ПОСТРОЕНИЕ ИИ ДЛЯ ИГРЫ В ЯПОНСКИЕ ШАХМАТЫ СЁГИ

Аннотация

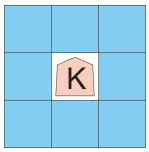
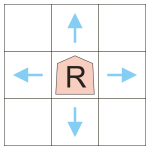
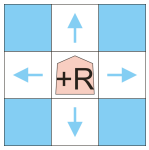
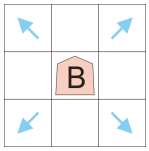
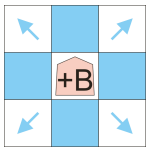
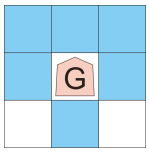
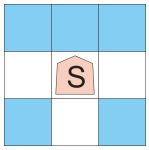
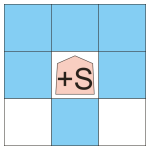
Построение программ, способных играть в интеллектуальные игры, — весьма интересная задача, имеющая большую практическую и научную ценность.
Первые попытки создать машины для игры в интеллектуальные игры (в первую очередь в шахматы) появились задолго до появления компьютеров. Около 1769 года появился известный шахматный аппарат «механический турок». Машина играла весьма неплохо, но весь её секрет заключался в спрятанном внутри сильном шахматисте.
В XX веке механические машины уступили свои позиции цифровым компьютерам. Первопроходцем в этой области (как и во многих других) можно назвать известного математика Алана Тьюринга. По современным меркам разработанный им алгоритм был весьма примитивным, а ввиду отсутствия доступа к реальным вычислительным машинам, выполнять алгоритм приходилось вручную.
Примерно в то же время Клод Шеннон обосновал наличие лучшего хода для любой позиции. Он даже предложил алгоритм нахождения такого хода для любой игры на конечной доске. К сожалению, несмотря на существование оптимального алгоритма, его практическая реализация невозможна ввиду ограниченности аппаратных и временных ресурсов.
Со времён Шеннона задача программирования интеллектуальных игр переместилась из области развлечений в область серьёзных исследований. За последние годы на основе исследований Шеннона были построены реализуемые на практике алгоритмы, с помощью которых компьютеры научились безошибочно играть в шашки и достигли выдающихся успехов в шахматах и го.
Японские шахматы сёги — одна из немногих игр, где человек пока сильнее компьютера. В первую очередь это обусловлено возможностью возвращения в игру взятых фигур, это правило резко увеличивает количество возможных ходов, а значит, затрудняет анализ игры. Если в шахматах из произвольной позиции можно сделать около 40 ходов, то в сёги число возможных ходов измеряется сотнями.
В данной исследовательской работе предпринята попытка создать программу, играющую в сёги на уровне сильного любителя. В процессе работы над проектом было реализовано множество известных алгоритмов (Alpha-Beta отсечение, выборочные продления, сортировка ходов), а также были разработаны специфические оптимизации для сёги. Особенностью разработанной программы является возможность работы в многопоточном режиме, что значительно повышает общее быстродействие.

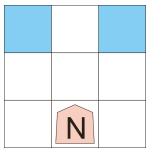
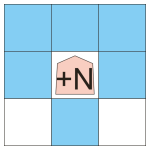
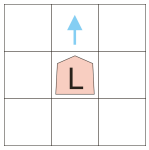
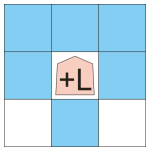
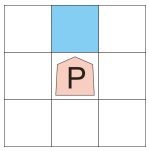
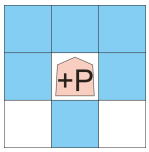
Правила игры

Прежде чем говорить о построении алгоритмов для игры в сёги, необходимо описать правила этой игры.

Фигуры

В сёги есть 8 различных фигур (если учитывать перевёрнутые фигуры, то 14). Правила ходов для различных фигур представлены в таблице:

Русское название	Японское название	Европейское обозначение	Ходы
Король	о:сё	K	
Ладья	хися	R	
Дракон	рю:о	+R	
Слон	какугё:	B	
Лошадь	рю:ма	+B	
Золото	кинсё:	G	
Серебро	гинсё:	S	
Перевёрнутое серебро	нари-гин	+S	

Русское название	Японское название	Европейское обозначение	Ходы
Конь	кэйма	N	
Перевёрнутый конь	нари-кэй	+N	
Стрела	кё:ся	L	
Перевёрнутая стрела	нари-кё:	+L	
Пешка	фухё:	P	
Перевёрнутая пешка	токин	+P	

Если проанализировать таблицу, можно сделать следующие наблюдения:

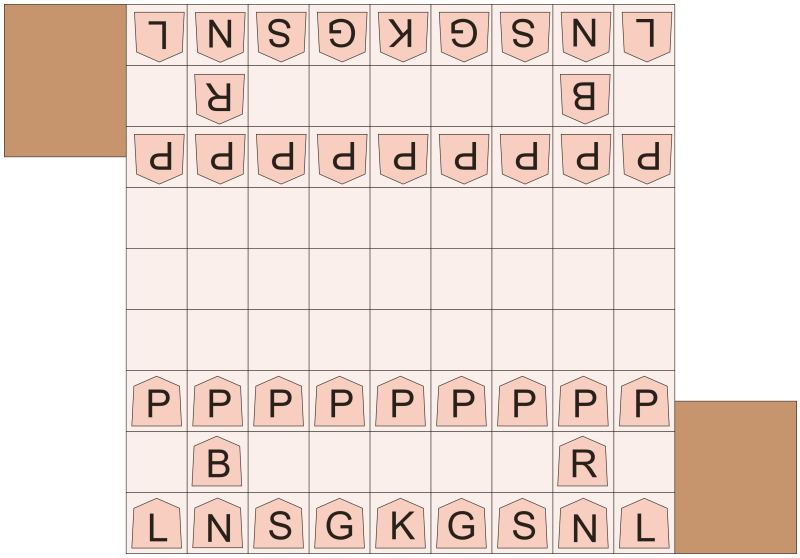
1. Дальнобойных фигур в сёги всего две: ладья и слон;
2. Возможности отступления многих фигур ограничены или совсем отсутствуют.

Эти наблюдения позволяют разбить все фигуры на четыре группы:

1. Король, обладающий абсолютной ценностью;
2. Старшие фигуры (ладья и слон), способные к дальнобойным атакам и быстрому отступлению;
3. Генералы (золотой и серебряный), чьи возможности наступления значительно превышают возможности отступления;
4. Младшие фигуры (конь, стрела, пешка), не способные отступать.

