Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования   
«Нижегородский государственный университет им. Н.И. Лобачевского»

Институт информационных технологий, механики и математики

**Отчет по лабораторной работе**

**АЛГОРИТМЫ ПОИСКА НАИЛУЧШЕГО ХОДА   
В ИГРЕ «РУССКИЕ ШАШКИ»**

**Выполнили**:

студент группы 0823-1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Грачева Е. А.

студент группы 0826-1

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Панов А. А.

**Руководитель**:

доц. каф. МОСТ ИИТММ

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Мееров И.Б.

Нижний Новгород

2017

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 4](#_Toc482959296)

[1 Постановка задачи 6](#_Toc482959297)

[2 Обзор методов поиска лучшего хода 7](#_Toc482959298)

[2.1 Алгоритм полного перебора 7](#_Toc482959299)

[2.2 Оценочная функция 9](#_Toc482959300)

[2.3 Поиск с отсечениями 10](#_Toc482959301)

[2.4 Форсированные варианты 12](#_Toc482959302)

[3 Программная реализация 14](#_Toc482959303)

[3.1 Алгоритмическое ядро (C++) 15](#_Toc482959304)

[3.1.1 Описание структур данных 15](#_Toc482959305)

[3.1.2 Описание алгоритмов 17](#_Toc482959306)

[3.2 Графический интерфейс (C#) 19](#_Toc482959307)

[3.2.1 Архитектура программной системы 19](#_Toc482959308)

[3.2.2 Описание структур данных 20](#_Toc482959309)

[3.2.3 Описание алгоритмов 21](#_Toc482959310)

[4 Руководство пользователя 23](#_Toc482959311)

[4.1 Режим Bot VS Bot 23](#_Toc482959312)

[4.2 Режим OnePlayer 23](#_Toc482959313)

[4.3 Режим Two Players 23](#_Toc482959314)

[4.4 Режим Constructor 24](#_Toc482959315)

[4.5 Настройки игры 24](#_Toc482959316)

[5 Апробация 25](#_Toc482959317)

[5.1 Противостояние с программой «Тундра» 25](#_Toc482959318)

[5.2 Противостояние с программой «Aurora Borealis» 25](#_Toc482959319)

[6 Сбор статистических данных 27](#_Toc482959320)

[6.1 Размерность задачи 27](#_Toc482959321)

[6.2 Алгоритм альфа-бета отсечений 29](#_Toc482959322)

[6.2.1 Дебют 29](#_Toc482959323)

[6.2.2 Миттельшпиль 29](#_Toc482959324)

[6.3 Форсирование 31](#_Toc482959325)

[6.3.1 Дебют 31](#_Toc482959326)

[7 Дальнейшие пути развития 32](#_Toc482959327)

[ЗАКЛЮЧЕНИЕ 33](#_Toc482959328)

[ЛИТЕРАТУРА 34](#_Toc482959329)

[ПРИЛОЖЕНИЕ 1. ФРАГМЕНТЫ ИСХОДНОГО КОДА (C++) 35](#_Toc482959330)

[ПРИЛОЖЕНИЕ 2. ФРАГМЕНТЫ ИСХОДНОГО КОДА (C#) 40](#_Toc482959331)

[ПРИЛОЖЕНИЕ 3. ИНТЕРФЕЙС ИГРЫ 46](#_Toc482959332)

[ПРИЛОЖЕНИЕ 4. ХОД ПАРТИИ С «AURORA BOREALIS» 48](#_Toc482959333)

ВВЕДЕНИЕ

Разработка компьютерных программ, играющих в популярные интеллектуальные игры, стала одним из актуальных направлений исследований в 60-е годы XX века. Существовавшие в то время компьютеры обладали по сегодняшним меркам достаточно скромными возможностями, что дополнительно мотивировало ученых разрабатывать эффективные алгоритмы и структуры данных, позволяющие не только экономить память, но и быстро обрабатывать данные. Среди них можно отметить процедуры компьютерного перебора с отсечением неперспективных вариантов, способы организации и использования дебютных баз, методы битового представления текущей позиции, алгоритмизация распределения времени на «обдумывание» ходов. В настоящий момент эти и многие другие решения применяются для разработки разных игровых программ, среди которых шахматы, шашки, го и многие другие.

В этой связи необходимо отметить достижения знаменитых советских математиков Г. Адельсона-Вельского, В. Арлазарова, М. Донского, разработавших шахматную программу *Каисса*, которая выиграла первый чемпионат мира среди шахматных программ в 1974г[1].Также важным этапом в истории развития компьютерных программ, играющих в интеллектуальные игры, стал выигрыш специализированного шахматного компьютера *DeepBlue*, разработанного компанией IBM, у чемпиона мира по шахматам Г. Каспарова. Данное событие, произошедшее в 1997 году, показало, что мощь организованного надлежащим образом интеллектуального компьютерного перебора становится неподвластной человеку.

Еще более очевидным данный факт стал совсем недавно, в 2016 году, когда программа *AlphaGo* выиграла матч у одного из сильнейших игроков в го, Ли Седоля. До этого времени считалось, что го является игрой, количество вариантов в которой настолько велико, что перебор оказывается неподвластным компьютеру и человек может реализовать преимущество в стратегическом мышлении. Однако разработчикам удалось доказать ошибочность этого мнения. Стоит отметить, что поиск ходов основывался не на традиционном поиске с отсечениями, а на методах машинного обучения, что имеет прямую связь с разработкой искусственного интеллекта.

Вне зависимости от используемых методов, разработка структур данных и алгоритмов для подобных игровых программ является интересной и сложной задачей, позволяющей изучить много нового и интересного, что и являлось основным содержанием представляемой нами работы. Наша основная задача заключалась в том, чтобы разработать прототип компьютерной программы для игры в шашки, позволяющей играть друг с другом, а также с компьютером. При этом мы базировались на классических методах построения таких программ, изучая и реализуя их с использованием средств языков программирования C++ и C#.

Отчет построен следующим образом. В разделе 1 приведена постановка задачи. В разделе 2 описаны основные алгоритмы поиска лучшего хода (полный перебор, перебор с отсечениями, форсирование вариантов и др.).Раздел 3 содержит описание программной реализации. Раздел 4 включает в себя описание разработанного приложения. В разделе 5 представлены результаты тестирования созданного компьютерного игрока при игре с известными программами «Тундра» и «Aurora Borealis». В разделе 6 представлены результаты исследования, имеющего цель собрать такие статистические данные, как, например, среднее количество возможных ходов для позиции. Подобные знания необходимы для того, чтобы была возможность приблизительно оценивать затрачиваемые ресурсы. В разделе 7 сформулированы планы по дальнейшему развитию результатов работы. В заключении сформулированы основные результаты.

# Постановка задачи

Были поставлены следующие задачи:

* изучить основные подходы к построению алгоритмов поиска лучшего хода в игровой программе;
* реализовать логику игры "Русские шашки";
* создать виртуального игрока, способного оценивать ситуацию на доске и определять наилучший ход;
* реализовать пользовательский интерфейс игры "Русские шашки";
* оценить качество игры созданного искусственного интеллекта в сравнении с другими известными игровыми программами, например, "*Тундра*" или "*Aurora Borealis*".

При создании приложения мы ориентировались на стандартные правила игры в классические «Русские шашки»[2].

# Обзор методов поиска лучшего хода

Определим функцию ***f :P→Z***, которая ставит в соответствие позиции **pos** из множества всевозможных позиций ***P***целое число, отражающее «выгодность» этой позиции для текущего игрока***.*** Такая функция называется *оценочной функцией*[3]*.*Чем больше значение функции, тем выгодней позиция. Таким образом, ставя каждому ходу в соответствие его оценку, мы бы могли однозначно определить лучший ход.

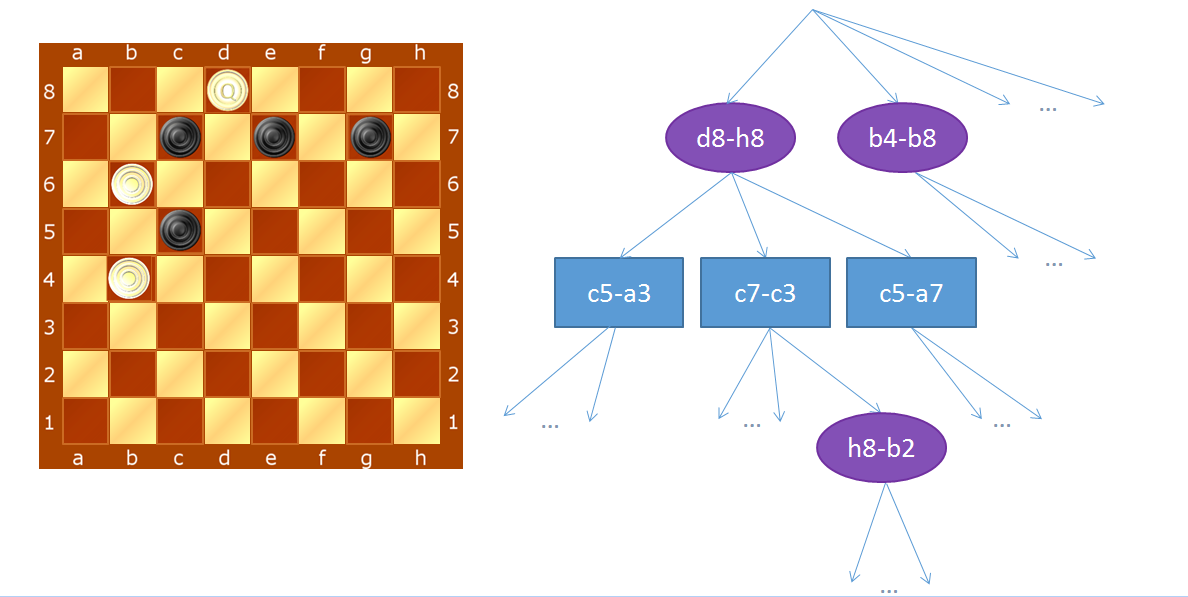
На компьютере с неограниченными вычислительными возможностями можно определить оценочную функцию следующим образом:

* присвоим всем финальным позициям значения –1, 0, 1, в зависимости от исхода игры (проигрыш, ничья, победа);
* применим рекурсивное правило где **pos** соответствует ходу текущего игрока, **pos'** соответствует ходу противника, а **pos→pos'** обозначает все допустимые ходы из позиции **pos**[1]*.*

Вычисление описанной выше оценочной функции потребует перебора всех этих позиций, что практически невозможно (канадским ученым потребовалось 200 компьютеров и 20 лет вычислений)[4].Следовательно, приходится иначе определять оценочную функцию и находить другие способы нахождения лучшего хода, которые, возможно, не будут такими точными, но зато будут не столь затратными по времени.

