МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

РЕАЛИЗАЦИЯ И СРАВНЕНИЕ АЛГОРИТМОВ ПОИСКА В ДЕРЕВЕ ПОЗИЦИЙ ИГРЫ

КУРСОВАЯ РАБОТА

студента 2 курса 251 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Синкевича Артема Александровича

Научный руководитель
доцент, к. ф.-м. н. _____ А. С. Иванова

Заведующий кафедрой
к. ф.-м. н., доцент _____ С. В. Миронов

Саратов 2021

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современном мире часто наблюдаются явления, при которых участники имеют несовпадающие интересы и продвигаются различными путями к достижению своих целей. В теории игр такие явления называются конфликтами, а частным и наиболее интересным случаем конфликта является игра. Игру можно рассматривать как упрощенную модель конфликта, которая лучше всего подходит для изучения многих практически важных задач.

Одним из простейших видов игр являются антагонистические игры, то есть игры двух и более игроков с прямо противоположными интересами. Если у игроков есть конечное число возможных действий для достижения победы, то такая игра называется конечной. Игра с полной информацией позволяет просчитать как свои действия, так и действия соперника. Типичным примером игры с полной информацией являются шахматы — игроки видят расположение на доске фигур обоих игроков, что позволяет каждому, руководствуясь логикой и правилами, угадать поведение оппонента на несколько ходов вперёд. В играх с нулевой суммой выигрыши игроков противоположны, например, 1 — победа первого игрока (проигрыш второго), 0 — ничья, -1 — выигрыш второго игрока (проигрыш первого). Другими примерами антагонистических игр с полной информацией и нулевой суммой являются шашки, крестики-нолики, го и многие другие настольные игры. Кроме того, такие игры детерминированы, то есть позиция зависит только от действий игроков, а не от каких-либо случайных событий, что позволяет всегда построить алгоритм выигрыша или гарантированной ничьей, для чего можно просчитать дерево всех возможных позиций игры.

В данной работе рассматривается игра «русские шашки» — наиболее известный вид шашек в России, правила игры в которые сходны с правилами игры в международные шашки, основное отличие — меньший размер доски (8x8). Таким образом, эта настольная игры была выбрана из-за известности и простоты правил. Несмотря на это, игра в шашки является достаточно сложной игрой — например, в английских шашках, считающихся более простыми, с доской 8x8 возможно около 10^{20} позиций. Значит невозможно создать компьютерного агента, перебирающего все возможные позиции, и требуются более сложные алгоритмы, которые позволили бы найти достаточно близкий к оптимальному ход за разумное время. На данный момент разработано множе-

ство алгоритмов, таких как алгоритм альфа-бета отсечения, NegaScout, SSS*, MTD(f) [1]. Наиболее известным из них является алгоритм (правило принятия решений) минимакс, на котором основаны многие другие алгоритмы, и который может показывать хорошие результаты при должной оптимизации для многих настольных игр.

Целью данной работы является изучение различных алгоритмов поиска в дереве позиций игры и сравнение их друг с другом по критериям времени работы, возможной глубины поиска и результативности игры.

В результате написания работы должны быть решены следующие задачи:

— создание клиент-серверного приложения для игры в шашки;
— реализация различных алгоритмов поиска в игровом дереве позиций;
— нахождение различных функций оценки позиций;
— сравнение реализованных алгоритмов, использующих созданные оценочные функции, по различным критериям.

1 Задача нахождения оптимального хода и методы её решения

Шашки, как и многие другие настольные игры, являются детерминированными играми с полной информацией и нулевой суммой, поэтому для нахождения оптимального хода из какой-либо позиции можно составить дерево, где вершинами являются позиции игры, рёбрами — ходы игроков, и в каждой вершине указывается её значение — 1, 0 или -1, в зависимости от выигрышности данной позиции. Для листьев (конечных позиций) значение задаётся напрямую, а для остальных вершин равно 1 (или -1), если данный игрок выигрывает при любом ходе (переходе по ребру в в вершину поддерева), или 0 при ничьей. Таким образом поиск лучшего хода переформулируется как построение дерева и поиск в нём пути к вершине, соответствующей победе в игре (или ничьей) с учётом ходов другого игрока [1].

Чтобы оценить размеры такого дерева, используют коэффициент ветвления, означающий количество допустимых ходов из позиции. Так как его нельзя вычислить точно, то применяют средний коэффициент (обозначим b), и тогда количество вершин в дереве — $O(b^d)$, где d — высота дерева (то есть количество ходов). Известно, что для английских шашек с доской 8x8 $b = 2.8$ и $d = 70$, для международных с доской 10x10 — $b = 4$ и $d = 90$, а для шахмат — $b = 35$ и $d = 70$ [2]. Из данных чисел можно сделать вывод, что построить или перебрать всё дерево игры за разумное время невозможно.

Возможным решением является введение функции оценки позиции — такой функции, которая сопоставляет позиции число, соответствующее предсказываемой выигрышности для данного игрока. Тогда не требуется перебирать всё дерево, а можно ограничиться определённой максимальной глубиной, и значения вершин на этой глубине считать с помощью оценочной функции. От точности функции и глубины поиска будет зависеть, насколько близко к оптимальному будет найденный ход. Данный подход используется в алгоритмах, основанных на правиле минимакса, которые будут рассмотрены далее.

1.1 Минимакс

Минимакс (Minimax) — правило принятия решений, используемое в теории игр, теории принятия решений, статистике и философии для минимизации максимально возможных потерь в худшем случае. Для игр множества игроков можно определить два значения: максимин и минимакс.

Значение максимина — наибольший результат, который может получить игрок, не зная действий других игроков, то есть это наименьший результат, к которому могут привести ходы других игроков, если они будут знать действия текущего игрока [3]. Это значение можно записать как

$$\underline{v_i} = \max_{a_i} \min_{a_{-i}} v_i(a_i, a_{-i}), \tag{1}$$

где i — индекс текущего игрока, $-i$ — остальные игроки, a_i — действие текущего игрока, a_{-i} — действия других игроков, v_i — оценочная функция. Из формулы следует, что учитываются худшие комбинации ходов противников (минимум) и лучшие действия игрока в таких случаях (максимум).

Значение минимакса — наименьший результат, к которому могут привести ходы других игроков, если они не будут знать действия текущего игрока, то есть наибольший результат, который может получить игрок, зная действия других игроков. Это значение можно записать как

$$\overline{v_i} = \min_{a_{-i}} \max_{a_i} v_i(a_i, a_{-i}), \tag{2}$$

где i, $-i$, a_i, a_{-i} такие же, как и для максимина. По формуле видно, что значение минимакса отличается от максимина порядком операторов максимума и минимума, то есть рассматриваются лучшие действия игрока и худшие комбинации ходов соперников для них.

Значение максимина не больше минимакса, то есть $\underline{v_i} \leqslant \overline{v_i}$, что видно из порядка операторов в соответствующих формулах. Также это следует из того, что максимин — максимальный результат игрока, ещё не знающего действий других игроков, в отличие от минимакса, где игрок знает ходы соперников. Кроме того, для игр двух игроков с нулевой суммой фон Нейманом доказано, что максимин равен минимаксу [4]. Также для таких игр выполняется равенство

$$\max(a, b) = -\min(-a, -b), \tag{3}$$

где a, b — значения оценочной функции для двух позиций и одного из игроков. Это свойство используется для более простой реализации алгоритма минимакса — алгоритма негамакс (negamax).

Общий алгоритм минимакса для двух игроков заключается в рекурсив-

ном спуске по дереву игровых позиций, пока не будет достигнуто ограничение по глубине или позиция будет конечной. В таких случаях алгоритм возвращает значение функции оценки этой позиции. Иначе возможны два случая:

— максимизируется результат игрока, тогда рекурсивно вычисляются значения для всех позиций, в которые игрок может перейти из данной, и возвращается максимум из них;

— минимизируется результат игрока, тогда рекурсивно вычисляются значения для всех позиций, в которые игрок может перейти из данной, и возвращается минимум из них.

Такой алгоритм запускается для заданной позиции и игрока, результат которого требуется максимизировать, и возвращает значение минимакса. Также, перед завершением алгоритма, можно сохранить лучший ход, максимизирующий результат игрока, что будет использоваться в реализации компьютерного агента (бота) с данным алгоритмом.