Начальная расстановка и очерёдность хода

В сёги играют два игрока, которых принято называть сэнтэ (ходящие первыми) и готэ (ходящие вторыми), на квадратной доске 9x9 клеток. Начальная расстановка фигур показана на рисунке:



Также у каждого игрока есть специальная подставка (комадай), куда помещаются взятые у противника фигуры. Также принято говорить, что взятые фигуры оказываются «в руке».

Перевороты

Три последние горизонтали (относительно каждого игрока) являются зоной переворота. Любая неперевёрнутая фигура (кроме золота и короля), начинающая или заканчивающая свой ход внутри этой зоны, может перевернуться и превратиться в другую фигуру. Правила превращения фиксированы и показаны в таблице:

Фигура	Превращённая фигура
P	+P
L	+L
N	+N
S	+S
B	+B
R	+R

Важно отметить, что решение о том переворачивать фигуру или нет, принимает игрок. Случаи, когда переворот необходим, описаны в разделе запрещённые ходы.

Сбросы

Правило сброса — это то, что делает игру сёги одной из самой сложных игр, придуманных человечеством. Суть правила заключается в том, что любую съеденную у противника фигуру можно выставить на любое свободное поле как свою.

В правиле сброса есть несколько ограничений, но они довольно просты:

1. Все фигуры сбрасываются не перевёрнутыми (даже если они сбрасываются в зону переворота);
2. Нельзя сбрасывать пешку на ту вертикаль, где уже есть пешка (токин не считается пешкой);
3. Нельзя сбрасывать пешку с матом.

Запрещённые ходы

Если противник сделал запрещённый ход, ему немедленно засчитывается поражение, поэтому крайне важно знать список запрещённых ходов:

1. Ходы, нарушающие правила (например, отсутпление золотом по диагонали);
2. Нарушение правил сброса (например, сброс второй пешки на вертикаль);
3. Ход, после которого фигура не сможет сделать ни одного хода. Это правило требует пояснения. Нельзя ходить пешкой на последнюю горизонталь без переворота или конём на последнюю или предпоследнюю горизонталь без переворота. Очевидно, что и сбрасывать фигуры таким образом нельзя.

Ситуация с оставлением короля под шахом не до конца ясна. Если игрок не заметил шах своему королю, то следующим ходом противник может (но не обязан) съесть короля. В этом случае игра считается законченной.

Завершение игры

Игра заканчивается, когда один из игроков поставил мат или сделал запрещённый ход. Но в сёги существуют дополнительные правила окончания игры:

1. Если позиция повторилась трижды в результате непрерывной серии шахов (т.н. «вечный шах»), то на четвёртый раз атакующий игрок должен сделать другой ход, иначе ему будет засчитано поражение.
2. Если же позиция повторилась 4 раза без объявления шаха, то игроки переигрывают партию без объявления результатов, но с переменой цвета на оставшееся время.

Вышеописанные правила делают ничейный результат крайне редким для сёги (не более 3% от всех сыгранных игр). Но тем не менее, ничья возможна в одном случае: если оба короля вошли в лагерь противника и укрепились там. Если оба игрока согласны с тем, что возникшая ситуация тупиковая, то происходит подсчёт очков. Все фигуры (в т.ч. и в руке) кроме короля, слона и ладьи оцениваются в 1 очко, слон и ладья — в 5 очков, король в подсчёте не участвует. В профессиональных партиях проигрывает игрок, у которого меньше 24 очков, если у обоих игроков не меньше 24 очков, то объявляется ничья. В любительских партиях побеждает тот, у кого не менее 27 очков, если у обоих игроков по 27 очков, то победа присуждает готэ.

Алгоритм нахождения лучшего хода

Алгоритм MiniMax

Впервые стратегия поиска лучшего хода для любой заданной позиции была описана ещё Клодом Шенноном. Если немного подумать, то можно заметить, что эта стратегия довольно проста и применима к любой игре.

Пусть у нас есть некоторая позиция, в которой нам необходимо сделать оптимальный ход. Вначале необходимо сгенерировать все разрешённые правилами ходы. Потом каждый из корректных ходов необходимо оценить. Ходы, ведущие к победе, будут оцениваться как +1, ведущие к поражению, — как -1, и ходы, ведущие к ничейным позициям, получат оценку 0.

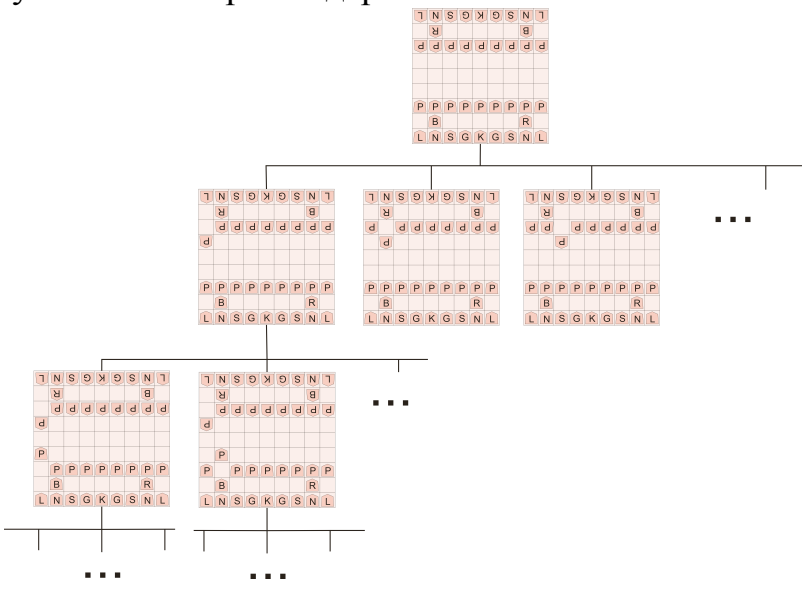
Для того, чтобы определить к какому результату приведёт нас рассматриваемый ход, мы должны предположить, что противник будет на каждый наш ход отвечать лучшим возможным способом. Т.е. фактически воспользуется рассматриваемым алгоритмом: сгенерирует все свои корректные ходы и среди них выберет лучший, причём для оценки того насколько хорош его ход, он предположит, что мы ответим лучшим возможным способом и т.д.

Получается, что в процедуре поиска лучшего хода с нашей стороны будет вызвана процедура поиска лучшего ответа со стороны противника, из которой будет вызвана процедура поиска лучшего нашего ответа на ответ противника, и т.д.[1] Функция поиска лучшего хода будет рекурсивно вызывать сама себя до тех пор пока не возникнет матовая или ничейная ситуация.

