Ultimate Tic-Tac-Toe

1. Постановка задачи:

Реализовать игру “альтернативные крестики-нолики”.

* 1. Правила данной игры:

Существует квадратное поле размером 3 х 3 (это поле будем называть *большим*, а его клетки - *большими* клетками). В каждой клетке большого поля находится ещё одно *маленькое* поле 3 х 3.

В игру играют 2 человека, один играет крестиками, а другой – ноликами. Каждый игрок ходит по очереди.

Суть игры: выиграть 3 больших клетки таким образом, чтобы они образовывали прямую (по вертикали, по горизонтали, или по диагонали). Как выиграть большую клетку? По такому же принципу – надо составить прямую из крестиков (или ноликов) в маленьких клетках.

Ограничения: тот, кто ходит первым, может походить куда-угодно. А все последующие ходы зависят от хода соперника, а именно: каждый последующий ход можно сделать только в *определенной* большой клетке, а она определяется *маленькой* клеткой, в которую походил соперник. Возможны варианты ничьи в маленьких полях, как быть? Можно считать, что клетка принадлежит обоим игрокам одновременно, или наоборот – никому. Также возможен вариант, когда маленькое поле, в которое игрок должен походить, не имеет свободных клеток. В таком случае игрок может походить куда угодно.

От выбранных правил может сильно отличаться стратегия игры, поэтому для дальнейших рассуждений необходимо определиться с конкретными правилами. В данном проекте будут использоваться такие правила:

* в “завершенном” (т.е. если кто-то выиграл) маленьком поле можно продолжать ходить. Это даст возможность “заставлять” соперника делать невыгодные ему ходы, что делает игру более интересной. Однако, из-за этого количество возможных ходов значительно увеличится, что может повлиять на качество разработки бота.
* если в маленьком поле ничья, то будет считаться, что клетка принадлежит обоим игрокам.
* если в маленьком поле нет свободных клеток, то игрок может походить в любое поле.

1. Программная реализация.

Требования:

- возможность нескольким игрокам играть одновременно;

- возможность создания комнат;

- красивый интерфейс;

- реализация регистрации, сохранения результатов, достижений и т.д.

Запланированный функционал:

- создание приложения на клиент-серверной архитектуре;

- реализация регистрации, сохранения результатов, достижений;

- возможность создания комнат;

- возможность ведения чата в комнате;

- создание рейтинга игроков;

- возможность играть как с человеком, так и с ботом;

1. Возможная реализация бота

Крестики-нолики являются игрой с полной информацией, поэтому итог игры предопределён в случае следования обоими игроками оптимальной стратегии, гарантирующей достижение выигрыша (или сохранение ничьей).

Игра с полной информацией — теоретико-игровой термин, обозначающий игру, где функция выплат одного или нескольких игроков не детерминирована стратегическим профилем.

Не вполне строго, но практически можно считать, что игра является игрой с полной информацией, если:

* игроки воздействуют на игровую ситуацию дискретными действиями — ходами, порядок ходов определён правилами и не зависит от таких параметров, как скорость реакции игроков (то есть очередной ход делает тот, кто должен его сделать по правилам, а не тот, кто первым догадался или успел его сделать);
* в любой момент игры все игроки имеют полную информацию о состоянии игры, то есть о позиции и всех возможных ходах любого из игроков.

Если, к тому же, ни в каких аспектах игры (правилах, возможности или очерёдности ходов, определении момента завершения игры или результата) не участвует элемент случайности, такая игра будет ещё и детерминированной.

Для любой детерминированной игры с полной информацией, теоретически, можно просчитать всё дерево возможных ходов игроков и определить последовательность ходов, которая гарантированно приведёт по крайней мере одного из них к выигрышу или ничьей, то есть всегда может быть построен алгоритм выигрыша или сведения игры вничью по крайней мере для одной из сторон.

В то время, как обычные крестики-нолики легко решаются с помощью поиска в глубину, альтернативные крестики-нолики решить не так-то просто.

Наиболее распространенная тактика искусственного интеллекта - минимаксная, она может использоваться для решения альтернативных крестиков-ноликов, но решение будет очень тяжело вычисляться и давать не всегда наилучший вариант. Это связано с тем, что, несмотря на относительно простые правила, альтернативные крестики-нолики не имеют никакой простой эвристической функции оценки. Эта функция необходима для минимаксного решения, поскольку она определяет, насколько хороша конкретная позиция.

