# Постановка задачи

Были поставлены следующие задачи:

* исследовать различные алгоритмы поиска лучшего хода в игровой программе;
* реализовать логику игры "Русские шашки";
* создать бота, способного оценивать ситуацию на доске и определять наилучший ход;
* реализовать пользовательский интерфейс игры;
* оценить качество игры бота в сравнении с другими известными игровыми программами, такими как "Тундра" или "Аврора".

### Логика игры "Русские шашки" определяется следующими правилами.

### Правила хода

* простая шашка ходит по диагонали вперёд на одну клетку;
* дамка ходит по диагонали на любое свободное поле как вперёд, так и назад.

### Правила взятия

* взятие обязательно. Побитые шашки и дамки снимаются только после завершения хода;
* простая шашка, находящаяся рядом с шашкой соперника, за которой имеется свободное поле, переносится через эту шашку на это свободное поле. Если есть возможность продолжить взятие других шашек соперника, то это взятие продолжается, пока бьющая шашка не достигнет положения, из которого бой невозможен. Взятие простой шашкой производится как вперёд, так и назад;
* дамка бьёт по диагонали, как вперёд, так и назад, и становится на любое свободное поле после побитой шашки. Аналогично, дамка может бить несколько фигур соперника и должна бить до тех пор, пока это возможно;
* при бое через дамочное поле простая шашка превращается в дамку и продолжает бой по правилам дамки;
* при взятии применяется правило [турецкого удара](https://ru.wikipedia.org/wiki/%D0%A2%D1%83%D1%80%D0%B5%D1%86%D0%BA%D0%B8%D0%B9_%D1%83%D0%B4%D0%B0%D1%80) — за один ход шашку противника можно побить только один раз. То есть, если при бое нескольких шашек противника шашка или дамка повторно выходит на уже побитую шашку, то ход останавливается;
* при нескольких вариантах взятия, например, одну шашку или две, игрок выбирает вариант взятия по своему усмотрению.

### Выигрыш партии

Партия считается выигранной в следующих случаях:

* если у одного из соперников побиты все шашки;
* если шашки одного из участников заперты и он не может сделать очередной ход.

### Ничья

Партия считается закончившейся вничью в следующих случаях:

* если три раза повторяется одна и та же позиция;
* если участник, имеющий три дамки (и более) против одной дамки противника, за 15 ходов не возьмёт дамку противника;
* если в течение 15 ходов игроки делали ходы только дамками, не передвигая простых шашек и не производя взятия.

# Дальнейшие пути развития

* реализовать базу дебютов;
* добавить различные алгоритмы для ведения игры на стадии эндшпиля, например, три дамки против одной и др.;
* предусмотреть возможность завершения игры вничью;
* убрать циклические повторы позиций
* и др.

### Описание алгоритмов

### Основные алгоритмы, используемые в программе (полный перебор[[1]](#footnote-2), перебор с отсечениями[[2]](#footnote-3), форсирование вариантов[[3]](#footnote-4), использование различных оценочных функций[[4]](#footnote-5)) описаны в разделе Алгоритмы. Здесь мы представим вспомогательные алгоритмы, необходимые для реализации поиска.

1. Генерация всевозможных ходов*[[5]](#footnote-6)*

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 «есть» 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 п.3 (см. далее);
4. в противном случае исследуем все вышеперечисленные направления, кроме того, откуда пришла шашка, на наличие взятий;
5. повторим алгоритм, начиная с п.2.

Алгоритм взятий для дамок будет выглядеть несколько 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 п.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 п.2, до тех пор, пока впереди не возникнет преграда (шашка или граница доски);
7. если впереди возможно взятие, повторим алгоритм, начиная с п.2.