При этом оценка итоговой позиции не представляет сложности, а при оценке родительских узлов пользуются следующими правилами:

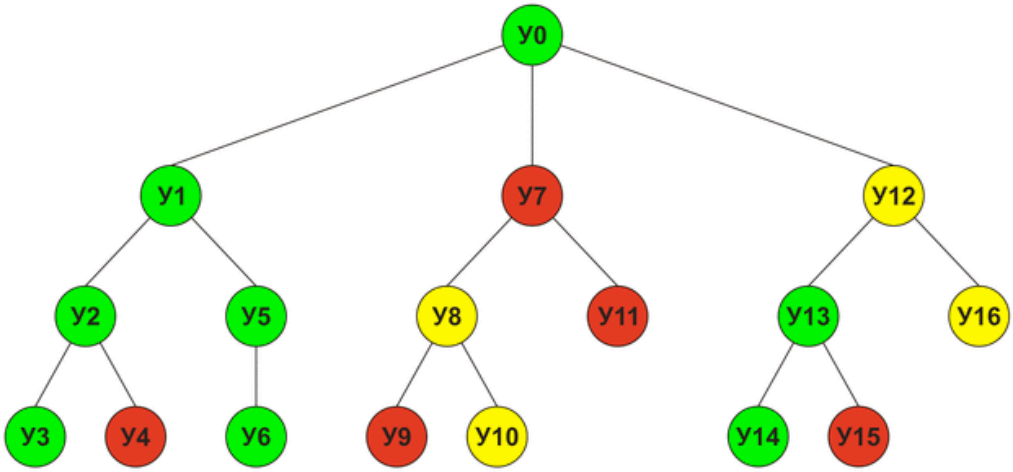
1. Оценка хода для первого игрока вычисляется как максимум оценок дочерних узлов;
2. Оценка хода для второго игрока вычисляется как минимум оценок дочерних узлов.

Такой алгоритм называют MiniMax. Если графически изобразить цепочку вызовов, то получится т.н. игровое дерево:



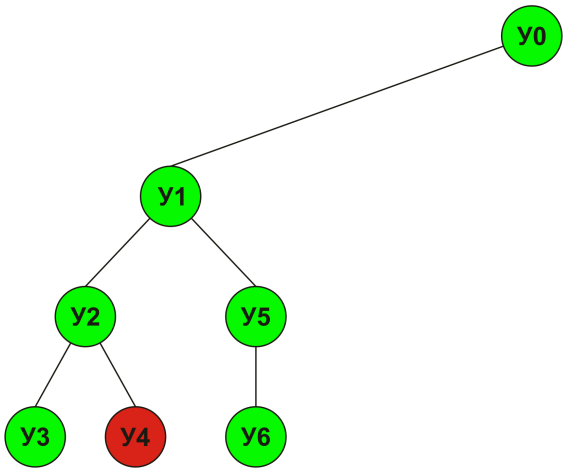
[1] Функции, вызывающие сами себя называются рекурсивными.

Каждый узел в этом дереве представляет собой один анализируемый ход. Оценка узла будет равна максимальной оценке его дочерних узлов. Ниже показан порядок в котором будут анализироваться узлы (зелёный — победа, красный — поражение, жёлтый — ничья):



Alpha-Beta отсечение

В рассмотренном выше примере для получения итоговой оценки узла У0, пришлось получить оценки для всех остальных узлов У1-У16, но на самом деле в этом не было необходимости: как только мы узнали, что узел У1 приводит к победе, необходимость в анализе узлов У7-У16 отпадает, т.к. любые оценки, полученные для тех узлов, не превысят оценку узла У1 (т.к. она максимальна), а это значит, что хода лучше чем У1 просто нет. Если учесть это, то анализируемое дерево будет выглядеть следующим образом:



Очевидно, что такой сокращённый анализ не ухудшает точность решения, но значительно сокращает время поиска лучшего хода. Именно идея «обрезки» неперспективных ветвей дерева лежит в основе алгоритма Alpha-Beta отсечений.

При использовании отсечений дополнительно вводятся две переменные: минимум для сэнтэ (A) и максимум для готэ (B). Сам алгоритм Alpha-Beta отсечений похож на алгоритм MiniMax, за исключением следующих пунктов:

1. Если оценка какого-либо хода выходит за диапазон [A,B], то анализ ветви можно досрочно прекратить, т.к. у игроков есть ходы, приводящие к лучшей позиции.
2. Если в какой-либо момент оценка первого игрока становится больше A, то значение A обновляется;
3. Аналогично поступают и с оценкой B для второго игрока.

Важным моментом при использовании этого алгоритма является порядок просмотра ходов. Если сначала рассматривать те ходы, которые сильнее всего сужают диапазон [A,B], то количество отсечённых ветвей будет достаточно большим. Возвращаясь к рассмотренному выше дереву, легко заметить, что порядок просмотра У7,У12,У1 даст гораздо меньший выигрыш в производительности. Поэтому перед использованием алгоритмов с отсечениями производят предварительную сортировку ходов по ожидаемой эффективности.

Конечно, заранее невозможно узнать, какой ход будет самым лучшим, но есть некоторые эвристические правила. Например, хорошими считаются ходы-взятия, шахи, ходы, улучшающие позицию, и т.д.

Функция оценки позиции

Все описанные выше алгоритмы производят просмотр игрового дерева на всю глубину, но реализовать такие алгоритмы на практике практически невозможно, поэтому глубину просмотра искусственно ограничивают: рекурсивные вызовы прекращают не только в случаях мата и ничьих, но и при достижении максимальной глубины анализа. Маты при просмотре дерева на ограниченную глубину встречаются сравнительно редко, а всем нематовым позициям приходится присваивать некоторую эвристическую оценку, для получения которой используют методы, описанные ниже.

Материальная оценка

В большинстве шахматоподобных игр весьма точной оценкой позиции является оценка материала. При этом каждой фигуре присваивается определённая стоимость, а оценкой позиции для игрока является разность между суммой его фигур и суммой фигур противника[2]. В сёги тоже используется оценка материала, таблица стоимости фигур приведена ниже:

Фигура	Стоимость
K	30000
R	100
+R	130
B	90
+B	120
G	60

[2] При это оценки сэнтэ и готэ будут различаться только знаком.

Фигура	Стоимость
S	50
+S	60
N	30
+N	60
L	20
+L	60
P	10
+P	70

Все фигуры в руке оцениваются на 30% дороже, т.к. их мобильность за счёт возможности сброса значительно возрастает.

Некоторых пояснений требует высокая стоимость короля и токина. Столь высокая стоимость короля объясняется тем, что король — фигура абсолютной важности, с его проигрышем партия заканчивается, поэтому король ценится дороже, чем все остальные фигуры вместе взятые. А стоимость токина так высока, потому что токин — лучшая фигура для обменов: при обмене токин-серебро один игрок получает в руку серебряного генерала, а второй игрок получается всего лишь пешку.