Однако алгоритмы искусственного интеллекта, которые не нуждаются в оценочных функциях, таких как алгоритм поиска по дереву Монте-Карло, не имеют проблем при игре в эту игру. Поиск по дереву в Монте-Карло основывается на случайном моделировании игр, чтобы определить, насколько хороша позиция вместо позиционной оценки, и поэтому может точно оценить, насколько хороша текущая позиция. Поэтому компьютерные реализации, использующие эти алгоритмы, как правило, превосходят минимаксные решения.

Проанализировав игру, можно выделить 4 основных подхода:

- полный перебор

- альфа-бета отсечение

- выделение какой-либо победной стратегии

- поиск по дереву методом Монте-Карло

Рассмотрим каждый из них.

* 1. Полный перебор

В сравнение можно привести обычные крестики-нолики. Количество всех возможных ходов при всех возможных раскладах игры равно 255168. Количество клеток в обычных крестиках-ноликах равно 9. Значит, можно сделать 9 вариантов ходов на первом шаге, 8 вариантов ходов на втором шаге и т.д. Можно увидеть такую зависимость 9 \* 8 \* 7 \* … \* 1 = 9!. Но в этом соотношении не учитывается то, что игра может закончится раньше, чем закончатся свободные места на поле. Если отнять их количество, то мы как раз и получим число 255168. А теперь можно проанализировать поставленную задачу. В данном поле 81 клетка, значит существует 81! = 5 \* вариантов ходов. Но это ещё без отсечения “завершенных” партий. Вывести математическое соотношение, чтобы сообщить точный ответ, затруднительно. Но можно предположить, что это число все равно будет непозволительно большое. Поэтому данный метод, скорее всего, не имеет смысла.

* 1. Альфа-бета отсечение.

Можно рассматривать ходы как игровое дерево. Теоретически, они должны оценивать все позиции, которые возникнут после всех возможных ходов, затем все возможные ходы после этих ходов и т. д. Каждый ход одного игрока называется «узел». Перебор ходов продолжается, пока программа не достигает максимальной глубины поиска или не определяет, что достигнута конечная позиция. Уже на основании оценки позиции выбирается наилучшая стратегия.

Данный принцип использует алгоритм поиска лучшего хода при игре в шахматы. Это можно назвать модификацией предыдущего метода, потому что здесь, по сути, тоже идет перебор, но он идет с отбрасыванием “плохих” веток развития алгоритма.

* 1. Нахождение победной стратегии

Также допускается нахождение какой-либо оптимальной стратегии, придерживаясь к которой можно побеждать (или оставаться в ничьей) каждый раз. Данная стратегия существует. Зная её, можно побеждать каждый раз, если игрок ходит первым.

* 1. Поиск по дереву методом Монте-Карло

Метод Монте-Карло для поиска по дереву достаточно давно применяется в играх для искусственного интеллекта. Задача алгоритма — выбрать наиболее выигрышный вариант развития событий. Дерево представляет из себя структуру, в которой помимо хода и указателей есть количество сыгранных и количество выигранных партий. На основе этих двух параметров метод выбирает следующий шаг.

1. Реализация приложения

Для обеспечения многопользовательской игры, необходима реализация приложения с клиент-серверной архитектурой. Для данных целей можно использовать как web-приложение, так и desktop-ное. В данном проекте будет использоваться desktop-ное приложение.

Выбранный язык программирования приложения: C#. На нем будет написана как серверная, так и клиентская часть.

Для реализации визуальной части под C# существует множество сред разработки:

* Unity
* Windows Forms
* DirectX

Будет использоваться Windows Forms, поскольку игра будет не требовательной, и Windows Forms имеет хорошую интеграцию с C#.

Для обмена сообщениями между клиентом и сервером существует ряд текстовых форматов для обмена данными:

* XML
* JSON
* YAML
* CSV

Поскольку XML является универсальным форматом, для которого существует множество удобных библиотек для любых сред программирования, то в этом проекте будет использоваться он.

Для учета данных пользователей (аккаунты, матчи, достижения) необходимо использовать СУБД. Существует большое количество различных СУБД, но в данном проекте будет использован MySQL Workbench, поскольку он отлично подходит для небольших проектов.

