“О сколько ты открытий чудных
готовишь просвещенья дух!”
А.С.Пушкин

Шахматы

Машина играет в шахматы! Утверждение, казавшееся невероятным в середине 20го века, ни у кого не вызывает удивления сейчас. К хорошему быстро привыкают.
В этой книге, не претендуя на полноту и абсолютную точность, изложены основные принципы построения шахматной программы среднего уровня.

Корнилов Е.Н. 2012

Введение.

Взгляд на шахматную игру - Шахматы это высокоинтеллектуальная игра, развлечение, отдых для ума и способ организации мышления. Шахматы учат планировать будущее и соразмерять свои возможности. Капабланка учил шахматной игре начиная с эндшпиля. Малофигурные окончания помогают овладеть управлением фигурами и понять цели или эвристики для каждой фигуры (а также их ценность).

Шахматы отражают философию человека и его модель поведения. Вопрос как правильно играть видимо из категории вопросов "Как правильно жить"

У чемпиона мира Алехина всегда была ясная цель (цели) для каждого хода. Это помогало партии не распадаться на части, а сохранять внутреннюю напряженность.

Сейчас играют не так как в начале 20-го века. Главной становится фигурная активность в том смысле как кто ее понимает. Наверное - это правильно.

Компьютеры наступают. То, что раньше было прерогативой исключительно человека, теперь заменяется машинным интеллектом. На рисунке изображено 'чудо' 20-21 века - шахматный компьютер 'Каспаров-Chess'. Очень удобно. Можно выставлять различные уровни игры и не ломать глаза об монитор компьютера.

Действительно ли машига 'думает' или это удачная иммитация? В принципе - какая разница, если это выглядит как настоящее. Найти подходящего партнера для партии в шахматы бывает порой непросто или даже невозможно. С годами люди начинают очень обостренно воспринимать игру. А машина всегда под рукой. Если нет настоящего шахматного компьютера - можно и с ChessMaster сыграть.

'Houdini' - нынешний чемпион мира

Недавно смотрел фильм 'Револьвер'. Главный герой здорово наловчился играть в шахматы. Всех обыгрывал по своей 'формуле'. Сыграли с ним раз, другой. Говорят - "Как ты это делаешь? Мы с тобой больше не играем".
В этом, видимо, основная проблемы многих современных движков - они слишком сильно играют. Их могут использовать профессионалы для анализа позиций или устраивать 'гладиаторские' бои с другими программами, но для обычного среднего пользователя они не представляют большого интереса. Кому нравится когда его бьют, душат и пр. фигурально выражаясь. Вот и приходится для развлечения откапывать старые программы или опять тот же ChessMaster использовать. В этом плане очень хорощо выглядят Kasparov-Chess версия для настольного ПК. Я лично выиграл и еще разок с удовольствием сыграл.

MinMax

MinMax есть основной переборный алгоритм поиска для 2х игроков. Он очень прост:

1. Мы получаем все ходы из данной позиции
2. Делаем по очереди все перемещения.
3. Оцениваем все позиции, полученные из исходной.
4. Запоминаем лучшую оценку и лучший ход.
5. Возвращаем лучшую оценку.

Основной неясный момент с функцией оценки – как она может точно оценить все промежуточные позиции в игре. До некоторой глубины поиска в качестве функции оценки используется вариант функции MinMax для оппонента. Затем вызывается функция статической оценки, приближенно подсчитываюшая все позиционные и материальные факторы.

Итак, нам нужно:

1. Функция MinMax для белых
2. Функция MinMax для черных
3. Оценочная функция
4. Функции генерации перемещений и изменения позиции.
5.

Вот, в принципе и все. Осталось сказать, что оценка берется разностная:

все преимущества белых – все преимущества черных

Этот алгоритм исследует все перемещения и общее кол-во узлов для глубины D и кол-ва перемещений в позиции N будет:

N^D

Рост дерева поиска носит экспоненциальный характер. Прошу запомнить этот очевидный, но такой неприятный факт.

NegaMax

Следующий шаг в развитии алгоритмов поиска называется NegaMax. В сущности это MiniMax, только вместо 2х функций поиска (для 2х сторон) используется одна с инвертированной оценкой. Как мы помним, оценка в играх для 2х игроков инвертированная:

BLACK_SCORE = -WHITE_SCORE

Вот пример на некоторм несуществующем языке программирования, похожем на Pascal и Си:

```
/*
    Ищет лучший ход и оценку
    Исследует все узлы
 */
Function NegaMax(depth,side,xside:integer):integer;
Var
  Score, best:integer;
Begin
  If depth <= 0 then
    Return Evaluate(side);

  Generate_all_moves(side);

  Best = -INFINITY;
  For each move do
  Begin
    Make_Move(move);

    Score = -NegaMax(depth-1,xside,side);

    Un_Mack_Move(move);
    If  score > best then
      Best = score;
  End;
  Return best;
End;

Function Evaluate(side:integer):integer;
Begin
  If side = WHITE then
    Return WhiteScore – BlackScore
  Else
    Return BlackScore – WhiteScore;
End;
```

Используемые термины:

1. depth – оставшаяся глубина поиска
2. side,xside – играющая сторона и сторона оппонента
3. Evaluate – функция статической оценки позиции

4. Generate_all_moves – ищет все перемещения для данной стороны
5. Make_Move – делает изменения в позиции
6. Un_Make_Move – отменяет изменения в позиции
7. INFINITY – некоторая очень большая велечина, условно принимаемая за бесконечночть

Данный алгоритм как и MinMax исследует все перемещения в каждой позиции и является экспоненциальным:

Количесво_исследованных_позиций = N^D

N – среднее кол-во ходов в каждой позиции
D – глубина поиска

В качестве зарисовки рассмотрим пример алгоритма, исследующего не все перемещения. Он гарантированно рассматривает только один лучший ход в узле. Остальные ходы будут рассмотрены, только если прогнозируемая оценка лучше достигнутой.

```
/*выборочный поиск*/
Function Search_One_Move(depth,side,xside:integer):integer;
Var
 Best,j:integer;
Begin
  If depth <= 3 then
    Return NegaMax(depth,side,xside);

  Generate_all_moves(side);

  //оценим все перемещения
  For j = 0 to max_moves-1 do
  Begin
   Make_Move(moves[j]);
   Moves_score[j] = -NegaMax(max(3,depth/2),xside,side);
   Un_Make_Move(moves[j]);
  End;

  //отсортируем по убыванию оценки
  Sort_Moves();


  //поиск – пока приблизительная оценка
  //лучше достигнутой
  Best = -INFINITY;
  For j = 0 to max_moves-1 do
  Begin
   If moves_score[j] <= best then
     Break;

   Make_Move(moves[j]);
   Moves_score[j] = - Search_One_Move (depth-1,xside,side);
   Un_Make_Move(moves[j]);
   If moves_score[j] > best then
```

```
    Best = moves_score[j];
  End;

  Return best;
End;
```

Данный алгоритм считает приблизительно в 2 раза глубже, чем NegaMax. Точность поиска не гарантирована.

На какую глубину считать от корня дерева поиска, сразу сказать трудно. Можно увеличивать глубину, пока не исчерпан лимит времени.

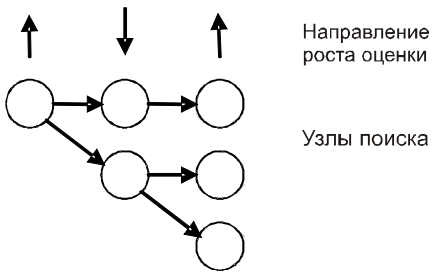
```
/* поиск от корня */
Function Root_Search(side,xside:integer):integer;
Begin
  Timer.reset();
  Depth = 3;

  Score = -INFINITY;
  While (depth < MAX_DEPTH) and
        Not timer.timeout() and
        (Score < WINE_SCORE) do
  Begin
    Score = Search_One_Move(depth,side,xside);
  End;

  Return best;
End;
```

AlphaBeta

Мы рассмотрели MinMax как полный перебор для определения лучшего хода в играх для 2х игроков. Он исследует все перемещения. Следующий шаг состоит в оптимизации этого алгоритма.
Казалось бы на первый взгляд, что нельзя досрочно прекратить поиск в узле без ухудшения качества поиска. Но это не так. В этом весь фокус-покус. Это похоже на функционирование клеток головного мозга (или во всяком случае, на то, как это понимают на данном этапе). Существует порог срабатывания. Входной сигнал в каждом узле накапливается, пока не превысит некоторого значения. Потом подается сигнал на выход.
Дерево поиска, формируемое рекурсивно, можно представить как своеобразную нейронную сеть. Лучшая оценка увеличивается, пока не достигнет некоторого предела. Какого? В этом весь вопрос.
Представьте себе некоторый узел поиска в дереве перебора. Оценка в нем только увеличивается. Достигнутый максимум передается далее. Оппонент занижает оценку. Если он ее уже занизил до нашего максимума, то дальше искать нет смысла. Все равно меньшая оценка не будет использована. Кому не понятно, я затрудняюсь объяснить. Это несложно, но объяснить затруднительно.




```
/*
    Ищет лучший ход и оценку
    Исследует при лучшем порядке
    перемещений только  SQRT( узлов)
  */
Function AlphaBeta(depth,alpha,beta,side,xside:integer):integer;
Var
  Score, best:integer;
Begin
  If depth <= 0 then
    Return Evaluate(side);
  Generate_all_moves(side);
  Sort_moves();

  Best = -INFINITY;
  For each move do
  Begin
    Make_Move(move);

    Score = -AlphaBeta(depth-1,-beta,-alpha,xside,side);
```

```
    Un_Mack_Move(move);
    If  score > best then
    begin
      Best = score;
      Alpha = max(alpha,best);
      If alpha >= beta then
       Break;
    End;
  End;
  Return best;
End;
```

Данная функция очень чувствительна к порядку перемещений. При лучшем порядке она может сосчитать в 2 раза глубже и исследует только SQRT(W) позиций, где W – число позиций при полном переборе.

Quies search

После того, как у AlphaBeta поиска исчерпана глубина перебора, у нас вызывалась функция статической оценки Evaluate. Она не учитывает размены фигур, что может приводить к серъезной погрешности. Традиционно принято в конце поиска считать только взятия при помощи специальной функции. Все ‘тихие’ перемещения не рассматриваются. Но может возникнуть ситуация, когда ‘тихий’ ход лучше взятия. Например, есть 2 взятия и оба ведут к потере материала. Для того, чтобы избежать серьезной погрешности в оценке, alpha завышается статической оценкой перед началом поиска. Этот трюк можно объяснить очень просто. Практически в любой позиции есть ход, не ухудшающий текущего положения и значит, оценка поиска не может быть хуже статической оценки позиции.

```
/*
    Ищет только взятия и превращения пешек
 */
Function Quies(alpha,beta,side,xside:integer):integer;
Var
  Score, best:integer;
Begin
  If depth <= 0 then
    Return Evaluate(side);

  Alpha := max(alpha, Evaluate(side));
  If Alpha >= Beta then
    Return beta;

  Generate_capture_moves(side);
  Sort_moves();

  Best = -INFINITY;
  For each move do
  Begin
    Make_Move(move);

    Score = - Quies (-beta,-alpha,xside,side);

    Un_Mack_Move(move);
    If  score > best then
    begin
      Best = score;
      Alpha = max(alpha,best);
      If alpha >= beta then
       Break;
    End;
  End;
  Return best;
End;
```

Обратите внимание, что параметра глубины поиска нет и все взятия рассматриваются до конца.

Взятия перед началом поиска должны быть отсортированы, например по принципу – (наиболее ценное взятие – наименее ценный нападающий). Взятие последней ходившей фигуры противника тоже имеет высокий приоритет.
Под шахом уходить в статический поиск нельзя, так как надо дать возможность королю уйти.

Хеширование

Как мы помним, от корня дерева поиска увеличение глубины перебора происходит постепенно с шагом (как правило) 1. Делается это по причине того, что неизвестно сразу, на какую глубину сможет сосчитать программа за отведенное время. Кроме того, поиск на глубину d-1 как правило занимает мало времени по сравнению с поиском на глубину d. Алгоритм AlphaBeta очень чуствителен к порядку ходов. Чтобы программа могла сосчитать на большую глубину необходимо сохранить проделанную работу.