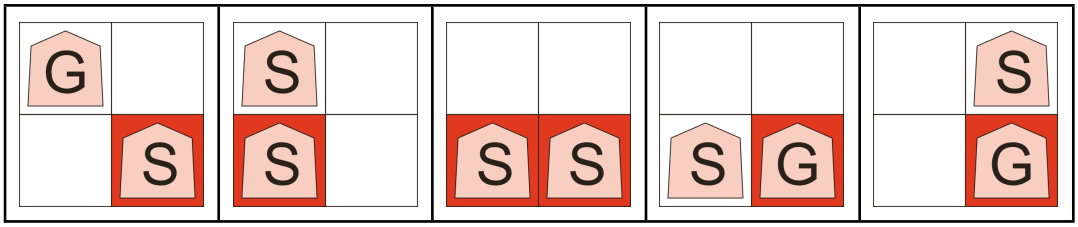
Стратегическая оценка

Прежде чем говорить о важности стратегической оценки, необходимо ещё раз отметить тот факт, что фигуры никогда не покидают игру и все взятые фигуры могут вернуться в любой момент. Этот факт кардинальным образом меняет подход к оценке позиции.

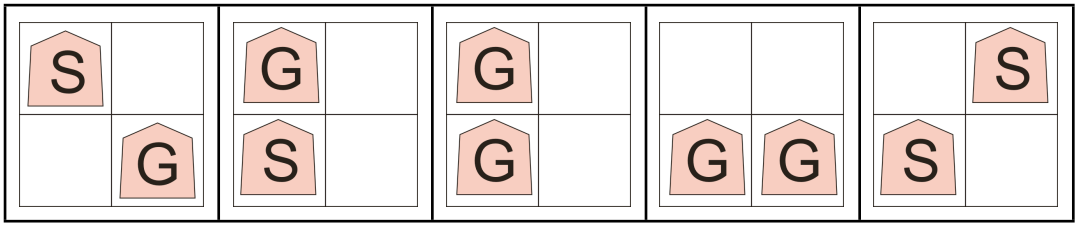
Во-первых, крайне эффективная в шахматоподобных играх эвристика обменов становится не только неэффективной, но и вредной. Если в шахматах выиграть одну пешку и постоянно идти на равноценные обмены, то в эндшпиле преимущество в одну пешку часто оказывается решающим. В сёги такие обмены могут привести к сбросам вражеских фигур в ваш лагерь.

Во-вторых, в сёги очень популярны жертвы. Например, в конце партии часто приходится отдавать старшую фигуру за золотого генерала или коня,чтобы получить в руку фигуру, которой можно поставить мат или сделать эффективную вилку.

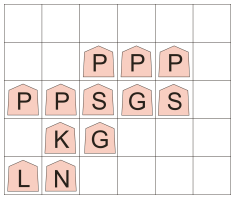
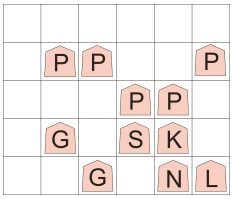
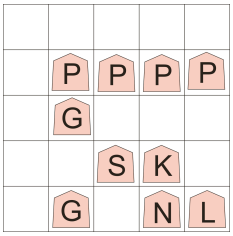
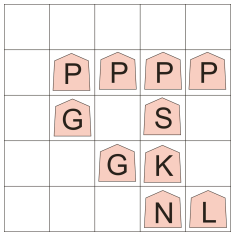
Если посмотреть, что общего у этих двух ситуаций, то можно заметить, что противник воспользовался неудачным расположением ваших фигур. В таких случаях принято говорить, что у фигур «плохая форма». В сёги форма зачастую оказывается важнее материального перевеса, поэтому характеристики формы обязательно должны оцениваться при оценке позиции. На рисунках ниже приведены примеры плохих форм (красным выделены ничем не защищённые генералы):



На следующей серии рисунков приведены примеры хороших форм:



Дальнейшим развитием идеи «хороших форм» являются крепости — укрепления для короля, обеспечивающие ему безопасность и защищающие от шахов. Примеры распространённых крепостей приведены в таблице:

Название	Ягура	Мино	Высокое мино	Серебряная корона
Крепость				

Понятно, что хорошие формы должны давать некоторый плюс в стратегическую оценку, а плохие формы, соответственно, ухудшать оценку, но помимо формы необходимо учитывать также контроль доски, близость генералов к своему или вражескому королю, наличие запертых фигур и т.д..

Подводя итог вышесказанному, надо ещё раз отметить, что в функции оценки позиции необходимо учитывать не только материальную составляющую, но и стратегическую.

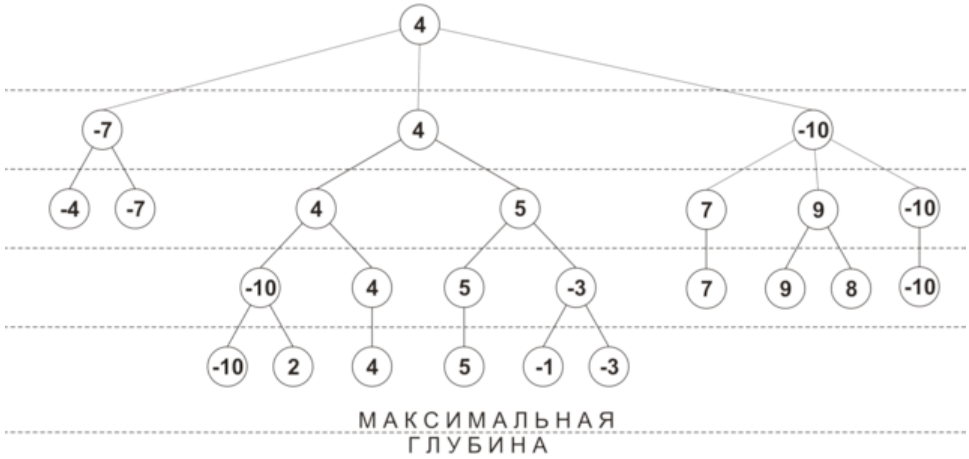
Алгоритм выборочных продлений

Уже рассмотренные алгоритмы MiniMax и Alpha-Beta отсечения всегда находят лучший ход на заданной глубине, но проблема в том, что глубина анализа (8-10 ходов) недостаточна для сильной игры. Профессионалы просчитывают позиции на 12-14 ходов, а некоторые позиции более чем на 20 ходов. Такая глубина для компьютеров пока недостижима[3].

Некоторым компромиссным вариантом является просмотр на значительную глубину только некоторых ветвей дерева, в то время как остальные ветви будут просматриваться на уменьшенную глубину. Такой подход называется методом выборочных продлений.

В шахматных программах обычно продлевают шахи, ходы пешки на последнюю горизонталь и сильные взятия. Но в сёги такой подход оказывается неэффективным. Было принято решение выполнять поиск без фиксированной глубины[4], увеличивая глубину шаг за шагом. При этом все возможные ходы теоретически могут быть рассмотрены на разную глубину, поэтому необходимо дополнительно запоминать глубину, на которую был рассмотрен каждый ход.

