                T *= alpha;
            }
            ps.println(robot.toString());
            psout.print(robot.toString());
        }

        private void firstGeneration() {
            robot = new AnnealRobot();
        }

        private AnnealRobot nextGeneration(double T, Random r) {
            byte[] automaton = robot.getAutomaton();
            for(int i = 0; i < automaton.length; ++i) {
                for(int j = 0; j < 8; ++j) {
                    double p = r.nextDouble();
                    if(p < T / MAX_T)
                        automaton[i] ^= 1 << j;
                }
            }
            return new AnnealRobot(automaton);
        }

        private double getEnergy(AnnealRobot r, final PrintStream
psout) {
            double[] results = new double[GAMES_COUNT];
            for(int i = 0; i < GAMES_COUNT; ++i)
            {
                final PlainGame game = new PlainGame(settings);
                final Robot secondRobot = new SecondTeamRobot();
                game.addRobot(r, TeamNumber.FIRST);
                game.addRobot(secondRobot, TeamNumber.SECOND);
                game.start();
                for (int j = 0; j < SMALL_ITERATIONS_COUNT; j++) {
                    game.nextTick();
                }
                final int firstScore =
game.getGameStatistics().getFirstTeamScore();
                final int secondScore =
game.getGameStatistics().getSecondTeamScore();
                psout.println(firstScore + ":" + secondScore + " ");
                results[i] = secondScore;
            }
            Arrays.sort(results);
            double result = 0.0;
            for(int i = 1; i != GAMES_COUNT - 1; ++i)
                result += results[i] / (GAMES_COUNT - 2);

            return result;
```

```
        }

        private final class SecondTeamRobot extends Robot {
            protected SecondTeamRobot() {
                super("Second");
            }

            @Override
            public void onStatus(StatusEvent event) {
                Vector ballPos = event.getBallInfo().getPosition();
                Vector myPos = event.getMyInfo().getPosition();

                boolean top = ballPos.getY() >
event.getFieldInfo().getHeight() / 2;

                try {
                    ballPos =
ballPos.add(event.getBallInfo().getSpeed().add(
                            new Vector(10.0, top ? 10.0 : -
10.0)).normalize()
                            .multiply(Vector.distance(ballPos, myPos)
/ 1.5));
                } catch (ArithmeticException e) {
                    // Ignore exception of normalize method
                }


setSpeedVector(ballPos.subtract(myPos).normalize().multiply(3.0));
            }
        }

        public static void main(String[] args) {
            new Thread(new AnnealMain()).start();
        }