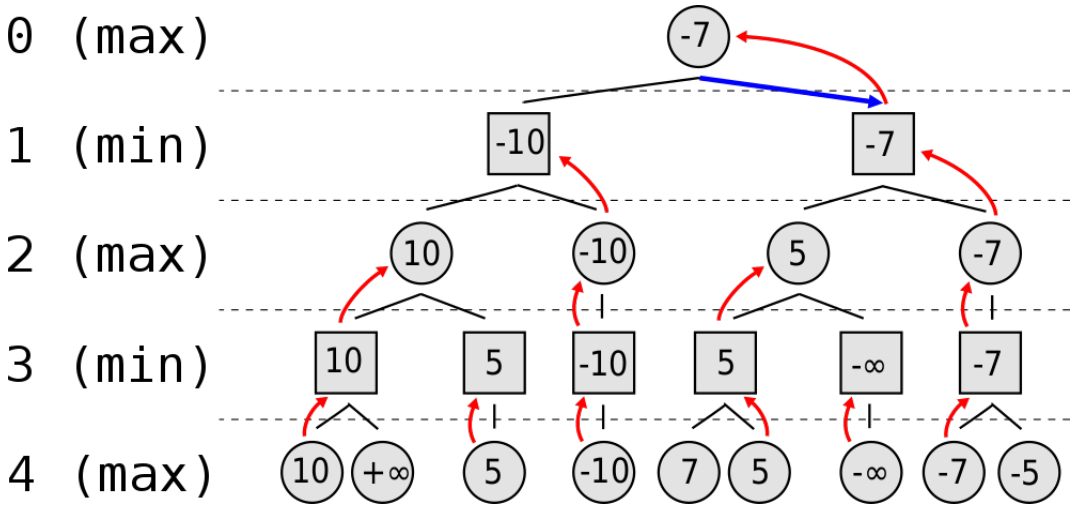

Рисунок 1 – Пример работы алгоритма минимакс

Рассмотрим работу алгоритма на примере рисунка 1. Кругами показаны позиции, из которых ходит игрок, для которого производится максимизация, а квадратами — позиции, из которых ходит противник, для которого производится минимизация результата. Так как максимизируется результат игрока, для которого запущен алгоритм, то на чётных уровнях (0, 2, 4) позиции обозначены кругами, на нечётных — квадратами. Как можно видеть, в данном случае ограничение по глубине — 4, и значения в листьях определены с помощью какой-либо оценочной функции. Также для каждой позиции красной стрелкой показан ход, выбранный с помощью соответствующего оператора (максимум

или минимум) для этого уровня. Кроме того, синей стрелкой показан ход, выбранный лучшим для игрока. Такой ход минимизирует максимальные потери.

В программах ботов, использующих алгоритм минимакс (`miniMaxWeak` и `miniMax`), реализована версия негамакс (код приведён в приложении A). В ней не требуется отдельно рассматривать случай максимизируемого и минимизируемого игроков, а достаточно изменить знак у вычисленных рекурсивно значений, и всегда выбирать максимум, по свойству 3. В данном коде `PlayerColor` — цвет игрока (белый или чёрный), `Move` — класс хода игрока, `Board` — класс доски. С функциями классов и другими деталями реализации можно ознакомиться в разделе 2. В данном коде используется оценочная функция `EvalPieceCount`, реализованные функции оценки описаны в разделе 1.5.

1.2 Альфа-бета отсечение

Альфа-бета отсечение — алгоритм поиска, оптимизация, уменьшающая количество вершин дерева игры, рассматриваемых в процессе работы алгоритма минимакса. Данный алгоритм не рассматривает ходы из позиции, если известно, что их результат будет хуже, чем у уже рассмотренных ходов. При применении на одном и том же дереве алгоритм альфа-бета отсечения получит тот же результат, что и алгоритм минимакса, но, в среднем, за меньшее время.

Альфа-бета отсечение изобреталось многократно разными авторами: Ричардс, Харт, Левин, Эдвардс независимо предлагали ранние версии [5], Брудно независимо придумал алгоритм альфа-бета отсечения, опубликовав свои результаты в 1963 г. [6]. Дональд Кнут и Рональд Мур улучшили алгоритм в 1975 г. [7]

Алгоритм поддерживает два значения — α и β — представляющих минимальное значение, которое может получить максимизируемый игрок, и максимальное значение, которое может получить минимизируемый игрок. Изначально $\alpha = -\infty$ и $\beta = +\infty$ (худшие возможные значения). Когда максимальное значение, получаемое минимизируемым игроком, становится меньше, чем минимальное значение, получаемое максимизируемым игроком, то есть $\beta < \alpha$, то дальнейшие ходы из позиции рассматривать не имеет смысла [8].

Точнее, алгоритм выполняет рекурсивный спуск по дереву игровых позиций, аналогичный алгоритму негамакса, но при рекурсивном переходе в вершину передаёт $\alpha' = -\beta$, $\beta' = -\alpha$ (изменение знака для другого игрока), и после рассмотрения этого хода обновляет α максимумом из текущего

значения и значения рассмотренной вершины. Если окажется, что $\alpha \geqslant \beta$, то рассмотрение ходов для данной позиции завершается.

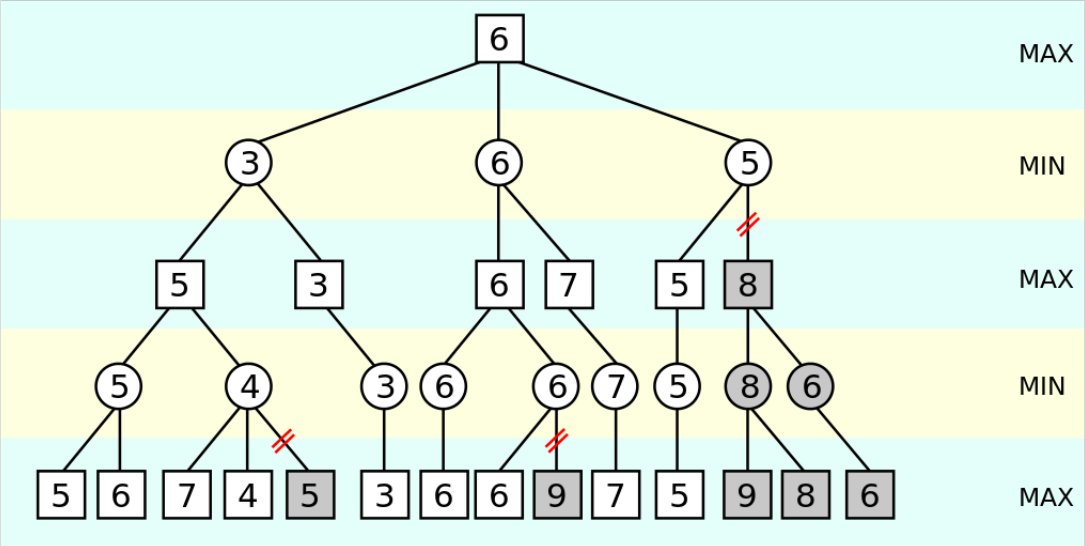

Рисунок 2 – Пример работы альфа-бета отсечения

Рассмотрим работу алгоритма на примере рисунка 2. Аналогично алгоритму минимакса, на каждом уровне выбирается максимальный или минимальный результат. Но некоторые поддеревья отсекаются (обозначены серым). Рассмотрим вершину со значением 5 (самая левая на уровне 2, при индексации, как на рис. 1). Так как на момент начала её обработки не было полностью рассмотрено ни одного хода ни у неё, ни у предков, то изначальный её результат (обозначим res) — $-\infty$, и $\alpha = -\infty$, $\beta = +\infty$. Пусть обработан левый ребёнок вершины, значение которого равно -5 (так как расположен на уровне минимизации), тогда новые значения вершины: $res = 5$, $\alpha = 5$, $\beta = +\infty$. Далее будет рассмотрен правый ребёнок, для которого алгоритм будет запущен с параметрами $\alpha = -\infty$, $\beta = -5$. После обработки первых двух ходов (значения 7 и 4) res вершины станет равным -4, и $\alpha = -4$, $\beta = -5$. Значит $\alpha \geqslant \beta$, и обработка данной вершины завершается. Очевидно, что минимизируемый игрок сможет добиться хотя бы результата 4, но у максимизируемого игрока уже есть ход, позволяющий получить результат 5, а значит дальнейшая обработка ходов из вершины не требуется.

Возможны варианты реализации альфа-бета отсечения, основанные на минимаксе или негамаксе. Также, различают fail-soft и fail-hard версии алгоритма: fail-soft реализация может возвращать значения минимакса, выходящие за границы (то есть меньше α или больше β), а при fail-hard реализации

значения всегда ограничены отрезком $[\alpha; \beta]$. В приложении Б приведена fail-soft реализация, основанная на алгоритме негамакса из предыдущего раздела. Можно заметить, что в реализации было достаточно добавить параметры α и β, обновление α и отсечение при $\alpha \geqslant \beta$.

1.3 NegaScout

NegaScout (или PVS — Principal Variation Search, поиск основного варианта) — алгоритм поиска, оптимизация альфа-бета отсечения; NegaScout в процессе работы не рассматривает позиции, которые не были бы обработаны алгоритмом альфа-бета отсечения, то есть является оптимизацией, как и альфа-бета отсечение по отношению к алгоритму минимакса.

Большинство ходов не допустимы для обоих игроков, поэтому не требуется полный обход дерева для вычисления точного результата. Он нужен только для основного варианта — последовательности ходов, оптимальной для обоих игроков. Поэтому, при рассмотрении ходов из позиции, кроме первого, проверяется, может ли он быть в основном варианте. Для этого выполняется запуск алгоритма с окном $\alpha' = -\alpha - 1$, $\beta' = -\alpha$ (при таких ограничениях будет значительно больше отсечений) и, если получен результат, не увеличивающий α, то можно считать, что ход неоптимален, и не требуется проводить полный обход поддерева. Иначе, если для результата (res) выполняется $\alpha < res < \beta$, то вариант, включающий данный ход, более оптимален, чем предыдущий основной вариант, и требуется обход поддерева с обычным окном ($\alpha' = -\beta$, $\beta' = -res$). Подробнее алгоритм описан в [9].