Изначально глубина просмотра каждого хода равна 0, а оценка хода равна -INF (т.е. проигрыш). На первой итерации выполняется анализ позиции на глубину 1 для всех ходов, при этом каждый ход получает свою оценку эффективности, а глубина его просмотра увеличивается на 1. Затем все ходы упорядочиваются по эффективности, и лучшие ходы анализируются на глубину 2. Если во время очередного продления выяснилось, что ход с максимальной оценкой уже проанализирован на максимальную глубину, то этот ход исключается из дальнейшего рассмотрения. Процесс углубления продолжается до тех пор, пока все ходы не будут просмотрены на минимальную глубину. На рисунке ниже показан примерный порядок построения дерева при использовании выборочных продлений (максимальная глубина — 4, минимальная глубина — 2, в узлах показана оценка позиция на текущей глубине просмотра):



[3] Если из произвольной позиции возможно в среднем 100 ходов, то увеличение глубины просмотра на единицу повысит время поиска приблизительно в 10 раз (при оптимальном порядке анализа ходов).

[4] На самом деле глубина ограничена двумя величинами: минимальной глубиной, на которую должны быть проанализированы все ходы, и максимальной глубиной, на которой продления прекращаются.

Дополнительным плюсом этого алгоритма является его эффективность в играх с ограничением времени: когда отведённое на ход время истекает, можно вернуть лучший из просчитанных результатов.

Кеширование

Известно, что кеширования является одним из самых эффективных способов повышения быстродействия программ. В контексте построения игровых алгоритмов кеширование используется для поиска уже проанализированных позиций.

Действительно, во время поиска лучшего хода некоторые позиции могут повторяться в пределах одной ветви или встречаться в разных ветвях дерева. Классический алгоритм рассматривал бы каждую такую позицию заново. Такие многократные вычисления значительно снижают общую производительность. Кеширование призвано избавить от необходимости выполнять повторные расчёты.

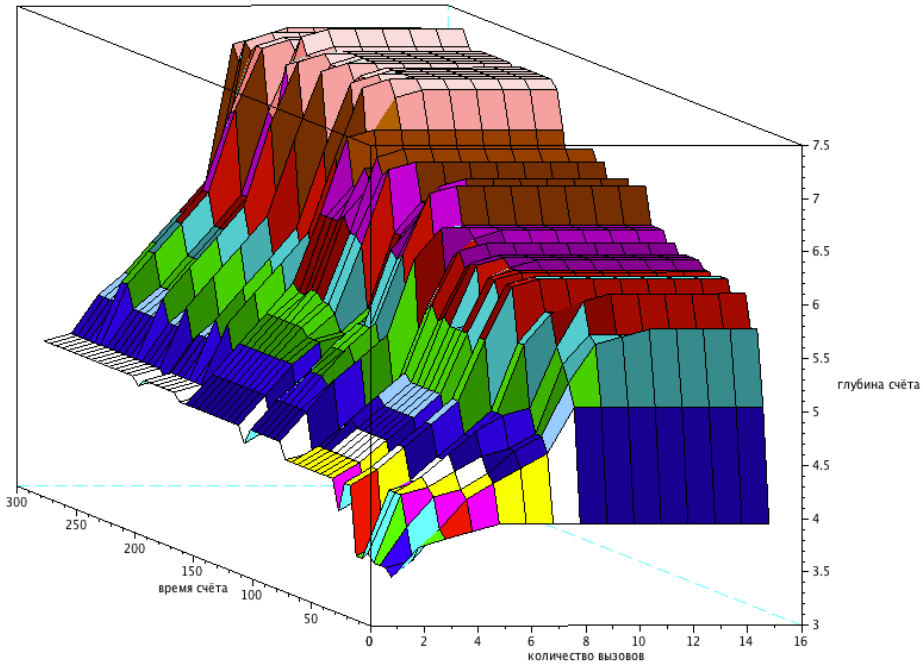
Поиск позиции в кеш основан на сравнении хеш-функции рассматриваемой позиции с хеш-функциями позиций, уже попавших в кеш. Особенностью большинства хеш-функций является наличие коллизий (ситуаций, когда разные входные данные дают одинаковое значение). Для игр такое поведение хеш-функций недопустимо, т.к. даже малейшие изменения в позиции очень сильно влияют на оптимальный ход. Учитывая вышесказанное, было принято решение использовать в качестве хеш-функции строку, однозначно описывающую позицию.

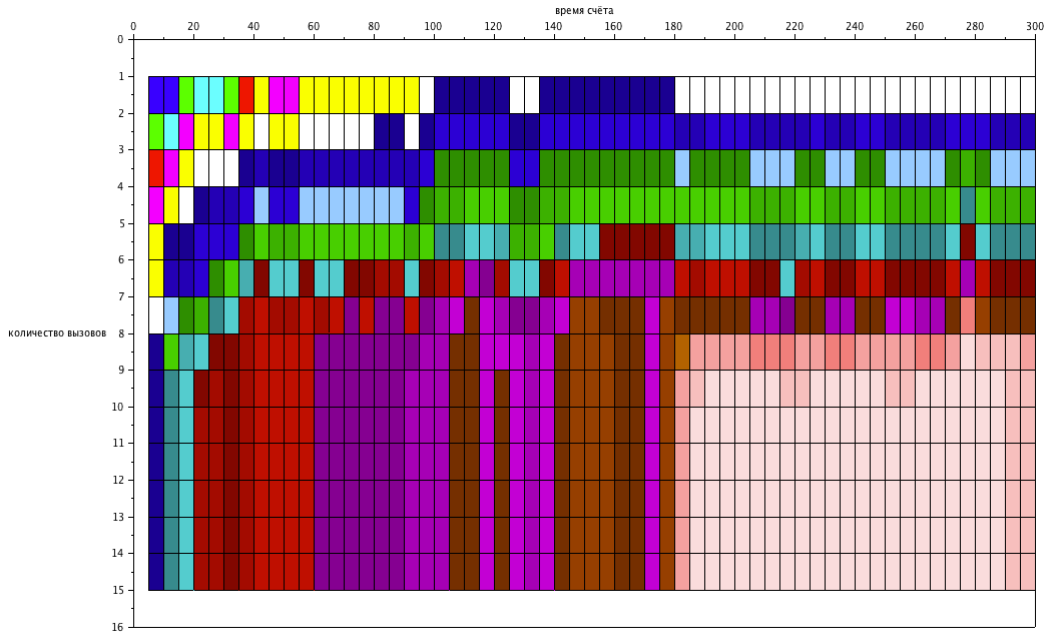
Для использования кеширования результатов в алгоритм поиска лучшего хода необходимо внести следующие изменения:

1. Перед началом просчёта проверять наличие позиции в кеш;
2. Если позиция содержится в кеш, то прекратить расчёт и вернуть полученное ранее значение;
3. При завершении просчёта записывать результат в кеш;

Есть ещё один довольно тонкий момент. Если в кеш уже содержится искомая позиция, просчитанная на малую глубину, этот результат тоже можно использовать. Вспомним, что алгоритм Alpha-Beta отсечений очень чувствителен к порядку ходов. Практика показывает, что оценка, полученная с малой глубиной просчёта, является довольно качественной эвристической оценкой качества хода. Т.е. значения, хранящиеся в кеш, могут быть использованы для сортировки ходов перед вызовом Alpha-Beta отсечений.