* 1. Обзор алгоритмов и средств разработки

После анализа материалов были рассмотрены такие алгоритмы:

* полный перебор
* перебор с отсечением
* альфа-бета отсечение
* победная стратегия
* поиск методом Монте-Карло

Проанализировав их, можно сделать вывод, что наилучшими вариантами являются альфа-бета отсечение и метод Монте-Карло. В проекте будет реализован как минимум один из них. Возможно, даже оба, чтобы можно было провести некоторое сравнение алгоритмов.

Для создания приложения будут использованы такие технологии:

* клиент-серверная архитектура
* приложение, написанное на C#, с использованием Windows Forms
* СУБД MySQL Workbench
* XML (с использованием SAX-parser)

1. Анализ алгоритмов решения данной задачи
   1. Альфа-бета отсечение

Альфа-бета-отсечение — алгоритм поиска, стремящийся сократить количество узлов, оцениваемых в дереве поиска алгоритмом минимакса. Предназначен для антагонистических игр и используется для машинной игры. В основе алгоритма лежит идея, что оценивание ветви дерева поиска может быть досрочно прекращено (без вычисления всех значений оценивающей функции), если было найдено, что для этой ветви значение оценивающей функции в любом случае хуже, чем вычисленное для предыдущей ветви. Альфа-бета-отсечение является оптимизацией, так как результаты работы оптимизируемого алгоритма не изменяются.

Алгоритм альфа-бета-отсечения получил название по следующим двум пара­метрам, которые представляют пределы в зарезервированных значениях, присутст­вующих во всех узлах вдоль этого пути:

• α = значение наилучшего варианта (т.е. варианта с самым высоким значением), который был сих пор найден в любой точке выбора вдоль пути для иг­рока MAX;

• β = значение наилучшего варианта (т.е. варианта с самым низким значени­ем), который был до сих пор найден в любой точке выбора вдоль пути для иг­рока MIN.

Алгоритм альфа-бета-поиска в процессе своей работы обновляет значения α и β, а также отсекает оставшиеся ветви в узле (т.е. прекращает рекурсивные вызовы), как только становится известно, что значение текущего узла хуже по сравнению с теку­щим значением α или β для игрока МАХ или MIN соответственно.

Но главным недостатком данного алгоритма то, что в данной задаче сложно выбрать “хорошую” оценочную функцию. Также данный алгоритм является лишь оптимизацией полного перебора. Так что работать он будет не так-то и быстро, как хотелось бы.

Предложения построения оценочной функции:

Альфа-бета поиск будет продумывать 3-5 хода вперед (насколько позволяет вычислительная мощность), поэтому необходима такая функция, которая сможет оценивать состояние игры в определенный момент времени.

Можно использовать такие параметры:

* Количество выигранных / проигранных маленьких полей
* Возможность выигрыша / проигрыша
* Количество созданных “вилок”
* Количество бессмысленных ходов, которые игрок может заставить сделать
* Существования ситуаций, которые однозначно приводят к победе

Исходя из этих правил была получена такая оценочная функция:

|  |
| --- |
| int evaluation\_function(State& state, Player player)  {  const int WIN\_SCORE = 1000;  const int LOOSE\_SCORE = -1000;  const int DRAW\_SCORE = 0;  const int MOVE\_SCORE = 10;  Board& board = state.board;  int score;  if (board.value == player)  {  score = WIN\_SCORE - MOVE\_SCORE \* state.move\_num;  }  else if (board.value == nextPlayer(player))  {  score = LOOSE\_SCORE + MOVE\_SCORE \* state.move\_num;  }  else if (board.value == Draw)  {  score = DRAW\_SCORE;  }  else  {  if (BOARD\_SCORE\_ALLOWED)  {  int my\_score = get\_board\_score(board, player, WIN\_SCORE);  int enemy\_score = get\_board\_score(board, nextPlayer(player), WIN\_SCORE);  score = my\_score - enemy\_score;  }  else  {  score = 0;  }  }  return score;  } |

Где state – текущее состояние игры, player – игрок, который делает ход

* 1. Метод Монте-Карло

Метод Монте-Карло - общее название группы численных методов, основанных на получении большого числа реализаций случайного процесса, вероятностные характеристики которого совпадают с аналогичными величинами решаемой задачи.