Время работы данного алгоритма сильно зависит от порядка ходов. Если у многих позиций первый рассматриваемый ход будет оптимальным, то остальные ходы будут быстро отсечены из-за «нулевого» окна. Иначе, при случайном порядке рассмотрения ходов, алгоритм может работать медленнее, чем обычное альфа-бета отсечение из-за необходимости повторных обходов поддеревьев.

В приложении В приведена реализация, основанная на алгоритме альфа-бета отсечения из предыдущего раздела. Можно заметить, что в реализации была добавлена переменная `first` для обработки первого хода и вызов алгоритма с нулевым окном для остальных ходов.

1.4 Таблица транспозиций

Алгоритму минимакса и другим, основанным на нём, во время работы требуется анализировать миллионы позиций, которые могут повторяться. Так как выполняется обход дерева в глубину, то информация о всех позициях не сохраняется. Во многих играх можно достичь определённой позиции разными способами; последовательности ходов, приводящие к одной позиции называются транспозициями. Для того, чтобы избежать многократной обработки одной и той же позиции, используются таблицы транспозиций — хеш-таблицы, в которых для некоторых позиций хранятся значения минимакса [10]. Перед обработкой позиции алгоритм минимакса может проверить её наличие в таблице транспозиций, и использовать уже вычисленное значение за амортизированное константное время. Кроме того, сохранённое значение позволяет не обрабатывать не только данную позицию, но и всё её поддерево, что позволяет достичь значительного ускорения (в шахматах до 50% в сложных позициях в середине игры, и до 5 раз в конце игры, сообщается в [11]). Но из-за ограничения поиска по глубине можно использовать значение из таблицы только в том случае, если оно было вычислено при большей глубине, чем сейчас, иначе будут пропущены какие-либо вершины из поддерева позиции. Помимо этого, размер оперативной памяти современных компьютеров не позволяет хранить все позиции, которые могут быть обработаны процессором за разумное время, поэтому можно хранить только их часть. Например, можно оставлять только наиболее часто встречающиеся позиции, а редкие — убирать, то есть таблица транспозиций будет представлять собой вид кеша.

Использование таблиц транспозиций с алгоритмом альфа-бета отсечения затруднено тем, что некоторые ветви отсекаются из-за рассмотренных ранее позиций, поэтому не всегда можно напрямую повторно использовать значение из таблицы. Для решения этой проблемы вместе со значением позиции сохраняется тип этого значения: точная оценка, оценка снизу, оценка сверху. Таким образом, точная оценка позволяет использовать это значение в качестве результата, оценка снизу — обновить α (максимумом), а оценка сверху — обновить β (минимумом). В последних двух случаях потребуется дальнейшая обработка позиции, но она будет ускорена уменьшенным альфа-бета окном. После того, как значение позиции вычислено, оно должно быть сохранено в таблице транспозиций (если в ней есть место): если результат не больше

изначального α (до обновления с помощью таблицы), то это — оценка сверху, если результат не меньше β — оценка снизу, в противном случае это — точная оценка.

В приложении Г приведена реализация алгоритма NegaScout с таблицей транспозиций, в качестве которой использовался класс `Dictionary`, реализующий хеш-таблицу. Для компактного хранения позиции в качестве ключа таблицы используется пара из 64-битного и 32-битного числа. Так как доска состоит из 64 клеток, из которых используются только 32 (тёмные), то для хранения бинарного значения каждой клетки достаточно 32 бит. Первое число содержит информацию о том, является ли каждое поле пустым, обычной шашкой или дамкой (два бинарных значения — является ли шашкой, является ли дамкой). Второе число содержит цвет каждой шашки. В качестве значения таблицы используется класс `TableEntry`, который хранит глубину, на которой было вычислено значение, тип оценки и само значение. Количество хранимых позиций ограничено константой $MAX\_TABLE\_COUNT = 2 \cdot 10^7$.

1.5 Оценочные функции

Оценочная функция — функция, позволяющая приближённо численно выразить вероятность игрока победить из данной позиции. Очевидно, что для составления качественной функции оценки потребуется проанализировать правила игры, для которой она разрабатывается, в данной работе — русские шашки.

1.5.1 Правила игры

Основные правила игры в русские шашки:

1. Игра ведётся на доске 8x8, у каждого игрока по 12 шашек, занимающих на тёмных полях по три ряда с каждой стороны доски.
2. Шашка ходит по диагонали вперёд на одну клетку.
3. Дамка ходит по диагонали на любое количество клеток в любую сторону, но не может перепрыгивать свои шашки или дамки.
4. Если какая-либо шашка или дамка может осуществить взятие, то его нужно выполнить.
5. При взятии обычная шашка перепрыгивает через шашку противника на следующее за ней свободное поле (оно необходимо для взятия); шашка противника удаляется с доски после завершения хода.

6. При взятии дамка перепрыгивает на любое свободное поле, следующее за шашкой противника.
7. При достижении обычной шашкой последнего ряда она превращается в дамку.

В данной работе считается, что игрок побеждает, если у его соперника побиты все шашки или нет ходов, и наступает ничья, если проведено 200 ходов и не была достигнута победа какого-либо из игроков. С полными правилами игры можно ознакомиться в [12].

1.5.2 Функция EvalPieceCount

Из условия победы очевидно, что чем больше шашек у одного игрока и чем меньше у другого, тем больше вероятность первого игрока победить. Таким образом, простейшая оценочная функция вычисляет разницу количеств шашек у игроков. Далее можно заметить, что дамка обладает большими возможностями, чем обычная шашка, и её стоит оценивать дороже. Например, можно взять стоимость обычной шашки как 1, а дамки — 2, и модифицировать вычисление разницы так, чтобы учитывалась цена шашки. Такая функция оценки реализована в боте `minMaxWeak` и названа `EvalPieceCount`. Её код представлен ниже. Для того, чтобы бот не зацикливался на одном и том же ходе, добавляется небольшое случайное число: в данной реализации результат функции умножается на 256 и добавляется случайный байт.

```
private int EvalPieceCount(PlayerColor currPlayer)
{
    int result = 0;
    for (int row = 0; row < Board.SIZE; row++)
        for (int col = 0; col < Board.SIZE; col++)
        {
            if (board.PiecesType[row, col] == PieceType.Empty)
                continue;
            // Обычная шашка: 1, дамка: 2
            int res = (board.PiecesType[row, col] == PieceType.Normal) ? 1 :
            ↪  2;
            // Разность сумм значений шашек текущего игрока и противника
            result += board.PiecesColor[row, col] == currPlayer ? res : -res;
        }
    return (result * 256) + random.NextByte();
}
```

1.5.3 Функция EvalPieceRow

Для дальнейшего улучшения функции оценки можно заметить, что чем ближе шашка к последнему (для игрока) ряду, тем больше у неё вероятность стать в будущем дамкой — более ценной фигурой. Чтобы это использовать, можно считать стоимость обычной шашки не константой, а суммой постоянного числа и числа рядов до последнего. Такая функция оценки названа в коде `EvalPieceRow` и реализована в боте `minMax` и других, основанных на нём. Код функции представлен ниже. Сравнение реализованных оценочных функций произведено в разделе 1.6.

```
private int EvalPieceRow(PlayerColor currPlayer)
{
    int result = 0;
    for (int row = 0; row < Board.SIZE; row++)
        for (int col = 0; col < Board.SIZE; col++)
        {
            int res = 0;
            // Обычная шашка: 5 + количество рядов от начала доски до шашки
            if (board.PiecesType[row, col] == PieceType.Normal)
                res = 5 + ((board.PiecesColor[row, col] == PlayerColor.White)
                ↪  ? row : (Board.SIZE - 1 - row));
            // Дамка: 15
            else if (board.PiecesType[row, col] == PieceType.King)
                res = 7 + Board.SIZE;
            // Разность сумм значений шашек текущего игрока и противника
            result += board.PiecesColor[row, col] == currPlayer ? res : -res;
        }
    return (result * 256) + random.NextByte();
}
```

1.6 Сравнение алгоритмов

1.6.1 Оценка сложности алгоритмов

Пусть коэффициент ветвления дерева позиций игры равен b, а ограниченная высота дерева — d. Так как алгоритм минимакса обрабатывает все вершины дерева, то его сложность в лучшем, в среднем, в худшем случае равна $O(b^d)$.

Так как алгоритм альфа-бета отсечения является оптимизацией алгоритма минимакса, то в худшем случае (при невозможности отсечь ни одно поддерево) его сложность будет совпадать со сложностью минимакса, то есть

$O(b^d)$. В лучшем случае, когда ходы из каждой позиции отсортированы так, что первый — всегда лучший, сложность будет составлять $O(b^{d/2})$, потому что для одного из игроков потребуется просматривать все возможные ходы, а для другого — только лучший. Если же порядок рассматриваемых ходов случайный, то сложность оценивается как $O(b^{3d/4})$ (как показано в [8]).