    }
```

Приложение 2. Построенный автомат

Для наглядности представим конечный автомат в виде графа. Каждое состояние автомата является деревом решений (разд. 3.3). На рис. 8–15 изображены деревья решений для каждого из восьми состояний. Деревья стоились автоматически в программе *GraphViz*.

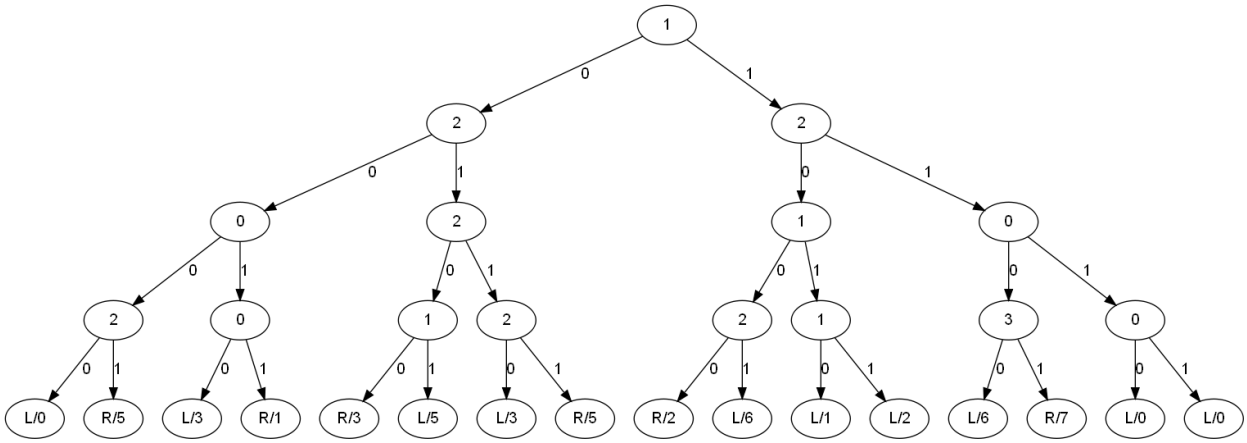

Рис. 8. Дерево решений состояния 1

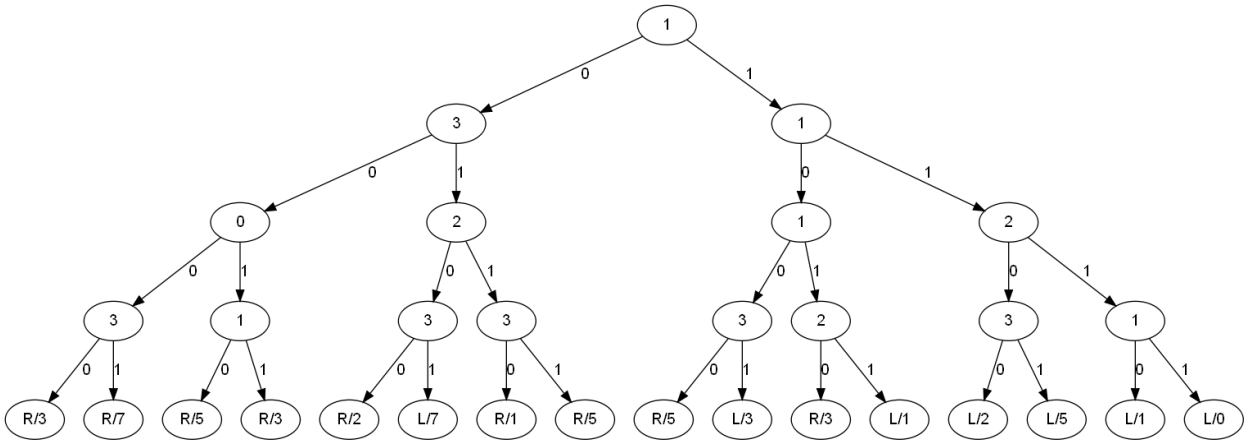

Рис. 9. Дерево решений состояния 2

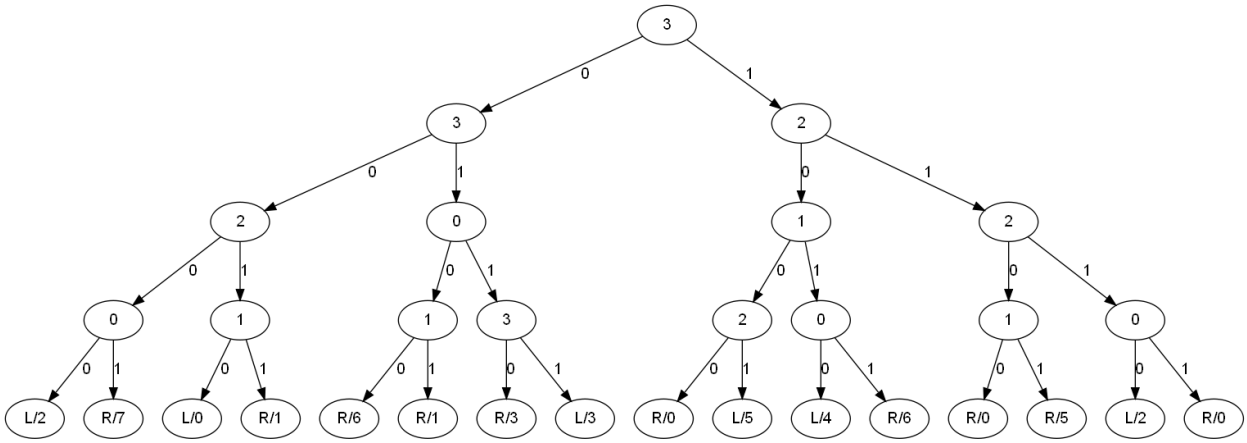

Рис. 10. Дерево решений состояния 3

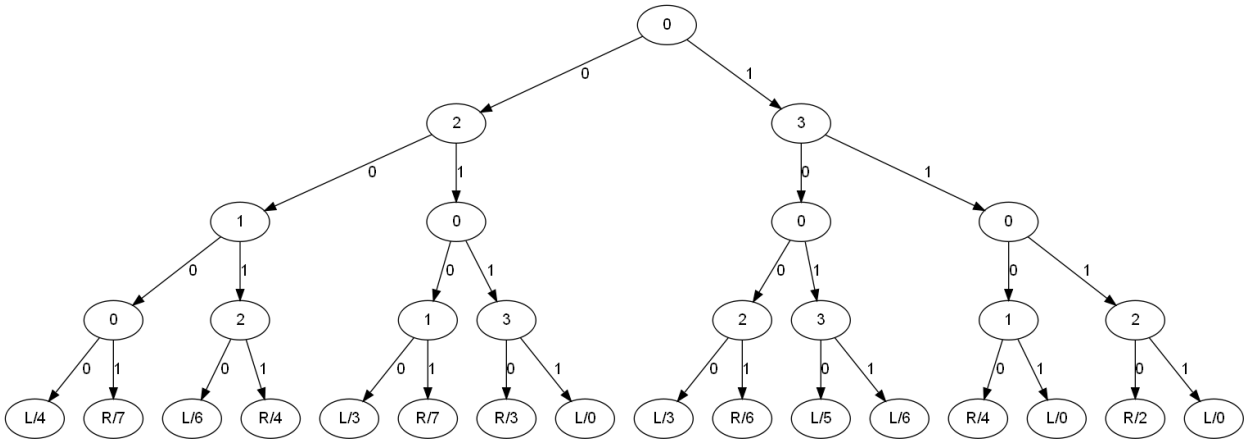

Рис. 11. Дерево решений состояния 4