## Алгоритм полного перебора

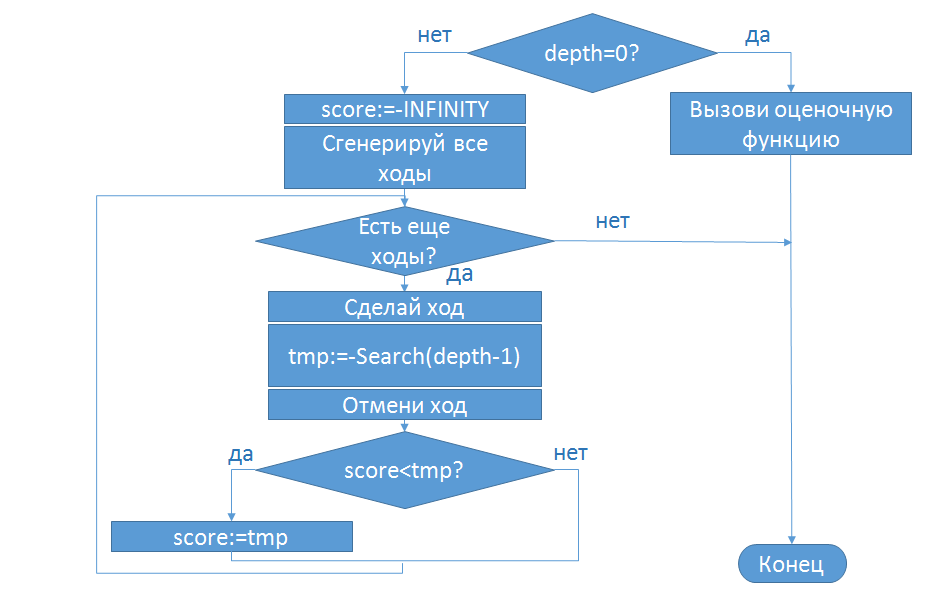
Самым простым способом получить если не лучший, то хотя бы осмысленный ход, является метод полного перебора с возвратом. Данный метод нетрудно реализовать в виде рекурсивного алгоритма. Так, если сгенерировать все возможные ходы из данной позиции для белых, затем для каждого такого хода повторить те же действия для черных, потом снова для белых и т.д., получится простой и понятный рекурсивный алгоритм, основанный на переборе позиций. Пример дерева игры, возникающего таким образом, представлен на рисунке 1.Однако, как было отмечено ранее, рассмотреть все возможные варианты до конца невозможно, поэтому приходится ограничивать глубину рекурсии. Пусть на каждом шаге известна текущая глубина рекурсии – параметр ***depth***, первоначально равный максимальной заданной глубине. Тогда если ***depth*** ≠ 0, то рекурсия будет вызываться с параметром ***depth-1***, в противном случае вычисления следует прекратить и вызвать *оценочную функцию[[1]](#footnote-1)*.Блок-схема алгоритма полного перебора представлена на рисунке 2.



*б)*

*а)*

*Рисунок 1. Пример дерева игры; а) позиция на доске, б) построенное для нее дерево*



*Рисунок 2. Блок-схема алгоритма полного перебора*

## Оценочная функция

Пусть P– множество всевозможных позиций на доске. Как уже было сказано, функция ***f : P → Z***, ставящая в соответствие некоторой позиции из множества ***P***целое число, отражающее «выгодность» этой позиции для текущего игрока, называется *оценочной функцией*.

Простейшая оценочная функция считает шашки на доске, т.е. суммирует вес всех белых шашек и вычитает из полученного результата сумму всех черных шашек. Если шашек поровну, функция вернет 0.

Установим вес шашек следующим образом:

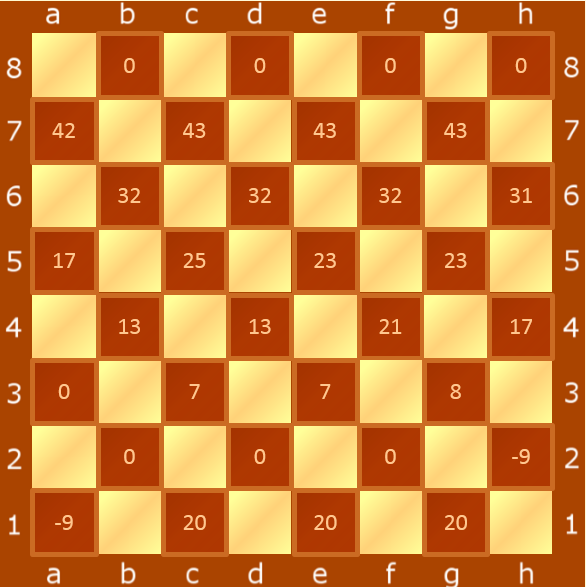
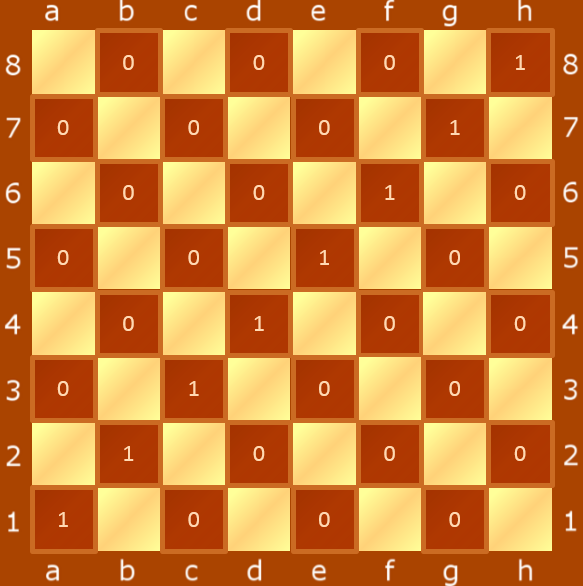
* простая шашка – 100;
* дамка – 300.

Таким образом, простая оценочная функция при расстановке шашек как на рисунке 1*a*для белого игрока вернет число 100, для черного игрока -100.

Более сложная оценочная функция учитывает ценность полей доски, которая определяется следующими принципами:

* надо стараться по возможности подвигаться вперед, т.к. чем ближе к последней горизонтали, тем сила полей больше;
* поскольку ценность шашек a1 и h2 является наименьшей, то их надо по возможности быстро ввести в игру;
* учитывая, что с полей c5 и f4 удобно препятствовать развитию сил противника, следует бороться за овладением центральными полями c5 и f4, стремиться к захвату центра;
* следует придерживаться принципа пропорционального, равномерного распределения шашек по обоим флангам;
* шашки с полей c1, e1, g1 лучше без особой надобности не сдвигать, чтобы не открывать поля последнего ряда;
* дамке по возможности стоит контролировать главную диагональ, т.к. она будет препятствовать продвижению противника в дамки.

Согласно этим правилам, ценность полей доски была обозначена следующим образом (рисунок 3):



*а) б)*

*Рисунок 3. Оценка клеток доски; а) для белой шашки, б) для дамки*

На практике оказалось, что компьютерный игрок, использующий оценочную функцию, считающую только материал, проигрывает компьютерному игроку, использующему более сложную оценочную функцию, что и следовало ожидать.