Что есть хеширование? Хеш-таблица – это набор позиций, который для ускорения поиска разбит на несколько групп. Найти конкретную позицию становится намного легче, если мы знаем, где ее искать. Если у нас 1000000 позиций и 1000 групп, то для поиска нам необходимо проверить 1000000/1000 = 1000 позиций. Номер конкретной группы определяется хеш-ключом. Хеш ключ есть отображение большого объекта в число. Естественно, могут возникать коллизии, когда 2 различных ключа указывают на одну позицию. Это неизбежно, но хорошая хеш-функция генерирует ключи с минимумом коллизий. Традиционно для генерации ключей используются простые числа. Например, чтобы получить ключ строки, можно сделать что-то вроде:

```
Var key: LongWord;

  Key = 0;
  For J = low(str) to high(str) do
   Key = (key shl 1)  hor  ( 31 * str[j] );
```

Для шахматных позиций традиционно используется метод Z-Obrist ключей. Он подразумевает, что для каждого поля, цвета и фигуры существует случайное число. Ключ позиции представляет собой сумму этих ключей.

```
Var  rnd_key:array[white..black, pawn..king, 0..63] of  int64;

For c = white to black do
 For p = pawn to king do
 For sq = 0 to 63 do
  Rnd_tbl[c,p,sq] = random64();


Procedure  insert_piece(var pos:position;  c,sq,p:integer);
Begin
  Pos.board[sq] := p;
  Pos.color[sq]  := c;
  Pos.hash_key = pos.hash_key hor rnd_tbl[c,p,sq];
End;

Procedure  remove_piece(var pos:position;  c,sq,p:integer);
Begin
  Pos.board[sq] = empty_square;
  Pos.color[sq] = no_color;
  Pos.hash_key  = pos.hash_key hor rnd_tbl[c,p,sq];
End;
```

Простейшая хеш-таблица в шахматах может хранить только лучшие ходы.

```
Var hash_tbl:array[0..SIZE-1] of record
                                  Side,depth:integer;
                                  Move:Move_Type;
                                  Hash_key:int64;
                                 End;

Procedure Insert_Position(_side:integer; _key:int64;
                          _move:Move_Type;
                          _depth:integer);
begin
  with hash_tbl[ key mod SIZE ] do
  begin
    if _depth >= depth then
    begin
      depth = _depth;
      hash_key = _key;
      side = _side;
      move = _move;
    end;
  end;
end;
```

Как видите, все несложно. Как только найден ход, улучшающий alpha, его можно вставить в хеш. Перед началом функции поиска нужно посмотреть, нет ли лучшего хода в хеше. Ходы главных изменений (оценка больше alpha но меньше beta можно хранить в хеше подольше, так как это самые ценные узлы и не должны перезатираться). Если вышла коллизия и хеш-ключ неправильно определил позицию, то нет ничего страшного, только лучший ход будет неверен.
Наша упрощенная хеш-таблица не имеет групп, а только входы. Она будет эффективно, если хеш-ключ дает хорошее распределение.

Если мы хотим использовать оценки из хеша, то нужно учитывать, что это является поистине неисчерпаемым источником ошибок в шахматных программах. Даже если нет коллизий ключей и мы храним компактное битовое представление позиции для сравнения, нам нужно учитывать:

8. Позиции равны только тогда когда имеют одинаковое фигурное расположение и одинаковые перемещения. То есть нам придется хранить информацию о доступности рокировок и возможности взятия пешкой на проходе.
9. Нужно делать корректировку матовых оценок в зависимости от глубины.
10. Перед использованием результата из хеша должна вызываться функция, определяющая повторы в игре.

Хеширование позволяет значительно ускорить поиск (особенно в эндшпиле). Ведь мы имеем фактически не дерево игры а граф. В одну позицию можно прийти различными путями.

BitBoard

В настоящее время модной стала техника битового представления позиции. Она позволяет представить позицию более компактно и эффективно выполнять некоторые операции. Основой BitBoard техники служит 64 битовое число, где 1 бит представляет 1 поле доски. Так как и бита могут быть только 2 значения, то и обозначать он может лишь (занято, свободно).

```
/* Тип BitBoard  - 64 битовое число без знака */
Type BitBoard = LongDword; // unsigned long long, unsigned int64
```

Основные битовые операции:

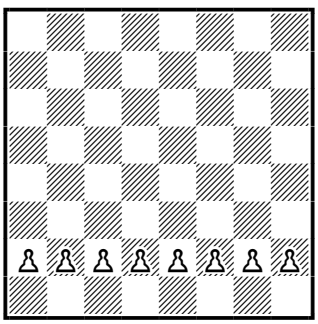
```
// установить бит с номером sq
W = w or ((BitBoard)1 shl sq);

// сбросить бит
W = w and not ((BitBoard)1 shl sq);

// сбросить младший бит
W = w and (w-1);

//получить номер первого бита
Asm
   MOV EAX, DWORD PTR w
   OR  EAX,EAX
  JZ @M1                  // младшее слово (первые 32 бита числа)
      BSF EAX,EAX          //  в EAX номер первого ненулевого бита
      Return EAX
@M1:
   MOV EAX, DWORD PTR (w+4)    // старшее слово числа
   BSF EAX,EAX
   ADD EAX,32
   Return EAX
End;
```

Белые пешки в битовой нотации будут выглядеть примерно так:



```
00000000
00000000
00000000
00000000
00000000
00000000
11111111
00000000
```

После хода e2-e4:

```
00000000
```

00000000
00000000
00000000
00001000
00000000
11110111
00000000

Маска всех белых фигур:

00000000
00000000
00000000
00000000
00000000
00000000
11111111
11111111

Пример кода, получающего номера полей с белыми фигурами:

```
W = white_pieces;
While  w <> 0 do
Begin
  Sq = get_first(w);
  W = w and (w-1);
End;
```

Сингулярная эвристика продлений.

Слово ***сингулярный*** значит ***странный*** или ***единственный***. Данная эвристика предусматривает углубленный просмотр некоторых вариантов с единственным ответом. Реализаций ее может быть множество.
В шахматах бывают позиции, когда несмотря на выигрыш в материале, игрок не может выправить положение некоторое время и вынужден защищаться. Это, как правило, бывает при глубокой угрозе королю.
Как реализовать подобную, чисто шахматную модель поведения средствами AlphaBeta поиска? Практика показывает, что речь идет как правило, об ожидании второй отсечки за бета. Если оценка поиска достигла beta предела и есть только один ход, дающий этот результат, то выполняется поиск на глубину +1 для уточнения результата.
Мы должны понимать, что значат alpha и beta пределы вообще. Представим себе некоторый вариант выборочного поиска, который использует специальную функцию для получения предполагаемой оценки.

```
Function main_search(...):integer;
...
For each move do
Begin
  Score = -selective_search(...);
  If score > alpha then
    Score = -main_search(...);
End;
```

Это стандартный выборочный поиск. Все узлы, предварительная оценка которых больше alpha являются кандидатами для более подробного рассмотрения.
Теперь давайте себе представим, что мы получили оценку меньше или равную alpha и предположительно, существует только один ход противника, дающий ему столь высокий результат. Велика вероятность того, что этот единственный ход ошибочен и если рассмотреть ситуацию глубже, то оппонент теряет а не приобретает. Нам потребуется специальная функция, выполняющая поиск всех вариантов кроме одного и функция выборочного поиска, возвращающая кроме оценки и лучший ход. Реализация может выглядеть примерно так.

```
./* ищет выборочно + единственный ответ */
Function main_search(depth,alpha,beta:integer; first_move:Tmove ...):integer;

If depth <= 0 then return Quies(alpha,beta);
Generate_And_Sort_Moves();


For each move do
Begin

  Best_move = 0;

  If Check() or Pawn_Move_Rank7() then  //расширения поиска
    Ext = 1
```

```
  Else
    Ext = 0;

  Score = -selective_search(depth-1+ext, &best_move, -beta,-alpha, ...);  //
выборочный поиск, возвращает результат и найденный ход

  If depth > 4 then
   If Ext == 0 then
   If score <= alpha then //поиск без одного хода на меньшую глубину
    Score = -singular_search((depth-1+ext)/2, best_move, -(alpha+1),
-alpha, ...);

  If (score > alpha) or ext then
    Score = -main_search(d-1, -beta, -alpha, best_move, ...);

  If score > alpha then
  Begin
    Alpha = score;
    If alpha >= beta then break;
  End;
End;
   Return alpha;
End;
```

Если мы вместо выражения ((depth-1+ext)/2) используем (depth-1+ext), то (возможно) программа будет играть очень осторожно и везде видеть возможную угрозу.
Впрочем, приведенная схема только демонстрирует основную идею и многое зависит от тонкой настройки программы.
Может, вам покажется интересной идея отслеживать только материальную грань легальных ходов. Тогда при поиске второго легального хода нужно использовать окно alpha+PAWN_VALUE/2.
Количество возможных ошибок при реализации сингулярной эвристики может уступать только реализации хеш-таблиц. Например :

11. Использует ли функция singular_search недействительное перемещение? Если да, то она явно не ищет второй ход.
12. Что делать если предварительный поиск вернул оценку <= alpha, а лучшего хода нет? Это может быть по причине использования результата из хеша, повтора позиций, отсечки недействительного перемещения и мало ли чего еще ☹

Выборочный поиск.

Мы рассмотрим только один вариант выборочного поиска. Их существует множество. ***Недействительное перемещение***, ***статическое отсечение*** и пр., тоже относятся к выборочному поиску.

Выборочный поиск по определению просматривает не все варианты и поэтому порядок ходов становится принципиальным. Просмотр каждого узла лучше начинать с осмысленного перемещения, а лучше и двух. Дело в том, что осмысленный ход установит более реалистичный alpha предел и поиск пойдет по другому.

Основная общая схема выборочного поиска может быть представлена как следующая:

```
Function search(...):integer;
 ...
 For each move do
 Begin
  Score = -selective_search(...);
  If score > alpha then
    Score = -search(...);
 End;
End;
```

В зависимости от того, высока или низка alpha, многие перемещения будут рассмотрены или отвергнуты.
Как же устроена функция selective_search? В простейшем случае не нужно отдельной функции и это есть вызов search() с глубиной d-1.

```
Const Min_Depth = 2

Function search(depth,alpha,beta:integer):integer;
 ...
 For each move do
 Begin
  If depth <= Min_Depth then
    Score = alpha+1
  else
    Score = -search(depth-2,-beta,-alpha);

  If score > alpha then
    Score = -search(depth-1,-beta,-alpha);
 End;
End;
```

Общий принцип нашего выборочного поиска состоит в том, что нужно последовательно увеличивая глубину, достигнуть beta предела. Мы это делали только с глубиной d-1, но можно расширить этот принцип.

```
Const Min_Depth = 2
```

```
Var best_move:array[0..MAX_DEPTH] of Move_Type;

./* Ищет не все  */
Function search(depth,ply,alpha,beta:integer):integer;
 Var score, D:integer;

 If check() then depth = depth+1;
 If depth <= 0 then return Quies(alpha,beta);

 Generate_And_Sort_Moves();
 Pick_Best_Move( best_Move[ply] );

 For each move do
 Begin
  If depth <= Min_Depth then
    Score = alpha+1
  Else begin
    Score = alpha+1;
    D = max(min_depth, depth/2+1);
    While (D <  depth-1) and (score > alpha) do
    begin
       Score = -search(D,ply+1,-beta,-alpha);
       D = D+1;
    End;
  End;

  If score > alpha then
    Score = -search(depth-1,ply+1,-beta,-alpha);

  If time_up then break;
  If score > alpha then
  Begin
     Alpha = score;
     Best_move[ply] = move;
     If alpha >= beta then break;
  End;

 End;
 Return alpha;
End;
```

Обратите внимание, что поиск каждого узла начинается с лучшего хода, полученного при d-1, кроме сложных позиций, где alpha увеличить не удалось и программа пытается это сделать с глубиной +1.

Недействительное перемещение.

Эвристика недействительного перемещения в свое время произвела настоящий фурор в шахматном программировании. Это связано, в первую очередь, с программой Fritz, которая очень точно анализировала сложные тактические позиции на сравнительно маломощных машинах.
Что же из себя представляет эта эвристика? Она использует тот же принцип статического отсечения. Считается, что в любой нормальной, где нет опасности цунгцванга, можно найти ход, не ухудшающий текущего положения.

13. Делается пропуск хода и очередь хода передается противнику. Поиск выполняется на меньшую глубину.
14. Полученный результат рассматривается как минимум возможной оценки. Если наш минимум уже сравнялся с beta, то поиск можно досрочно прекратить и вернуть beta/
15. Поиск недействительного перемещения выполняется до основного поиска. Так как глубина поиска сокращается существенно (3,…) и число отсечек за beta довольно велико, то общее число исследуемых позиций сокращается и программа может сосчитать глубже.
16. Поиск недействительного перемещения позволяет сосчитать глубже в среднем на 1-2 полухода за тоже время. Кроме того, он проводит какоето подобие статического отсечения и позволяет сгладить эффект горизонта. Эффект горизонта возникает из за того, что программа не успевает до конца исследовать возможные тактические осложнения и поэтому будет неверно ориентироваться в игре.
17. Поиск недействительного перемещения немного изменяет общий стиль игры. Если вызывать недействительное перемещение только по достижении определенной глубины (2 или 4) полухода, то это приведет к лучшему планированию игры.

```
/*  Поиск с отсечкой по результату недействительного перемещения */
Function AlphaBeta(depth,alpha,beta,side,xside,is_null_search:integer):integer;
Var
  Score, best:integer;
Begin
  If depth <= 0 then
    Return Evaluate(side);

  If not InCheck() then
   If not IsEndGame then
    If is_null_search = 0 then
    begin
      Make_Null_Move();
      Score = -AlphaBeta(depth-3,beta-1,beta,side,xside,1);
      UnMake_Null_Move();
      If score >= bets then return beta;
    End;

  Generate_all_moves(side);
  Sort_moves();

  Best = -INFINITY;
  For each move do
  Begin
```

```
    Make_Move(move);

    Score = -AlphaBeta(depth-1,-beta,-alpha,xside,side,0);

    Un_Mack_Move(move);
    If  score > best then
    begin
      Best = score;
      Alpha = max(alpha,best);
      If alpha >= beta then
       Break;
    End;
  End;
  Return best;
End;
```

LMR.