Эффективность использования кеш можно оценить с помощью следующего графика, на котором показана зависимость глубины просчёта позиции в зависимости от времени и количества повторных обращений.





По графику видно, что использование кеш значительно повышает глубину просчёта для повторяющихся позиций. Например, при ограничении времени в 5 секунд на ход использование кеш позволяет добиться такой же глубины анализа, как и 100-секундный анализ без использования кеш.

Т.о. можно сделать вывод, что использование кеширования значительно повышает скорость анализа, но требует значительных затрат памяти на хранение обработанных позиций.

Программная реализация

Для реализации описанных алгоритмов была выбрана среда разработки Delphi XE2. Эта среда отличается простотой освоения, высокой скоростью компиляции, эффективностью полученного кода и возможностью кросс-компиляции под различные платформы (Windows 32/64 bit, Mac OS).
Весь проект был разбит на функционально независимые модули:

Модуль	Функциональное назначение
UGlobal	Модуль, содержащий объявления всех базовых типов (фигуры, рука, доска и т.д.) и глобальных подпрограмм.
UFigMoves	Модуль, содержащий процедуры генерации простых перемещений фигур.
UMovesList	Модуль, содержащий процедуры и функции работы со списками ходов (сортировка, очистка, генерация).
UHash	Модуль, содержащий процедуры и функции вычисления хеша, записи и чтения кеш.
UGameTree	Модуль, содержащий реализации алгоритмов Alpha-Beta отсечений и алгоритма выборочных продлений.
USolveThread	Модуль обеспечивающий возможность многопоточной работы.
UFMain	Модуль, обеспечивающий графический интерфейс.

Рассмотрим каждый из этих модулей подробнее.

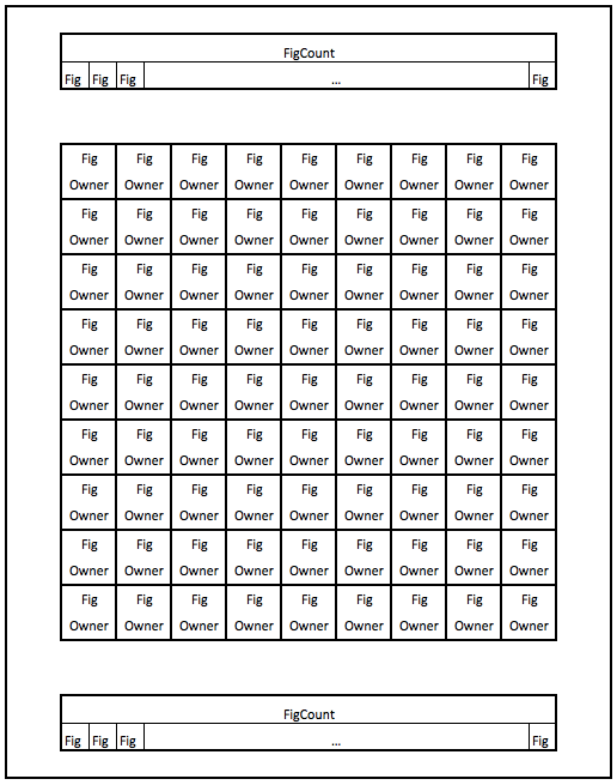
UGlobal

В этом модуле собраны объявления типов, константы и процедуры, необходимые всем другим модулям. Рассмотрим объявление самых важных типов.

```
type
  TColor = (sente,gote);
  TFigType = (_E,_P,_RP,_L,_RL,_N,_RN,_S,_RS,_G,_B,_RB,_R,_RR,_K);
  TFig = record
    Fig: TFigType;
    owner: TColor;
  end;
  TABoard = array [1..MaxN,1..MaxM] of TFig;
  TRHand = record
    FigCount: byte;
    Hand: array [1..MaxHandCount] of TFigType;
  end;
  TRPosition = record
    HandSente,HandGote: TRHand;
    Board: TABoard;
  end;
```

Играющие стороны (TColor), а также тип фигуры (TFigType) объявлены с помощью перечислимых типов. Фигуры на доске (TFig) представлены с

помощью записей из двух полей: тип фигуры и сторона-владелец. Сама доска (TABoard) представляет собой массив фигур. «Руки» игроков (TRHand) представлены с помощью записей из двух полей: количества фигур и, собственно, массива фигур. Наконец, позиция (TRPosition) представляется с помощью записи, объединяющей в себе все доску и руки противников. Т.о. позицию можно рассматривать как «матрёшку», состоящую из более простых типов:



Также в модуле UGlobal объявлены некоторые важные константы, например стоимости фигур, или размеры доски.

UFigMoves

Этот модуль экспортирует всего одну функцию:

```
function GetFigMove(const Board: TABoard; color: TColor; Fig: TFigType;
Figi,Figj: byte;  var Moves: TAMoves): byte;
var
r: byte;
begin
  r:=0;
  case Fig of
    _P: r:=PMoves(Figi,Figj,Board,color,Moves);
    _RP: r:=RPMoves(Figi,Figj,Board,color,Moves);
    _L: r:=LMoves(Figi,Figj,Board,color,Moves);
    _RL: r:=RLMoves(Figi,Figj,Board,color,Moves);
    _N: r:=NMoves(Figi,Figj,Board,color,Moves);
    _RN: r:=RNMoves(Figi,Figj,Board,color,Moves);
    _S: r:=SMoves(Figi,Figj,Board,color,Moves);
    _RS: r:=RSMoves(Figi,Figj,Board,color,Moves);
    _G: r:=GMoves(Figi,Figj,Board,color,Moves);
    _B: r:=BMoves(Figi,Figj,Board,color,Moves);
    _RB: r:=RBMoves(Figi,Figj,Board,color,Moves);
    _R: r:=RMoves(Figi,Figj,Board,color,Moves);
    _RR: r:=RRMoves(Figi,Figj,Board,color,Moves);
```

```
    _K: r:=KMoves(Figi,Figj,Board,color,Moves);
  end;
  GetFigMove:=r;
end;
```
На вход функции поступает текущее состояние доски, а также тип и положение фигуры, ходы которой необходимо сгенерировать. У этой функции есть несколько важных особенностей:

1. Функция генерирует только перемещения, т.е. сгенерировать с её помощью сбросы невозможно;
2. Функция возвращает т.н. относительные перемещения, т.е. для короля она вернёт следующий набор ходов: (-1,-1), (-1,0), (-1,+1),...,(+1,+1).
3. Функция возвращает набор относительных перемещений с точки зрения сэтэ (при этом для стрелы сторона владелец всё-таки учитывается).