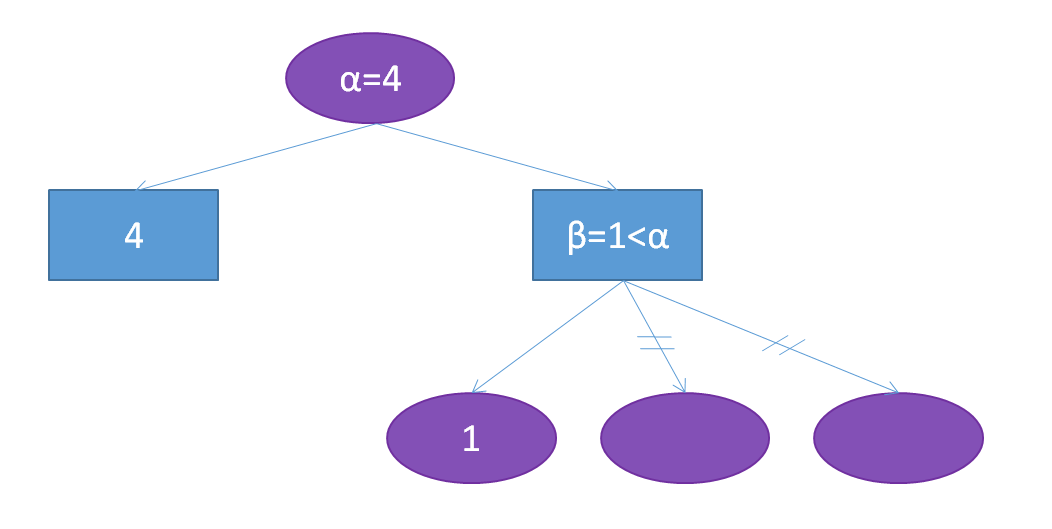
## Поиск с отсечениями

Рассмотрим основной алгоритм оптимизации перебора –­­ алгоритм *alpha-beta*[5]. Суть его в том, что для получения оценки такой же точности, как и при полном переборе, совершенно не обязательно просматривать все варианты. Для определения отсекаемых вариантов не требуется знать особенностей данной игры.

В рекурсивной функции мы должны ввести две новые переменные — максимум для белых (alpha) и максимум для черных (beta). При начальном вызове обе эти величины равны минимально возможному значению. Если в какой-то позиции, например, для белых, мы получили результат, превышающий максимум для белых, достигнутый до этого, мы увеличиваем этот максимум.

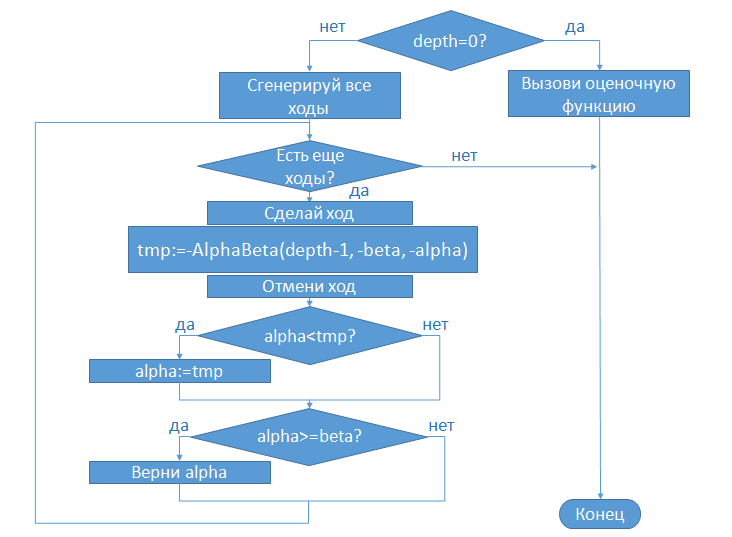
Теперь рассмотрим некоторую стадию игры. Представим, что где-то в глубине дерева перебора мы обнаружили, что максимумы черных и белых сравнялись. Допустим, мы просчитываем позицию для черных. Если мы продолжим перебирать в данной позиции, то максимум для черных может еще увеличиться, а может остаться прежним, но он уже сравнялся с максимумом белых. Это значит, что когда программа поднимется на уровень выше (рекурсивно), результат не будет записан, т. к. он не превышает максимума для белых в этой позиции, т.е. в данной позиции для черных мы можем прекратить перебор и вернуть полученный результат досрочно, поскольку дальше считать нет смысла[5].

Проиллюстрируем вышесказанное на следующем примере (рисунок 4). Пусть на каком-либо шаге рекурсии максимум для белых (alpha) равен 4. Сделаем какой-либо ход и просчитаем возможные ходы для черных (опустимся на уровень ниже). Пусть при дальнейших вычислениях для одного из ходов результат равен 1 (beta = 1). Если впоследствии найдется ход, оценка которого будет меньше 1, то он будет записан в beta, но не будет записан в alpha, и наоборот, если найдется ход, оценка которого будет больше 1, то он не будет записан даже в beta, следовательно, и не будет записан в alpha. Получается, дальнейшие вычисления бессмысленны.



*Рисунок 4. Пример работы алгоритма alpha-beta*

Блок-схема алгоритма поиска с отсечениями представлена на рисунке 5.

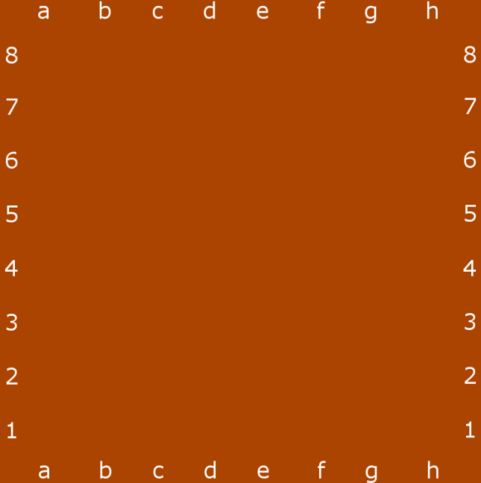


*Рисунок 5. Блок-схема алгоритма поиска с отсечениями*

При сборе статистики[[2]](#footnote-2) выяснилось, что при использовании алгоритма *alpha-beta* мы просматриваем примерно в 80 раз меньше позиций, чем при обычном поиске.

## Форсированные варианты

В некоторых ситуациях, например, в случае размена, прекращение вычислений по достижении максимальной глубины рекурсии может привести к крайне неверной оценке позиции (*эффект горизонта*). Рассмотрим пример (рисунок 6).Пусть максимальная глубина рекурсии равна 2. Ход белых. Даже неопытному игроку очевидно, что лучший ход – d8-h8, поскольку он приводит к полному уничтожению противника. Однако компьютер не увидит преимуществ такого хода и сходит h2-b8.Чтобы избежать подобных ситуаций, отдельные ветки стоит просчитывать на большую глубину.



*Рисунок 6. Пример позиции, для которой необходимо форсирование*

В шашках форсированными вариантами являются взятия. Поэтому в конце каждой ветки рекурсии вызывается упрощенная функция поиска, рассматривающая только взятия. Если же их не было, то сразу вызывается оценочная функция.

# Программная реализация

Проект состоит из двух частей — вычислительного ядра и графического интерфейса (рисунок 7). В вычислительной части написаны различные функции поиска лучшего хода, то есть искусственный интеллект для компьютерного игрока. Графическая оболочка предоставляет визуализацию игры, предоставляет настройки компьютерного игрока и дает возможность ходить реальному игроку.

Вычислительное ядро

Графическая оболочка

позиция

лучший ход

*Рисунок 7. Схема взаимодействия графической оболочки и вычислительного ядра*

Нахождение наилучшего хода требует большого объема вычислений и для обеспечения наилучшей производительности вычислительное ядро написано в виде динамической библиотеки на C++.

Графическая оболочка, написана на C#, так как данный язык предоставляет удобный графический инструментарий.

Итоговое приложение отображает игру с помощью графической оболочки и находит лучший ход с помощью вычислительного ядра.

## Алгоритмическое ядро (C++)

### Описание структур данных

Далее приведены основные классы, необходимые для организации работы компьютерного игрока.

#### Class Checker

Класс *Checker* описывает структуру хранения шашки. В нем кроме идентификатора шашки, представляющего собой целое число от 1 до 12, присутствуют еще 2 типа полей. Первый тип описывает основные признаки шашки, такие как:

* цвет;
* тип (шашка или дамка);
* координаты клетки, на которой стоит шашка.

Второй тип полей класса необходим для организации двухсвязного списка на массиве[[3]](#footnote-3). В эту группу входят следующие поля:

* номер следующей по списку шашки;
* номер предыдущей по списку шашки.

#### ClassListOfCheckers

Класс *ListOfCheckers*является типом, представляющим собой список на массиве. Необходимость его создания обусловлена тем, что в программе необходимо как обращаться к шашке по индексу, так и часто добавлять и удалять шашки. Единственное поле класса – массив *Checker[[4]](#footnote-4)* на 13 элементов (12 шашек + голова). Благодаря особой структуре класса *Checker* возможно организовать список.

Также для удобства был создан класс *LChIterator*, представляющий собой итератор для класса *ListOfCheckers*.

#### Class Board

Доска представляет собой одномерный массив из указателей на *Checker*. Это необходимо для того, чтобы была возможность легко находить шашку, зная ее координаты.

#### Class Move

При реализации алгоритмов появляется необходимость хранить ход шашки. Данный класс предоставляет все необходимые для этого поля и методы. Ход характеризуют следующие признаки:

* идентификатор шашки, которая ходит;
* первоначальные координаты этой шашки;
* координаты шашки после того, как был сделан ход;
* съеденные шашки;
* стала ли шашка дамкой в процессе хода.

#### Class Cache

На каждом шаге рекурсии необходимо хранить возможные ходы. Можно было бы создавать на каждом шаге рекурсии массив ходов, но, во-первых, это может привести к дополнительным накладным расходам, во-вторых, выделять место пришлось бы с большим запасом, поскольку неизвестно, сколько ходов может быть сгенерировано для определенной позиции. Поэтому мы выделяем место для хранения ходов в виде большого массива один раз в самом начале, а во время работы рекурсии записываем туда ходы. Очищение массива происходит посредством перемещения указателя на конец массива в сохраненную точку. Данный массив представлен классом *Cache*.

Сбор статистики[[5]](#footnote-5) позволяет примерно оценить размеры данного массива. Так в случае использования алгоритма альфа-бета при глубине вычислений 12 его размер приблизительно равен 250 элементов.