1.6.2 Сравнение оценочных функций

Все сравнения в этом разделе выполнены на системе с процессором Ryzen 5 4600H (4 ГГц) и операционной системой Windows 10. Для каждой пары ботов с фиксированными глубинами поиска запускалось последовательно 3 игры, и был взят последний результат. Так как все боты детерминированы при фиксированных начальных значениях генераторов случайных чисел, то результаты этих 3 игр совпадают, а время работы отличается незначительно. В представленных ниже таблицах +1 означает победу белого игрока, -1 — чёрного; времена работы указаны в миллисекундах.

Для оценки выигрышности бота с алгоритмом минимакса был создан бот `random`, выполняющий случайные ходы. При сравнении бота `miniMaxWeak` с глубиной поиска 1 со случайным ботом (который играл за чёрного игрока) оказалось, что первый выигрывал в большинстве случаев, а при увеличении глубины до 2 бот с алгоритмом минимакса выигрывал всегда.

Для сравнения качества оценочных функций было проведено сравнение ботов `miniMaxWeak` (с функцией `EvalPieceCount`, при глубине поиска 4 и 6) и `miniMax` (с функцией `EvalPieceRow`, при глубине поиска 1-8). Результаты представлены в таблицах 1 и 2. Можно видеть, что вторая оценочная функция позволяет добиться немного лучших результатов: `miniMax` с глубиной 4 выигрывает против `miniMaxWeak` с глубиной 4, и `miniMax` с глубиной 5 играет вничью с `miniMaxWeak` с глубиной 6. Поэтому она используется во всех ботах, кроме `random` и `miniMaxWeak`.

1.6.3 Сравнение времени работы

Для сравнения времени работы алгоритмов было проведено попарное сравнение ботов, их реализующих: `alphaBeta` и `miniMax` с глубиной 7 (таблица 3), `alphaBeta` и `alphaBeta` с глубиной 9 (таблица 4), `negaScout` и `alphaBeta` с глубиной 7 (таблица 5), `negaScout` и `negaScout` с глубиной 9 (таблица 6), `negaScoutTransposition` и `negaScoutTransposition` с глубиной 9 (таблица

Таблица 1 – Сравнение ботов miniMax (глубина 1-8, белый игрок) и miniMaxWeak (глубина 4, чёрный игрок)

Глубина	Результат	Ср. время бота №1	Макс. время бота №1	Ср. время бота №2	Макс. время бота №2
1	-1	0,17	10	0,79	14
2	-1	0,15	10	1,09	14
3	-1	0,2	11	0,71	15
4	+1	2,64	16	1,76	14
5	+1	7,2	36	1,31	15
6	+1	18,2	90	0,67	13
7	+1	88,1	579	0,86	15
8	+1	275,7	1250	0,56	14

Таблица 2 – Сравнение ботов miniMax (глубина 1-8, белый игрок) и miniMaxWeak (глубина 6, чёрный игрок)

Глубина	Результат	Ср. время бота №1	Макс. время бота №1	Ср. время бота №2	Макс. время бота №2
1	-1	0,18	10	32,3	153
2	-1	0,19	10	18,2	73
3	-1	0,2	11	24,8	169
4	-1	2,68	15	279,2	1071
5	0	5,03	33	104,6	198
6	+1	39,9	343	38	234
7	+1	79,7	339	12,7	55
8	+1	701,5	4241	25,6	120

7). С помощью этих таблиц были составлены графики среднего и максимального времени на ход: 3 и 4. Из них видно, что:

1. Альфа-бета отсечение позволяет значительно ускорить алгоритм минимакса: в данной реализации бот `alphaBeta` при глубине поиска 9-10 показывает время работы, сравнимое с ботом `miniMax` с глубиной поиска 7-8.
2. Из-за дополнительных обходов NegaScout немного медленнее, чем алгоритм альфа-бета отсечения при небольшой глубине поиска, но при глубине 10 показывает лучшие результаты.
3. Таблица транспозиций достаточно ускоряет NegaScout, что позволяет увеличить глубину поиска с 10 до 11.

Таким образом, все реализованные оптимизации позволили увеличить глубину

поиска с 8 до 11, что позволяет добиться довольно высокого уровня игры.

Таблица 3 – Сравнение ботов alphaBeta (глубина 1-10, белый игрок) и miniMax (глубина 7, чёрный игрок)

Глубина	Результат	Ср. время бота №1	Макс. время бота №1	Ср. время бота №2	Макс. время бота №2
1	-1	0,2	10	87,4	476
2	-1	0,17	10	103,2	530
3	-1	0,18	10	425,2	3169
4	-1	0,43	13	146,5	1125
5	-1	3,09	18	581	3187
6	-1	1,72	26	28,1	224
7	-1	16,5	97	248,8	2076
8	+1	35,62	127	78,1	343
9	+1	229,1	1025	326,6	2062
10	+1	1088,8	7307	214,5	982

Таблица 4 – Сравнение ботов alphaBeta (глубина 1-10, белый игрок) и alphaBeta (глубина 9, чёрный игрок)

Глубина	Результат	Ср. время бота №1	Макс. время бота №1	Ср. время бота №2	Макс. время бота №2
1	-1	0,21	10	67	224
2	-1	0,14	9	45,9	214
3	-1	0,13	10	62,1	283
4	-1	0,22	11	45,1	282
5	-1	0,92	15	56,2	208
6	-1	1,69	18	37,8	210
7	-1	9,15	38	82,9	436
8	-1	26,2	91	92	403
9	0	536,7	1085	595,9	1588
10	+1	208	973	70	289

Таблица 5 – Сравнение ботов negaScout (глубина 1-10, белый игрок) и alphaBeta (глубина 7, чёрный игрок)

Глубина	Результат	Ср. время бота №1	Макс. время бота №1	Ср. время бота №2	Макс. время бота №2
1	-1	0,23	10	18,7	121
2	-1	0,2	10	22,6	203
3	-1	0,25	10	19,3	60
4	-1	0,55	12	9,9	42
5	-1	1,63	14	9,5	51
6	-1	4,11	22	13,9	93
7	-1	16	122	13,6	70
8	+1	22,8	98	11	58
9	+1	143,3	729	10,1	53
10	+1	609,5	5324	12,2	78

Таблица 6 – Сравнение ботов negaScout (глубина 1-10, белый игрок) и negaScout (глубина 9, чёрный игрок)

Глубина	Результат	Ср. время бота №1	Макс. время бота №1	Ср. время бота №2	Макс. время бота №2
1	-1	0,2	10	107	393
2	-1	0,21	10	95,4	413
3	-1	0,24	11	106,8	467
4	-1	0,32	12	64,3	648
5	-1	1,8	14	99,2	422
6	-1	2,4	24	69,1	485
7	-1	22	115	174,8	1043
8	-1	35,4	407	79,1	588
9	0	619	1875	489,5	1146
10	+1	198	883	76,3	366

Таблица 7 – Сравнение ботов negaScoutTransposition (глубина 1-11, белый игрок) и negaScoutTransposition (глубина 9, чёрный игрок)

Глубина	Результат	Ср. время бота №1	Макс. время бота №1	Ср. время бота №2	Макс. время бота №2
1	-1	0,26	14	86,6	450
2	-1	0,27	14	25,3	94
3	-1	0,22	14	93,5	499
4	-1	0,54	16	77,3	467
5	-1	1,52	18	79,8	242
6	-1	5,07	32	100	637
7	-1	14	57	78,6	336
8	-1	39,1	173	94,2	662
9	0	178	784	247,2	1692
10	+1	161,1	587	62,5	233
11	+1	680,6	2953	108,7	604