Алгоритм ***LMR*** основан на более углубленном исследовании первых ***3х*** легальных перемещений. ***Остальные*** перемещения рассматриваются на ***меньшую*** глубину и пересчитываются только если оценка превышает alpha.
Почему это работает? Я понимаю LMR как вариант генетического алгоритма.

Например, у нас есть стадо баранов. Нам нужно выделить лучших или даже вывести новую породу. Мы берем N быранов, выбираем из них 3х лучших по нашим критериям. Затем эти 3 лучших плодятся, получается опять N баранов. Из них берем опять 3х лучших и так далее. В результате мы получим какое то пиковое значение или ничего. Исключения, когда число лучших экземпляров первышает 3 возможны, но должны быть обоснованны..

Можно представить задачу и так. Берем N баранов. 3 лучших получают привилегированные условия для размножения – корм, лучших самок и т.д. Остальные – на общих основаниях. Из ***общего*** потомства выбираем 3х лучших и повторяем цикл сначала.

Так можно решить сложную задачу, не имеющую четкого алгоритма. Например, необходимо установить параметры, влияющие на стойкость микросхем к экстремальным условиям. Есть ящик с микросхемами, полигон для испытаний и аппаратура для измерения параметров. Алгоритм может выглядеть примерно так:

Берем N микросхем, подвергаем их испытаниям. Отбрасываем 30% худших и заменяем их новыми из ящика. Повторяем процесс 10 раз, потом смотрим – чем наша партия отличается от общей массы – наличие примесей, марка производителя и .д.

Заметьте – это ничего не гарантирует, но тем не менее, поле поиска существенно ограничивается и есть система для отбора – лучшие и прочие. Вероятность успеха существенно повышается.

LMR и подобные алгоритмы в шахматах могут поразительно чувствовать задачи, основанные на фигурной активности. Но заметьте – это все же не полный перебор и ошибка возможна.

```
/* алгоритм LMR */
Function search(alphja,beta,depth):integer;

 …

 For each move do begin

   If (legal_moves <= 3) or check() then
     Score = -search(-beta,-alpha,depth-1)
   Else begin
     Score = -search(-(alpha+1), -alpha, depth-2);
     If score > alpha then
      Score = -search(-beta,-alpha,depth-1);
   End;

   …
 End;
 Return alpha;
```

```
End;
```

Это несколько вольная трактовка алгоритма. Если вы хотите уточнить нюансы, можно обратиться к специализированным статьям по этой теме. В частности, статьи С.Маркова можно найти в интернете. Как я понимаю, он является официальным автором этого алгоритма.

Остается добавить, что модифицированный вариант LMR используется в большинстве современных шахматных программ. Он очень хорошо себя зарекомендовал в соревнованиях между шахматными движками. Программа играет как бы на ‘удушение’ противника. Такой стиль игры – дело вкуса. Я так понимаю, что REBEL, например, не использует LMR. ChessMaster – вроде бы тоже.

PVS

Так называется классический алгоритм поиска в шахматных программах. Когда вызывается функция поиска от корня дерева перебора, она может возвратить не только лучший ход и его оценку, но и целую последовательность перемещений, являющуюся, по мнению программы, лучшей игрой для обеих сторон (PV). Последовательность перемещений я буду для краткости называть строкой. Например, для начала игры, строка может быть такая:

e2e4 e7e4 g1f3 g8f6 …

Технику извлечения PV (главное изменение) я в свое время посмотрел у Брюса Мориленда. Сайт, сейчас, к сожалению не активен.

Оценка всех перемещений в главном изменении будет больше alpha но меньше beta. Во время поиска будет найдено множество отрывочного PV, которое никогда не попадет в главное изменение. Нам, таким, образом , потребуется буфер в рекурсивной функции поиска для сохранения возможного отрывочного PV. Сразу хочу огвориться, что если у вас есть хэш-таблица, вы всегда сможете ее настроить так, чтобы она запоминала ходы PV и даже 2-3 лучших хода для каждого узла. Мы рассмотрим классический способ извлечения PV. Он не требует хэша и не требователен к размеру памяти.

```
/*  извлечение главной строки изменения */
Type PV_Line = array[0..MAX_PLY-1] of Move_type;

Var pv:PV_Line;

Function PVS_Search(depth,alpha,beta,side,xside:integer;
                    Var pv:PV_Line):integer;
Var
  Score, best:integer;
  Tmp_line:PV_Line;
Begin
  If depth <= 0 then
    Return Evaluate(side);
  Generate_all_moves(side);
  Sort_moves();
  PickPvMove(pv[ply]); // pv ход на верх списка перемещений

  Best = -INFINITY;
  For each move do
  Begin
    Make_Move(move);

    Ply++;
    If(pv_move()) then
      Score = - PVS_Search (depth-1,-beta,-alpha,xside,side);
    Else begin
      Score = - PVS_Search (depth-1,-(alpha+1),-alpha,xside,side);
      If (score > alpha) and (score < beta) then
```

```
      Score = - PVS_Search (depth-1,-beta,-alpha,xside,side);
    End;
    Ply--;

    Un_Mack_Move(move);
    If time_up then break;

    If  score > best then
    begin
      Best = score;
      If best > alpha then
      begin
        Alpha = best;
        If alpha >= beta then
          Break;
        Else begin
          Pv[ply] = mv;
          J = ply+1;
          While (j < MAX_PLY) and (tmp_line[j] <> Zerro_move) do
          Begin
           Pv[j] = tmp_line[j];
           J = j+1;
          End;
          Pv[j] = Zerro_Move;
        End;
      End;

    End;
  End;
  Return best;
End;
```

Ход главного изменения просматривается первым с полным окном alpha – beta. Остальные ходы сначала просматриваются с минимальным окном alpha,alpha+1. Вероятно, эти перемещения не есть хорошие и можно сразу спрогнозировать окно поиска. Если же возникло исключение и перемещение завышает alpha, то оно будет рассмотрено с полным окном для получения точной оценки и возможного отрывка PV. Если результат поиска больше alpha и меньше beta, то найденный отрывок PV запоминается в буфере.

Если такой алгоритм вызывать от корня дерева поиска и постепенно увеличивать глубину (пока не будет исчерпан лимит времени), то каждый последующий поиск будет начинаться с лучшего хода и быдет выполняться значительно быстрее.

```
Function root_search:move_type;
Var pv:PV_Line;

  Generate_and_sort_legal_moves();
  Timer_reset();

  For search_depth = 2 to MAX_PLY-10 do begin
```

```
    Score = -PV_Search(search_depth,-INF,INF,pv);
    If time_ip then break;
  End;
  Return pv[0]; // возвращаем лучший ход
End;
```

Зачем нужен PVS? Во первых, он быстрее. Во вторых, если у нас не полный перебор, а какой то варинт выборочного поиска, (например эвристика недействительного перемещения), то рассматривание главного изменения первым значительно улучшает игру программы.

Все главные изменения, найденные во время поиска, можно добавлять ***в специальную таблицу PV*** и таким образом, мы будем иметь в памяти все малое дерево игры, которую программа собиралась поиграть. Этот простой прием дает неплохой результат для нестабильного выброчного поиска. Он позволяет программе найти хоть что то при том, что она многое отсекает.

Оценка позиции

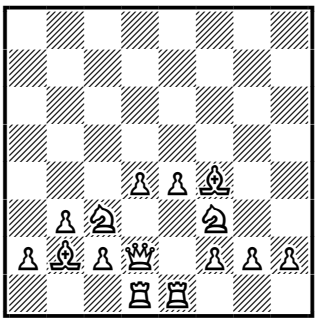
Несмотря на все сложность, оценка в шахматах основывается на нескольких простых принципах:

6. Приоритет центра и фигурная активность
7. Безопасность короля
8. Проходные пешки и структура пешек вообще

Рассмотрим эти принципы подробнее.

Приоритет центра.

Централизация характерна практически для всех игр. Фигура в центре активнее по 2 причинам: она быстрее может достичь любого поля доски и одновременно мешает сделать тоже самое противнику. Особенно сильны в центре недальнобойные фигуры: пешка, конь и даже король в конце игры. Для дальнобойных фигур все не так очевидно, но принцип тот же. Для ладьи обычно центр не премируют. Ладья с любого поля доски атакует одинаковое количество полей и, кроме того, ладья в начале середине игры откровенно боится центра из-за своей ценности и неповоротливости.



Примерная оценка для черных ***пешек*** (они движутся сверху вниз) [1]

```
( 0, 0, 0, 0, 0, 0, 0, 0,
  4, 4, 4, 0, 0, 4, 4, 4,
  6, 8, 2,10,12, 2, 8, 6,
  6, 8,12,16,18,12, 8, 6,
  8,12,16,24,26,16,12, 8,
 12,16,24,32,34,24,16,12,
 12,16,24,32,34,24,16,12,
  0, 0, 0, 0, 0, 0, 0, 0);
```

Здесь мы видим, что пешка сильнее в центре и ближе к полям превращения. Кроме того, пешки теряют, если меняют координуту X на более близкую к краю.
Две центральные пешки наиболее важны. Если у одной из сторон нет центральных пешек, то это очень серьезный недостаток и может быть компенсирован только фигурной активностью.

Конь тоже очень чувствителен к центральным полям. Конь, как известно, ходит буквой **Г** и ему бывает нелегко допрыгать до определенного поля. Поэтому конь в углу доски сильно

[1] *таблица взята из GnuChess*

штрафуется.

```
( 0, 4, 8,10,10, 8, 4, 0,
  4, 8,16,20,20,16, 8, 4,
  8,16,24,28,28,24,16, 8,
 10,20,28,32,32,28,20,10,
 10,20,28,32,32,28,20,10,
  8,16,24,28,28,24,16, 8,
  4, 8,16,20,20,16, 8, 4,
  0, 4, 8,10,10, 8, 4, 0);
```

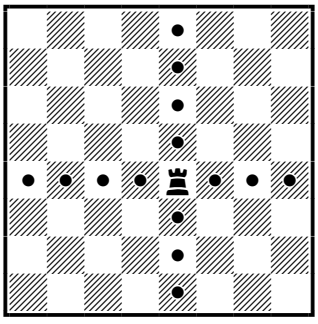
Для коня также бывает сильна позиция форпоста (терминология Нимцовича). Это значит, что конь на половине противника защищен своей пешкой и находится на поле которое не может быть атаковано пешками противника. Взять такого коня нельзя, так как получится проходная пешка. Преимуществом также является, если конь на открытой вертикали и может быть поддержан ладьей. Такой форпост стесняет противника надолго и может служить причиной перелома хода партии.

Для ***слонов*** приоритет центра сочетается со стремлением занять главные диагонали. Слон на главных диагоналях a1h8 и a8h1 имеет несомненное преимущество.

```
(14,14,14,14,14,14,14,14,
 14,22,18,18,18,18,22,14,
 14,18,22,22,22,22,18,14,
 14,18,22,22,22,22,18,14,
 14,18,22,22,22,22,18,14,
 14,18,22,22,22,22,18,14,
 14,22,18,18,18,18,22,14,
 14,14,14,14,14,14,14,14);
```

Из приведенной таблицы видно, что на центральных полях оценка одинакова для того, чтобы слон мог ‘поиграть’. Кроме того, централизация для слона значит немного меньше, чем для коня.

Ладья, как уже было сказано, с любого поля атакует одинаковое количество полей и премия для централизации отсутствует.



Ферзь имеет небольшую премию для центральных полей, но он и боится центра в начале игры из за фигурных угроз противника.

Король имеет премию для центра даже больше, чем конь. В начале-середине партии король боится центра из за матовых угроз.

// Оценка короля для эндшпиля (нет угроз или нет пешек)

```
( 0, 4, 8,12,12, 8, 4, 0,
  4,16,20,24,24,20,16, 4,
  8,20,28,32,32,28,20, 8,
 12,24,32,36,36,32,24,12,
 12,24,32,36,36,32,24,12,
  8,20,28,32,32,28,20, 8,
  4,16,20,24,24,20,16, 4,
  0, 4, 8,12,12, 8, 4, 0);
```

Фигурная активность.