### Описание алгоритмов

Основные алгоритмы, используемые в программе(полный перебор[[6]](#footnote-6), перебор с отсечениями[[7]](#footnote-7), форсирование вариантов[[8]](#footnote-8), использование различных оценочных функций[[9]](#footnote-9)) описаны в соответствующем разделе[[10]](#footnote-10). Здесь же мы представим вспомогательные алгоритмы, необходимые для реализации поиска лучшего хода.

Реализация данных алгоритмов представлена в приложении 1.

#### Генерациявсевозможных ходов

Генерация ходов для позиции в шашках не является тривиальной задачей, поскольку, во-первых, возможны взятия сразу нескольких шашек, во-вторых, передвижения дамки (особенно с взятиями) требуют внимательного изучения правил игры. Алгоритм взятий для шашек можно описать следующим образом:

1. исследуем все четыре направления (по диагонали влево-вверх, вправо-вверх, вправо-вниз и влево-вниз) на наличие взятий (рассматриваются только соседние клетки);
2. если возможно взятие, «перепрыгнем» через битую шашку и поменяем ее цвет на противоположный (это необходимо, поскольку брать шашку дважды и убирать ее с доски до окончания хода нельзя по правилам), иначе завершим поиск взятий;
3. если текущая позиция находится на последнем ряду, изменим тип шашки на дамку и применим алгоритм взятий для дамки, начиная с шага 3 (см. далее);
4. в противном случае исследуем все вышеперечисленные направления, кроме того, откуда пришла шашка, на наличие взятий;
5. повторим алгоритм, начиная с шага 2.

Алгоритм взятий для дамок немного сложнее:

1. исследуем все четыре направления на наличие взятий (рассматриваются не только соседние клетки);
2. если возможно взятие, «перепрыгнем» через битую шашку (поменяв ее цвет) и поставим дамку на первую свободную клетку, иначе завершим поиск взятий;
3. исследуем направления, перпендикулярные направлению, с которого пришла дамка, на наличие взятий;
4. если в этих направлениях есть взятия, повторим алгоритм, начиная с шага 2, и вернемся в эту же точку;
5. в противном случае передвинем шашку на 1 клетку в первоначальном направлении (если это невозможно, то заверши поиск взятий);
6. повторим алгоритм, начиная с 2, до тех пор, пока впереди не возникнет преграда (шашка или граница доски);
7. если впереди возможно взятие, повторим алгоритм, начиная с шага.

Функция поиска ходов[[11]](#footnote-11)выполняет следующий алгоритм:

1. сохраним текущее состояние массива ходов[[12]](#footnote-12);
2. рассмотрим шашку текущего игрока;
3. проверим, может ли она взять какую-нибудь шашку противника;
4. если нет, то вызовем функцию поиска хода для шашки (учитывая, что она может быть как простой шашкой, так и дамкой) и повторим алгоритм, начиная с шага 2;
5. иначе вернем массив ходов в первоначальное состояние;
6. выполним функцию поиска взятий для шашки (учитывая ее тип);
7. для следующей шашки повторим алгоритм, начиная с шага 6;

#### Совершение и отмена хода

Также для выполнения рекурсивного поиска необходимо уметь делать ход и отменять его. Ход совершается по следующему алгоритму:

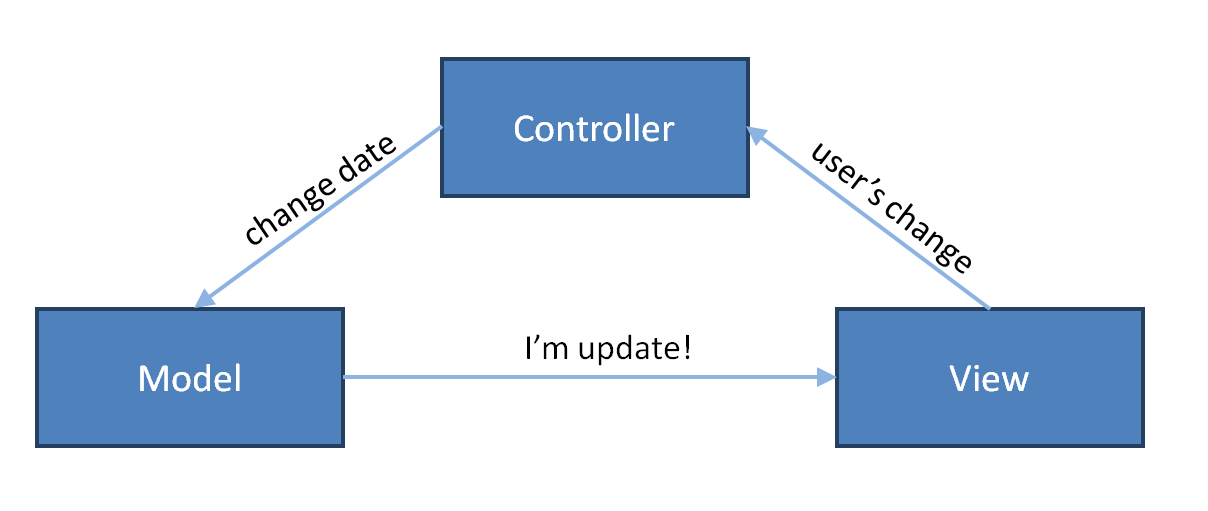
1. удалим все съеденные шашки с доски;
2. удалим основную шашку с доски;
3. поставим шашку на новое место;
4. если необходимо, изменим тип шашки.

Отмена хода выполняется в обратном порядке.

## Графический интерфейс (C#)

### Архитектура программной системы

Для удобства графическая часть выполнена по схеме MVC (Module, View, Controller)[6] (рис.8):



*Рисунок 8. Схема MVC*

В ней данные разделяются на три отдельных компонента: модель, представление и контроллер – таким образом, что модификация каждого компонента может осуществляться независимо. Также изменение и отображение данных происходит строго по заданной схеме:

*Модель (Model)* предоставляет данные и реагирует на команды контроллера, изменяя своё состояние;

*Представление (View)* отвечает за отображение данных модели пользователю, реагируя на изменения модели;

*Контроллер (Controller)* интерпретирует действия пользователя, оповещая модель о необходимости изменений.

### Описание структур данных

#### Class Game

В классе хранятся данные, необходимые для ведения игры – игровая доска, списки фигур, настройки компьютерного игрока. Также в классе реализована игровая механика, позволяющая ходить реальному и компьютерному игроку. Класс *Game*имеет возможность оповещать о наличии обновлений класс *GameForm[[13]](#footnote-13)*, который отвечает за отображение в концепции MVC. В свою очередь, класс *Game* в концепции MVC реализует *модель*.

#### Class LogicBoard

Этот класс представляет собой структуру хранения для игровой доски. Доска представляет собой двумерный массив клеток, в каждой из которых может находиться шашка. Кроме того, данный класс реализует в себе методы поиска возможных ходов для фигур.

#### ClassGameForm

В классе *GameForm* хранится интерактивное отображение игры. В концепции MVC реализует *отображение*. Данный класс не имеет прямого доступа к модели, однако он может при наличии события (например, пользовательского ввода) направить запрос классу *Controller[[14]](#footnote-14),* который, в свою очередь, при необходимости оповестит модель о произошедших изменениях.

#### ClassSettingForm

Класс *SettingForm* представляет собой инструмент для настройки компьютерного игрока. Позволяет устанавливать следующие параметры поиска лучшего хода:

* глубину вычислений (чем больше, тем сложнее);
* вид поиска (обычный поиск, поиск с отсечениями, форсирование[[15]](#footnote-15));
* тип оценочной функции (простая или сложная[[16]](#footnote-16)).

В концепции MVCтакже реализует *отображение*.

#### Class Controller

Класс *Controller* выполняет различные функции. Во-первых, сего помощью реализуется связь отображения и модели. В случае если класс *GameForm*зафиксировал некоторое событие, то он оповещает об этом контроллер, а последний, в свою очередь, информирует модель о необходимости изменений. Во-вторых, данный класс реализует основную логику приложения, т.е. он отвечает за взаимодействие пользователя и отображения.

### Описание алгоритмов

Основные алгоритмы, используемые в графическом интерфейсе:

* поиск хода;
* поиск пути для компьютерного игрока
* шаг игры;

Реализация данных алгоритмов представлена в приложении 2.

#### Поиск хода

Алгоритм, когда пользователь выделяет шашку, подсвечивает клетки, на которые можно сходить. Поиск хода осуществляется по алгоритму в пункте 3.1.2.1

#### Поиск пути для компьютерного игрока

Алгоритм ищет, как добраться до выбранной компьютерным игроком клетки. В качестве входных данных имеется начальная клетка пути, конечная и список шашек, которые были съедены по пути следования. Необходимо добраться до конечной клетки и отобразить процесс движения. Алгоритм с помощью поиска в ширину находит последовательность правильных ходов, затем отображает их по таймеру.