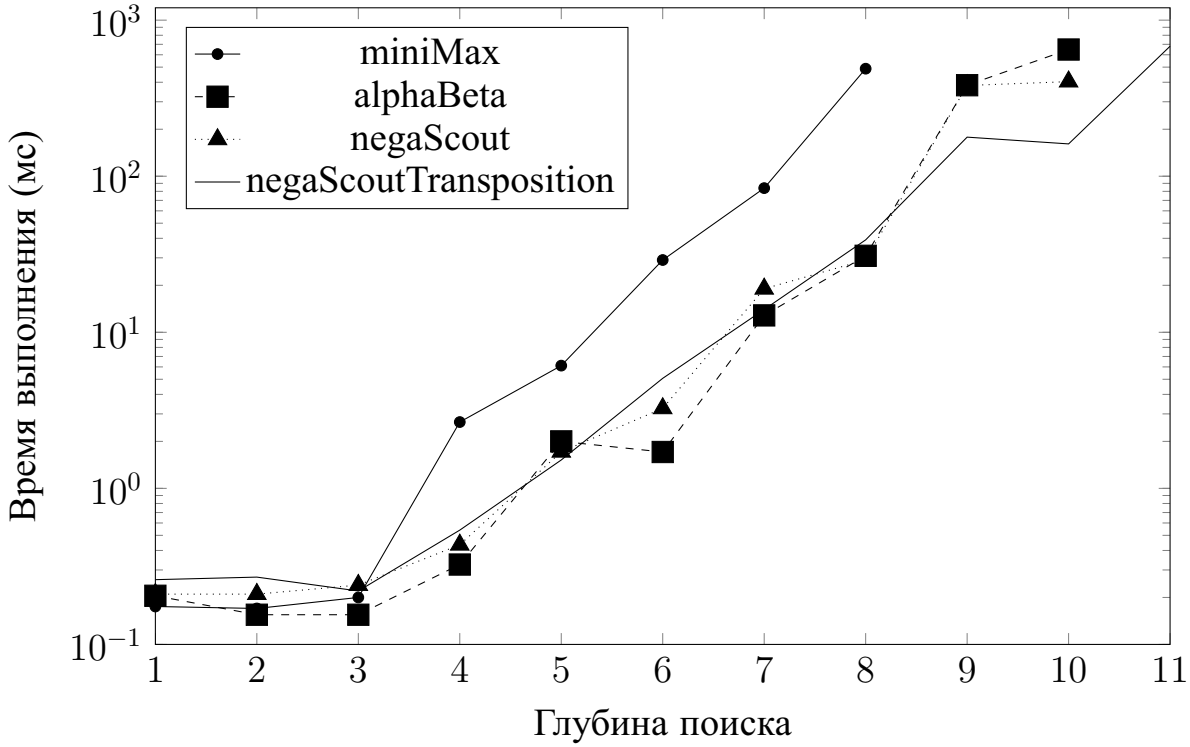

Рисунок 3 – Среднее время работы ботов в зависимости от глубины поиска

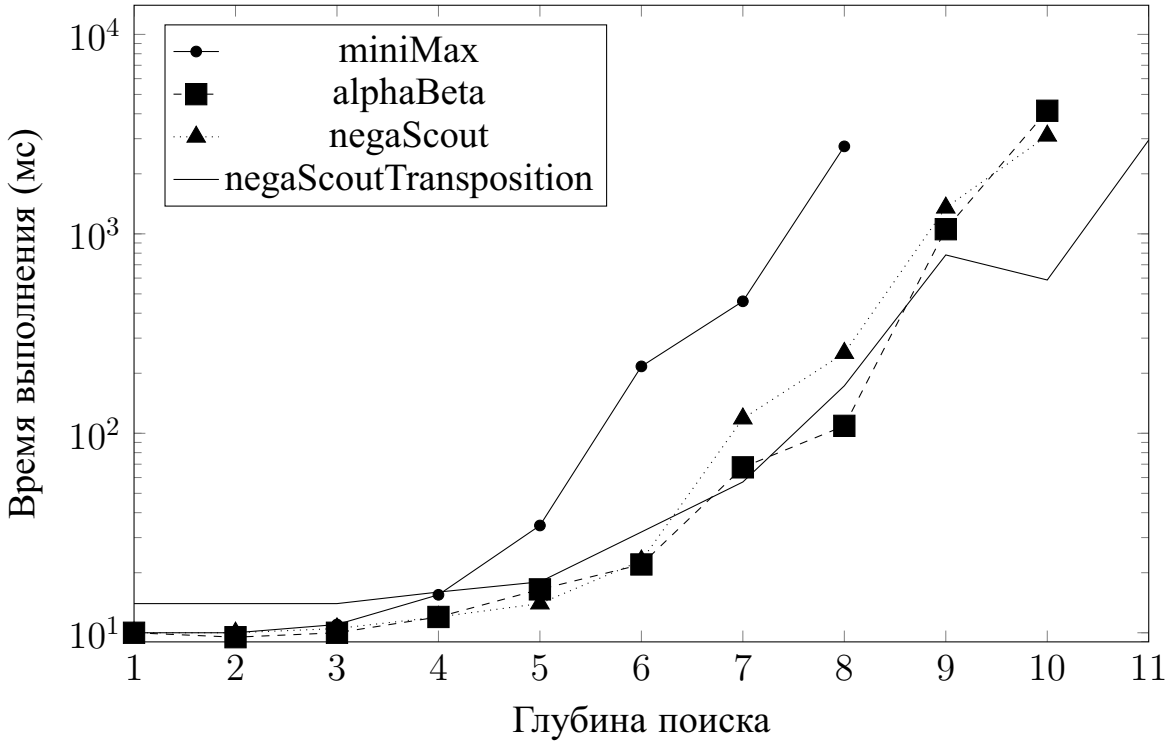

Рисунок 4 – Максимальное время работы ботов в зависимости от глубины поиска

2 Детали реализации

Проект состоит из нескольких частей: общей библиотеки `CheckersLib`, содержащей основные классы (для доски, хода, клиента); клиента с графическим интерфейсом `CheckersClient` (`CheckersClient.Windows` для ОС Windows и `CheckersClient.Gtk` для ОС, основанных на Linux); сервера игры `CheckersServer`; клиента-бота `CheckersBot`, который содержит всех приведённых выше ботов; и `CheckersBotTest` — небольшой программы для сравнения ботов. Все программы написаны на языке C# и используют платформу .NET Core 3.1. Для взаимодействия по сети используется библиотека WebSocketSharp, реализующая протокол WebSocket.

2.1 CheckersLib

Общая библиотека содержит 3 класса, используемых в других программах:

2.1.1 Board

`Board` — класс доски размер 8x8, где размер задаётся константой `SIZE`. Каждое поле описывается с помощью двух двумерных массивов `PiecesType` и `PiecesColor`. Первый задаёт тип шашки в каждом поле (пустое поле — `Empty`, обычная шашка — `Normal`, дамка — `King`). Второй задаёт цвета шашек (`Black` или `White`).

У класса три конструктора: первый может создавать либо пустую доску, либо с начальной расстановкой, второй создаёт из строки определённого формата, третий копирует другую доску. Также имеются вспомогательные методы для обновления доски другой доской и определения цвета клетки. Для работы ботов добавлены методы `DoMove` и `UndoMove`, позволяющие выполнить или отменить ход, не создавая новую доску. Также есть методы `GetMoves`, `GetMovesDFS` и `CanCapture`: первые два выполняют поиск в глубину и позволяют найти все возможные ходы для заданного игрока из определённой позиции, а второй — проверяет, может ли заданный игрок совершить взятие определённой шашкой. Эти методы используются в `CanPlayerMove` и `CanPlayerCapture` — методах, проверяющих все позиции для определённого игрока. Для компактного хранения доски используются методы `ToString` и `GetHash`. Первый возвращает строку, которая однозначно задаёт доску и используется для пересылки по сети, а второй возвращает пару чисел, которая

используется в качестве ключа в таблице транспозиций (описано в разделе 1.4).

2.1.2 Move

`Move` — класс, описывающий ход игрока. Для этого хранится список пройденных полей `Cells`, список побитых за ход шашек `Used`, `PosKing` — номер поля в списке, когда шашка становится дамкой, и `Player` — цвет игрока, совершающего ход. Помимо этого, в классе есть конструкторы, создающие объект по перечисленным выше параметрам, из строки или из другого хода. Для преобразования хода в строку созданы два метода: `ToString` — создаёт более компактную строку для передачи по сети, и `ToHumanReadableString` — для отображения хода в клиенте с графическим интерфейсом.

2.1.3 BaseClient

`BaseClient` — базовый класс, описывающий сетевого клиента-игрока, который может быть человеком или ботом. Данный класс хранит цвет игрока `Color`, состояние клиента `ClientState` (`NotConnected` — не подключен, `NotParticipating` — подключен, но не участвует в игре, `ThisPlayerMove` — ход данного игрока, `OtherPlayerMove` — ход соперника), а также `webSocket` для связи по сети с сервером и `userName` — имя игрока на сервере. Данный класс обрабатывает сообщения от сервера, получая их с помощью метода `WebSocketOnMessage` и вызывая нужные события, которые обрабатываются в данном классе или его наследнике. Например, есть события связи с сервером (`WebSocketOnError` — ошибка соединения, `WebSocketOnClose` — отключение от сервера), подключения к серверу (`JoinAccepted` — подключение удалось, `JoinRejected` — подключение отвергнуто), выбора цвета игрока (`SetWhiteAccepted`, `SetWhiteRejected`, `SetBlackAccepted`, `SetBlackRejected`), получения информации о игре от сервера (`PlayerListReceived` — получен список игроков, `MoveReceived` — получен совершённый ход, `BoardReceived` — получена обновлённая доска), запроса хода от игрока (`ThisPlayerMoveRequested`, `OtherPlayerMoveRequested`), обновления состояния игры (`WhiteWon` — белый игрок выиграл, `BlackWon` — чёрный игрок выиграл, `Draw` — ничья, `GameInterrupted` — игра прервана). В данном классе добавлены обработчики событий для обновления состояния и отправки информации (выбора цвета, хода) на сервер.

2.2 CheckersClient

Для игры между двумя людьми, между человеком и ботом, или наблюдения за игрой между двумя ботами создан клиент с графическим интерфейсом, использующий кроссплатформенный фреймворк Eto.Forms.