UMovesList

UMovesList — один из самых больших и сложных модулей в программе. В нём содержится множество важных и довольно сложных процедур и функций. Все эти процедуры и функции можно условно разделить на три больших класса:

1. Процедуры сортировки ходов;
2. Процедуры генерации и проверки ходов;
3. Вспомогательные процедуры.

Если в сортировке ходов нет ничего сложного (используется обычный алгоритм сортировки выбором[5]), то процедуры генерации и проверки ходов необходимо рассмотреть дополнительно.

В первую очередь рассмотрим процедуру преобразования относительных координат, возвращаемых функцией GetFigMove, в абсолютные:

```
procedure RelativeToAbsolute(var N: word; var Moves: TAMoves; const Board:
TABoard; color: TColor);
var
i: word;
begin
  for i:=1 to N do
  begin
    if (color=gote) and (Board[Moves[i].fromi,Moves[i].fromj].Fig in
[_P,_RP,_RL,_N,_RN,_S,_RS,_G]) then
      Moves[i].toi:=-Moves[i].toi;
    Moves[i].toi:=Moves[i].fromi+Moves[i].toi;
    Moves[i].toj:=Moves[i].fromj+Moves[i].toj;
  end;
end;
```

Далее идут процедуры проверки на корректность хода, возможность сброса и возможность переворота (CheckMove, CanDrop и CanReverse). Особый интерес представляет функция MustReverse, проверяющая ход на обязательность переворота:

```
function MustReverse(Fig: TFigType; color: TColor; fromi,toi: byte): boolean;
var
fl: boolean;
begin
  fl:=CanReverse(Fig,color,fromi,toi);
```

[5] Использование более эффективных алгоритмов быстрой и пирамидальной сортировки оказалось неэффективным из-за сравнительно малого количества ходов

```
  if fl then
  begin
    fl:=Fig in [_P,_R,_B];
    if Fig in [_N,_L] then
    begin
      if color=sente then
        fl:=toi in [1..2]
      else
        fl:=toi in [MaxN-1..MaxN];
    end;
  end;
  MustReverse:=fl;
end;
```

Эта функция обязывает пешку, слона и ладью переворачиваться при первой же возможности. Это кажется логичным, т.к. переворот только усиливает эти фигуры, но в недавней партии на титул Осё Сато-Ватанабэ было показано, что ход слоном без переворота помогает избежать мата[6].

На основе описанных выше функций можно построить алгоритм генерирующий все корректные перемещения:

```
procedure GenerateBasicMoves(var N: word; var Moves: TAMoves; const Board:
TABoard; color: TColor);
var
i,j: word;
k,tmpN: word;
tmpMoves: TAMoves;
begin
    for i:=1 to MaxN do
    for j:=1 to MaxM do
      if (Board[i,j].Fig<>_E) and (Board[i,j].owner=color) then
      begin
        InitMovesList(tmpMoves); tmpN:=0;
        tmpN:=GetFigMove(Board,color,Board[i,j].Fig,i,j,tmpMoves);
        for k:=1 to tmpN do
        begin
          N:=N+1;
          Moves[N]:=tmpMoves[k];
        end;
      end;
  RelativeToAbsolute(N,Moves,Board,color);

  tmpN:=N;
  for i:=1 to tmpN do
    if
CanReverse(Board[Moves[i].fromi,Moves[i].fromj].Fig,color,Moves[i].fromi,Moves[i
].toi) then

AddMove(N,Moves,Moves[i].fromi,Moves[i].fromj,Moves[i].toi,Moves[i].toj,true);
  InitMovesList(tmpMoves); tmpN:=0;
  for i:=1 to N do
    if CheckMove(Moves[i],Board,color) then

AddMove(tmpN,tmpMoves,Moves[i].fromi,Moves[i].fromj,Moves[i].toi,Moves[i].toj,Mo
ves[i].reverse);
  InitMovesList(Moves); N:=0;
  for i:=1 to tmpN do
```

---

[6] Подробный анализ ситуации — http://www.youtube.com/watch?feature=player_detailpage&v=5EpyBxQAHyk#t=1004s

```
AddMove(N,Moves,tmpMoves[i].fromi,tmpMoves[i].fromj,tmpMoves[i].toi,tmpMoves[i].
toj,tmpMoves[i].reverse);
end;
```
И алгоритм, генерирующий корректные сбросы:
```
procedure GenerateDrops(var N: word; var Moves: TAMoves; const Board: TABoard;
color: TColor; Hand: TRHand);
var
i,j,k: word;
begin
  for k:=1 to Hand.FigCount do
    if (k>1) and (Hand.Hand[k]=Hand.Hand[k-1]) then
      Continue
    else
    begin
      for i:=1 to MaxN do
        for j:=1 to MaxM do
          if CanDrop(Hand.Hand[k],i,j,color,Board) then
            AddMove(N,Moves,0,k,i,j,false);
    end;
end;
```
Объединив эти две процедуры, можно получить все возможные корректные ходы из заданной позиции.
Также в модуле UMovesList объявлена процедура MakeMove, которая выполняет конкретный ход.

UHash

В модуле UHash собраны процедуры и функции, предназначенные для работы с кешем. Особенностью реализации кеша является то, что он хранится не в ОЗУ, а на жёстком диске. Это, конечно, сильно снижает его быстродействие, но особенность сёги в огромном количестве возможных ходов, поэтому хранение кеш в ОЗУ не представляется возможным.
Наиболее важной является функция получения хеша от позиции. Выше уже говорилось, что хеш должен максимально точно соответствовать позиции, поэтому в качестве хеш функции используется строковое представление позиции.
```
function GetHash(const POS: TRPosition; color: TColor): string;
var
r: string;
i,j: byte;
begin
  r:='';
  if color=sente then
    r:=r+'S'
  else
    r:=r+'G';
  r:=r+IntToStr(POS.HandSente.FigCount);
  for i:=1 to POS.HandSente.FigCount do
    r:=r+FigToChar(POS.HandSente.Hand[i]);
  r:=r+IntToStr(POS.HandGote.FigCount);
  for i:=1 to POS.HandGote.FigCount do
    r:=r+FigToChar(POS.HandGote.Hand[i]);
  for i:=1 to MaxN do
    for j:=1 to MaxM do
      if POS.Board[i,j].Fig=_E then
        r:=r+'e'
```

```
      else
      begin
        if POS.Board[i,j].owner=sente then
          r:=r+'s'
        else
          r:=r+'g';
        r:=r+FigToStr(POS.Board[i,j].Fig);
      end;
  GetHash:=r;
end;
```

Из модуля экспортируются следующие функции[7]:

```
procedure InitHash;
function HashExists(const POS: TRPosition; color: TColor; HL: byte): boolean;
function GetHashResult(const POS: TRPosition; color: TColor; HL: byte): integer;
procedure WriteHash(const POS: TRPosition; color: TColor; HL: byte; MC: word;
var Moves: TAMoves);
function GetMovesOrder(const POS: TRPosition; color: TColor; HL: byte; var
Moves: TAMoves): word;
```

С их помощью можно производить запись в хеш, чтение из него, а также производить переупорядочивание ходов.