#### Шаг игры

Алгоритм работы класса *Game* описывает логику игры (очередность хода и взятий, а с помощью оповещений об изменении меняет отображение) и реализован с помощью конечного автомата. Опишем его с помощью таблицы переходов.

|  |  |  |  |
| --- | --- | --- | --- |
| **Текущее состояние** | **Действие пользователя** | **Действие** | **Следующее состояние** |
| wait | нажатие на шашку своего цвета | выделить шашку и подсветить возможные ходы | wait step |
| wait step | нажатие на подсвеченную клетку пути | сходить на подсвеченную клетку | end |
| wait step | нажатие иную клетку | очистить выделенные клетки | wait |
| wait eat | нажатие на бьющую шашку своего цвета | подсветить возможные ходы | wait eat select |
| wait eat | нажатие на иную клетку |  | wait eat |
| wait eat select | нажатие на подсвеченную клетку пути | переместить шашку  если взятий больше нет, взять шашку противника | end |
| wait eat select | нажатие на подсвеченную клетку пути | переместить шашку  если есть еще взятия, подсветить возможные ходы | eating |
| wait eat select | нажатие на иную клетку | очистить выделенные клетки, за исключением бьющих шашек своего цвета | wait eat |
| eating | нажатие на подсвеченную клетку пути | переместить шашку  если взятий больше нет, взять шашки противника | end |
| eating | нажатие на подсвеченную клетку пути | переместить шашку  если есть еще взятия, подсветить возможные ходы | eating |
| eating | нажатие на иную клетку |  | eating |

# Руководство пользователя

В процессе создания приложения мы старались обеспечить максимальную естесственность пользовательского интерфейса. Скриншоты игры представлены в приложении 3.

Приложение «Русские шашки» позволяет играть в следующих нижеперечисленных режимах.

## Режим Bot VS Bot

Данный режим игры позволяет тестировать эффективность различных алгоритмов поиска лучшего хода. Пользователь может настраивать различные параметры двух компьютерных игроков[[17]](#footnote-17) и запускать их в одной партии в качестве противников.

## Режим OnePlayer

Режим игры «один игрок» предусмотрен для тех пользователей, которые жалуются на отсутствие противника по силам. Игрок может настраивать способности компьютерного игрока желаемым образом[[18]](#footnote-18). Сложность может варьироваться от самой простой (противник ходит практически случайным образом) до наибольшей, соответствующей максимальным возможностям программы.

## Режим Two Players

Режим «два игрока» позволяет пользователям играть друг с другом с одного ПК.

## Режим Constructor

Конструктор предназначен для генерации различных первоначальных позиций. Он дает возможность тестировать компьютерного игрока, решать различные логические задачи и просто начинать игру не со стандартной расстановки шашек. Интерфейс, используемый в данном режиме игры, достаточно естественный и не требует пояснений.

## Настройки игры

Окно настроек вызывается каждый раз перед началом игры, кроме случая, когда выбран режим «два игрока». Оно позволяет выбирать цвет шашек для игроков, в режиме «конструктор» определять, кем является каждый из игроков – компьютером или человеком, и настраивать сложность компьютерного игрока. Компьютерный игрок характеризуется следующими критериями:

* глубиной вычислений (чем больше, тем сложнее);
* видом поиска (обычный поиск, поиск с отсечениями, форсирование[[19]](#footnote-19));
* типом оценочной функции (простая или сложная[[20]](#footnote-20)).

Пример окна настроек для режима «конструктор» представлен в приложении 3.

# Апробация

Способность созданного компьютерного игрока противостоять другим игрокам оценивалась с помощью непосредственного проведения турнира. Противниками являлись такие известные программы, как *«Тундра»* и *«Aurora Borealis»*, занявшие соответственно четвертое и пятое место на международном чемпионате мира по русским шашкам среди компьютерных программ 2008года[7].

## Противостояние с программой «Тундра»

Борьба компьютерного игрока с данной программой закончилась победой «Тундры». Выяснилось, что наш искусственный интеллект не умеет грамотно вести игру на стадии эндшпиля: если вначале и в середине партии он играет хорошо, то в конце игры он просто разрозненно двигает шашки, в то время как в действиях противника просматривается четкая структура. Вывод: необходимо рассматривать стадию эндшпиля отдельно и использовать для нее более совершенные алгоритмы, касающиеся непосредственно игры «русские шашки».

## Противостояние с программой «Aurora Borealis»

Непосредственным преимуществом программы «Aurora Borealis» перед «Тундрой»[[21]](#footnote-21) является наличие различных уровней сложности игры. «Aurora Borealis» предусматривает шесть таких уровней (уровни отсортированы по возрастанию сложности):

* 3-й разряд;
* 2-й разряд;
* 1-й разряд;
* кандидат;
* мастер;
* гроссмейстер.

Наша программа обыграла противника 3-го разряда[[22]](#footnote-22); партии с игроком 2-го, 1-го разряда завершились вничью с наличием у нашего игрока двух дамок, а у противника одной; игра против кандидата завершилась вничью с одной дамкой у каждой стороны. При игре с противником, настроенным на уровень сложности «мастер», тестируемый искусственный интеллект в какой-то момент не уложился во время, предложенное на обдумывание хода – 10 минут.

Основной проблемой также является неспособность нашей программы вести эндшпили. Также стоит заметить, что оба противника вычисляли лучший ход значительно быстрее тестируемого игрока, особенно это было заметно ближе к концу игры, когда благодаря практически пустой доске и при наличии нескольких дамок наша программа «думала» значительно дольше, чем это требовалось.

# Сбор статистических данных

При сборе статистики ставились следующие цели:

* оценка размерности задачи (число всех рассмотренных позиций в зависимости от глубины перебора);
* оценка преимущества алгоритма альфа-бета отсечений перед алгоритмом полного перебора;
* оценка преимущества форсирования перед поиском с отсечениями

Для решения этих задач был организован сбор и вывод необходимы данных в процессе игры. Были рассмотрены три стадии игры – дебют, миттельшпиль, эндшпиль.

Определим условные границы стадий игры:

* дебют – первые пять ходов игрока
* эндшпиль – последние пять ходов игрока
* миттельшпиль – все ходы между дебютом и эндшпилем

В случае ничьи эндшпилем все также считаются последние пять ходов.

## Размерность задачи

Статистика собиралась во время борьбы примерно равных компьютерных противников, поэтому нередко противники играли вничью. В таком случае к концу игры у игроков оставалось по несколько дамок, и из-за их наличия и большого свободного пространства на доске число ходов в эндшпиле росло существенно быстрее, чем в других стадиях игры. Результаты представлены в таблице 1. Если же один из противников выигрывал, то число возможных ходов было гораздо меньше. Вследствие такой неоднозначности не будем рассматривать эндшпили в исследованиях дальнейших исследованиях.

*Таблица 1. Зависимость числа рассматриваемых вершин дерева игры от глубины перебора*

## Алгоритм альфа-бета отсечений

### Дебют

Результаты оценки преимущества алгоритма на стадии дебюта представлены в таблице 2.

*Таблица 2. Зависимость числа рассматриваемых вершин дерева игры от глубины перебора для алгоритма альфа-бета в дебюте*

### Миттельшпиль

Результаты оценки преимущества алгоритма на стадии миттельшпиля представлены в таблицах 3, 4.

*Таблица 3. Зависимость числа рассматриваемых вершин дерева игры от глубины перебора на стадии миттельшпиля*

*Таблица 4. Зависимость числа рассматриваемых вершин дерева игры от глубины в алгоритме альфа-бета на стадии миттельшпиля*

В результате получилось, что алгоритм альфа-бета начиная с глубины 8, в среднем рассматривает 1,25%позиций от числа рассмотренных полным перебором. Это подтверждается и другими экспериментами, например, чтобы с помощью функции полного перебора с глубиной 12 просчитать первый ход потребуется более минуты (в среднем 81 секунда).

## Форсирование

### Дебют

Таблица

Число позиций, рассмотренное форсированием, может увеличиться до трех раз, по сравнению с нефорсированной версией. Это не сильно сказывается на времени работы, но существенно улучшает игру. При этом по статистике форсируются около 5% ходов.

Результаты миттельшпиля аналогичны.

# Дальнейшие пути развития

Дальнейшая разработка программы будет продолжена в нескольких направлениях. Во-первых, будут совершенствоваться алгоритмы. Предполагаются следующие улучшения:

* распараллеливание алгоритма «альфа-бета»
* создание базы дебютов;
* создание базы эндшпилей;
* добавлением различных алгоритмов для ведения игры на стадии эндшпиля (три дамки против одной и др.);
* кроме того, реализация некоторых других классических приемов ведения игры;
* добавление возможности сведения игры вничью;
* удаление циклических повторов позиций;
* и др.

Также рассматривается возможность использования методов машинного обучения.

Во-вторых, предусматриваются следующие нововведения:

* добавление возможности играть с другими игроками по сети и интернету;
* создание приложения «Русские шашки» для Android;
* усовершенствование интерфейса (добавление базы позиций, анализ проведенной игры и многое другое).

ЗАКЛЮЧЕНИЕ

В процессе работы сделано следующее:

* Были изучены и реализованы известные алгоритмы и структуры данных, необходимые для организации поиска лучшего хода в игре «Русские шашки».
* На основании этих алгоритмов был создан искусственный интеллект, способный делать ходы на уровне, по крайней мере, игрока 2-го разряда.
* Был создан графический интерфейс игры «русские шашки», позволяющий играть как с искусственным интеллектом, так и друг с другом.
* Разработанный компьютерный игрок был протестирован, в качестве противников выступали известные шашечные программы.
* Были собраны различные статистические данные, касающиеся таких величин, как, например, среднее количество ходов из одной позиции или средняя глубина форсирования ходов. Результаты экспериментов показали, что наибольший прирост эффективности был получен от алгоритма «перебор с отсечением»
* Предполагается дальнейшее совершенствование компьютерного игрока. Например, планами на ближайшее будущее являются создание базы дебютов и эндшпилей, реализация некоторых классических приемов игры (анализ форсированных позиций, типовые приемы игры в эндшпиле и др.).