2.2.1 Интерфейс

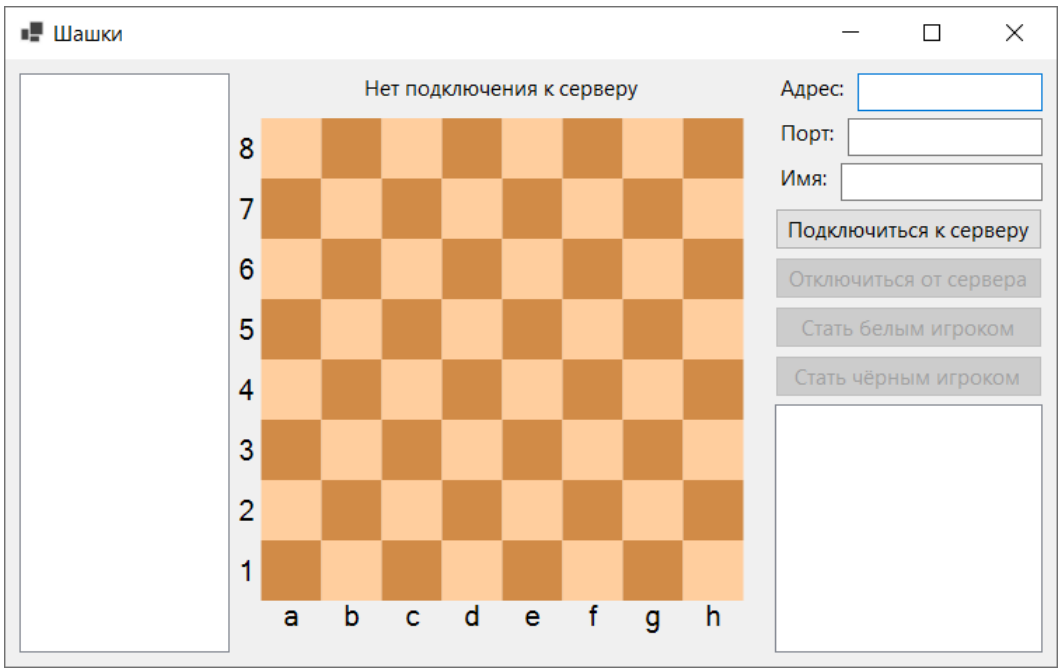

Рисунок 5 – Окно клиента после запуска

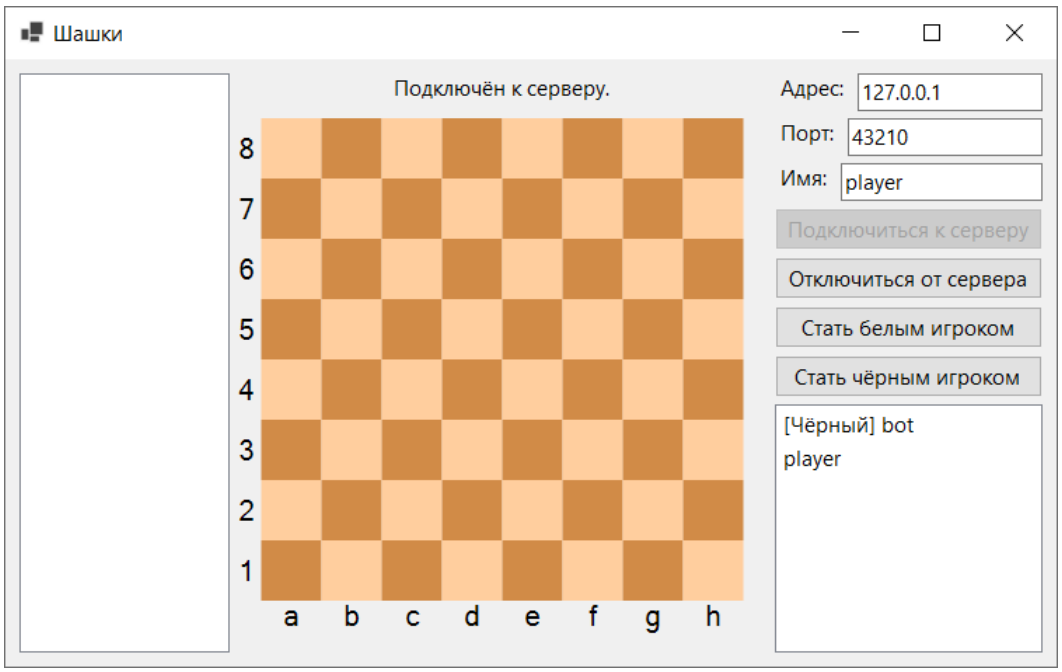

Рисунок 6 – Окно клиента после подключения к серверу

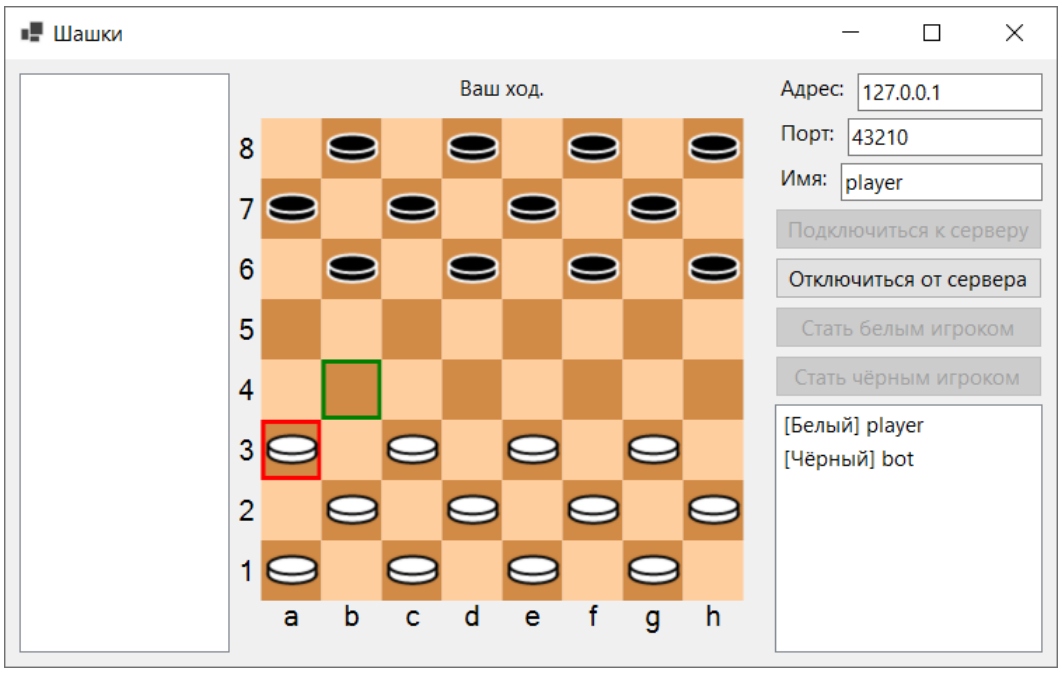

Рисунок 7 – Окно клиента при совершении хода

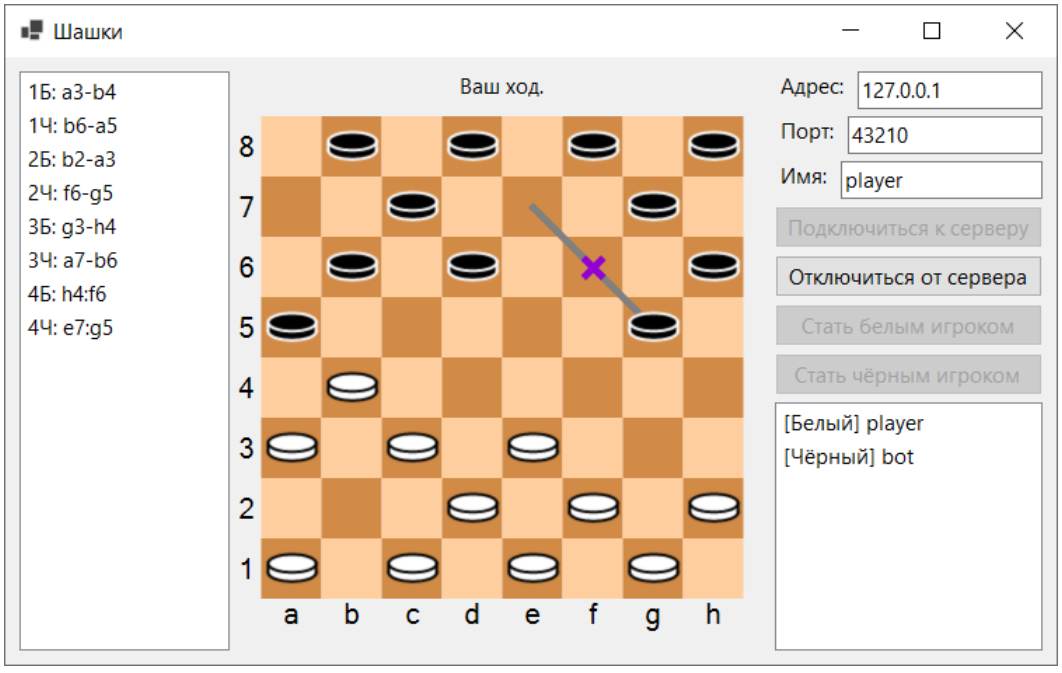

Рисунок 8 – Окно клиента при демонстрации хода соперника

Приложение состоит из одного окна (рисунок 5). В правой части находятся поля «Адрес», «Порт» и «Имя», с помощью которых задаётся подключение к серверу. Адресом может быть имя домена или IP-адрес, порт должен быть целым числом, меньшим 65536, а имя пользователя должно быть строкой от 1 до 30 символов, причём на сервере не должно быть игрока с тем же именем. После ввода этих параметров выполняется подключение с помощью кнопки

«Подключиться к серверу». Если произойдёт ошибка, то появится соответствующее диалоговое окно, иначе станут активными кнопки «Отключиться от сервера», «Стать белым игроком», «Стать чёрным игроком», обновится статус подключения и список игроков, что можно видеть на рисунке 6 (к серверу уже подключён бот в качестве чёрного игрока).