UGameTree

Модуль UGameTree можно назвать «мозгом» программы. Именно в нём реализованы алгоритмы Alpha-Beta отсечений и выборочных продлений.

```
function AB(POS: TRPosition; color: TColor; Depth: byte; A,B: integer): integer;
var
Moves: TAMoves;
i,MovesCount: word;
OLDPos: TRPosition;
begin
  if TimeOut then
  begin
    AB:=-INF;
    Exit;
  end;
  if HashExists(POS,color,Depth) then
  begin
    AB:=GetHashResult(POS,color,Depth);
    Exit;
  end;
  MovesCount:=0;
  if (Depth=0) or Mate(POS.HandSente,POS.HandGote,color) then
  begin
    AB:=EvaluteMaterial(POS,color)+EvaluatePos(POS,color);
    exit;
  end;
  if color=sente then
    MovesCount:=GenerateMoves(POS.Board,color,POS.HandSente,Moves)
  else
    MovesCount:=GenerateMoves(POS.Board,color,POS.HandGote,Moves);
  for i:=1 to MovesCount do
  begin
    if A>=B then
      Break;
    OLDPOS:=POS;
```

[7] Во всех функциях параметр HL означает глубину просчёта позиции. Если одна позиция сначала просчитывалась на глубину 1, а потом на глубину 7, то оба эти результата будут хранится в кеш.

```
      MakeMove(POS,Color,Moves[i]);
      Moves[i].r:=-AB(POS,opcolor(color),Depth-1,-B,-A);
    POS:=OLDPos;
    if Moves[i].r>A then
      A:=Moves[i].r;
  end;
  if MovesCount>0 then
  begin
    WriteHash(POS,Color,Depth,MovesCount,Moves);
  end;
  AB:=A;
end;
```

Представленная реализация алгоритма Alpha-Beta отсечений поддерживает ограничение времени (выход по TimeOut) и кеширование результатов. С помощью данной функции можно получить оценку лучшего возможного хода из данной позиции.

Для поиска самого лучшего хода используется алгоритм выборочных продлений. Сначала генерируются все ходы:

```
if color=sente then
  MovesCount:=GenerateMoves(POS.Board,Color,POS.HandSente,Moves)
else
  MovesCount:=GenerateMoves(POS.Board,Color,POS.HandGote,Moves);
for i:=1 to MovesCount do
  Moves[i].heuristic:=0;
```

А затем в цикле производится оценка каждого хода с постепенным увеличением глубины анализа:

```
ST:=Now; fl:=true; MC:=MovesCount;
while fl do
begin
  D:=Moves[1].heuristic+1;
  SetHeuristic(Moves,MC,D);
  ParallelEvaluation(POS,color,D,MC,Moves);
  MC:=MovesCount;
  SortMovesByResDown(Moves,1,MC);
  FirstToLast(Moves,MC);
  MC:=SameRes(Moves,MC);
  SortMovesByHeuristicUP(Moves,1,MC);
  MC:=SameHeuristic(Moves,MC);
  fl:=false;
  for i:=1 to MovesCount do
    fl:=fl or (Moves[i].heuristic<MinDepth);
  fl:=fl and (not TimeOut);
end;
```

Некоторых пояснений требует процедура ParallelEvaluation. Ключевой её особенностью является работа в несколько потоков. При входе в функцию создаётся массив потоков, каждый из которых будет анализировать один из возможных ходов:

```
SetLength(ThrArr,MovesCount+1);
for i:=1 to MovesCount do
begin
  ThrArr[i]:=SolveThread.Create(true);
  ThrArr[i].ThrPOS:=POS;
  MakeMove(ThrArr[i].ThrPOS,Color,Moves[i]);
  ThrArr[i].ThrDepth:=Depth-1;
  ThrArr[i].ThrColor:=opcolor(color);
  ThrArr[i].ThrCompleted:=false;
```

```
  ThrArr[i].ThrInd:=i;
  ThrArr[i].Resume;
end;
```
Внутри каждого потока выполняется алгоритм Alpha-Beta отсечений. Когда будут завершены все потоки (критерием завершения является переменная ThrCompleted), каждый ход получит свою оценку.

USolveThread и UFMain

Эти модули являются в некотором роде вспомогательными. Модуль USolveThread содержит только объявление класса потока и вызов процедуры отсечений, а в модуле UFMain содержится реализация графического интерфейса, значительная часть кода модуля UFMain была сгенерировна средой Delphi XE2 автоматически.

Результаты работы

В результате проведённой работы были изучены алгоритмы, используемые в построении современных игровых программ, были предложены авторские оптимизации исследованных алгоритмов и была создана программа, играющая в сёги на уровне начинающего любителя.

В сравнении с распространёнными игровыми программами разработанная программа показала весьма неплохие результаты:

1. Shogi Lvl. 100 побеждает вплоть до 10 уровня (из 100);
2. KShogi55 (вариант 5 на 5) побеждает вплоть до 3 уровня (из 8).

Сравнительно низкие результаты объясняются тем, что программы Shogi Lvl. 100 и KShogi55 начиная с определённого уровня подключают к просчёту дебютные базы (специальные таблицы заранее просчитанные на глубину >20).

В произвольных позициях разработанная программа лишь незначительно уступает коммерческим аналогам.

В дальнейшем планируется внедрить несколько оптимизаций, которые теоретически могут значительно увеличить быстродействие и силу игры. Среди таких возможных оптимизаций можно отметить следующие:

1. Оптимизация структур данных;
2. Вычисление потенциальных ходов (кики) на этапе изменения позиции;
3. Изменения характера генерируемых ходов в зависимости от текущей глубины.

Приложение: Примеры партий

В данном разделе приведены два примера игр разработанной программы против программ Shogi Lvl. 100 и против программы KShogi55.

Shogi Lvl. 100	KShogi55