ЛИТЕРАТУРА

1. *Адельсон-Вельский Г.М., Арлазаров В.Л., Битман А.Р., Донской М.В.* Машина играет в шахматы. – М.: Наука, 1983.
2. [Русские шашки // Федерация шашек России](http://shashki.ru/variations/draughts64/):[http://shashki.ru/variations/draughts64]
3. Бернхардсон Э.Глубокое обучение для… шахмат: [http://datareview.info/article/glubokoe-obuchenie-dlya-shahmat], 11.02.2016.
4. *Schaeffer J., BurchN.,Björnsson Y., Kishimoto A., Müller M., Lake R., Lu P., SutphenS.* Checkers Is Solved //American Association for the Advancement of Science, 1200 New York Avenue NW, Washington, DC 2005.
5. *Корнилов Е.Н.* Программирование шахмат и других логических задач.– СПб.:БХВ-Петербург, 2005.
6. *Фримен Э., Фримен Э.* Паттерны проектирования. – СПб.: Питер, 2011.
7. I чемпионат мира по русским шашкам среди компьютерных программ: [https://ru.wikipedia.org/wiki/Чемпионат\_мира\_по\_русским\_шашкам\_среди\_компьютерных\_программ\_2008].
8. *Ваныкина Г., Сундукова Т.*[Структуры и алгоритмы компьютерной обработки данных](http://www.intuit.ru/studies/courses/648/504/info): [http://www.intuit.ru/studies/courses/648/504/info].
9. *Гергель В.П., Лабутина А. А.* Учебно-образовательный комплекс по методам программирования //Нижний Новгород: ННГУ им. Н.И.Лобачевского. – 2007.

# ПРИЛОЖЕНИЕ 1. ФРАГМЕНТЫ ИСХОДНОГО КОДА (C++)

//ОЦЕНОЧНАЯ ФУНКЦИЯ (УЧИТЫВАЕТ СТОИМОСТЬ КЛЕТОК ДОСКИ)

int SmartEvaluate(int color)

{

LChIterator it;

int sum = 0;

for (it = checkers[color].begin(); it; it++)

sum += Price[it->GetType()] + ev\_board[it->GetColor()][it-> GetType()][it->GetCoord()];

for (it = checkers[!color].begin(); it; it++)

sum -= Price[it->GetType()] + ev\_board[it->GetColor()][it-> GetType()][it->GetCoord()];

return sum;

}

//ПОЛНЫЙ ПОИСК

int UsualSearch(int color, int depth, Move\* bestMove, int ev\_num = 0) {

if (depth == 0) return evaluate[ev\_num](color); //если на мах глубине

int score = -INF;

Move\* saved = cache.GetpLast();

Generate(checkers[color]);

for (Move\* pmove = saved; pmove < cache.GetpLast(); pmove++) {

MakeMove(\*pmove);

tmp = -UsualSearch(!color, depth - 1,0, ev\_num);

UnMakeMove(\*pmove);

if (score < tmp) {

score = tmp;

if (bestMove) \*bestMove = \*pmove;

}

}

cache.Rollback(saved);

return score;

}

//ПОИСК С ОТСЕЧЕНИЯМИ

int SearchAlphaBeta(int color, int depth, int alpha, int beta,

Move \* bestMove, int ev\_num=0) {

if (depth == 0) return evaluate[ev\_num](color); //если на мах глубине

Move\* saved = cache.GetpLast();

Generate(checkers[color]);

for (Move\* pmove = saved; pmove < cache.GetpLast(); pmove++) {

MakeMove(\*pmove);

tmp = -SearchAlphaBeta(!color, depth - 1, -beta, -alpha,0, ev\_num);

UnMakeMove(\*pmove);

if (tmp > alpha) {

alpha = tmp;

if (bestMove) \*bestMove = \*pmove;

}

if (alpha>=beta){

cache.Rollback(saved);

return alpha;

}

}

cache.Rollback(saved);

return alpha;

}

//ФОРСИРОВАННЫЕ ПРОДЛЕНИЯ

int QuiesCapt(int color, int alpha, int beta, int ev\_num = 0)

{

Move\* savedF = cache.GetpLast();

GenerateCaptures(checkers[color]); //генерирует только взятия

if (cache.GetpLast() - savedF <= 0) {

return evaluate[ev\_num](color);

}

for (Move\* pmove = savedF; pmove < cache.GetpLast(); pmove++) {

MakeMove(\*pmove);

tmp = -QuiesCapt(!color, -beta, -alpha, ev\_num);

UnMakeMove(\*pmove);

if (tmp > alpha) {

alpha = tmp;

}

if (alpha >= beta) {

cache.Rollback(savedF);

return alpha; }

}

cache.Rollback(savedF);

return alpha;

}

//ГЕНЕРАЦИЯ ХОДОВ

int SearchMoveChecker(Checker& ch);//поиск ходов для шашки

int SearchMoveDamka(Checker& ch);//поиск ходов для дамки

int SearchEatChecker(Checker& ch);//поиск взятий для шашки

int SearchEatDamka(Checker& ch);//поиск взятий для дамки

typedef int (\*pSearch)(Checker& ch);

const pSearch SearchMove[2] = {SearchMoveChecker, SearchMoveDamka };

const pSearch SearchEat[2] = { SearchEatChecker, SearchEatDamka };

//основная функция

void Generate(ListOfCheckers& list) {

ListOfCheckers::iterator it;

int mustEat;

Move\* saved;

mustEat = 0; saved = cache.GetpLast();

for (it = list.begin(); it && (!mustEat); it++)

mustEat = SearchMove[it->GetType()](\*it);

if (mustEat) {

cache.Rollback(saved);

for (it--; it; it++) {

board[it->GetCoord()] = 0;

SearchEat[it->GetType()](\*it);

board[it->GetCoord()] = &(\*it);

}

}

}

static int color;

static int coord;

static int num;

static int coordEaten;

static int eaten[12]; static int nEaten = 0;

static int type\_bool;

//осуществляют рекурсию, смотрят в 3 направлениях

void SearchEatDamkaInRay(int);

void SearchEatCheckerInRay(int route)

{

int f = 0;

MakeMoveChecker(route);//встала за съеденной шашкой

if (::type\_bool) {

if (CanEatDamka(Perp(route))) {

f++;

SearchEatDamkaInRay(Perp(route));

}

} //продолжает поиск в оставшемся направлении на правах дамки

else {

if (CanEatChecker(route)) {

f++;

SearchEatCheckerInRay(route);

}

if (CanEatChecker(Perp(route))) {

f++;

SearchEatCheckerInRay(Perp(route));

}

if (CanEatChecker(-Perp(route))) {

f++;

SearchEatCheckerInRay(-Perp(route));

}

}

//сохранение ходa, если нельзя больше есть

if (f == 0) cache.Push(GetMove(::coord));

UnMakeMoveChecker(route);

}

void SearchEatDamkaInRay(int route)

{

int f = 0, savedCoord, savedCoordEaten;

MakeMoveDamka(savedCoord, savedCoordEaten);//встала на съеденную шашку

//идем вперед и смотрим влево-вправо

do {

::coord += route;

if (CanEatDamka(Perp(route))) {

f++;

SearchEatDamkaInRay(Perp(route));

}

if (CanEatDamka(-Perp(route))) {

f++;

SearchEatDamkaInRay(-Perp(route));

}

} while (CanMove(::coord + route));

//если впереди препятсвие, смотрим, нельзя ли есть

if (CanEatDamka(route)) {

f++;

SearchEatDamkaInRay(route);

}

//сохранение всех ходов, если некуда больше есть

if (f == 0) do {

cache.Push(GetMove(::coord));

::coord -= route;

} while (CanMove(::coord));

UnMakeMoveDamka(savedCoord, savedCoordEaten);

}

//смотрят в 4 направлениях, вызывают функции, осущ. рекурсию

int SearchEatChecker(Checker& ch)

{

::color = ch.GetColor(); ::coord = ch.GetCoord(); ::num = ch.GetNum(); temp\_move.SetStartCoord(::coord);

for(int j=0; j<4; j++) if (CanEatChecker(direct[j]))

SearchEatCheckerInRay(direct[j]);

return 0;

}

int SearchEatDamka(Checker& ch)

{

::color = ch.GetColor(); ::coord = ch.GetCoord(); ::num = ch.GetNum();

temp\_move.SetStartCoord(::coord);

for (int j = 0; j<4; j++) if (CanEatDamka(direct[j]))

SearchEatDamkaInRay(direct[j]);

return 0;

}

//возвращают 1, если можно есть

int SearchMoveChecker(Checker& ch)

{

::color = ch.GetColor(); ::coord = ch.GetCoord(); ::num = ch.GetNum(); temp\_move.SetStartCoord(::coord);

int j;

for (j = 0; j<4; j++)

if (CanEatChecker(direct[j])) return 1;

for (j = 0; j<2; j++)

if (CanMove(::coord + forward[::color][j])) {

if (OnLastRow(::coord + forward[::color][j])) ::type\_bool = 1;

cache.Push(GetMove(::coord + forward[::color][j]));

::type\_bool = 0;

}

return 0;

}

int SearchMoveDamka(Checker& ch)