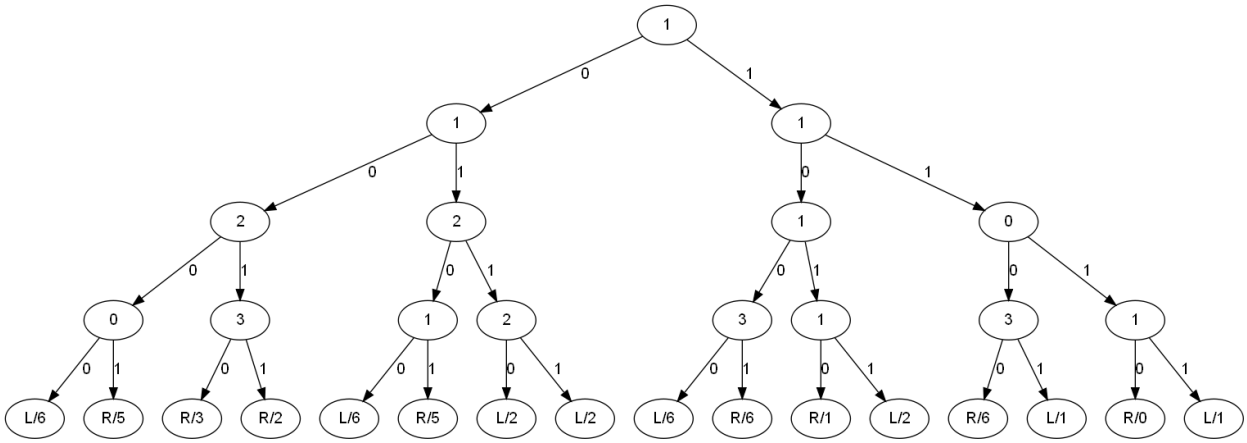

Рис. 12. Дерево решений состояния 5

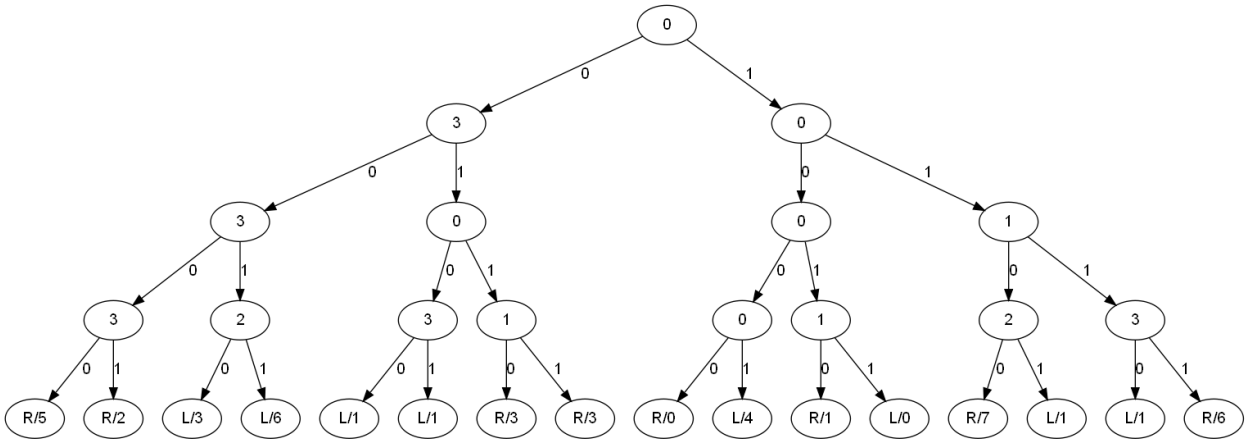

Рис. 13. Дерево решений состояния 6

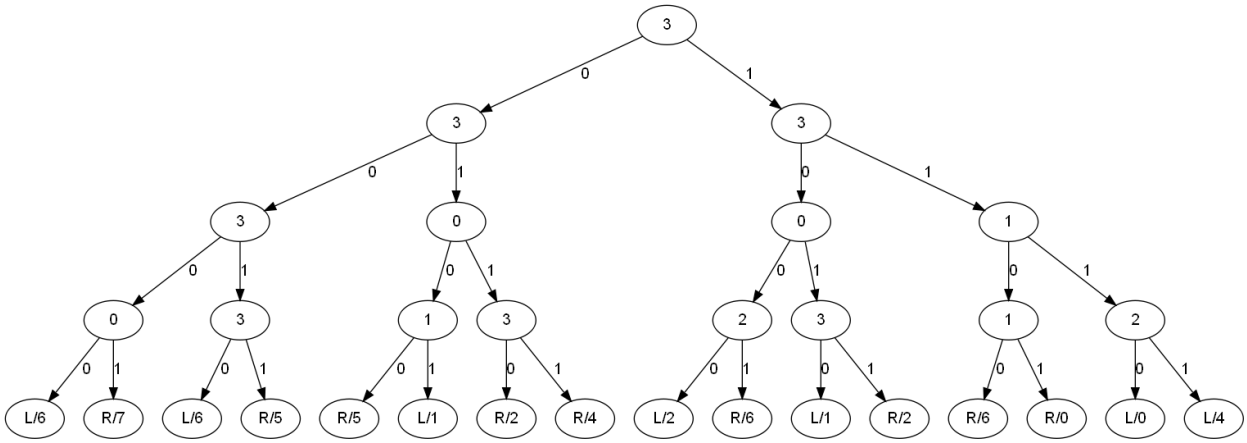

Рис. 14. Дерево решений состояния 7

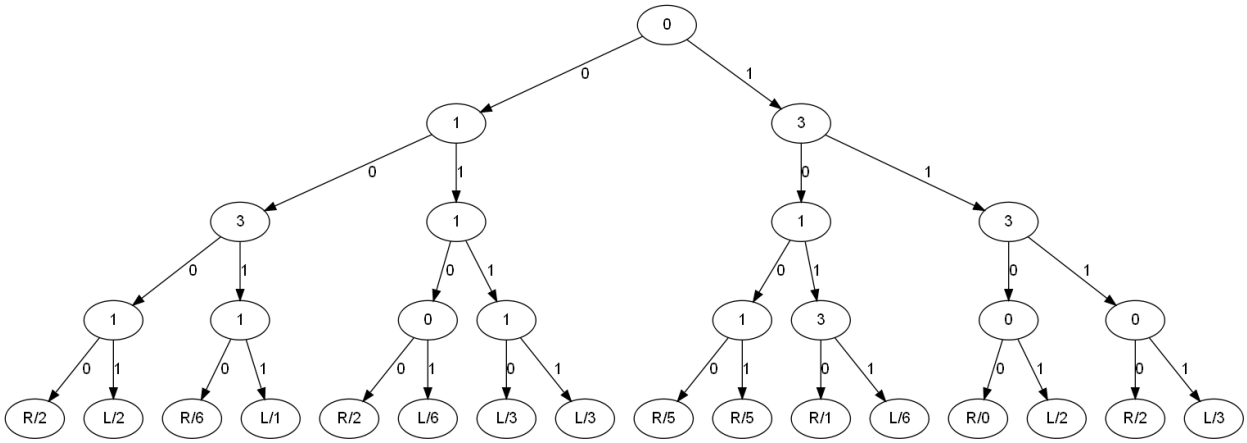

Рис. 15. Дерево решений состояния 8