После того, как два игрока (или бота), подключённых к серверу, выберут цвета, начнётся игра. Игроки ходят по очереди, и если в данный момент ходит человек, то он может сделать ход на доске, выбирая шашку и клетку (или несколько), в которую она будет перемещаться, нажатиями левой кнопки мыши (рисунок 7). Красным цветом показывается выбранная шашка, зелёным — доступные клетки, синим — уже выбранные поля. Ходы соперника также показываются на доске с помощью серой линии и фиолетовых крестов, указывающих на взятые шашки (рисунок 8). Кроме того, все ходы записываются в шашечной нотации и показываются слева от доски. После завершения игры показывается соответствующее диалоговое окно.

2.2.2 Классы

Класс `MainForm` реализует окно приложения и обрабатывает события сетевого клиента (перечисленные в описании `BaseClient`), показывая диалоговые окна, обновляя состояние кнопок, полей и доски, которая реализована классом `BoardControl`.

Класс `BoardControl` хранит состояние доски (поле `Board`) и позволяет отрисовывать её в форме и выбирать на ней ходы. Для этого класс содержит константы для цветов доски, выделенных, доступных, использованных полей, для цвета линии последнего хода и побитых шашек. Для обработки нажатий кнопки мыши и выбора хода используются переменные `clicking` (нажата ли в данный момент левая кнопка мыши), `clickingCell` (нажимаемое поле доски), `selectedCell` (выбранное поле), `availableMoves` (доступные ходы), `availableCells` (доступные поля), `usedCells` (использованные поля), `lastMove` (последний ход).

Состояние отрисовки может меняться с помощью методов `SetLastMove` (установка показываемого последнего хода), `Reset` (сброс всех доступных, выбранных, использованных полей), а сама отрисовка происходит в методе `Paint`: сначала отрисовываются поля доски (с обводкой нужного цвета), затем линия последнего хода и побитые шашки, после — все шашки на доске, и в

конце отрисовываются подписи для строк и столбцов. Для получения изображения шашки используется вспомогательный метод PaintPiece.

Обработка нажатий кнопки мыши происходит в методах MouseDown и MouseUp. Первый записывает координаты нажимаемого поля, используя вспомогательный метод GetCellCoords (для перевода координат мыши в координаты доски). Второй обрабатывает нажатие на поле, и, в зависимости от текущего состояния, оно может стать выбранным (причём может завершиться ход) или выбор будет отменён.

PlayerClient — наследник BaseClient, класс сетевого клиента-игрока, реализует паттерн Singleton («Одиночка»). Все его события обрабатываются в классе MainForm.

2.3 CheckersServer

CheckersServer — сервер для игры в шашки, к которому могут подключаться клиенты по протоколу WebSocket. По умолчанию сервер прослушивает все подключения на порту 43210. Для каждого клиента создаётся экземпляр класса ServerClient, которыми управляет класс Server.

2.3.1 Server

Класс Server хранит сессии клиентов (поле sessions), и экземпляры классов для клиентов (clients); для управления игрой хранятся идентификаторы белого и чёрного игроков (whitePlayerId и blackPlayerId), доска (board) и количество совершённых ходов (moves). Сервер обрабатывает события подключения игроков (OnClientJoin), выбора цвета (OnClientSetWhite и OnClientSetBlack), отправляет список игроков, доску, ходы всем игрокам (SendPlayerList, SendBoard и SendMove). Добавление и удаление клиентов производится с помощью методов AddClient и RemoveClient. Когда два игрока выбирают цвета, начинается игра вызовом метода StartGame. Для запроса хода используется метод RequestMove, затем ход обрабатывается с помощью OnClientMakeMove. Игра прерывется методом StopGame. В случае победы одного из игроков, ничьей или прерванной игры вызывается SendWhiteWon, SendBlackWon, SendDraw или SendGameInterrupted.

2.3.2 ServerClient

Класс `ServerClient` представляет собой серверную часть сетевого клиента. Данный класс хранит имя пользователя (`UserName`), его роль в игре (`Role`, может быть `Spectator` — наблюдатель, `Black` — чёрный игрок, `White` — белый игрок). Класс обрабатывает открытие и закрытие подключения (`OnOpen` и `OnClose`) и получает сообщения от клиента (`OnMessage`), вызывая соответствующие события (`Join`, `SetWhite`, `SetBlack`, `MakeMove`).

2.4 CheckersBot

`CheckersBot` — клиент-бот, позволяющий подключить одного из описанных выше ботов (`random`, `miniMaxWeak`, `miniMax`, `alphaBeta`, `negaScout`, `negaScoutTransposition`) к заданному серверу. Программа принимает множество аргументов (параметры сервера, игры и бота), которые можно узнать при запуске без аргументов. Программа создаёт экземпляр наследника класса `BotClient` и обрабатывает подключение к серверу и события завершения игры.

`BotClient` — наследник `BaseClient`, класс сетевого клиента-бота. Для своей работы хранит доску (`board`), генератор случайных чисел (`random`) и его начальное значение (`seed`). Для измерения среднего и максимального времени, потраченного на ход, хранятся значения `moveTimeSum` (сумма времени ходов), `moveCnt` (количество ходов) и `maxMoveTime` (максимальное время хода). Класс содержит абстрактный метод `GetMove`, который должен возвращать ход, совершаемый ботом; он реализуется в наследниках класса. Этот метод вызывается в `OnThisPlayerMoveRequested` (при запросе хода данного игрока), где измеряется время его выполнения.

2.5 CheckersBotTest

Данная программа используется для сравнения результативности и времени работы разных ботов. Она принимает в качестве аргументов количество игр и параметры двух ботов (формат можно узнать при вызове без аргументов). Далее программа запускает сервер, первого и второго клиента-бота, и перехватывает их вывод, считая количество побед, поражений, ничьих.

ЗАКЛЮЧЕНИЕ

В данной работе создано приложение для игры в шашки и компьютерные агенты (боты), использующие рассмотренные алгоритмы для игры с другими ботами или человеком. Сначала разработана простейшая версия с использованием алгоритма минимакса и несложной оценочной функции, которая затем была улучшена, после чего добавлено альфа-бета отсечение, и в итоге реализован алгоритм NegaScout с таблицей транспозиций. Также было проведено сравнение времени работы, максимальной глубины поиска при сохранении разумного времени выполнения, результативности игры против случайного агента и при игре между ботами с разной глубиной поиска. Выяснилось, что проведённые оптимизации алгоритма минимакса позволили добиться достаточно высокого уровня игры бота.

В ходе написания работы были решены следующие задачи:

- было создано клиент-серверное приложение для игры в шашки;
- реализованы различные алгоритмы поиска в игровом дереве позиций;
- найдены различные функции оценки позиций;
- проведено сравнение реализованных алгоритмов в плане времени работы и созданных оценочных функций по результативности.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Pijls W.*, *de Bruin A.* Game tree algorithms and solution trees // Theoretical Computer Science. — 2001. — Vol. 252, no. 1. — P. 197–215. — ISSN 0304-3975. — DOI: https://doi.org/10.1016/S0304-3975(00)00082-7. — CG'98.
2. *Allis L.* Searching for Solutions in Games and Artificial Intelligence. — Ponsen & Looijen, 1994. — ISBN 9789090074887.
3. Game Theory / M. Maschler [et al.]. — Cambridge University Press, 2013. — ISBN 9781107005488.
4. *Neumann J. v.* Zur theorie der gesellschaftsspiele // Mathematische annalen. — 1928. — Vol. 100, no. 1. — P. 295–320.
5. *Edwards D.*, *Hart T.* The Alpha-Beta Heuristic (AIM-030). Massachusetts Institute of Technology, 1963.
6. *Marsland T. A.* Computer chess methods // Encyclopedia of Artificial Intelligence. — 1987. — Vol. 1. — P. 159–171.
7. *Knuth D. E.*, *Moore R. W.* An analysis of alpha-beta pruning // Artificial Intelligence. — 1975. — Vol. 6, no. 4. — P. 293–326. — ISSN 0004-3702. — DOI: https://doi.org/10.1016/0004-3702(75)90019-3.
8. *Russell S.*, *Norvig P.*, *Canny J.* Artificial Intelligence: A Modern Approach. — Prentice Hall/Pearson Education, 2003. — (Prentice Hall series in artificial intelligence). — ISBN 9780137903955.
9. *Reinefeld A.* Spielbaum-Suchverfahren. — Springer, 1989. — (Informatik-Fachberichte). — ISBN 9783540507420.
10. *Marsland T. A.* The Anatomy of Chess Programs. // Deep Blue Versus Kasparov: The Significance for Artificial Intelligence. — 1997. — P. 24–26.
11. *Slate D. J.*, *Atkin L. R.* Chess 4.5—the Northwestern University chess program // Chess skill in Man and Machine. — Springer, 1983. — P. 82–118.
12. Русские шашки [Электронный ресурс] / Федерация шашек России. — URL: https://shashki.ru/variations/draughts64/ ; Загл. с экрана, яз. рус.