Под фигурной активностью подразумевается прежде всего ***мобильность***. Также могут учитываться некоторые специфические для шахмат моменты:

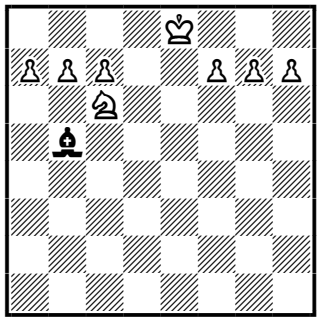
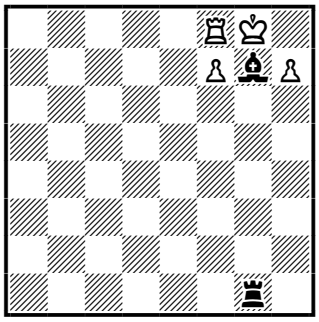
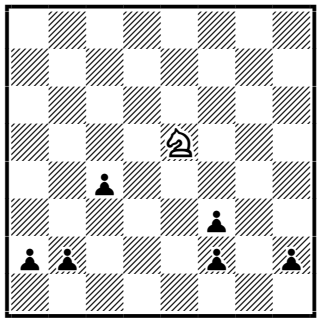
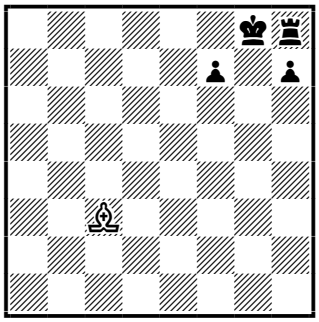
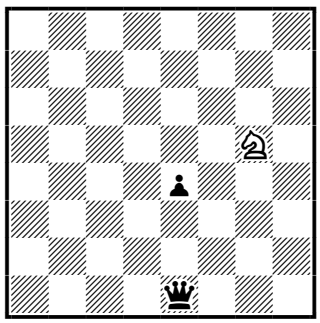
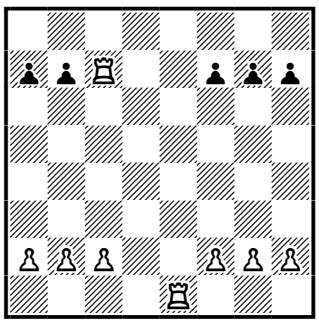
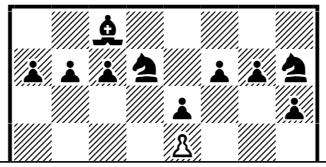
18. ***Связка***. Когда фигура атакована дальнобойной фигурой противника и не может уйти так как открывает короля (мертвая связка) или более ценную фигуру.
19. ***Вскрытый шах*** и ***X-Ray*** атака вообще. Это когда фигура после хода освобождает линию для атаки дальнобойной фигуры. Известный прием ‘***мельница’*** как раз основан на вскрытом шахе.
20. ***‘Вилка’*** или атака на 2 фигуры одновременно. Особенный специалисты по вилкам – конь и пешка.
21. ***Зависающая фигура –*** атакована и не защищена или атакована менее ценной фигурой. Это не является существенным преимуществом, но для динамической характеристики позиции может быть полезна.
22. ***Защитник более ценный, чем нападающий –*** это чисто программисткий трюк для придания некоторой остроты и шаловливости в игре.
23. Для ладей существенными являются следующие бонусы: занятие открытой вертикали и 7-ой горизонтали. Две ладьи на 7-ой горизонтали – это, как правило, выигрыш. Там ладья стесняет противника, осуществляет фланговую атаку на пешки и угрожает королю на 8-ой горизонтали. Для ладей также важна консалидация – когда 2 ладьи поддерживают друг друга.
24. Ферзь может иметь некоторую премию для начальной мобильности, но основной его целью является король оппонента.
25. Конь кроме центра также стремится быть ближе к королю оппонента.

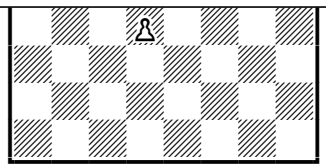
Рассмотрим мобильность подробнее. Она считается для дальнобойных фигур путем подсчета кол-ва полей, на которые фигура может походить. Далее существуют специальные таблицы для каждой фигуры, по которым кол-во ходов переводится в премию по мобильности. Особенно интересной особенностью является уменьшение роста премии для последней трети таблицы. Это ведет к тому, что программа вынуждена развивать другие фигуры, чтобы достичь максимальной оценки (консолидация сил).
Вот примерная таблица для слона:

-3, 1, 3, 5, 7, 8, 9, 10, 10, 11, 11, 11, 12,12,12,12,12

Существует подход, при котором мобильности даются огромные премии. Это хорошо срабатывает для сложных тактических позиций, когда невозможно просчитать варианты до конца и все, в конечном счете, упирается в фигурную активность. Я лично сторонник реалистичной оценки – как человек бы оценил определенные факторы, так и программа

считает.

	
Конь С6 ‘связан’ (PIN)	‘Вскрытый шах’ (X-Ray). Слон, уходя с g6, объявляет мат.
	
Конь e5 нападает одновременно на 2 незащищенные пешки f3 и c4.	Ладья h8 атакована менее ценной фигурой. Кроме того, ей некуда уйти.
	
Конь g5 атаковал пешку e4. Она хоть и защищена, но ферзь – более ценная фигура, чем конь. Он ‘перегружен’ защитой пешки и его реальная мобильность ограничена.	Ладьи стремятся занять открытую вертикаль и 7-ю горизонталь. Основная их цель – фланговая атака на пешки и угроза королю.
	

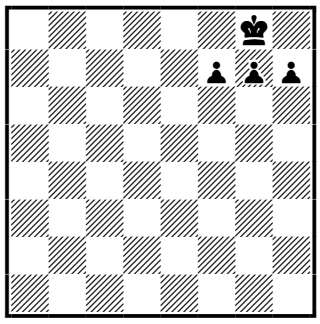
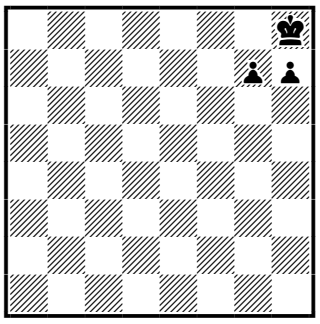
 У черных 2 неактивные фигуры. Слон c8 – у него ограничена мобильность и конь h7 – он заперт в углу доски.	

Безопасность короля.

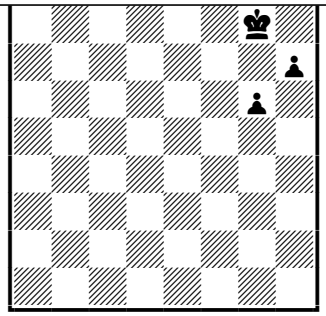
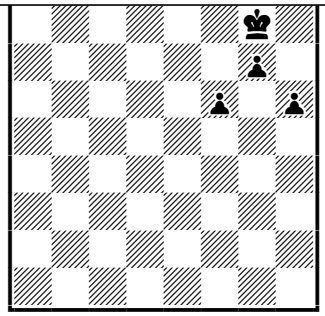
Безопасность короля является вторым после центра приоритетом в игре. Вообще то она является первым и главным приоритетом, но без центра и развитых фигур, как показала практика, не может быть эффективной атаки на короля. [2] Безопасность короля для шахматной программы можно разбить на несколько пунктов:

1. Король боится центральных полей в начале-середине игры.
2. Пешечный экран короля
3. Близость ферзя и коня к королю.
4. Король на открытой вертикали (без пешек)
5. Фигурное давление в квадрате короля
6. Открытость короля для шахов

Если у противника нет ферзя, то безопасность короля в среднем улучшается.
Рассмотрим пешечный экран короля подробнее. Наиболее крайний случай, когда короля не прикрывает ни одна пешка. Это очень плохо. Пешка как наименее ценная фигура в фигурных разменах чаще дает выигрыш материала. Кроме того, пешка 2-мя своими диагональными атаками отбирает у противника 2 поля в непосредственной близости от короля. Все это делает пешку лучшим защитником.

 Идеальный пешечный экран короля. Нет слабых поле на f6,g6,h6 и нет открытых линий для атаки. Кроме того, есть возможность пешечного маневра. Это ценно, так как пешки не ходят назад.	 Тоже приемлемо. Линия g открыта, но это делается, как правило, для активизации ладьи.

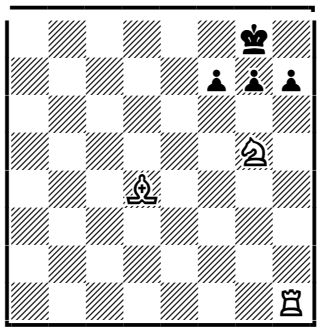
[2] А. Нимцович

	
Неприемлемое положение. Открыта диагональ a1h8 и если у противника есть чернопольный слон, это может усугубить ситуацию. Кроме того, поля h6, f6 являются слабыми и могут быть заняты фигурами противника. Такое пешечное положение допустимо только при малой фигурной угрозе противника.	Все три пешки в наличии, но положение неважное. Слабое поле g6 является отличной целью для фигур противника.

Если пешки экрана короля являются слабыми (изолированными, отсталыми), то это существенно снижает безопасность.

Фигурное давление в квадрате короля можно измерять 2я способами. Можно, как в диаграммах Бронштейна, просто считать количество атак противника в каждую клетку. Можно также делать разностную оценку:

Если кол-во атак оппонента в клетку **больше** кол-ва защит **То**
Поле является слабым



В приведенной позиции по Бронштейну будет 4 атаки в квадрат короля, а при разностной оценке – только 1 слабое поле h6, так как оно защищено только королям, а атаковано конем и ладьей. Мне кажется, что по Бронштейну точнее.

Открытость для шахов определяется как количество возможных шахов в позиции. Затем используется таблица с прогрессивной оценкой, по которой количество шахов переводится в штраф.

Пример таблицы:

0, 6, 6*2+2, 6*3+4, 6***4+6, 6***5+4, 6*6+2, 6*7, 6*8-2, ...

Первые 4 шаха штраф растет прогрессивно, потом я позволил себе фривольность и стал постепенно уменьшать рост. Это сделано для того, чтобы программа могла ‘поиграть’ в

формально плохих позициях. Например, если ферзь один без поддержки, то король боится открытости для шахов, но до известной степени. Потом ему становится ‘все равно’ и он может играть и преследовать свои цели.

Структура пешек.

Пешки в шахматах можно условно подразделить на несколько категорий:

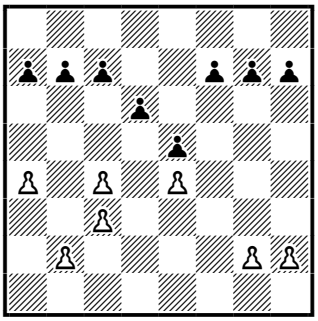
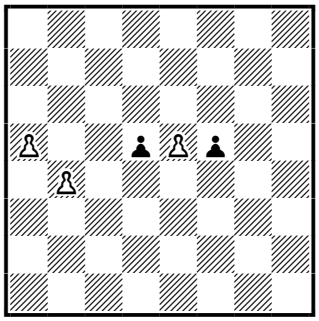
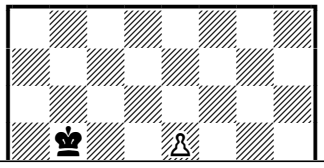
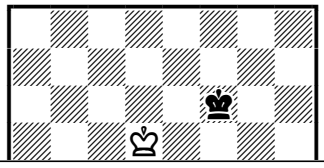
6. Сдвоенные
7. Отсталые
8. Изолированные
9. Проходные

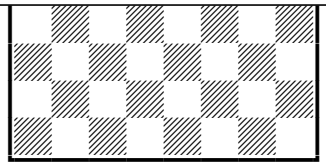
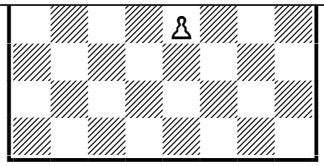
Сдвоенным пешкам дается небольшой штраф. Основная причина – ограничение мобильности. Известна примечательная партия Алехина, где он строил проходные центральные пешки и выиграл. Это лишний раз доказывает, что величина штрафа должна быть небольшой.

Отсталые пешки могут награждаться минимальным штрафом. Это дело вкуса. Основная суть состоит в том, чтобы премировать каждую фигурную атаку на слабую пешку противника. Без этого велика вероятность, что программа даже отдаленно не будет понимать, что такое шахматная игра.

Изолированные пешки являются существенным недостатком в позиции. Они отвлекают фигуры для защиты и могут являться легкой добычей противника. Простейший путь к победе состоит в том, чтобы разменами испортить пешечную структуру противника, а затем постепенно отыграть слабые пешки.

Проходные пешки получают прогрессивную премию в зависимости от близости к полю превращения. Премия эта увеличивается ближе к эндшпилю. Когда на доске только короли и пешки, существенным является ‘***правило квадрата***‘ . Оно гласит, что если король дальше от поля превращения, чем проходная (вне квадрата), то он не успевает ее задержать.

<table>
<tr>
<td> 
Пешка b2 является отсталой, пешка e4 – изолированной, пешки c3,c4 – сдвоенными.</td>
<td> 
Пешка e5 является проходной, пешки a5,b4 – связанными проходными</td>
</tr>
<tr>
<td>  </td>
<td>  </td>
</tr>
</table>

	
Король в квадрате пешки, он успевает ее задержать.	Король, сопровождая свою проходную пешку, стремится встать впереди ее. Король противника стремится этому помешать.

Материальная оценка.

Материальная оценка есть средняя стоимость фигур, не учитывающая многих факторов. Любой игрок при размене должен представлять примерную ценность фигур. Вот оценка по Капабланке:

1. Слон, Конь – 3 пешки
2. Ладья - 6 пешек
3. Ферзь – Ладья + Легкая фигура

То, что годится для человека, бывает трудно объяснить машине. В среднем, оценка по Капабланке годится для оценочной функции только в эндшпиле. В самом деле – ладьей мат ставится также легко как и двумя слонами, а ферзь уступает в силе двум ладьям, когда он один и линии для ладей открыты.

Рассмотрим традиционную материальную оценку для шахматных программ. Это, конечно, не догма.

1. Пешка – 100
2. Конь - 350
3. Слон - 360
4. Ладья – 540
5. Ферзь - 1100 !?