Алгоритм поиска возможных ходов реализован в виде следующих функций:

int SearchEatChecker(Checker& ch);//поиск и запись взятий для шашки, возвращает 0

int SearchEatDamka(Checker& ch);//поиск и запись взятий для дамки, возвращает 0

int SearchMoveChecker(Checker& ch);//поиск и запись простых ходов для шашки, возвращает 1, //если возможны взятия, иначе возвращает 0

int SearchMoveDamka(Checker& ch);//поиск и запись простых ходов для дамки, возвращает 1, //если возможны взятия, иначе возвращает 0

void Generate(ListOfCheckers& list);//обобщает все вышеперечисленные функции

void Generate(ListOfCheckers& list);//генерирует только взятия (для форсирования)

Функция *Generate* выполняет следующий алгоритм:

1. сохраним текущее состояние кэша;
2. рассмотрим шашку из списка шашек, если такая есть, иначе завершим алгоритм;
3. если это простая шашка, то вызовем для нее функцию *SearchMoveChecker*, иначе вызовем функцию *SearchMoveDamka*;
4. если функция вернула значение 0, то повторим алгоритм для следующей шашки, начиная с п.2;
5. если функция вернула значение 1, то вернем кэш в сохраненное состояние;
6. запустим для текущей шашки функцию *SearchEatChecker* или *SearchEatDamka,* в зависимости от ее типа;
7. для следующей шашки повторим алгоритм, начиная с п.6;
8. если такой шашки нет, то завершим алгоритм.
9. Совершение хода и отмена хода[[6]](#footnote-7)

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. присвоим полю *Coord* текущей шашки новое значение;
5. если необходимо, изменим тип шашки.

Отмена хода выполняется в обратном порядке.

### Описание структур данных

Далее представлены основные классы, используемые в части, отвечающей за ход искусственного интеллекта.

1. Класс Checker

Класс представляет собой структуру хранения для шашки.

class Checker

{

int Color; //цвет шашки, 0-белый или 1-черный

int Type; //тип шашки, 0-шашка или 1-дамка

int Coord; //координаты шашки (3 бита – координата по x, 3 бита – координата по y)

int Num; //идентификатор шашки

int NextNum; //номер следующей шашки, для реализации списка шашек на массиве[[7]](#footnote-8)

int PrevNum; //номер предыдущей шашки

public:

Checker(): Color(0), Type(0), Coord(0), Num(0), NextNum(0), PrevNum(0) { }

~Checker() {}

void SetColor(int color);

void SetType(int type);

void SetCoord(int coord);

void SetNum(int num); //устанавливается 1 раз

int GetColor();

int GetType();

int GetCoord();

int GetNum();

//изменяют поле на противоположное значение

void ChangeColor();

void ChangeType();

//дружественные классы

friend class ListOfCheckers;//список шашек

friend class LChIterator;//итератор для перемещения по списку шашек

protected://методы, предназначенные для использования классом ListOfCheckers

void SetNextNum(int num);

void SetPrevNum(int num);

int GetNextNum();

int GetPrevNum();

};

1. Класс ListOfCheckers и класс LChIterator

Класс *ListOfCheckers* является типом, представляющим список шашек на массиве. Необходимость его создания обусловлена тем, что в программе необходимо как обращаться к шашке по индексу, так и часто удалять и добавлять шашки. Класс *LChIterator* является итератором к этому классу.

class ListOfCheckers //список шашек

{

Checker List[13]; //12 шашек + голова, чтобы список никогда не был пустым

//шашки нумеруются от 1 до 12

public:

ListOfCheckers();

~ListOfCheckers() {}

//генерирует первоначальную позицию по заданным массивам типов и координат

void GenerateInitialPosition(int color, int\* types, int\* coords, int n);

void Insert(int num); //вставляет шашку в список

void Delete(int num); //удаляет шашку из списка

int IsEmpty();//проверяет, пустой ли список

Checker& operator[] (int i); //оператор индексации

void Clean(); //очищает стек

void Bind(int n);//связывает список друг за другом

//итератор

typedef LChIterator iterator;

iterator begin() { return iterator(&(List[List[0].GetNextNum()]),List); }

iterator end() { return iterator(List, List); }

};

class LChIterator //итератор

{

Checker\* begin; //указатель на голову списка

Checker\* p; //шашка, на которую указывает итератор

public:

LChIterator() {}

//инициализатор полей

LChIterator(Checker\* p\_, Checker\* begin\_) :p(p\_), begin(begin\_) {};

//арифметические операторы

LChIterator& operator++();

LChIterator& operator--();

LChIterator& operator++(int);

LChIterator& operator--(int);

//операторы приведения типов

operator Checker\* ();

operator int();

//операторы разыменования

Checker\* operator->();

Checker operator\* (LChIterator it);

//операторы сравнения

friend int operator==(LChIterator it, Checker\* pch);

friend int operator!=(LChIterator it1, LChIterator it2);

};

1. Класс Board

Доска представляет собой одномерный массив из указателей на Checker. Это необходимо для того, чтобы, зная координату, сразу находить соответвствующую ей шашку.

class Board

{

Checker\* board[64];

public:

Board();

Board(ListOfCheckers& white, ListOfCheckers& black);//заполняет доску фигурами

void Set(ListOfCheckers& white, ListOfCheckers& black); //заполняет доску фигурами

int IsEmpty(int cell);//проверяет, пустая ли клетка

Checker\*& operator [](int cell);//индексация

void Clean();//очищает доску

};

1. Класс Move

При реализации алгоритмов выясняется, что нужно как-то хранить ход шашки. Следующий класс предоставляет все необходимые для этого поля и методы.

class Move

{

int Color;//цвет шашки, которая ходит

int StartCoord;//начальные координаты хода

int FinalCoord;//конечные координаты хода

int Num;//идентификатор перемещающейся шашки

int Type;//изменился ли тип шашки во время хода

int NEaten;//число съеденных шашек

int Eaten[12] = { 0 };//массив съеденных шашек

public:

Move() :Color(0), Type(0), StartCoord(0), FinalCoord(0), Num(0), NEaten(0) {}

//операторы сравнения

friend int operator==(const Move& m1, const Move& m2);

friend int operator!=(const Move& m1, const Move& m2);

~Move() {}

//устанавливают поля класса

void Set(int \_color, int \_s\_coord, int \_f\_coord, int \_num, int \_type);

void SetColor(int color);

void SetStartCoord(int coord);

void SetFinalCoord(int coord);

void SetNum(int num);

void SetType(int type);

void SetNEaten(int neaten);

void SetEaten(int\* arrEaten);//записывает номера съеденных шашек в массив

//возвращают значения полей класса

int GetColor();

int GetStartCoord();

int GetFinalCoord();

int GetNum();

int GetType();

int GetNEaten();

void GetEaten(int\* arrEaten);//возвращает съеденные шашки в массив arrEaten

};

1. Класс Cache

На каждом шаге рекурсии необходимо где-то хранить возможные ходы. Можно было бы создавать на каждом шаге рекурсии массив ходов, но, во-первых, это может занимать некоторое время, 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 просто записываем туда ходы. Очищение происходит посредством перемещения указателя на конец массива в сохраненную точку.

Такая структура хранения представлена следующим классом.

class Cache

{

private:

int Size;//размер кэша

Move \*pData;//указатель на начало

Move \*pLast;//указатель на конец

public:

Cache():Size(0), pData(0), pLast(0){};

Cache(int size) : Size(size);//выделяет динамически массив размера size

void Push(Move data); //запись данных в кэш

Move& Pop();//извлечение из кэша

Move\* GetpLast();//получить указатель на конец кэша

void Rollback(Move \*saved);//очищение кэша до сохраненной позиции saved

int CurPos(); //получить текущий размер кэша

Move operator[](int i);//индексирование

bool IsFull();//проверяет, заполнен ли кэш

bool IsEmpty(); //проверяет, пуст ли кэш

void Clean(); //очищает кэш полностью

~Cache();

};

1. Реализация см такое-то приложение [↑](#footnote-ref-2)
2. [↑](#footnote-ref-3)
3. [↑](#footnote-ref-4)
4. [↑](#footnote-ref-5)
5. [↑](#footnote-ref-6)
6. [↑](#footnote-ref-7)
7. См п.2 [↑](#footnote-ref-8)