В нашей задаче (поиск оптимального хода): для текущей позиции p выбираются все возможные позиции pi и с каждой из этих позиций разыгрывается большое количество случайных партий. Позиция, для которой соотношение побед и поражений будет наилучшим, выбирается для следующего хода.

Достоинства:

* Высокий уровень игры при большом количестве разыгрываемых случайных партий.
* Хорошо поддается распараллеливанию.
* Линейная сложность.

Недостатки:

* Хорошие ходы могут не попасть в случайную выборку.
* Для сильной игры требуется производительный компьютер.

Шаг 1: Выбор — Selection. На этом шаге алгоритм выбирает ход своего противника. Если такой ход существует — мы его выберем, если нет — добавим.

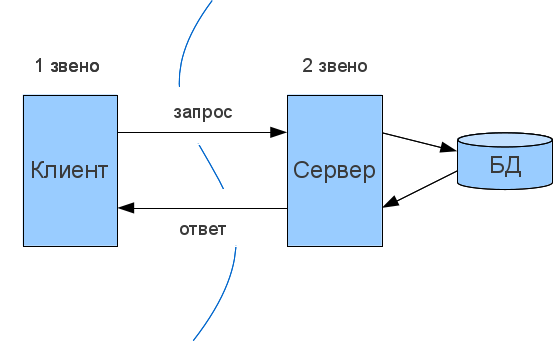
Шаг 2: Расширение — Expansion. К выбранному узлу с ходом противника мы добавим узел со своим ходом и с нулевыми результатами.

Шаг 3: Симуляция — Simulation. Отыграем партию от текущего состояния игрового поля до чей-либо победы. Отсюда мы возьмём только первый ход (т.е. свой ход) и результаты.

Шаг 4: Обратное распространение — Backpropagation. Результаты из симуляции мы будем распространять от текущего до корня. Ко всем родительским узлам мы добавим единицу в количество сыгранных партий, а если мы наткнёмся на узел победителя — то в такой узел мы добавим единицу в количество выигранных партий.

1. Описание предлагаемого решения

Необходимо реализовать многопоточное клиент-серверное приложение с поддержкой статистики и данных. Формальная схема данного приложения:



Клиент-серверная архитектура необходима для таких целей:

* Многопользовательская игра
* Хранение данных пользователя
* Хранение сыгранных партий
* Формирование статистики
* Обеспечение взаимодействия с пользователями

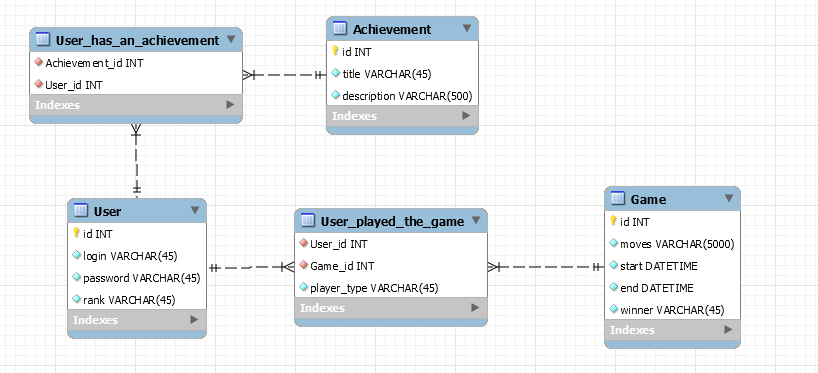
Функции, доступные клиенту:

* Зарегистрироваться
* Авторизоваться
* Получить информацию о своих достижениях
* Посмотреть статистику игр
* Создать комнату
* Подключиться к комнате
* Отправлять сообщения в комнате
* Играть с другими пользователями
* Играть с ботом

Задачи сервера:

* Поддерживать созданные комнаты
* Отвечать на запросы клиента
* Обеспечивать взаимодействие пользователей
* Вести статистику
* Поддерживать систему в целостном виде
* Обеспечивать работу с базой данных

Физическая модель данных (схема БД в 3НФ):



Клиент-серверное взаимодействие будет происходить по принципу Request – Response. Т.е. клиент делает какой-либо запрос к серверу, тот его обрабатывает и возвращает ответ. Во время игры клиент будет постоянно опрашивать сервер (с определенной частотой) “А не походил ли мой соперник”. Т.е. поскольку сервер сам по себе не может ничего отправлять, то клиент должен сам спрашивать его о получении новых данных.