Эта оценка примерно актуальна для начала-середины игры. Она позволяет избежать 2х распространенных ошибок:

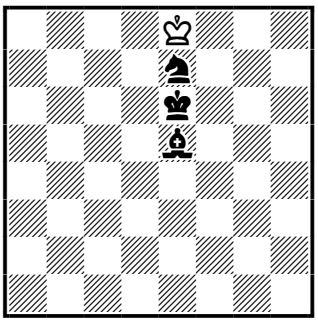
1. Размена 2х легких фигур на ладью и пешку. В начале партии лишние 2 легкие фигуры быстро отыграют материал.
2. Размена 3х пешек на легкую фигуру. Такой размен должен быть обоснован. 3 центральные пешки + активность в атаке вполне могут стоить легкую фигуру, но в большинстве случаев это неоправданно.

Остается проблема с ферзем. Одинокий ферзь без поддержки не стоит 2х ладей, но в начале игры, ферзем, безусловно, жертвовать нельзя.
Также необходимо учитывать некоторые случаи окончаний:

1. Два коня мат поставить не могут и следовательно, в окончании ***2 коня и король против короял*** материальная оценка коней – 0.
2. Слон и конь мат ставят, но ст трудом и стоят меньше ладьи.
3. Одна легкая фигура не стоит ничего, если нет пешек и у противника нет возможности поставить мат.

Мат конем и слоном.

Если на доске сложилось подобное окончание, то программа должна уметь поставить мат одинокому королю. Алгоритм матования примерно такой: Нужно оттеснить короля сначала к краю доски, а потом в угол, который может взять под бой слон. При этом нужно стараться держать фигуры ближе друг к другу и ключевой является, видимо, подобное расположение:

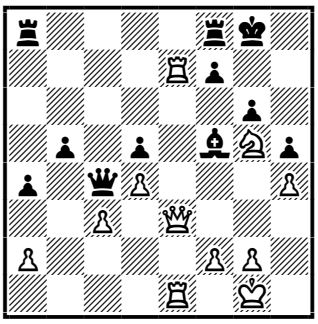


В данной позиции короля нужно оттеснить в угол h8. Для программы этих знаний должно хватить при достаточно глубоком поиске. Обратите внимание, что в подобных окончаниях позиционная оценка для сильнейшей стороны имеет небольшое значение.

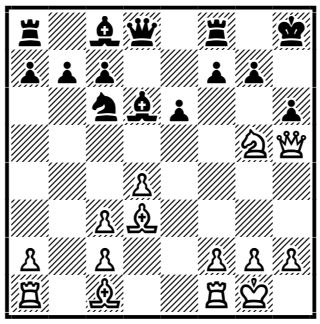
Тестовые позиции.

Чтобы быстро оценить тактические способности программы, бывает полезно заставить ее порешать тестовые задачи. Это, конечно, не даст полное представление о ее стиле игры и планировании перемещений.
Я тут привел несколько позиций, на которых спотыкалась моя программа. Если ваш движок решает их и решает быстро, то он будет быстро находить матовые комбинации и иметь неплохую тактическую прочность в острых положениях.

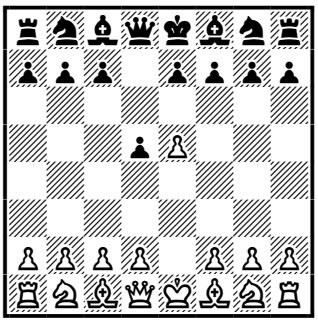
1 Задачи на фигурную активность и эффект горизонта



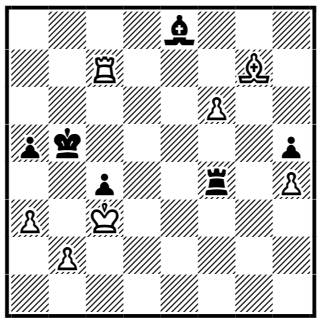
Ход черных. Это простая задача на глубину и интеллектуальность поиска. Если белые возьмут ферзем пешку a2, то последует Фе5! С последующим разменом по f7 и занятием 7ой горизонтали. Пешка того не стоит. Верное решение за черных Ла8-с8, но пойдет и а4-а3.



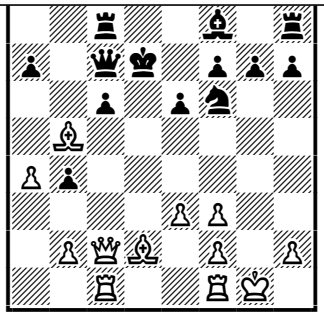
Ход черных. Эта задача на фигурную активность. Белые угрожают конем на h7 и f7. Единственное верное решение – Фе8.



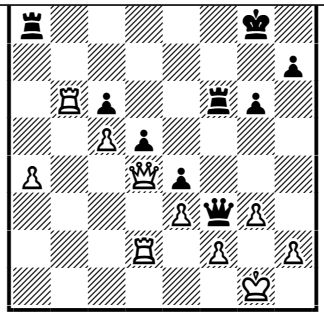
Ход черных. Это дебютная ловушка. Белые сделали ошибку, продвинув пешку на e5, нужно лишь найти правильный ответ. Лучший ход – с7-с5, но пойдет и Cf5 и даже e7-e6. Но никак уж не Кс6.



Ход белых. Выигрывает Kd2! Но сойдет и а3а4.



Та интересная задача возникла в реальной партии. Первое, что напрашивается за белых, это Лf1-d1! Усиливает централизацию и давление на черного короля. Позволяет, не ввязываясь в драку, тихим выжидательным ходом выиграть всю партию. Машина упрямо стремилась сделать размен Фc2:c6? Тестирование с другими движками показало: Genius находил решение практически сразу, Fritz находил, но как всегда, медленно.



Лучшее решение за белых, безусловно, Лd2-b2. Вроде бы выигрывает и Лb6:c6!? С последующим Ф:d5 Программа этого не видит, так как так глубоко не считает.

Пример шахматной программы.

Шахматное программирование - область поистине необъятная и поэтому я решил выбрать главное. Данная программа служит главным образом введением в тему для тех кто не знаком с предметом. Многие вопросы остались за кадром. Объем кода и так стал довольно значительным.

Шахматная игра представляет собой в сущности не дерево поиска, а граф. Для реализации графа служат в основном хеш таблицы. Выдергивание конкретных позиций из контекста стало источником поистине неисчерпаемых ошибок.

Для простой игровой программы совсем не обязательно вникать в эти сложные и запутанные вопросы. Гораздо приятнее сосредоточиться на более интересных аспектах шахматной игры.

Примечание: Данный код хоть и написан по мотивам работающей программы и GnuChess, но однако, специально не тестировался и возможно наличие неточночтей.

Генерация перемещений

Структуры данных.

Первым делом нужно определиться со структурами данных. Основные элементы следующие:

1. Фигуры
2. Цвет сторон
3. Перемещение
4. Позиция (доска 8*8 клеток)
5. История игры (список перемещений и доп., информация)
6. Массив атаки (нужен для определения атаки на поле)

```
#define  PAWN 1
#define  KNIGHT 2
#define  BISHOP 3
#define  ROOK 4
#define  QUEEN 5
#define  KING 6
```

```
#define  WHITE 0
#define  BLACK 1
#define  NEUTRAL 2


union Move{

  unsigned to:6;
  unsigned from:6;
  unsigned piece:3;      //12
  unsigned cap_piece:3;  //15
  unsigned kind:8;       //18
  unsigned prom_piece:3; //26
};

//kind
#define LEFT_CASTL  1
#define RIGHT_CASTL 2
#define CAPTURE     4
#define EN_PASSANT  8
#define PROMOTE     16


PACKED_STRUCT Position{

  Move in_mv;     // ход в позицию
  int  side;      // кто ходит
  int  piece[64],color[64]; // фигуры - цвет
  int  can_castl[2];        // возможна ли рокировка
};
```

  Это не самое оптимальное представление данных, но одно из самых понятных. За основу я взял GnuChess (сильно урезанный). Кто действительный автор - кто знает. Может Jon Stanback?. Некоторые моменты все же требуют пояснений. Тип NEUTRAL взят не 0 а 2 для более удобной индексации массивов от 0. Тип Move (union - битовые поля) это не самый удачный выбор в силу того, что на разных машинах порядок битов может быть различным. Я решил этим пренебречь для простой программы.

  Остановимся подробнее на типе Position. Основная задача здесь - это удобство доступа и однозначное определение конкретной позиции с учетом специфики шахматной игры. Я придерживался следующего правила:

  Позиции являются одинаковыми, если фигуры находятся на тех же местах и списки перемещений из данных позиций идентичны. Два момента неприятно влияют

на сравнение позиций. Это взятие пешки на проходе (нужен ход оппонента пешкой через клетку в данную позицию) и генерация рокировок. Напомню правило: между королем и ладьей не должно быть фигур, клетки от старого до нового места короля включительно не должны быть атакованы, и, король и ладья должны находиться на исходных местах и не разу не перемещаться. Для совсем простой программы сойдет правило: рокировка не делается под шахом.

Если мы можем точно сравнить две позиции, это избавит от многих проблем в будущем. Алгоритм сравнения примерно такой:

1. Если ход в позицию пешкой через клетку и ее можно взять на проходе - то позиции сравниваются побайтово.
2. В остальных случаях позиции сравниваются без учета хода in_mv.

Случаи повторов в игре рассматриваются отдельно. То есть - история игры не влияет на формальное сравнение двух позиций.

Начальная позиция в игре будет инициализирована следующим образом:

```
rnbqkbnr
||||||||
........
........
........
........
PPPPPPPP
RNBQKBNR
```

```
struct Position start_pos = {
  0,
  WHITE,
  {
    4,2,3,5,6,3,2,4,
    1,1,1,1,1,1,1,1,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,
    1,1,1,1,1,1,1,1,
    4,2,3,5,6,3,2,4

  },
  {
```

```
    1,1,1,1,1,1,1,1,
    1,1,1,1,1,1,1,1,

    2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,

   },
   {LEFT_CASTL|RIGHT_CASTL, LEFT_CASTL|RIGHT_CASTL}
  };
```

Перемещения дальнобойных фигур.

Для начала рассмотрим, как получить перемещения слона. Принцип аналагочный и для других фигур. Слон ходит по диагоналям на любое число полей до первой фигуры. Если это фигура противника, то данное поле включается в список перемещений.

```
struct Position P;

#define MAP(sq)    ((sq&~7)<<1) | (sq&7))
#define UNMAP(sq)  ((sq&~7)>>1) | (sq&7))

int bishop_dir[] = { 17, 15, -15, -17, 0 };
#define Y 16
int knight_dir[] = { 2*Y+1, 2*Y-1, 2+Y, -2+Y,
                     2-Y, -2-Y, -2*Y+1, -2*Y-1, 0};
#undef Y
int rook_dir[]   = {16,1,-1,-16};
int queen_dir[]  = {17,15,16,1,-1,-16,-15,-17,0};
int *king_dir    = queen_dir;


/* получает перемещения
   дальнобойных фигур
*/
int brq_moves(int piece, int sq, int c,
              int dir[], int list[]){
 int j,n,u,*start = list;

  for(j = 0; dir[j]; j++)
   for(n = MAP(sq) + dir[j];
       (n & 0x88)==0;
       n += dir[j])
   {
     u = UNMAP(n);
```

```
    if(P.color[u]==NEUTRAL)
      *list++ = (piece<<12) | (sq<<6) | u;
    else{
      if(P.color[u]!=c)
        *list++ = (CAPTURE<<18) | (P.piece[u]<<15) |
                (piece<<12) | (sq<<6) | u;
      break;
    }
  }
  return list-start; // количество ходов
}
```

Если белый слон стоит на C1, то вызов функции будет следующий:

```
   moves += brq_moves(BISHOP, C1, WHITE,
              bishop_dir, (int*)list_of_moves);
```

Приращения при движении фигуры задаются в размерности 8*16 а не 8*8. Это делается для облегчения определения выхода за пределы доски. Если мы рассмотрим битовое представление координаты 0..63, то увидим, что оно включает X-координату и Y-координату и они никогда не пересекаются, так как их допустимые значения лежат в диапазоне 0..7.

```
  0b111000  маска для Y
  0b000111  маска для X
```

Макрос MAP сдвигает Y влево на 1.

```
  0b1110000  маска для Y после сдвига
```

Таким образом, между X и Y образуется 1 дополнительный бит (0b0001000), который будет установлен, если X > 7. Для проверки переполнения Y служит бит 0b10000000. Общая маска переполнения будет 0b10001000 или 0x88 в шестнадцатеричной системе счисления.
Если X или Y меньше 0, тогда аналогично установятся биты 0 или 7, и можно опытным путем убедиться, что выход за пределы доски будет корректно отслеживаться.

Премещения коня и короля.

Очень похоже на предыдущую функцию, только линия не строится.

```
int nk_moves(int piece, int sq, int c,
             int dir[], int list[]){
 int j,n,u,*start = list;

  for(j = 0; dir[j]; j++)
   if(n = ((MAP(sq) + dir[j])&0x88)==0)
   {
     u = UNMAP(n);
     if(P.color[u]==NEUTRAL)
       *list++ = (piece<<12) | (sq<<6) | u;
     else if(P.color[u]!=c)
        *list++ = (CAPTURE<<18) | (P.piece[u]<<15) |
                  (piece<<12) | (sq<<6) | u;
   }
   return list-start; // количество ходов
}
```

Переремещения пешек.