{

::color = ch.GetColor(); ::coord = ch.GetCoord(); ::num = ch.GetNum(); temp\_move.SetStartCoord(::coord);

int i, j;

for (j = 0; j<4; j++)

if (CanEatDamka(direct[j])) return 1;

for (j = 0; j<4; j++)

for (i = 1; CanMove(::coord + i\*direct[j]); i++)

cache.Push(GetMove(::coord + i\*direct[j]));

return 0;

}

# ПРИЛОЖЕНИЕ 2. ФРАГМЕНТЫ ИСХОДНОГО КОДА (C#)

//ГЕНЕРАЦИЯ ВОЗМОЖНЫХ ХОДОВ ДЛЯ ВЫБРАННОЙ ШАШКИ

public bool CanMoveInRay(int dx, int dy/\*orts of ray\*/)

{

switch (GetFigure())

{

case Figure.checker:

{

if (Checker.dy[(int)color] != dy) return false;

int x1 = x + dx;

int y1 = y + dy;

if (Inside(x1, y1) && Game.board[x1, y1].isEmpty())

{

way\_x = x1;

way\_y = y1;

return true;

}

return false;

}

case Figure.damka:

{

int x1 = x + dx;

int y1 = y + dy;

if (Inside(x1, y1) && Game.board[x1, y1].isEmpty())

{

way\_x = x1;

way\_y = y1;

return true;

}

return false;

}

}

return false;

}

public bool DamkaCanEatInRay(int dx, int dy, int x1, int y1)

{

while (Inside(x1, y1) && Inside(x1 + dx, y1 + dy) &&

IsEmptyInBoard(x1, y1))

{

x1 += dx;

y1 += dy;

}

if (Inside(x1, y1) && Inside(x1 + dx, y1 + dy) &&

CanBeEaten(Game.board[x1, y1], Game.board[x1 + dx, y1 + dy]))

return true;

return false;

}

public bool CanEatInRay(int dx, int dy/\*orts of ray\*/)

{

int x1 = x + dx;

int y1 = y + dy;

switch (GetFigure())

{

case Figure.checker:

if (Inside(x1, y1) && Inside(x1 + dx, y1 + dy) &&

CanBeEaten(Game.board[x1, y1], Game.board[x1 + dx, y1 + dy]))

{

eaten\_x = x1;

eaten\_y = y1;

return true;

}

return false;

case Figure.damka:

while (Inside(x1, y1) && Inside(x1 + dx, y1 + dy) &&

IsEmptyInBoard(x1, y1))

{

x1 += dx;

y1 += dy;

}

if (Inside(x1, y1) && Inside(x1 + dx, y1 + dy) &&

CanBeEaten(Game.board[x1, y1], Game.board[x1 + dx, y1 + dy]))

{

eaten\_x = x1;

eaten\_y = y1;

return true;

}

return false;

}

return false;

}

//поиск ходов

public void SearchWay(Moves m)

{

for (int i = 0; i < 2; i++)

for (int j = 0; j < 2; j++)

{

if (CanMoveInRay(dx[i], dy[j]))

{

m.AddWay(way\_x, way\_y);

if (m.selectedChecker.GetFigure() == Figure.damka)

while (Inside(way\_x + dx[i], way\_y + dy[j]) &&

IsEmptyInBoard(way\_x + dx[i], way\_y + dy[j]))

{

way\_x += dx[i];

way\_y += dy[j];

m.AddWay(way\_x, way\_y);

}

}

}

}

//поиск взятий

public void SearchEat(Moves m)

{

for (int i = 0; i < 2; i++)

for (int j = 0; j < 2; j++)

{

if (CanEatInRay(dx[i], dy[j]))

{

m.AddCanBeEaten(Game.board[eaten\_x, eaten\_y].GetChecker());

if (m.selectedChecker.GetFigure() == Figure.damka)

{

int dx1 = dx[i], x1 = eaten\_x;

int dy1 = dy[j], y1 = eaten\_y;

bool f = false;

while (Inside(x1 + dx1, y1 + dy1) &&

IsEmptyInBoard(x1 + dx1, y1 + dy1))

{

x1 += dx1;

y1 += dy1;

for (int k = 0; k < 2; k++)

for (int n = 0; n < 2; n++)

{

if ((dx[k] != -dx[i] || dy[n] != -dy[j]) &&

DamkaCanEatInRay(dx[k], dy[n], x1, y1))

{

m.AddWay(x1, y1);

f = true;

}

}

}

if (f == true) continue;

while (Inside(eaten\_x + dx[i], eaten\_y + dy[j]) &&

IsEmptyInBoard(eaten\_x + dx[i], eaten\_y + dy[j]))

{

eaten\_x += dx[i];

eaten\_y += dy[j];

m.AddWay(eaten\_x, eaten\_y);

}

}

else m.AddWay(eaten\_x + dx[i], eaten\_y + dy[j]);

}

}

}

//ШАГ ИГРЫ

public enum StatusGame

{

wait,

waitStep,

waitEat,

waitEatSelect,

eating,

endEating,

сalculating,

gameOver

}

private StatusGame statusGame = StatusGame.wait;

int w\_n, b\_n;

int[] w\_types, b\_types;

int[] w\_coords, b\_coords;

//шаг для игры с компьютерным игроком

public bool BotStep()

{

switch (statusGame)

{

case StatusGame.wait:

statusGame = StatusGame.сalculating;

SetArraysForBotStep();

int res = CallBot(w\_coords, w\_types, w\_n, b\_coords, b\_types, b\_n,

(int)color, statusDepth[(int)color], (int)statusSearch[(int)color],(int)statusEvaluate[(int)color]);

DecipherRes(res);

if (botMove.end == 1)

{

statusGame = StatusGame.gameOver;

return true;

}

if (botMove.eaten.Count == 0)

{

ShowBotWay(botMove.way[0].Item1, botMove.way[0].Item2);

statusGame = StatusGame.waitStep;

}

else

{

botMove.SearchInterm();

ShowBotWay(botMove.interm[0].Item1, botMove.interm[0].Item2);

if (botMove.eaten.Count>1) statusGame = StatusGame.eating;

else statusGame = StatusGame.endEating;

}

return false;

case StatusGame.waitStep:

MoveChecker(botMove.selectedChecker, botMove.way[0].Item1,

botMove.way[0].Item2);

botMove.Clear();

statusGame = StatusGame.wait;

return true;

case StatusGame.eating:

MoveChecker(botMove.selectedChecker, botMove.interm[0].Item1,

botMove.interm[0].Item2);

botMove.interm.Remove(botMove.interm[0]);

ShowBotWay(botMove.interm[0].Item1, botMove.interm[0].Item2);

if (botMove.interm[0] == botMove.way[0]) statusGame =

StatusGame.endEating;

return false;

case StatusGame.endEating:

MoveChecker(botMove.selectedChecker, botMove.way[0].Item1,

botMove.way[0].Item2);

foreach (Checker ch in botMove.eaten)

DeleteChecker(ch.x, ch.y);

botMove.Clear();

statusGame = StatusGame.wait;

return true;

case StatusGame.gameOver:

return true;

case StatusGame.сalculating:

return false;

}

return false;

}

//шаг для игры с человеком

public bool HumanStep(int x, int y) //true если ход закончен

{

bool isWay = board[x, y].GetIsWay();

switch (statusGame)

{

case StatusGame.wait:

if (board[x, y].GetColor() == color)

{

SelectCheckerAndSearchWay(x, y);

statusGame = StatusGame.waitStep;

}

return false;

case StatusGame.waitStep:

if (board[x, y].GetIsWay())

{

MoveChecker(moves.selectedChecker, x, y);

ClearWays();

statusGame = StatusGame.wait;

return true;

}

if (board[x, y].GetIsLight())

return false;

if (board[x, y].GetColor() == color)

{

ClearWays();

ClearSelected();

SelectCheckerAndSearchWay(x, y);

return false;

}

ClearWays();

ClearSelected();

statusGame = StatusGame.wait;

return false;

case StatusGame.waitEat:

if (board[x, y].GetIsLight())

{

SelectCheckerAndSearchEat(x, y);

statusGame = StatusGame.waitEatSelect;

return false;

}

ClearWays();

return false;

case StatusGame.waitEatSelect:

if (board[x,y].GetIsWay())

{

if (MakeEat(moves.selectedChecker, x, y) == true)

{

ClearWays();

ClearSelectedCanEat();

statusGame = StatusGame.wait;

return true;

}

else

{

ClearWays();

ClearSelectedCanEat();

SelectCheckerAndSearchEat(x, y);

statusGame = StatusGame.eating;

return false;

}

}

if (board[x, y] == moves.selectedChecker)

return false;

if (board[x,y].GetIsLight())

{

ClearWays();

SelectCheckerAndSearchEat(x, y);

return false;

}

ClearWays();

statusGame = StatusGame.waitEat;

return false;

case StatusGame.eating:

if (board[x, y].GetIsWay())

{

if (MakeEat(moves.selectedChecker, x, y) == true)

{

ClearWays();

ClearSelectedCanEat();

statusGame = StatusGame.wait;

return true;

}

ClearWays();

ClearSelectedCanEat();

SelectCheckerAndSearchEat(x, y);

return false;