ПРИЛОЖЕНИЕ А

Программный код алгоритма минимакс

```
private int Minimax(PlayerColor currPlayer, int depth, ref Move maxMove)
{
    // Достигнута максимальная глубина поиска
    if (depth == 0)
        return EvalPieceCount(currPlayer);
    // Проверка, можно ли совершить взятие
    bool canCapture = board.CanPlayerCapture(currPlayer);
    // Все возможные ходы
    var allMoves = new List<Move>();
    for (int row = 0; row < Board.SIZE; row++)
        for (int col = 0; col < Board.SIZE; col++)
            allMoves.AddRange(board.GetMoves(currPlayer, (row, col),
            ↪ canCapture));
    // Нет ходов, проигрышная позиция
    if (allMoves.Count == 0)
        return -INF + 1;

    // Наименьший возможный результат
    int result = -INF;
    // Перебор возможных ходов
    foreach (var move in allMoves)
    {
        // Применение хода
        board.DoMove(move);
        // Рекурсивный поиск (для другого игрока)
        int res = -Minimax(1 - currPlayer, depth - 1, ref maxMove);
        if (res > result)
        {
            // Обновление результата максимумом
            result = res;
            // Обновление лучшего хода
            if (depth == maxDepth)
                maxMove = move;
        }
        // Отмена хода
        board.UndoMove(move);
    }
    return result;
}
```

ПРИЛОЖЕНИЕ Б

Программный код алгоритма альфа-бета отсечения

```
private int AlphaBetaPruning(PlayerColor currPlayer, int depth, int alpha, int beta, ref Move maxMove)
{
    // Достигнута максимальная глубина поиска
    if (depth == 0)
        return EvalPieceRow(currPlayer);

    // Проверка, можно ли совершить взятие
    bool canCapture = board.CanPlayerCapture(currPlayer);
    // Все возможные ходы
    var allMoves = new List<Move>();
    for (int row = 0; row < Board.SIZE; row++)
        for (int col = 0; col < Board.SIZE; col++)
            allMoves.AddRange(board.GetMoves(currPlayer, (row, col), canCapture));
    // Нет ходов, проигрышная позиция
    if (allMoves.Count == 0)
        return -INF + 1;

    // Наименьший возможный результат
    int result = -INF;
    // Перебор возможных ходов
    foreach (var move in allMoves)
    {
        // Применение хода
        board.DoMove(move);
        // Рекурсивный поиск (для другого игрока)
        int res = -AlphaBetaPruning(1 - currPlayer, depth - 1, -beta, -alpha, ref maxMove);
        if (res > result)
        {
            // Обновление результата максимумом
            result = res;
            // Обновление лучшего хода
            if (depth == maxDepth)
                maxMove = move;
        }
        // Отмена хода
        board.UndoMove(move);
```

```
37
38          alpha = Math.Max(alpha, res);
39          if (alpha >= beta)
40              break;
41      }
42      return result;
43  }
```

ПРИЛОЖЕНИЕ В

Программный код алгоритма NegaScout

```
private int NegaScout(PlayerColor currPlayer, int depth, int alpha, int beta,
    ref Move maxMove)
{
    // Достигнута максимальная глубина поиска
    if (depth == 0)
        return EvalPieceRow(currPlayer);

    // Проверка, можно ли совершить взятие
    bool canCapture = board.CanPlayerCapture(currPlayer);
    // Все возможные ходы
    var allMoves = new List<Move>();
    for (int row = 0; row < Board.SIZE; row++)
        for (int col = 0; col < Board.SIZE; col++)
            allMoves.AddRange(board.GetMoves(currPlayer, (row, col),
                canCapture));
    // Нет ходов, проигрышная позиция
    if (allMoves.Count == 0)
        return -INF + 1;

    // Наименьший возможный результат
    int result = -INF;
    // Перебор возможных ходов
    bool first = true;
    foreach (var move in allMoves)
    {
        // Применение хода
        board.DoMove(move);
        // Рекурсивный поиск (для другого игрока)
        int res;
        if (first)
        {
            res = -NegaScout(1 - currPlayer, depth - 1, -beta, -alpha, ref
                maxMove);
            first = false;
        }
        else
        {
            res = -NegaScout(1 - currPlayer, depth - 1, -alpha - 1, -alpha,
                ref maxMove);
```

```
36              if (alpha < res && res < beta)
37                  res = -NegaScout(1 - currPlayer, depth - 1, -beta, -res, ref
                    ↪  maxMove);
38          }
39          if (res > result)
40          {
41              // Обновление результата максимумом
42              result = res;
43              // Обновление лучшего хода
44              if (depth == maxDepth)
45                  maxMove = move;
46          }
47          // Отмена хода
48          board.UndoMove(move);
49
50          alpha = Math.Max(alpha, res);
51          if (alpha >= beta)
52              break;
53      }
54      return result;
55  }
```

ПРИЛОЖЕНИЕ Г

Программный код алгоритма NegaScout с таблицей транспозиций

```
private int NegaScoutTransposition(PlayerColor currPlayer, int depth, int alpha, int beta, ref Move maxMove)
{
    // Достигнута максимальная глубина поиска
    if (depth == 0)
        return EvalPieceRow(currPlayer);

    // Проверка, можно ли совершить взятие
    bool canCapture = board.CanPlayerCapture(currPlayer);
    // Все возможные ходы
    var allMoves = new List<Move>();
    for (int row = 0; row < Board.SIZE; row++)
        for (int col = 0; col < Board.SIZE; col++)
            allMoves.AddRange(board.GetMoves(currPlayer, (row, col), canCapture));
    // Нет ходов, проигрышная позиция
    if (allMoves.Count == 0)
        return -INF + 1;

    int alphaOrig = alpha;

    // Проверка наличия позиции в таблице транспозиций
    var boardHash = board.GetHash();
    if (transpositionTable.ContainsKey(boardHash))
    {
        var entry = transpositionTable[boardHash];
        // Запись в таблице для большей глубины и того же игрока
        if (entry.Depth <= depth && (depth - entry.Depth) % 2 == 0)
        {
            switch (entry.Type)
            {
                // Точное значение
                case TableEntryType.Exact:
                    return entry.Value;
                // Оценка снизу
                case TableEntryType.LowerBound:
                    alpha = Math.Max(alpha, entry.Value);
                    break;
                // Оценка сверху
```

```
38                 case TableEntryType.UpperBound:
39                     beta = Math.Min(beta, entry.Value);
40                     break;
41             }
42 
43             if (alpha >= beta)
44                 return entry.Value;
45         }
46     }
47 
48     // Наименьший возможный результат
49     int result = -INF;
50     // Перебор возможных ходов
51     bool first = true;
52     foreach (var move in allMoves)
53     {
54         // Применение хода
55         board.DoMove(move);
56         // Рекурсивный поиск (для другого игрока)
57         int res;
58         if (first)
59         {
60             res = -NegaScoutTransposition(1 - currPlayer, depth - 1, -beta,
               ↪  -alpha, ref maxMove);
61             first = false;
62         }
63         else
64         {
65             res = -NegaScoutTransposition(1 - currPlayer, depth - 1, -alpha -
               ↪  1, -alpha, ref maxMove);
66             if (alpha < res && res < beta)
67                 res = -NegaScoutTransposition(1 - currPlayer, depth - 1,
                   ↪  -beta, -res, ref maxMove);
68         }
69         if (res > result)
70         {
71             // Обновление результата максимумом
72             result = res;
73             // Обновление лучшего хода
74             if (depth == maxDepth)
75                 maxMove = move;
```

```
76          }
77          // Отмена хода
78          board.UndoMove(move);
79
80          alpha = Math.Max(alpha, res);
81          if (alpha >= beta)
82              break;
83      }
84
85      // Добавление позиции в таблицу транспозиций
86      if (transpositionTable.Count < MAX_TABLE_COUNT ||
        ↪ transpositionTable.ContainsKey(boardHash))
87      {
88          if (!transpositionTable.ContainsKey(boardHash))
89              transpositionTable[boardHash] = new TableEntry();
90
91          var entry = transpositionTable[boardHash];
92          // Оценка сверху
93          if (result <= alphaOrig)
94              entry.Type = TableEntryType.UpperBound;
95          // Оценка снизу
96          else if (result >= beta)
97              entry.Type = TableEntryType.LowerBound;
98          // Точное значение
99          else
100             entry.Type = TableEntryType.Exact;
101         entry.Depth = depth;
102         entry.Value = result;
103     }
104
105     return result;
106 }
```