Пешки ходят вперед на одну клетку и с начальной позиции могут переместиться на 2 клетки. Бьют на 1 клетку вперед по диагоналям. Если пешка сделала первый ход на 2 клетки и пересекла 'битое' поле, то ее можно взять 'на проходе' - EN_PASSANT. Это единственный случай, когда поле взятия не совпадает с полем перемещения.

```
    a2-a4  b4xa3!
```

Дойдя до последнего поля пешка может превратиться в любую(?!) фигуру. Для корректной игры достаточно ферзя и коня. Таким образом - получается 2 хода на одно поле. Правила несколько избыточны, но приходится им следовать.

```
    a7-a8q
    a7-a8n
```

```
#define ROW(sq) (sq>>3)   // sq / 8
#define COLUMN(sq) (sq&7) // sq % 8

/*ходы белых пешек */
int white_pawn_moves(int sq, int list[]){
   int *start = list;


   switch(ROW(sq))
   {
     case 1: // превращения пешки
     {
```

```
      if(P.color[sq-8]==NEUTRAL) //ход вперед
       *list++ = (QUEEN<<26)|(PROMOTE<<18) | (PAWN<<12) |
                 (sq<<6) | (sq-8);
       *list++ = (KNIGHT<<26)|(PROMOTE<<18) | (PAWN<<12) |
                 (sq<<6) | (sq-8);

      if(COLUMN(sq)>0) //взятия влево
       if(P.color[sq-8-1]==BLACK)
        *list++ = (QUEEN)<<26)|(CAPTURE<<18) | (P.piece[sq-8-1]<<15) |
                  (PAWN<<12) | (sq<<6) | (sq-8-1);
      if(COLUMN(sq)<7) //взятия вправо
       if(P.color[sq-8+1]==BLACK)
        *list++ = (QUEEN<<26)|(CAPTURE<<18) | (P.piece[sq-8+1]<<15) |
                  (PAWN<<12) | (sq<<6) | (sq-8+1);
    }break;

    case 7:  // первый ход пешкой через клетку
      if(P.color[sq-8]==NEUTRAL)
       if(P.color[sq-16]==NEUTRAL)
         *list++ = (PAWN<<12)|(sq<<6)|(sq-16);
    default: // ход пешкой и взятия
    {
      if(P.color[sq-8]==NEUTRAL)
       *list++ = (PAWN<<12)|(sq<<6)|(sq-8);

      if(COLUMN(sq)>0)
       if(P.color[sq-8-1]==BLACK)
        *list++ = (CAPTURE<<18) | (P.piece[sq-8-1]<<15) |
                  (PAWN<<12) | (sq<<6) | (sq-8-1);
      if(COLUMN(sq)<7)
       if(P.color[sq-8+1]==BLACK)
        *list++ = (CAPTURE<<18) | (P.piece[sq-8+1]<<15) |
                  (PAWN<<12) | (sq<<6) | (sq-8+1);

    }

   }//switch

   return list-start;
}
```

Взятие пешки на проходе.

Если пещка сделала первый ход на 2 клетки м пересекла поле 'битое' пешкой противника, но ее можно взять 'на проходе'.

Рокировки.

Как уже говорилось, рокировка возможна только если король и ладья ни разу не перемещались и поля от старого до нового места короля (включительно) не атакованы противником. Между королем и ладьей в исходных позициях не должно быть фигур.

Рокировки в длинную и короткую стороны.

```
Ke1-c1 (o-o-o)
Ke1-g1 (o-o)
```

Генератор ходов.

Сводим все вместе.

```
int generate_all_moves(int c, int list[]){

  int cnt = 0;
  int sq;

  if(c==BLACK)
  {
    for(sq=0; sq<64; sq++)
     if(P.color[sq]==BLACK))
      switch(P.piece[sq])
      {
        case BISHOP:
            cnt += brq_moves(
                  BISHOP, sq, c,
                  bishop_dir, list + cnt);
        break;

        case ...  // остальные фигуры


      }


  }else{

     for(sq = 63; sq >= 0; sq--)
       if(P.color[sq]==WHITE)
        switch(P.piece[sq])
        {

          ...
        }
```

```
  }

  return cnt;
}
```

Position в целом, нсколько избыточен по размеру и оптимальнее применить BitBoard.

Изменения в позиции.

Мы рассмотрели тип Position, минимально характеризующий конкретную позицию. Для игры этого, однако, недостаточно. Нам придется ввести еще один тип, содержащий дополнительные служебные переменные.

```
struct PosInfo{

  int piece_cnt[2][8];  // количество фигур
  int mtl[2];           // материал
  int static_score[2];
  int pawn_column_cnt[2][8];
  int king_sq[2];
} PI;

extern PACKED_STRUCT Position P; //старый тип данных
```

Все объекты у нас будут статическими и поэтому методы можно не делать членами какого то класса.

Изменения в позиции удобно отслеживать с помощью отдельных процедур. Это поможет собрать ветвящиеся и расходящиеся ветви алгоритмов программы в одной точке. Удобно осуществлять отладку и проверочный контроль на стадии разработки.

```
extern int value[6];
extern int score_table[2][8][64];
#define COLUMN(sq)  ((sq)&7)

/* вставляет фигуру в текущую позицию на свободную
клетку*.

insert_piece(p,c,sq){

  P.piece[sq]=p;
```

```
  P.color[sq]=c;

  PI.piece_cnt[c][p]++;
  PI.mtl[c] += value[p];
  PI.static_score[c] += score_table[c][p][sq];
  PI.pawn_column_cnt[c][COLUMN(sq)]++;

  if(p==KING)
   PI.king_sq[c] = sq;
}


/*удаляет фигуру с доски*/

remove_piece(sq){

  int p = P.piece[sq];
  int c = P.color[sq];

  P.piece[sq]=0;
  P.color[sq]=NEITRAL;

  PI.piece_cnt[c][p]--;
  PI.mtl[c] -= value[p];
  PI.static_score[c] -= score_table[c][p][sq];
  PI.pawn_column_cnt[c][COLUMN(sq)]--;
}
```

Осталось написать функции MakeMove, UnMakeMove - делающие пошаговую корректировку перемещения.

Наш тип данных Move не имееет достаточно полей чтобы вместить все служебные переменные. Нам потребуется еще один вспомогательный тип данных.

```
struct MoveInfo{

  ...

  int can_castl[2];
  int in_mv;
};

#define MAX_GAME 400
#define MAX_PLY  60

Move game_list[MAX_GAME+MAX_PLY];
MoveInfo info_list[MAX_GAME+MAX_PLY];

int ply;      //глубина поиска (default - 0)
```

```
int game_cnt; //текущая позиция в игре

int side, xside;


struct PosInfo{
   ...

   //add
   int left_castl[2], right_castl[2];

}PI;

struct Position{

  ... // см. ранее
}P;



/*делает перемещение*/
MakeMove(Move mv){

 int i = game_cnt + ply;

 info_list[i].can_castl[WHITE] = P.can_castl[WHITE];
 info_list[i].can_castl[BLACK] = P.can_castl[BLACK];
 info_list[i].in_mv = P.in_mv;

 if(mv.piece==PAWN && move_before_en_pass(mv))
   P.in_mv = mv;
 else
   P.in_mv = 0;


 game_list[i] = mv;

 remove_piece( mv.from );

 if(mv.kind & CAPTURE)
 {
   int sq = mv.to;
   if(mv.kind & PROMOTE)
     if(side==WHITE) sq += 8;
     else sq -= 8;

   remove_piece( sq );
 }


 insert_piece( (mv.kind & PROMOTE) ? mv.prom_piece :
                mv.piece,
```

```
            side, mv.to );


 if(mv.kind & LEFT_CASTL){
   remove_piece(mv.from-4);
   insert_piece(ROOK, side, mv.from-1);

   P.can_castl[side] = 0;
   PI.left_castl[side] = 1;

 }else if(mv.kind & RIGHT_CASTL){
   remove_piece(mv.from+3);
   insert_piece(ROOK, side, mv.from+1);

   P.can_castl[side] = 0;
   PI.right_castl[side] = 1;

 }else if(mv.piece==ROOK){ //нет одной рокировки

   if(COLUMN(mv.from) == 0) //примерно так
     P.can_castl[side] &= ~LEFT_CASTL;
   else if(COLUMN(mv.from)==7)
     P.can_castl[side] &= ~RIGHT_CASTL;

 }else if(mv.piece==KING){ //нет рокировок

   P.can_castl[side] = 0;
 }

 // стороны в игре меняются местами
 side ^= 1;
 xside ^= 1;
 P.side ^= 1;


}


/*отменяет перемещение*/
UnMakeMove(Move mv){

  // все в обратном порядке
 int i = game_cnt + ply;

 P.can_castl[WHITE] = info_list[i].can_castl[WHITE];
 P.can_castl[BLACK] = info_list[i].can_castl[BLACK];
 P.in_mv            = info_list[i].in_mv;

 remove_piece( mv.to );

 if(mv.kind & CAPTURE)
 {
```

```
  int sq = mv.to;
  if(mv.kind & PROMOTE)
    if(side==WHITE) sq += 8;
    else sq -= 8;

  insert_piece( mv.cap_piece, xside, sq );
 }


 insert_piece( mv.piece, side, mv.from );


 if(mv.kind & LEFT_CASTL){
  remove_piece(mv.from-1);
  insert_piece(ROOK, side, mv.from-4);

  PI.left_castl[side] = 0;
 }else if(mv.kind & RIGHT_CASTL){
  remove_piece(mv.from+1);
  insert_piece(ROOK, side, mv.from+3);

  PI.right_castl[side] = 0;
 }

 // стороны в игре меняются местами
 side ^= 1;
 xside ^= 1;
 P.side ^= 1;


}
```

Эти функции были бы тривиальны, если бы не рокировки и прочие нестандартные перемещения.

Буфер перемещений.

У нас есть 2 переменные, отражающие текущую позицию в игре.

games_cnt - сколько ходов было сделано(и запомнено в game_list)
ply - глубина поиска

Если поиска нет, то последний ход (в данную позицию)

будет game_list[game_cnt-1].
Если идет поиск - game_list[game_cnt+ply-1].

Введем буфер, куда будут генерироваться перемещения.

```
Move tree[MAX_PLY*100];
int  tree_cnt[MAX_PLY];
```

Массив tree_cnt содержит счетчики количества перемещений
для каждого ply. Если например мы сгенерировали ходы
для ply = 0, то перемещения будут располагаться в буфере
tree[tree_cnt[0]]..tree[tree_cnt[1]].

```
extern int generate_all_moves(int c, int list[]);
```

```
/*генерирует перемещения в буфер*/
generate(){

  tree_cnt[ply+1] = tree_cnt[ply] +
      generate_all_moves(side, &tree[tree_cnt[ply]]);

}
```

Определение атаки на поле

Мы используем один из самых простых приемов. Чтобы не
искать понапрасну дальнобойные фигуры, могущие бить
данное поле, введем 4 массива, показывающие вероятную
атаку.

Суть дела вот в чем. Наще поле sq [0..63] имеет
координаты X и Y для представления доски 8*8.