}

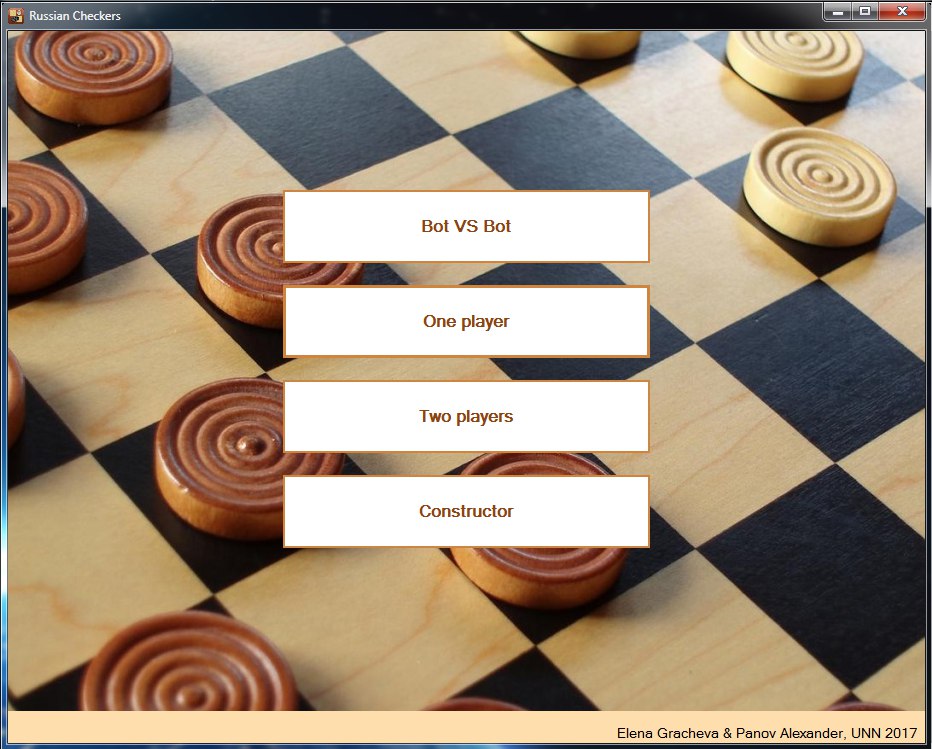
return false;

}

return false;

}

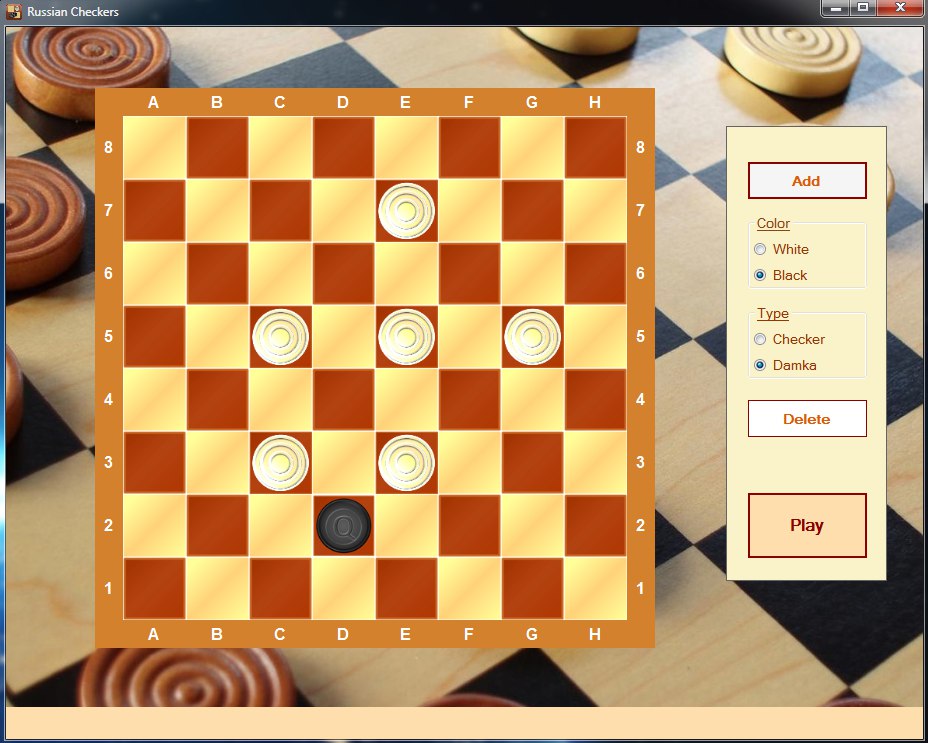
ПРИЛОЖЕНИЕ 3. ИНТЕРФЕЙС ИГРЫ

****

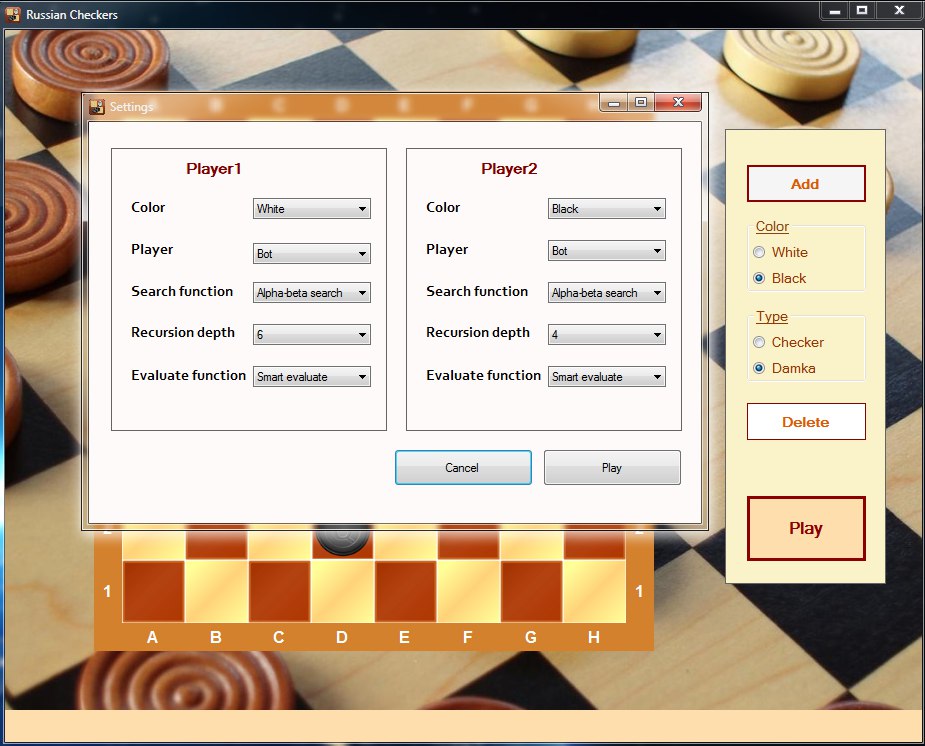
*Рисунок 1. Меню игры*

****

*Рисунок 2. Ход игры*

******

*Рисунок 3. Конструктор*

****

*Рисунок 4. Настройки*

# ПРИЛОЖЕНИЕ 4. ХОД ПАРТИИ С «AURORA BOREALIS»

|  |  |
| --- | --- |
| 1. g3-h4 f6-g5 2. h4-f6 g7-e5 3. c3-d4 e5-c3 4. d2-b4 h6-g5 5. h2-g3 b6-c5 6. b2-c3 g5-f4 7. e3-g5 c5-d4 8. c3-e5 d6-h2 9. a1-b2 h8-g7 10. g5-h6 a7-b6 11. b2-c3 e7-d6 12. b4-a5 d8-e7 13. f2-e3 g7-f6 14. c1-b2 d6-c5 15. c3-b4 e7-d6 16. b2-c3 f6-e5 17. c3-d4 e5-c3 18. b4-d2 b8-a7 | 1. e3-f4 c5-d4 2. d2-c3 d4-b2 3. a3-c1 b6-c5 4. e1-f2 c5-d4 5. f4-g5 d4-c3 6. g5-f6 c3-b2 7. c1-a3 d6-c5 8. f2-e3 c7-d6 9. a5-b6 c5-b4 10. a3-e7 a7-c5 11. e7-d8 c5-b4 12. d8-e7 f8-d6 13. h6-g7 b4-a3 14. g7-f8 d6-e5 15. f6-d4 a3-b2 16. f8-g7 b2-c1 17. g7-h6 c1-f4 18. h6-c1 |

1. См. п.2.2 [↑](#footnote-ref-1)
2. См. п.6 [↑](#footnote-ref-2)
3. См. п.3.1.1.2 [↑](#footnote-ref-3)
4. См. п.3.1.1.1 [↑](#footnote-ref-4)
5. См. п.6 [↑](#footnote-ref-5)
6. См. п.2.1 [↑](#footnote-ref-6)
7. См. п.2.2 [↑](#footnote-ref-7)
8. См. п.2.3 [↑](#footnote-ref-8)
9. См. п.2.4 [↑](#footnote-ref-9)
10. См. п.2 [↑](#footnote-ref-10)
11. Реализация см. Приложение 1 [↑](#footnote-ref-11)
12. См. п.3.1.1.5 [↑](#footnote-ref-12)
13. См. п. 3.2.2.3 [↑](#footnote-ref-13)
14. См. п. 3.2.2.5 [↑](#footnote-ref-14)
15. См. п.2.1 – 2.3 [↑](#footnote-ref-15)
16. См. п.2.4 [↑](#footnote-ref-16)
17. См. п.4.5 [↑](#footnote-ref-17)
18. См. п.4.5 [↑](#footnote-ref-18)
19. См. п.2.1 – 2.3 [↑](#footnote-ref-19)
20. См. п.2.4 [↑](#footnote-ref-20)
21. См. п.5.1 [↑](#footnote-ref-21)
22. Ход партии см. Приложение 4 [↑](#footnote-ref-22)