Каждая сыгранная игра будет вносится в БД, что позволяет реализовать просмотр своих предыдущих игр. Значит, будет реализована некоторая история матчей.

Всю информацию, внесенную в БД можно использовать для формирования некоторой статистики, такой как: рейтинг игроков, достижения.

Пользователю будет доступна игра с ботом. Бот будет реализован на каком-либо из алгоритмов, перечисленных выше. Возможно, будет реализовано несколько вариантов ботов для проведения сравнительной характеристики.

1. Моделирование состояний игры

Для обработки информации о состоянии игры необходимо удобно представлять данные в памяти. Предлагаемая архитектура:

|  |  |
| --- | --- |
| class SmallBoard  private:  integer board[3][3];  public:  state checkGameState();  void setMove(x, y);  bool isFree(); | class BigBoard  private:  SmallBoard field[3][3];  public:  state checkGameState();  void setMove(x1, y1, x2, y2); |

По сути, класс BigBoard содержит в себе 9 экземпляров SmallBoard. Можно было бы сделать один класс, в котором было бы поле 9 х 9, но при таком устройстве неудобно было бы обрабатывать данные. Преимуществом предложенной архитектуры является то, что легко проверять, в каком состоянии находятся маленькие доски, также это сильно увеличивает читабельность и понятность кода.

Формат передачи ходов между сервером и клиентом:

P: x1, y1, x2, y2 - где, P – {X, O}, x1, x2, y1, y2 – {1, 2, 3}

Пример: X:1,3,2,1; это означает, что игрок “X” сделал ход в маленькой доске с координатами [1, 3] в клетку с координатами [2, 1].

История партий будет хранится в БД как набор последовательностей ходов. Пример X:1,3,2,1; O:2,3,1,1; X:3,2,1,1…

Основной функцией бота будет такова:

BigBoard getBestMove(BigBoard currentState);

Т.е. в функцию передает один единственный параметр – текущее состояние игры, а функция будет возвращать лучший ход. Состояние игры в момент времени задается объектом типа BigBoard.

1. Тестирование

Предусматривается несколько этапов тестирования:

* Первичная проверка адекватности бота
* Проверка игры с реальным человеком
* Поскольку система будет зависеть от некоторых параметров, то можно тестировать ботов, “заставляя“ играть их друг с другом, анализируя, какой бот играет лучше.

Также будет проводится тестирование эффективности: как будет зависеть качество бота от количества машинных операций?

Тестирование предусматривает проверку различных алгоритмов, их сравнение по качеству/времени работы, а также выявление наилучших параметров системы.

Будет использован портал hakerrank.com для тестирования полученного алгоритма с другими вариантами решения данной проблемы.

Для получения оценок по времени и памяти будет использоваться экспериментальный способ: регистрация оценок посредством запуска программы при различных данных. Как параметр различия данных можно использовать глубину обхода дерева решений.

Для проверки устойчивости алгоритма при внесении ошибок также будет использоваться экспериментальный метод на трех этапах игры: начале, средине и конце, что позволит понять, в какой момент игры сделанные ошибки будут являться более критичными.

Необходимо исследование того, как влияет то, кто будет первый ходить.

1. Визуализация

Поскольку будет предусмотрен режим просмотра сыгранных матчей, то можно сделать такую функцию: анализ своих матчей машиной. Т.е. во время просмотра своих игр можно будет смотреть на каждый свой ход, при этом машина будет также анализировать вашу игру, делая какие-либо оценки, при этом можно будет включить режим визуализации, что позволит лучше понять логику оценки данного хода. Визуализироваться будет алгоритм альфа-бета отсечения, поскольку это более наглядно, чем, скажем, метод Монте-Карло, потому что позволяет увидеть различные варианты игры (причем не всегда лучшие), что позволяет понять, чем тот или иной ход был лучше / хуже.

1. Описание процесса игры

В многопользовательском режиме игра будет происходить в реальном времени. Также игра будет идти на время. Т.е. игрок, у которого кончится время – проиграет.

Игры с ботом не будут считаться рейтинговыми. Также они будут идти не на время. Игру с ботом можно назвать режимом практики.