1. Если есть ферзь или ладья с такой же координатой
   X, то данное поле может быть атаковано по вертикали.
2. Если есть ферзь или ладья с такой же координатой
   Y, то поле может быть атаковано по горизонтали
3. (X+Y) (QUEEN | BISHOP) ----> восходящие лиагонали
4. (X-Y) (QUEEN | BISHOP) ----> нисходящие диагонали

```
struct PosInfo{

  ...
```

```
  //add
  int atk_x[2][8], atk_y[2][8], atk_x_plus_y[2][16],
      atk_x_sub_y[2][16];

};

#define COLUMN(sq) (sq&7)
#define ROW(sq)    (sq>>3)

insert_piece(p,c,sq){

  ...
  //add
  if(p==QUEEN || p==ROOK)
  {
    PI.atk_x[c][COLUMN(sq)]++;
    PI.atk_y[c][ROW(sq)]++;
  }
  if(p==QUEEN || p==BISHOP)
  {
    PI.atk_x_plus_y[c][COLUMN(sq)+ROW(sq)]++;
    PI.atk_x_sub_y[c][COLUMN(sq)-ROW(sq)+7]++;
  }
}


remove_piece(sq){

  ...
  //add
  if(p==QUEEN || p==ROOK)
  {
    PI.atk_x[c][COLUMN(sq)]--;
    PI.atk_y[c][ROW(sq)]--;
  }
  if(p==QUEEN || p==BISHOP)
  {
    PI.atk_x_plus_y[c][COLUMN(sq)+ROW(sq)]--;
    PI.atk_x_sub_y[c][COLUMN(sq)-ROW(sq)+7]--;
  }
}



/*сканирует линию до первой фигуры*/
int scan_line(map_sq,d,c){

  register int u;

      for(map_sq+=d; (map_sq & 0x88)==0; map_sq+=d)
      {
        u = UNMAP(map_sq);
```

```
        if(P.color[u]==c)
          return P.pos[u];
        if(P.color[u] != NEUTRAL) break;
      }
      return 0;
}

#undef min
#undef abs
int min(a,b){ return a<b ?  a : b; }
int abs(a)  { return a<0 ? -a : a; }

/* расстояние между полями на доске*/
int Distance(a,b){

  return  min( abs(COLUMN(a)-COLUMN(b)),
               abs(ROW(a)-ROW(b))
             );

}


/*определяет атаку на поле*/
int sq_attack(sq,c){
   int u,n,p;
   int map_sq = MAP(sq);


    //пешки
    if(c==WHITE)  u = sq+8;
    else u = sq-8;

    if(COLUMN(sq)>0 && P.pos[u-1]==PAWN &&
      P.color[u-1]==c)
        return PAWN;
    if(COLUMN(sq)<7 && P.pos[u+1]==PAWN &&
      P.color[u+1]==c)
        return PAWN;

    //дальнобойные фигуры
    if(PI.atk_x[c][COLUMN(x)]){
      //вверх
      if((p=scan_line(map_sq,-16,c))==QUEEN ||
          p==ROOK) return p;

      //линия вниз
      if((p=scan_line(map_sq,16,c))==QUEEN ||
          p==ROOK) return p;
    }


    if(PI.atk_y[c][ROW(sq)])
```

```
    {
      // линии влево и вправо
     if((p=scan_line(map_sq,-1,c))==QUEEN ||
        p==ROOK) return p;
     if((p=scan_line(map_sq,1,c))==QUEEN ||
        p==ROOK) return p;
    }

    //диагонали
    if(PI.atk_x_plus_y[c][COLUMN(sq)+ROW(sq)])
    {

      // восходящие (a1-h8)
      if((p=scan_line(map_sq,-15,c))==QUEEN ||
        p==BISHOP) return p;
      if((p=scan_line(map_sq,15,c))==QUEEN ||
        p==BISHOP) return p;

    }
    if(PI.atk_x_sub_y[c][COLUMN(sq)-ROW(sq)+7])
    {

      // нисходящие (a8-h1)
      if((p=scan_line(map_sq,-17,c))==QUEEN ||
        p==BISHOP) return p;
      if((p=scan_line(map_sq,17,c))==QUEEN ||
        p==BISHOP) return p;

    }

    // атаки коня и короля
   if(PI.piece_cnt[c][KNIGHT])
    for(j = 0; j < 8; j++)
     if( ((n = map_sq+knight_dir[j])&0x88)==0 )
     {
       u = UNMAP(n);
       if(P.pos[u]==KNIGHT)
        if(P.color[u]==c)
          return KNIGHT;
     }


    if(Distance(sq, PI.king_sq[c])==1)
     return KING;


    return 0;
}
```

Уфф! Этот не очень эстетичный код не всегда выполняется,
а только если на линиях есть фигуры противника.

Данная функция требуется, в основном, для определения шаха. Она должна работать максимально быстро и не отнимать время процессора. Наш вариант несмотря на простоту, не требует 64-битового типа данных и соответственно, такого же процессора.

Ужасные макросы MAP, UNMAP можно довольно элегантно реализовать на ассемблере.

```
#define MAP(sq)    ((sq&~7)<<1) | (sq&7))
#define UNMAP(sq)  ((sq&~7)>>1) | (sq&7))

0b00111111  // original
0b01110111  // map

// MAP(sq)
mov EAX,sq
shl EAX,5
shr AL,1     // shift right 8 bit
shr EAX,4
//  return EAX


// UNMAP(sq)
mov EAX,sq
shl EAX,4
shl AL,1
shr EAX,5
//  return EAX
```

Статическое размещение объектов дает возможность компилятору более эффективно вычислять адреса смещения. Это по сути - константы. Для скоростной программы это немаловажно.

Поиск.

Поиск в нашей программе подразделяется на 2 этапа: полный широтный поиск и поиск взятий. Второй этап служит для смягчения эффекта горизонта - чтобы по истечении некоторой глубины программа не прерывала поиск сразу, а возвращала некоторую приблизительную оценку.

```
/*
  поиск
  alpha-beta алгоритм
  число исследуемых позиций при наилучшем порядке
```

```
  перемещений -  sqrt(D)
  D - число позиций при полном переборе
*/
int alpha_beta(int depth, int alpha, int beta){

  int c, legal_moves;

  c = side;
  legal_moves = 0;

  if(depth <= 0 || ply > MAX_PLY-10)
   return Quies(alpha,beta); // поиск взятий

  generate();
  sort_moves();


  for(j = tree_cnt[ply]; j < tree_cnt[ply+1]; j++)
  {
    PickBestMove(j, tree_cnt[ply+1]-1);
    mv = tree[j];
    MakeMove(mv);
    if(Check(c)){UnMakeMove(mv); continue;};

    legal_moves++;


    ply++;
    score = -alpha_beta(depth+extensiens(mv,c)-1,
                        -beta,
                        -alpha);
    ply--;


    UnMakeMove(mv);

    if(TimeUp()) return 0;
    if(score > alpha)
    {
      alpha = score;

      save_best(mv);

      if(score >= beta) break;
    }
  }

  if(legal_moves==0)
    if(Check(c)) return INF - ply-1; //mate
    else return 0; // draw

  return alpha;
```

```
}


/*расширение глубины поиска на 1*/
int extensiens(Move mv, int c){

   if(Check(c^1)) return 1;  //шах противнику

   if(mv.piece==PAWN)  // пешка пошла на 7 гор.
    if(ROW(mv.to)==1 ||
       ROW(mv.to)==6) return 1;

   return 0;
}
```

В поиске взятий испольтзуется завышения alpha статической оценкой. В нормальных, не деградирующих позициях всегда можно найти ход, не ухудшающий текущего положения.
Рассматриваются только перемещения, дающие материальный прирост - это взятия и превращения пешки.
Прочие ходы вычеркнуты, так как их статический максимум ограничен оценкой после хода и возможный прирост оценки невелик по сравнению со взятиями.
Для большой глубины поиска - это лучше чем ничего.
Некоторые варианты статического поиска оставляют просмотр шахов. Если позиция под шахом, вызыватся основная функция с глубиной 1. Какой вариант выбрать - дело вкуса.

```
/*поиск взятий*/
int Quies(int alpha, int beta){



  if(ply > MAX_PLY-10)
   return Evaluate(alpha,beta); // оценка

  alpha = max(alpha,Evaluate(alpha,beta));
  if(alpha >= beta) return beta;


  generate_captures();
  sort_moves();


  for(j = tree_cnt[ply]; j < tree_cnt[ply+1]; j++)
  {
    PickBestMove(j, tree_cnt[ply+1]-1);
    mv = tree[j];
    MakeMove(mv);
```

```
    ply++;
    score = -Quies(-beta,-alpha);
    ply--;


    UnMakeMove(mv);

    if(TimeUp()) return 0;
    if(score > alpha)
    {
      alpha = score;

      save_best(mv);

      if(score >= beta) break;
    }
  }


  return alpha;
}
```

Поиск от корня.

В поиске от корня дерева перебора мы применим итеративные углубления. Поиск на глубину N-1 занимает гораздо меньше времени, чем на глубину N. Предварительный поиск также позволяет упорядочить порядок перемещений и таким образом, помогает сосчитать на большую глубину.

```
#define INF 30000          // недостижимый предел
#define MATE (INF-MAX_PLY)


Move root_search(){

  Move move = 0;
  int score=0;
  int a,b;

  reset_search_var();

  for(d = 2;
      d < MAX_PLY-10 && !TimeUp() && score < MATE;
      d++)
  {
```

```
    a = max(-INF, score - 30);
    b = min(INF,  score + 30);

    score = -search(d, a, b);

    if(!TimeUp())
    {
      if(score >= b)
        score = -search(d,a,INF);
      else if(score <= a)
        score = -search(d,-INF,b);
    }


    if(!TimeUp())
    {
      move = get_search_move();

    }
  }



  return move;

}//endp


reset_search_var(){

  Timer_Reset();

}
```

Сортировка перемещений.

Алгоритм поиска alpha-beta считает существенно быстрее
при хорошем порядке ходов. Поэтому на сортировку перемещений
нужно обратить особое внимание.
Введем 2 массива лучших ходов для каждого ply глубины
поиска и таблицу истории наиболее важных полей.

```
int killer1[MAX_PLY];
int killer2[MAX_PLY];
```

В фукции поиска найденный ход заносится в эти массивы.

```
int history[2][8][64];
int value[0,100,350,360,540,1100,30000];
#define MAX_HIST 1000


/*запоминает лучший ход*/
save_best(Move mv){
  int p,c,sq;

  if((int)mv != killer1[ply])
  {
     killer2[ply] = killer1[ply];
     killer1[ply] = (int)mv;

  }
  if(++history[mv.piece][mv.to] > MAX_HIST)
    for(c=WHITE; c < BLACK; c++)
      for(p=PAWN; p <= KING; p++)
        for(sq=0; sq<64; sq++)
          history[c][p][sq] /= 2;

}


int sort_val[MAX_PLY*100];

/*сортирует перемещения*/

sort_moves(){
  Move mv;
  int j, s;

     for(j = tree_cnt[ply];
         j < tree_cnt[ply+1];
         j++){

       mv = tree[j];
       s = 0;

       if((int)mv==killer1[ply])
         s = INF+2;
       else if((int)mv==killer2[ply])
         s = INF+1;
       else if(mv.kind & CAPTURE){

           s = MAX_HIST + value[mv.cap_piece] - mv.piece;

       }else if(mv.kind & PROMOTE){

           s = MAX_HIST + value[QUEEN];
       }else{
```

```
          s = history[side][mv.piece][mv.to];

          //добавим систему обхода доски
          if(side==BLACK)
            s += ( (63+(int)mv.to - mv.from)) >> 1) -
                 mv.piece;
          else
            s +=
             ( (63+(63-mv.to) - (63-mv.from)) >> 1) -
                 mv.piece;
       }
       sort_val[j] = s;
     }


}
```

Оценка позиции.

Наша несложная программа будет использовать упрощенную оценку позиции.

1. Статические таблицы для каждой фигуры. Инициализуются перед началом поиска.
2. Динамические факторы. Вычисляются для каждой позиции индивидуально.

Для шахматной игры характерны следующие основные позиционные ценности:

1. Приоритет центра.
2. Безопасность короля.
3. Проходные пешки (и структура пешек вообще)

Приоритет центра включает в себя собственно положение фигуры (близость к центру), и фигурную активность - это как правило мобильность.

Кроме позиционных факторов существует еще и материальная оценка фигур. Она представляет из себя средние показатели фигурной активности и удобна для поверхностного сравнения материала.

```
score = (material + pos_score) -
        (opponent_material + opponent_pos_score);

//material
```

```
int value[] = {0,100,350,360,540,1100,30000};



/* упрощенный вариант оценочной функции */
int Evaluate(int alpha, int beta){

 int score[2],s,c,sq;


   score[WHITE] = PI.mtl[WHITE];
   score[BLACK] = PI.mtl[BLACK];

   if(material_draw())
   {
     score[WHITE] /= 2;  // ----> 0
     score[BLACK] /= 2;
   }

   score[WHITE] += PI.static_score[WHITE];
   score[BLACK] += PI.static_score[BLACK];

   s = score[side] - score[xside];
   if(s + 500 < alpha) return alpha;
   if(s - 500 > beta)  return beta;


   for(c=WHITE; c <= BLACK; c++)
    for(sq=0; sq < 64; sq++)
     if( P.color[sq]==c )
     {
       s = 0;

       // tempo
       if(c==WHITE)
       {
         if(ROW(sq)<4)  s += 1;
       }else{
         if(ROW(sq)>=4) s += 1;
       }


       switch(P.piece[sq])
       {

         case PAWN:
         {
            int tbl[]=
            {
              0, 0, 0, 0, 0, 0, 0, 0,
             50,56,60,64,64,60,56,50,
              0, 0, 0, 0, 0, 0, 0, 0,
```

```
  0, 0, 7,11,11, 7, 0, 0,
  0, 0, 8,12,12, 8, 0, 0,
  1, 0, 1, 6, 6, 1, 0, 1,
  1, 2, 1, 2, 2, 1, 2, 1,
  0, 0, 0, 0, 0, 0, 0, 0
};
int passed[8]=
   { 0, 0, 30, 18, 10, 8, 6, 0};

if(c==WHITE)
{
  s += tbl[sq];
  if(passed_pawn(sq,c))
  {
    s += passed[ROW(sq)];
    // квадрат проходной пешки и король
    if(end_game())
     if(distance(sq, COLUMN(sq)) <
        distance(PI.king_sq[c^1], COLUMN(sq)))
        score += 70;
  }
  if(P.color[sq-8]==BLACK)
   s -= 1; //blocked

}else{
  s += tbl[63-sq];
  if(passed_pawn(sq,c))
  {
     s += passed[7-ROW(sq)];
    // квадрат проходной пешки и король
    if(end_game())
     if(distance(sq, 56+COLUMN(sq)) <
        distance(PI.king_sq[c^1], 56+COLUMN(sq)))
        score += 70;

  }
  if(P.color[sq+8]==WHITE)
    s -= 1;
}

if(PI.pawn_column_cnt[c][COLUMN(sq)] > 1)
  s -= 2; //double pawn
if(backward_pawn(sq,c))
{
  s -= 1;

  //кол-во атак на слабую пешку
  int x = COLUMN(sq);
  int y = ROW(sq);
  s -= PI.atk_x[c^1][x] +
       PI.atk_y[c^1][y] +
       PI.atk_x_plus_y[c^1][x+y] +
```

```
        PI.atk_x_sub_y[c^1][7+x-y];
        //+count_atk_knight(sq,c^1);


    if(isol_pawn(sq,c))
      s -= 4;
  }
}break; //pawn



case KNIGHT:
{
  int tbl[]={
   -8, 0, 0, 0,  0, 0, 0,-8,
    0, 2, 6, 0,  6, 6, 2, 0,
    0, 6, 8, 0,  7, 8, 6, 0,
    0, 6, 7,10, 10, 7, 6, 0,

    0, 6, 7,10, 10, 7, 6, 0,
    0, 6, 8, 7,  7, 8, 6, 0,
    0, 2, 6, 6,  6, 6, 2, 0,
   -8, 0, 0, 0,  0, 0, 0,-8
  };

  s += tbl[sq];
  if(distance(sq, PI.king_sq[c^1]) <= 3)
    score += 2


}break; //knight



case KING:
{
   int tbl[]={

    0, 0, 2, 4,    4,   2, 0, 0
    0, 6, 6, 6,    6,   6, 6, 0,
    2, 6,10,10,    10, 10, 6, 2,
    4, 6,10,16,    16, 10, 6, 4,

    4, 6,10,16,    16, 10, 6, 4,
    2, 6,10,10,    10, 10, 6, 2,
    2, 6,10,10,    6,   6, 6, 0,
    0, 0, 2, 4,    4,   2, 0, 0

   };

   if(PI.piec_cnt[WHITE][PAWN] +
      PI.piece_cnt[BLACK][PAWN]==0)
```

```
    {
     // end game
     s += tbl[sq];
     if(PI.mtl[c^1] > PI.mtl[c])
      s += tbl[sq]; //боится мата в углу
    }
    else
    {
      //король боится центра в
      //начале-середине игры и
      //стремится в центр ближе
      //к концу партии

      int *mtl = PI.piece_cnt[c^1];
      int emtl = mtl[QUEEN]*11 +
                 mtl[ROOK]*6   +
                 mtl{BISHOP]*4 +
                 mtl[KNIGHT]*3;
      double stage = (double)emtl /
                  11+12+8+6;


      s += -tbl[sq]*1.2*stage  +
           tbl[sq]*(1.0 - stage);

      //пешечный экран
      int t = 0;
      if(PI.right_castl[c]) t += 6;
      else if(PI.left_castl[c]) t += 4;
      else if(P.can_castl[c]==0)
        t -= 6;
      if(PI.pawn_column_cnt[c][COLUMN(x)]==0)
        t -= 6;      // open line
      if(!friend_pawn(c))
        t -= 10;
      s += t * stage;


      // вскрытый шах и связка
      int x = COLUMN(sq);
      int y = ROW(sq);
      s -= PI.atk_x[c^1][x] +
           PI.atk_y[c^1][y] +
           PI.atk_x_plus_y[c^1][x+y] +
           PI.atk_x_sub_y[c^1][7+x-y];

    }
}break; //king



case QUEEN:
```

```
        {
          int tbl[]={
            0,0,0,0,0,0,0,0,
            0,0,0,0,0,0,0,0,
            0,0,1,1,1,1,0,0,
            0,0,1,1,1,1,0,0,
            0,0,1,1,1,1,0,0,
            0,0,1,1,1,1,0,0,
            0,0,0,0,0,0,0,0,
            0,0,0,0,0,0,0,0
          };

          s += tbl[sq];
          s += 7-distance(sq,PI.king_sq[c^1]);
        }break;



        case ROOK:
        {
          int tbl[]={0,0,1,1,1,1,0,0};
//          int mob[14]=
//          {0,0,1,1,2,2, 3,3,3,4, 4,4,3,3};
//          s += mob[rook_mobile(sq,c)];


          s += tbl[COLUMN(sq)];

          // ладья на открытой линии
          if(PI.pawn_column_cnt[c][COLUMN(sq)]==0)
          {
            s += 2;
            if(PI.pawn_column_cnt[c^1][COLUMN(sq)]==0)
              s += 2;
          }


          //ладья на 7-ой горизонтали
          if(c==WHITE)
          {
             if(ROW(sq)==1)
               s += 4;
          }else{

             if(ROW(sq)==6)
               s += 4;
          }


          //2 ладьи на одной линии
          if( PI.atk_x[c][COLUMN(sq)] +
              PI.atk_y[c][ROW(sq)] > 2 )
```

```
            s += 1;

          if( PI.piece_cnt[c][ROOK]>1 )
            s += 1;

        }break;



        case BISHOP:
        {
          int tbl[]={
            1, 0, 0, 0,  0, 0, 0, 1,
            0, 4, 3, 2,  2, 3, 4, 0,
            0, 3, 6, 5,  5, 6, 3, 0,
            0, 2, 5, 6,  6, 5, 2, 0,

            0, 2, 5, 6,  6, 5, 2, 0,
            0, 3, 6, 5,  5, 6, 3, 0,
            0, 4, 3, 2,  2, 3, 4, 0,
            1, 0, 0, 0,  0, 0, 0, 1
          };
//          int mob[14]=
//          {-1,0,1,1,2,2, 3,3,3,4, 4,4,3,3};
//          s += mob[bishop_mobile(sq,c)];

          s += tbl[sq];
          if(PI.piece_cnt[c][BISHOP]>1)
            s += 1;
        }break;

        score[c] += s;
      } //switch piece
    }//endif




  s = score[side] - score[xside];
  if(s+50 < alpha) return alpha;
  if(s-50 > beta)  return beta;


  // 2 этап оценки, он редко выполняется
  for(c=WHITE; c <= BLACK; c++)
   for(sq=0; sq < 64; sq++)
    if( P.color[sq]==c )
     switch(P.piece[sq])
     {
       case PAWN:
       {
         if(backward_pawn(sq,c))
```

```
          score[c] -= count_atk_knight(sq,c^1);
        }break;

        case BISHOP:
        {
          int mob[14]=
          {-1,0,1,1,2,2, 3,3,3,4, 4,4,3,3};
          score[c] += mob[bishop_mobile(sq,c)];
        }break;

        case ROOK:
        {
          int mob[14]=
          {0,0,1,1,2,2, 3,3,3,4, 4,4,3,3};
          score[c] += mob[rook_mobile(sq,c)];
        }break;
       }//switch




    return score[side] - score[xside];

}//endp



//--------------------------------------


/*64-битовый тип данных, 1 бит - 1 клетка доски*/
typedef unsigned long long U64;


/*добавления в старый код*/
struct Pos_Info{
  ...
  //add
  U64 pawns[2];
}PI;



insert_piece(p,c,sq){
  ...
  //add

  if(p==PAWN)
    PI.pawns[c] |= (U64)1 << sq;
}
```

```
remove_piece(...){
  ...
  //add

  if(p==PAWN)
   PI.pawns[c] &= ~((U64)1 << sq);

}


/*битовая маска вверх от клетки*/
U64 up_line(int sq){
return
0xb\
00000001\
00000001\
00000001\
00000001\
00000001\
00000001\
00000001\
00000001    <<  (63-sq);
}

/*битовая маска вниз от клетки*/
U64 down_line(int sq){
return
0xb\
10000000\
10000000\
10000000\
10000000\
10000000\
10000000\
10000000\
10000000    >>  sq;
}

/*проходная пешка*/
int passed_pawn(int sq, int c){

  if(c==WHITE)
  {
    if((up_line(sq-8) & PI.pawns[c^1])==0)
     if(COLUMN(sq)==0 || (up_line(sq-8-1)&PI.pawns[c^1]==0))
      if(COLUMN(sq)==7 || (up_line(sq-8+1)&PI.pawns[c^1]==0))
        return 1;
  }else{
    if((down_line(sq+8) & PI.pawns[c^1])==0)
     if(COLUMN(sq)==0 || (down_line(sq+8-1)&PI.pawns[c^1]==0))
      if(COLUMN(sq)==7 || (down_line(sq+8+1)&PI.pawns[c^1]==0))
        return 1;
```

```
  }
  return 0;
}


int backward_pawn(int sq, int c){

  if(c==BLACK)
  {
     if(COLUMN(sq)==0 || (up_line(sq-1)&PI.pawns[c]==0))
      if(COLUMN(sq)==7 || (up_line(sq+1)&PI.pawns[c]==0))
        return 1;
  }else{
     if(COLUMN(sq)==0 || (down_line(sq-1)&PI.pawns[c]==0))
      if(COLUMN(sq)==7 || (down_line(sq+1)&PI.pawns[c]==0))
        return 1;
  }
  return 0;
}

/*изолированная пешка*/
int isol_pawn(int sq, int c){
  int x = COLUMN(sq);
  if(x==0 || PI.pawn_column_cnt[c][x-1]==0)
   if(x==7 || PI.pawn_column_cnt[c][x+1]==0)
    return 1;
  return 0;
}


/*король и пешки*/
int end_game(){
  #define Z PI.piece_cnt

  return Z[0][KNIGHT] + Z[0][BISHOP] +
         Z[0][ROOK]   + Z[0][QUEEN] == 0

           &&

         Z[1][KNIGHT] + Z[1][BISHOP] +
         Z[1][ROOK]   + Z[1][QUEEN] == 0;

}


/*только ничья в лучшем случае*/
int mtl_draw(int c){
  if(PI.piece_cnt[c][PAWN] +
     PI.piece_cnt[c][ROOK] == 0)
     if(PI.mtl[c] < 2 * value[BISHOP])
       return 1;
  return 0;
```

```
}

/* ничья - упрощенно */
int material_draw(){

  if(mtl_draw(WHITE))
   if(mtl_draw(BLACK))
    return 1;
  return 0;
}


/*количество атак коня на клетку*/
int count_atk_knight(int sq, int c){

  int i,n,u,cnt;

  cnt = 0;

  for(j = 0; j<8; j++)
   if(n = ((MAP(sq) + knight_dir[j])&0x88)==0)
   {
     u = UNMAP(n);
     if(P.piece[u]==KNIGHT)
      if(P.color[u]==c)
       cnt++;
   }
   return cnt;
}
```

Повторы позиций.

Ничейный результат бывает в 3-х случаях:

1. 50 'пустых' ходов (согл. правилам игры)
2. 3 раза повтор позиции (правила игры)
3. Позиция повторилась после корня дерева
   поиска. Это зацикливание и ведет к ничьей.

Для упрощенного сравнения позиций добавим 64-х
битовый хеш ключ и массив истории игры - массив
ключей.

```
U64 rnd_tbl[2][8][64];
U64 key_list[MAX_GAME+MAX_PLY];
#define SZ (1<<12)
int rep_cnt[SZ];
```

```
struct Move_Info{
  ...
  //add
  int empty_move_cnt;
}info_list[MAX_GAME+MAX_PLY];

struct Pos_Info{
  ...
  //add
  int empty_move_cnt;
  U64 key;
}PI;


insert_piece(p,c,sq){
  ...
  //add
  PI.key ^= rnd_tbl[c][p][sq];
}


remove_piece(...){
  ...
  //add
  PI.key ^= rnd_tbl[c][p][sq];
}


MakeMove(Move mv){
  ...

  //add - в конец функции
  rep_pos_push(mv);
}


UnMakeMove(Move mv){
  //add - в начало функции
  rep_pos_pop();

  ...
}



init_game_var(){

  int c,p,sq;

  for(c=WHITE; c <= BLACK; c++)
   for(p=PAWN; p <= KING; p++)
    for(sq=0; sq < 64; sq++)
```

```
    rnd_tbl[c][p][sq] = (U64)rand() ^
                        ((U64)rand()<<15) ^
                        ((U64)rand()<<31) ^
                        ((U64)rand()<<47) ^
                        ((U64)rand()<<55);
}



/* добавляет текущую позицию*/
rep_pos_push(Move mv){

   info_list[game_cnt+ply].empty_move_cnt =
            PI.empty_move_cnt;
   if(mv.kind != 0 || mv.piece<=PAWN)
     PI.empty_move_cnt = 0;
   else
     PI.empty_move_cnt++;


   key_list[game_cnt+ply] = PI.key;
   pos_cnt[PI.key & (SZ-1)]++; // key % SZ
}



rep_pos_pop(Move mv){

   PI.empty_move_cnt =
      info_list[game_cnt+ply].empty_move_cnt;
   pos_cnt[PI.key & (SZ-1)]--;
}


/*повторы позиций и 50 'пустых' ходов

  массив истории игры сканируется назад до
  первого 'необратимого хода' - взятие, рокировка,
  ход пешкой.
  если позиция встретилась второй раз в поиске или
  3 раза за всю игру+поиск, то функция возвращает 1
*/
int repetition(){

  int cnt,j;

  if(PI.empty_move_cnt/2 >= 50)
    return 1;


  cnt = 0;
```

```
  if( pos_cnt[PI.key & (SZ-1)] > 1 )
   for(j = game_cnt+ply;
       j >= 0 && j > game_cnt-25;
       j--)
   {
     Move mv = game_list[j];

     if(PI.key==key_list[j])
     {
      cnt++;
      if(cnt==3) return 1;
      if(cnt==2)
       if(j > game_cnt) return 1;
     }
     if(mv.kind != 0 || mv.piece<=PAWN)
      break;
   }
   return 0;
}


int search(...){

  //add
  if(ply>0)
  {
     ply--;
     int rep = repetition();
     ply++;
     if(rep) return DRAW; // 0
  }

  ...
}
```