Нюансы игры: в многопользовательской игре МОЖНО закрывать приложение, а затем зайти снова и продолжить играть (если игра ещё не закончилась), поскольку состояние игры хранится на сервере. В то время, как игра с ботом идет на стороне клиента, поэтому при закрытии приложения игра не будет сохраняться.

Реализация рейтинговой системы:

Изначально каждый пользователь имеет начальный рейтинг в 1000 единиц. За каждую его сыгранную рейтинговую игру его показатель рейтинга изменяется. Формула реализована так, что слабый игрок, проиграв сильному, почти ничего не потеряет, а сильный – почти ничего не получит.

Зато если слабый игрок выиграет сильного, он получит много, а сильный – много потеряет.

Rank = Rank ± 25 ± delta, delta = |rank1 – rank2| / 25

Некоторые примеры:

Игрок\_1 (2200) – Игрок\_2 (1700): выиграл Игрок\_2: результат:

Рейтинг Игрока\_1 = 2200 - 25 – 500 / 25 = 2155.

Рейтинг Игрока\_2 = 1700+ 25 + 500 / 25 = 1745.

Т.е. при таком раскладе первый игрок потерял 53 очка, а второй – получил.

Рассмотрим обратный случай с теми же значениями:

Рейтинг Игрока\_1 = 2200 + 25 – 700 / 25 = 2205.

Рейтинг Игрока\_2 = 1700 - 25 + 700 / 25 = 1695.

Тут можно увидеть, что сильный игрок получил всего 5 очков, а слабый – потерял. Такая система разработана для того, чтобы сильные игроки не могли себе накручивать рейтинг на слабых, и наоборот – чтобы слабые игроки не теряли слишком много, проигрывая сильным.

Нюанс: если delta >= 25, тогда будет считаться, что рейтинг изменять не нужно. Потому что может получиться случай, что сильный игрок, сыграв с очень слабым, может потерять рейтинг, даже если он выиграет.

1. Анализ стратегий

Проанализировав игру можно выделить такие особенности:

* Тот, кто начинает первый имеет небольшое преимущество в виде бонуса: возможности закончить игру первым.
* Можно заметить, что ошибка на первом ходу игры не так критична по двум причинам: поскольку это начало игры, то существует ещё огромное множество развитий игры, что дает сложному боту шанс реабилитироваться; также можно заметить, что “плохой” ход не настолько уж и плохой. Ведь модуль разности оценочной функции между плохим и хорошим ходом равен 4, что по меркам алгоритма очень мало.
* Развив мысль предыдущего пункта, было выявлено, что каждая сделанная ошибка имеет более плохие последствия, если она сделана ближе к концу игры.
* Анализируя алгоритм, можно сделать вывод, что глубина обхода будет давать выигрыш в силе бота лишь до определенного момента.

Источники:

* [Англоязычная вики альтернативных крестиков-ноликов](https://en.wikipedia.org/wiki/Ultimate_tic-tac-toe)
* [Англоязычная вики крестиков-ноликов](https://en.wikipedia.org/wiki/Tic-tac-toe)
* [Русскоязычкая вики крестиков-ноликов](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B5%D1%81%D1%82%D0%B8%D0%BA%D0%B8-%D0%BD%D0%BE%D0%BB%D0%B8%D0%BA%D0%B8)
* [Пост с habr.com](https://habr.com/post/183764/)
* Richards D. J., Hart T. P. The Alpha-Beta Heuristic (AIM-030)
* [Англоязычная вики алгоритма минимакс](https://en.wikipedia.org/wiki/Minimax)
* [Описание алгоритма минимакс](https://tproger.ru/translations/tic-tac-toe-minimax/)
* [Русскоязычная вики метода Монте-Карло](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%9C%D0%BE%D0%BD%D1%82%D0%B5-%D0%9A%D0%B0%D1%80%D0%BB%D0%BE)
* [Англоязычная вики метода Монте-Карло](https://en.wikipedia.org/wiki/Monte_Carlo_method)
* Fishman, G. S. (1995). Monte Carlo: Concepts, Algorithms, and Applications.
* Оуэн Г. Теория игр.
* Коннолли Т., Бегг К. Базы данных. Проектирование, реализация и сопровождение. Теория и практика
* Валерий Коржов. Многоуровневые системы клиент-сервер
* Ник Рендольф, Дэвид Гарднер, Майкл Минутилло, Крис Андерсон. Visual Studio 2010 для профессионалов